a1a # tooling

Checkstyle Rules Reference

TCSS 305 Programming Practicum

This reference documents the specific Checkstyle rules enforced in TCSS 305. Use this as a quick lookup when you encounter violations.

Configuration File

Our Checkstyle configuration: `tcss305_checkstyle.xml` (Checkstyle 12.x compatible)

> **Note**
>
> Our style uses K&R / Java standard brace placement—opening braces go on the same line as the declaration, not the next line.

Naming Conventions

Instance Fields: `my` Prefix

All instance (non-static) fields must start with the `my` prefix.

```
// Correct
private String myName;
private BigDecimal myPrice;
private final List<Item> myItems;

// Incorrect - will trigger violation
private String name;
private BigDecimal price;
private final List<Item> items;
```

Why? The `my` prefix makes instance fields instantly recognizable, distinguishing them from local variables and parameters. This is a TCSS 305 convention.

Constants: UPPER_SNAKE_CASE

Static final fields (constants) use all uppercase with underscores.

```
// Correct
private static final int MAX_QUANTITY = 100;
private static final String DEFAULT_NAME = "Unknown";

// Incorrect
private static final int maxQuantity = 100;
private static final int MAXQUANTITY = 100;
```

Classes: PascalCase

Class and interface names start with uppercase, each word capitalized.

```
// Correct
public class StoreItem { }
public interface ShoppingCart { }

// Incorrect
public class storeItem { }
public class Store_Item { }
```

Methods and Variables: camelCase

Methods, parameters, and local variables use camelCase.

```
// Correct
public void calculateTotal() { }
int itemCount = 0;

// Incorrect
public void CalculateTotal() { }
public void calculate_total() { }
int ItemCount = 0;
```

Modern Java Features: Naming Rules

Modern Java constructs follow consistent naming conventions:

Lambda parameters use camelCase:

```
// Correct
items.forEach(item -> process(item));
map.computeIfAbsent(key, k -> createValue(k));

// Incorrect
items.forEach(Item -> process(Item));  // Should be lowercase
```

Pattern variables (from `instanceof` pattern matching) use camelCase:

```
// Correct
if (obj instanceof String text) {
    System.out.println(text.length());
}

// Incorrect
if (obj instanceof String Text) { ... }  // Should be lowercase
```

Record components use camelCase (no `my` prefix):

```
// Correct
public record Point(int x, int y) { }
public record Person(String name, int age) { }

// Incorrect
public record Point(int myX, int myY) { }  // No "my" prefix for records
```

Record type parameters follow the single capital letter convention:

```
// Correct
public record Pair<K, V>(K first, V second) { }

// Incorrect
public record Pair<Key, Value>(Key first, Value second) { }
```

Formatting Rules

Line Length: Maximum 120 Characters

No line should exceed 120 characters. Break long lines appropriately.

```
// Too long (over 120 characters)
public void
processCustomerOrderWithDiscountAndShippingCalculationAndReturnsPolicyValidati
on(Customer customer, Order order) {

// Correct - broken into multiple lines
public void processCustomerOrderWithDiscountAndShippingCalculation(
        final Customer customer,
        final Order order) {
```

> **Tip**
>
> IntelliJ shows a vertical line at column 120. Keep your code to the left of that line.

Indentation: 4 Spaces

Use 4 spaces for indentation, not tabs.

```
public class Example {
    private int myValue;  // 4 spaces

    public void method() {
        if (condition) {
            doSomething();  // 8 spaces (2 levels)
        }
    }
}
```

Brace Style: Same Line

Opening braces go on the same line as the declaration.

```
// Correct (sameline style)
public class Example {
    public void method() {
        if (condition) {
            // code
        }
    }
}

// Incorrect (nextline style - not our convention)
public class Example
{
    public void method()
    {
        if (condition)
        {
            // code
        }
    }
}
```

Whitespace Rules

Around operators:

```
// Correct
int sum = a + b;
boolean result = x == y;

// Incorrect
int sum=a+b;
boolean result = x==y;
```

After commas:

```
// Correct
method(arg1, arg2, arg3);

// Incorrect
method(arg1,arg2,arg3);
```

No whitespace after `(` or before `)`:

```
// Correct
method(arg1, arg2);

// Incorrect
method( arg1, arg2 );
```

