Similar to variable names, constants reduce the cognitive load required by having extra to track the 'meaning' of special numbers. We can understand the number through its constant identifier. Using constants also allows the value to be updated, potentially in multiple places, easily.

Vertical whitespace can help to group logical blocks of code so that they can be understood as a group rather than forcing the reader to understand the whole section of code together.

Meaningful variable names make software easier to understand without requiring as much context. We can infer the use intended functionality of code without needing to understand what a variable 'actually' means.

```
// unmeaningful: requires external
// knowledge of x, y, and z
// to understand.
for (int x : y) {
    z++;
}
```

```
// meaningful: expected functionality
// is clear without additional context.
for (int studentID : students) {
    studentCount++;
}
```

```
// Function to write to a file
public static void writeToFile(String filePath) {
    try (BufferedWriter writer = new BufferedWriter(new FileWriter(filePath))) {
        writer.write("Hello, world!\n");
        writer.write("This is a test file.");
    } catch (IOException e) {
        e.printStackTrace();
    }
}

// Function to read from a file
public static List<String> readFromFile(String filePath) {
    List<String> lines = new ArrayList<>();
    try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {
        String line;
        while ((line = reader.readLine()) != null) {
            lines.add(line);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    return lines;
}
public static void generateCustomException(int number) throws
        NumberTooBigException, NumberTooSmallException {
    if (number < 30) {
        throw new NumberTooSmallException("The number was too small");
    } else if (number == 30) {
        throw new NumberTooSmallException();
    } else if (number > 50) {
        throw new NumberTooBigException("The number was too big");
    } else if (number == 50) {
        throw new NumberTooBigException();
    }
}
```

Missing Javadoc description of the method.
List implementation (ArrayList) used instead of interface.
Magic number (256) used instead of constant.
Missing horizontal whitespace around operators.

```
@Test
public void testSum() {
    assertEquals(6, FileExample.sum(Arrays.asList(1, 2, 3)));
}
@Test
public void testSquare() {
    assertEquals(25, FileExample.square(5));
}
```

```
public class StreamOperationsExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        // Filter: Get even numbers
        List<Integer> evenNumbers = numbers.stream()
                .filter(n -> n % 2 == 0)
                .collect(Collectors.toList());

        // Map: Square each number
        List<Integer> squaredNumbers = numbers.stream()
                .map(n -> n * n)
                .collect(Collectors.toList());

        // FlatMap: Flatten a stream of lists
        List<List<Integer>> nestedNumbers = Arrays.asList(
                Arrays.asList(1, 2),
                Arrays.asList(3, 4),
                Arrays.asList(5, 6)
        );
        List<Integer> flattenedNumbers = nestedNumbers.stream()
                .flatMap(List::stream)
                .collect(Collectors.toList());

        // Reduce: Sum all numbers
        int sum = numbers.stream()
                .reduce(0, Integer::sum);

        // Sorted: Sort numbers in ascending order
        List<Integer> sortedNumbers = numbers.stream()
                .sorted()
                .collect(Collectors.toList());

        // Limit and Skip: Get numbers 6 to 10
        List<Integer> limitSkipNumbers = numbers.stream()
                .skip(5)
                .limit(5)
                .collect(Collectors.toList());

        // Distinct: Get distinct numbers
        List<Integer> distinctNumbers = numbers.stream()
                .distinct()
                .collect(Collectors.toList());

        // Count: Count the number of elements
        long count = numbers.stream()
                .count();

        // AllMatch, AnyMatch, NoneMatch: Check conditions
        boolean allPositive = numbers.stream().allMatch(n -> n > 0);
        boolean anyEven = numbers.stream().anyMatch(n -> n % 2 == 0);
        boolean noneNegative = numbers.stream().noneMatch(n -> n < 0);

        // FindFirst: Find the first number
        numbers.stream()
                .findFirst()
                .ifPresent(n -> System.out.println("First number: " + n));
    }
}
```

## Java Collections

### Arrays

- Fixed-length, mutable sequences of homogeneous items.

### Built-in Collections

- `Stack`: LIFO (Last In, First Out)
  - `empty()`
  - `peek()`
  - `pop()`
  - `push(obj)`
- `List`: Ordered collection, dynamic size
  - `int size()`
  - `boolean isEmpty()`
  - `boolean contains(E item)`
  - `boolean add(E item)`
  - `void add(int index, E element)`
  - `E remove(int index)`
  - `E get(int index)`
  - Major implementations: `ArrayList`, `LinkedList`
- `ArrayList`: A resizable array implementation of the List interface.
  - `Additional Methods: remove(Object o), clear().`
- `LinkedList`: A doubly-linked list implementation of the List interface.
  - `Additional Methods: addFirst(E e), addLast(E e), removeFirst(), removeLast().`
- `Set`: Unordered collection, no duplicates
  - `int size()`
  - `boolean contains(E item)`
  - `boolean add(E e)`
  - `boolean remove(E item)`
  - Major implementations: `TreeSet`, `HashSet`
- `Map`: Key-value pairs
  - `int size()`
  - `boolean containsKey(K key)`
  - `boolean containsValue(V value)`
  - `V get(K key)`
  - `V put(K key, V value)`
  - `V remove(K key)`
  - `Set<K> keySet()`
  - Major implementations: `TreeMap`, `HashMap`

