Table Of Content

About Me

⇔ For more related content refer to MY **LinkedIn**, **YouTube**, **Medium**
⇔ **Let's Connect over a Call: https://topmate.io/sidharth_shukla/**

Coding Standards

1. Naming Conventions

- **Classes:** Use PascalCase (e.g., EmployeeDetails, OrderService).
- **Methods:** Use camelCase with action verbs (e.g., calculateTotal, fetchData).
- **Variables:** Use camelCase (e.g., totalAmount, userName).
- **Constants:** Use UPPER_CASE with underscores (e.g., MAX_LIMIT, DEFAULT_TIMEOUT).
- **Packages:** Use all lowercase letters (e.g., com.company.project).

2. Code Formatting

Indentation: Use 4 spaces per indentation level (avoid tabs).

Line Length: Keep lines under 80-120 characters.

Braces Placement: Follow the K&R style (same line for opening braces):

```
if (condition) {
    // code block
}
```

Whitespace: Add spaces around operators and after commas for better readability:

```
int total = count + 5;

list.add(item1, item2);
```

3. Comments

Use Javadoc: Add comments for classes, methods, and complex logic:java
Copy code

```
/**

* Calculates the total amount including tax.

* @param price The base price of the item.

* @param taxRate The tax rate applied.

* @return The total amount.

*/

public double calculateTotal(double price, double taxRate) { ... }
```

Avoid obvious comments (e.g., // Add two numbers for a + b).

Use // for inline or single-line comments and /* */ for multi-line comments.

4. Error Handling

Always handle exceptions gracefully using try-catch:

```
try {

    // code that may throw exception

} catch (Exception e) {

    logger.error("Error message", e);

}
```

Avoid empty catch blocks or generic exception handling without logging.

5. Code Structure

Class Structure Order:

1. Package declaration
2. Import statements
3. Class-level comments
4. Fields
5. Constructors
6. Public methods
7. Private methods

One Class Per File: Each class should have its own file, and the file name should match the class name.

6. Avoid Magic Numbers

Replace hardcoded values with constants:

final int MAX_RETRIES = 3;

for (int i = 0; i < MAX_RETRIES; i++) { ... }

7. Collections and Generics

- Always use generics for collections:java
  Copy code

  List<String> names = new ArrayList<>();
-

8. Access Modifiers

Use the least restrictive access modifier necessary:

private → protected → public

Encapsulate fields with getters and setters.

9. Code Optimization

Avoid unnecessary object creation:

String result = new String("Hello"); // Avoid this

- String result = "Hello"; // Preferred
-

- Use StringBuilder for string concatenation in loops.

10. Testing

- Write unit tests for all public methods.
- Follow naming conventions for test methods (e.g., shouldCalculateTotalCorrectly).

++

Naming Convention

1. Use Explanatory Variables

Bad:
int d; // total execution time in seconds
Good:
int totalExecutionTimeInSeconds;
int elapsedTimeSinceTestStart;
int pageLoadTimeInSeconds;

2. Make Meaningful Distinctions
Bad:
```
void interactWithElements(WebElement e1, WebElement e2) {
   e1.click();
   e2.sendKeys("Test Data");
}
```
Good:
```
void interactWithElements(WebElement button, WebElement textField) {
   button.click();
   textField.sendKeys("Test Data");
}
```

3. Use Pronounceable Names
Bad:
```
class LgnPg {
   private WebElement usrnmFld;
   private WebElement pswFld;
}
```
Good:
```
class LoginPage {
   private WebElement usernameField;
   private WebElement passwordField;
}
```

4. Use Searchable Names
Bad:

```
for (int i = 0; i < 10; i++) {
    driver.findElement(By.id("element" + i)).click();
}
```
Good:
```
final int MAX_ELEMENTS = 10;
for (int elementIndex = 0; elementIndex < MAX_ELEMENTS; elementIndex++) {
    driver.findElement(By.id("element" + elementIndex)).click();
}
```
5. Replace Magic Numbers with Named Constants
Bad:
```
driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
```
Good:
```
final int IMPLICIT_WAIT_TIME = 30;
driver.manage().timeouts().implicitlyWait(IMPLICIT_WAIT_TIME, TimeUnit.SECONDS);
```

