Chapter 7: Testing and Debugging

Object Oriented Programming with Java

Test levels · Unit testing and JUnit · Assertions · Boundary values · Regression testing · Test automation · Debuggers · Profilers · Code coverage · Static analysis · Property-based testing · Memory analysis

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/java/slides/Chapter_07_Testing_and_Debugging.html

INTRODUCTION

Learning Outcomes

Test levels

Distinguish unit, module, sub-system, system and acceptance testing, and say what each one checks.

Unit testing with JUnit

Write automated tests with `@Test`, `@BeforeEach` and assertions such as `assertEquals` and `assertThrows`; read a failure message.

Good test design

Choose boundary values, write parameterized tests, keep tests deterministic and repeatable, and explain why regression tests matter.

Debugging

Set breakpoints, step through a program (Step Into / Over / Out) and watch variables to find the cause of a bug.

Analysis tools

Know which question a profiler, coverage tool, static analyzer (SpotBugs), property-based testing, Selenium, Memory Analyzer and VisualVM each answer.

Right tool, right question

Decide whether a problem calls for a test, a debugger or a profiler.

Testing and Debugging: Why?

"Testing is not about finding bugs; it is about making sure the system works as expected."
Michael Bolton (software testing expert)

"The best way to predict the future is to create it."
Often attributed to Abraham Lincoln; also to Peter Drucker and Alan Kay

"A bug is just an undocumented feature."
Anonymous programmer joke

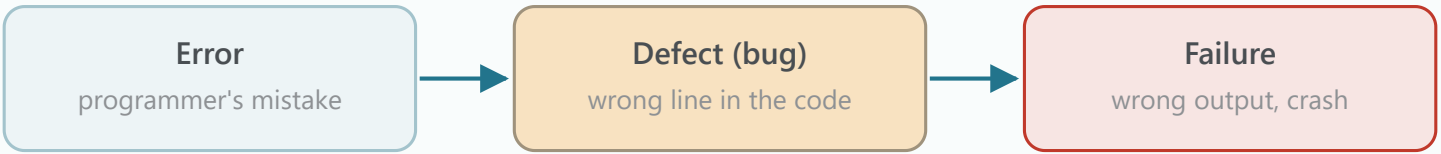
"Quality is not an act, it is a habit."
Attributed to Aristotle; actually Will Durant's paraphrase of him

"Not testing is like playing Russian roulette with your software."
Anonymous

"Testing shows the presence, not the absence of bugs."
Edsger W. Dijkstra

Testing checks systematically that a program shows the **expected behavior**. No test suite **proves** a program correct; it only raises confidence.

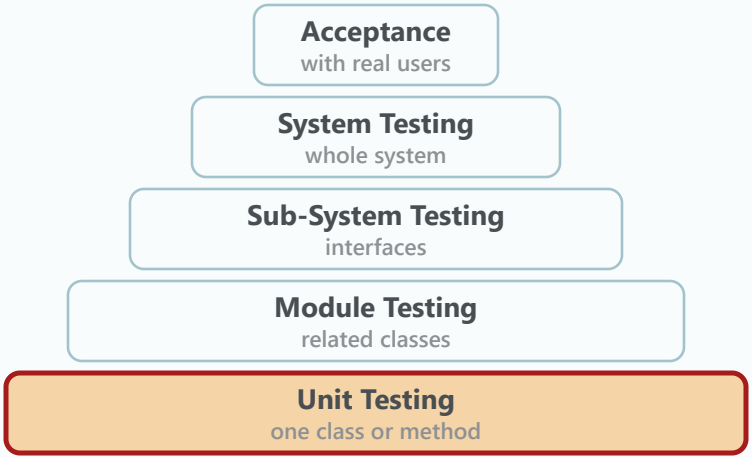
Error, Defect, Failure: Testing vs. Debugging



Testing reveals the failure → debugging finds the defect → the fix removes it

	Testing	Debugging
Goal	**Check** whether the program works as expected	**Find the cause** of a known failure and fix it
Question	"Does it work?"	"Why doesn't it work?"
Result	Pass / fail (✓ / ✗)	The faulty line and a fix
Tools	JUnit, Selenium, code coverage	Debugger, stack trace, `println`
When	Continuously, after every change	When a test fails or a user reports a bug

Levels of Testing in Software Development



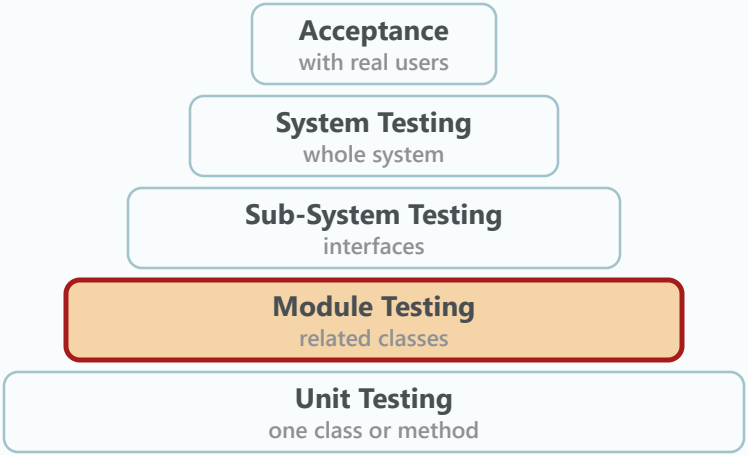
bottom to top: small part → whole system

Unit Testing

- Focuses on **individual, independent components** (a class, a method).
- Makes sure each component works correctly **on its own**.
- Example: does `BankAccount.withdraw()` throw an exception when funds are insufficient?

Unit tests: the smallest parts, the most tests, the fastest to run.

Levels of Testing in Software Development



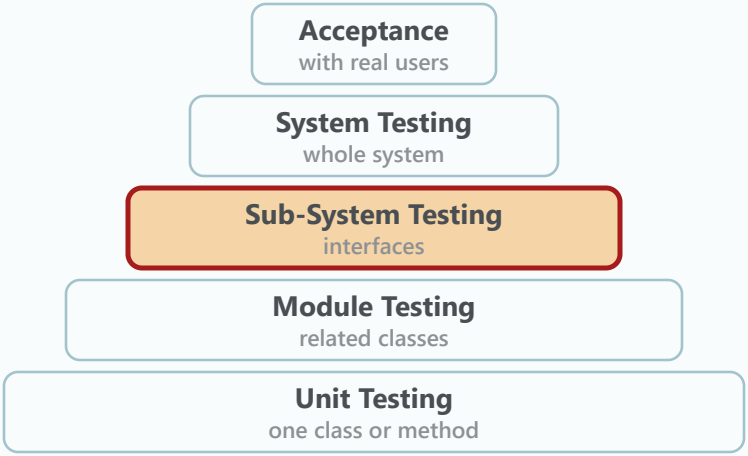
bottom to top: small part → whole system

Module Testing

- Tests a **module** made of related components.
- Checks the **integration** and behavior of the components inside it.
- Example: do the `Account`, `Transaction` and `Receipt` classes transfer money correctly together?

Module tests: related components together.

Levels of Testing in Software Development



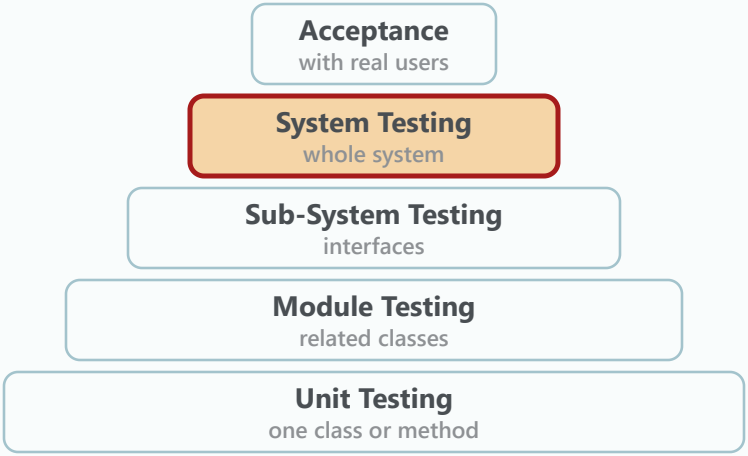
bottom to top: small part → whole system

Sub-System Testing

- Tests for **interface mismatches** between sub-systems.
- Checks **communication and data flow** between components.
- Example: do the account and database sub-systems use the same date format?

Sub-system tests: the interfaces where sub-systems meet.

Levels of Testing in Software Development



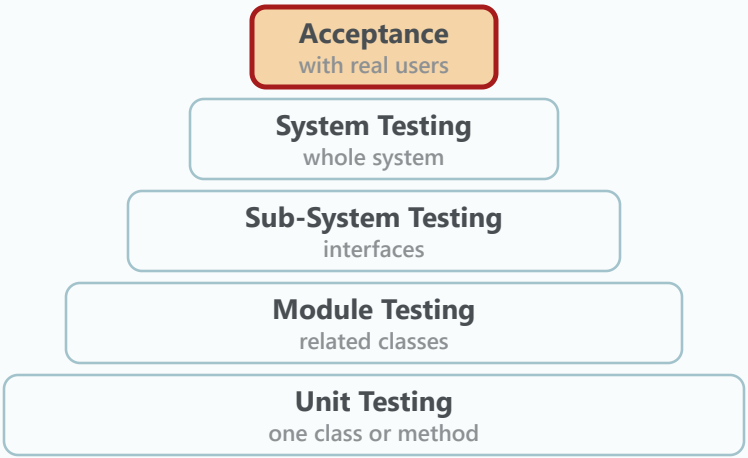
bottom to top: small part → whole system

System Testing

- Tests the **interactions** between sub-systems as a whole.
- Checks that **functional** (what it does) and **non-functional** (speed, security, usability) requirements are met.
- Example: with 1000 concurrent users, does a transfer finish in under 2 seconds?

System tests: the whole system, functional and non-functional requirements.

Levels of Testing in Software Development



bottom to top: small part → whole system

Acceptance Testing

- Tests the system with **real data** instead of simulated data.
- Confirms its real-world **usability and performance**.
- Done **together with users**: **alpha** testing at the developer's site, **beta** testing in the user's own environment.

Acceptance tests: can the customer say “I accept this system”?

Test Levels at a Glance

Level	What is tested?	Who tests?	Data
Unit	One class / method	Developer	Small, hand-picked
Module	A few classes working together	Developer	Scenario data
Sub-system	Interfaces between sub-systems	Developer, test team	Scenario data
System	Whole system; speed, security	Test team	Realistic
Acceptance (alpha / beta)	Customer requirements	With the users	Real data

Black-box testing

Tests only **input → expected output**, without looking inside the code. Acceptance and system tests are mostly like this.

White-box testing

Uses knowledge of the code to exercise every branch and loop. Unit tests and code coverage take this view.

A bug caught at a low level is cheap and quick to find; the same bug found in acceptance testing or by a user costs far more to fix.

First: Testing by Hand

CLASS UNDER TEST

```
public class Calculator {
  public int add(int a, int b) {
    return a + b;
  }

  public int multiply(int a, int b) {
    return a * b;
  }
}
```

MANUAL TEST PROGRAM

```
public class ManualTest {
  public static void main(String[] args) {
    Calculator calc = new Calculator();
    System.out.println("2 + 3 = " + calc.add(2, 3));
    System.out.println("2 * -3 = " + calc.multiply(2, -3));
    // Correct? We look at the screen and decide.
  }
}
```

Console

```
2 + 3 = 5
2 * -3 = -6
```

PROBLEMS WITH THIS APPROACH

- A **human eye** judges whether the result is right; it gets tired and makes mistakes.
- When the code changes, everything must be **retried by hand**.
- The test code sits in `main` and is later deleted: the effort is lost.
- Who will read 100 output lines for 100 methods?

