a1a # java-fundamentals

Java Packages: Organizing Your Code

TCSS 305 Programming Practicum

This guide explains Java packages and why they're essential for organizing code in professional projects. You'll learn how packages work, the naming conventions that make them effective, and how to use them in the UW Bookstore project.

What is a Package?

A **package** in Java is a way to group related classes together. Think of it like a filing cabinet—instead of tossing all your papers into one drawer, you organize them into labeled folders that make sense.

```
Filing Cabinet (Your Project)
├── Business Documents (model package)
│   ├── Item.java
│   ├── Cart.java
│   └── StoreItem.java
├── User Interface (view package)
│   ├── BookstoreFrame.java
│   └── ItemRowPanel.java
└── Data Files (io package)
    └── InventoryLoader.java
```

But packages are more than just folders. They provide: - **Logical grouping** - Related classes live together - **Namespacing** - Prevents name collisions between classes - **Access control** - Controls what code can access what - **Clear architecture** - Shows how your code is organized at a glance

> **Note**
>
> Every Java class belongs to a package. If you don't declare one, the class goes into the "default package"—but professional code always uses explicit packages.

Why Packages Matter

1. Organization at Scale

Small programs with 2-3 classes don't need much organization. But real applications have hundreds or thousands of classes. Without packages, finding anything becomes a nightmare.

Imagine looking for a class called `Item` in a project with 500 files all in one folder. Now imagine looking for it when you know it's in `edu.uw.tcss.model`—you can go directly there.

2. Avoiding Name Collisions

What happens if two developers both create a class called `Item`? Without packages, you can only have one class with that name in your project.

With packages, no problem:

```
edu.uw.tcss.model.Item       // Our Item class
com.oracle.jdbc.Item         // Some database library's Item class
org.apache.commons.Item      // Another library's Item class
```

Each one is distinct because of its package. This becomes critical when using external libraries.

3. Access Control

Packages enable **package-private** visibility—you can share code within a package but hide it from the rest of the application. This is a powerful tool for encapsulation that we'll explore later in this guide.

4. Industry Standard

Every professional Java project uses packages. Every library you import is organized into packages. Learning to work with packages now prepares you for real-world development.

> **Tip**
>
> When you import a class like `java.util.ArrayList`, you're using packages. `java.util` is the package, and `ArrayList` is the class within it.

Package Naming Conventions

Java has specific conventions for naming packages, and following them is important for compatibility and professionalism.

The Reverse Domain Name Convention

Package names typically start with a reversed domain name:

Domain	Package Prefix
`uw.edu`	`edu.uw`
`google.com`	`com.google`
`apache.org`	`org.apache`

Why reversed? To guarantee global uniqueness. The University of Washington owns `uw.edu`, so any package starting with `edu.uw` belongs to UW. No one else will accidentally use the same prefix.

Our Course Package Structure

In TCSS 305, our base package is:

```
edu.uw.tcss
```

This breaks down as: - `edu` - Educational institution - `uw` - University of Washington - `tcss` - The Tacoma School of Computing and Software Systems

Naming Rules

Required conventions: - All lowercase letters - No underscores, hyphens, or special characters - Use dots to separate levels: `edu.uw.tcss.model` - Start with your organization's reversed domain

Examples:

Correct	Incorrect	Problem
`edu.uw.tcss.model`	`edu.uw.tcss.Model`	Uppercase letter
`edu.uw.tcss.view`	`edu.uw.tcss.user_interface`	Underscore

Correct	Incorrect	Problem
`com.example.utils`	`com.example.my-utils`	Hyphen

Packages and Directory Structure

Here's something crucial to understand: **packages correspond directly to directory structure**. The package name tells Java exactly where to find the class file.

The Mapping

```
Package:     edu.uw.tcss.model.StoreItem
Directory:   src/edu/uw/tcss/model/StoreItem.java
```

Each dot in the package name becomes a folder separator:

```
edu.uw.tcss.model
 │   │    │    │
 │   │    │    └── model/
 │   │    └─── tcss/
 │   └── uw/
 └── edu/
```

The Package Statement

Every Java file must declare its package at the very top:

```java
package edu.uw.tcss.model;

import java.math.BigDecimal;

public class StoreItem implements Item {
    // ...
}
```

Important

The package statement must be the first non-comment line in your file, and it must exactly match the directory structure. If `StoreItem.java` is in `src/edu/uw/tcss/model/`, the package must be `edu.uw.tcss.model`.

