MHP

A PORSCHE COMPANY

ADVANCED PROGRAMMING METHODS

CLEAN CODE. TESTING IN JAVA

SERGIU LIMBOI, ANDREA IORDACHE, DARIUS FLOREA

29.10.2025

AGENDA

- Clean code - introduction
- Clean code principles
- Other best practices
- Why do we need testing?
- Unit testing
- Integration testing
- Q & A

MHP
A PORSCHE COMPANY

Clean code - Introduction

What is Clean Code?

"Anybody can write code that a computer can understand.
Good programmers write code that humans can understand.“
(Martin Fowler)

What is Clean Code?

- *Clean Code: A Handbook of Agile Software Craftsmanship* – Robert C. Martin (2008)

- Code should be easy to:
  - read
  - understand
  - maintain

Before Clean Code

- DRY principle – “Don’t repeat yourself!”
- KISS principle – “Keep it simple, stupid!”
- GRASP – “General Responsibility Assignment Software Patterns”
  (e.g., high cohesion, low coupling, polymorphism, etc.)
- SOLID principles

Before Clean Code

- SOLID principles

Before Clean Code

- SOLID principles
  - **S**ingle responsibility principle

Before Clean Code

- SOLID principles
  - **S**ingle responsibility principle
  - **O**pen-closed principle

Before Clean Code

- SOLID principles
  - ➢ **S**ingle responsibility principle
  - ➢ **O**pen-closed principle
  - ➢ **L**iskov substitution principle

Before Clean Code

- SOLID principles
  - **S**ingle responsibility principle
  - **O**pen-closed principle
  - **L**iskov substitution principle
  - **I**nterface segregation principle

Before Clean Code

- SOLID principles
  - **S**ingle responsibility principle
  - **O**pen-closed principle
  - **L**iskov substitution principle
  - **I**nterface segregation principle
  - **D**ependency inversion principle

What are the benefits of Clean Code?

- Improved readability
- Easier maintenance
- Better team collaboration
- Increased quality and reliability
- Debugging and issue resolution

I/I MHP
A PORSCHE COMPANY

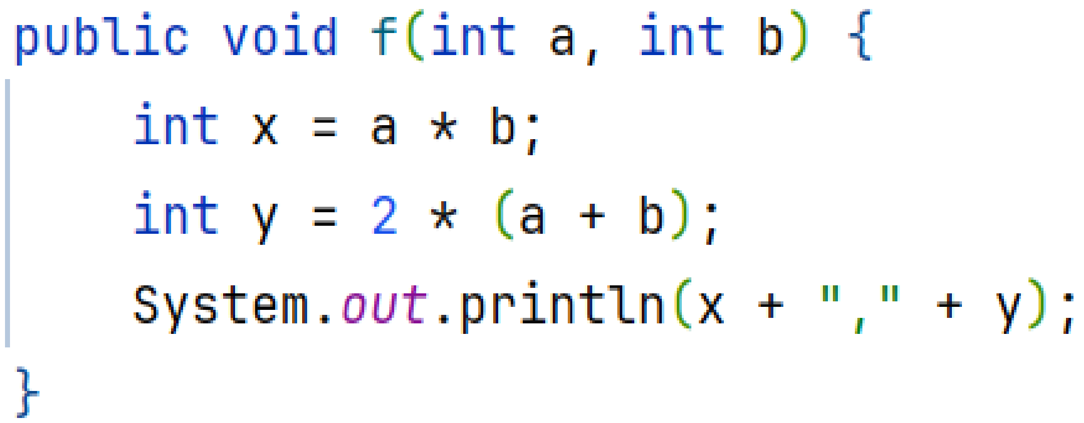
Clean code principles

Naming

- Use intention-revealing names
- Avoid mental mapping

Before

After

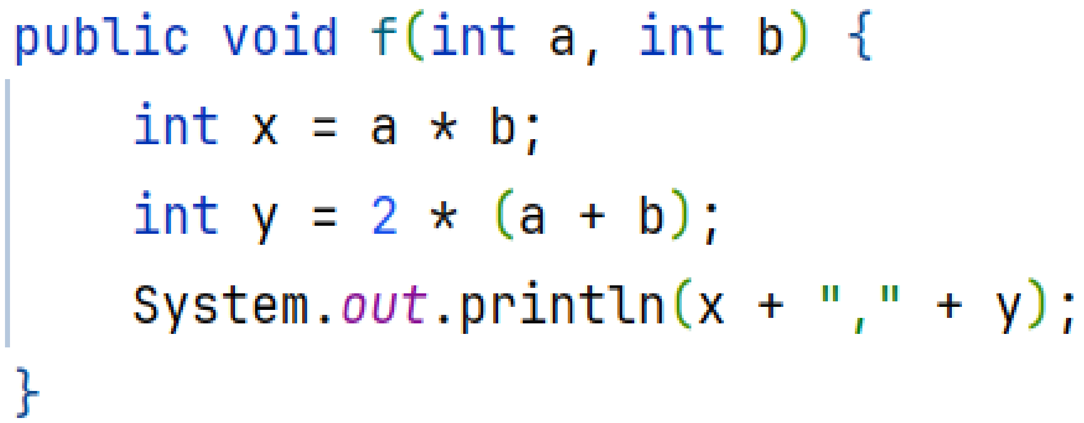


```java
public void f(int a, int b) {
    int x = a * b;
    int y = 2 * (a + b);
    System.out.println(x + "," + y);
}
```

Naming

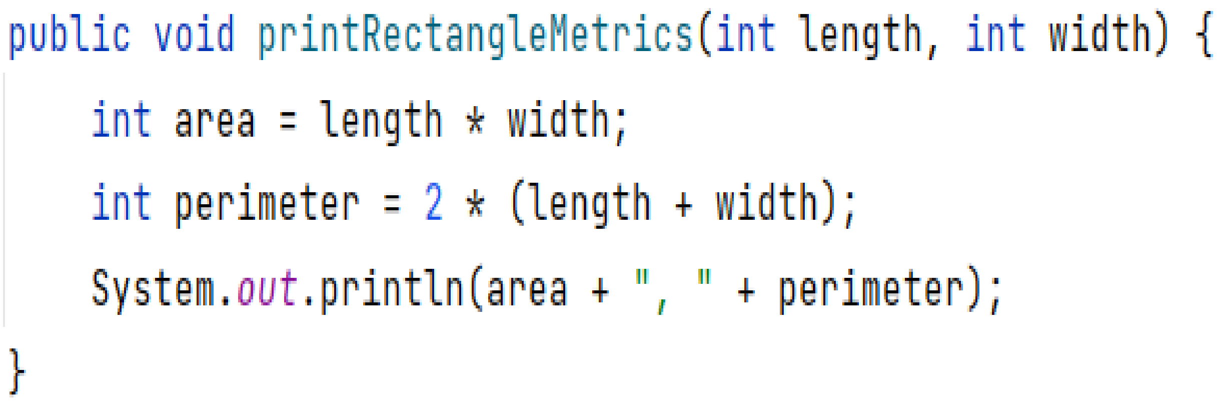
- Use intention-revealing names
- Avoid mental mapping

Before



```java
public void f(int a, int b) {
    int x = a * b;
    int y = 2 * (a + b);
    System.out.println(x + "," + y);
}
```

After

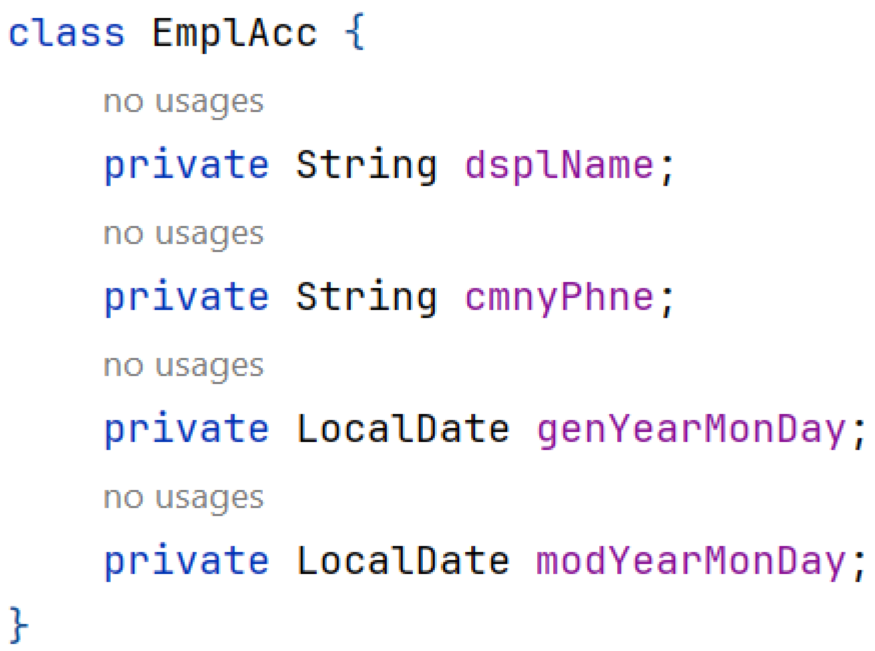


```java
public void printRectangleMetrics(int length, int width) {
    int area = length * width;
    int perimeter = 2 * (length + width);
    System.out.println(area + ", " + perimeter);
}
```

Naming

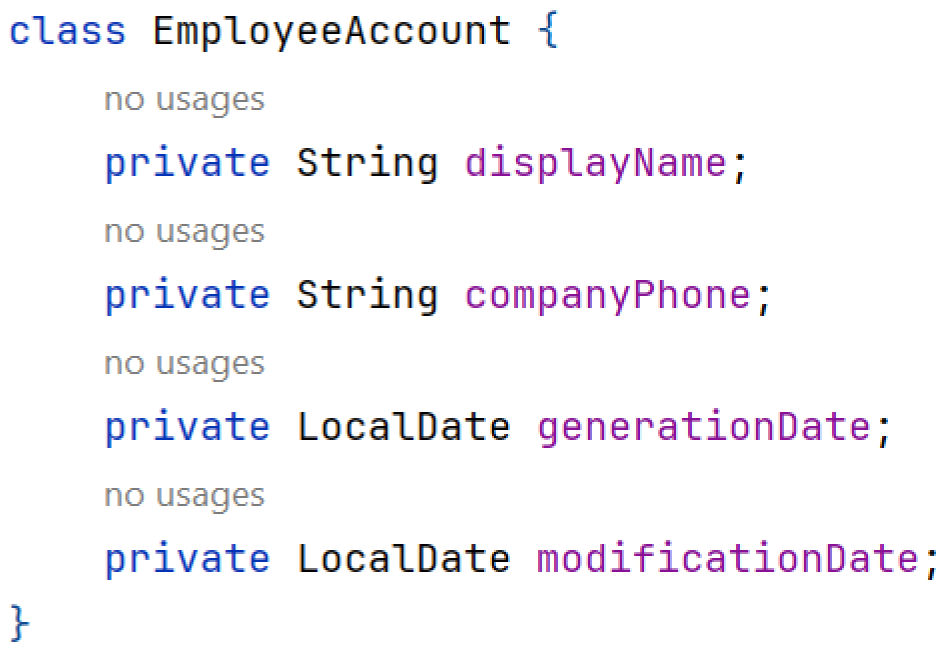
- Avoid ambiguous abbreviations

Before



```
class EmplAcc {
    no usages
    private String dsplName;
    no usages
    private String cmnyPhne;
    no usages
    private LocalDate genYearMonDay;
    no usages
    private LocalDate modYearMonDay;
}
```

After



```
class EmployeeAccount {
    no usages
    private String displayName;
    no usages
    private String companyPhone;
    no usages
    private LocalDate generationDate;
    no usages
    private LocalDate modificationDate;
}
```

Naming

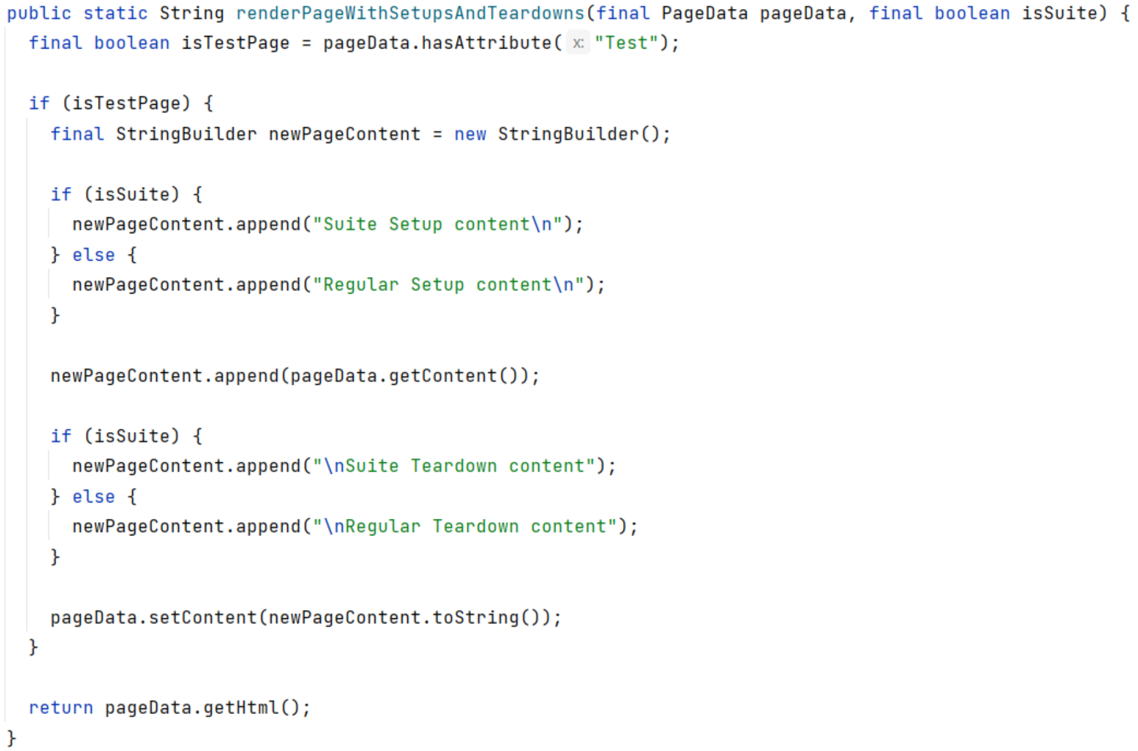
- Class names:
  - nouns or noun phrases
  - e.g., *Student*, *Address*, *Customer*, *AddressParser*

- Method names:
  - verb or verb phrases
  - e.g., save, *processPayment*, *deleteAccount*
  - for accessors, mutators, predicates: *get*, *set*, *is*