The fix: **write the expected value into the code** and let the computer compare: **automated unit testing**.

JUnit

- **JUnit** is the most widely used **unit testing framework** for Java. Tests are ordinary Java methods; JUnit finds them, runs them and reports the results.
- In Java a **unit** usually means a **class**; its tests check the class's methods one by one.
- JUnit was designed for unit testing but is quite **flexible**; it is also used for:
  - **Module testing**: scenarios that involve several classes.
  - **System testing**: system-level interactions.
- IntelliJ IDEA, Eclipse, NetBeans, VS Code and Maven / Gradle run JUnit tests directly.

Version	How is a test written?
JUnit 3	Extend `TestCase`; method names start with `test`
JUnit 4	`@Test` annotation, `org.junit`
JUnit 5	`@Test`, `org.junit.jupiter.api`; lambdas, parameterized tests

The examples use **JUnit 5** (Jupiter). Adding it with Gradle:

```
testImplementation("org.junit.jupiter:junit-jupiter:5.11.4")
```

Test classes go in `src/test/java`; the classes under test go in `src/main/java`.

A First JUnit 5 Test

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

class CalculatorTest {

  @Test
  void addsTwoNumbers() {
    Calculator calc = new Calculator();
    assertEquals(5, calc.add(2, 3));
  }

  @Test
  void multipliesByNegative() {
    Calculator calc = new Calculator();
    assertEquals(-6, calc.multiply(2, -3));
  }
}
```

Test Results

✓ `addsTwoNumbers()`

✓ `multipliesByNegative()`

2 tests · 2 passed · 0 failed

`@Test`

Tells JUnit the method is a test. It is `void` with no parameters; no `main` is needed, and it need not be `public`.

`assertEquals(expected, actual)`

Expected value first, then the **actual** result. If they differ, the test **fails** (✗). `import static` saves writing the class name.

Test name = behavior

A name like `addsTwoNumbers` tells you what broke when the test fails.

Assertion Methods

Method	Passes when	Example (all pass)
`assertEquals(e, a)`	`e` and `a` are equal (`equals`)	`assertEquals(5, 2 + 3)`
`assertEquals(e, a, delta)`	Floating-point difference ≤ delta	`assertEquals(0.3, 0.1 + 0.2, 1e-9)`
`assertNotEquals(u, a)`	Not equal	`assertNotEquals(4, 2 + 3)`
`assertTrue` / `assertFalse`	Condition is true / false	`assertFalse("abc".isEmpty())`
`assertNull` / `assertNotNull`	Value is / is not `null`	`assertNotNull("x")`
`assertSame(a, b)`	Same object (`==`)	`assertSame(s, s)`
`assertArrayEquals`	Arrays equal element by element	`assertArrayEquals(new int[] {1, 2}, new int[] {1, 2})`
`assertThrows(T.class, code)`	The code throws an exception of that type	`assertThrows(ArithmeticException.class, () -> …)`
`assertAll(…)`	All assertions hold (all are tried)	`assertAll(() -> …, () -> …)`
`fail("message")`	Never; fails the test on purpose	`fail("not written yet")`

Without a delta, `assertEquals(0.3, 0.1 + 0.2)` fails: `expected: <0.3> but was: <0.30000000000000004>`. The last parameter is an optional message: `assertEquals(4, 2 + 3, "wrong sum")` → `wrong sum ==> expected: <4> but was: <5>`

The Class Under Test: `BankAccount`

```
public class BankAccount {
  private double balance;
  public BankAccount(double balance) {
    this.balance = balance;
  }
  public void deposit(double amount) {
    if (amount <= 0) {
      throw new IllegalArgumentException("Invalid amount");
    }
    balance += amount;
  }
  public void withdraw(double amount) {
    if (amount > balance) {
      throw new IllegalStateException("Insufficient funds");
    }
    balance -= amount;
  }
  public double getBalance() {
    return balance;
  }
}
```

WHICH BEHAVIORS SHOULD WE TEST?

Call (balance 100)	Expected
`deposit(50)`	balance 150
`withdraw(30)`	balance 70
`withdraw(500)`	`IllegalStateException`; balance stays 100
`deposit(-10)`	`IllegalArgumentException`
`deposit(0)` (boundary)	`IllegalArgumentException`
`withdraw(100)` (boundary)	balance 0, no exception

Each row becomes a separate test method. Cover **normal** cases, **invalid input** and **boundaries**.

Anatomy of a Test: Arrange, Act, Assert

```java
class AaaTest {

  @Test
  void withdrawReducesBalance() {
    // Arrange
    BankAccount acct = new BankAccount(100);
    // Act
    acct.withdraw(30);
    // Assert
    assertEquals(70.0, acct.getBalance(), 0.001);
  }
}
```

Test Results

✓ `withdrawReducesBalance()`

1 test · 1 passed

1 · Arrange

Set up the object and the data.

2 · Act

Call the method under test **once**.

3 · Assert

Compare the result with the expected one.

Each test checks **one behavior**. The balance is a `double`, so we pass `delta = 0.001`.

Shared Setup: `@BeforeEach`

`BankAccountTest`

```
class BankAccountTest {
  private BankAccount acct;

  @BeforeEach
  void setUp() {
    acct = new BankAccount(100);
  }

  @Test
  void depositIncreasesBalance() {
    acct.deposit(50);
    assertEquals(150.0, acct.getBalance(), 0.001);
  }

  @Test
  void overdraftThrows() {
    assertThrows(IllegalStateException.class,
      () -> acct.withdraw(500));
    assertEquals(100.0, acct.getBalance(), 0.001);
  }
}
```

- The `@BeforeEach` method runs **before every test**: each test starts with a **fresh** account (balance 100).
- JUnit creates **a new instance of the test class for every test method**; one test's changes to `acct` do not affect another.
- `assertThrows` runs the code in its second argument and **expects** an exception of that type. `() -> acct.withdraw(500)` is a **lambda expression**: "a piece of code to run".
- We also check that the balance is **unchanged** after the exception.

Test Results

✓ `overdraftThrows()`
✓ `depositIncreasesBalance()`

2 tests · 2 passed

How Does JUnit Run Tests?

```
▶ class BankAccountTest {
2   private BankAccount acct;
3
4   @BeforeEach
5   void setUp() {
6     acct = new BankAccount(100);
7   }
8
9   @Test
10  void depositIncreasesBalance() {
11    acct.deposit(50);
12    assertEquals(150.0, acct.getBalance(), 0.001);
13  }
14
15  @Test
16  void overdraftThrows() {
17    assertThrows(IllegalStateException.class,
18      () -> acct.withdraw(500));
19    assertEquals(100.0, acct.getBalance(), 0.001);
20  }
21 }
```

heap

test object #1

`acct` `null`

JUnit finds the `@Test` methods and creates **a new test object for the first test**. The field `acct` is still `null`.

Step 1 / 8

CALL STACK

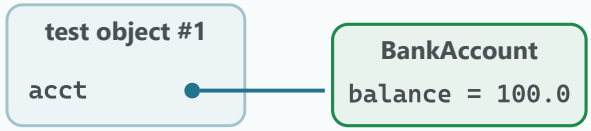
`JUnit`

OUTPUT

How Does JUnit Run Tests?

```
1  class BankAccountTest {
2    private BankAccount acct;
3
4    @BeforeEach
5    void setUp() {
▶      acct = new BankAccount(100);
7    }
8
9    @Test
10   void depositIncreasesBalance() {
11     acct.deposit(50);
12     assertEquals(150.0, acct.getBalance(), 0.001);
13   }
14
15   @Test
16   void overdraftThrows() {
17     assertThrows(IllegalStateException.class,
18       () -> acct.withdraw(500));
19     assertEquals(100.0, acct.getBalance(), 0.001);
20   }
21 }
```

heap



`@BeforeEach` runs before the test: a new account, balance 100.

Step 2 / 8

CALL STACK

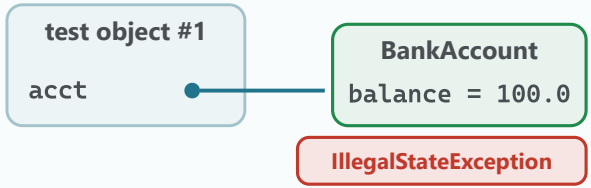
- `setUp()`
- JUnit

OUTPUT

How Does JUnit Run Tests?

```
1  class BankAccountTest {
2    private BankAccount acct;
3
4    @BeforeEach
5    void setUp() {
6      acct = new BankAccount(100);
7    }
8
9    @Test
10   void depositIncreasesBalance() {
11     acct.deposit(50);
12     assertEquals(150.0, acct.getBalance(), 0.001);
13   }
14
15   @Test
16   void overdraftThrows() {
17     assertThrows(IllegalStateException.class,
▶        () -> acct.withdraw(500));
19     assertEquals(100.0, acct.getBalance(), 0.001);
20   }
21 }
```

heap



JUnit picks the order: `overdraftThrows` runs first. `assertThrows` calls the lambda: `withdraw(500)` throws.

Step 3 / 8

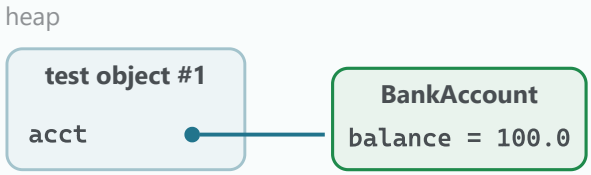
CALL STACK

- **assertThrows(…)**
- overdraftThrows()
- JUnit

OUTPUT

How Does JUnit Run Tests?

```
1  class BankAccountTest {
2    private BankAccount acct;
3
4    @BeforeEach
5    void setUp() {
6      acct = new BankAccount(100);
7    }
8
9    @Test
10   void depositIncreasesBalance() {
11     acct.deposit(50);
12     assertEquals(150.0, acct.getBalance(), 0.001);
13   }
14
15   @Test
16   void overdraftThrows() {
17     assertThrows(IllegalStateException.class,
18       () -> acct.withdraw(500));
▶      assertEquals(100.0, acct.getBalance(), 0.001);
20   }
21 }
```



✓ test 1 passed

The exception has the expected type: `assertThrows` passes. The balance is still 100, so the second assertion passes too.

Step 4 / 8

CALL STACK

`overdraftThrows()`

JUnit

OUTPUT

```
✓ overdraftThrows()
```

How Does JUnit Run Tests?

```
▶ class BankAccountTest {
  private BankAccount acct;

  @BeforeEach
  void setUp() {
    acct = new BankAccount(100);
  }

  @Test
  void depositIncreasesBalance() {
    acct.deposit(50);
    assertEquals(150.0, acct.getBalance(), 0.001);
  }

  @Test
  void overdraftThrows() {
    assertThrows(IllegalStateException.class,
      () -> acct.withdraw(500));
    assertEquals(100.0, acct.getBalance(), 0.001);
  }
}
```

heap

test object #2

`acct` `null`

A **brand-new** test object (#2) for the second test. The old object and its account are no longer used.

Step 5 / 8

CALL STACK

`JUnit`

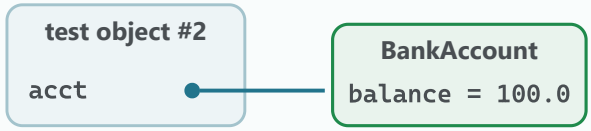
OUTPUT

```
✓ overdraftThrows()
```

How Does JUnit Run Tests?

