# MANUAL TESTING

## 1 Terminologies used in software industry

* 1. REQ: requirement
  2. Defect: it’s a deviation in the with respect to requirement
  3. Bug: it’s a informal name of a defect. Whenever a defect is found by a test engineer it should be released to the developers. Once the developers accept the defect it is termed as defect.
  4. Error: mistake in the source code is known as error.
  5. Issue: problem faced by the customer or the end users.
  6. Failure: multiple issues will lead to the failure

## 2 QA vs QC

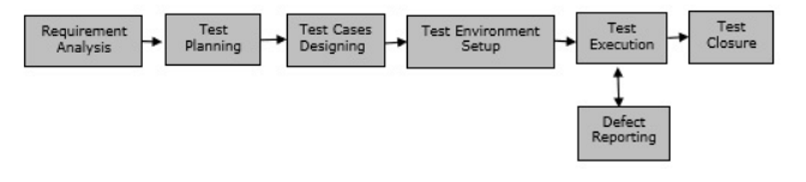
* 1. QA : is quality assurance. It is a process oriented
  2. QC: is a quality control. It is a product oriented.

|  |  |
| --- | --- |
| QA | QC |
| It is a procedure that focus on providing assurance that quality requested will be achieved. | It is a procedure that focus on fulfilling the quality requested. |
| QA aims to prevent the defect | QC aims to identify the defect. |
| It is a method to manage quality verification | It is a method to verify the quality validation |
| It does not involve executing the programs. | It always involves executing the programs. |
| It is a preventive technique. | It is a corrective technique. |

## 3 Agile Terms

* 1. **Agile Manifesto:** A document outlining the principles and values that guide Agile practices
  2. **Scrum:** A popular Agile framework that uses a team-based, iterative approach to development.
  3. **Sprint:** A time-boxed period (typically 1-4 weeks) during which a team works on a specific set of tasks.
  4. **Backlog:** A prioritized list of work items, features, or requirements.
  5. **User Story:** A short description of a feature or requirement from the user's perspective.
  6. **Product Owner:** The individual responsible for prioritizing the backlog and ensuring it reflects stakeholder needs.
  7. **Scrum Master:** A facilitator who helps the team apply Agile practices and removes impediments.
  8. **Definition of Done:** A set of criteria that must be met for a piece of work to be considered complete.
  9. **Burndown Chart:** A visual representation of the remaining work in a sprint or release.
  10. **Continuous Integration:**A practice of regularly merging code changes into a central repository.
  11. **Bottleneck:** Bottlenecks are issues that slow down or stop the development process of a project.
  12. **Kanban board:** The Kanban board is one of the primary tools of the Kanban method. It’s a board, divided into columns, with each column typically representing a stage in a workflow. The most basic workflow setup for a kanban board is “To Do,” “In Progress,” and “Done”.
  13. **Sprint Retrospective:** The Sprint Retrospective is a special Scrum meeting where the Scrum team reviews their performance during the Sprint.
  14. **RTM:** Requirement Traceability matrix

## 4 STLC





## 5 Software development models

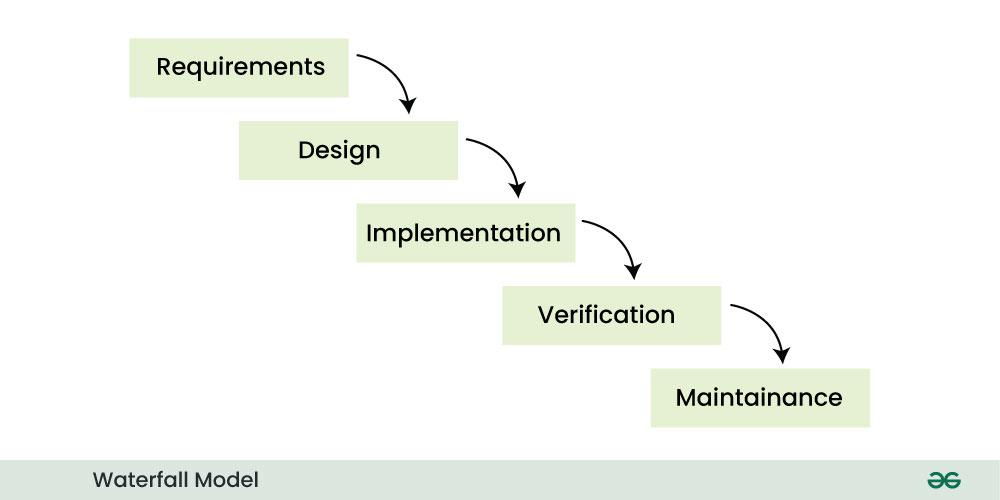
### **5.1 V model:** is an SDLC model, it is also called Verification and Validation Model. V-Model is widely used in the **Software Development Process**, and it is considered a disciplined model. In V-Model, the execution of each process is sequential, that is, the new phase starts only after the previous phase ends.



* It is based on the association of testing phase with each development phase that is in V-Model with each development phase, its testing phase is also associated in a V-shape in other words both [**Software Development**](https://www.geeksforgeeks.org/software-development/)and testing activities take place at the same time.
* So in this model, Verification Phase will be on one side, Validation Phase will be on the other side that is both the activities run simultaneously and both of them are connected to each other in V-Shape through Coding Phase, hence it is called V-Model.
* **V-Design:** In V-Design the left side represents the development activity, the right side represents the testing activity.

### **5.2 Waterfall model:** is a famous and good version of [SDLC(System Development Life Cycle)](https://www.geeksforgeeks.org/software-development/)for software engineering. The waterfall model is a linear and sequential model, which means that a development phase cannot begin until the previous phase is completed. We cannot overlap phases in waterfall model.

Phases of Waterfall model

**Waterfall Model**

Similarly waterfall model also works, once one phase of development is completed then we move to the next phase but cannot go back to the previous phase. In the waterfall model, the output of one phase serves as the input for the other phase.

