Software Quality

1. **Factors in Project Success & Failure**
    A. **Software Crisis**
        - Many high-profile failures happened due to software bugs:
            - AT&T (1990): long-distance outage from a missing break in a switch statement.
            - Ariane 5 rocket (1996): exploded 37 seconds after launch.
            - NYSE (2001): halted trading.
            - British Airways (2017): affected 75,000 passengers.
        - Lesson: Software bugs can have massive real-world consequences.

    B. **The CHAOS Report (Standish Group)**
        - 1995: Only 16.2% of IT projects succeeded.
        - 2012:
            - 39%: Successful (on-time, on-budget).
            - 43%: Challenged (late or over-budget).
            - 18%: Impaired (canceled).

2. **Case Study: HealthCare.gov Launch (2013)**
    A. **Major Issues**
        - Performance: Slow response time (>8s).
        - Stability: Crashes; availability only 43%.
        - Functionality: Bugs; incomplete data.
        - Scalability: Couldn't handle >1,100 users (expected 50k+).
        - Error rate: ~6% per page.
        - Completion rate: <30%.

    B. **Engineering Failures**
        - Inadequate testing: Started full testing 2 weeks before launch.
        - No end-to-end or stress testing.
        - Requirements kept changing; late coding started.
        - Poor change management: no agile, no flexibility.

    C. **Management Failures**
        - Only 10 devs on a crucial module near launch.
        - Poor coordination, fragmented authority.
        - Contracting prioritized politics over performance.

    D. **Aftermath**
        - CGI was replaced by Accenture in 2014.
        - Total cost: Over $2 billion.
        - Lessons:
            - Adopt Agile practices.
            - Test early and continuously.
            - Ensure clear roles and proper planning.

3. **Software Reliability**

   Key Metrics

   - Performance: How fast it responds (e.g., page loads in <2s).
   - Scalability: Handles more users (e.g., 10K concurrent users).
   - MTTF (Mean Time to Failure): Avg. time software works before crashing.
   - MTTR (Mean Time to Repair): Time to fix it after failure.
   - MTBF = MTTF + MTTR.

   Availability

   - Formula: Availability = [MTTF / (MTTF + MTTR)] × 100%
   - Example: “Five nines” (99.999%) = 5 mins downtime/year.

4. **Error Rate vs. Completion Rate**
   - Error Rate: % of operations (e.g., page loads) that fail.
   - Completion Rate: % of attempts that finish the process.
   - A process can complete with errors (so completion ≠ success).

5. **Software Testing**

   A. **Testing Levels**
      - Unit Testing: Individual functions/modules (done by devs).
      - Integration Testing: Check how modules work together.
      - System Testing: End-to-end testing of the full system.
      - Acceptance Testing: Done by customers/users to accept or reject.

   B. **Regression Testing**
      - Re-run previous tests after code changes.
      - Prevents old bugs from reappearing.

6. **Verification vs. Validation**

Concept	**Question**	**Example**
Verification	Are we building the software right?	Functional testing, code review
Validation	Are we building the right software?	Usability tests, user feedback

7. **V-Model in SDLC**
    - Left Side: Specification & design.
    - Right Side: Corresponding tests.

Development Phase	**Corresponding Test**
Unit design	Unit testing
Subsystem design	Integration testing
System specs	System testing
Business needs	Acceptance testing

8. **Software Quality Attributes**

   External Qualities
    - Correctness: Matches the specification.
    - Reliability: Works consistently.
    - Safety: Avoids dangerous failures.
    - Robustness: Handles errors gracefully.
    - Performance/Scalability/Efficiency
    - Usability: Easy for users.
    - Security: Protects data and system.

   Internal Qualities
    - Maintainability: Easy to update.
    - Reusability: Use in other projects.
    - Portability: Run on other platforms.

9. **Correctness vs. Reliability vs. Robustness**
    - Correctness = All or nothing (either 100% or not correct).
    - Reliability = Probabilistic success under normal conditions.
    - Robustness = Handles unexpected inputs or crashes gracefully.

   Example:
    - A traffic light system:
        - Correct = green/red logic always works.
        - Robust = flashes yellow in error mode.
        - Safe = never gives two conflicting green signals.

10. **Cost of Software Defects**
    A. **Cost Increases Over Time**
        - Defects found earlier are cheaper to fix.
            - Code review: 3 mins.
            - System testing: 1400 mins.

B. Defects Lifecycle

- Most defects are introduced early (requirements/design).
- Most are found late (testing or after release).
- Prevention = cheap. Repair = expensive.

11. Cost of Quality

- Prevention Costs: Training, planning, code review.
- Internal Failure Costs: Fixing bugs before release.
- External Failure Costs: Support, returns, lost trust.

12. Acceptance Testing Techniques

- Random Testing: Uses statistical sampling.
- Alpha Testing: Simulated users in a controlled setting.
- Beta Testing: Real users in real environments.

13. Summary of Key Concepts

- Spectrum of software qualities: Not just correctness, but also usability, security, maintainability, etc.
- Quality = conformance to requirements (not user expectation).
- Testing is continuous — not a single phase.
- Use the V-model to map design phases to test phases.
- Focus on early defect detection to save time and cost.
- Understand the metrics (MTTF, MTTR, Availability, etc.).

Testing Throughout The SDLC

1. **Why Testing Is Part of the SDLC**
    - Testing = Verification + Validation
        - Verification: Did we build it right?
        - Validation: Did we build the right thing?
    - Testing must align with development activities.
    - It starts early and continues throughout the lifecycle.

2. **Testing in SDLC Models**

    A. Waterfall Model
        - Testing happens after development.
        - Risk: bugs found too late.

    B. V-Model (Verification & Validation Model)
        - Testing is planned alongside development phases.
        - Each dev phase has a matching test phase.