```
1  class BankAccountTest {
2    private BankAccount acct;
3
4    @BeforeEach
5    void setUp() {
▶      acct = new BankAccount(100);
7    }
8
9    @Test
10   void depositIncreasesBalance() {
11     acct.deposit(50);
12     assertEquals(150.0, acct.getBalance(), 0.001);
13   }
14
15   @Test
16   void overdraftThrows() {
17     assertThrows(IllegalStateException.class,
18       () -> acct.withdraw(500));
19     assertEquals(100.0, acct.getBalance(), 0.001);
20   }
21 }
```

heap



`@BeforeEach` runs again: this test also starts at balance 100.

Step 6 / 8

CALL STACK

- `setUp()`
- JUnit

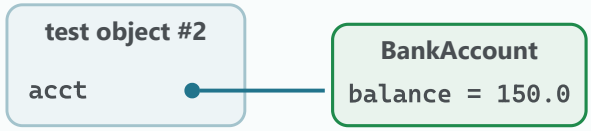
OUTPUT

```
✓ overdraftThrows()
```

How Does JUnit Run Tests?

```
1  class BankAccountTest {
2    private BankAccount acct;
3
4    @BeforeEach
5    void setUp() {
6      acct = new BankAccount(100);
7    }
8
9    @Test
10   void depositIncreasesBalance() {
▶      acct.deposit(50);
12     assertEquals(150.0, acct.getBalance(), 0.001);
13   }
14
15   @Test
16   void overdraftThrows() {
17     assertThrows(IllegalStateException.class,
18       () -> acct.withdraw(500));
19     assertEquals(100.0, acct.getBalance(), 0.001);
20   }
21 }
```

heap



Act: `deposit(50)` ; the balance becomes 150.

Step 7 / 8

CALL STACK

- **depositIncreasesBalance()**
- JUnit

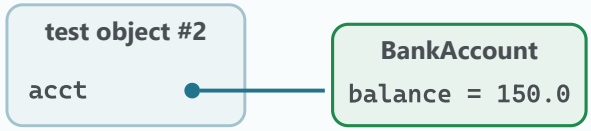
OUTPUT

```
✓ overdraftThrows()
```

How Does JUnit Run Tests?

```
1  class BankAccountTest {
2    private BankAccount acct;
3
4    @BeforeEach
5    void setUp() {
6      acct = new BankAccount(100);
7    }
8
9    @Test
10   void depositIncreasesBalance() {
11     acct.deposit(50);
▶      assertEquals(150.0, acct.getBalance(), 0.001);
13   }
14
15   @Test
16   void overdraftThrows() {
17     assertThrows(IllegalStateException.class,
18       () -> acct.withdraw(500));
19     assertEquals(100.0, acct.getBalance(), 0.001);
20   }
21 }
```

heap



✓ test 2 passed

Assert: 150.0 = 150.0 (difference ≤ 0.001). Both tests passed, and they ran **independently**.

Step 8 / 8

CALL STACK

- `depositIncreasesBalance()`
- `JUnit`

OUTPUT

```
✓ overdraftThrows()
✓ depositIncreasesBalance()
```

Testing Exceptions: `assertThrows`

```java
class BankExceptionTest {

  @Test
  void overdraftMessage() {
    BankAccount acct = new BankAccount(100);
    IllegalStateException e = assertThrows(
      IllegalStateException.class, () -> acct.withdraw(500));
    assertEquals("Insufficient funds", e.getMessage());
  }

  @Test
  void negativeDepositRejected() {
    BankAccount acct = new BankAccount(100);
    assertThrows(IllegalArgumentException.class,
      () -> acct.deposit(-10));
  }

  @Test
  void wrongExpectation() {
    BankAccount acct = new BankAccount(100);
    assertThrows(IllegalStateException.class,
      () -> acct.withdraw(50));
  }
}
```

- `assertThrows` **returns** the thrown exception, so its message can be checked too.
- Testing that bad input is **rejected** matters as much as testing good input.
- The last test is wrong on purpose: withdrawing 50 TL is valid, so nothing is thrown.

Test Results

✓ `overdraftMessage()`
✓ `negativeDepositRejected()`
✗ `wrongExpectation()`

```
Expected
java.lang.IllegalStateException to
be thrown, but nothing was thrown.
```

3 tests · 2 passed · 1 failed

In JUnit 4: `@Test(expected = …class)`.

Test Lifecycle Annotations

```
class LifecycleTest {

  @BeforeAll
  static void start() { System.out.println("BeforeAll"); }

  @BeforeEach
  void setUp() { System.out.println("  BeforeEach"); }

  @Test
  void testA() { System.out.println("    testA"); }

  @Test
  void testB() { System.out.println("    testB"); }

  @AfterEach
  void cleanUp() { System.out.println("  AfterEach"); }

  @AfterAll
  static void finish() { System.out.println("AfterAll"); }
}
```

Output

```
BeforeAll
  BeforeEach
    testA
  AfterEach
  BeforeEach
    testB
  AfterEach
AfterAll
```

Annotation	When does it run?
`@BeforeAll`	**Once** before all tests (`static`)
`@BeforeEach`	Before each test
`@AfterEach`	After each test (even if it fails)
`@AfterAll`	Once after all tests (`static`)

The **order of the tests** is deterministic but need not match the source order: never write tests that depend on each other.

A Failing Test: From Red to Green

`ArrayUtils` (FIRST VERSION)

```
public class ArrayUtils {
  public static int max(int[] arr) {
    int best = 0;
    for (int i = 0; i < arr.length; i++) {
      if (arr[i] > best) {
        best = arr[i];
      }
    }
    return best;
  }
}
```

`ArrayUtilsTest`

```
class ArrayUtilsTest {

  @Test
  void positiveArray() {
    int[] arr = {3, 9, 4};
    assertEquals(9, ArrayUtils.max(arr));
  }

  @Test
  void allNegativeArray() {
    int[] arr = {-5, -2, -8};
    assertEquals(-2, ArrayUtils.max(arr));
  }
}
```

The code compiles and looks right at first glance. Let's write two tests: one positive array, one all-negative array.

A Failing Test: From Red to Green

`ArrayUtils` (FIRST VERSION)

```
public class ArrayUtils {
  public static int max(int[] arr) {
    int best = 0;
    for (int i = 0; i < arr.length; i++) {
      if (arr[i] > best) {
        best = arr[i];
      }
    }
    return best;
  }
}
```

Test Results

✓ `positiveArray()`

✗ `allNegativeArray()`

```
org.opentest4j.AssertionFailedError: expected:
<-2> but was: <0>
```

2 tests · 1 passed · **1 failed**

expected = what we wanted (-2), **but was** = what came back (0). 0 is not even in the array!

Red: the initial value `best = 0` is never exceeded when all elements are negative.

A Failing Test: From Red to Green

`ArrayUtils` **(FIXED)**

```
public class ArrayUtils {
  public static int max(int[] arr) {
    int best = arr[0];                // first element
    for (int i = 1; i < arr.length; i++) {
      if (arr[i] > best) {
        best = arr[i];
      }
    }
    return best;
  }
}
```

Test Results

✓ `positiveArray()`
✓ `allNegativeArray()`

2 tests · 2 passed

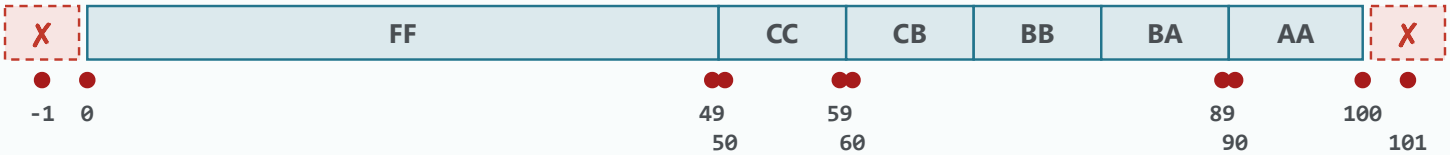
The maximum now starts from the **first element**. The test stays for good: if the bug returns, it is caught at once.

Green: after the fix the same tests were run again, and both passed.

Which Values to Test? Boundary Values

```java
public class Grade {
  public static String letterGrade(int score) {
    if (score < 0 || score > 100) {
      throw new IllegalArgumentException("Invalid score: " + score);
    }
    if (score >= 90) return "AA";
    if (score >= 80) return "BA";
    if (score >= 70) return "BB";
    if (score >= 60) return "CB";
    if (score >= 50) return "CC";
    return "FF";
  }
}
```

- Bugs cluster at **boundaries**: `>` instead of `>=`, `<` `100` instead of `<= 100`…
- **Boundary value analysis**: pick both ends of each range (49 / 50, 89 / 90) and the first invalid values (−1, 101).
- **Equivalence classes**: 61, 65 and 68 lie in the same range and give the same result; one of them is enough.



● tested value: both ends of each range and the first invalid value

Parameterized Tests JUnit 5

```java
class GradeTest {

  @ParameterizedTest
  @CsvSource({"0, FF", "49, FF", "50, CC", "59, CC",
        "60, CB", "89, BA", "90, AA", "100, AA"})
  void boundaries(int score, String expected) {
    assertEquals(expected, Grade.letterGrade(score));
  }

  @Test
  void outOfRangeRejected() {
    assertThrows(IllegalArgumentException.class,
      () -> Grade.letterGrade(-1));
    assertThrows(IllegalArgumentException.class,
      () -> Grade.letterGrade(101));
  }
}
```

- `@ParameterizedTest` : the same test method runs again and again **with different data**.
- `@CsvSource` : each string is one row; its comma-separated values are converted to the parameters (`score` , `expected`) in order.
- Dependency: `junit-jupiter-params` (included in `junit-jupiter`).

boundaries(int, String)

- ✓ `[1] 0, FF`
- ✓ `[2] 49, FF`
- ✓ `[3] 50, CC`
- ✓ `[4] 59, CC`
- ✓ `[5] 60, CB`
- ✓ `[6] 89, BA`
- ✓ `[7] 90, AA`
- ✓ `[8] 100, AA`
- ✓ `outOfRangeRejected()`

9 tests · 9 passed

A Parameterized Test, Step by Step

```
1 class GradeTest {
▶   @ParameterizedTest
▶   @CsvSource({"0, FF", "49, FF", "50, CC", "59,
  CC",
▶         "60, CB", "89, BA", "90, AA", "100, AA"})
5   void boundaries(int score, String expected) {
6     assertEquals(expected,
  Grade.letterGrade(score));
7   }
8 }
```



both ends of each range are tested

JUnit reads the 8 strings in `@CsvSource`; each becomes a separate test call.

Step 1 / 9

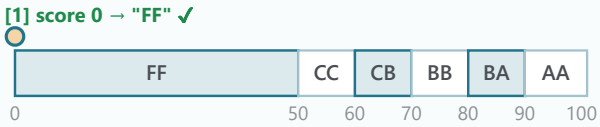
VARIABLES

`rows = 8`

OUTPUT

A Parameterized Test, Step by Step

