a1a # tooling

IntelliJ Inspections Reference

TCSS 305 Programming Practicum

This reference documents the IntelliJ IDEA code inspections enabled in TCSS 305. Use this as a quick lookup when you see warnings in the editor.

Inspections vs Checkstyle

TCSS 305 uses **two complementary tools** for code quality:

Tool	Purpose	When It Runs
Checkstyle	Style rules (formatting, naming, Javadoc)	Build time / CI
IntelliJ Inspections	Code quality & complexity analysis	Real-time in editor

Key differences:

- **Checkstyle** catches formatting issues (line length, whitespace, brace placement)
- **Inspections** catch design issues (complexity, coupling, potential bugs)
- Some rules overlap intentionally (method count, return points, nesting depth)
- Both must pass for full credit on assignments

> **Tip**
>
> Fix issues as you code. IntelliJ shows inspections in real-time with yellow/red highlights. Don't wait until submission to address warnings.

Complexity Limits

These inspections enforce limits on code complexity. All are aligned with Checkstyle where applicable.

Class Complexity

Limit: 80

Measures total complexity of a class based on all its methods. If exceeded, your class is doing too much—consider splitting it.

Cyclomatic Complexity

Limit: 10

Counts the number of independent paths through a method. Each `if`, `for`, `while`, `case`, `&&`, `||` adds to complexity.

```
// High complexity - too many branches
public String categorize(int value) {
    if (value < 0) {
        if (value < -100) {
            return "very negative";
        } else if (value < -50) {
            return "negative";
        } else {
            return "slightly negative";
        }
    } else if (value == 0) {
        return "zero";
    } else {
        // ... more branches
    }
}

// Lower complexity - use early returns or extract methods
public String categorize(int value) {
    if (value < 0) {
        return categorizeNegative(value);
    }
    if (value == 0) {
        return "zero";
    }
    return categorizePositive(value);
}
```

Nesting Depth

Limit: 3 (aligned with Checkstyle)

Maximum depth of nested control structures. Deep nesting is hard to read.

```
// Too deeply nested
public void process(List<Order> orders) {
    for (Order order : orders) {                    // Level 1
```

```
        if (order.isValid()) {                      // Level 2
            for (Item item : order.getItems()) {    // Level 3
                if (item.inStock()) {               // Level 4 - TOO DEEP!
                    // ...
                }
            }
        }
    }
}

// Refactored with early continue and extracted method
public void process(List<Order> orders) {
    for (Order order : orders) {
        if (!order.isValid()) {
            continue;
        }
        processOrderItems(order);
    }
}
```

Multiple Return Points

Limit: 3 (aligned with Checkstyle)

Maximum number of `return` statements in a method. Too many returns make control flow hard to follow.

> **Note**
>
> The `equals()` method is exempt from this rule since multiple guard clauses are common.

Non-Comment Source Statements

Limit: 50 (aligned with Checkstyle)

Maximum executable statements per method. Long methods should be broken into smaller, focused methods.

Anonymous Class Complexity

Limit: 3

Anonymous inner classes should be simple. If complexity exceeds 3, extract to a named class or use a lambda.

Anonymous Class Method Count

Limit: 1

Anonymous classes should have at most one method. For multiple methods, use a named inner class or separate class.

Size Limits

Method Count

Limit: 50 (aligned with Checkstyle)

Maximum methods per class. Classes with too many methods likely have too many responsibilities.

Field Count

Limit: 10

Maximum instance fields per class. Too many fields suggest the class is doing too much.

> **Note**
>
> Constants (`static final` fields) and enum constants are excluded from this count.

Parameters Per Method

Limit: 8 (aligned with Checkstyle)

Maximum parameters for a method. Too many parameters are hard to remember and often indicate a design problem.

Fix: Group related parameters into an object, or reconsider the method's responsibility.

Parameters Per Constructor

Enabled with default limit. Constructors with many parameters may benefit from the Builder pattern.

Constructor Count

Enabled with default limit. Too many constructors can indicate overloaded responsibilities.

Class Coupling

Limit: 15

Counts the number of other classes a class depends on. High coupling makes code harder to maintain and test.

Fix: Apply the Single Responsibility Principle. Extract helper classes or use interfaces to reduce direct dependencies.

> **Note**
>
> Java standard library classes and library classes are excluded from this count—only your project classes count toward the limit.

Inheritance & Structure

Class Inheritance Depth

Enabled with default limit. Deep inheritance hierarchies are fragile and hard to understand. Prefer composition over inheritance.

Class Nesting Depth

Limit: 1

Limits how deeply inner classes can be nested. Deeply nested classes are hard to navigate.

Class With Only Private Constructors

Warns when a class has only private constructors but isn't a utility class. Consider making it `final` or adding a public factory method.

Assignment Inspections

These catch accidental or confusing assignments:

Assignment To For Loop Parameter

Warns when you modify the loop variable inside the loop body. This is usually a bug.

```java
// Bad - modifying loop variable
for (int i = 0; i < 10; i++) {
    i = i + 2;  // Confusing! Use a different variable
}
```

Assignment To Method Parameter

Warns when you reassign a method parameter. Parameters should be treated as immutable.

```java
// Bad
public void process(String value) {
    value = value.trim();  // Reassigning parameter
    // ...
}

// Good - use final and a local variable
public void process(final String value) {
    final String trimmed = value.trim();
    // ...
}
```

Assignment To Lambda Parameter

Same as above, but for lambda parameters.

Assignment To Static Field From Instance Method

Warns when an instance method modifies a static field. This is usually a design error.

