a1c # java-fundamentals

Sealed Types and Records

TCSS 305 Programming Practicum

This guide introduces two powerful features added to Java in recent versions: **sealed types** (Java 17) and **records** (Java 16). These features work together to create type-safe, maintainable code. You will encounter both in Assignment 1C, where the provided `Item` interface is sealed and the `ItemOrder` and `CartSize` types are records.

Why These Features Matter

Before Java 16 and 17, developers faced two common challenges:

1. **Uncontrolled inheritance:** Any class could extend your class or implement your interface, making it impossible to reason about all possible subtypes.
2. **Boilerplate for data classes:** Simple data-carrying classes required writing constructors, getters, `equals()`, `hashCode()`, and `toString()`—often hundreds of lines for what should be a simple concept.

Sealed types and records solve these problems elegantly. Even better, they work together beautifully: sealed hierarchies with record implementations create concise, type-safe domain models.

Part 1: Sealed Types

The Problem: Uncontrolled Hierarchies

Consider the `Item` interface from our bookstore application. Without restrictions, anyone could create new implementations:

```
// Without sealed: anyone can implement Item
public interface Item {
    String getName();
    BigDecimal getPrice();
}

// Someone creates an unexpected implementation
```

```
public class DigitalItem implements Item {
    // Now exists in the codebase...
}

// Someone else creates another
public class GiftCardItem implements Item {
    // And another...
}
```

This creates several problems:

- **Code reasoning:** You cannot know all possible `Item` types. Switch statements or if-else chains might miss cases.
- **Maintenance:** Adding a new method to `Item` requires updating unknown implementations scattered across the codebase.
- **API design:** Library authors cannot control how their types are extended.

The Solution: sealed Keyword

The `sealed` keyword restricts which classes can extend a class or implement an interface:

```
public sealed interface Item permits StoreItem, StoreBulkItem {
    String getName();
    BigDecimal getPrice();
}
```

This declaration says: "`Item` can only be implemented by `StoreItem` and `StoreBulkItem` — and nothing else."

> **Important**
>
> The `permits` clause explicitly lists every allowed subtype. The compiler enforces this restriction—attempting to implement `Item` from an unlisted class produces a compilation error.

Permitted Subtypes: Three Choices

Every class that extends a sealed class or implements a sealed interface must be one of:

Modifier	Meaning
`final`	This class cannot be extended further
`sealed`	This class is also sealed with its own `permits` clause

Modifier	Meaning
`non-sealed`	This class opens the hierarchy—anyone can extend it

In Assignment 1C, the hierarchy looks like this:

```
// Sealed interface - only AbstractItem can implement it
public sealed interface Item permits AbstractItem {
    // ...
}

// Sealed abstract class - only StoreItem and StoreBulkItem can extend it
public sealed abstract class AbstractItem implements Item
        permits StoreItem, StoreBulkItem {
    // ...
}

// Final classes - the hierarchy ends here
public final class StoreItem extends AbstractItem {
    // ...
}

public final class StoreBulkItem extends AbstractItem {
    // ...
}
```

This creates a **closed hierarchy**: the compiler knows every possible `Item` type.

Why AbstractItem Permits Only Two Subclasses

The design decision to limit `AbstractItem` to exactly `StoreItem` and `StoreBulkItem` is intentional:

1. **Domain completeness:** These two classes represent the complete pricing model—items with simple pricing and items with bulk discounts. No other pricing model is needed.
2. **Type safety:** Code that handles items can exhaustively cover all cases without a catch-all "else" clause.
3. **Future-proofing:** If a new item type is needed, the hierarchy must be explicitly modified. This forces a conscious design decision rather than ad-hoc additions.

> **Note**
>
> The sealed hierarchy reflects a design principle: your domain model should represent exactly what exists in your problem space—no more, no less. An item is either a regular store item or a bulk item. There is no third option.