```
1 class GradeTest {
2   @ParameterizedTest
3   @CsvSource({"0, FF", "49, FF", "50, CC", "59,
  CC",
4           "60, CB", "89, BA", "90, AA", "100, AA"})
5   void boundaries(int score, String expected) {
▶       assertEquals(expected,
  Grade.letterGrade(score));
7   }
8 }
```



both ends of each range are tested

Row 1: `"0, FF"` is split at the comma; `"0"` is converted to an `int` for `score`, and `"FF"` goes to `expected`.

Step 2 / 9

VARIABLES

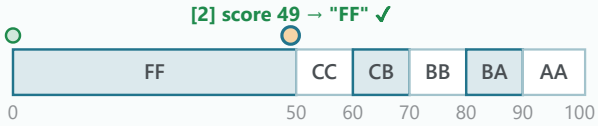
`score = 0` `expected = "FF"` `actual = "FF"`

OUTPUT

```
✓ [1] 0, FF
```

A Parameterized Test, Step by Step

```
1 class GradeTest {
2   @ParameterizedTest
3   @CsvSource({"0, FF", "49, FF", "50, CC", "59, CC",
4         "60, CB", "89, BA", "90, AA", "100, AA"})
5   void boundaries(int score, String expected) {
▶     assertEquals(expected, Grade.letterGrade(score));
7   }
8 }
```



both ends of each range are tested

49: the **upper end** of FF. Had we written `> 48` instead of `>= 50`, it would fail here.

Step 3 / 9

VARIABLES

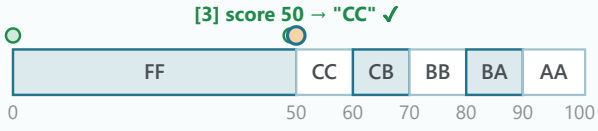
`score = 49` `expected = "FF"` `actual = "FF"`

OUTPUT

```
✓ [1] 0, FF
✓ [2] 49, FF
```

A Parameterized Test, Step by Step

```
1 class GradeTest {
2   @ParameterizedTest
3   @CsvSource({"0, FF", "49, FF", "50, CC", "59, CC",
4         "60, CB", "89, BA", "90, AA", "100, AA"})
5   void boundaries(int score, String expected) {
▶     assertEquals(expected, Grade.letterGrade(score));
7   }
8 }
```



both ends of each range are tested

50: the **lower end** of CC; with 49 it tests the boundary from both sides.

Step 4 / 9

VARIABLES

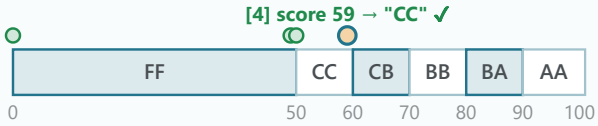
`score = 50` `expected = "CC"` `actual = "CC"`

OUTPUT

```
✓ [1] 0, FF
✓ [2] 49, FF
✓ [3] 50, CC
```

A Parameterized Test, Step by Step

```
1 class GradeTest {
2   @ParameterizedTest
3   @CsvSource({"0, FF", "49, FF", "50, CC", "59,
  CC",
4           "60, CB", "89, BA", "90, AA", "100, AA"})
5   void boundaries(int score, String expected) {
▶       assertEquals(expected,
  Grade.letterGrade(score));
7   }
8 }
```



both ends of each range are tested

59: the upper end of CC.

Step 5 / 9

VARIABLES

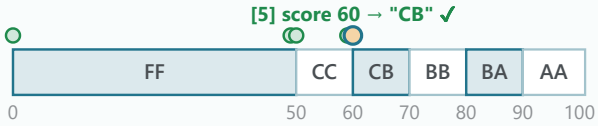
`score = 59` `expected = "CC"` `actual = "CC"`

OUTPUT

```
✓ [1] 0, FF
✓ [2] 49, FF
✓ [3] 50, CC
✓ [4] 59, CC
```

A Parameterized Test, Step by Step

```
1 class GradeTest {
2   @ParameterizedTest
3   @CsvSource({"0, FF", "49, FF", "50, CC", "59,
  CC",
4           "60, CB", "89, BA", "90, AA", "100, AA"})
5   void boundaries(int score, String expected) {
▶       assertEquals(expected,
  Grade.letterGrade(score));
7   }
8 }
```



both ends of each range are tested

60: the lower end of CB.

Step 6 / 9

VARIABLES

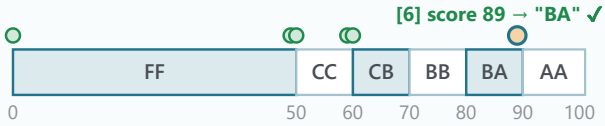
`score = 60` `expected = "CB"` `actual = "CB"`

OUTPUT

```
✓ [1] 0, FF
✓ [2] 49, FF
✓ [3] 50, CC
✓ [4] 59, CC
✓ [5] 60, CB
```

A Parameterized Test, Step by Step

```
1 class GradeTest {
2   @ParameterizedTest
3   @CsvSource({"0, FF", "49, FF", "50, CC", "59,
  CC",
4           "60, CB", "89, BA", "90, AA", "100, AA"})
5   void boundaries(int score, String expected) {
▶       assertEquals(expected,
  Grade.letterGrade(score));
7   }
8 }
```



both ends of each range are tested

89: the upper end of BA.

Step 7 / 9

VARIABLES

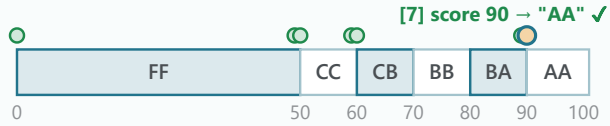
`score = 89` `expected = "BA"` `actual = "BA"`

OUTPUT

```
✓ [1] 0, FF
✓ [2] 49, FF
✓ [3] 50, CC
✓ [4] 59, CC
✓ [5] 60, CB
✓ [6] 89, BA
```

A Parameterized Test, Step by Step

```
1 class GradeTest {
2   @ParameterizedTest
3   @CsvSource({"0, FF", "49, FF", "50, CC", "59,
  CC",
4         "60, CB", "89, BA", "90, AA", "100, AA"})
5   void boundaries(int score, String expected) {
▶     assertEquals(expected,
  Grade.letterGrade(score));
7   }
8 }
```



both ends of each range are tested

90: the lower end of AA.

Step 8 / 9

VARIABLES

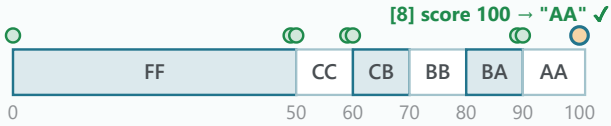
`score = 90` `expected = "AA"` `actual = "AA"`

OUTPUT

```
✓ [1] 0, FF
✓ [2] 49, FF
✓ [3] 50, CC
✓ [4] 59, CC
✓ [5] 60, CB
✓ [6] 89, BA
✓ [7] 90, AA
```

A Parameterized Test, Step by Step

```
1 class GradeTest {
2   @ParameterizedTest
3   @CsvSource({"0, FF", "49, FF", "50, CC", "59,
  CC",
4           "60, CB", "89, BA", "90, AA", "100, AA"})
5   void boundaries(int score, String expected) {
▶       assertEquals(expected,
  Grade.letterGrade(score));
7   }
8 }
```



both ends of each range are tested

100: the last valid value; if the `score > 100` check is right, it returns AA.

Step 9 / 9

VARIABLES

`score = 100` `expected = "AA"` `actual = "AA"`

OUTPUT

```
✓ [1] 0, FF
✓ [2] 49, FF
✓ [3] 50, CC
✓ [4] 59, CC
✓ [5] 60, CB
✓ [6] 89, BA
✓ [7] 90, AA
✓ [8] 100, AA
```

What Makes a Good Unit Test: FIRST

F · Fast
Finishes in milliseconds, so thousands can run after every change. No files, network or database.

I · Independent
Does not depend on another test, on test order, or on state left by an earlier test.

R · Repeatable
Gives the same result on every machine, every time: randomness and the clock are under control.

S · Self-validating
The result is ✓ or ✗; no human needs to read the output.

T · Timely
Written with the code (or even before it), not months later.

And readable
A descriptive name, the AAA pattern, one behavior per test. A test documents how the code is meant to be used.

Unit Test or Integration Test?

	Unit test	Integration test
Scope	One class / method, **isolated**	Several components working together
Speed	Milliseconds	Seconds (database, files, network)
External dependency	Replaced by a simple fake (a test double)	Real database, file, service
Bugs it finds	Local logic errors; location obvious	Mismatched interfaces, bad configuration

TEST NAMES THAT DESCRIBE BEHAVIOR

```
@Test void test1() { … }
@Test void discountForSixItems() { … }
@Test void noDiscountForFive() { … }
```

- Test **observable behavior**, not implementation details.
- Unit tests avoid `Thread.sleep`, real networks and files: they make tests slow and flaky.
- Tests share no mutable state (e.g. a `static` list).

Regression Testing

- Checks that **everything that worked before still works** after a change. A regression is a step backward: something that worked breaks again.
- Method: rerun **all existing tests** after every change.



Tests must be deterministic

Under the same conditions a test must give **the same result every time**. A test that sometimes passes and sometimes fails (“flaky”) destroys trust.

Tests must be repeatable

Design tests so they can be **repeated** at different times, on different machines and in different states.

Testing Random Code Deterministically

```java
import java.util.Random;

public class Die {
  private final Random random;

  public Die(Random random) {        // injected
    this.random = random;
  }

  public int roll() {
    return random.nextInt(6) + 1;
  }
}
```

FIVE ROLLS WITH THE SAME SEED (42)

Console