What Happens When They Don't Match

If the package declaration doesn't match the file location, you get a compilation error:

```
StoreItem.java:1: error: class StoreItem is public,
should be declared in a file named StoreItem.java
package edu.uw.tcss.model;  // File is actually in edu/uw/tcss/
^
```

IntelliJ will also show this as a red error and offer to fix it automatically.

Our Project's Package Structure

The UW Bookstore project is organized into five packages, each with a specific responsibility:

```
edu.uw.tcss
├── app      → Application entry point
├── model    → Business logic (what the app does)
├── view     → User interface (what users see)
├── io       → Input/output (reading files)
└── res      → Resources (constants, strings)
```

edu.uw.tcss.app

Purpose: Application startup

Contains: `BookstoreMain.java`

This is where the program starts. The `main()` method lives here, and it creates the window and loads initial data. It's kept separate so the rest of the code doesn't depend on how the application is launched.

edu.uw.tcss.model

Purpose: Core business logic

Contains: `Item`, `Cart`, `StoreItem`, `StoreBulkItem`, `StoreCart`, `ItemOrder`

This is the heart of the application—the classes that represent items, shopping carts, and orders. These classes don't know anything about the GUI or how data is stored. They just implement the business rules.

edu.uw.tcss.view

Purpose: Graphical user interface

Contains: `BookstoreFrame`, `ItemListPanel`, `ItemRowPanel`, and other GUI components

All the Swing components live here. These classes handle what the user sees and how they interact with the application.

edu.uw.tcss.io

Purpose: File input/output

Contains: `InventoryLoader`

Reading inventory data from files is separated into its own package. If we later wanted to load from a database or web service, we'd add classes here without touching the model or view.

edu.uw.tcss.res

Purpose: Constants and resources

Contains: `R.java`

String literals, colors, dimensions, and other constants are centralized here. This prevents magic numbers scattered throughout the code and makes it easy to change values in one place.

> **Note**
>
> This separation of concerns is called **layered architecture**. Each layer has a specific job, and the boundaries between them are clear. You'll see this pattern in most professional applications.

Import Statements

When you need to use a class from another package, you must **import** it.

Basic Import

```
package edu.uw.tcss.view;

import edu.uw.tcss.model.Item;      // Import a specific class
import edu.uw.tcss.model.ItemOrder;  // Import another class

public class ItemRowPanel {
    private Item myItem;           // Now we can use Item
```

```
    private ItemOrder myOrder;     // And ItemOrder
}
```

Why We Avoid Star Imports

You might see imports like this:

```
import edu.uw.tcss.model.*;  // Import everything from model package
```

This is called a **star import** or **wildcard import**. While it works, we avoid it in TCSS 305 (and Checkstyle will flag it) for several reasons:

1. **Clarity** - Explicit imports show exactly what your class depends on
2. **Avoiding conflicts** - If two packages have a class with the same name, star imports can cause ambiguity
3. **Documentation** - The imports serve as a quick reference for what external classes you're using

> **Warning**
>
> Checkstyle is configured to flag star imports as errors. Always import specific classes, not entire packages.

IntelliJ Auto-Import

IntelliJ makes importing easy:

1. Type a class name that needs importing (it appears in red)
2. Press `Alt+Enter` (Windows/Linux) or `Option+Enter` (macOS)
3. Select the correct class from the list

IntelliJ will add the import statement automatically.

Organizing Imports

Over time, imports can get messy—unused imports accumulate, and the order becomes random. IntelliJ can fix this:

- Press `Ctrl+Alt+O` (Windows/Linux) or `Cmd+Option+O` (macOS)
- Or use **Code > Optimize Imports**

This removes unused imports and sorts the remaining ones alphabetically.

Access Modifiers and Packages

Packages interact with Java's **access modifiers** to control visibility. Understanding this is key to good encapsulation.