Functions

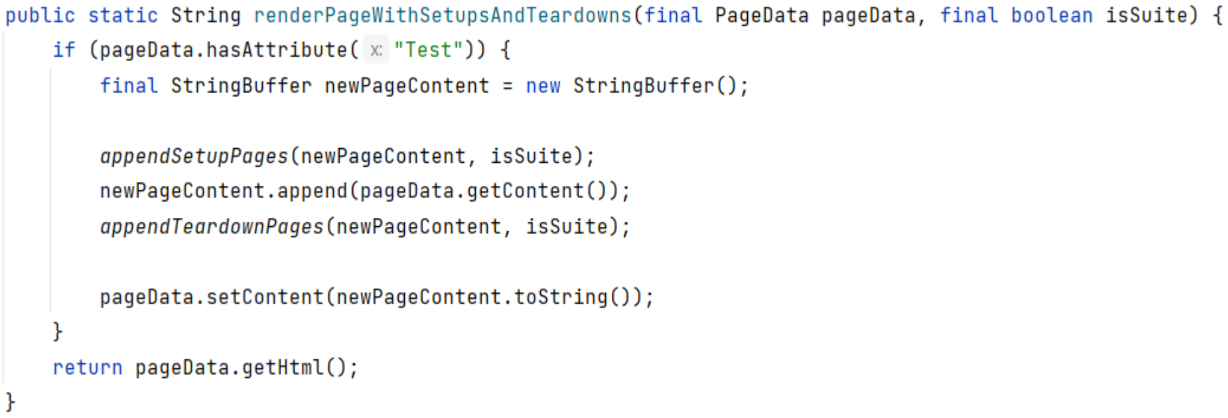
- Small
  - ➢ should hardly be 20 lines long

Before



```java
public static String renderPageWithSetupsAndTeardowns(final PageData pageData, final boolean isSuite) {
  final boolean isTestPage = pageData.hasAttribute( x: "Test");

  if (isTestPage) {
    final StringBuilder newPageContent = new StringBuilder();

    if (isSuite) {
      newPageContent.append("Suite Setup content\n");
    } else {
      newPageContent.append("Regular Setup content\n");
    }

    newPageContent.append(pageData.getContent());

    if (isSuite) {
      newPageContent.append("\nSuite Teardown content");
    } else {
      newPageContent.append("\nRegular Teardown content");
    }

    pageData.setContent(newPageContent.toString());
  }

  return pageData.getHtml();
}
```

After

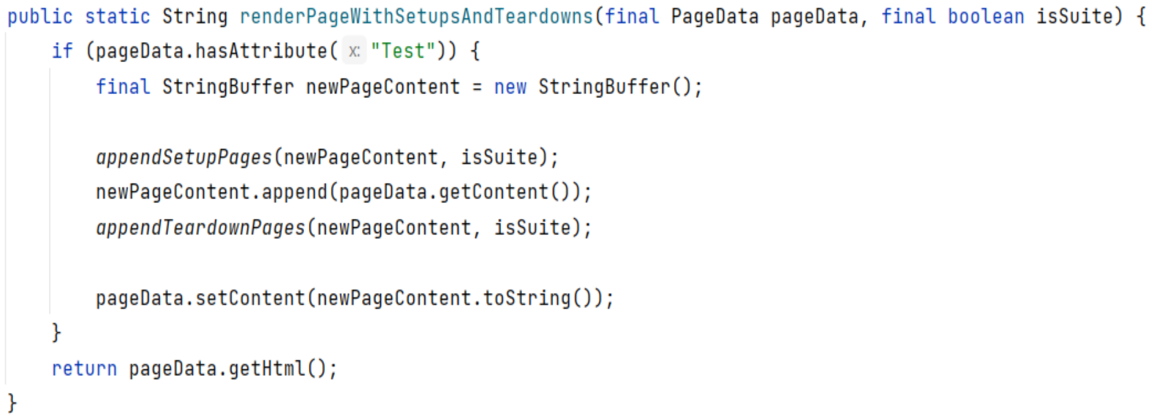


```java
public static String renderPageWithSetupsAndTeardowns(final PageData pageData, final boolean isSuite) {
    if (pageData.hasAttribute( x: "Test")) {
        final StringBuffer newPageContent = new StringBuffer();

        appendSetupPages(newPageContent, isSuite);
        newPageContent.append(pageData.getContent());
        appendTeardownPages(newPageContent, isSuite);

        pageData.setContent(newPageContent.toString());
    }
    return pageData.getHtml();
}
```

Functions

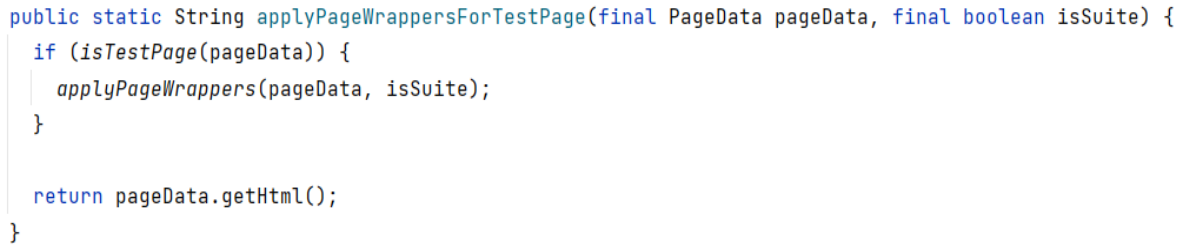
- Do one thing
  - ➢ contains steps that are one level of abstraction below the stated name of the function
  - ➢ if you can extract another function from it with a name that is not merely a restatement of its implementation, the function is doing too much

Before



```java
public static String renderPageWithSetupsAndTeardowns(final PageData pageData, final boolean isSuite) {
    if (pageData.hasAttribute( x: "Test")) {
        final StringBuffer newPageContent = new StringBuffer();

        appendSetupPages(newPageContent, isSuite);
        newPageContent.append(pageData.getContent());
        appendTeardownPages(newPageContent, isSuite);

        pageData.setContent(newPageContent.toString());
    }
    return pageData.getHtml();
}
```

After



```java
public static String applyPageWrappersForTestPage(final PageData pageData, final boolean isSuite) {
  if (isTestPage(pageData)) {
    applyPageWrappers(pageData, isSuite);
  }

  return pageData.getHtml();
}
```

Functions

- Descriptive names
  - smaller & more focused functions => easier to choose a descriptive name
  - long descriptive names are better than long descriptive comments

- Function arguments
  - ideal number of arguments is 0
  - next comes 1, 2 and 3 (should already be avoided where possible)
  - more than 3 arguments require special justification and should not be used anyway

Comments

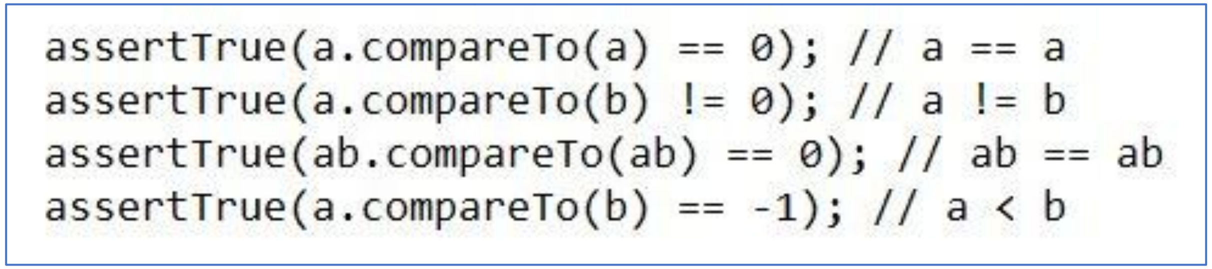
- Comments are, at best, a necessary evil
- Used to compensate for our failure in expressing our intentions through code
- Comments often lie
- Inaccurate comments are worse than no comments at all
- The only source of accurate information: the code
- Comments do not make up for bad code

Comments – Good comments

- Legal comments
  - copyright and authorship statements

- Explanation of intent
  - intent behind a decision, not just useful information about the implementation

- Clarification
  - translating the meaning of some obscure arguments / return values

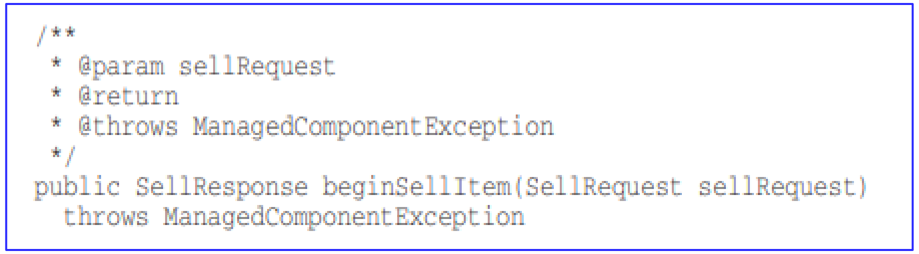


```
assertTrue(a.compareTo(a) == 0); // a == a
assertTrue(a.compareTo(b) != 0); // a != b
assertTrue(ab.compareTo(ab) == 0); // ab == ab
assertTrue(a.compareTo(b) == -1); // a < b
```

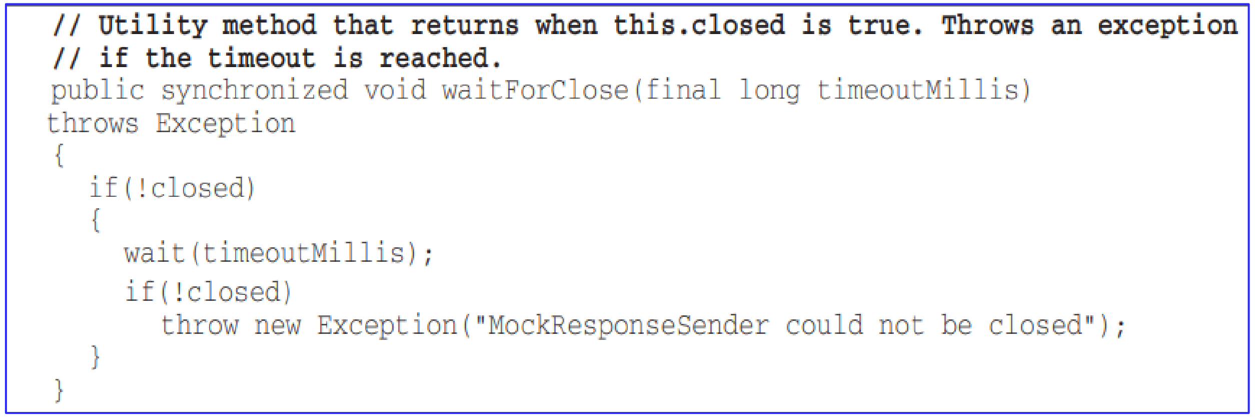
- TODO comments
- Javadocs in Public APIs

Comments – Bad Comments

- Obsolete comments
  - ➢ Comments that have gotten old, irrelevant and incorrect
- Commented-Out Code
- Redundant Comments
  - ➢ Comments that are nothing but noise
  - ➢ Comments that restate the obvious and provide no new information



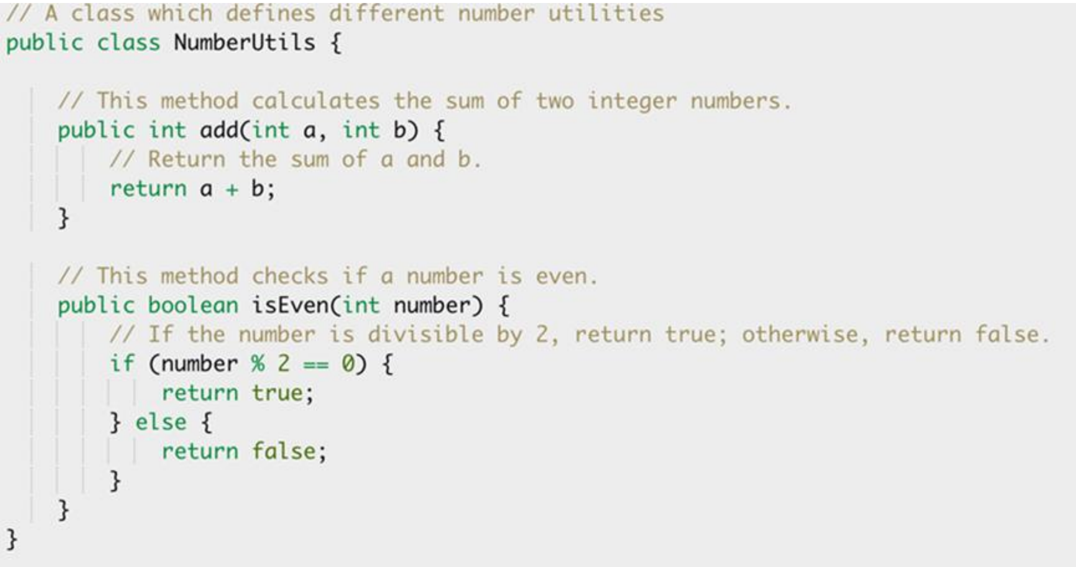
```
/**
 * @param sellRequest
 * @return
 * @throws ManagedComponentException
 */
public SellResponse beginSellItem(SellRequest sellRequest)
  throws ManagedComponentException
```



```
// Utility method that returns when this.closed is true. Throws an exception
// if the timeout is reached.
public synchronized void waitForClose(final long timeoutMillis)
throws Exception
{
  if(!closed)
  {
    wait(timeoutMillis);
    if(!closed)
      throw new Exception("MockResponseSender could not be closed");
  }
}
```