```
3 4 1 3 1
```

```java
class DieTest {

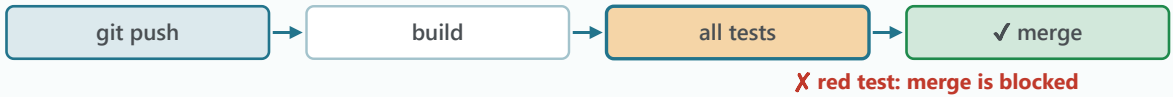
  @Test
  void sameSeedSameRolls() {
    Die d1 = new Die(new Random(42));
    Die d2 = new Die(new Random(42));
    for (int i = 0; i < 100; i++) {
      assertEquals(d1.roll(), d2.roll());
    }
  }
}
```

The `Random` object is **passed in** (dependency injection). With a fixed **seed** the test gets the same numbers on every run; had the class called `new Random()` itself, the results would change each time.

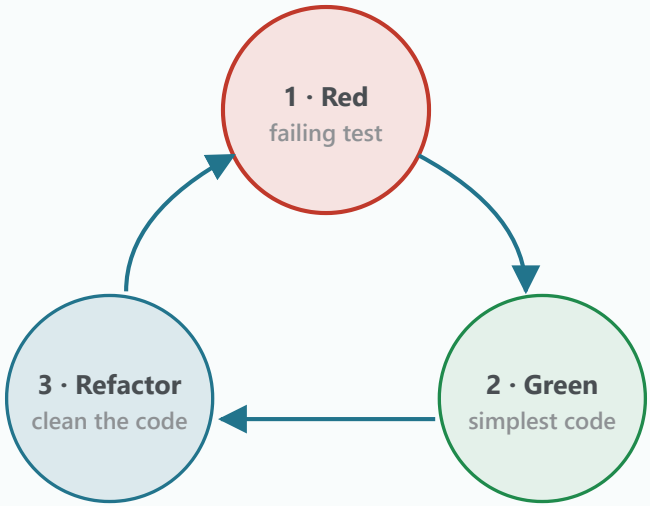
Automated Tests

- Write **automated tests** instead of testing code **by hand**.
- The effort is about the **same**, but you end up with an automated test.
- That test can be rerun **whenever you like** (every build, every change).
- It is **wrong** to think that software that worked once needs no more testing: code changes, libraries change, environments change.
- **Continuous integration** (CI, e.g. GitHub Actions, Jenkins) runs all tests on a server after every `git push`; a red test blocks the merge.

	Manual	Automated
Initial cost	Low	Somewhat higher
Cost to repeat	**Same every time**	**≈ 0**
Speed	Minutes	Seconds
Reliability	Human error	Same every time
Regression	Rarely done	On every change
As documentation	None	Usage examples



Test-Driven Development (TDD)



- **Test-Driven Development**: write the test **before the code**.
- **Red**: write a test for behavior that does not exist yet; it fails (or does not compile).
- **Green**: write the **simplest** code that makes it pass.
- **Refactor**: clean up the code; as long as the tests stay green, the behavior is preserved.
- Example: first write `assertEquals("CC", Grade.letterGrade(50))`, then the `letterGrade` method. Each new grade range is one more cycle.

The payoff: every line of code is born with a test, and regression tests pile up on their own.

The Debugger

A debugger lets you inspect **the state of a running program**. With a debugger you can:

- Run the program **one statement** (one line) at a time
- See the **source code** with the current line marked
- View **variable values** and change them if needed
- Set **breakpoints** in the program
- Run the program **up to** a given breakpoint
- Inspect a crashed or stopped program through a **dump file** (a *core file* in C; in Java a `.hprof` heap dump and a thread dump)

IDE debuggers
IntelliJ IDEA, Eclipse, NetBeans, VS Code: click next to a line number to set a breakpoint, then start with **Debug**.

println debugging
Quick, but it clutters the code, needs recompiling and is easy to forget to remove. A debugger leaves the code alone.



DEBUGGING

Breakpoints and Stepping Commands

Command	What it does	IntelliJ	Eclipse
Breakpoint	When execution reaches this line, the program stops **before running it**	Ctrl+F8	Ctrl+Shift+B
Step Over	Runs the line; any method call completes **without stepping into it**	F8	F6
Step Into	Steps **into** the method called on the line	F7	F5
Step Out	Finishes the current method and returns to the **caller**	Shift+F8	F7
Resume	Runs until the next breakpoint	F9	F8
Evaluate / Watch	Evaluates an expression (e.g. `arr[i] > best`) now, or keeps watching it	Alt+F8	Ctrl+Shift+I

Conditional breakpoint
Stops only when a condition holds, e.g. `i == 999`: catches the exact moment in a loop of thousands of iterations.

Exception breakpoint
Stops the moment a given exception (e.g. `NullPointerException`) is thrown.

Shortcuts are for the default keymap; the same commands are in the Run / Debug menu.

Finding the Defect with a Debugger

```
1  public class FindMax {
2    public static int max(int[] arr) {
3      int best = 0;
4      for (int i = 0; i < arr.length; i++) {
5        if (arr[i] > best) {
6          best = arr[i];
7        }
8      }
9      return best;
10   }
11
12   public static void main(String[] args) {
▶      int[] temps = {-5, -2, -8};
14     int result = max(temps);
15     System.out.println("Max: " + result);
16   }
17 }
```

arr

[0]	[1]	[2]
−5	−2	−8

A breakpoint is set on line 5 (`if`). The program was started with **Debug**.

Step 1 / 8

VARIABLES

—

CALL STACK

`main()`

OUTPUT

Finding the Defect with a Debugger

```
1  public class FindMax {
2    public static int max(int[] arr) {
3      int best = 0;
4      for (int i = 0; i < arr.length; i++) {
5        if (arr[i] > best) {
6          best = arr[i];
7        }
8      }
9      return best;
10   }
11
12   public static void main(String[] args) {
13     int[] temps = {-5, -2, -8};
▶      int result = max(temps);
15     System.out.println("Max: " + result);
16   }
17 }
```

arr

[0]	[1]	[2]
−5	−2	−8

Step Into: we step into the call to `max`.

Step 2 / 8

VARIABLES

`temps = [-5, -2, -8]`

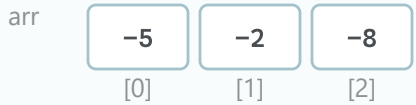
CALL STACK

`main()`

OUTPUT

Finding the Defect with a Debugger

```
1  public class FindMax {
2    public static int max(int[] arr) {
▶      int best = 0;
4      for (int i = 0; i < arr.length; i++) {
5        if (arr[i] > best) {
6          best = arr[i];
7        }
8      }
9      return best;
10   }
11
12   public static void main(String[] args) {
13     int[] temps = {-5, -2, -8};
14     int result = max(temps);
15     System.out.println("Max: " + result);
16   }
17 }
```



A new frame: `arr` refers to the same array. `best` is not assigned yet.

Step 3 / 8

VARIABLES

`arr = [-5, -2, -8]`

CALL STACK

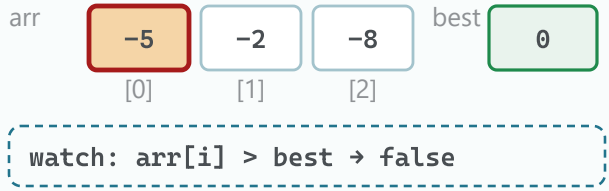
`max(arr)`

`main()`

OUTPUT

Finding the Defect with a Debugger

```
1  public class FindMax {
2    public static int max(int[] arr) {
3      int best = 0;
4      for (int i = 0; i < arr.length; i++) {
▶        if (arr[i] > best) {
6          best = arr[i];
7        }
8      }
9      return best;
10   }
11
12   public static void main(String[] args) {
13     int[] temps = {-5, -2, -8};
14     int result = max(temps);
15     System.out.println("Max: " + result);
16   }
17 }
```



At the breakpoint: `best = 0`, `arr[0] = -5`.
The watched expression is **false**.

Step 4 / 8

VARIABLES

`arr = [-5, -2, -8]` `best = 0` `i = 0`

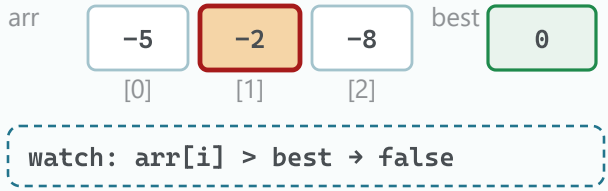
CALL STACK

- `max(arr)`
- `main()`

OUTPUT

Finding the Defect with a Debugger

```
1  public class FindMax {
2    public static int max(int[] arr) {
3      int best = 0;
4      for (int i = 0; i < arr.length; i++) {
▶        if (arr[i] > best) {
6          best = arr[i];
7        }
8      }
9      return best;
10   }
11
12   public static void main(String[] args) {
13     int[] temps = {-5, -2, -8};
14     int result = max(temps);
15     System.out.println("Max: " + result);
16   }
17 }
```



Resume: `i = 1`, `-2 > 0` is false again; `best` is unchanged.

Step 5 / 8

VARIABLES

`arr = [-5, -2, -8]` `best = 0` `i = 1`

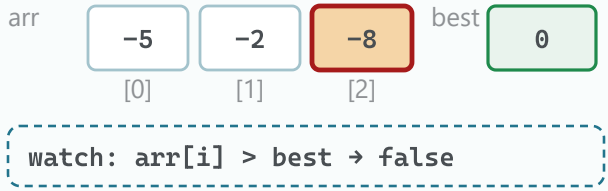
CALL STACK

- `max(arr)`
- `main()`

OUTPUT

Finding the Defect with a Debugger

```
1  public class FindMax {
2    public static int max(int[] arr) {
3      int best = 0;
4      for (int i = 0; i < arr.length; i++) {
▶        if (arr[i] > best) {
6          best = arr[i];
7        }
8      }
9      return best;
10   }
11
12   public static void main(String[] args) {
13     int[] temps = {-5, -2, -8};
14     int result = max(temps);
15     System.out.println("Max: " + result);
16   }
17 }
```



`i = 2`, `-8 > 0` is false. Suspicious: `best` never changes!

Step 6 / 8

VARIABLES

`arr = [-5, -2, -8]` `best = 0` `i = 2`

CALL STACK

- `max(arr)`
- `main()`

OUTPUT

Finding the Defect with a Debugger

```
1  public class FindMax {
2    public static int max(int[] arr) {
3      int best = 0;
4      for (int i = 0; i < arr.length; i++) {
5        if (arr[i] > best) {
6          best = arr[i];
7        }
8      }
▶      return best;
10   }
11
12   public static void main(String[] args) {
13     int[] temps = {-5, -2, -8};
14     int result = max(temps);
15     System.out.println("Max: " + result);
16   }
17 }
```



The loop ends and 0 is returned. **Defect found**: the initial value 0 is wrong when all elements are negative. It should be `best = arr[0]`.

Step 7 / 8

VARIABLES

`arr = [-5, -2, -8]` `best = 0`

CALL STACK

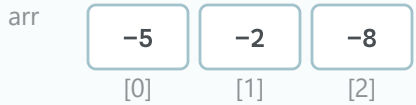
`max(arr)`

`main()`

OUTPUT

Finding the Defect with a Debugger

```
1  public class FindMax {
2    public static int max(int[] arr) {
3      int best = 0;
4      for (int i = 0; i < arr.length; i++) {
5        if (arr[i] > best) {
6          best = arr[i];
7        }
8      }
9      return best;
10   }
11
12   public static void main(String[] args) {
13     int[] temps = {-5, -2, -8};
14     int result = max(temps);
▶      System.out.println("Max: " + result);
16   }
17 }
```



The failure on screen: `Max: 0`. After the fix, this case is added as a **unit test**.

Step 8 / 8

VARIABLES

`temps = [-5, -2, -8]` `result = 0`

CALL STACK

`main()`

OUTPUT

```
Max: 0
```

Step Into, Step Over, Step Out

```
1  public class StepDemo {
2    static int square(int x) {
3      return x * x;
4    }
5
6    static int sumSquares(int a, int b) {
7      int sq1 = square(a);
8      int sq2 = square(b);
9      return sq1 + sq2;
10   }
11
12   public static void main(String[] args) {
▶      int s = sumSquares(3, 4);
14     System.out.println("s = " + s);
15   }
16 }
```

Stopped at a breakpoint

Debug

The breakpoint is on line 13: the line **has not run yet**.

Step 1 / 9

VARIABLES

—

CALL STACK

`main()`

OUTPUT

Step Into, Step Over, Step Out

```
1  public class StepDemo {
2    static int square(int x) {
3      return x * x;
4    }
5
6    static int sumSquares(int a, int b) {
▶      int sq1 = square(a);
8      int sq2 = square(b);
9      return sq1 + sq2;
10   }
11
12   public static void main(String[] args) {
13     int s = sumSquares(3, 4);
14     System.out.println("s = " + s);
15   }
16 }
```

Step Into F7

Step Into: we are inside `sumSquares`; a new frame is pushed on the stack.

Step 2 / 9

VARIABLES

a = 3 b = 4

CALL STACK

sumSquares(3, 4)

main()

OUTPUT

Step Into, Step Over, Step Out

```
1  public class StepDemo {
2    static int square(int x) {
3      return x * x;
4    }
5
6    static int sumSquares(int a, int b) {
7      int sq1 = square(a);
▶      int sq2 = square(b);
9      return sq1 + sq2;
10   }
11
12   public static void main(String[] args) {
13     int s = sumSquares(3, 4);
14     System.out.println("s = " + s);
15   }
16 }
```

Step Over F8

Step Over: `square(3)` ran, but we **did not step into it**; `sq1 = 9`.

Step 3 / 9

VARIABLES

`a = 3` `b = 4` `sq1 = 9`

CALL STACK

`sumSquares(3, 4)`

`main()`

OUTPUT

Step Into, Step Over, Step Out

