JUnit annotations: In JUnit 4, **@Before** runs before every test method to set up a fresh test environment, such as creating new objects, resetting fields, or loading small fixtures; its JUnit 5 equivalent is *@BeforeEach*. **@After** runs after every test to clean up by closing files, deleting temporary data, or resetting external state, with *@AfterEach* as its JUnit 5 equivalent. **@BeforeClass** runs once before any test methods in the class to perform expensive or shared setup tasks like starting an embedded database or initializing caches; it must be static in JUnit 4, and its JUnit 5 equivalent is *@BeforeAll* (static unless using a per-class test instance). **@AfterClass** runs once after all tests complete to tear down shared resources such as servers or global connections; it must also be static, and its JUnit 5 equivalent is *@AfterAll*. Use *@Before*/*@After* for per-test setup and cleanup (temporary files, resetting state) and *@BeforeClass*/*@AfterClass* for shared, resource-heavy setup and teardown. Avoid common mistakes such as forgetting to make @BeforeClass/@AfterClass methods static in JUnit 4, performing heavy setup in @Before when it could go in @BeforeClass, sharing mutable state across tests without resetting it, and mixing JUnit 4 and JUnit 5 annotations.
Parameterized testing in JUnit allows a single test method to run multiple times with different input values, reusing the same assertion logic for many cases instead of writing separate tests. In JUnit 5, this is done using ***@ParameterizedTest*** together with a *source annotation* that automatically supplies test inputs. It helps eliminate duplicate test methods by separating test logic from varying data and makes it easy to cover *boundary values* and *equivalence classes* thoroughly. The core annotations include ***@ParameterizedTest*** (marks the test for repeated runs), ***@ValueSource*** (provides a list of literal values such as numbers or strings), ***@CsvSource*** (provides multiple arguments per test, e.g., input–expected pairs), **@MethodSource** (links to a factory method returning a Stream or Iterable of complex arguments), and ***@CsvFileSource*** (reads arguments from an external CSV file on the classpath). Examples include checking even numbers using @ValueSource, verifying string uppercasing with input–expected pairs via @CsvSource, and testing addition using data from a @MethodSource. Parameterized tests are ideal for *testing utility functions, validators, and small logic-heavy methods*, especially where the behavior stays constant but inputs and expected outputs vary.

Review: *Static testing* inspects artifacts (requirements, specs, designs, code, guides, test cases) without executing code, complementing dynamic testing rather than replacing it. IEEE recognizes management reviews, technical reviews, inspections, walkthroughs, and audits, each focused on fitness to purpose, standards conformance, and early defect detection across lifecycle artifacts.
Management review: management-led evaluation of plans, schedules, progress, and effectiveness of approaches to ensure fitness for purpose. **Technical review**: peer evaluation of a product's suitability and standard/specification conformance to find discrepancies before later stages. **Inspection**: formal, preparation-heavy peer examination of requirements/design/code by non-authors to detect faults and standard violations, with materials distributed beforehand. **Walkthrough**: formal author-led presentation of the artifact to a small group where reviewers listen and question to understand and spot issues. **Audit**: independent examination of product/process for compliance with specs, standards, contracts, or other criteria.
Informal Review: Desk checks, casual team meetings, or pair programming reviews. Aim: quick error discovery and continuous quality improvement. ***Pair programming*** encourages continuous review as a work product (design or code) is created. The benefit is immediate discovery of errors and better work product quality as a consequence.
Formal technical reviews (FTRs): *Objectives* include uncovering errors in function/logic/implementation, verifying requirements satisfaction, standards conformance, development uniformity, and project manageability. *Typical structure*: 3–5 participants, ≤2 hours prep each, ≤2 hours meeting, focused on a concrete work product (e.g., a design slice or a component's code). Roles: producer (author), review leader (readiness and distribution), reviewers (prepare notes), recorder (captures issues) for structured and accountable outcomes. ***Producer***—the individual who has developed the work product –informs the project leader that the work product is complete and that a review is required. ***Review leader***—evaluates the product for readiness, generates copies of product materials, and distributes them to two or three *reviewers* for advance preparation. ***Reviewer(s)***—expected to spend between one and two hours reviewing the product, making notes, and otherwise becoming familiar with the work. ***Recorder***—reviewer who records (in writing) all important issues raised during the review. ***Conducting Review*:** Review the product, not the producer; set and maintain an agenda; limit debate and rebuttal; note problems without solving all in-meeting; keep written notes; require prep; use tailored checklists; allocate time/resources; train reviewers; and review the review process itself.
Code Review Checklist: **Data declaration**: correct types/lengths/storage classes and consistent initialization where provided (e.g., string vs char array). **Computation**: mixed types or sizes in arithmetic, potential overflow/underflow, and numerical stability concerns. **Control flow**: loop termination, switch defaults/breaks, unreachable or dead code indicators. **Comparison**: off-by-one or operator mistakes and careful floating-point comparisons using an EPSILON pattern when needed. **Complexity and risk**: use cyclomatic complexity bands (1–10 low risk, 11–20 moderate, 21–50 high, >50 very high/near untestable) to flag modules needing simplification or added tests.