Common Inspections

Magic Number

Warns about numeric literals that should be named constants.

```java
// Bad - what does 0.08 mean?
return price * 0.08;

// Good - self-documenting
private static final double TAX_RATE = 0.08;
return price * TAX_RATE;
```

Local Can Be Final

Suggests adding `final` to local variables that aren't reassigned.

```
// IntelliJ suggests:
final String name = customer.getName();
final int count = items.size();
```

> **Tip**
>
> IntelliJ can auto-fix this. Press `Alt+Enter` on the warning and select "Make variable final".

Missing @Override Annotation

Warns when a method overrides a superclass method but lacks `@Override`.

```
// Missing annotation - will warn
public String toString() {
    return myName;
}

// Correct
@Override
public String toString() {
    return myName;
}
```

On Demand Import (Star Import)

Warns about `import java.util.*;` style imports. Always use explicit imports.

Equals And Hashcode

Warns when `equals()` is overridden without `hashCode()`, or vice versa. Both must be overridden together.

Comparable Implemented But Equals Not Overridden

Warns when a class implements `Comparable` but doesn't override `equals()`. These should be consistent.

Utility Class Inspections

Utility Class Without Private Constructor

Warns when a class with only static methods has an accessible constructor.

```java
// Bad - can instantiate a utility class
public class MathUtils {
    public static int add(int a, int b) { ... }
}

// Good
public final class MathUtils {
    private MathUtils() {
        // Prevent instantiation
    }

    public static int add(int a, int b) { ... }
}
```

Utility Class With Public Constructor

Similar to above. Utility classes should have private constructors.

Non-Final Utility Class

Utility classes should be `final` to prevent subclassing.

Constructor & Initialization

Implicit Call To Super

Warns when a constructor doesn't explicitly call `super()`. While Java inserts it automatically, being explicit is clearer.

Overridable Method Call During Object Construction

Warns when a constructor calls a method that could be overridden. This is dangerous because the subclass isn't fully initialized yet.

```java
// Dangerous - setup() could be overridden
public class Parent {
    public Parent() {
        setup();  // Warning!
    }

    protected void setup() { }
}
```

Other Enabled Inspections

Inspection	Description
SystemOutErr	Warns about `System.out` and `System.err` usage (use logging instead)
ReturnNull	Warns about returning `null` (consider `Optional` or throwing an exception)
LiteralAsArgToStringEquals	Prefer `"constant".equals(variable)` over `variable.equals("constant")`
NegativelyNamedBooleanVariable	Avoid names like `notFound` or `isInvalid` (double negatives confuse)
LimitedScopeInnerClass	Inner class could be local to a method
InterfaceMayBeAnnotatedFunctional	Interface with one method could use `@FunctionalInterface`
OverlyLongLambda	Lambda is too long; extract to a method
PublicMethodNotExposedInInterface	Consider adding public methods to an interface

---

Intentionally Disabled Inspections

These inspections are disabled because they conflict with course conventions or create too much noise:

LawOfDemeter

Reason: Too noisy with modern Java streams and fluent builders.

```
// This would trigger Law of Demeter warnings but is idiomatic Java:
items.stream()
     .filter(Item::isAvailable)
     .map(Item::getPrice)
     .reduce(BigDecimal.ZERO, BigDecimal::add);
```

ClassCanBeRecord / RecordCanBeClass

Reason: The course teaches traditional classes before introducing records. These suggestions would be premature.

ConstantOnWrongSideOfComparison

Reason: "Yoda conditions" (`null == x`) are less readable than `x == null` .

UnnecessaryModifier

Reason: Keeping `public` on `main()` methods is clearer for teaching purposes.

Unused Symbol (METHOD and CLASS options)

Reason: Starter code may have methods/classes for students to implement. These aren't "unused"—they're waiting to be completed.

> **Note**
>
> Unused local variables, fields, and parameters are still flagged. Only unused methods and classes are exempt.

Quick Reference Table

Inspection	Limit	Aligned with Checkstyle?
Class Complexity	80	—
Cyclomatic Complexity	10	Yes
Nesting Depth	3	Yes
Multiple Return Points	3	Yes
Non-Comment Source Statements	50	Yes
Method Count	50	Yes

Inspection	Limit	Aligned with Checkstyle?
Field Count	10	—
Parameters Per Method	8	Yes
Class Coupling	15	—
Anonymous Class Complexity	3	—
Anonymous Class Method Count	1	—
Class Nesting Depth	1	—

Viewing Inspections in IntelliJ

1. **Real-time:** Warnings appear as yellow highlights in the editor
2. **Gutter icons:** Yellow/red marks in the right gutter show line locations
3. **Problems tool window:** `View > Tool Windows > Problems` shows all issues
4. **Run inspections manually:** `Code > Inspect Code...` for a full report

> **Tip**
>
> Hover over any highlighted code to see the inspection name and suggested fix. Press `Alt+Enter` (or `Option+Enter` on Mac) for quick-fix options.

Further Reading

> **External Resources**
>
> - IntelliJ IDEA Inspections - Official JetBrains documentation
> - Java Code Inspection Reference - Complete list of Java inspections

References

Primary Texts:

- Bloch, J. (2018). *Effective Java* (3rd ed.). Addison-Wesley. Chapter 4: Classes and Interfaces; Chapter 9: General Programming.
- Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall. Chapter 3: Functions; Chapter 10: Classes.
- Horstmann, C. S. (2022). *Core Java, Volume I: Fundamentals* (12th ed.). Oracle Press. Chapter 5: Inheritance.

Tooling Documentation:

- IntelliJ IDEA Code Inspection — Official inspection documentation
- Java Code Inspection Reference — Complete list of all Java inspections
- Configuring Inspection Profiles — Managing inspection settings

Design Principles:

- Cyclomatic Complexity — McCabe complexity metric explanation
- Coupling (Computer Programming) — Class coupling concepts

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