Dev Phase	**Test Phase**
Requirements	Acceptance Testing
System Design	System Testing
Architecture Design	Integration Testing
Module Design / Coding	Unit / Component Testing

    C. Iterative-Incremental Model
        - Dev is broken into small cycles (sprints).
        - Testing happens after each increment.
        - Regression testing becomes more critical after every update.

3. **Test Levels Overview**

Level	**Scope**	**Performed By**
Unit	Individual functions/classes	Developer
Integration	Interfaces between components	Developer/Test team
System	Whole system (black box)	QA/Test team
Acceptance	Real-world usage scenarios	Customer/User

4. **Unit/Component Testing**
    - Tests: Methods, classes, modules.
    - Uses stubs/drivers to simulate missing components.
    - Focus: Functionality, memory usage, edge cases.

5. **Integration Testing**
    - Tests connections between components/systems:
        - APIs, databases, OS, files, hardware
    - Types:
        - Component integration: software parts
        - System integration: external systems
    - Strategies:
        - Start with risky/critical modules
        - Use incremental integration (not Big Bang)

6. **System Testing**
    - Tests the complete integrated system.
    - Based on:
        - Requirements
        - Use cases
        - Risk analysis
    - Should simulate the production environment.
    - Uses black box then white box techniques.

7. **Acceptance Testing**
    - Confirms the system is ready for delivery.
    - Types:
        - User Acceptance Testing (UAT): By end-users
        - Operational Testing: Backup, recovery, admin tasks
        - Regulatory/Contractual: Legal, safety, compliance
        - Alpha/Beta Testing: Pre-release user feedback

8. **Test Types**

Category	**Examples**
Functional (Black Box)	Interoperability, Compliance, Suitability
Non-Functional	Performance, Usability, Security, Portability
Structural (White Box)	Code coverage, branches, logic paths
Change-related	Regression, Confirmation

9. **Functional Testing**
    - Checks what the system does (based on specs).
    - Covers workflows, business logic, expected outputs.

10. Non-Functional Testing

- Checks how the system behaves under conditions like:
  - Load
  - Security
  - Usability
  - Efficiency

11. Structural Testing (White Box)

- Internal logic & code structure.
- Tools measure:
  - Statement coverage
  - Branch coverage
  - Path coverage

12. Testing After Changes

A. Confirmation Testing

- Verify that a bug fix actually fixed the bug.
- Always done after bug fixing.

B. Regression Testing

- Ensure new changes don't break old functionality.
- Grows over time and should be automated.
- Only rerun affected or critical parts if the full suite is too big.

13. Maintenance Testing

- Done on an already deployed system.
- Triggered by:
  - New features
  - Patches or hotfixes
  - Upgrading environments (OS, DB, etc.)
  - Migrating platforms
  - Retiring a system (e.g., archiving data)

🔍 **Impact Analysis**

- Used to decide how much retesting is needed.
- Focus on areas affected by change.

Cheat Sheet Summary

Concept	Meaning
Verification	Build it right
Validation	Build the right thing
Test Levels	Unit, Integration, System, Acceptance
Testing Types	Functional, Non-functional, Structural
Change Testing	Regression, Confirmation, Maintenance
Best Model	V-Model promotes early and structured testing
Automation	Critical for large regression test suites

Test case

1. **What Testing Actually Is**
    - Misconception: Testing doesn't improve software.
        - Like weighing yourself doesn’t make you lose weight.
        - Testing helps identify issues—fixing them is what improves quality.

2. **What Is a Test Case?**
    - A test case defines the input, action, and expected output to test one behavior of the software.
    - Usually stored in a test suite.
    - Includes:
        - ID/Name
        - Requirement tested
        - Preconditions (e.g., "user is logged in")
        - Steps (clicks, inputs, etc.)
        - Expected Result
    - ✅ Must be repeatable – same outcome every time.

3. **How to Write Good Test Cases**
    A. **Understand the context**
        - Know the variables, limits, domain rules.
        - Think like the user.
        - Consider mistakes, invalid inputs, edge cases.

    B. **What to test**
        - Correct behavior: Right output for valid input.
        - Robustness: Proper error handling for bad input.
        - User Acceptance: Real-world usage.

    C. **Rules for good test cases**
        - Be simple and transparent.
        - Avoid assumptions.
        - Stick to specs.
        - Cover edge/boundary cases.
        - Include:
            - Normal cases
            - Illegal input (e.g. negative numbers, empty fields)
            - Impossible conditions (e.g. triangle sides: 2, 3, 10)
            - Input types (e.g. text in a number field)

4. **Testing Techniques**
    - **Boundary Value Analysis (BVA):**
        - Focus on edge values: min, max, just inside/outside limits.
        - Example: if valid age is 18–60, test 17, 18, 60, 61
    - **Permutations:**
        - For input sets: test combinations like (3,4,5), (4,3,5), (5,4,3)
    - **Illegal Inputs:**
        - Negative values, text instead of number, null values, etc.

5. **Testing Tips**
    - Don’t aim to test everything—pick tests likely to find bugs.
    - Focus on:
        - Edge conditions (0, -1, max values)
        - Empty inputs, nulls
        - Repeated actions (does "undo" still work after "redo"?)
    - Be cautious with comparisons:
        - Floating points: a == b can fail due to tiny differences.
        - Use .equals() in Java for strings, not ==.

6. **Scaffolding (Support Code)**
    - Scaffolding = test support code (not part of the product).
    - Includes:
        - Drivers: fake “main” to run tests
        - Stubs: fake functions called by the code
        - Harness: links all parts together (e.g., JUnit)

7. **Oracles: How to Check Test Results**
    - Oracle = how we decide if a test passed.
    - Comparison-based: Compare output to expected result.
    - Self-checking: System checks internal data for errors.
    - Capture & Replay:
        - Run once manually.
        - Record input/output.
        - Replay automatically later.