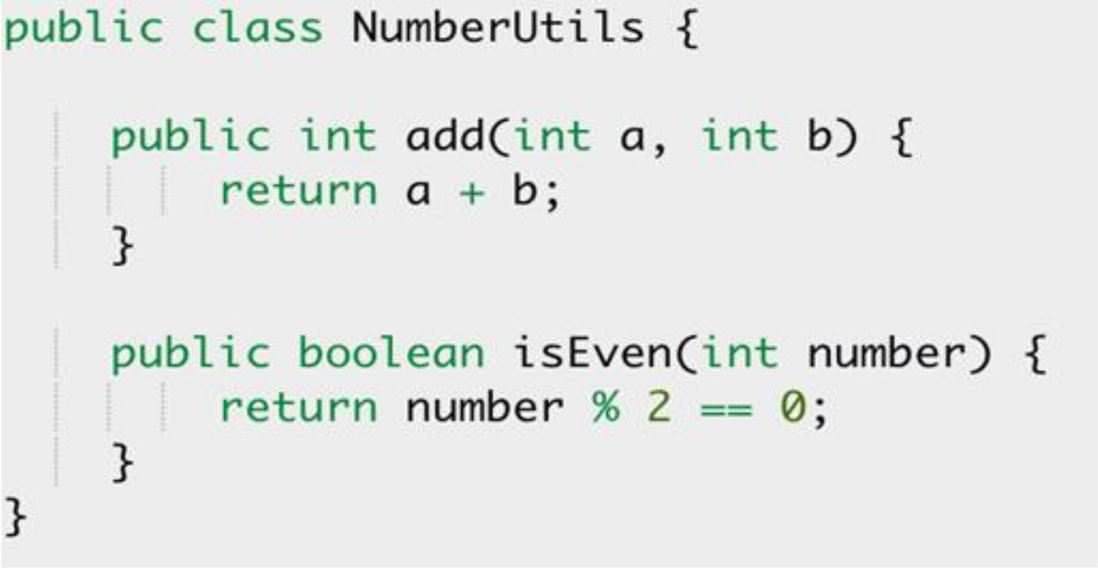
Comments

Before



```java
// A class which defines different number utilities
public class NumberUtils {

    // This method calculates the sum of two integer numbers.
    public int add(int a, int b) {
        // Return the sum of a and b.
        return a + b;
    }

    // This method checks if a number is even.
    public boolean isEven(int number) {
        // If the number is divisible by 2, return true; otherwise, return false.
        if (number % 2 == 0) {
            return true;
        } else {
            return false;
        }
    }
}
```

After



```java
public class NumberUtils {

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

    public boolean isEven(int number) {
        return number % 2 == 0;
    }
}
```

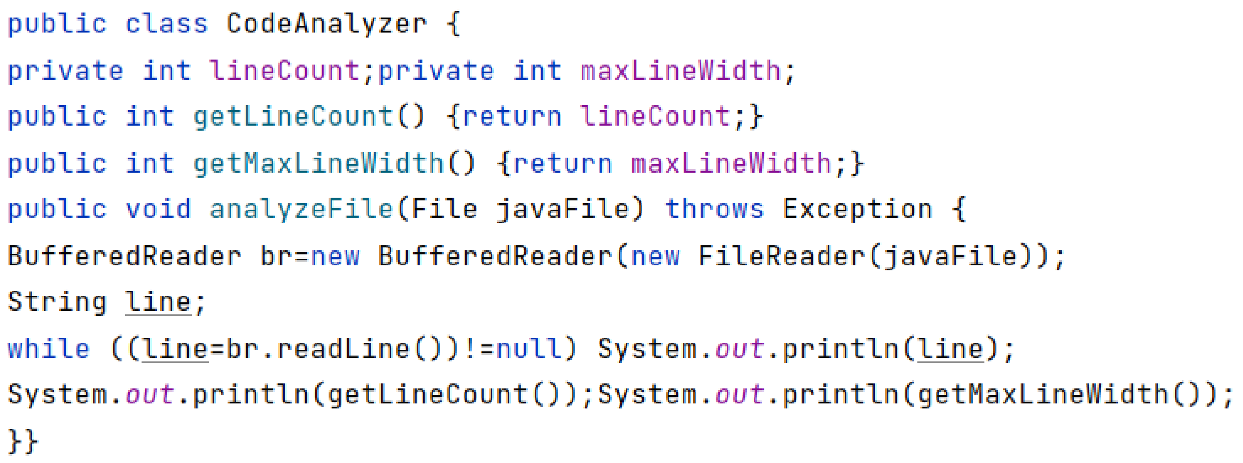
Formatting

- Vertical formatting
  - blank lines to separate logical sections of your code
  - variables declared as close to their usage as possible
  - function call dependencies pointing in the downward direction

- Horizontal formatting
  - short lines
  - white spaces around operators, parenthesis, brackets, etc.
  - consistent indentation

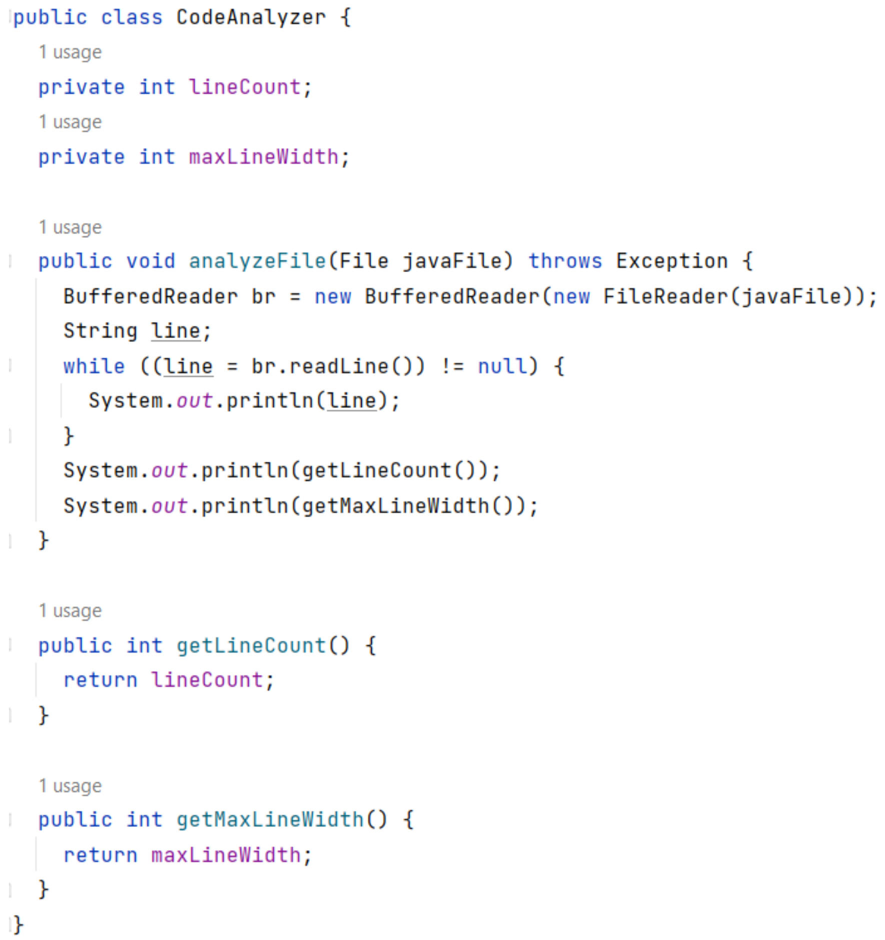
Formatting

Before



```
public class CodeAnalyzer {
private int lineCount;private int maxLineWidth;
public int getLineCount() {return lineCount;}
public int getMaxLineWidth() {return maxLineWidth;}
public void analyzeFile(File javaFile) throws Exception {
BufferedReader br=new BufferedReader(new FileReader(javaFile));
String line;
while ((line=br.readLine())!=null) System.out.println(line);
System.out.println(getLineCount());System.out.println(getMaxLineWidth());
}}
```

After



```
public class CodeAnalyzer {
  1 usage
  private int lineCount;
  1 usage
  private int maxLineWidth;

  1 usage
  public void analyzeFile(File javaFile) throws Exception {
    BufferedReader br = new BufferedReader(new FileReader(javaFile));
    String line;
    while ((line = br.readLine()) != null) {
      System.out.println(line);
    }
    System.out.println(getLineCount());
    System.out.println(getMaxLineWidth());
  }

  1 usage
  public int getLineCount() {
    return lineCount;
  }

  1 usage
  public int getMaxLineWidth() {
    return maxLineWidth;
  }
}
```

Objects and Data Structures

- Encapsulation
- Abstraction
- Data Transfer Objects (DTOs)
- The Law of Demeter (principle of least knowledge)

Objects and Data Structures

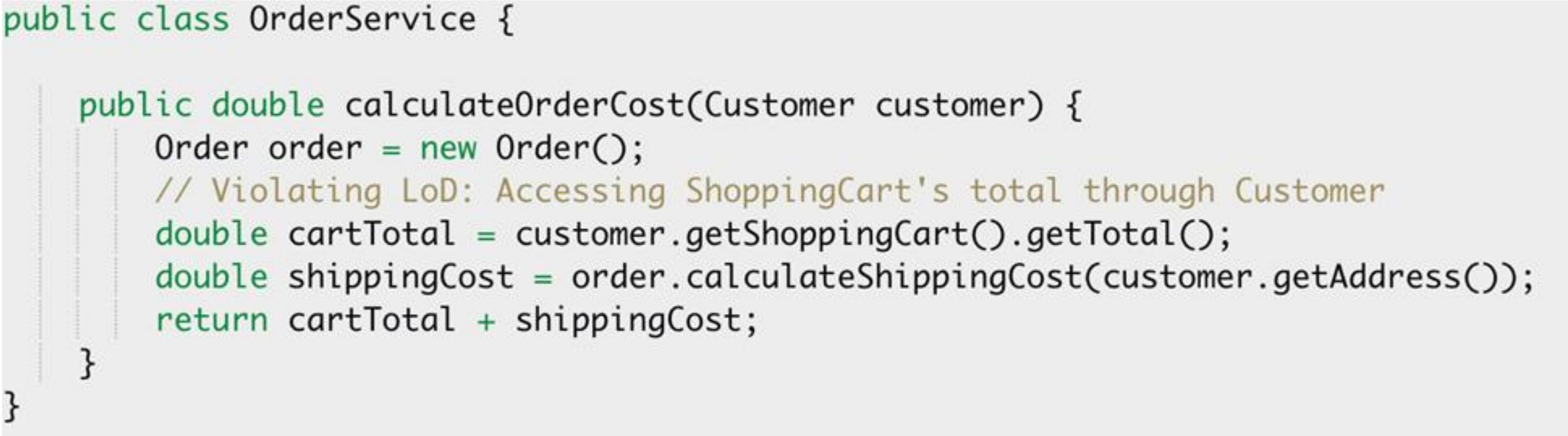
- The Law of Demeter

A method *f* of a class *C* should only use methods of:

- *C*
- an object held in an instance variable of *C*
- an object created by *f*
- an object passed as argument to *f*

Objects and Data Structures

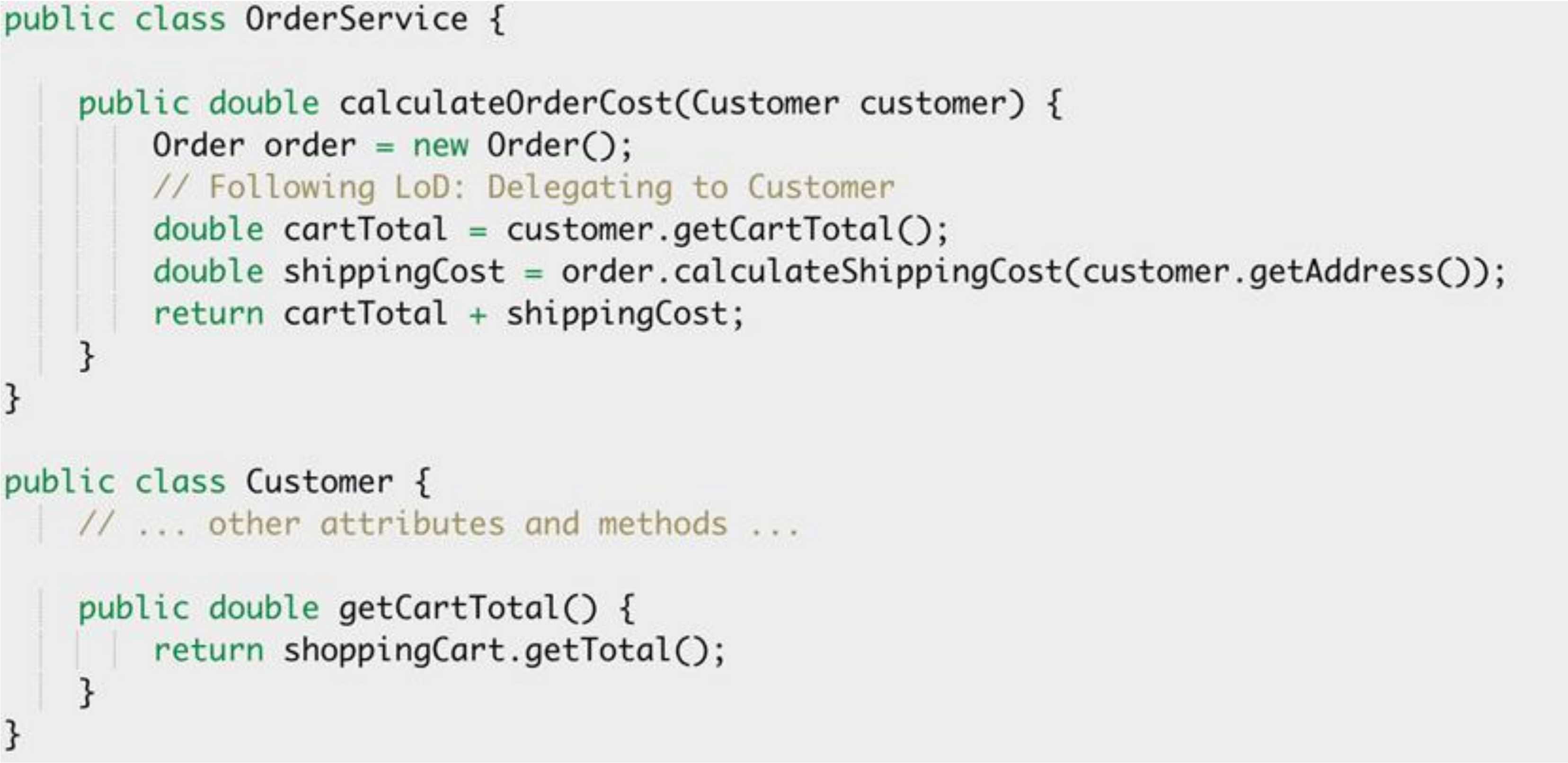
- The Law of Demeter



```
public class OrderService {

    public double calculateOrderCost(Customer customer) {
        Order order = new Order();
        // Violating LoD: Accessing ShoppingCart's total through Customer
        double cartTotal = customer.getShoppingCart().getTotal();
        double shippingCost = order.calculateShippingCost(customer.getAddress());
        return cartTotal + shippingCost;
    }
}
```