Method Chaining: Leading Dot Style

When chaining method calls across multiple lines, the dot (`.`) must be on the new line, not at the end of the previous line.

```
// Correct - leading dot style
BigDecimal total = myOrders.values().stream()
        .map(this::calculateOrderTotal)
        .reduce(BigDecimal.ZERO, BigDecimal::add)
        .setScale(2, RoundingMode.HALF_EVEN);

// Incorrect - trailing dot style
BigDecimal total = myOrders.values().stream().
        map(this::calculateOrderTotal).
        reduce(BigDecimal.ZERO, BigDecimal::add).
        setScale(2, RoundingMode.HALF_EVEN);
```

Why? Leading dots make the chain structure clearer—each line starts with the operation being performed. This is the modern convention for fluent APIs and stream operations.

> **Tip**
>
> If your entire chain fits on one line (under 120 characters), keep it on one line. Only break to multiple lines when needed.

Ternary Operator: Allowed

The ternary operator (`? :`) is permitted in TCSS 305. Use it for simple, readable conditional expressions.

```
// Allowed - clear and concise
String status = isActive ? "Active" : "Inactive";
int max = (a > b) ? a : b;

// Consider if/else for complex conditions
// Bad - too complex for ternary
String result = (x > 0 && y < 10) ? (z == 0 ? "zero" : "nonzero") :
"negative";
```

Tip

Use ternary for simple cases. If the logic is complex or nested, an `if/else` statement is clearer.

Lambda Bodies: One Statement Maximum

Lambda expressions must contain at most **one executable statement**. If you need more logic, extract it to a helper method.

```
// Correct - single expression
items.forEach(item -> System.out.println(item));
names.stream().map(name -> name.toUpperCase()).toList();

// Correct - method reference (preferred when applicable)
items.forEach(System.out::println);
names.stream().map(String::toUpperCase).toList();

// Incorrect - multiple statements (will trigger violation)
items.forEach(item -> {
    System.out.println(item);
    processItem(item);  // Two statements - not allowed!
});

// Fix: Extract to a helper method
items.forEach(this::printAndProcess);

private void printAndProcess(final Item item) {
    System.out.println(item);
    processItem(item);
}
```

Why? Lambdas should be concise. Complex lambda bodies reduce readability and make debugging harder. Helper methods are testable and reusable.

Avoid Duplicate String Literals

The same string literal should not appear multiple times in your code. Define constants instead.

```
// Incorrect - duplicate literals
if (status.equals("ACTIVE")) { ... }
if (status.equals("ACTIVE")) { ... }  // Duplicate!

// Correct - use a constant
private static final String STATUS_ACTIVE = "ACTIVE";

if (status.equals(STATUS_ACTIVE)) { ... }
if (status.equals(STATUS_ACTIVE)) { ... }
```

Why? Constants prevent typos, enable IDE refactoring, and make the code's intent clearer. If a string is used multiple times, it's likely meaningful enough to deserve a named constant.

> **Note**
>
> This rule ignores strings in static initializers and annotations, where repetition is sometimes unavoidable.

Javadoc Requirements

> **Note**
>
> Javadoc is required for **non-private** members (public, protected, and package-private). Only private members are exempt, though you may add Javadoc to them if helpful.

Class Javadoc

Non-private classes (public, protected, and package-private) must have Javadoc with `@author` and `@version` tags.

```
/**
 * Brief description of what this class does.
 * <p>
 * Additional details if needed. Can span multiple paragraphs
 * using the HTML paragraph tag.
 *
 * @author Your Name
 * @version Quarter Year (e.g., Winter 2025)
 */
public class StoreItem implements Item {
```

> **Important**
>
> The `@author` tag must contain your name as it appears in Canvas. The `@version` tag should include the current quarter and year.

Method Javadoc

Non-private methods (public, protected, and package-private) require Javadoc with `@param`, `@return`, and `@throws` tags as applicable.

```
/**
 * Calculates the total price for the given quantity of this item.
 *
 * @param quantity the number of items to calculate
 * @param useMembership true if membership pricing should apply
 * @return the total price as a BigDecimal
 * @throws IllegalArgumentException if quantity is negative
 */
public BigDecimal calculateTotal(final int quantity,
                                 final boolean useMembership) {
```

Tag requirements: - `@param` - one for each parameter, in order - `@return` - required unless method returns `void` - `@throws` - one for each exception the method can throw