Functions
1. Do One Thing
Bad:
```
public void testLoginAndNavigate() {
    driver.findElement(By.id("username")).sendKeys("user");
    driver.findElement(By.id("password")).sendKeys("password");
    driver.findElement(By.id("loginButton")).click();
    driver.findElement(By.linkText("Dashboard")).click();
}
```
Good:
```
public void testLoginAndNavigate() {
    login("user", "password");
    navigateToDashboard();
}
private void login(String username, String password) {
    driver.findElement(By.id("username")).sendKeys(username);
    driver.findElement(By.id("password")).sendKeys(password);
    driver.findElement(By.id("loginButton")).click();
}
private void navigateToDashboard() {
    driver.findElement(By.linkText("Dashboard")).click();
}
```
2. Use Descriptive Names
Bad:
```
public void chkBtn() {
    // code to verify button functionality
}
```
Good:
```
public void verifyLoginButtonFunctionality() {
    // code to verify button functionality
}
```
3. Prefer Fewer Arguments
Bad:
```
public void setupDriver(WebDriver driver, int waitTime, String url) {
    driver.manage().timeouts().implicitlyWait(waitTime, TimeUnit.SECONDS);
    driver.get(url);
}
```

Good:

```
public void setupDriver(DriverConfig config) {
    driver.manage().timeouts().implicitlyWait(config.getWaitTime(), TimeUnit.SECONDS);
    driver.get(config.getUrl());
}
```

4. Avoid Side Effects

Bad:

```
public void validateAndInitializeSession() {
    if (isUserLoggedIn()) {
        initializeSession();
    }
}
```

Good:

```
public boolean validateSession() {
    return isUserLoggedIn();
}
public void initializeSessionIfValid() {
    if (validateSession()) {
        initializeSession();
    }
}
```

Classes

1. Single Responsibility Principle

Bad:

```
class LoginPage {
    public void login(String username, String password) {
        // login code
    }

    public void generateReport() {
        // report generation code
    }
}
```

Good:

```
class LoginPage {
    public void login(String username, String password) {
        // login code
    }
}
class ReportPage {
    public void generateReport() {
        // report generation code
    }
}
```

Comments

1. Explain Yourself in Code

Bad:

```
// Check if login button is enabled
if (driver.findElement(By.id("loginButton")).isEnabled()) {
    // Click login button
    driver.findElement(By.id("loginButton")).click();
}
```

Good:

```
WebElement loginButton = driver.findElement(By.id("loginButton"));
if (loginButton.isEnabled()) {
    loginButton.click();
}
```

2. Avoid Commenting Out Code
Bad:

```
// driver.findElement(By.id("username")).sendKeys("user");
// driver.findElement(By.id("password")).sendKeys("password");
driver.findElement(By.id("loginButton")).click();
```

Good:
Remove the commented-out code if it's no longer necessary.
Summary
These examples demonstrate how clean coding practices can be applied to Selenium tests to make them more modular, readable, and maintainable. By following these principles, your test code becomes less error-prone and easier to extend over time.

Code Optimization

1. Avoid Unnecessary Object Creation
Optimization: Reuse objects instead of creating new ones repeatedly.

Before:

```
String message = new String("Hello World");
```

After:

```
String message = "Hello World"; // String literal reuses the object
```

2. Use StringBuilder for String Manipulation
Optimization: Avoid using String concatenation in loops, as String is immutable and creates new objects.

Before:

```
String result = "";
for (int i = 0; i < 10; i++) {
    result += i; // Inefficient
}
```

After:

```
StringBuilder result = new StringBuilder();
for (int i = 0; i < 10; i++) {
    result.append(i); // Efficient
}
```

3. Use Enhanced For Loop
Optimization: Enhanced for-loops are easier to read and avoid potential index errors.

Before:
```
for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i));
}
```
After:
```
for (String item : list) {
    System.out.println(item);
}
```

4. Cache Length or Size in Loops
Optimization: Avoid recalculating size() or length in every iteration.

Before:
```
for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i));
}
```
After:
```
int size = list.size();
for (int i = 0; i < size; i++) {
    System.out.println(list.get(i));
}
```

5. Use Lazy Initialization
Optimization: Delay the creation of objects until they are needed.

Before:
```
private List<String> data = new ArrayList<>();
```
After:
```
private List<String> data;
public List<String> getData() {
    if (data == null) {
        data = new ArrayList<>();
    }
    return data;
}
```

6. Use Streams for Bulk Operations
Optimization: Use Java Streams for concise and efficient operations.

Before:
```
List<String> names = new ArrayList<>();
for (Person p : people) {
    if (p.getAge() > 18) {
        names.add(p.getName());
    }
```

```
}
```
After:
```
List<String> names = people.stream()
                        .filter(p -> p.getAge() > 18)
                        .map(Person::getName)
                        .collect(Collectors.toList());
```

7. Avoid Synchronized Collections for Single-Threaded Scenarios
Optimization: Use non-synchronized collections like ArrayList instead of Vector in single-threaded scenarios.