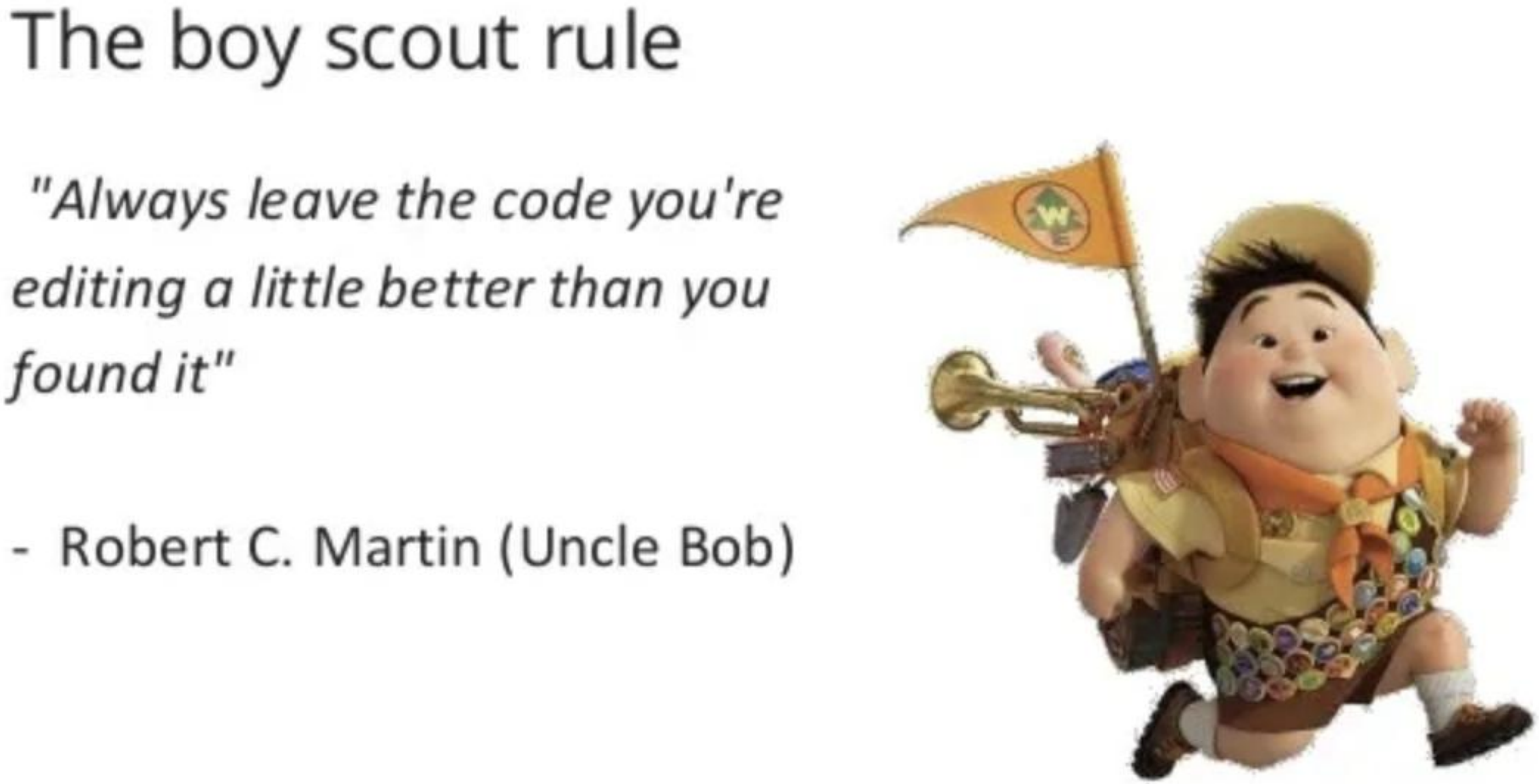
Objects and Data Structures

- The Law of Demeter



```
public class OrderService {

    public double calculateOrderCost(Customer customer) {
        Order order = new Order();
        // Following LoD: Delegating to Customer
        double cartTotal = customer.getCartTotal();
        double shippingCost = order.calculateShippingCost(customer.getAddress());
        return cartTotal + shippingCost;
    }
}

public class Customer {
    // ... other attributes and methods ...

    public double getCartTotal() {
        return shoppingCart.getTotal();
    }
}
```

Other best practices



The boy scout rule

"Always leave the code you're editing a little better than you found it"

- Robert C. Martin (Uncle Bob)

Other Best Practices

Before

- import java.util.*;
- Import java.awt.*;

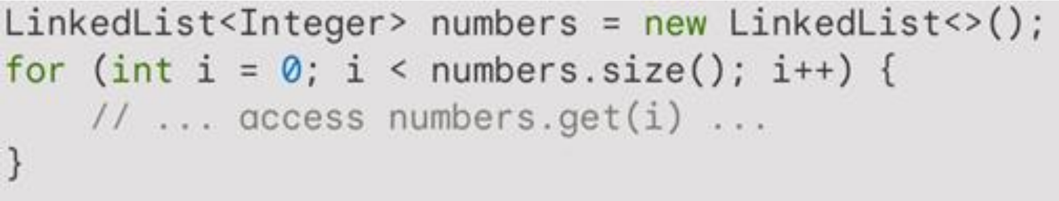
After

- Import java.util.list;
- //other imports

Other Best Practices

Before

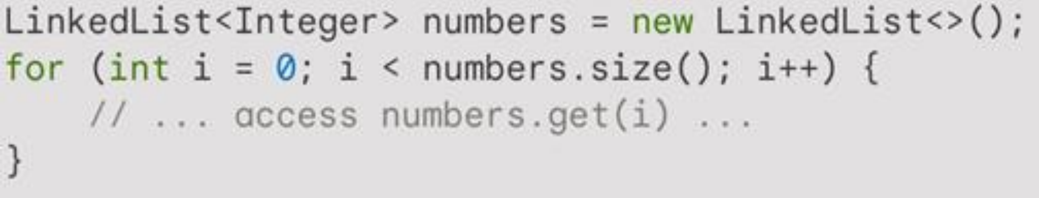
After



```
LinkedList<Integer> numbers = new LinkedList<>();
for (int i = 0; i < numbers.size(); i++) {
    // ... access numbers.get(i) ...
}
```

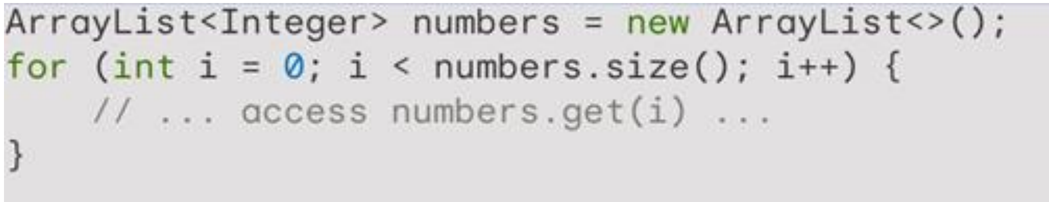
Other Best Practices

Before



```
LinkedList<Integer> numbers = new LinkedList<>();
for (int i = 0; i < numbers.size(); i++) {
    // ... access numbers.get(i) ...
}
```

After

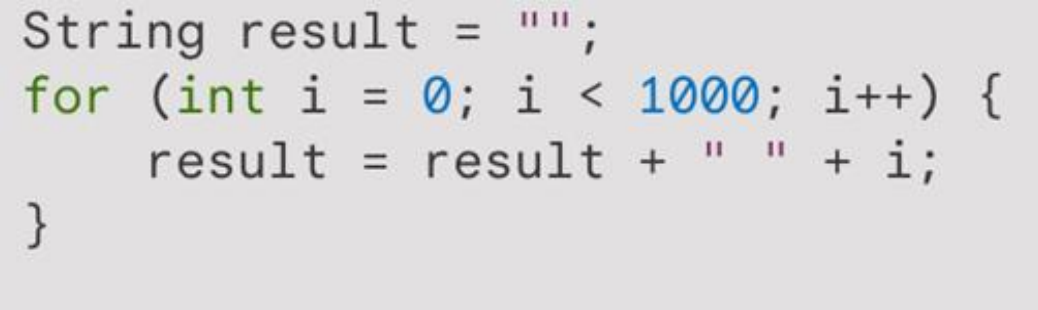


```
ArrayList<Integer> numbers = new ArrayList<>();
for (int i = 0; i < numbers.size(); i++) {
    // ... access numbers.get(i) ...
}
```

Other Best Practices

Before

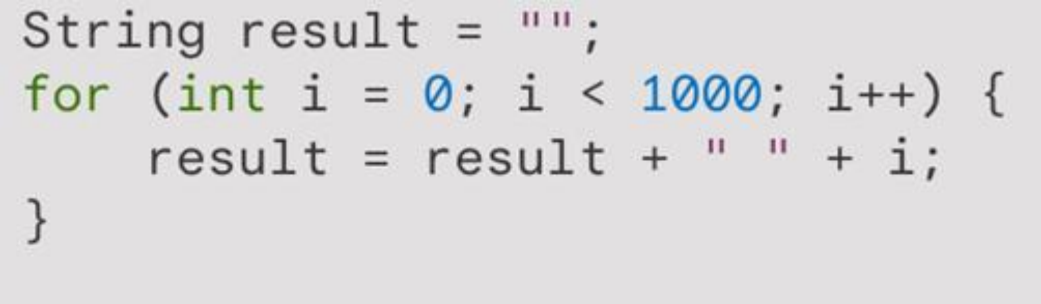
After



```
String result = "";
for (int i = 0; i < 1000; i++) {
    result = result + " " + i;
}
```

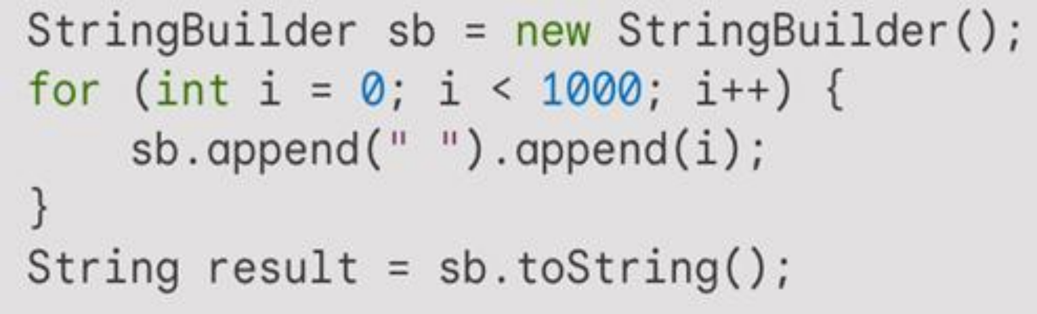
Other Best Practices

Before



```
String result = "";
for (int i = 0; i < 1000; i++) {
    result = result + " " + i;
}
```

After

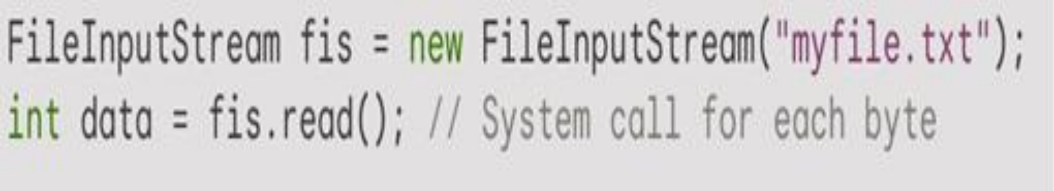


```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 1000; i++) {
    sb.append(" ").append(i);
}
String result = sb.toString();
```

Other Best Practices

Before

After

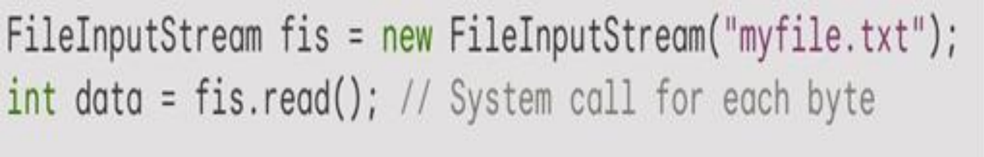


```
FileInputStream fis = new FileInputStream("myfile.txt");
int data = fis.read(); // System call for each byte
```

- **Without buffering:** If you read the file one byte at a time using `FileInputStream`, you would make 1,048,576 system calls!

Other Best Practices

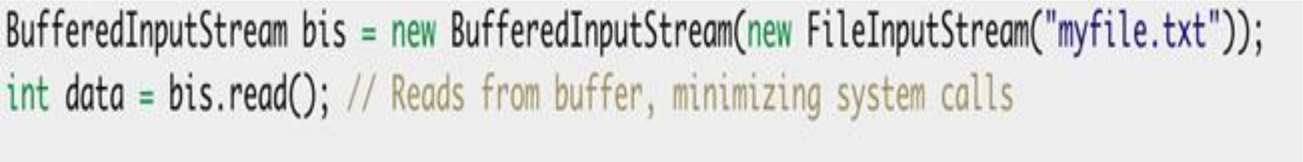
Before



```java
FileInputStream fis = new FileInputStream("myfile.txt");
int data = fis.read(); // System call for each byte
```

- **Without buffering:** If you read the file one byte at a time using `FileInputStream`, you would make 1,048,576 system calls!

After



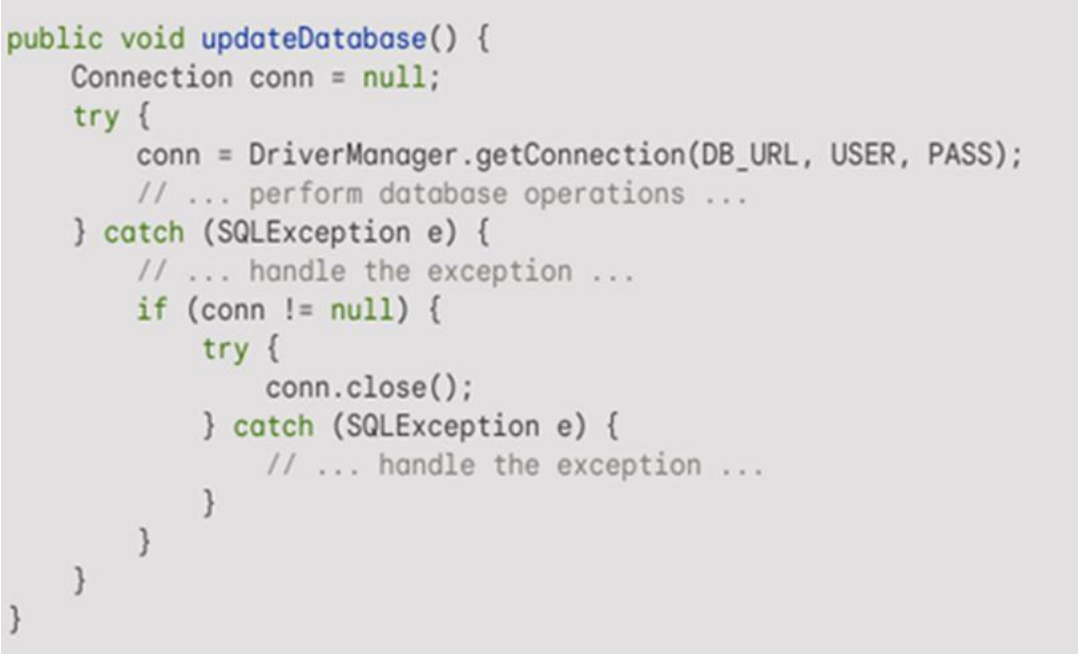
```java
BufferedInputStream bis = new BufferedInputStream(new FileInputStream("myfile.txt"));
int data = bis.read(); // Reads from buffer, minimizing system calls
```

- **With buffering:** Using `BufferedInputStream` with a default buffer size of 8KB, you would make approximately 128 system calls (1MB / 8KB).