Field Javadoc

Non-private fields (public, protected, and package-private) must have Javadoc explaining their purpose. Private fields do not require Javadoc.

```
/**
 * The name of this item as displayed to customers.
 */
private final String myName;

/**
 * The unit price of this item in US dollars.
 */
private final BigDecimal myPrice;
```

Modifier Order

When multiple modifiers are used, they must appear in this order:

1. `public` / `protected` / `private`

2. `abstract`
3. `static`
4. `final`
5. `transient`
6. `volatile`
7. `synchronized`
8. `native`
9. `strictfp`

```
// Correct
public static final int MAX_SIZE = 100;
private static final NumberFormat CURRENCY_FORMAT = ...;

// Incorrect (wrong order)
final static public int MAX_SIZE = 100;
static private final NumberFormat CURRENCY_FORMAT = ...;
```

Common Violations and Fixes

"Line is longer than 120 characters"

Fix: Break the line at a logical point.

```
// Before - too long
throw new IllegalArgumentException("The provided value must be positive but
was: " + value + " which is not allowed in this context");

// After - broken at a logical point
throw new IllegalArgumentException(
        "The provided value must be positive but was: " + value
        + " which is not allowed in this context");
```

"Missing Javadoc comment"

Fix: Add the required Javadoc.

```
// Before
public String getName() {
    return myName;
}

// After
/**
 * Returns the name of this item.
```

```
     *
     * @return the item name
     */
    public String getName() {
        return myName;
    }
```

"Name 'price' must match pattern '(^my[A-Z][a-zA-Z0-9]*$)'"

Fix: Add the `my` prefix to instance fields.

```
// Before
private BigDecimal price;

// After
private BigDecimal myPrice;
```

"Lambda has N statement(s) (max allowed is 1)"

Fix: Extract the lambda body to a helper method.

```
// Before - multiple statements in lambda
items.forEach(item -> {
    log.info("Processing: " + item);
    process(item);
});

// After - extracted to helper method
items.forEach(this::logAndProcess);

private void logAndProcess(final Item item) {
    log.info("Processing: " + item);
    process(item);
}
```

"'X' is not followed by whitespace"

Fix: Add a space after the indicated token.

```
// Before
if(condition) {

// After
if (condition) {
```

"'{' is not preceded by whitespace"

Fix: Add a space before the opening brace.

```
// Before
public void method(){

// After
public void method() {
```

The `final` Keyword

> **Tip**
>
> While not strictly required by Checkstyle, using `final` on parameters and local variables is a best practice that IntelliJ Inspections may flag.

```
// Good practice - final parameters
public void setPrice(final BigDecimal price) {
    myPrice = price;
}

// Good practice - final local variables when not reassigned
public String formatPrice() {
    final NumberFormat formatter = NumberFormat.getCurrencyInstance();
    return formatter.format(myPrice);
}
```

Using `final` signals that the value won't change, making code easier to reason about.

Quick Reference Table

Element	Convention	Example
Class	PascalCase	`StoreItem`, `ShoppingCart`
Interface	PascalCase	`Item`, `Cart`
Method	camelCase	`calculateTotal()`, `getName()`
Instance field	my + PascalCase	`myName`, `myPrice`
Parameter	camelCase	`name`, `price`, `itemCount`

Element	Convention	Example
Local variable	camelCase	`total`, `itemCount`
Constant	UPPER_SNAKE_CASE	`MAX_SIZE`, `DEFAULT_NAME`
Line length	Max 120 characters	—
Lambda body	Max 1 statement	—
Indentation	4 spaces	—
Braces	Same line	`if (x) {`

Further Reading

External Resources

- Checkstyle Checks Reference - Complete list of all Checkstyle rules
- Sun Code Conventions - Oracle's original Java style guide (basis for many Checkstyle rules)

References

Primary Texts:

- Bloch, J. (2018). *Effective Java* (3rd ed.). Addison-Wesley. Chapter 9: General Programming.
- Horstmann, C. S. (2022). *Core Java, Volume I: Fundamentals* (12th ed.). Oracle Press. Chapter 3: Fundamental Programming Structures.
- Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall. Chapter 2: Meaningful Names.

Style Guides and Tools:

- Checkstyle Documentation — Official Checkstyle documentation and rule reference
- Google Java Style Guide — Industry-standard style guide

- Oracle Code Conventions for Java — Original Sun/Oracle style conventions

---

This guide is part of TCSS 305 Programming Practicum, School of Engineering and Technology, University of Washington Tacoma.