Before:
```
Vector<String> list = new Vector<>();
```
After:
```
ArrayList<String> list = new ArrayList<>();
```

8. Close Resources Properly
Optimization: Use try-with-resources to ensure resources are closed automatically.

Before:
```
BufferedReader reader = null;
try {
    reader = new BufferedReader(new FileReader("file.txt"));
    String line = reader.readLine();
} finally {
    if (reader != null) {
        reader.close();
    }
}
```
After:
```
try (BufferedReader reader = new BufferedReader(new FileReader("file.txt"))) {
    String line = reader.readLine();
}
```

9. Use Parallel Streams for Large Data Sets
Optimization: Process large collections faster using parallel streams.

Before:
```
list.stream().forEach(System.out::println);
```
After:
```
list.parallelStream().forEach(System.out::println);
```

10. Use final for Constants and Variables
Optimization: Declare variables as final when they are not meant to change, improving readability and allowing certain optimizations.

Before:
```
int maxLimit = 100;
```
After:
```
final int MAX_LIMIT = 100;
```

11. Use Primitives Instead of Wrapper Classes

Optimization: Prefer primitive data types (int, double) over wrappers (Integer, Double) to avoid unnecessary memory overhead.

Before:
```
Integer count = 5;
```
After:
```
int count = 5;
```
12. Avoid Redundant if Conditions
Optimization: Simplify redundant or unnecessary if statements.
Before:
```
if (isActive == true) {
    performAction();
}
```
After:
```
if (isActive) {
    performAction();
}
```

13. Use Efficient Collections
Optimization: Choose the right collection based on use case (e.g., HashMap for fast lookups, ArrayList for sequential access).
Example:
```
Map<String, Integer> map = new HashMap<>(); // For fast key-value lookups
List<String> list = new ArrayList<>(); // For sequential access
```

14. Avoid Redundant Object Casting
Optimization: Use generic types to avoid unnecessary casting.
Before:
```
List list = new ArrayList();
list.add("test");
String value = (String) list.get(0);
```
After:
```
List<String> list = new ArrayList<>();
list.add("test");
String value = list.get(0);
```

Problem Statement

Perception That Automation Tests Are "Less Critical"

- **Reason:** Teams may prioritize code reviews for application code over test code, assuming automation tests are less important.
- **Consequence:** Poorly written test scripts lead to flaky tests, false positives/negatives, and a loss of trust in the test suite.

2️⃣ Lack of Awareness About Best Practices

- **Reason:** Teams may not fully understand the value of clean, maintainable automation code or the impact of skipping reviews.
- **Consequence:** Automation code becomes hard to maintain, debug, or scale, especially in large projects.

3️⃣ Time Constraints

- **Reason:** Teams often face tight deadlines and skip reviewing automation scripts to save time.
- **Consequence:** Short-term gains lead to long-term losses as unreviewed scripts require frequent fixes and rework.

4️⃣ Lack of Ownership or Accountability

- **Reason:** Automation code may be seen as the QA team's responsibility alone, with little involvement from developers or other stakeholders.
- **Consequence:** Lack of collaboration results in inconsistent coding practices and overlooked issues.

5️⃣ No Defined Standards for Test Code

- **Reason:** Teams may not have well-established coding standards for automation scripts.
- **Consequence:** Inconsistent test scripts make it harder to maintain and share across teams.

6️⃣ Tools and Frameworks Are Misunderstood as "Perfect"

- **Reason:** Some believe that test automation frameworks/tools inherently enforce good practices.
- **Consequence:** Poorly implemented tests cause flaky results, regardless of the tool’s capabilities.

7️⃣ Lack of Skilled Reviewers

- **Reason:** Team members may lack the expertise to review automation test scripts effectively.
- **Consequence:** Potential issues in the test code remain undetected, leading to unreliable test coverage.

8️⃣ Tests Are Written in Isolation

- **Reason:** Automation engineers may work in silos, leading to less collaboration and fewer opportunities for peer review.
- **Consequence:** Missed opportunities for shared learning and improvement.

9️⃣ "It Works" Mentality

- **Reason:** If tests pass, they are often accepted as sufficient without reviewing the underlying code quality.
- **Consequence:** Fragile and inefficient tests that fail when application changes occur.

🔟 Focus on Quantity Over Quality

- **Reason:** Teams may prioritize the number of automated test cases written over their quality or reliability.
- **Consequence:** Test suites become bloated with redundant or low-value scripts.

How to Address This?

- **Raise Awareness:** Educate teams on the long-term benefits of reviewing test automation code.
- **Standardize Practices:** Define and enforce coding standards for test scripts.
- **Allocate Time:** Incorporate code reviews into the sprint cycle for both application and test code.
- **Leverage Tools:** Use tools like GitHub, Bitbucket, or Amazon Q Developer to streamline the code review process.
- **Collaborate:** Involve developers and QA engineers in reviewing test scripts for better alignment.
- **Train Reviewers:** Provide training on automation best practices and frameworks.