The Four Access Levels

Modifier	Same Class	Same Package	Subclass	Everywhere
`private`	Yes	No	No	No
(none)	Yes	Yes	No	No
`protected`	Yes	Yes	Yes	No
`public`	Yes	Yes	Yes	Yes

Package-Private (Default) Visibility

When you don't specify any modifier, you get **package-private** visibility:

```
package edu.uw.tcss.model;

class Helper {  // No modifier = package-private
    void doSomething() {
        // ...
    }
}
```

This `Helper` class is visible to other classes in `edu.uw.tcss.model`, but invisible to classes in other packages. This is useful for:

- Implementation details that shouldn't be part of the public API
- Classes that only support other classes in the same package
- Internal utilities that shouldn't be used directly by external code

When to Use Each

Modifier	Use When
`public`	The class/method is part of your API—others should use it
`protected`	Subclasses need access, but general users don't
(default)	Only classes in the same package should use this
`private`	Only this class should access this member

Tip

Start with the most restrictive access that works. You can always make something more visible later, but making it less visible might break existing code.

Gen AI & Learning: Package Structure Decisions

AI coding assistants can help generate package declarations and import statements, but they often suggest generic structures like `com.example`. When using AI tools, always verify that generated code follows your project's established package conventions (e.g., `edu.uw.tcss.*` for this course). AI tools may also suggest star imports (`import java.util.*`) which violate our Checkstyle rules. Understanding *why* we organize packages the way we do helps you evaluate and correct AI suggestions.

Creating Packages in IntelliJ

IntelliJ makes package management straightforward.

Creating a New Package

1. Right-click on `src` (or an existing package) in the Project panel
2. Select **New > Package**
3. Enter the full package name (e.g., `edu.uw.tcss.util`)
4. IntelliJ creates all necessary directories

Creating a Class in a Package

1. Right-click on the package in the Project panel
2. Select **New > Java Class**
3. Enter the class name (IntelliJ adds the package statement automatically)

Moving Classes Between Packages

Sometimes you need to reorganize. Don't just drag files—use refactoring:

1. Right-click the class you want to move
2. Select **Refactor > Move** (or press `F6`)
3. Choose the destination package
4. IntelliJ updates:
5. The package statement in the file
6. All import statements in other files that reference this class
7. The file's location in the directory structure

> **Caution**
>
> Never manually move Java files between directories. Always use IntelliJ's Refactor > Move feature. Manual moves will break package declarations and imports, causing compilation errors.

Summary

Packages are fundamental to organizing Java code:

Concept	Key Point
Purpose	Group related classes, prevent name collisions, control access
Naming	Reverse domain (`edu.uw.tcss`), all lowercase, dots separate levels
Structure	Package names map directly to directory structure
Imports	Use specific imports, avoid star imports, let IntelliJ help
Access	Package-private (default) visibility is powerful for encapsulation

The UW Bookstore project demonstrates clean package organization: `app` for startup, `model` for business logic, `view` for GUI, `io` for file handling, and `res` for constants.

Further Reading

External Resources

- Oracle Java Tutorial: Packages - Official Java documentation
- Java Package Naming Conventions - Oracle's naming guidelines
- Baeldung: Guide to Java Packages - Practical tutorial with examples
- IntelliJ IDEA: Managing Dependencies - IDE documentation

References

Primary Texts:

- Horstmann, C. S. (2022). *Core Java, Volume I: Fundamentals* (12th ed.). Oracle Press. Chapter 4: Objects and Classes (Package section).
- Bloch, J. (2018). *Effective Java* (3rd ed.). Addison-Wesley. Item 15: Minimize the accessibility of classes and members.
- Reges, S., & Stepp, M. (2020). *Building Java Programs* (5th ed.). Pearson. Chapter 8: Classes (Package organization).

Language Documentation:

- Oracle Java Tutorial: Packages — Official package documentation
- Oracle Java Tutorial: Package Naming Conventions — Naming guidelines
- Oracle JDK 25: Controlling Access to Members — Access modifiers documentation

Tooling:

- IntelliJ IDEA: Managing Packages — IDE package management
- IntelliJ IDEA: Refactoring — Moving classes between packages

---

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