Other Best Practices

Before

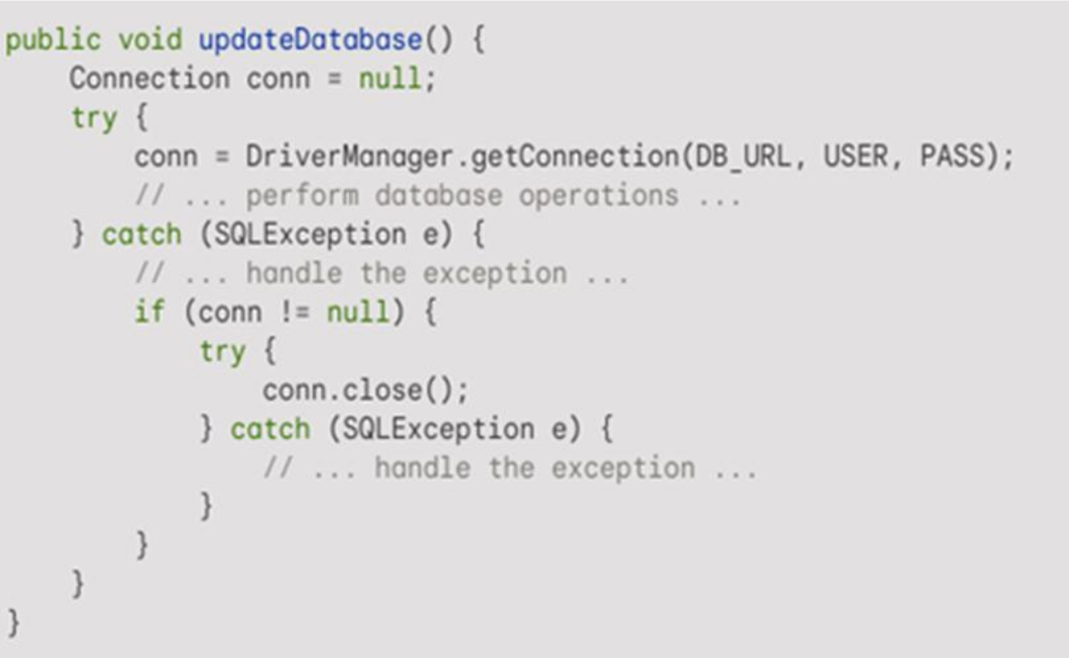
After



```java
public void updateDatabase() {
    Connection conn = null;
    try {
        conn = DriverManager.getConnection(DB_URL, USER, PASS);
        // ... perform database operations ...
    } catch (SQLException e) {
        // ... handle the exception ...
        if (conn != null) {
            try {
                conn.close();
            } catch (SQLException e) {
                // ... handle the exception ...
            }
        }
    }
}
```

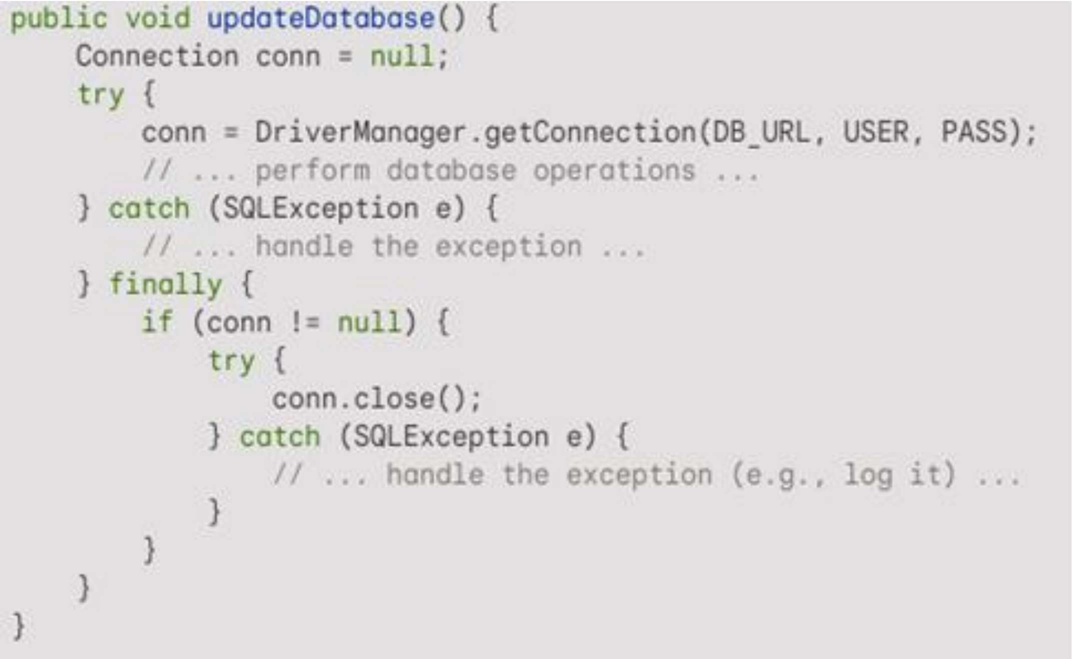
Other Best Practices

Before



```
public void updateDatabase() {
    Connection conn = null;
    try {
        conn = DriverManager.getConnection(DB_URL, USER, PASS);
        // ... perform database operations ...
    } catch (SQLException e) {
        // ... handle the exception ...
        if (conn != null) {
            try {
                conn.close();
            } catch (SQLException e) {
                // ... handle the exception ...
            }
        }
    }
}
```

After

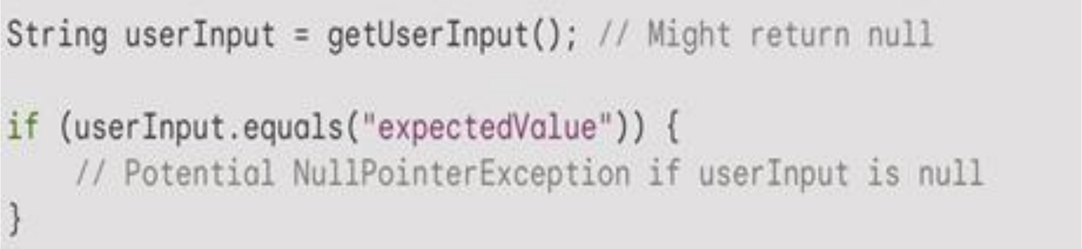


```
public void updateDatabase() {
    Connection conn = null;
    try {
        conn = DriverManager.getConnection(DB_URL, USER, PASS);
        // ... perform database operations ...
    } catch (SQLException e) {
        // ... handle the exception ...
    } finally {
        if (conn != null) {
            try {
                conn.close();
            } catch (SQLException e) {
                // ... handle the exception (e.g., log it) ...
            }
        }
    }
}
```

Other Best Practices

Before

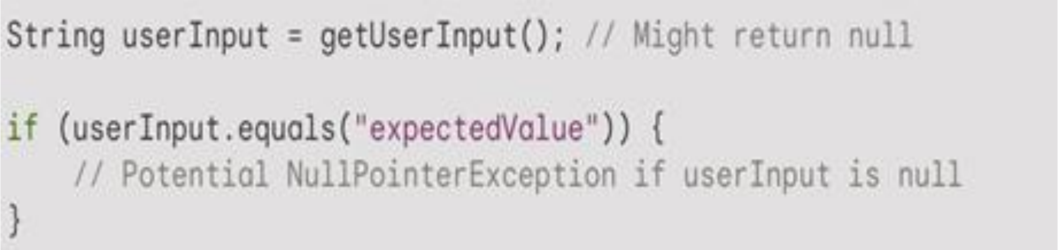
After



```
String userInput = getUserInput(); // Might return null

if (userInput.equals("expectedValue")) {
    // Potential NullPointerException if userInput is null
}
```

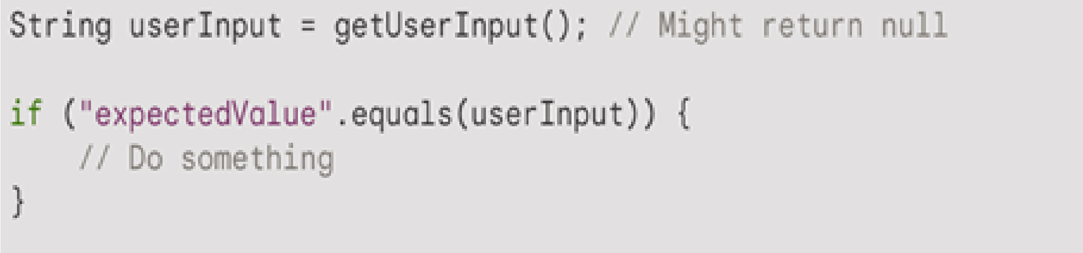
Other Best Practices

Before



```
String userInput = getUserInput(); // Might return null

if (userInput.equals("expectedValue")) {
    // Potential NullPointerException if userInput is null
}
```

After

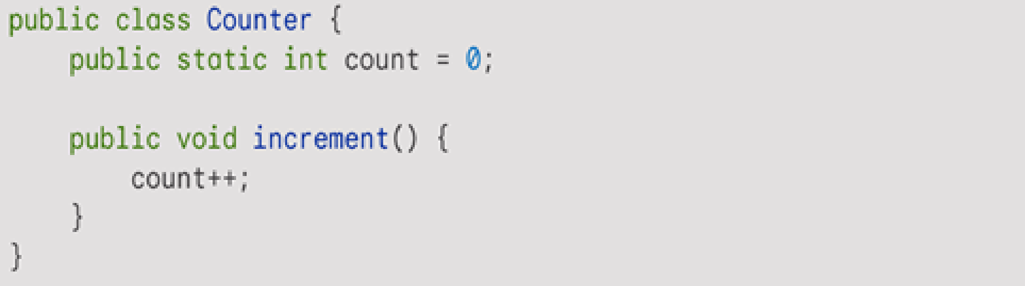


```
String userInput = getUserInput(); // Might return null

if ("expectedValue".equals(userInput)) {
    // Do something
}
```

Other Best Practices

Before

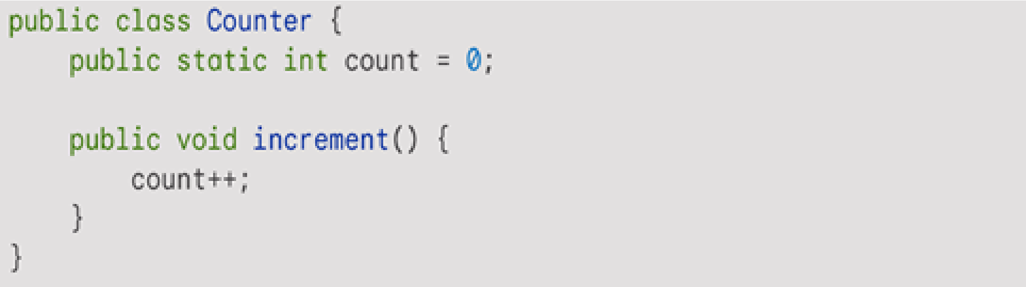
After



```java
public class Counter {
    public static int count = 0;

    public void increment() {
        count++;
    }
}
```

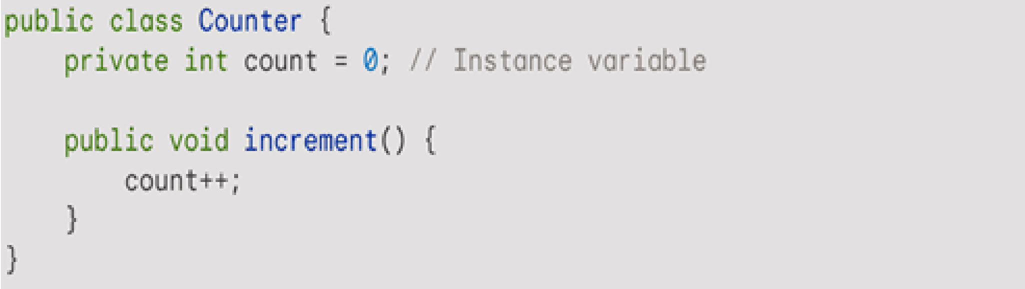
Other Best Practices

Before



```java
public class Counter {
    public static int count = 0;

    public void increment() {
        count++;
    }
}
```

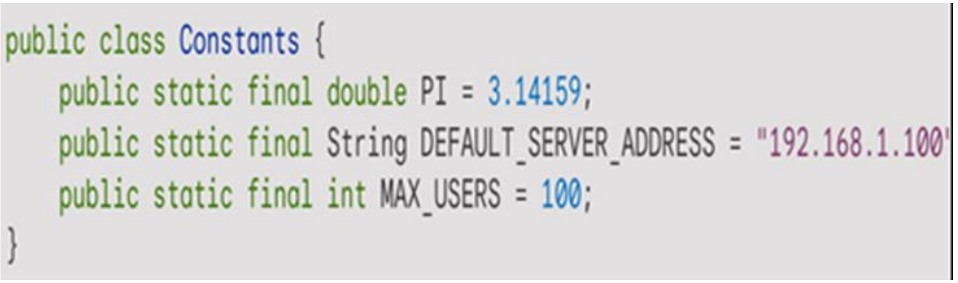
After



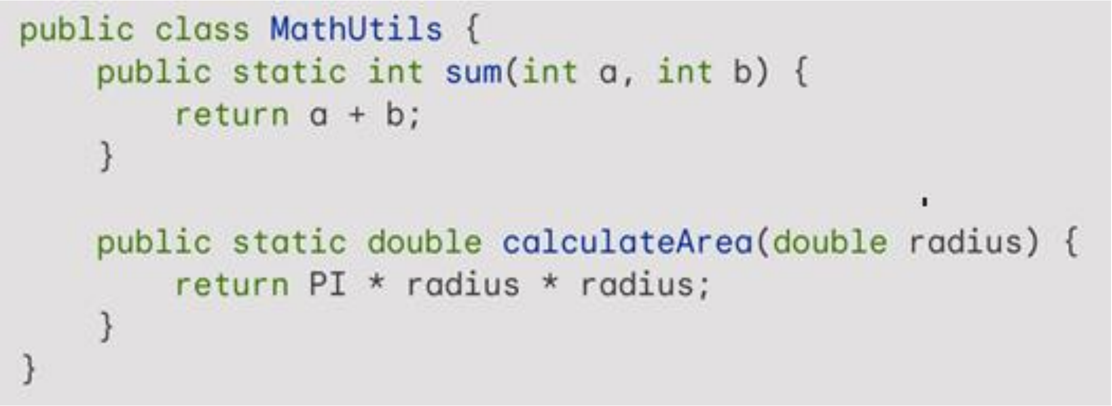
```java
public class Counter {
    private int count = 0; // Instance variable

    public void increment() {
        count++;
    }
}
```

Other Best Practices

When is static good?