Code smells to spot quickly:
- **Duplicated code** risks inconsistent fixes and latent defects when one copy changes without others being updated.
- **Long method** is harder to understand, review, test, and reuse, often hiding multiple responsibilities that should be split.
- **Large class** accumulates too many responsibilities, reducing cohesion and making changes risky and hard to localize.
- **Long parameter list** hurts readability and often signals missing objects or configuration encapsulation.
- **Temporary field** indicates attributes used only in special cases that should be localized or moved to narrow contexts.

Refactoring strategy (how to act on smells):
- Reduce duplication by extracting common behavior into a single method/class and routing all callers to it to ensure one change point and consistent behavior.
- Shorten long methods with Extract Method and clarify intent, enabling naming to communicate behavior and making logic more testable and reviewable.
- Break large classes into cohesive ones based on responsibilities and data affinity to raise cohesion and reduce the scope of changes per class.
- Replace long parameter lists with value objects or builder/config objects to improve readability and reduce call-site errors in argument order/types.
- Localize temporary fields by moving them into helper methods or small helper classes so attributes reflect true object invariants and typical usage.

OO Design: Improve design quality (not just code) using abstraction, encapsulation, inheritance, polymorphism, and quality metrics. ***Abstraction*****:** Focus on essential features; ignore irrelevant details. **Data abstraction:** defines structure (attributes). *Procedural abstraction:* defines operations (methods). **Information Hiding**: Hide internal details (algorithms, data structures, resource policies). Interact only through controlled interfaces (public methods). ***Benefits:*** Reduces side effects. Localizes design changes. Promotes encapsulation. Avoids global data dependencies. Increases software robustness and maintainability. Public Methods(+), Private attributes(-), Protected Data Members(#). **Inheritance**: Mechanism for code reuse and hierarchy creation. Subclasses inherit attributes and behaviors from parent classes. Supports extension and specialization of existing components. **Polymorphism**: Means “many forms.” A polymorphic reference can refer to different object types at runtime. The method executed depends on the actual object type, not the reference type. Achieved via inheritance or interfaces. Benefits: Leads to flexible, elegant, and robust software. Simplifies code as in same method behaves differently for different objects.