```
1  public class StepDemo {
2    static int square(int x) {
▶      return x * x;
4    }
5
6    static int sumSquares(int a, int b) {
7      int sq1 = square(a);
8      int sq2 = square(b);
9      return sq1 + sq2;
10   }
11
12   public static void main(String[] args) {
13     int s = sumSquares(3, 4);
14     System.out.println("s = " + s);
15   }
16 }
```

Step Into F7

Step Into: this time we are inside `square(4)`; only `x` is visible.

Step 4 / 9

VARIABLES

`x = 4`

CALL STACK

- `square(4)`
- `sumSquares(3, 4)`
- `main()`

OUTPUT

Step Into, Step Over, Step Out

```
1  public class StepDemo {
2    static int square(int x) {
3      return x * x;
4    }
5
6    static int sumSquares(int a, int b) {
7      int sq1 = square(a);
▶      int sq2 = square(b);
9      return sq1 + sq2;
10   }
11
12   public static void main(String[] args) {
13     int s = sumSquares(3, 4);
14     System.out.println("s = " + s);
15   }
16 }
```

Step Out `Shift+F8`

Step Out: `square` returned 16; we are back on the calling line, about to finish the assignment.

Step 5 / 9

VARIABLES

`a = 3` `b = 4` `sq1 = 9`

CALL STACK

`sumSquares(3, 4)`

`main()`

OUTPUT

Step Into, Step Over, Step Out

```
1  public class StepDemo {
2    static int square(int x) {
3      return x * x;
4    }
5
6    static int sumSquares(int a, int b) {
7      int sq1 = square(a);
8      int sq2 = square(b);
▶      return sq1 + sq2;
10   }
11
12   public static void main(String[] args) {
13     int s = sumSquares(3, 4);
14     System.out.println("s = " + s);
15   }
16 }
```

Step Over F8

Step Over: `sq2 = 16` .

Step 6 / 9

VARIABLES

`a = 3` `b = 4` `sq1 = 9` `sq2 = 16`

CALL STACK

`sumSquares(3, 4)`

`main()`

OUTPUT

Step Into, Step Over, Step Out

```
1  public class StepDemo {
2    static int square(int x) {
3      return x * x;
4    }
5
6    static int sumSquares(int a, int b) {
7      int sq1 = square(a);
8      int sq2 = square(b);
9      return sq1 + sq2;
10   }
11
12   public static void main(String[] args) {
▶      int s = sumSquares(3, 4);
14     System.out.println("s = " + s);
15   }
16 }
```

Step Out `Shift+F8`

Step Out: `sumSquares` returned 25; we are back in `main`.

Step 7 / 9

VARIABLES

—

CALL STACK

`main()`

OUTPUT

DEBUGGING · STEP BY STEP

Step Into, Step Over, Step Out

```
1  public class StepDemo {
2    static int square(int x) {
3      return x * x;
4    }
5
6    static int sumSquares(int a, int b) {
7      int sq1 = square(a);
8      int sq2 = square(b);
9      return sq1 + sq2;
10   }
11
12   public static void main(String[] args) {
13     int s = sumSquares(3, 4);
▶      System.out.println("s = " + s);
15   }
16 }
```

Step Over F8

Step Over: `s = 25` .

Step 8 / 9

VARIABLES

`s = 25`

CALL STACK

`main()`

OUTPUT

Step Into, Step Over, Step Out

```
1  public class StepDemo {
2    static int square(int x) {
3      return x * x;
4    }
5
6    static int sumSquares(int a, int b) {
7      int sq1 = square(a);
8      int sq2 = square(b);
9      return sq1 + sq2;
10   }
11
12   public static void main(String[] args) {
13     int s = sumSquares(3, 4);
14     System.out.println("s = " + s);
▶    }
16 }
```

Step Over F8

Step Over: the output is printed. **Resume** ends the program.

Step 9 / 9

VARIABLES

`s = 25`

CALL STACK

`main()`

OUTPUT

```
s = 25
```

Debugging on the Command Line: `jdb`

```java
public class Jdb {
  static int square(int x) {
    int result = x * x;
    return result;
  }

  public static void main(String[] args) {
    int sum = 0;
    for (int i = 1; i <= 3; i++) {
      sum += square(i);
    }
    System.out.println("Sum: " + sum);
  }
}
```

COMPILE AND START UNDER THE DEBUGGER

```
javac -g Jdb.java
jdb Jdb
```

`-g` : also writes local variable names into the class file; the `locals` command needs them.

jdb (Java Debugger) is the **command-line** debugger that ships with the JDK; it does with commands what an IDE does with clicks.

Command	Meaning
`stop at Class:line`	sets a breakpoint
`clear Class:line`	removes a breakpoint
`run` / `cont`	starts / continues
`step` / `next` / `step up`	Step Into / Over / Out
`where`	call stack
`locals`, `print x`	local variables, one value

To attach to a running application, start it with `java -agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=5005 App` ; then an IDE or `jdb -attach 5005` connects.

A `jdb` Session

jdb Jdb (1)

```
> stop at Jdb:4
Deferring breakpoint Jdb:4.
It will be set after the class is loaded.
> run
VM Started: Set deferred breakpoint Jdb:4
Breakpoint hit: "thread=main", Jdb.square(),
line=4 bci=4
main[1] where
  [1] Jdb.square (Jdb.java:4)
  [2] Jdb.main (Jdb.java:10)
main[1] locals
Method arguments:
x = 1
Local variables:
result = 1
```

jdb Jdb (2)

```
main[1] cont
Breakpoint hit: "thread=main", Jdb.square(),
line=4 bci=4
main[1] print result
 result = 4
main[1] clear Jdb:4
Removed: breakpoint Jdb:4
main[1] cont
> Sum: 14
```

- The breakpoint is on line 4 (`return result;`): the program stops before running it.
- `where` : `main` (line 10) → `square` . `locals` : `x = 1` , `result = 1` .
- `cont` stops again on the loop's second pass: `result = 4` .
- With the breakpoint cleared, the program finishes: 1 + 4 + 9 = 14.

Systematic Debugging

1 **Reproduce**
Find the smallest input that triggers the failure every time (e.g. `{-5, -2, -8}`).

A bug you cannot reproduce, you cannot debug.

Systematic Debugging

1 **Reproduce**
Find the smallest input that triggers the failure every time (e.g. `{-5, -2, -8}`).

A bug you cannot reproduce, you cannot debug.

2 **Write it as a test**
Write a unit test that fails on this input: the red test is proof of the bug.

The test will also show that the fix works.

Systematic Debugging

1. **Reproduce**
   Find the smallest input that triggers the failure every time (e.g. `{-5, -2, -8}`).

   A bug you cannot reproduce, you cannot debug.

2. **Write it as a test**
   Write a unit test that fails on this input: the red test is proof of the bug.

   The test will also show that the fix works.

3. **Narrow it down**
   Start at the first line of the stack trace; watch variables at a breakpoint and halve the suspect region.

   Look for the line where a value first goes wrong. `assert` statements in the code (enabled with `-ea`) catch the problem where it starts.

Systematic Debugging

1. **Reproduce**
   Find the smallest input that triggers the failure every time (e.g. `{-5, -2, -8}`).

   A bug you cannot reproduce, you cannot debug.

2. **Write it as a test**
   Write a unit test that fails on this input: the red test is proof of the bug.

   The test will also show that the fix works.

3. **Narrow it down**
   Start at the first line of the stack trace; watch variables at a breakpoint and halve the suspect region.

   Look for the line where a value first goes wrong. `assert` statements in the code (enabled with `-ea`) catch the problem where it starts.

4. **Form and test a hypothesis**
   Propose an explanation such as "`best` starts at 0, so it never updates", then confirm it in the debugger.

   Changing code on a hunch breeds new bugs.

Systematic Debugging

1. **Reproduce**
   Find the smallest input that triggers the failure every time (e.g. `{-5, -2, -8}`).

   A bug you cannot reproduce, you cannot debug.

2. **Write it as a test**
   Write a unit test that fails on this input: the red test is proof of the bug.

   The test will also show that the fix works.

3. **Narrow it down**
   Start at the first line of the stack trace; watch variables at a breakpoint and halve the suspect region.

   Look for the line where a value first goes wrong. `assert` statements in the code (enabled with `-ea`) catch the problem where it starts.

4. **Form and test a hypothesis**
   Propose an explanation such as "`best` starts at 0, so it never updates", then confirm it in the debugger.

   Changing code on a hunch breeds new bugs.

5. **Fix, then run all tests**
   The new test must turn green and the old ones stay green (no regression).

   The test stays in the suite for good.

Profilers

- A profiler shows **where a program spends its time**: which method was called how often, how long it took, how much memory it allocated.
- It can show the **call graph** (who called whom) in several forms: a flat list, a tree, a flame graph.
- The technical details **vary** with the compiler and JVM version.
- Rule: **measure first, then optimize**. Most of the time goes into a small "hot spot"; optimizing on a hunch usually hits the wrong place.

Old way: `java -Xprof`
The built-in profiler of older JVMs; **removed in Java 10**. On Java 21:

```
OpenJDK 64-Bit Server VM warning: Ignoring option -Xprof; support was removed in 10.0
```

Today
JDK Flight Recorder (JFR) + **JDK Mission Control**, **VisualVM**, IDE profilers. Below, the program runs with a JFR recording, then we summarize it.

```
java -XX:StartFlightRecording=filename=rec.jfr Profile
jfr summary rec.jfr
```

Measuring the Hot Spot

```
public class Profile {
  static String concatPlus(int n) {
    String s = "";
    for (int i = 0; i < n; i++) {
      s += i;  // new String each pass
    }
    return s;
  }

  static String concatBuilder(int n) {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < n; i++) {
      sb.append(i);
    }
    return sb.toString();
  }
  // main: see the next box
}
```

- `s += i` builds **a new String** each pass and copies the old one: total work grows as n^2.
- `StringBuilder` appends to one buffer: work grows as n. Hundreds of times faster.