### **5.3 Spiral model:** is a software development process model. This model has characteristics of both iterative and waterfall models. This model is used in projects which are large and complex. This model was named spiral because if we look at its figure, it looks like a spiral, in which a long curved line starts from the center point and makes many loops around it. The number of loops in the spiral is not decided in advance but it depends on the size of the project and the changing requirements of the user. We also call each loop of the spiral a phase of the software development process.

**Spiral Model**

In Spiral Model the entire process of software development is described in four phases which are repeated until the project is completed.

A software project goes through these loops again and again in iterations. After each iteration a more and more complete version of the software is developed. The most special thing about this model is that risks are identified in each phase and they are resolved through prototyping. This feature is also called Risk Handling.

Since it also includes the approaches of other SDLC models, it is also called Meta Model. It was first developed by Barry Boehm in 1986.

### 5.4 Agile model

is a combination of iterative and incremental models, that is, it is made up of iterative and incremental models.

* In Agile model, focus is given to process adaptability and customer satisfaction.
* In earlier times, iterative waterfall model was used to create software. But in today's time developers have to face many problems. The biggest problem is that in the middle of software development, the customer asks to make changes in the software. It takes a lot of time and money to make these changes. So to overcome all these shortcomings, the agile model was proposed in the 1990s.

The Agile Model was created mainly to make changes in the middle of **software development** so that the software project can be completed quickly.

Agile Model

* In the agile model, the software product is divided into small incremental parts. In this, the smallest part is developed first and then the larger one.
* And each incremental part is developed over iteration.
* Each iteration is kept small so that it can be easily managed. And it can be completed in two-three weeks. Only one iteration is planned, developed and deployed at a time.

## 6 Manual Testing

* Testing the functionality of an application with respect to given REQ.
* Checking the application with the intent of finding the defect.
* Checking the behaviour of an application to see whether it meets customer REQ or not.
* Testing the process of QA and QC.

TYPES OF TESTING

### 6.1 FUNCTIONAL TESTING

is a type of software testing that validates the software system against the functional requirements/specifications.

#### 6.1.1UNIT TESTING

* + It is testing the smallest element of SW/module
  + Make use white box testing (testing done on internal code)
  + No dependency between modules/ units

#### 6.1.2 INTEGRATION TESTING

* + Type of SW testing in which different units/ modules are combined and tested together
  + Big bang / top-down / bottom-up / white box testing

#### 6.1.3 SYSTEM TESTING

* + type of software testing that is performed on a complete integrated system to evaluate the compliance of the system with the corresponding requirements
  + Block box testing ( no internal testing)

#### 6.1.4 ACCEPTANCE TESTING

* + This is type testing done to verify the whether SW meets the requirement of the customer which Is documented into the requirement gathering.
  + This was the only feedback testing in waterfall model

#### 6.1.5 REGRESSION TESTING

* + a software testing practice that ensures an application still functions as expected after any code changes, updates, or improvements.

#### 6.1.6 SANITY TESTING

* + Sanity testing is a subset of regression testing. After receiving the software build, sanity testing is performed to ensure that the code changes introduced are working as expected. This testing is a checkpoint to determine if testing for the build can proceed or not.
  + It is subset of regression testing

#### 6.1.7 SMOKE TESTING

* + **the preliminary check of the software after a build and before a release**. This type of testing finds basic and critical issues in an application before critical testing is implemented

#### **6.1.8 USABILITY TESTING** It is testing where the end users are the testers

### **6.2 NON-FUNCTIONAL TSTING**

is defined as a type of Software testing to check non-functional aspects (performance, usability, reliability, etc) of a software application. It is designed to test the readiness of a system as per nonfunctional parameters which are never addressed by functional testing

#### **6.2.1 PERFORMANCE TESTING**

a testing measure that evaluates the speed, responsiveness and stability of a computer, network, software program or device under a workload

#### **6.2.2 LOAD TESTING**

type of Performance Testing that determines the performance of a system, software product, or software application under real-life based load conditions

#### **6.2.3 STRESS TESTING**

is a software testing technique that determines the robustness of software by testing beyond the limits of normal operation

#### **6.2.4 VOLUME TESTING**

is a type of Software Testing, where the software is subjected to a huge volume of data. It is also referred to as flood testing. Volume testing is done to analyse the system performance by increasing the volume of data in the database**.**

#### **6.2.5 SCALABILITY TESTING**

a type of load testing that measures the application's ability to scale up or down as a reaction to an increase in the number of users.

#### **6.2.6 RECOVERY TESTING**

is the activity of testing how well an application is able to recover from crashes, hardware failures and other similar problems

#### **6.2.7 COMPATIBILITY TESTING**

a form of non-functional software testing -- meaning it tests aspects such as usability, reliability and performance -- that is used to ensure trustworthy applications and customer satisfaction. Compatibility tests are crucial to the successful performance of applications.

#### 6.2.8 SECURITY **TESTING**

a process intended to reveal flaws in the security mechanisms of an

information system that protect data and maintain functionality as intended.

## 7 Software Build

a build is **the process of converting source code files into standalone software artifact(s) that can be run on a computer, or the result of doing so**

### 7.1 Difference Between Smoke and Sanity

|  |  |
| --- | --- |
| **SMOKE TESTING** | **SANITY TESTING** |
| **Smoke Testing** is performed to ascertain that the **critical functionalities** of the program are **working fine.** | **Sanity testing** is done at random to verify that **each functionality** is **working as expected.** |
| Smoke testing exercises the **entire system** from end to end. | Sanity testing exercises only the **particular component** of the entire system. |
| The main objective of the testing is to verify the **stability** of the build. | The main objective of the testing is to verify the **rationality** of the build. |
| Smoke testing is usually **documented and scripted.** | Sanity testing **is not** documented and is unscripted. |
| This testing is performed by the **developers or testers.** | Sanity testing in software testing is usually performed by **testers.** |
| It is a well **elaborate and planned** testing. | This **is not a planned** test and is done only when there is a shortage of time. |
| This is a **wide** and **deep** testing. | This is a **wide** and **shallow** testing. |
| Smoke testing is a subset of [Acceptance testing.](https://www.practitest.com/qa-learningcenter/best-practices/what-is-uat-testing/) | Sanity testing is a subset of **Regression Testing.** |

Types of Testing

### 7,2 Black box vs white box

**Black box Testing:** [Black Box Testing](https://www.browserstack.com/guide/black-box-testing) is a [software testing technique](https://www.browserstack.com/guide/what-is-software-testing) in which testers assess an application’s functionality without knowing its internal code or structure. The primary goal is to validate the software’s outputs based on various inputs, ensuring they meet the specified requirements and behave as expected.