## Exception Handling

### Custom Exceptions

- Create by extending `Exception` class.
- Example:

```
java
Copy code
public class OopsADaisy extends
Exception {
    public OopsADaisy() { super(); }
    public OopsADaisy(String message)
{ super(message); }
    public OopsADaisy(Exception cause)
{ super(cause); }
}
```

### Best Practices

- Use common exceptions when possible.
- Always catch more specific exceptions before generic ones.

## Key Principles and Practices

### SOLID Principles

- **Single Responsibility Principle**: A class should have only one reason to change.
- **Open-Closed Principle**: Classes should be open for extension but closed for modification.
- **Liskov Substitution Principle**: Subclasses should be substitutable for their parent classes without affecting functionality. Precondition may be weakened, postcondition may be strengthened.
- **Interface Segregation Principle**: Many client-specific interfaces are better than one general-purpose interface.
- **Dependency Inversion Principle**: Depend on abstractions, not concrete implementations.

### Coupling and Cohesion

- **Cohesion**: Measure of how well the elements within a module work together.
  - High cohesion is preferable.
- **Coupling**: Measure of how closely connected different modules are.
  - Low coupling is preferable.
  - Types: Content, Common, External, Control, Stamp, Data.

### Code Smells and Refactoring

- **Code Smells**: Indicators of potential issues in code (e.g., duplication, long methods, feature envy).
- **Refactoring**: Process of restructuring existing code without changing its external behaviour.
  - Done to improve readability and reduce complexity.

### Testing and Debugging

- **JUnit**: Framework for unit testing in Java.
  - Use annotations like `@Test`, `@Before`, and `@After`.
  - Use assertions to check expected outcomes (`assertEquals`, `assertTrue`, `assertFalse`).
- **Debugging**: Use breakpoints, stepping (step over, step into) to diagnose issues.
- **Test-Driven Development (TDD)**: Write tests before implementing functionality.

### Java Input/Output (I/O)

- **Streams**: Used for reading and writing bytes.
- **Readers and Writers**: For text data.
- **File Operations**: Create, delete, read, and write files using `Path` and `Files` classes.

### Generics

- Allow classes and methods to operate on objects of various types while providing compile-time type safety.
- Common type parameters: `T`, `E`, `K`, `V`, `N`.
- **Bounded Generics**: Restrict the types that can be used (e.g., `<T extends Number>`).

## Important Practices

### Specifications

- Should be clear, concise, and rule out incorrect implementations.
- Types: Informal (normal comments), Semi-formal (structured English), Formal (mathematical constraints).

### Defensive Programming

- Explicitly check for invalid inputs.
- Ensure robustness by handling errors gracefully.

## Additional Concepts

### Event-Driven Programming

- Uses callbacks and the observer pattern for handling events.
- **Observer Pattern**: Allows multiple listeners to respond to events.

### Version Control

- Use version control systems (e.g., Git) to track changes, manage versions, and collaborate.
- Key commands: `git init`, `git add`, `git commit`, `git log`, `git diff`.

## Functional Interfaces:

Functional interfaces are interfaces that contain exactly one abstract method.

They can have multiple default or static methods, but only one abstract method.

Java provides several built-in functional interfaces in the `java.util.function` package, such as `Predicate`, `Consumer`, `Function`, and `Supplier`.

You can also define your own functional interfaces by annotating them with the `@FunctionalInterface` annotation, which is optional but serves as a reminder of the intended use.

**Law of Demeter** (LoD):

The Law of Demeter is a design principle in object-oriented programming that suggests that a method should only interact with its immediate dependencies and avoid cascading method calls through multiple objects. In simpler terms, an object should only "talk to its friends" and not to the friends of its friends. This principle promotes loose coupling between classes and helps to manage complexity by limiting the interactions between objects.

**Black Box Testing**: A testing method where the internal workings or structure of the software being tested are not known to the tester. Tests are based on specifications and requirements.

**Glass Box Testing**: A testing method where the tester has knowledge of the internal workings or structure of the software being tested. Tests are based on the implementation details of the code.

**Code Coverage**:

**Statement Coverage**: Determines if each statement in the code has been executed at least once during testing.

**Branch Coverage**: Ensures that each decision point in the code has been tested both with true and false conditions.

**Path Coverage**: Requires that every possible path through the code is executed at least once during testing, but this level of coverage may be impractical for complex programs.

**Path Coverage**: Ensures that every possible path through the code is traversed at least once during testing, but achieving this level of coverage may not always be feasible due to the exponential growth in the number of paths for complex programs.