8. **Smoke Tests**
    - Quick, shallow set of test cases.
    - Verify basic functionality (e.g., login works, no crash).
    - Used to confirm a build is “testable”.
    - Not a substitute for full testing.

9. **Test Execution**
    - Happens after developers deliver the "alpha build."
    - Typically includes multiple rounds:
        - Test new features.
        - Regression tests – rerun old tests to check nothing broke.

- When is testing done?
    - No defects found, or remaining bugs meet acceptance criteria.

10. Automated Test Execution

- Test design = creative work (done by humans).
- Test execution = mechanical work (can be automated).
- Design once, run many times with different data.

11. From Specifications to Concrete Tests

- **Early test design might say:**
    - "Large positive number"
    - "Sorted list"
- **During execution, generate:**
    - 420023
    - [“Alpha”, “Beta”, “Omega”]

12. Defect Tracking

- **A defect tracking system:**
    - Logs all bugs
    - Tracks status (open, in progress, fixed)
    - Coordinates testers, developers, and PMs

13. Trustworthy Tests

- Test one thing at a time.
- Prefer 10 small tests over 1 huge one.
- Avoid logic in tests (if/else, loops, try/catch).
- Failing fast helps find issues quicker.

14. Summary (Cheat Sheet)

Concept	**Key Idea**
Test Case	A repeatable set of steps to test a specific behavior
Good Test	Covers edge, normal, and error conditions
Scaffolding	Support code (drivers, stubs, harness)
Oracle	Defines how to judge pass/fail
Smoke Test	Lightweight test to verify readiness
Regression Test	Re-run to check new changes didn’t break anything
Trustworthy Tests	Small, focused, no logic, self-checking
Test Done When	All tests pass or defects meet criteria

Static Testing + White Box Testing

1. **White Box Testing: What It Is**
    - Also called Structural Testing.
    - Tests the internal logic of the program.
    - Requires access to the source code.
    - Test cases are written based on:
        - Code paths
        - Branches
        - Conditions
        - Loops

2. **Why White Box Testing?**
    - Checks what the code actually does, not what it's supposed to do.
    - Catches issues like:
        - Divide by zero
        - Unused variables
        - Unreachable code
    - Defects are detected early, reducing cost.

3. **V-Model Connection**
    - Left side = design → Right side = test

Phase	Corresponding Test
Requirements	Acceptance testing
System design	System testing
Module design	Integration testing
Code	Unit testing (White box)

4. **Static Testing (No Execution)**
    A. **Types**
        - Desk Checking: Dev checks their own code informally.
        - Code Walkthrough: Group review, informal.
        - Code Review: More structured than walkthrough.
        - Fagan Inspection: Formal, roles assigned (Author, Moderator, Scribe, Inspectors).

    B. **Static Analysis Tools**
        - Detect:
            - Unused variables
            - Type mismatches
            - Unreachable code
            - Memory leaks
            - Portability issues
            - Cyclomatic complexity

5. **Structural Testing (Requires Execution)**
    - Types:
        - Unit/Functional Testing
        - Code Coverage Testing
        - Code Complexity Testing

6. **Code Coverage Types**

Type	**Goal**
Statement Coverage	Every line runs at least once
Path Coverage	Every unique execution path
Condition Coverage	Each condition in a decision is true & false
Function Coverage	Each function is called at least once

🧪 Statement Coverage is NOT enough – it might skip logic bugs hidden in condition branches.

7. **Control Structures to Test**
    - Sequential code: Run straight.
    - If-else: Test both branches.
    - Loops:
        - Skip the loop (zero times)
        - Run once
        - Run max times
        - Off-by-one (classic bug!)

8. **Cyclomatic Complexity (CC)**
    - Measures number of independent paths in a program.
    - High CC = higher testing effort.

    Formulas:
    - CC = E - N + 2
      (Edges - Nodes + 2)

    - CC = P + 1
      (P = predicate nodes/decision points)

    Example:
    - 1 if-statement → 2 paths → CC = 2

9. **White-Box Coverage Techniques**
    A. **Statement Coverage**
        - Every line runs once.
        - Weak—might miss logic bugs.

    B. **Decision/Branch Coverage**
        - Each decision (if/switch) has both true and false outcomes tested.

    C. **Condition Coverage**
        - Each Boolean sub-condition gets true and false values.
        - E.g., (a > 1 && b == 0) must test:
            - a > 1, a ≤ 1
            - b == 0, b ≠ 0

    D. **Decision + Condition Coverage**
        - Combines the above for better assurance.

    E. **Multiple Condition Coverage**
        - Tests all combinations of sub-conditions.
        - Example:
            - 3 conditions = 2^3 = 8 combinations

10. **Limitations of Each Type**

Type	**Weakness**
Statement	May skip conditions
Decision	May miss some conditions
Condition	May not test combined outcomes
Decision+Condition	Better, but complex
Multiple Condition	Best, but time-consuming

11. **Example Analysis**

    if (a > 1 && b == 0) { x = x / a; }
    if (a == 2 || x > 1) { x = x + 1; }
    - Paths: ace, abd, etc.
    - Use condition coverage to ensure all branches and sub-conditions are tested.

12. **Challenges in White Box Testing**
    - Developers often don't test their own code deeply.
    - 100% coverage ≠ 0 bugs.
    - May miss real-world use cases.