Ex: Functional, regression, UI, Ad-hoc, Compatibility, Penetration, Security

**White box Testing:** [White box testing](https://www.browserstack.com/guide/white-box-testing), or glass box testing, is a software testing technique that focuses on the software’s internal logic, structure, and coding. It provides testers with complete application knowledge, including access to source code and design documents. This enables the testers to inspect and verify the software’s inner workings, infrastructure, and integrations.

Ex: Unit, Static code analysis

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Black Box Testing** | **White Box Testing** |
| Focus Area | Focuses on validating functionality and outputs based on requirements. | Focuses on verifying internal code structure, logic, and flow. |
| Testing Approach | Based on inputs and expected outputs. | Based on a detailed analysis of code paths, conditions, and loops. |
| Test Design | Designed using functional specifications and user requirements. | Designed using code structure, flowcharts, and logic diagrams. |
| Performed By | Usually performed by QA testers. | Typically performed by developers or white box testers. |
| Level of Testing | Mostly used for system testing, acceptance testing, and integration testing. | Commonly used for unit testing and sometimes integration testing. |
| Types of Bugs Found | Detects missing functionalities, interface issues, and incorrect outputs. | Identifies logical errors, hidden bugs in paths, and unreachable code. |
| Time Required | Generally faster to design but slower to execute due to lack of internal insight. | May take more time to design but provides deeper and faster debugging. |
| Automation Suitability | Easily automated for end-to-end and regression testing. | Best suited for automating unit and integration level tests. |

### 7.3. Static vs Dynamic

|  |  |
| --- | --- |
| Static Testing | Dynamic Testing |
| It is performed in the early stage of software development | It is performed at the later stage of the software development. |
| Testing is done without executing the program | Testing is done by executing the program |
| Also known as Verification testing | Also known as validation testing |
| It consists of reviews, walkthroughs, inspection, etc., | It consists of functional and non-functional testing, and data/control flow analysis. |
| Static testing assesses code and documentation | Dynamic testing gives bugs/bottlenecks in the software system. |
| Performs a dry run on the code as part of the static analysis. | Code is fully analysed for different paths by executing it. |
| It generally takes a shorter time. | It usually takes a longer time as it involves running several test cases. |
| Static testing covers structural and statement coverage testing. | Dynamic testing covers the executable file of the code. |
| It includes requirement documents, design documents, program specifications, etc., | It includes unit testing, integration testing, system testing, performance testing, security testing, etc., |

### 7.4 Verification vs validation

|  |  |  |
| --- | --- | --- |
| Parameter | Verification Testing | Validation Testing |
| Definition | Ensures the product meets specified requirements at various stages of development. | Ensures the product meets the stakeholders’ true needs and expectations. |
| Objective | To confirm that the product is being built correctly according to requirements and design specifications. | To confirm that the right product has been built and meets the end-user needs and requirements. |
| Timing | Performed at each stage of development (e.g., during coding, design, etc.). | Performed after the development is complete, often at the end of the project or phase. |
| Activities | Reviews, inspections, code analysis, unit testing. | User acceptance testing, system testing, beta testing. |
| Type of Testing | Static and dynamic testing activities. | Primarily dynamic testing activities. |
| Documentation Reviewed | Design documents, requirement specifications, code. | Final product, user feedback, system performance. |
| Example | Ensuring that a software module conforms to the design document. | Ensuring that a software application meets the requirements and expectations of the users. |

## 8 Testing Documents

### 8.1 Test Scenarios:

#### High-level descriptions of what needs to be tested, focusing on specific functionality or user workflows. These helps identify critical areas for testing without diving into detailed steps.

### 8.2 Test Cases:

#### Detailed documents specifying test steps, input data, expected results and actual results for individual test scenarios. They guide testers on how to execute tests systematically.

### 8.3 Test Plan*:*

#### A comprehensive document outlining the testing process’s scope, objectives, resources, schedule, and approach. It serves as a roadmap for the entire testing lifecycle.

* **Context of testing** (e.g., test scope, test objectives, test basis)
* **Assumptions and constraints of the test project**
* **Stakeholders** (e.g., roles, responsibilities, relevance to testing, hiring and training needs)
* **Communication** (e.g., forms and frequency of communication, documentation templates)
* **Risk register** (e.g., product risks, project risks)
* **Test approach** (e.g., test levels, test types, test techniques, test deliverables, entry criteria and exit criteria, independence of testing, metrics to be collected, test data requirements, test environment requirements, deviations from the test policy and test strategy)
* **Budget and schedule**

### 8.4 Requirement Traceability Matrix (RTM)*:*

#### A document that maps the requirements to the test cases, ensuring every requirement is tested and nothing is overlooked. It helps maintain traceability throughout the testing process.

### 8.5 Test Strategy*:*

#### A high level document that defines the testing approach, including methodologies, tools, team responsibilities and risk management strategies. It’s the foundation for planning and execution.

### **8.6 Test Data*:***

[Test data](https://www.browserstack.com/guide/what-is-test-data) is the information needed to execute test cases. This includes input values, expected outputs, and boundary values to validate software behaviour in different scenarios.