```
public static void main(String[] args) {
  long t0 = System.nanoTime();
  String a = concatPlus(50_000);
  long t1 = System.nanoTime();
  String b = concatBuilder(50_000);
  long t2 = System.nanoTime();
  long ms1 = (t1 - t0) / 1_000_000;
  long ms2 = (t2 - t1) / 1_000_000;
  System.out.println("+= : " + ms1 + " ms");
  System.out.println("SB : " + ms2 + " ms");
  System.out.println(a.equals(b));
}
```



Output of one run (varies by machine)

```
+= : 873 ms
SB : 2 ms
true
```

Code Coverage Tools

- Shows **which parts of the code ran** during a test run.
- Reveals code that **never runs** or is never used.
- Helps find **untested** regions of code.
- Coverage is a key tool for making software more **reliable**.
- Tools: **JaCoCo** (Maven / Gradle), IntelliJ “Run with Coverage”, Eclipse **EclEmma**. Results are shown by coloring the source lines.

Metric	What is counted?
Line	Lines run at least once
Branch	Both the true and false outcome of every `if` / loop condition
Method	Methods called at least once

100% coverage ≠ bug-free code. Coverage says a line **ran**, not that its result was **right**: a test with no `assert` raises coverage too.

Code Coverage: An Example

`Sign` : COVERAGE AFTER TWO TESTS

```
public class Sign {
  public static String sign(int x) {
    if (x > 0) {
      return "positive";
    } else if (x < 0) {
      return "negative";
    }
    return "zero";
  }
}
```

ran never ran

Metric	Coverage
Line (5 executable lines)	4 / 5 = 80%
Branch (2 conditions × true / false)	3 / 4 = 75%

`SignTest`

```
class SignTest {
  @Test
  void positive() {
    assertEquals("positive", Sign.sign(5));
  }

  @Test
  void negative() {
    assertEquals("negative", Sign.sign(-3));
  }
}
```

Both tests are green (✓), but the **false** branch of `x < 0` and `return "zero"` were never tried. The coverage report points to the missing test: `sign(0)`.

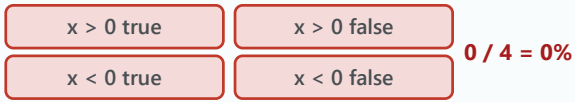
Code Coverage, Step by Step

```
1  public class Sign {
▶    public static String sign(int x) {
3      if (x > 0) {
4        return "positive";
5      } else if (x < 0) {
6        return "negative";
7      }
8      return "zero";
9    }
10 }
```

lines



branches



tests: –

There are 5 executable lines (3, 4, 5, 6, 8) and 4 branches from 2 conditions. No test has run yet: all red.

Step 1 / 8

VARIABLES

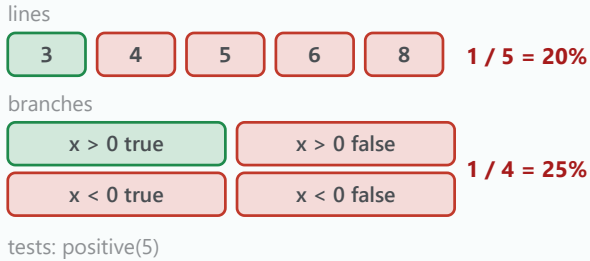
`lines = 0/5` `branches = 0/4`

CALL STACK

empty

Code Coverage, Step by Step

```
1  public class Sign {
2    public static String sign(int x) {
▶      if (x > 0) {
4        return "positive";
5      } else if (x < 0) {
6        return "negative";
7      }
8      return "zero";
9    }
10 }
```



`sign(5)` : line 3 runs and the condition takes the **true** branch.

Step 2 / 8

VARIABLES

`x = 5`

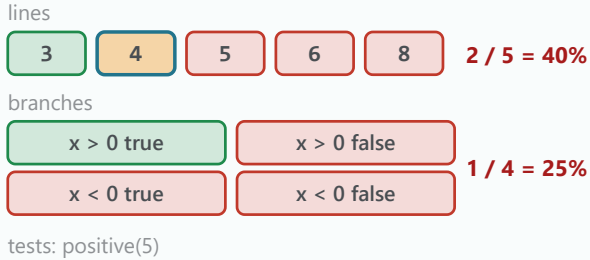
CALL STACK

`sign(5)`

`positive()`

Code Coverage, Step by Step

```
1  public class Sign {
2    public static String sign(int x) {
3      if (x > 0) {
▶        return "positive";
5      } else if (x < 0) {
6        return "negative";
7      }
8      return "zero";
9    }
10 }
```



Returns "positive"; the test passes. Coverage: 2/5 lines, 1/4 branches.

Step 3 / 8

VARIABLES

`x = 5`

CALL STACK

`sign(5)`

`positive()`

Code Coverage, Step by Step

```
1  public class Sign {
2    public static String sign(int x) {
3      if (x > 0) {
4        return "positive";
▶      } else if (x < 0) {
6        return "negative";
7      }
8      return "zero";
9    }
10 }
```

lines

3 | 4 | 5 | 6 | 8 — 3 / 5 = 60%

branches

x > 0 true | x > 0 false

x < 0 true | x < 0 false — 3 / 4 = 75%

tests: positive(5), negative(-3)

`sign(-3)` : `x > 0` is false, `x < 0` is true. Two new branches.

Step 4 / 8

VARIABLES

`x = -3`

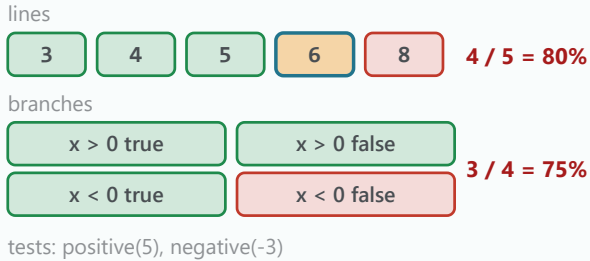
CALL STACK

`sign(-3)`

`negative()`

Code Coverage, Step by Step

```
1  public class Sign {
2    public static String sign(int x) {
3      if (x > 0) {
4        return "positive";
5      } else if (x < 0) {
▶        return "negative";
7      }
8      return "zero";
9    }
10 }
```



Returns "negative". Both tests are green.

Step 5 / 8

VARIABLES

`x = -3`

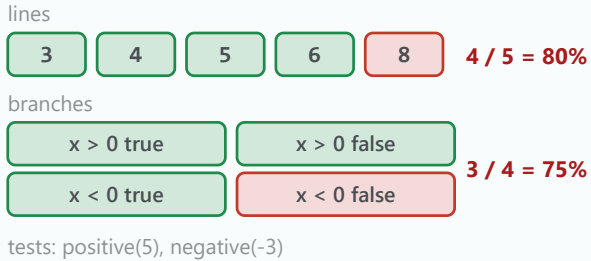
CALL STACK

`sign(-3)`

`negative()`

Code Coverage, Step by Step

```
1  public class Sign {
2    public static String sign(int x) {
3      if (x > 0) {
4        return "positive";
5      } else if (x < 0) {
6        return "negative";
7      }
▶      return "zero";
9    }
10 }
```



Report: 80% lines, 75% branches. Line 8 and the false branch of `x < 0` were **never tried**: the missing test is `sign(0)`.

Step 6 / 8

VARIABLES

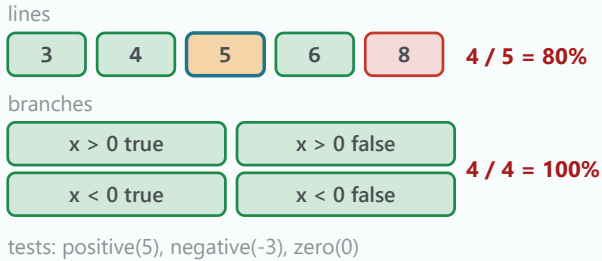
`lines = 4/5` `branches = 3/4`

CALL STACK

empty

Code Coverage, Step by Step

```
1  public class Sign {
2    public static String sign(int x) {
▶      if (x > 0) {
4        return "positive";
▶      } else if (x < 0) {
6        return "negative";
7      }
8      return "zero";
9    }
10 }
```



New test `sign(0)`: both conditions are false. Branch coverage 4/4.

Step 7 / 8

VARIABLES

`x = 0`

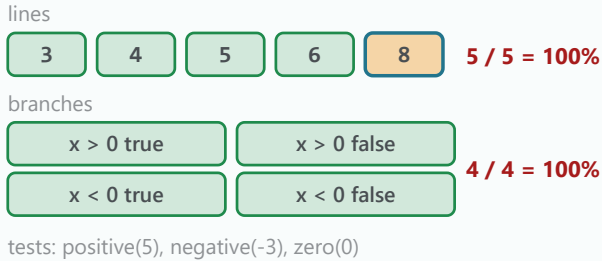
CALL STACK

`sign(0)`

`zero()`

Code Coverage, Step by Step

```
1  public class Sign {
2    public static String sign(int x) {
3      if (x > 0) {
4        return "positive";
5      } else if (x < 0) {
6        return "negative";
7      }
▶      return "zero";
9    }
10 }
```



Returns "zero": 100% line and branch coverage. Still, coverage only says the lines **ran**; `assertEquals` checks correctness.

Step 8 / 8

VARIABLES

`x = 0`

CALL STACK

`sign(0)`

`zero()`

Static Analysis: FindBugs and SpotBugs

- **FindBugs** is a tool for **static analysis** of Java code: it examines the compiled code **without running** the program.
- It finds bugs **quickly**: `null` dereferences, wrong comparisons, unclosed resources…
- FindBugs is no longer developed; its successor is **SpotBugs** (Maven, Gradle, IDE plugins). Similar tools: PMD, Checkstyle, SonarQube.

The input is `yes`, yet the output is `Rejected`. The compiler says nothing; SpotBugs flags both lines.

```java
import java.util.Scanner;

public class SpotDemo {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    String answer = sc.nextLine();
    // result ignored:
    answer.trim();
    // String compared with ==:
    if (answer == "yes") {
      System.out.println("Approved");
    } else {
      System.out.println("Rejected");
    }
  }
}
```

SpotBugs pattern	Warning
`ES_COMPARING_STRINGS_WITH_EQ`	Strings compared with `==`; use `equals`
`RV_RETURN_VALUE_IGNORED`	`trim()` returns a new String; the result is discarded

The First Static Analyzer: the Compiler (`-Xlint`)

```java
public class Lint {
  public static void main(String[] args) {
    int day = 2;
    switch (day) {
      case 1:
        System.out.println("Monday");
      case 2:
        System.out.println("Tuesday");
      case 3:
        System.out.println("Wednesday");
        break;
    }
    int x = 10 / 0;
  }
}
```

java Lint

```
Tuesday
Wednesday
Exception in thread "main"
java.lang.ArithmeticException: / by zero
    at Lint.main(Lint.java:13)
```

javac -Xlint:all Lint.java

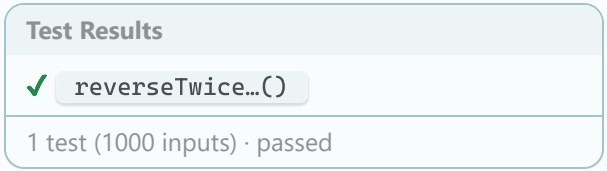
```
Lint.java:13: warning: [divzero] division by
zero
    int x = 10 / 0;
                ^
Lint.java:7: warning: [fallthrough] possible
fall-through into case
      case 2:
      ^
Lint.java:9: warning: [fallthrough] possible
fall-through into case
      case 3:
      ^
3 warnings
```

- Plain `javac Lint.java` compiles **without a single warning**.
- `-Xlint:all` reports suspicious constructs: `case`s with a missing `break` and division by a constant zero.
- Reading warnings is the cheapest way to find a bug **before running** the code.

QuickCheck: Property-Based Testing

- **QuickCheck generates tests automatically**: the tool, not the programmer, picks random inputs.
- Instead of single examples, the programmer writes a **property**: "this must hold for every input".
- If an input breaks the property, it is reported as a **counterexample** (shrunk when possible).
- The original is for Haskell; there are versions **for many languages**; in Java: **jqwik** and **junit-quickcheck**.

The test on the right applies the idea in plain JUnit: 1000 random strings.