13. Examples to Know

A. if (x == y && length(z) == 0 && FLAG)
   - 3 conditions → $2^3 = 8$ test cases for full multiple condition coverage.

B. DO I=1 to TABSIZE WHILE (NOTFOUND)
   Test all loop exit paths:
   - Found early
   - Found last
   - Never found
   - Exceeds TABSIZE

14. Summary (Cheat Sheet)

Topic	**Key Idea**
White Box Testing	Based on code structure
Static Testing	No execution, code review
Structural Testing	Execute with code instrumentation
Coverage Types	Statement, Branch, Condition, Path
Cyclomatic Complexity	Measures # of independent paths
Testing Constructs	If-else, loops, sequential code
Tool Support	Static analyzers (e.g., Lint), IDEs (JUnit), debuggers

Black Box testing techniques

1. **What Is Black Box Testing?**
    - Tests external behavior of the software without looking at the code.
    - Based on requirements/specifications.
    - Focus: Inputs, outputs, expected behavior.
    - Code structure is ignored.

    🔄 Example:
    If you're testing a login form:
        - You check if it accepts valid inputs and rejects invalid ones.
        - You don’t care what logic or conditions are used in the backend.

2. **Test Development Process**

Phase	**Purpose**
Test Analysis	Identify test conditions (e.g., "login with wrong password")
Test Design	Create test cases (inputs + expected outputs)
Test Implementation	Organize cases into procedures/scripts, set schedule and priority

3. **Knight Capital Case Study (Why Testing Matters)**
    - 2012: $460 million loss in 45 minutes.
    - Old code was accidentally reactivated.
    - Orders multiplied 1,000x due to test failures.
    - Lessons: Test deployment logic, remove unused code, and test under all conditions.

4. **Key Features of Black Box Testing**
    - Based on functional specs (not implementation).
    - Tests what the system should do.
    - Focuses on:
        - Correct outputs
        - Error handling
        - System performance/behavior

5. **What Does Black Box Testing Detect?**
    - Missing/incorrect functions
    - UI or API interface errors
    - Incorrect logic
    - Boundary value failures
    - Performance issues
    - Initialization and shutdown bugs

6. **Black Box Techniques (Main Ones)**
    A. **Equivalence Partitioning (EP)**
        - Divide input into classes that should behave the same.
        - Test one value from each class.

    Example:

    Input: Age (valid range: 18–60)
        - Valid: {18–60}
        - Invalid: {<18}, {>60}
        - Pick one from each: 17, 30, 61

    B. **Boundary Value Analysis (BVA)**
        - Test values at the edge of input domains.
        - More bugs occur at boundaries (e.g., 0, max, just beyond).

    Example:

    Age (18–60)
        - Test: 17, 18, 59, 60, 61

    C. **Decision Table Testing**
        - Useful when behavior depends on multiple Boolean rules.
        - Create a table with:
            - Conditions (true/false)
            - Actions

    Example:

    Discount logic with 3 inputs → $2^3 = 8$ combinations.

    D. **State Transition Testing**
        - Good for systems that change state based on input.
        - Think ATM:
            - Insert card → Enter PIN → Retry or Lock card
        - Test transitions: valid, invalid, repeat attempts

    E. **Use Case Testing**
        - Derived from real user interactions.
        - Includes:
            - Pre-conditions
            - Normal and alternate flows
            - Post-conditions

    Example:

    Online training: learner enrolls, attends course, submits assignment.

7. **How to Write Tests (Steps)**
    - Break spec into testable features.
    - Select representative values (normal, invalid, edge).
    - Write test cases:
        - Inputs + expected outputs.
    - Organize into test scripts/procedures.

8. **General Testing Types**

✅ Functional (Black Box)

- Based on requirements
- Effective for missing logic bugs

✅ Structural (White Box)

- Based on code paths
- Misses code that doesn't exist (e.g., missing features)

9. **Other Techniques**

A. **Experience-Based Testing**

- Error guessing: Based on tester's past experience.
- Exploratory testing: Test + learn + adapt at the same time.
- Special value testing: Out-of-box, extreme, or "weird" inputs (e.g., emoji in username).

B. **Limit Testing**

- Test system limits: max input size, max concurrent users, etc.

C. **Stress Testing**

- Push system beyond limits (e.g., simulate 10,000 users at once).

D. **Security Testing**

- Try to hack or exploit the system.
- Tools: ZAP, Metasploit, W3af

E. **Usability Testing**

- Get real users to test UI
- Identify pain points and UX issues

F. **Recovery Testing**

- Crash the app or pull power.
- Check if it restarts cleanly and data is safe.

Cheat Sheet Summary

Technique	**Use**
Equivalence Partitioning	Group inputs, pick one from each class
Boundary Value Analysis	Test limits (min, max, just beyond)
Decision Table	Logical rules (true/false) combinations
State Transition	Inputs cause state changes (e.g., ATM)
Use Case Testing	Test real-world business flows

📌 **Final Tips**

- Use EP + BVA together for best coverage.
- Prioritize high-risk or high-usage areas.
- Always test:
    - Normal use
    - Errors
    - Boundaries
    - Unexpected inputs
- Black Box = Fast to start, powerful, but limited by unclear specs.

Unit Testing and JUnit

1. **What Is Unit Testing?**
    - Unit = one class or method (like LoginManager, or calculateTotal()).
    - Tests its functionality, correctness, and accuracy.
    - Usually done by developers (not testers).
    - Uses both white box and black box techniques.

2. **Why JUnit?**
    - Java framework to write and run unit tests.
    - Quick feedback → if something breaks, you'll know immediately.
    - Encourages better code (small, testable units).
    - Free, simple, widely supported (Eclipse, NetBeans, IntelliJ).

3. **JUnit Basics**
    - JUnit test class: Tests another class.
    - @Test annotation: Marks a method as a test case.
    - Each test method:
        - Call the code.
        - Uses assert statements to check the result.