### **8.7 Bug Report**

A detailed record of issues or defects found during testing, including their description, severity, reproduction steps and status. It helps developers fix & track bugs efficiently.

### **8.8 Test Execution Report*:***

A summary of the results from executed test cases shows which tests passed, failed or skipped. This provides an overall view of the testing progress and software quality.

## **9 SEVERITY**:

Severity will tell how much that defect effect to the customer business is known as severity

Types of severity

1. Blocker

2. Critical

3. Major

4. Minor

5. Trivial: this defect is negligible

**PRIORITY:** Priority says which defect has to be fixed first by the developer. For every defect we have to set priority.

Different types of priority

1. High

2. Medium

3. Low

**Error:** A mistake made by a developer, tester, or even a user.

Can be a coding mistake (e.g., syntax error, logical error) or a misunderstanding of requirements.

Example: Typo in code, incorrect algorithm implementation, misinterpretation of a requirement.

**Defect:** A flaw or imperfection in the software.

A deviation from the expected behavior or requirement.

Example: Software crashes when a specific input is entered, a feature not working as described in the requirements.

* **Defects cluster together**: A small number of system components usually contain most of the defects discovered or are responsible for most of the operational failures (Enders 1975). This phenomenon is an illustration of the Pareto principle. Predicted defect clusters, and actual defect clusters observed during testing or in operation, are an important input for risk-based testing

**Issue:** A general term for any problem or concern related to software.

Can include defects, but also broader concerns like usability problems, security vulnerabilities, or even changes to the project scope.

Example: A UI element is not intuitive for users, a security vulnerability is discovered, or there's a disagreement about the scope of a feature.

# Java

## 1 Language Fundamentals

### 1.1 Identifiers

A name in java is called Identifier. It may be class name, methos name, variable name.

Rules of Identifier

* The only allowed characters are “a to z”, “A to Z”, “0 to 9”, “\_”, “$”. Other than these words we will get compile time error.
* Identifier can’t start with number.
* Identifiers are case sensitive.
* No limitation on length, but it’s not recommended.
* Reserved words can’t be identifiers.
* All Predefined classes and interface name can be used as identifiers but not recommended.

### 1.2 Reserved Words

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| abstract | continue | for | new | switch |
| assert | default | Goto | package | synchronized |
| boolean | Do | If | private | this |
| break | double | implements | protected | throw |
| byte | Else | import | public | throws |
| case | Enum | instanceof | return | transient |
| catch | extends | Int | short | try |
| char | final | interface | static | void |
| class | finally | long | strictfp | volatile |
| const\* | float | native | super | while |

### 1.3 Data Types

Primitive Data types.

Numeric Data types Char Data Types (1 bytes) Boolean Data Types

Integra Data Types Floating

.byte (1bytes) 1. float (4 bytes) 5 to 6 decimal places

.int (2 bytes) 2. Double (8 bytes) 14 to 15 decimal places

.int (4 bytes)

.long (8 bytes)

### 1.4 Literals

A constant value that can be assigned to a variable is called literals

Int x =10

Integral literal

* Decimal – 0 to 9
* Octal- 0 to 7
* Hexa decimal- 0 to 9, a to f, A to F
* Floting point Literals- Every floating point literals are double type hence we cant assign directly to a float. Hence define by f or F at end.
* Boolean literals: true and false
* String Literals: Combination of charecters
* Char literals: ‘a’, ‘b’
* \n: Newline. Moves the cursor to the beginning of the next line.
* \t: Tab. Inserts a horizontal tab space.
* \b: Backspace. Moves the cursor one position back.
* \r: Carriage return. Moves the cursor to the beginning of the current line.
* \\: Backslash. Inserts a literal backslash character.
* \": Double quote. Inserts a double quote character within a string.
* \': Single quote. Inserts a single quote character within a string.

### 1.5 Array

#### Array Declaration

* An array is an indexed collection of fixed no of homogenous data elements
* Main advantage of array is we can represent multiple values under the same name so that readability of the improved
* Limitations are
  + No chance to increase or decrease size
  + Same data types
* We can resolve this problem by using collection

Single dimensional array

int[] a;

int a[];

int []a;

2D array

int[][] a;

int [][]a;

int a[][];

int [] []a;

3D array

int [][][] a;

int a[][][];

### 1.6 Array Creation

* Every array in java is an object. Hence we can create by using new operator

int [] a = new int [3]

* while creating array we need to specify array size else we will get CE
* 0 is not allowed, if -ve value we will get NegativeArraySizeExcetion
* To specify array size allowed data types are byte, short, int, char\\
* 2D array are implemented as array of arrays

Int [][][] a = new int [2][][]

#### Array initialization

Whenever we create a array automatically every element is initialized with default value

* We can initialize each value

int [] a = new int [5]

a[0]=1;

a[1]=2

.

.

a[4]=5;

* When we try to access array value out of range we will get AIOBE

#### Declaration, creation, initialization in one line

int [] a= {10,20,30,40,50};

char[] a ={‘a’,’b’,’c’,’d’,’e’};

int[] b= {{1,2,3,4},{1,2,3,4}};

#### Length vs length()

length is a final variable applicable for array. It represents size of array.