Pattern Matching (Preview)

One powerful benefit of sealed types is **exhaustive pattern matching**. When the compiler knows all possible subtypes, it can verify that switch expressions handle every case:

```
// With sealed types, the compiler knows all cases are covered
public String describeItem(Item item) {
    return switch (item) {
        case StoreItem s -> "Regular item: " + s.getName();
        case StoreBulkItem b -> "Bulk item: " + b.getName();
        // No default needed - compiler knows these are all possibilities
    };
}
```

> **Looking Ahead**
>
> Pattern matching with sealed types becomes very powerful in later assignments. For now, understand that sealed hierarchies enable the compiler to help you write correct code by ensuring you handle every case.

Part 2: Records

The Problem: Boilerplate-Heavy Data Classes

Before records, a simple data class required substantial boilerplate:

```
// Traditional data class - lots of code for a simple concept
public final class ItemOrder {
    private final Item item;
    private final int quantity;

    public ItemOrder(Item item, int quantity) {
        this.item = item;
        this.quantity = quantity;
    }

    public Item getItem() {
        return item;
    }

    public int getQuantity() {
        return quantity;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (!(obj instanceof ItemOrder other)) return false;
```

```
        return quantity == other.quantity &&
               Objects.equals(item, other.item);
    }

    @Override
    public int hashCode() {
        return Objects.hash(item, quantity);
    }

    @Override
    public String toString() {
        return "ItemOrder[item=" + item + ", quantity=" + quantity + "]";
    }
}
```

That is approximately 35 lines for a class that just holds two values. Most of this code is mechanical and error-prone.

The Solution: Records

Records provide a concise syntax for immutable data carriers:

```
public record ItemOrder(Item item, int quantity) { }
```

This single line generates:

- A `private final` field for each component (`item` and `quantity`)
- A **canonical constructor** that initializes all fields
- **Accessor methods** named after the components (not `getItem()`, but `item()`)
- `equals()`, `hashCode()`, and `toString()` based on all components

> **Accessor Naming Convention**
>
> Record accessors use the component name directly:
>
> - **Record:** `order.item()` and `order.quantity()`
> - **Traditional:** `order.getItem()` and `order.getQuantity()`
>
> This is intentional—records use a different naming convention than JavaBeans-style getters.

Automatic Implementations

Records automatically implement the equals/hashCode contract correctly:

```
record ItemOrder(Item item, int quantity) { }
```

```
ItemOrder order1 = new ItemOrder(pen, 5);
ItemOrder order2 = new ItemOrder(pen, 5);

order1.equals(order2);  // true - same item and quantity
order1.hashCode() == order2.hashCode();  // true - contract satisfied
```

The generated `toString()` provides useful output:

```
System.out.println(order1);
// Output: ItemOrder[item=StoreItem[name='Pen', price=1.99], quantity=5]
```

> **Testing Records**
>
> Because records auto-generate `equals()` and `hashCode()`, you typically do not need to test the equals/hashCode **contract** (reflexivity, symmetry, transitivity). That would be testing the Java compiler. However, you may write tests that document expected equality behavior as specification tests.

Compact Constructors for Validation

Records support **compact constructors** for validation without redundant parameter declarations:

```
public record ItemOrder(Item item, int quantity) {
    // Compact constructor - no parameter list
    public ItemOrder {
        Objects.requireNonNull(item, "Item cannot be null");
        if (quantity < 0) {
            throw new IllegalArgumentException("Quantity cannot be negative");
        }
        // Fields are automatically assigned after validation
    }
}
```

Notice the compact constructor has no parentheses—it implicitly receives the same parameters as the canonical constructor. The field assignments happen automatically after the constructor body executes.

Compare this to a traditional constructor with explicit assignments:

```
// This also works, but is more verbose
public record ItemOrder(Item item, int quantity) {
    public ItemOrder(Item item, int quantity) {
        Objects.requireNonNull(item, "Item cannot be null");
        if (quantity < 0) {
            throw new IllegalArgumentException("Quantity cannot be negative");
        }
        this.item = item;      // Must explicitly assign
```

```
        this.quantity = quantity;  // Must explicitly assign
    }
}
```

> **Tip**
>
> Use compact constructors when you only need validation. The automatic field assignment reduces errors and keeps the focus on the validation logic.