```
class PropertyTest {

  static String randomText(Random r) {
    int len = r.nextInt(20);      // 0..19
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < len; i++) {
      sb.append((char) ('a' + r.nextInt(26)));
    }
    return sb.toString();
  }

  @Test
  void reverseTwiceGivesSame() {
    Random r = new Random(42);    // fixed seed
    for (int k = 0; k < 1000; k++) {
      String s = randomText(r);
      String t = TextUtils.reverse(s);
      assertEquals(s, TextUtils.reverse(t));
    }
  }
}
```

A Property Test Finds the Bug in the First Version

```
class PropertyTest2 {

  @Test
  void maxIsLargestElement() {
    Random r = new Random(7);
    for (int k = 0; k < 1000; k++) {
      int[] arr = new int[1 + r.nextInt(5)];
      for (int i = 0; i < arr.length; i++) {
        arr[i] = r.nextInt(201) - 100;      // -100..100
      }
      int best = ArrayUtils.max(arr);
      boolean found = false;
      for (int x : arr) {
        assertTrue(best >= x);              // property 1
        if (x == best) found = true;
      }
      assertTrue(found,                       // property 2
        "Counterexample: " + Arrays.toString(arr));
    }
  }
}
```

Run against the first (buggy) version of `max`:

Test Results

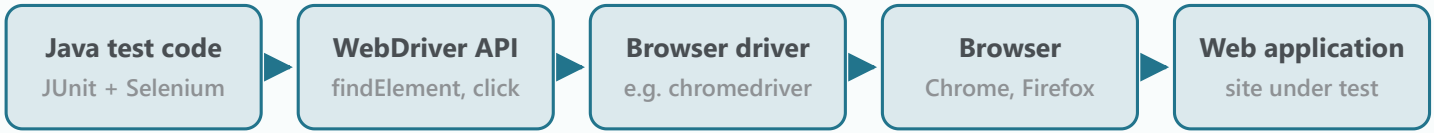
✗ `maxIsLargestElement()`
`Counterexample: [-60, -69, -39, -65, -13] ==> expected: <true> but was: <false>`

1 test · 1 failed

- **Property 1** (the result is not less than any element) **passes** even for the buggy version: 0 is greater than every negative.
- **Property 2** (the result is an element of the array) catches the bug.
- With the fixed version all 1000 inputs pass. Lesson: write **complete** properties; a weak property approves buggy code too.

Selenium

- **Selenium** is a **framework** for testing web applications.
- It makes **browser-based** tests easy: it drives a real browser from code, clicks links, fills in forms, presses buttons and checks that the expected text appears.
- It works with Java, Python, C#, JavaScript and more; in Java it is usually called from a JUnit test.



commands flow left to right; page state (text, title) flows back to the test

Typical scenario
Open the login page → type user name and password → "Log in" → does the title say "Welcome"?

Test level
Tests the whole as the user sees it: **system** and **acceptance** tests.

Cost
Much slower and more brittle than unit tests; keep them few (the top of the pyramid).

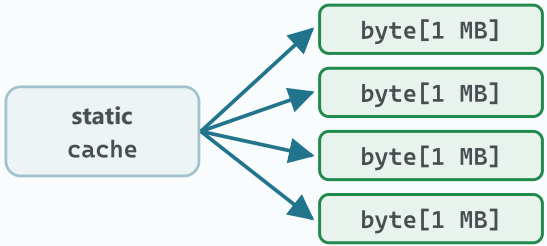
A Memory Leak Example

```java
import java.util.ArrayList;
import java.util.List;

public class Leak {
  static List<byte[]> cache = new ArrayList<>();

  static void handle(int id) {
    byte[] data = new byte[1024 * 1024];  // 1 MB
    cache.add(data);                      // never removed!
  }

  public static void main(String[] args) {
    for (int i = 1; ; i++) {
      handle(i);
      if (i % 20 == 0) {
        System.out.println(i + " requests handled");
      }
    }
  }
}
```



all still reachable → GC cannot free them

java -Xmx256m Leak

```
20 requests handled
…
120 requests handled
Exception in thread "main"
java.lang.OutOfMemoryError: Java heap
space
    at Leak.handle(Leak.java:8)
    at Leak.main(Leak.java:14)
```

The garbage collector (GC) frees only **unreachable** objects; a leak means unneeded objects **still have a reference**. `-Xmx256m` : heap at most 256 MB.

Eclipse Memory Analyzer (MAT)

- Finds memory leaks by **analyzing the Java heap**.
- Used for **memory optimization**: which objects take how much space, and who holds them?
- Input: a **heap dump** (`.hprof`): a snapshot of every object at one moment.
- Key reports: **Leak Suspects**, **Dominator Tree** (who holds the most memory), **Histogram** (object count per class).

TAKING A HEAP DUMP

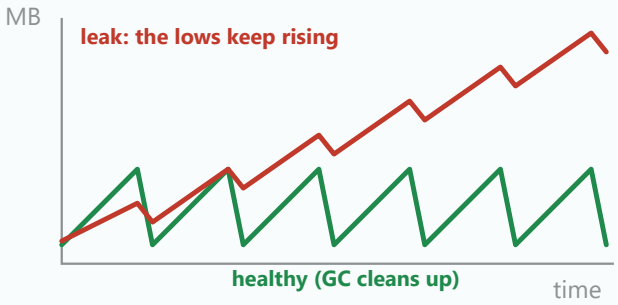
```
// on OutOfMemoryError
java -XX:+HeapDumpOnOutOfMemoryError Leak
// from a running program
jps
jcmd <pid> GC.heap_dump heap.hprof
```

What does the example show?

The Leak Suspects report shows that the `Leak.cache` list and its `byte[]` arrays hold almost the whole heap. Fix: remove data once it is done, or use a size-bounded cache.

VisualVM

- Visualizes **memory use**, **thread activity**, CPU load and more, **live**.
- Used to **monitor and analyze the performance** of Java applications: it attaches to running JVMs, profiles them and takes heap and thread dumps.
- Free and open source (visualvm.github.io); it used to ship with the JDK as `jvisualvm`.
- Command-line counterparts: `jps` (list JVMs), `jstat`, `jcmd`, `jconsole`.



The heap graph gives a leak away: if the lows after each garbage collection keep rising, objects are not being freed. Next, take a heap dump and examine it in MAT.

Summary: Which Tool for Which Question?

Question	Technique / tool
Does this method give the right result?	**Unit testing**: JUnit, `@Test`, `assertEquals`, `assertThrows`
Do the parts work together, as a whole and for the user?	Module, sub-system, system and **acceptance** (alpha / beta) testing; **Selenium** for the web
Did the change break something old?	**Regression testing**: all automated tests run on every change (CI)
Which inputs should I pick?	**Boundary values**, equivalence classes, `@ParameterizedTest`
Is it right for inputs I did not pick?	**Property-based testing** (QuickCheck, jqwik)
How much of the code do my tests run?	**Code coverage**: JaCoCo, EclEmma
Are there bugs visible without running the code?	**Static analysis**: `javac -Xlint`, SpotBugs (FindBugs)
Why does it misbehave?	**Debugger**: breakpoints, Step Into / Over / Out, watches; `jdb`
Where does the time go?	**Profiler**: JFR, VisualVM
Why does memory fill up?	Monitor with **VisualVM**; examine a heap dump in **Eclipse Memory Analyzer**

SUMMARY

Common Mistakes

Swapping expected and actual

`assertEquals(calc.add(2, 3), 5)` also works, but on failure the message reads backwards: `expected` shows the actual result.

Forgetting the delta for doubles

`assertEquals(0.3, 0.1 + 0.2)` fails: `expected: <0.3> but was: <0.30000000000000004>`.

Missing `import static`

`assertEquals(5, x)` → compile error: `cannot find symbol: method assertEquals(int,int)`.

Mixing JUnit 4 and 5

`org.junit.Test` and `org.junit.jupiter.api.Test` differ; the wrong `import` can stop tests from running at all.

`@BeforeAll` method not `static`

JUnit 5 will not run the tests: `@BeforeAll method '…' must be static unless the test class is annotated with @TestInstance(Lifecycle.PER_CLASS).`

Tests that depend on each other

Relying on an object left by another test: when the order changes, tests fail at random. Each test sets up its own data.

Tests with no assertions

Calling a method without any `assert`: the test is always ✓ and proves nothing.

Testing only the “happy path”

Boundaries (0, empty array, negatives), invalid input and exceptions get forgotten; that is exactly where bugs live.

SUMMARY

Sample Problems (1)

PROBLEM 1: WHICH TESTS PASS, WHICH FAIL?

```
class QuizTest {
  @Test void a() { assertEquals(7, 3 + 4); }
  @Test void b() { assertEquals(0.3, 0.1 + 0.2); }
  @Test void c() { assertTrue("Java".length() == 4); }
  @Test void d() { assertThrows(ArithmeticException.class,
      () -> System.out.println(10 / 2)); }
}
```

PROBLEM 2

If `letterGrade` had `score > 50` by mistake instead of `score >= 50`, which row of the parameterized test would fail? With what message?

PROBLEM 3

What is the minimum number of tests for 100% **branch coverage** of the method below? Choose the inputs.

```
public class Abs {
  static int abs(int x) {
    if (x < 0) {
      return -x;
    }
    return x;
  }
}
```

▶ **Show answers**

SUMMARY

Sample Problems (2)

PROBLEM 4: WHAT IS WRONG WITH THIS TEST CLASS?

```
class CounterTest {
  static int count = 0;

  @Test void increment() {
    count++;
    assertEquals(1, count);
  }

  @Test void incrementTwice() {
    count += 2;
    assertEquals(2, count);
  }
}
```

PROBLEM 5

In `StepDemo`, the debugger is stopped at `int s = sumSquares(3, 4);`. After two **Step Over**s in a row, which line are we on, what is on the screen, and what is `s`?

▶ **Show answers**

SUMMARY

Sample Problems (3): Which Tool?

Problem 6: Which technique or tool fits each situation?

1. The program crashes with `OutOfMemoryError` after 10 minutes.
2. Generating a report takes 30 seconds; which method is slow?
3. Did the new discount code break the old invoice calculation?
4. Does the site's sign-up form work in every browser?
5. Are there forgotten `break`s in a `switch`?
6. A loop produces a wrong value on iteration 10,000; what are the variables at that moment?

▶ **Show answers**

SUMMARY

Exercises (1)

1. Write a JUnit 5 test class for a `Rectangle` class (`width` , `height` , `area()` , `perimeter()`). A negative side length must throw `IllegalArgumentException` .
2. Use `@ParameterizedTest` to test `static boolean isPrime(int n)` at the boundary values (0, 1, 2, 3, 4, 97, 100).
3. Add `transfer(BankAccount target, double amount)` to `BankAccount` ; write its tests first (TDD). Which tests do you need?

▶ **Show answers**

SUMMARY

Exercises (2)

4. A method that averages an array gives wrong results for some arrays (e.g. 1.0 for `{1, 2}`). Write down the steps to find the defect with a debugger.
5. Write a test that checks the property “in a sorted array no element is greater than the next one” on 1000 random arrays. Is this property enough on its own?
6. Which test must be added to bring branch coverage of `Sign` to 100%?

▶ **Show answers**

References

- JUnit Team, *JUnit 5 User Guide*, junit.org/junit5/docs/current/user-guide (Writing Tests, Assertions, Parameterized Tests).
- Oracle, *The Java Tutorials* and *JDK 21 Tool Specifications*: `javac` (`-Xlint`), `jdb`, `jcmd`, `jfr`.
- Oracle, *Programming With Assertions* (Java SE documentation).
- J. Gosling et al., *The Java Language Specification*, Java SE 21 Edition (14.10 The assert Statement).
- P. Deitel and H. Deitel, *Java: How to Program*, 11th ed., Pearson.
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12th ed., Pearson (Chapter 7: Exceptions, Assertions, and Logging; debugging tips).
- K. Beck, *Test-Driven Development: By Example*, Addison-Wesley.
- Tool documentation: SpotBugs (spotbugs.github.io), JaCoCo (jacoco.org), Selenium (selenium.dev), Eclipse Memory Analyzer (eclipse.dev/mat), VisualVM (visualvm.github.io), jqwik (jqwik.net).

The End

Chapter 7: Testing and Debugging

Previous: Chapter 6: Exceptions · Next: Chapter 8: Input and Output