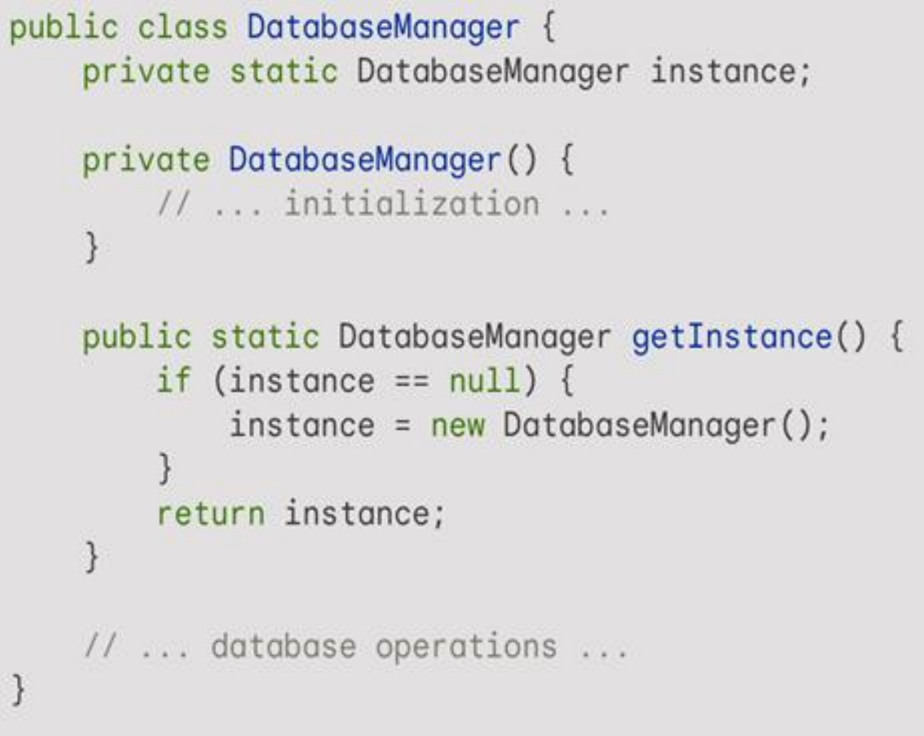
```
public class Constants {
    public static final double PI = 3.14159;
    public static final String DEFAULT_SERVER_ADDRESS = "192.168.1.100"
    public static final int MAX_USERS = 100;
}
```



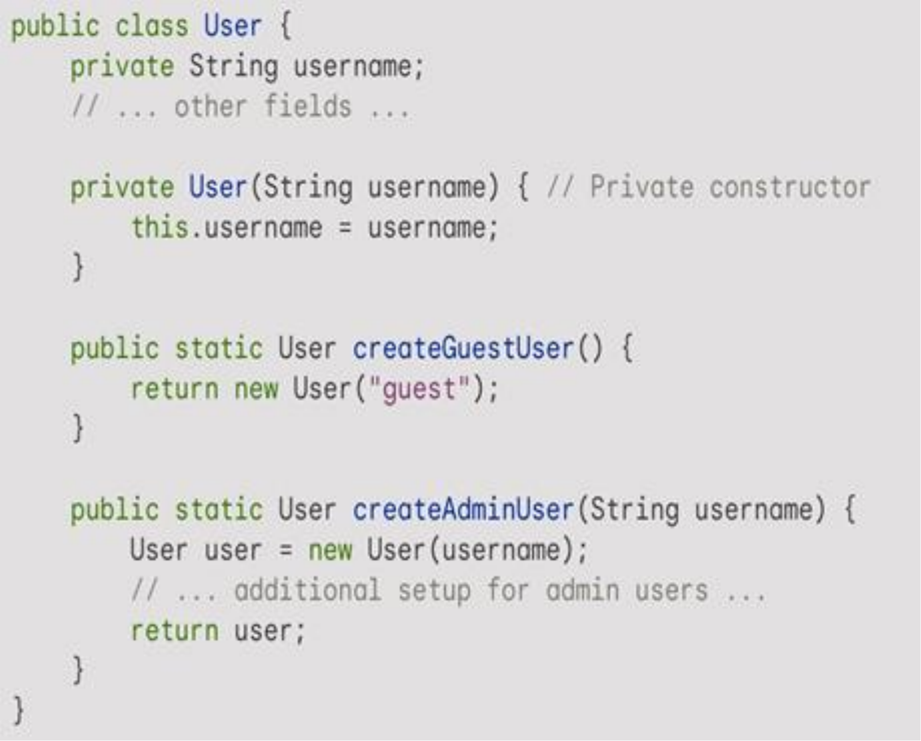
```
public class MathUtils {
    public static int sum(int a, int b) {
        return a + b;
    }

    public static double calculateArea(double radius) {
        return PI * radius * radius;
    }
}
```

Other Best Practices

When is static good?



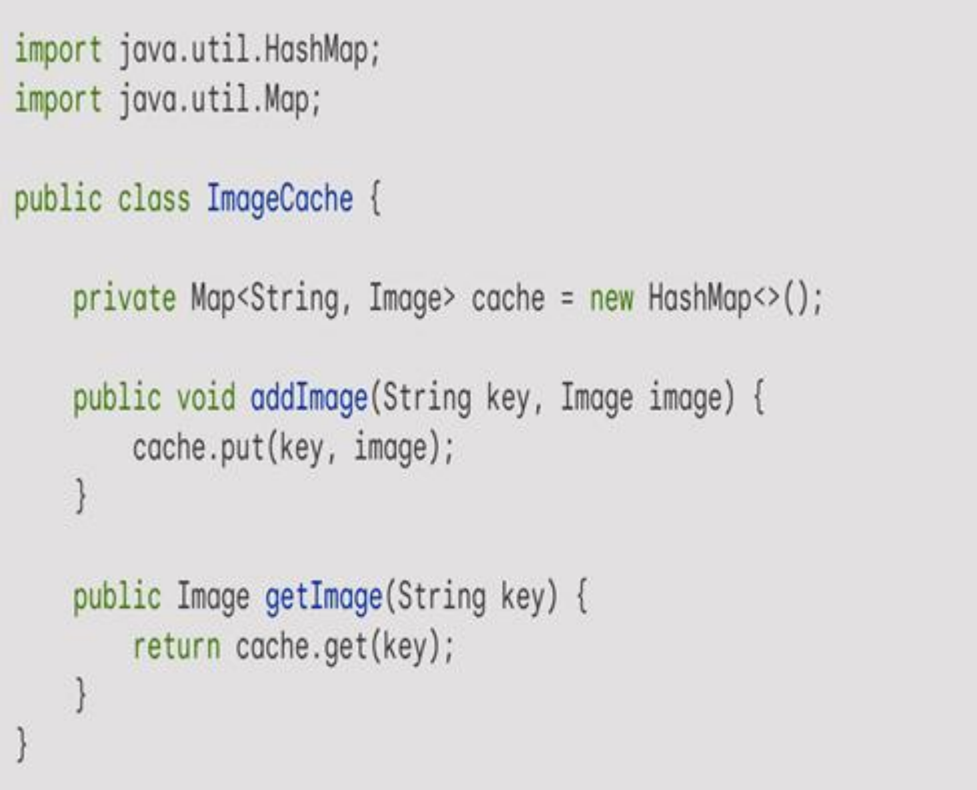
```
public class DatabaseManager {
    private static DatabaseManager instance;

    private DatabaseManager() {
        // ... initialization ...
    }

    public static DatabaseManager getInstance() {
        if (instance == null) {
            instance = new DatabaseManager();
        }
        return instance;
    }

    // ... database operations ...
}
```



```
public class User {
    private String username;
    // ... other fields ...

    private User(String username) { // Private constructor
        this.username = username;
    }

    public static User createGuestUser() {
        return new User("guest");
    }

    public static User createAdminUser(String username) {
        User user = new User(username);
        // ... additional setup for admin users ...
        return user;
    }
}
```

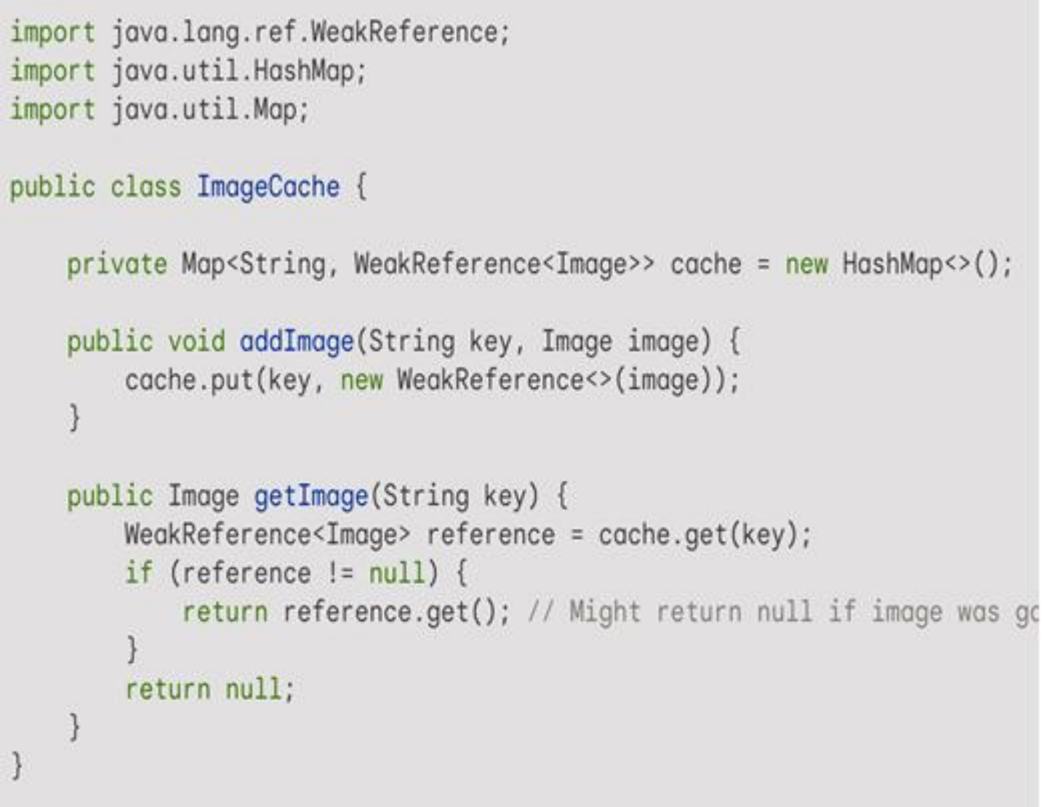
Other Best Practices

Before



```java
import java.util.HashMap;
import java.util.Map;

public class ImageCache {

    private Map<String, Image> cache = new HashMap<>();

    public void addImage(String key, Image image) {
        cache.put(key, image);
    }

    public Image getImage(String key) {
        return cache.get(key);
    }
}
```

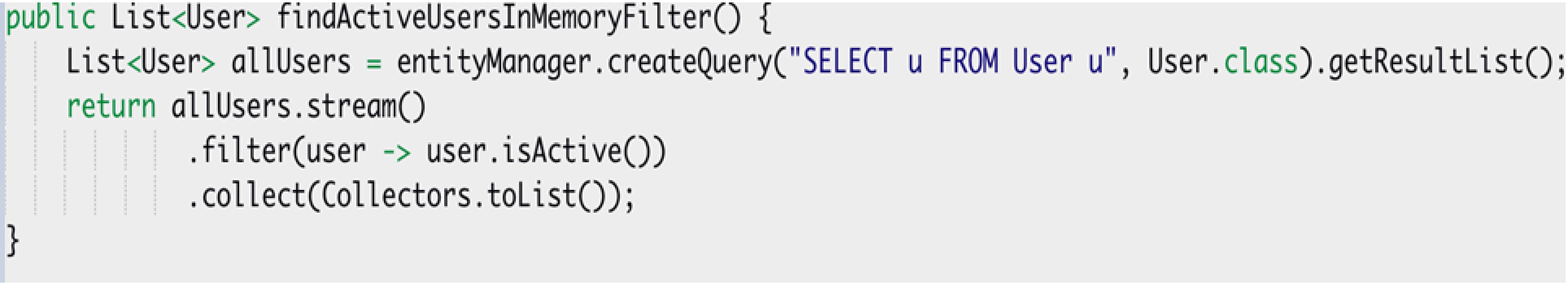
After



```java
import java.lang.ref.WeakReference;
import java.util.HashMap;
import java.util.Map;

public class ImageCache {

    private Map<String, WeakReference<Image>> cache = new HashMap<>();

    public void addImage(String key, Image image) {
        cache.put(key, new WeakReference<>(image));
    }

    public Image getImage(String key) {
        WeakReference<Image> reference = cache.get(key);
        if (reference != null) {
            return reference.get(); // Might return null if image was gc
        }
        return null;
    }
}
```

Other Best Practices

Before

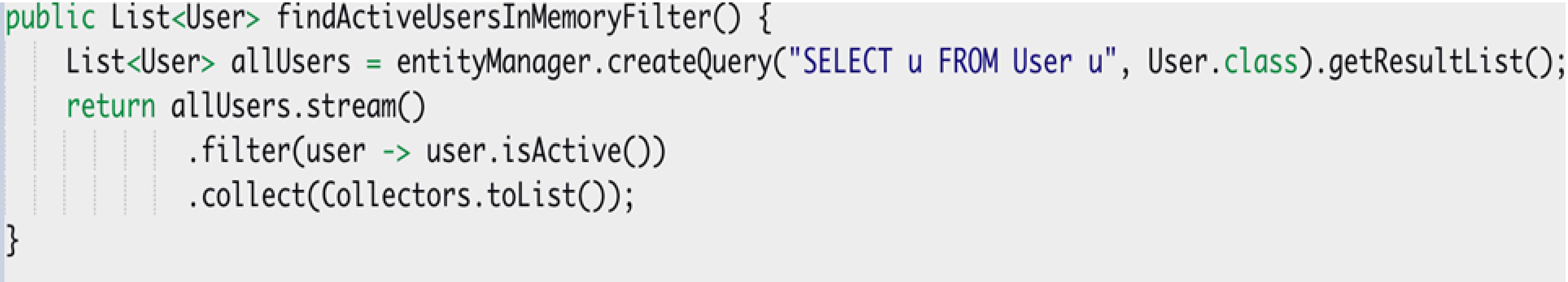


```
public List<User> findActiveUsersInMemoryFilter() {
    List<User> allUsers = entityManager.createQuery("SELECT u FROM User u", User.class).getResultList();
    return allUsers.stream()
            .filter(user -> user.isActive())
            .collect(Collectors.toList());
}
```