Modularity is the process of decomposing a software system into separate, interchangeable components that work together through well-defined interfaces. Each module should perform a specific, cohesive function and interact with other modules only when necessary. The goal of modularity is **functional independence**, meaning each module has a single, well-defined purpose (high cohesion) and minimal reliance on others (low coupling).
Coupling is the degree to which a module is “connected” to other modules in the system. It measures how strongly one component is connected to another. Ideally, software systems should aim for *low coupling*, where modules interact through simple, well-defined data interfaces rather than relying on each other’s internal workings. *High coupling*, on the other hand, means modules are heavily dependent on each other, which increases the likelihood of system-wide failures when changes are made to one part of the code. There are several types of coupling, ranked from worst to best:
Content Coupling: One module is directly connected to the inner working of another module. Meaning where one module directly accesses another’s internal logic or data. Eliminate this by typically hiding the data from outside world (make it private), and then we develop a public method so that it can modify the data member as needed (hiding the implementation logic). Fix by making internals private and exposing narrow methods so callers never touch inside parts.
Common Coupling: Two modules share a global data item. Eliminate this by passing the object to separate object types, and they keep track of their own instances. Fix by deleting globals and flowing data through parameters or dedicated services, with each class owning its own state.
Control Coupling: Data in one module is used to determine the order of execution of another module. Occurs when one module controls another’s behavior by passing control flags or signals. Eliminate this by making independent new methods for each scenario instead of passing parameters such as signal/control flag. Fix by splitting into explicit methods or using polymorphism/strategy so the callee decides without a flag.
Stamp Coupling: One module passes non-global data structures/objects to another module. fix by passing just the needed fields or a smaller DTO/interface.
Data Coupling: One module passes elementary data types, such as int, float, and char, as parameters to the other; this is the most desirable form. Keep parameters minimal and cohesive to avoid drifting into stamp/control coupling.

Cohesion: The degree to which all elements of a component are directed towards a single task and all elements directed towards that task are contained in a single component. In a well-designed module, all its functions and data work together to achieve a single, clearly defined purpose — this is known as **high cohesion**. Conversely, **low cohesion** occurs when a module performs multiple unrelated tasks, which makes the code confusing, difficult to maintain, and error-prone. **Fixing** it by: Splitting unrelated tasks into separate functions/classes so each unit does one meaningful job. Keep operations that work on the same data structure together, and move unrelated operations to the owners of their data. If the output of one step feeds the next, keep the pipeline together but split each step into its own small function within that pipeline module. Ensure every element contributes to one well-defined function, which is the ideal end state after refactoring.
A system with **high cohesion** and **low coupling** is considered well-designed because each component focuses on a single task and interacts minimally with others. Such systems are easier to test, maintain, and extend, leading to higher software quality. Conversely, **high coupling** and **low cohesion** result in fragile systems where a small change can cause cascading errors across multiple modules. This combination makes testing and debugging difficult, reduces reusability, and increases maintenance costs.

***C-K Metrics for OO Design*:** Chidamber and Kemerer (C-K) metrics suite 6 desirable attributes:
Weighted Methods per Class (WMC) measures the total number of methods in a class, weighted by their cyclomatic complexity. A higher WMC indicates a more complex class that is harder to test and maintain. Keeping WMC low helps maintain simplicity. **Depth of Inheritance Tree (DIT)** indicates the number of levels of inheritance from the root class. As DIT increases, a class inherits more behavior, making it harder to predict. A DIT value greater than 5 is generally considered problematic. **Number of Children (NOC)** counts the direct subclasses of a class. While inheritance promotes reuse, a very high NOC (greater than ten) may suggest excessive subclassing or misuse of inheritance, which complicates maintenance. **Response for a Class (RFC)** measures the number of methods that can be executed in response to a message received by an object. A high RFC indicates a complex class with higher coupling and testing difficulty. Typically, RFC values above 50 are discouraged. **Coupling Between Object Classes (CBO)** measures how many other classes a particular class is connected to. High CBO reduces modularity and increases the chance that changes in one class will affect others. **Lack of Cohesion in Methods (LCOM)** assesses how closely related the methods in a class are to each other. A high LCOM value means that methods do not share common data, indicating poor cohesion and a potential need for class restructuring.

Distributed software: A distributed information system has requesters (clients) and service providers that cooperate through layers such as presentation, application logic, and resource/data management, with different tier allocations producing 1-tier, 2-tier, 3-tier, or n-tier deployments. Web application categories highlighted are conventional client-server, service-oriented architectures (SOA), and cloud computing, reflecting how functionality is partitioned and delivered over the network. A typical 3-tier web application has server scripts handling requests/responses and data management modules persisting and retrieving information behind the application logic layer.