4. **Structure of a Test**

```
@Test
void createAndSetName() {
 Value v1 = new Value();
  v1.setName("Y");
  assertEquals("Y", v1.getName());
}
```

    - Input: v1.setName("Y")
    - Expected Output: "Y"
    - Assertion: assertEquals(expected, actual)

5. **JUnit Test Elements**

Concept	**Meaning**
Test case	One method to test one scenario
Test suite	Group of test classes
Assertion	Statement to verify expected outcome

6. **JUnit Assertions**

Method	**Purpose**
assertTrue(cond)	Pass if condition is true
assertFalse(cond)	Pass if false
assertEquals(exp, act)	Pass if values are equal
assertSame(a, b)	Same object reference
assertNull(obj)	obj must be null
assertArrayEquals(a1, a2)	Arrays equal element-by-element
assertEquals(a, b, delta)	For floats/doubles
fail()	Always fails – used when exception was expected

7. **Exception Testing**

Case 1: Let it throw

```
@Test
void exceptionTest() {
   Exception e = assertThrows(IllegalArgumentException.class,
      () -> myClass.method(null));
            assertEquals("Null value not allowed", e.getMessage());
         }
```

Case 2: Catch it manually

```
@Test
void testWithFail() {
   try {
      method(null);
      fail("Exception should have occurred");
   } catch (IllegalArgumentException e) {
      assertEquals("Null value", e.getMessage());
   }
}
```

8. **Set-Up and Tear-Down**
   - @BeforeEach: Run before every test case.
   - @AfterEach: Run after every test case.
   - @BeforeAll: Run once before all tests (static).
   - @AfterAll: Run once after all tests (static).

```
@BeforeEach void setup() { /* prepare */ }
@AfterEach void cleanup() { /* clean up */ }
```

9. Timed Tests

```
@Test @Timeout(5)
void testLongTask() {
   doWork(); // must finish in 5 seconds
}
```

10. Parameterized Tests

- Run the same test with multiple inputs.

```
@ParameterizedTest
@ValueSource(strings = {"racecar", "radar"})
void testPalindrome(String word) {
   assertTrue(isPalindrome(word));
}
```

11. Test Verdicts

Verdict	Meaning
Pass	All assertions succeeded
Fail	One or more assertions failed
Error	Test crashed (e.g., null pointer)

12. JUnit Best Practices

- Write many small, focused tests (one assert per method).
- Use meaningful names.
- Make tests independent (don't rely on previous ones).
- Avoid logic in test methods (e.g., no loops or conditions).
- Always test edge cases, exceptions, and representative inputs.
- Follow TDD (Test-Driven Development): write the test before writing the code.

13. JUnit Limitations

- Not ideal for:
  - GUI testing
  - Output-dependent code (e.g., System.out.println)
- Encourages pure functions (which is a good thing!)

14. JUnit in Eclipse or NetBeans

- JUnit 5 is integrated.
- Just right-click your test class → Run as → JUnit Test.
- Can add JUnit library manually if needed.

15. Key Files in Sample Project

- BinarySearch.java = Class under test
- BinarySearchTest.java = JUnit test class
- Example tests:

```
@Test
void testSearch1() {
    int[] a = {1, 3, 5, 7};
    assertTrue(BinarySearch.search(a, 3) == 1);
}
```

16. Quick Cheat Sheet

Task	**Code**
Assert equals	assertEquals(expected, actual)
Check null	assertNull(obj)
Compare floats	assertEquals(a, b, 0.01)
Exception test	assertThrows(Exception.class, () -> {...})
Setup before all	@BeforeAll static void init()
Repeat test with data	@ParameterizedTest @ValueSource(...)

Regression testing

1. **What Is Regression Testing?**
    - It's retesting to make sure a change hasn't broken anything.
    - Triggered by:
        - Fixing a bug
        - Adding a feature
        - Refactoring code
        - Changing environments (e.g., OS upgrade)

    🧠 Key Idea:
    "If you touch the code, run the old tests again."

2. **Why Regression Testing Is Critical**

Situation	**Risk**
You fixed bug A	Bug B breaks unexpectedly
You added feature X	Feature Y starts crashing
Something worked yesterday	Fails today after a small change

- Without regression testing, old bugs can come back, or new bugs may appear in stable areas.

3. **How Regression Testing Works**
    - Maintain a test suite:
        1. All past test cases (unit, integration, system)
    - Every time something changes:
        1. Re-run existing tests.
        2. Add new ones if needed.
        3. Compare output with previous results.
        4. Investigate failures.

Relative to the update.

4. **What Triggers Regression Testing?**

Trigger	**Example**
Bug fix	Fixing login error breaks user profile update
New feature	Adding payment causes cart to crash
UI changes	Button name changed, breaks automation script
Config changes	Server upgrade from Tomcat 9 to 10

5. **Core Problems in Regression Testing**
   - **A. Test Suite Maintenance**
     - What needs to be updated when a feature changes?
     - Can we remove obsolete or redundant test cases?
   - **B. Cost**
     - Rerunning all tests = slow + expensive (especially manual).
     - Full suite might not scale with large systems.
   - **C. Fragile Test Suites**
     - Changing a button label shouldn't break 50 test cases.
     - Tests should be modular and resilient.

6. **Test Case Cleanup**

Type	Action
Obsolete	Feature removed or redesigned → delete the test
Redundant	Multiple tests checking the same thing → reduce or merge them

7. **Selecting & Prioritizing Tests**
   - **A. Full Rerun (if fast)**
     - If your tests run in minutes, just run them all.
   - **B. When It's Expensive:**
     You must:
     1. Select which tests to run.
     2. Prioritize based on impact.

8. **Regression Test Selection Methods**
   - **A. Code-Based**
     - Run only tests that cover modified code.
     - Example:
       - If only CartService.java changed, don't test unrelated features like password reset.
   - **B. Specification-Based**
     - Prioritize tests for changed requirements or features.
     - Caution: Even unrelated tests might catch bugs indirectly.