Records Are Immutable

Records are inherently **immutable**—there are no setters, and fields are final:

```
ItemOrder order = new ItemOrder(pen, 5);
order.quantity = 10;  // Compilation error - fields are final
```

To "change" a record, you create a new instance:

```
ItemOrder updated = new ItemOrder(order.item(), 10);
```

Immutability provides significant benefits:

- **Thread safety:** Immutable objects can be shared across threads without synchronization
- **Predictability:** The object's state cannot change unexpectedly
- **Simplicity:** No defensive copies needed when passing records around

When to Use Records vs Regular Classes

Use **records** when:

- The class is primarily a data carrier
- Immutability is desired or acceptable
- You want automatic `equals()`, `hashCode()`, and `toString()`
- The class has no complex behavior beyond holding data

Use **regular classes** when:

- You need mutable state
- You need to customize which fields participate in equality
- You need inheritance (records are implicitly `final`)
- The class has significant behavior beyond data access

Use Case	Type
`ItemOrder` (holds an item and quantity)	Record
`CartSize` (holds two counts)	Record
`StoreItem` (has pricing logic, inheritance)	Class
`StoreCart` (mutable collection, complex behavior)	Class

Part 3: Working Together

Sealed types and records complement each other beautifully. Consider how they work together in Assignment 1C:

Sealed Hierarchy with Mixed Types

```
// Sealed interface defines the contract
public sealed interface Item permits AbstractItem {
    String getName();
    BigDecimal getPrice();
    BigDecimal calculateTotal(int quantity, boolean useMembershipPricing);
}

// Sealed abstract class provides common implementation
public sealed abstract class AbstractItem implements Item
        permits StoreItem, StoreBulkItem {
    // Shared fields and methods
}

// Final classes complete the hierarchy
public final class StoreItem extends AbstractItem { }
public final class StoreBulkItem extends AbstractItem { }

// Records for data transfer
public record ItemOrder(Item item, int quantity) { }
public record CartSize(int itemOrderCount, int itemCount) { }
```

Design Insights

This design illustrates several principles:

1. **Sealed for behavior, records for data:** `Item`, `StoreItem`, and `StoreBulkItem` have significant behavior (pricing calculations), so they use the sealed class hierarchy.

`ItemOrder` and `CartSize` are simple data carriers, so they use records.

2. **Closed where needed:** The item type hierarchy is closed because the pricing model is fixed. The cart can hold any `Item` without restriction.
3. **Immutable data flow:** `ItemOrder` and `CartSize` records are immutable, ensuring data integrity as they move through the system.

> **Gen AI & Learning: Understanding Modern Java Features**
>
> When working with AI coding assistants, understanding sealed types and records helps you:
>
> - **Read AI-generated code** that uses these modern features correctly
> - **Ask better questions** about type hierarchies and data modeling
> - **Evaluate suggestions** for whether sealed or record is appropriate
>
> AI tools can explain how these features work, but **you** must understand when to apply them. The decision between sealed classes, records, and traditional classes requires domain knowledge that you develop through practice.

Sealed Interfaces with Record Implementations

Later in the quarter, you will encounter a common pattern in event-driven programming: a sealed interface that defines an event type, with records implementing each specific event kind.

```
public sealed interface GameEvent permits MoveEvent, ScoreEvent, GameOverEvent
{
    long timestamp();
}

public record MoveEvent(long timestamp, int x, int y) implements GameEvent { }
public record ScoreEvent(long timestamp, int points) implements GameEvent { }
public record GameOverEvent(long timestamp, boolean won) implements GameEvent
{ }
```

This pattern combines the benefits of both features:

- **Sealed interface:** Ensures exhaustive handling of all event types
- **Records:** Provide immutable, concise event data with automatic equality

> **Looking Ahead**
>
> You'll see this pattern in the group project when implementing game events with `PropertyChangeListener`.