Sysout(a.length);

length() is method is applicable for String, representing number of charecters

String s =”abhi”

Sysout(s.length()); 4

#### Anonymous array

Array without name is anonymous array its used for instant only

New int[]{10,20,30}

### 1.7 Types of Variables

Based on types of value represented by a variable all variable are divided into 2 types

Primitive Variables

Reference Variables

1. Primitive Variables
   * Can be used to represent primitive values
2. Reference variables
   * Can be used to refer objects

Based on purpose and position of declaration all variable are divided into 3 types

##### Instance variable

* + If the value of a variable is variable from object to object
  + For every object a separate instance variable will be created
  + The scope of instance variable is exactly same as the scope of the object because instance variable will be created at the time of object creation and destroyed at object distraction
  + instance variable is declared inside class but outside of any method, block, constructor
  + instance variable can bot be accessed from static area directly can be accessed by object reference
  + inside same class we can access directly
  + no need to assign value every time JVM will provide default values

class test

{

String s;

int x;

p s v m ( String args[]){

test t = new test ();

sopln(t.s); null

sopln(t.x); 0

}

}

##### Static variable

* + if the value of a variable is not varied from object to object then it is never recommended to declare it as instance variable. We have to declare it as static variable at class level using static key word
  + Static variable copy will be created at class level and copy will be shared by all objects of that class
  + static variable will be created at time of class loading destroyed at class unloading
  + static variable should be declared at class level outside of any method
  + static variable can be accessed by class name or object reference
  + if we make changes in static variable of any once instance of class that will changes in all instances of class

##### Local variable

* + to meet temporary requirement of the programmer local variable will be created
  + local variable will be stack
  + these will be created while executing specific block in which we declare it and destroyed once the block completed
  + local variable JVM will not assign default value

### 1.8 Main() Method

* + whether the class contains main() or not & whether the main () is properly declared or not checking this is responsibility of JVM
  + if JVM not find main() we will get run tine exception NOSuchMethodError:main

public static void main (String args[])

* + any changes in syntax will get sun time exception

## 2 Operators and Assignment

### 2.1 Increment / decrement operator

#### 2.1.1 Increment

* + pre increment

int x =++y; this will increment value before assigning and then assign

* + Post increment

Int x = y++; this will assign value first and increment the value

#### 2.1.2Decrement

* + pre decrement

int x =--y; this will decrement value before assigning and then assign

* + Post decrement

Int x = y--; this will assign value first and decrement the value

### 2.2 Arithmetic operators

+, -, \*, /, %

String concatenation is possible by +

### 2.3 Relational operators

>, < , >=, <=

This will return Boolean value upon comparing two value

Not applicable for object type

To compare string we can not use == this will compare object reference

S1.equals(s2) will compare strings

### 2.4 Equality operators

== , !=

### 2.5 Bitwise operators

& and , | or, ^ xor

### 2.5 Short circuit operators

&& , ||

### 2.6 Instance of operators

To check an object is an instance of a class or not

### 2.7 Assignment operators

=

### 2.8 conditional operators

( condition ? exp if true: exp Iffalse)

### 2.9 new operators

to create new object of a class

## 3 Flow control

### 3.1 Selection Statement

#### 3.1.1 if-else

if(statement){  
 if statement id true

}

else{

if statement is false

}

* Argument to if should be Boolean
* Curly braces are optional

#### 3.1.2Switch

* If several option present, then go for switch
* Curly braces are compulsory

switch(x)

{

Case 1:

Action 1;

break;

Case 2:

Action 2;

break;

default:

default case;

)

### 3.2 Iterative Statement

#### 3.2.1 while

* When we don’t know the number of iterations

while(rs.next()) {

{

action

}

#### 3.2.2 do-while

* If we want to execute loop at least once

Do

{

} while (b)

#### 3.2.3 for

for (initialization ; condition ; increment)

#### 3.2.4 for-each

* This is enhanced for loop for retrieving elements from array and collection

for(int x: a)

### 3.3 Transfer Statement

#### 3.3.1 Break

* We use to exit the loop
* Inside labelled block to break that block execution based on some conditions

switch(b){ for (int I =0; i<10; i++){

. if (i==5)

. break;

. sopln(i);

Break }

.

.

}

#### 3.3.2 Continue

* We use continue to skip current iteration and continue for the next inside loop

for(int i=0;i<10;i++){

if(i%2==0)

continue;

sopln(i);

} to print odd

#### 3.3.3 Return

#### 3.3.4 Try

#### 3.3.5 Catch

#### 3.3.6 final

## 4 Declaration and Access modifiers

### 4.1 Java source file Structure

* A java program can contain any number of classes but at most one class can be declared as public

#### 4.1.1 packages

* It is an encapsulation mechanism to group related classes and interfaces into single module
* This resolve naming conflict
* To provide security to the class and interfaces
* It improves modularity of app

### 4.2 Class modifiers

* The only applicable modifiers are

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Visibility | Private | Default | Protected | Public |
| Same class | Y | Y | Y | Y |
| Child class of same package | N | Y | Y | Y |
| Non child class of same package | N | Y | Y | Y |
| Child class outside package | N | N | Y | Y |
| Non child class outside package | N | N | N | Y |

**Public Class**

* If a class is declared as public, then we can access from anywhere

**Default Class**

* Default class can be access in same package

**Final modifiers**

* Final modifier is applicable for classes, methods and variable
* If a method is declared as final we can not override the method in child class
* If a class is declared as final we can not create child classes of it
* It is used when we want to avoid changes in implementation

**Abstract modifiers**

* Is applicable for classes and methods

**Abstract methods**

* When we don’t know the implementation of a class we will declare the method as abstract. Methods declared as abstract should end with ;
* Child classes are responsible for implementation

**Abstract Class**

* When we don’t want to instantiate the class we declare it as abstract
* We need to extend or create a child class of it

**Final variable**

* If a instance variable is declared as Final we should assign a value

**Final static variable**

* For normal static variable it is not required to perform initialization for final static variable we should

**Final Local Variable**

* Should assign value

**Static modifiers**

* Applicable for variable and method
* If variable is varied from object to object, then we should go for instance variable. If it is fixed for all object, then we should go for static variable
* Static methods should have implementation compulsory and abstract no need of implementation.