9. **Test Prioritization Strategies**

Strategy	Priority Rule
Round Robin	Least recently run gets priority
Track Record	Tests that found bugs in the past go first
Structural	Run tests that cover untouched code areas

10. Best Practices

- Maintain a stable, up-to-date test suite.
- Run regression tests after every change.
- Use automation to reduce cost/time.
- Add new test cases for every new feature or bug fix.
- Use test management tools to track obsolete or flaky cases.

11. Quick Cheat Sheet

Concept	**Description**
Regression Testing	Ensures changes haven't broken existing behavior
Test Suite	All past tests run after each change
Triggers	Bug fix, feature add, refactor, config change
Obsolete Test	Feature removed, test no longer valid
Redundant Test	Similar to others, can be merged or removed
Code-Based Selection	Run tests touching changed code only
Spec-Based Selection	Based on updated specs
Prioritization	Run high-risk or high-value tests first

Advanced Topics

SECTION 1: Web Testing

What's the Difference?

Website	**Web Application**
Informational (static pages)	Interactive (user input, processing)
Example: Wikipedia	Example: Gmail, Amazon, Moodle

What Is Web Testing?

- Tests websites or web apps for:
  - Bugs
  - Errors
  - UI issues
  - Database or form errors
- Must be done before going live

Why Web Testing Is Challenging

- Many combinations:
  - OS (Windows, Mac, Linux)
  - Browsers (Chrome, Firefox, Safari, etc.)
  - Devices and screen sizes
- New tech = richer apps = harder testing

Types of Web Testing

1. Functionality Testing
2. Usability Testing
3. Interface Testing
4. Compatibility Testing
5. Performance Testing
6. Security Testing

1. **Functionality Testing**
   Tests what the system does:
   - All links (internal, external, email)
   - All forms (validation, inputs, defaults)
   - Cookies (enabled/disabled, deletion, encryption)
   - HTML/CSS compliance (syntax, SEO readability)
   - Database integration (correct queries and results)

   Example test cases:
   - Submit form with invalid email → error shown?
   - Delete operation → confirmation message appears?

2. **Usability Testing**
   Checks how user-friendly the website is:
   - Navigation (menus, buttons, links work and are visible)
   - Content (spelling, readability, colors, image alignment)
   - Alignment and consistency (fonts, tooltips, layout)

   Example:
   - Home button present on all pages?
   - Can you navigate with a keyboard?

3. **Interface Testing**
   Tests interaction between:
   - Web server ↔ App server
   - App ↔ Database
   - Page ↔ Page

4. **Compatibility Testing**
   Ensures the app works across:
   - Browsers (Chrome, Firefox, Edge, etc.)
   - OS platforms (Windows, macOS, Linux)
   - Devices

   Example:
   - Does the app render properly on Firefox and Safari?

5. **Performance Testing**
   Tests how well the app performs:
   - Response time under different speeds
   - Load testing: Can it handle 1,000+ users?
   - Stress testing: What happens if overloaded?

6. **Security Testing**
   Covers:
   - Data encryption (e.g., password, card info)
   - Authentication (roles and rights)
   - Session timeout
   - HTTPS enforcement

   Example:
   - Are passwords encrypted
   - Can users access admin pages without permission?

Test Automation

- Saves time by automating repetitive test cases
- Tool: Selenium (popular open-source tool)
  - Supports Java, Python, C#, etc.
  - Tests web apps across platforms and browsers

SECTION 2: Mobile Testing

Mobile App vs. Mobile Device Testing

Type	Tests
Mobile Testing	Hardware: screen, memory, Wi-Fi, Bluetooth
Mobile App Testing	Software/app features, usability, performance

Types of Mobile Applications

1. Native App – Installed, fastest (e.g., WhatsApp)
2. Web App – Accessed via browser
3. Hybrid App – Installed, but runs in browser shell (e.g., Instagram)

Why Is Mobile Testing Hard?

- Many device types (screen sizes, hardware)
- Many OS versions (Android, iOS)
- Network variability (3G, 4G, Wi-Fi)
- Input types (touch, voice, gestures)

Types of Mobile App Testing

1. **Functional Testing**
   Tests app behavior:
   - Mandatory fields work
   - App reacts properly to phone calls
   - Works across networks (2G/3G/4G/5G)
   - Payment gateway works
   - Navigation is smooth

2. **Performance Testing**
   - Response time under load
   - App behavior with poor networks
   - Battery usage and memory leaks
   - Longevity under stress

3. **Security Testing**
   - Brute force protection
   - Password protection and session timeout
   - SQL injection prevention
   - Certificate pinning
   - Prevent malicious injections and file caching attacks

4. **Usability Testing**
    - Buttons sized for touch
    - Icons are meaningful
    - Easy navigation
    - Clear error messages
    - Works in different orientations/languages

5. **Compatibility Testing**
    - Test on various devices
    - UI adjusts to screen size
    - App resumes after call interruption
    - Readable text and responsive layout

6. **Recoverability Testing**
    - App behavior after crash/power loss
    - How it resumes after network disconnection

7. **Other Tests**
    - Installation/uninstallation
    - Charger effects, low battery
    - Interrupted transaction
    - Keyboard behavior during errors

Tools for Mobile App Testing

Tool	**Notes**
Appium	Open-source, supports iOS + Android
Kobiton	AI-powered, manual + automated
Ranorex	Mobile + web automation
Eggplant	UI + performance testing
Selendroid	Android-only
iOSDriver	iOS-only

Manual vs Automated Testing

- Manual: Better for usability, one-time edge cases
- Automated: Best for regressions and repetitive tasks

📌 Final Summary

Topic	Key Idea
Web Testing	Functional, security, performance testing of web apps
Mobile Testing	Covers devices + apps
App Testing Types	Functional, usability, performance, security, etc.
Automation Tools	Selenium (web), Appium, Ranorex, Kobiton (mobile)

Code Coverage

1. **What Is Code Coverage?**
    - Definition: Measures the % of your codebase executed during testing.
    - Report shows how much of:
        - Statements
        - Branches
        - Functions
        - Lines

          were covered by your test suite.