After

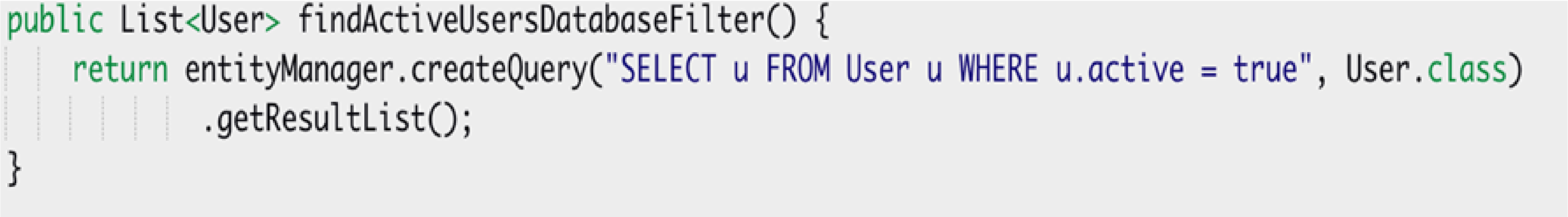
Other Best Practices

Before



```
public List<User> findActiveUsersInMemoryFilter() {
    List<User> allUsers = entityManager.createQuery("SELECT u FROM User u", User.class).getResultList();
    return allUsers.stream()
                .filter(user -> user.isActive())
                .collect(Collectors.toList());
}
```

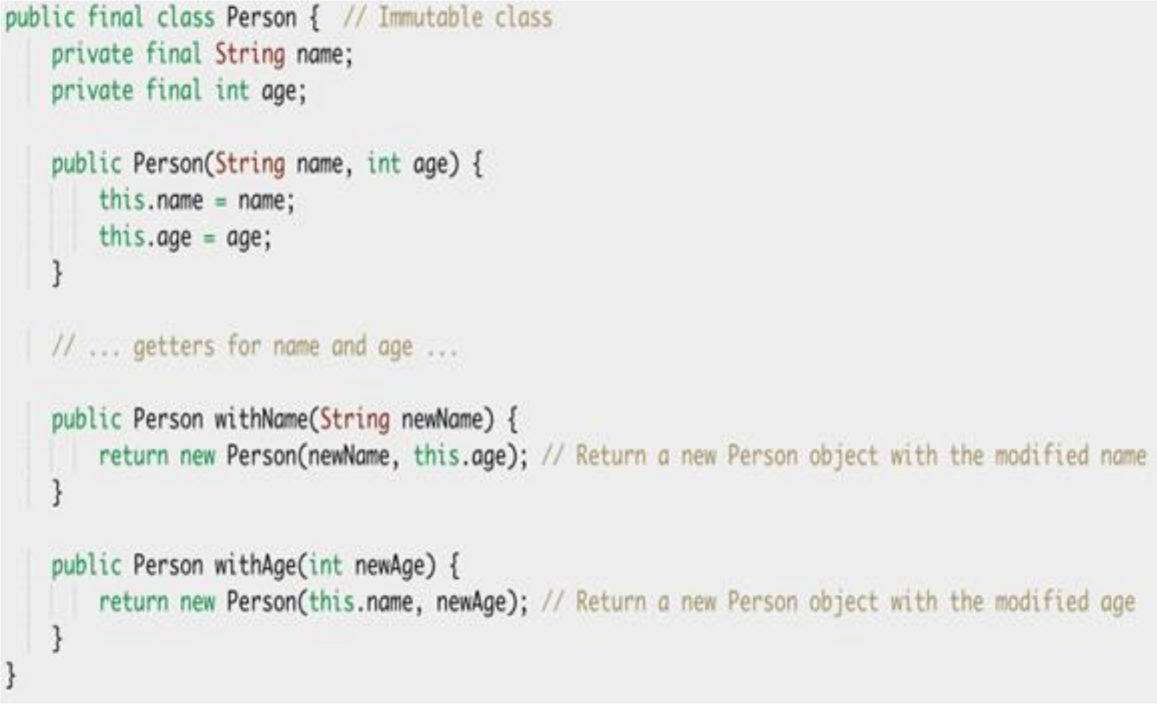
After



```
public List<User> findActiveUsersDatabaseFilter() {
    return entityManager.createQuery("SELECT u FROM User u WHERE u.active = true", User.class)
                .getResultList();
}
```

Other Best Practices

Immutable classes



```java
public final class Person {  // Immutable class
    private final String name;
    private final int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // ... getters for name and age ...

    public Person withName(String newName) {
        return new Person(newName, this.age); // Return a new Person object with the modified name
    }

    public Person withAge(int newAge) {
        return new Person(this.name, newAge); // Return a new Person object with the modified age
    }
}
```

•**Thread Safety:** Immutable objects are inherently thread-safe. You can share them across multiple threads without worrying about synchronization issues or data corruption.
•**Simplified Reasoning:** It's easier to understand and debug code that uses immutable objects because their state doesn't change unexpectedly.
•**Caching and Reuse:** Immutable objects can be safely cached and reused, potentially improving performance.
•**Error Prevention:** Immutability helps prevent common errors like accidental modification of shared objects.

Other Best Practices

Monitoring and profiling

- **Why:** Monitoring database performance and identifying bottlenecks is essential for optimization.
- **How:**
  - Use database monitoring tools to track performance metrics.
  - Use profiling tools to analyze query execution plans.

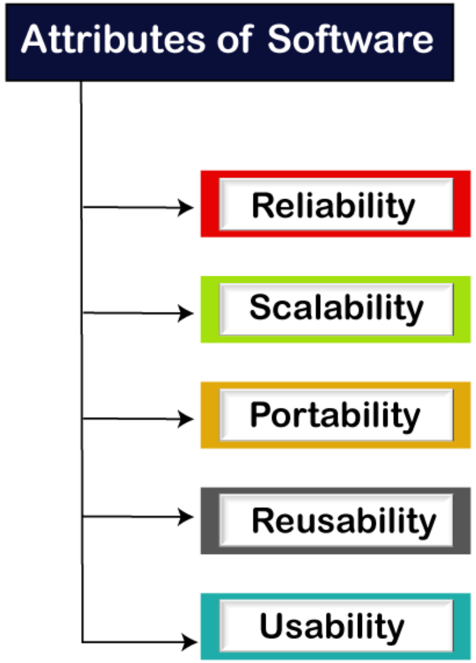
Using Transactions

- **Why:** Transactions ensure data consistency and integrity by grouping multiple database operations into a single unit of work.
- **How:**
  - Use the `Connection` object's `setAutoCommit(false)` to start a transaction.
  - Use `commit()` to save changes or `rollback()` to undo changes.
  - Keep transactions as short as possible to minimize locking and improve concurrency.

Why do we need testing?

Software testing

- It is the process of determining the correctness of a software product based on several features and evaluating the execution of several software components to determine bugs.



Attributes of Software

Reliability

Scalability

Portability

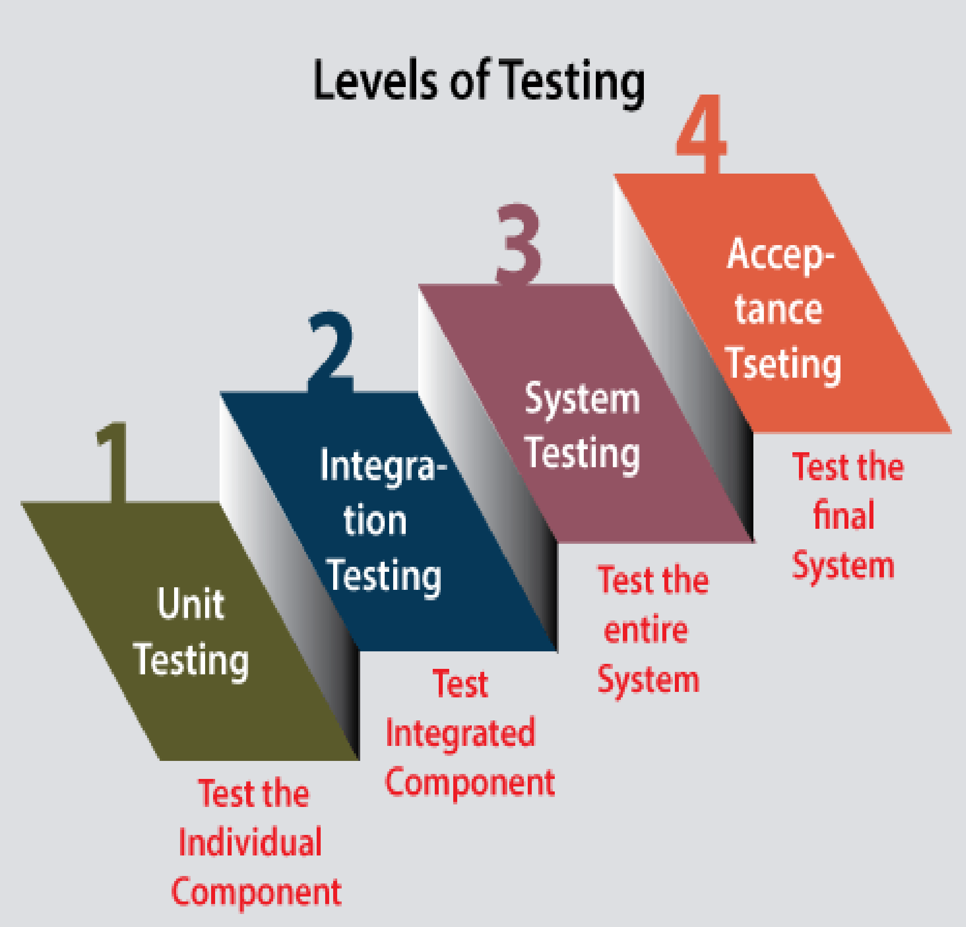
Reusability

Usability

https://www.javatpoint.com/software-testing-tutorial

Levels of testing

- **Unit testing**
  - Tests if several modules/components fulfils the requirements or not.
  - It is also a level of functional testing.
  - Used for understanding the business logic.
- **Integration Testing**
  - Tests the interaction between modules and components.
  - □ After we know each components works independently, we need to check the integration between them.
  - e.g., Test the database connection, reading data from a text file, etc.
- **System Testing**
  - Tests functional and non-functional requirements
  - Also called end-to-end (E2E) testing
  - Tests the whole system
  - e.g., testing a GUI application (when you click the save button and until the data is saved onto the database).
- **Acceptance Testing** – mainly done by the client (business side)



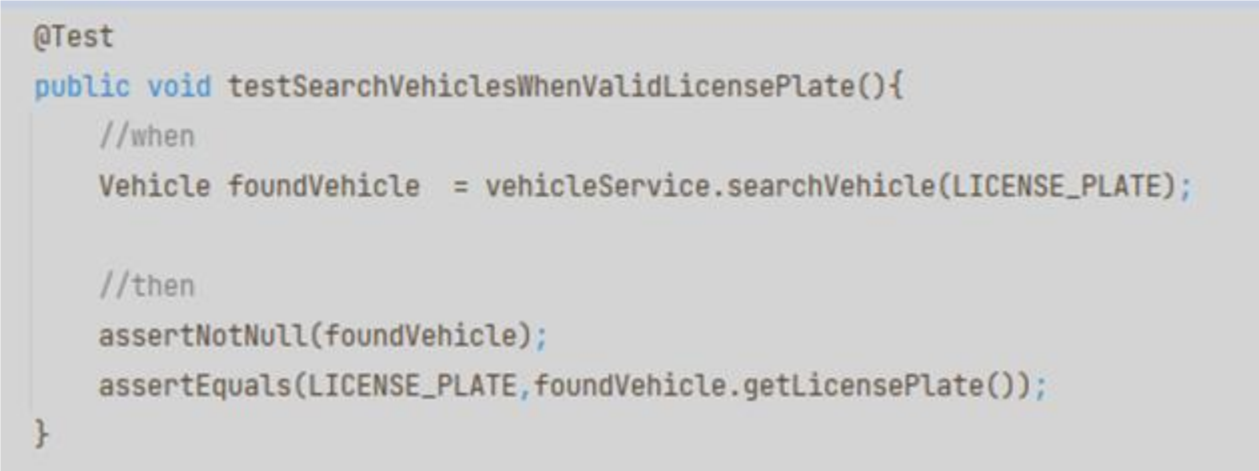
Unit testing

Unit testing in Java

- The most used framework is JUnit (current major version is JUnit 5- JUnit 6 released in September 2025)
- You need to download the JUnit jar if you use it into your project without a build tool like Maven or Gradle
- **JUnit annotations:**
    - @Test -indicates it is a test case that needs to be executed
    - @BeforeEach/ @Before (JUnit 4) – it is used if you need to execute something as a precondition before each test case
    - @BeforeAll/BeforeClass (JUnit4)– used to execute something before all test cases (e.g., connection to the database)
    - @AfterEach/ @After (JUnit4) – used to execute something as a post condition (e.g., delete data from the database)
    - @AfterAll/ @AfterClass (JUnit4) – used to execute something after all test cases (e.g., close a file)
    - @Test(expected=Exception.class) - can be used it you need to handle an exception during the test execution
    - @Disable/ @Ignore (JUnit4)- ignores the execution of a test case

Unit testing in Java

- Assertion methods in JUnit
  - □ assertTrue(boolean condition)
  - □ assertFalse(boolean condition)
  - □ assertNull(Object actual)
  - □ assertNotNull(Object actual)
  - □ assertEquals(… expected, … actual) where … could be double, int, short, String, etc.
  - □ assertNotEquals(…expected,… actual)
  - □ ……



```java
@Test
public void testSearchVehiclesWhenValidLicensePlate(){
    //when
    Vehicle foundVehicle  = vehicleService.searchVehicle(LICENSE_PLATE);

    //then
    assertNotNull(foundVehicle);
    assertEquals(LICENSE_PLATE,foundVehicle.getLicensePlate());
}
```