**Synchronized modifiers**

* Applicable for methods and blocks not class and variable
* If a method and block is declared then at a time one thread is allowed to operate

### 4.3 interfaces

#### 4.3.1 Introduction

* Any service requirement specification or 100% abstract class is an interface
* We can achieve security

#### 4.3.2 Declaration and implementation

We can declare interface by keyword interface

interface inter

{

Mthod1;

}

If a class implements interface, it should provide implementation for all method of that interface otherwise we should declare it as abstract

Implementation should be declared as public

#### 4.3.3 Interface Methods

Every interface method is public and abstract

#### 4.3.4 Interface variables

An interface can contain variable main purpose is to specify constant at requirement level

Public: to make this variable available for every implementation

Static: without existing object also implementation class can access this variable

Final: implementation class can access this variable but cannot modify

#### 4.3.5 Marker Interface

If an interface wont contain any method and by implementing that interface if our object will get ability such type of interface are marker interface

**Serializable** we can send object

**Cloneable** our object will be in position to provide exactly duplicate

#### 4.3.6 Abstract vs concrete vs interface

When we don’t know any implementation and we have requirement we should go for **Interface**

We know partial implementation then we go for **abstract class**

When we know implementation the **concrete class**

## 5 OOPs

### Data hiding

Hiding the data so outsider can not access directly

Using private key we declare to implement data hiding

Class account {

private double balance=1000.00;

}

### Abstraction

Hiding the internal implementation and highlighting the set of implementations is abstraction

We can achieve the security as implementation is hidden

### Encapsulation

Encapsulating data and corresponding methods into single module

If any class follows data hiding and abstraction is called encapsulated class

Encapsulation= data hiding + abstraction

### Tightly encapsulated classes

If all members of class are private, then class is said to be tightly encapsulated

Getter and setter methods should be used

### Is -a relationship

Is also known as inheritance

By using extend keyword we can implement is a relationship

Main advantage is reusability

1. Single inheritance
2. Multiple inheritance- not supported in Java
3. Multilevel inheritance
4. Hierarchical inheritance

### Has -a relationship

Composition or aggregator

New keyword is used to implement has a relationship

### Method signature

Method signature does not include return type

Within a class methods with same name is not allowed

### Overriding

Run time polymorphism or dynamic polymorphism or late binding

In overriding method name and argument should be matched

Method declared as **final** cannot be overridden

**Private** methods are not visible outside of class

**Abstract** methods of parent class should be overridden in child class to provide implementation

### Overloading

When two methods have same name with different argument

In overloading method resolution child argument will get priority

### Method hiding

|  |  |
| --- | --- |
| Method hiding | Overriding |
| Both methods should be static | Both methods should be non-static |
| Method resolution takes care by compiler based on reference type | Method resolution takes care by JVM based on runtime |
| Compile type polymorphism | Runtime polymorphism |

### Static control flow

When we execute the java class .class should be loaded

Identification of static member from top to bottom

Execution of static variable and static block from top to bottom

Main method execution

At the time of class loading if we want some activities to be done then we need to declare them in static block

Any number of static blocks are fine execution will be from top to bottom

### Instance control flow

When we create an object following events will occur

* Identifying instance member from top to bottom
* Execute instance variable assignment and instance block from top to bottom
* Execution of contractor
* Its one-time activity

When we create child class

* Identification of instance member from parent to child
* Instance variable and instance block only in parent
* Execution of parent class contractor
* Instance variable and instance block in child
* Execution of child class constructor

### Contractor

Object creation is not enough we should perform initialization

When we create a object of class certain code will be executed by default is called contractor

Class Student {

Int id;

String name;

Student (String name, int id){

this.name= name;

this.id= id;

}

P S V M (Sting args)

}

When we want to initialize the instance variable then we should go for contractor

Other than initialization all things should go for instance block

Contractor will not take any arguments instance block will take

### Singleton class

For any java class if we are allowed to create only once class the its is singleton class

Instead of creating object at each instance one object is shared by all

## 6. collections

### 6.1 Arrays

An array is an indexed collection of fixed number od homogeneous data elements

Limitation

* Fixed size
* Only homogeneous data elements allowed we can resolve this by object type array
* No built-in data structure

### 6.2 9 key interfaces of collection framework

#### 6.2.1 collection (interface)

When we want to represent group of individual object as an single entity the we go for collection

Collection is an interface can be used to represent a group of individual object as a single entity

Collections is an utility class present in java.util.package to define several utility for collection

#### 6.2.2 List

Where insertion order is preserved, and duplicates are allowed

##### 6.2.2.1 ArrayList

Heterogeneous objects allowed

Null insertions is possible

ArrayList al = new ArrayList (int initialCapacity)

##### 6.2.2.2 LinkedList

LinkedList l = new LinkedList();

##### 6.2.2.3 VectorList

Vector v = new vector();

#### 6.2.3 set

Duplicates are not allowed and insertion order not preserved

##### 6.2.3.1 HashSet

Iterator is universal cursor

Remove and read operation

Iterator it = c.iterator();

##### 6.2.3.2 LinkedHashSet

#### 6.3.4 SortedSet

Object in certain sorting order

SortedSet tailSet(Obejct obj) -Returns sorted set greater than or equal to object

SorteSet subset(object obj, object obj2)- whose object are >= i=obj1 but <= obj2

6.3.5 NavigableSet

Child of sorted set

Collection 🡪set 🡪 sorted set 🡪 navigable set 🡪tree set

#### 6.3.6 Queue

#### 6.3.7 Map

Key value pair duplicate not allowed

* HashMap

#### 6.3.8 sorted map

Represent map object into certain sorted order

#### 6.3.9 navigable map

Child of sorted map

## 7 Generics

### 7.1 Introduction

Array are safe w.r.t. type, but when we try to insets other than the declared type we get compile time error. So type casting is must. Generics was introduced

ArrayList <String> al = new ArrayList<String>

To ArrayList <T> al = new ArrayList<T>;

### 7.1 Generic classes

### 7.3 Bounded classes

### 7.4 Generic methods

# BDD Cucumber

## Background vs scenario Outline

“In Cucumber, both Background and Scenario Outline are used to improve reusability in feature files, but they serve different purposes:

* **Background** is used to define **common steps** that should run before **each scenario** in a feature file. It's like a shared setup section—useful when all scenarios need the same starting point, like launching the browser or navigating to a login page.