    Example:
    If you have 100 lines of code, and your tests run 85 of them → 85% code coverage.

2. **Why Use Code Coverage?**
    - Find untested code.
    - Shows which parts need more tests.
    - Helps teams prioritize test writing.
    - Some teams block deployments if coverage drops below a threshold (e.g., 80%).

3. **Misconceptions**
    - High code coverage ≠ high test quality.
    - It only tells you code was executed, not that it was correctly tested.
    - Can give a false sense of security.
    - Doesn't replace code reviews or good test design.

4. **How to Use It Effectively**
    - Start by running a coverage report.
    - Check for:
        - Missed branches/paths
        - Untested conditions
    - Use it to reflect:
        - Did I forget to test that case?
        - Was it too complex to test?

5. **Benefits**
    - Objective, repeatable metric.
    - Easy to automate (e.g., in CI/CD).
    - Helps enforce testing discipline.
    - Encourages better software design (more modular/testable code).

6. **Best Practices**
    From Google's Code Coverage Guidelines:
    A. **Use Code Coverage As a Tool, Not a Goal**
        - Don't chase 100% coverage just for the number.
        - Focus on what's not covered.
        - Ask: Is this gap risky?

B. Combine Coverage With Other Metrics

- Use with:
  - Test quality reviews
  - Mutation testing (checks test effectiveness)
  - Manual test insights

C. No Universal Ideal Percentage

Coverage	Meaning
<60%	Poor
60–75%	Acceptable
75–90%	Good
>90%	Excellent, but diminishing returns

- Adjust based on:
  - Business risk
  - Code stability
  - Complexity

7. Gating Deployments

- Some teams block deployments if coverage drops.
- Options:
  - Gate on new/changed code only
  - Gate on overall delta
  - Exclude untestable code (e.g., logs, throwaway branches)
- Be careful: don’t treat it as a checkbox.

8. Common Pitfalls

Mistake	Why It’s Bad
Writing useless tests	Wastes time and maintenance effort
Copy-paste tests	Doesn’t validate behavior
Testing just to hit %	Doesn’t improve quality
Ignoring review for "covered" code	Coverage ≠ correctness

9. Types of Coverage

Type	What It Measures
Line/Statement	Were all lines executed?
Branch	Were all if/else branches taken?
Function	Were all functions called?
Path	Were all paths through the code taken? (more advanced)

10. Code Coverage in Practice

Good example:

```
public int divide(int a, int b) {
    if (b == 0) throw new IllegalArgumentException();
    return a / b;
}
```

Tests needed:

- divide(10, 2) → normal
- divide(10, 0) → exception

Both line and branch coverage are needed to hit the if and return.

11. Incremental Improvement

- Start small: check what's not covered.
- Adopt the boy-scout rule: improve coverage a little every time.
- Focus on:
    - Frequently changing code
    - High-risk modules

📌 12. Final Summary Cheat Sheet

Concept	Key Idea
Code Coverage	Measures what % of code is tested
Coverage ≠ Quality	It shows execution, not validation
Best Use	Identify gaps, not hit a %
Ideal %	Depends on context (60–90% range)
Google's Advice	Pragmatic use, focus on untested code, not 100%

Testing Metrics and Tools

1. **What is a Metric?**
    - A metric is a standard of measurement.
    - In testing, metrics help measure software quality and process efficiency.
    - Example: Total number of defects, % of tests passed.

2. **Why Use Testing Metrics?**
    - "You can't improve what you can't measure."
    - Metrics help:
        - Identify problems early.
        - Improve the testing process.
        - Justify decisions with data.
        - Track readiness, quality, and progress.

3. **Types of Testing Metrics**
    A. Process Metrics
        - Measure SDLC process efficiency.
        - Example: Defects per phase, review effectiveness.

    B. Product Metrics
        - Measure software quality.
        - Example: Defect density, requirement coverage.

    C. Project Metrics
        - Track team or tool performance.
        - Example: Test cases executed per day.

4. **Manual Testing Metrics**
    - Base Metrics (Raw Data)
        - of test cases written
        - passed/failed/blocked test cases
        - of defects found

    - Calculated Metrics (Derived from Base)
        - % Test Coverage
        - % Test Completion
        - Defect detection efficiency

5. **Key Calculated Metrics**
    Test Coverage Metrics
        A. Test Execution Coverage
            - Formula:
              (Executed tests / Total planned tests) × 100%

B. Requirement Coverage
   - Formula:
     (Requirements covered by tests / Total requirements) × 100%

Test Effectiveness

- Measures the bug-finding power of your tests.
- Formula:
  (Defects found during testing / Total defects found pre + post release) × 100%
- Example: If 80 defects were caught before release and 20 after:
  - Effectiveness = (80 / 100) × 100% = 80%

Test Effort Metrics

A. Test runs per time
   - Example: 30 tests/day.

B. Defects per hour
   - Formula:
     Defects found / Testing hours
   - Example: 12 bugs in 6 hours = 2 bugs/hour

C. Bugs per test
   - Formula:
     # of bugs / # of tests
   - Example: 6 bugs from 30 tests = 0.2 bugs/test

D. Avg. time to test a fix
   - Formula:
     Total time to verify fixes / # of bugs
   - Example: 4 hrs / 3 bugs = 1.33 hrs/bug

Test Quality & Tracking Metrics

Metric	**Formula**	**Use**
% Passed	(Passed / Executed) × 100%	Test quality
% Failed	(Failed / Executed) × 100%	Test effectiveness
% Critical Defects	(Critical / Total defects) × 100%	Severity focus
% Fixed Defects	(Fixed / Reported defects) × 100%	Efficiency check

Test Efficiency Metric

- Repair Time/Defect
- Formula:
  Total fix time / # of defects
- Measures how efficiently bugs are fixed.