Assignment 1C Context

In Assignment 1C, you will work with:

Provided Records (Do Not Implement)

- `ItemOrder(Item item, int quantity)` — Represents an order for a specific item
- `Cart.CartSize(int itemOrderCount, int itemCount)` — Holds cart size information

These are provided for you. Use them by calling their accessors:

```
ItemOrder order = new ItemOrder(someItem, 5);
Item theItem = order.item();     // NOT getItem()
int qty = order.quantity();       // NOT getQuantity()

Cart.CartSize size = cart.getCartSize();
int orders = size.itemOrderCount();
int total = size.itemCount();
```

Sealed Hierarchy (You Implement Parts)

- `Item` interface (provided, sealed)
- `AbstractItem` abstract class (you complete the stub)
- `StoreItem` final class (you implement)
- `StoreBulkItem` final class (you implement)

Your classes must use `final` because the sealed hierarchy requires it:

```
public final class StoreItem extends AbstractItem {
    // Your implementation
}
```

Common Mistakes

1. Using get-Prefix with Record Accessors

```
// WRONG - records don't use get prefix
int qty = order.getQuantity();

// CORRECT - use component name directly
int qty = order.quantity();
```

2. Forgetting final on Sealed Subtypes

```
// WRONG - compiler error in sealed hierarchy
public class StoreItem extends AbstractItem { }

// CORRECT - must be final, sealed, or non-sealed
public final class StoreItem extends AbstractItem { }
```

3. Trying to Extend a Record

```
// WRONG - records are implicitly final
public class SpecialOrder extends ItemOrder { }

// CORRECT - compose instead
public record SpecialOrder(ItemOrder baseOrder, String specialNote) { }
```

4. Attempting to Mutate Record Fields

```
// WRONG - records are immutable
order.quantity = 10;

// CORRECT - create a new record
ItemOrder updated = new ItemOrder(order.item(), 10);
```

Summary

Concept	Key Point
Sealed types	Restrict which classes can extend/implement a type
permits clause	Explicitly lists all allowed subtypes
final/sealed/non-sealed	Every permitted subtype must declare one of these
Records	Concise syntax for immutable data carriers
Record accessors	Use `item()` not `getItem()`
Compact constructors	Validate without explicit field assignment
When to use records	Data carriers with value-based equality

Concept	Key Point
When to use sealed	Controlled hierarchies with exhaustive handling

Sealed types and records represent modern Java's approach to safer, more expressive code. Sealed types ensure type hierarchies are well-defined and complete. Records eliminate boilerplate for data classes while guaranteeing immutability and correct equality semantics. Together, they enable cleaner domain models that are easier to reason about and maintain.

Further Reading

External Resources

- JEP 395: Records - The official Java Enhancement Proposal for records
- JEP 409: Sealed Classes - The official Java Enhancement Proposal for sealed classes
- Oracle: Record Classes - Official tutorial on records
- Oracle: Sealed Classes and Interfaces - Official tutorial on sealed types
- Baeldung: Java 14 Records - Practical tutorial with examples
- Baeldung: Sealed Classes - Practical tutorial on sealed types

---

References

Primary Texts:

- Horstmann, C. S. (2022). *Core Java, Volume I: Fundamentals* (12th ed.). Oracle Press. Chapter 4: Objects and Classes — Sections on records and sealed classes.
- Bloch, J. (2018). *Effective Java* (3rd ed.). Addison-Wesley. Item 17: Minimize mutability (records embody this principle).

Language Specifications:

- JEP 395: Records — Final specification for record classes (Java 16)
- JEP 409: Sealed Classes — Final specification for sealed classes (Java 17)

Language Documentation:

- Oracle JDK 25: Record Classes — Official language guide for records

- Oracle JDK 25: Sealed Classes and Interfaces — Official language guide for sealed types
- Oracle JDK 25: java.lang.Record — Record class API documentation

---

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