Yes, the Background section itself doesn’t contain code, but the **steps written inside it do trigger code** through their **matching step definitions**—just like any other scenario step.

Background:

Given User launches the browser

And User navigates to the login page

These steps will **automatically run before every scenario** in the feature file. The actual logic (code) is implemented in the **step definition class**, like:

@Given("User launches the browser")

public void launchBrowser() {

driver = new ChromeDriver();

}

@And("User navigates to the login page")

public void navigateToLoginPage() {

driver.get("https://example.com/login");

}

* **Scenario Outline**, on the other hand, is used when you want to run the **same scenario multiple times** with **different sets of input data**. It acts like a loop, using an Examples table to pass different values for each test iteration.

No, the **Scenario Outline itself does not contain code**, but the **steps inside it** are linked to **step definition methods**—just like in regular scenarios.

What makes Scenario Outline special is that it allows you to **run the same scenario multiple times** with **different test data** provided in the Examples table.

## Basic structure

**Feature: Login Feature** ← Feature File written in Gherkin

**Background:** **Given The application is launched** ← Reusable steps for every scenario

**@Smoke** ← Tag

**Scenario: Valid login** ← Scenario

**Given** User is on login page

**When** User enters valid credentials

**Then** User should see dashboard

**Scenario Outline:** Invalid login

**Given** User is on login page

**When** User enters "<username>" and "<password>"

**Then** Error message should be displayed

Examples:

| username | password |

| user1 | wrong1 |

| user2 | wrong2 |

## Common BDD terms

|  |  |
| --- | --- |
| **Term** | **Meaning** |
| **BDD** | Behavior-Driven Development – a software development method that encourages collaboration between QA, developers, and business using plain language. |
| **Feature File** | A .feature file written in **Gherkin** language describing the application behavior. |
| **Gherkin** | A plain-English language used to write BDD scenarios. Uses keywords like Given, When, Then. |
| **Scenario** | A single test case in a feature file that describes a specific behavior or use case. |
| **Step** | A line in the scenario that describes a step of behaviour using keywords: Given, When, Then, And, But. |
| **Given** | Describes the **initial context** (precondition) of the scenario. |
| **When** | Describes the **action** or event performed by the user/system. |
| **Then** | Describes the **expected outcome** or result of the action. |
| **And / But** | Used for readability when there are **multiple Given/When/Then** steps. |
| **Step Definition** | The actual Java (or other language) code that executes the logic for a step. |
| **Glue Code** | The package/folder that contains step definitions, hooks, and support classes. |
| **Hooks** | Special blocks like @Before, @After, @BeforeStep, @AfterStep that run automatically before/after scenarios or steps. |
| **Tags** | Keywords like @Smoke, @Regression used to group and run specific scenarios. |
| **Cucumber** | A tool that supports BDD by reading Gherkin feature files and executing matching step definitions. |
| **Test Runner** | A Java class (with JUnit/TestNG) that runs all BDD tests by linking feature files and glue code. |
| **Background** | A section in the feature file that defines steps common to all scenarios in that file. |
| **Scenario Outline** | A template for running a scenario multiple times with different sets of test data. |
| **Examples** | The data table provided under a Scenario Outline to repeat the same test with different inputs. |
|  |  |

## Feature file vs step definition

|  |  |
| --- | --- |
| **Feature File** | **Step Definition** |
| **Written in Gherkin language** | **Written in Java (or other supported language)** |
| Describes **what** the system should do | Describes **how** to do it using code |
| Used by **Business Analysts, Testers, and Developers** | Used by **Developers/Test Automation Engineers** |
| Contains **scenarios** and **steps** (Given/When/Then) | Contains methods with @Given, @When, @Then annotations |
| Example: Login.feature | Example: LoginSteps.java |
| Stored in: src/test/resources/features/ | Stored in: src/test/java/stepDefinitions/ |

Create a feature file and create a runner class and run it, when there are no step definition file found for the feature it will generate the step definition file

## Project Structure

**src/**

**├── test/**

**│ ├── java/**

**│ │ ├── pages/**

**│ │ │ └── LoginPage.java**

**│ │ ├── stepDefinitions/**

**│ │ │ ├── Hooks.java**

**│ │ │ └── LoginSteps.java**

**│ │ └── runners/**

**│ │ └── TestRunner.java**

**│ └── resources/**

**│ └── features/**

**│ └── Login.feature**

## Hooks

* **Hooks** are special code blocks in Cucumber that run **before** or **after** each scenario or step. They are mainly used to manage **setup** and **teardown** operations.

|  |  |  |
| --- | --- | --- |
| **Hook Type** | **Trigger Timing** | **Use Case Example** |
| @Before | **Before each scenario** | Launch browser, setup environment |
| @After | **After each scenario** | Close browser, take screenshot on failure |
| @BeforeStep | **Before each step** | Log each step start, prepare pre-step data |
| @AfterStep | **After each step** | Capture screenshot after step |
| @BeforeAll (Java 8+) | **Before all scenarios** | Global setup like DB connection (Rare) |
| @AfterAll (Java 8+) | **After all scenarios** | Global teardown like DB disconnect |

## 7. Data Table

A data table in Cucumber is a way to deliver a set of data to a Step in a Scenario in a tabular manner. In Cucumber, a data table is defined by a series of pipes (|) to split the columns and hyphens (-) that divide the header row from the data rows. For instance, the following data table has two rows of data and the following two columns: "Username" and "Password":

## 8. Data driven

Cucumber can handle dynamic input by using a scenario outline with an Examples table. Here is a brief sample of code:

Scenario Outline: Login **with** dynamic credentials

Given User is on login page

When User enters "<username>" and "<password>"