Mocking

- The process of creating several versions of the objects existing in the code and simulating the behavior of the real ones
- Goal: to test parts of the code in isolation
- Mocking is part of unit testing, used in TDD

- Why do we really need mocks?
  - We can remove external dependencies (DB connection, text files, sockets, etc.) from a unit test to test only the business logic without the interactions between the systems/components
  - Targets for mocking:
    - Database connections
    - Web services
    - Classes with side effects
- Mock libraries: Mockito, PowerMock, EasyMock, WireMock
- **Mockito** is an open-source framework used to create mocks (aka test doubles)

Mockito methods

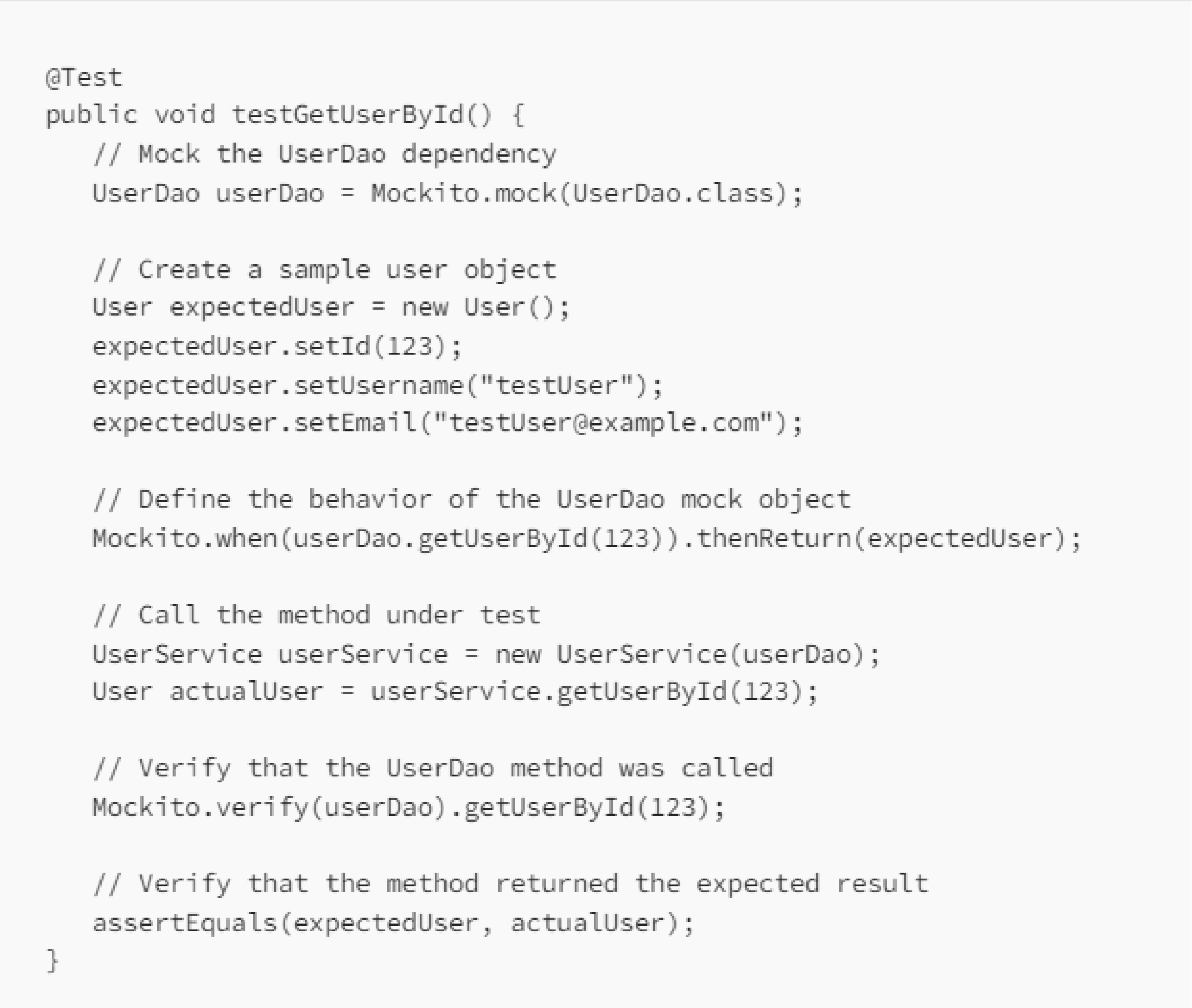
- mock()
  - Creates a mock object
  - UserRepository repo = Mockito.mock(UserRepository.class)

- when()
  - sets a functionality to a mock object.

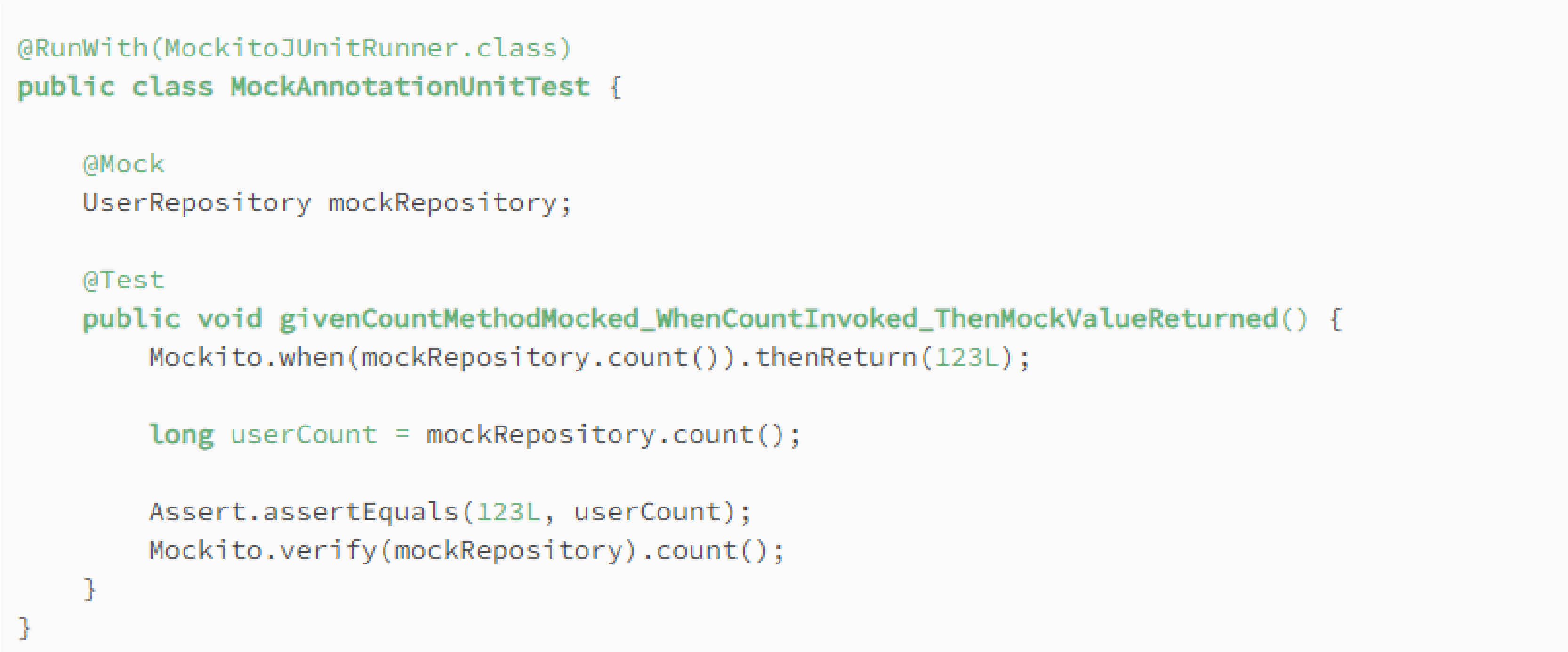
```
//add the behavior of calc service to add two numbers
when(calcService.add(10.0,20.0)).thenReturn(30.00);
```

- verify()
  - Checks that a mock method was called with the required parameters or not

```
//test the add functionality
Assert.assertEquals(calcService.add(10.0, 20.0),30.0,0);

//verify call to calcService is made or not with same arguments.
verify(calcService).add(10.0, 20.0);
```



```
@Test
public void testGetUserById() {
   // Mock the UserDao dependency
   UserDao userDao = Mockito.mock(UserDao.class);

   // Create a sample user object
   User expectedUser = new User();
   expectedUser.setId(123);
   expectedUser.setUsername("testUser");
   expectedUser.setEmail("testUser@example.com");

   // Define the behavior of the UserDao mock object
   Mockito.when(userDao.getUserById(123)).thenReturn(expectedUser);

   // Call the method under test
   UserService userService = new UserService(userDao);
   User actualUser = userService.getUserById(123);

   // Verify that the UserDao method was called
   Mockito.verify(userDao).getUserById(123);

   // Verify that the method returned the expected result
   assertEquals(expectedUser, actualUser);
}
```



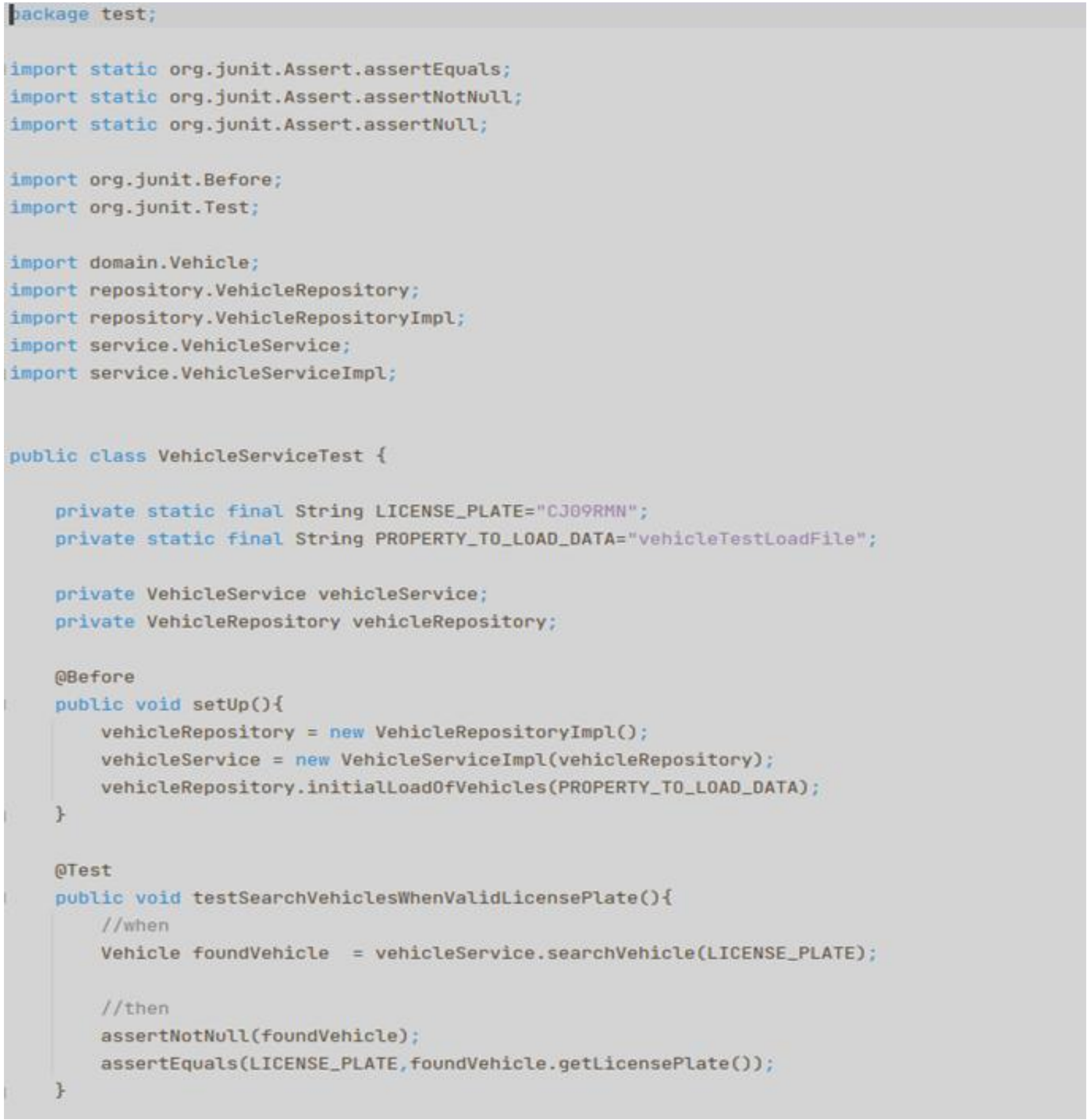
```
@RunWith(MockitoJUnitRunner.class)
public class MockAnnotationUnitTest {

    @Mock
    UserRepository mockRepository;

    @Test
    public void givenCountMethodMocked_WhenCountInvoked_ThenMockValueReturned() {
        Mockito.when(mockRepository.count()).thenReturn(123L);

        long userCount = mockRepository.count();

        Assert.assertEquals(123L, userCount);
        Mockito.verify(mockRepository).count();
    }
}
```

Integration testing

Integration testing in Java

- Used to test the interaction between components
- We need to be sure data is saved into a text file/database
- Is the repository logic working correctly?
- Is data sent properly to a source file?

- Other scenarios for integration testing:
  - Data transfer via sockets
  - Send messages through a messaging queue (e.g.Kafka)
  - Send data through APIs (HTTP calls)



```java
package test;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertNull;

import org.junit.Before;
import org.junit.Test;

import domain.Vehicle;
import repository.VehicleRepository;
import repository.VehicleRepositoryImpl;
import service.VehicleService;
import service.VehicleServiceImpl;


public class VehicleServiceTest {

    private static final String LICENSE_PLATE="CJ09RMN";
    private static final String PROPERTY_TO_LOAD_DATA="vehicleTestLoadFile";

    private VehicleService vehicleService;
    private VehicleRepository vehicleRepository;

    @Before
    public void setUp(){
        vehicleRepository = new VehicleRepositoryImpl();
        vehicleService = new VehicleServiceImpl(vehicleRepository);
        vehicleRepository.initialLoadOfVehicles(PROPERTY_TO_LOAD_DATA);
    }

    @Test
    public void testSearchVehiclesWhenValidLicensePlate(){
        //when
        Vehicle foundVehicle  = vehicleService.searchVehicle(LICENSE_PLATE);

        //then
        assertNotNull(foundVehicle);
        assertEquals(LICENSE_PLATE,foundVehicle.getLicensePlate());
    }
```

Q&A