3-tier client-server	Presentation, application logic, and data/resource layers split across requester and provider tiers	Lets you target tests per tier and validate contracts between layers as part of gray-box plans .
Service-oriented (SOA)	Application behavior composed from services with networked boundaries and interoperability concerns	Encourages contract, schema, and integration tests in addition to UI flows to catch cross-service faults .
Cloud architectures	Hosted services/platforms offering elastic, distributed deployment of app tiers	Testing must account for environment variability and scaling, requiring automation and environment-aware assertions .

Gray-box testing blends black-box and white-box by using end-user outcomes together with system knowledge like architecture, environment, and component interoperability to design stronger tests than behavior-only or code-only views alone. Factors considered include expected user behavior, internal technical details, runtime environment constraints, and how components integrate across services and tiers to reveal distributed interaction faults.
Why gray-box suits web apps: Web apps are loosely coupled and distributed, so tests must combine UI-level scenarios with knowledge of service endpoints, session management, and data flows across layers to observe and isolate faults effectively. Data for one feature may be scattered across devices and services, so only using black-box UI checks can miss integration and consistency issues detectable with internal knowledge of where and how data is processed and stored. Heterogeneous devices, browsers, and configurations along with dynamic adaptation to local settings demand tests that both observe user outcomes and intentionally target architecture-specific risks and cross-component contracts.

Challenges in Web Application Testing: Web applications are loosely coupled and components are distributed. Data pertaining to one application is scattered among different devices. Heterogeneity of devices, configurations and software systems. Dynamic adaptation to local settings.
Selenium is a toolkit for automating web application testing across multiple browsers and programming languages, allowing assertions to compare expected and actual web content similar to JUnit. It originally included Selenium RC (now obsolete) for remote and parallel multi-browser testing, but modern use centers on Selenium IDE for quick recording and WebDriver for programmable, robust automation. **Selenium IDE** records user interactions and adds verifications/assertions to check element presence, text, titles, and values—useful for quick or smoke tests. Assertions stop execution on failure, while verifications continue to report multiple issues. Common commands include verifyTextPresent, verifyTitle, verifyElementPresent, and verifyValue. However, IDE is limited by browser support, lack of loops/conditionals, reliance on Selenese scripts, and poor handling of dynamic content, leading to the adoption of WebDriver for complex suites. **WebDriver** enables programmatic element access and manipulation through locators. It supports key actions such as getAttribute, sendKeys, clear, submit, getLocation, and getSize, enabling rich interactions and validation of dynamic web states. While RC offered early remote and parallel testing, modern WebDriver frameworks and cloud grids achieve the same goals with more maintainable, code-driven approaches.

4. Data Path Testing : Draw the control flow graph for the following SequentialSearch method. Then design test cases to test the SequentialSearch algorithm implementation.

```
int SequentialSearch(int[] a, int searchItem)        — (1)
{
int itemIndex = -1; Boolean found= false;           — (2)
for(location = 0 ; location < a.length ; location++)
             (3)           (4)                 (5)
    {
        if(a[location] = = searchItem)  (6)
        {
          found = true;
          itemIndex = location;         } (7)
          break;
        }
    }
     return itemIndex;    — (8)
}
```

a) Draw Dataflow graph and then identify D-U pairs and D-U paths for following variables in the above code segment.

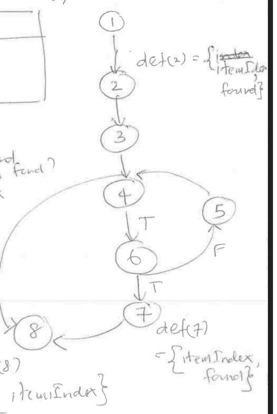
itemIndex
found

D-u path
itemIndex ⇒ 2→ 3→ 4→8 ⇒ 1→2→3→4→8
⇒ 7→ 8 ⇒ 1→ 2→3→4→6→ 7→8

	Du Pair	
~~index~~ itemIndex	(2, 8)	(7, 8)
found	(2,)	(7,)

b) Design test cases based on D-U paths identified.