Other Metrics

- Cumulative test time
- Cost of testing
- Defects found post-release (shows missed bugs)

---

6. Code Coverage Metrics
   A. Segment Coverage
      - Goal: 85%
      - Tracks % of code blocks (segments) tested.

   B. Call-Pair Coverage
      - Goal: 100%
      - Measures coverage of module interactions (calls between methods).

   C. Requirements Coverage
      - % of requirements tested.

7. Quality Metrics
   A. Defect Removal %
      - Formula:
        (Bugs fixed before release / Total known bugs) × 100%

   B. Defect Detection Efficiency
      - Formula:
        (Defects found internally / (Internal + Customer-found)) × 100%

8. Testing Tools Overview
   A. Why Use Tools?
      - Reduce human error
      - Save time on repeat tasks
      - Automate hard/impossible manual testing
      - Increase reliability, coverage, and efficiency

9. Categories of Testing Tools

Management Tools

- Test Management (ALM): Track test cases, traceability, results.
- Defect Management: Log bugs (e.g. Jira).
- Requirements Tools: Trace requirements to tests.
- CI Tools: Jenkins, GitHub Actions.

Static Testing Tools (No Code Execution)

- Review Tools: Manage peer reviews
- Static Analysis:
  - SpotBugs (Java bug patterns)
  - SourceMonitor (code metrics)
  - JDepend (Java package design quality)
- Modeling Tools: StarUML (UML diagrams)

Test Design Tools

- Generate inputs, expected results from:
  - Requirements
  - State diagrams
  - GUIs
- Includes:
  - Model-Based Testing
  - TDD, ATDD, BDD (e.g., Given-When-Then)

Test Execution & Logging Tools

Tool	**Purpose**
Unit test frameworks	JUnit, NUnit
Test harnesses	Use drivers & stubs for isolated component testing
Test comparators	Compare actual vs. expected outputs
Coverage tools	Track test coverage (statements, branches)

Performance & Monitoring Tools

- Simulate large data loads.
- Monitor system behavior (memory, CPU, users).
- Detect issues like:
  - Memory leaks
  - Null pointer crashes
  - Load stress

Specialized Tools

- Usability Testing
- Localization
- Security
- Data migration/quality

10. Tool Considerations (Risks & Benefits)

✅ Benefits:

- Consistency
- Speed
- Objectivity (no bias is.)
- Good for regression tests

❌ Risks:

- Unrealistic expectations
- High setup & maintenance cost
- Over-reliance (bad tests still mean bad results)
- Some tools affect the outcome (probe effect)

11. Special Testing Approaches

A. Data-driven Testing

- Test logic stays the same, but data changes.
- Great for regression and form testing.

B. Keyword-driven Testing

- Keywords (like “Login”, “Click”) + test data in spreadsheet.
- Tester writes tests using natural actions.

12. Interview & Exam Concepts

Topic	Key Point
QA vs. QC	QA = process, QC = product
Verification	“Are we building it right?”
Validation	“Are we building the right thing?”
Branch Coverage	All possible branches taken
Static Analysis	Find bugs without running code
Defect Cost	The later you find it, the more it costs

Summary Sheet (Quick Recall)

Metric Type	**Example**
Base Metric	# test cases, # bugs
Calculated	% coverage, effectiveness
Process Metric	Defects per phase
Product Metric	Defect density
Project Metric	Tests/day, time/bug
Code Coverage	% of code executed
Quality	% fixed, % failed, % passed
Tools	JUnit, Jenkins, Selenium, SpotBugs

Metric Category	Metric Name	Formula	What It Measures
Coverage Metrics	Test Execution Coverage	(Executed Test Cases / Total Test Cases) × 100%	% of tests actually run from the plan
	Requirement Coverage	(Requirements Covered / Total Requirements) × 100%	% of requirements tested
Effectiveness Metrics	Test Effectiveness	(Defects Found in Testing / Total Defects) × 100%	% of bugs found before release
Effort Metrics	Tests per Time	Number of Tests / Time Period	Testing speed (e.g., 30 tests/day)
	Defects per Hour	Defects Found / Hours of Testing	Bug discovery rate
	Bugs per Test	Defects Found / Total Tests	Test quality
	Time to Test a Bug Fix	Total Time to Test Fixes / Number of Bugs	Average time spent per bug fix
Tracking & Quality	% Passed Tests	(Passed Tests / Executed Tests) × 100%	Quality of executed test set
	% Failed Tests	(Failed Tests / Executed Tests) × 100%	Problem frequency in tested code
	% Critical Defects	(Critical Defects / Total Defects) × 100%	Defect severity level
	% Fixed Defects	(Fixed Defects / Reported Defects) × 100%	Team's effectiveness at fixing bugs
Efficiency Metrics	Defect Repair Efficiency	Total Fix Time / Total Defects	Average time per fix
Code Coverage	Segment Coverage	(Covered Segments / Total Segments) × 100%	% of code segments hit during test
	Call-Pair Coverage	(Call Pairs Hit / Total Call Pairs) × 100%	% of module interactions exercised
Quality Metrics	Defect Removal Efficiency (DRE)	(Fixed Defects Pre-Release / Total Known Defects) × 100%	% of bugs fixed before release
	Defect Detection Efficiency (DDE)	(Internal Defects / (Internal + Customer Defects)) × 100%	% of defects found by testers vs. customers