And User clicks on login button

Then User should be logged **in** successfully

Examples:

| username | password |

| user1 | pass123 |

| user2 | pass456 |

# Selenium

## Browser/Commands

|  |  |
| --- | --- |
| **Command / Method** | **Description** |
| get("url") | Opens a URL in the browser |
| getTitle() | Returns page title |
| getCurrentUrl() | Returns current page URL |
| getPageSource() | Returns HTML source of the page |
| close() | Closes the current browser window |
| quit() | Closes all browser windows and ends the session |

## Navigation commands

|  |  |
| --- | --- |
| **Command** | **Description** |
| navigate().to("url") | Navigates to a URL |
| navigate().back() | Go back in browser history |
| navigate().forward() | Go forward |
| navigate().refresh() | Refresh the current page |

## WebElement interaction

|  |  |
| --- | --- |
| **Command** | **Description** |
| findElement(By) | Locates a single web element |
| findElements(By) | Locates multiple web elements |
| click() | Clicks the element |
| sendKeys("text") | Types text into a field |
| clear() | Clears text from input field |
| getText() | Gets visible text |
| getAttribute("name") | Gets attribute value |
| isDisplayed() | Checks if element is visible |
| isEnabled() | Checks if element is enabled |
| isSelected() | Checks if checkbox/radio is selected |

## Waits

|  |  |
| --- | --- |
| **Command** | **Description** |
| **Implicit Wait** | driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(10)); |
| **Explicit Wait (WebDriverWait)** | new WebDriverWait(driver, Duration.ofSeconds(10)).until(ExpectedConditions.elementToBeClickable(locator)); |
| **Fluent Wait** | Advanced explicit wait with polling |

## Select class

|  |  |
| --- | --- |
| **Command** | **Description** |
| selectByVisibleText("text") | Select option by text |
| selectByIndex(int) | Select by index |
| selectByValue("value") | Select by value |
| getOptions() | List all options |
| getFirstSelectedOption() | Get selected option |

Select select = new Select(driver.findElement(By.id("dropdown")));

select.selectByVisibleText("Option 1");

## Window handles

|  |  |
| --- | --- |
| **Command** | **Description** |
| getWindowHandle() | Returns current window ID |
| getWindowHandles() | Returns all open window IDs |
| switchTo().window(handle) | Switch to another window/tab |

## Frames

|  |  |
| --- | --- |
| **Command** | **Description** |
| switchTo().frame(index/name/WebElement) | Switch to frame |
| switchTo().defaultContent() | Back to main content |

## 8. Alerts

|  |  |
| --- | --- |
| **Command** | **Description** |
| switchTo().alert() | Switch to alert box |
| accept() | Click OK |
| dismiss() | Click Cancel |
| getText() | Read alert message |
| sendKeys("text") | Enter text in prompt alert |

Alert alert = driver.switchTo().alert();

alert.accept();

## mouse and keyboard actions

|  |  |
| --- | --- |
| **Command** | **Description** |
| moveToElement() | Hover over element |
| click() | Mouse click |
| doubleClick() | Double click |
| contextClick() | Right-click |
| dragAndDrop(source, target) | Drag source and drop to target |
| sendKeys(Keys.ENTER) | Keyboard actions |

Actions actions = new Actions(driver);

actions.moveToElement(element).click().build().perform();

|  |  |
| --- | --- |
| **Key** | **Constant** |
| moveToElement(element) | Mouse hover over element |
| click() | Mouse click |
| doubleClick() | Double click on element |
| contextClick() | Right-click on element |
| dragAndDrop(source, target) | Drag from source to target |
| clickAndHold() | Click and hold mouse |
| release() | Release mouse |
| sendKeys(Keys.KEY) | Keyboard action (like ENTER, TAB) |
| build().perform() | Execute the action |

Always finish with .perform() or .build().perform()

## Robot class

Robot robot = new Robot();

robot.keyPress(KeyEvent.VK\_DOWN);

robot.keyPress(KeyEvent.VK\_ENTER);

|  |  |
| --- | --- |
| **Key** | **Constant** |
| Enter | KeyEvent.VK\_ENTER |
| Tab | KeyEvent.VK\_TAB |
| Ctrl | KeyEvent.VK\_CONTROL |
| Shift | KeyEvent.VK\_SHIFT |
| Esc | KeyEvent.VK\_ESCAPE |
| A-Z | KeyEvent.VK\_A to KeyEvent.VK\_Z |
| 0-9 | KeyEvent.VK\_0 to KeyEvent.VK\_9 |

## Page Factory

public class LoginPage {

@FindBy(id="username")

WebElement usernameField;

@FindBy(id="password")

WebElement passwordField;

@FindBy(id="login")

WebElement loginBtn;

## Locators

|  |  |  |  |
| --- | --- | --- | --- |
| **Locator Type** | **Syntax** | **Example** | **Use Case** |
| **ID** | By.id("id") | By.id("username") | Best when ID is unique |
| **Name** | By.name("name") | By.name("email") | Good if name is available & unique |
| **Class Name** | By.className("class") | By.className("input-text") | Use when class is unique (avoid spaces) |
| **Tag Name** | By.tagName("tag") | By.tagName("input") | General identification of elements |
| **Link Text** | By.linkText("text") | By.linkText("Login") | For full visible link texts |
| **Partial Link Text** | By.partialLinkText("partial") | By.partialLinkText("Log") | When link text is dynamic or partial |
| **XPath** | By.xpath("xpath expression") | By.xpath("//input[@id='email']") | Most flexible; supports complex logic |
|  |  | By.xpath("//div[contains(@class,'menu')]") | Handles dynamic attributes |
| **CSS Selector** | By.cssSelector("CSS expression") | By.cssSelector("input#email") | Fast and cleaner alternative to XPath |
|  |  | By.cssSelector("input[name='username']") | Use multiple attributes |

## ScreenShot

File src = ((TakesScreenshot) driver).getScreenshotAs(OutputType.FILE);

## 14.

## 15.

## 16.

## 17.

## 18.