	Inputs	expected output (index, find)
T1	100 , empty array	-1 , false
T2	100, {100, 1, 200}	0, true



1
2
def(2) = {~~index~~ itemIndex, found}
3
4
5
T
F
6
T
7
def(7) = {itemIndex, found}
8
use(8) {itemIndex}

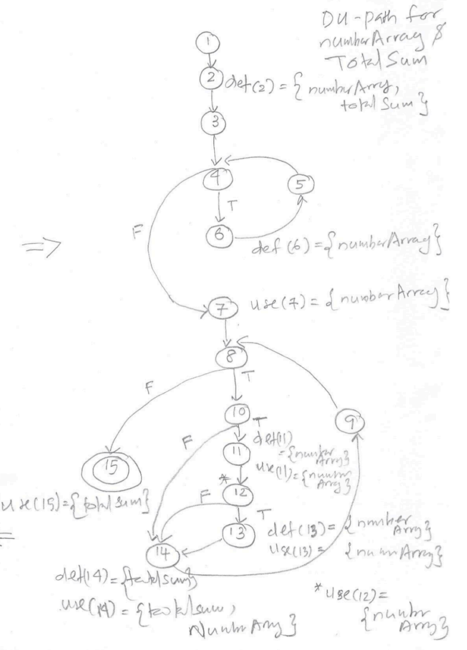
```
public static boolean IsValidMod10Number(String  number) — (1)
  {
          int [] numberArray = new int[number.length()];  }
          boolean checkBit = false;                        } (2)
          int sumTotal =0;  (3)       (4)          (5)
                    for(int i=0; i < number.length() ; i++)
                    {
                              numberArray[i] = (int) number.charAt(i);  (6)
                    }
                    for(int index = numberArray.length -1 ; index >= 0 ; index--)
                    {          (7)                         (8)           (9)
                              if(checkBit)  → (10)
                              {
                              numberArray[index] *=2;  — (11)
                                        if(numberArray[index] > 9) — (12)
                                        {
                                        numberArray[index] -= 9;  — (13)
                                        }
                              }
                    sumTotal += numberArray[index];  }
                    checkBit = !checkBit;            } (14)
                    }
          return sumTotal % 10 ==0; (15)
  }
```

Du-Pairs

Variable	Du-pair
totalsum	(2, 14) (2, 15) (14, 14) (14, 15)
number Array	(2, -), (6,7), (6, 11) (11, 12), (11, 13), ([6, 11, 13]$^{+}$, 14) (or)

⇐

⇒



Du- paths

totalsum:

pair(2,14) P1: 2→ 3→ 4→ 7→ 8→ 15
~~*~~ 7→ 8→ 10→ 11→ 14→ 9→ 8→ 15

pair(2,15) P2(1): 2→ 3→ 4→ [6→ 5→ 4]* → 7→ 8 → [10→ 11→ 12→ 13→ 14→ 9→ 8]→ 15

P2(2): 2→ 3→ 4→ [6→ 5→ 4]* → 7→ 8 → [10→ 11→ 12→ 14 → 9→ 8]* → 15

P2(3):

pair(14,14): No path

pair(1~~4~~,15) (P(3)): 14→15

P3 is part of P2

Compromized paths

P1: Empty string

P2(1): String with one digit

P2(2): string with at least two digits and ~~one~~ ~~on~~ the second digit >4

P2(3): String with atleast two digits and all digits are less than < 5

Test cases

Test case #	Test data (input)	expected output (totalsum)		
1 (P1)	number =""	0		
2 ($P_{2.1}$)	number="6"	6		
3 ($P_{2.2}$)	number="1321"	10	⟵	(1+4+3+2)
4 ($P_{2.3}$)	number="1567"	17	⟵	(7+3+5+2)