

CS2030 Programming Methodology II
AY23/24, Y1S2
Notes

Sim Ray En Ryan

December 3, 2025

Contents

| | | |
|----------|--|----------|
| 1 | Object-Oriented Programming (OOP) | 3 |
| 1.1 | Pillars of OOP | 3 |
| 1.2 | Pros and Cons of OOP | 3 |
| 2 | Programming Refresher and Basics | 3 |
| 2.1 | JShell Commands | 3 |
| 2.2 | Data Types and Initialization | 3 |
| 2.3 | Classes and Methods | 4 |
| 3 | Immutability, Generics, and Functional Concepts | 4 |
| 3.1 | Immutability and Side Effects | 4 |
| 3.2 | Java Generics and Type Variance | 5 |
| 3.3 | Pure Functions and Composition | 5 |
| 4 | Interfaces, Inheritance, and Subclasses | 6 |
| 4.1 | Interfaces | 6 |
| 4.2 | Inheritance | 6 |
| 4.3 | Object Class Methods | 6 |
| 4.4 | Abstract Classes | 6 |
| 5 | Advanced Concepts: Contexts, Streams, and Concurrency | 7 |
| 5.1 | Against null and Optional | 7 |
| 5.2 | Abstract Classes | 7 |
| 6 | Advanced Concepts: Contexts, Streams, and Concurrency | 7 |
| 6.1 | Against null and Optional | 7 |
| 6.2 | Java Streams | 8 |
| 6.3 | Error Handling | 8 |
| 6.4 | Asynchronous Programming | 9 |
| 7 | Software Design Principles (SOLID) | 9 |
| 8 | Memory Management | 9 |

1 Object-Oriented Programming (OOP)

1.1 Pillars of OOP

- **Encapsulation:** Hides internal details of an object and restricts direct access to its components. Provides data protection by reducing the risk of interference.
- **Inheritance:** Allows a class to inherit properties from another class, promoting reusability.
- **Abstraction:** Simplifies complex systems through abstract types that are meant to be extended by subclasses. Deals with variables locally (Abstraction Barrier).
- **Polymorphism:** Allows different objects to be treated the same way, typically through method overloading and overriding. Which method is invoked is decided during run-time.

1.2 Pros and Cons of OOP

- **Pros:**
 - **Modularity:** Easier to maintain, update, and extend.
 - **Data Protection:** Achieved via encapsulation (e.g., `private` fields), reducing risk of interference.
- **Cons:**
 - **Performance Overhead:** Due to abstraction layers.
 - **Inheritance Issues:** Such as the Diamond Problem.

2 Programming Refresher and Basics

2.1 JShell Commands

The JShell tool offers commands for interactive programming:

- `/help`, `/exit`, `/reset`, `/reload`.
- `/list`, `/drop`, `/vars`, `/types`, `/imports`, `/methods`.
- `/edit`, `/save`, `/open`, `/history`, `/set`.

2.2 Data Types and Initialization

- **Primitive Types:** `Boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, `char`.

- **Subtyping (Primitives):** `byte <: short <: int <: long <: float <: double`. Also `char <: int`.
- **Initialization:** Primitive types are initialized to 0. Non-primitive types are initialized to null.
- **Constants (Class Fields):** Declared as `private final <type> <VAR_NAME> = jval`.

2.3 Classes and Methods

- **Class Components:** Classes have fields and methods, which create instances (objects).
- **Access Modifiers:**
 - `private`: Can only be used within the same class.
 - `public`: Can be used by all classes.
 - `protected`: Gives access within the same package. Use when you want to show child-parent relationship.
 - `default/package`: Access is restricted to the same package.
- **Constructor Method:** Used with the `new` keyword to instantiate an object, has no return type. Constructors are static.
- **Accessor and Mutator:** `Gets` vs. `Sets` methods. Encourages the "Tell, Don't Ask" principle by making computation methods instead of calculating outside (Abstraction Barrier).
- **Main Method:** Declared as `public final static void main(String[] args) {}`. Operators are also static.
- **Method Overloading:** Having methods with the same name but a different number or type of parameters. Works for class constructors.

3 Immutability, Generics, and Functional Concepts

3.1 Immutability and Side Effects

- **Immutability Pros:** No side effects, focus on input/output relation, reusability, easier to audit, and easier to parallelize (deterministic).
- **Immutability Cons:** Requires maintaining copies of the data structure incrementally.
- **Effect-Free Programming:** Achieving immutability by making every method return a new instance of the object.

- **Aliasing:** Avoid sharing references to prevent interference, which may require more resources.

3.2 Java Generics and Type Variance

- **Parameterized Types:** E.g., `ImList<E>`, `ImList<String>`.
- **Wrapper Classes:** Primitive types cannot be type arguments, so they must be put into Wrapper classes (e.g., `int` \rightarrow `Integer`).
- **Autoboxing and Auto Unboxing** handle automatic conversion between primitives and their wrappers.
- **Generics are Invariant:** `ImList<Integer>` cannot be assigned to `ImList<Object>`, even though `Integer` $<:$ `Object`.
- **Upper Bounded Wildcard** (`? extends E`): Used for **Reading** (covariant). If `S` $<:$ `T`, then `C` $< S$ $<:$ `C` $< ? \text{ extends } T$. Example: `List<Circle>` can store objects that extend `Shape`, but cannot store non-shapes.
- **Lower Bounded Wildcard** (`? super E`): Used for **Writing** (contravariant). If `S` $<:$ `T`, then `C` $< T$ $<:$ `C` $< ? \text{ super } S$. Example: `ImList<Esort(Comparator` $< ? \text{ Super } E \text{ cmp}) allows the comparator to compare objects that are supertypes of `E`.$

3.3 Pure Functions and Composition

- **Pure Function:** A function that is **deterministic** (always produces the same output for the same input) and has **no side effects** (e.g., modifying external state, throwing exceptions).
- **Referential Transparency:** An expression can always be replaced by its value if there are no side effects.
- **Function Composition:**
 - `f.andThen(g).apply()`: Default `<VFunction` $< T, V \text{ andThen } (Function$ $< ? \text{ Super } R, ? \text{ extends } V \text{ after})$. Applies `f`, then `g` to the result of `f`.
 - `g.compose(f).apply()`: Default `<VFunction` $< V, R \text{ andThen } (Function$ $< ? \text{ Super } V, ? \text{ extends } T \text{ before})$. Applies `f`, then `g` to the result of `f`.
- **Currying:** Transforming a function that takes multiple arguments (e.g., `x + y`) into a sequence of functions, each taking a single argument (e.g., `x` \rightarrow (`y` \rightarrow `x + y`)).
- **Handling Non-Pure functions:** Use `Maybe/Optional`, `ImList`, `Stream`, or `Try`.

4 Interfaces, Inheritance, and Subclasses

4.1 Interfaces

- An interface defines a **contract** between the client and the implementer.
- Classes **implement** multiple interfaces (e.g., `class Circle implements Shape`).
- Interface methods are inherently **public** and must have the same return type when implemented.
- Methods that return the interface type (e.g., `Moveable`) may lose specific method access (e.g., cannot call `getArea()` afterward).
- **Java APIs:** Includes `List`, `Collection`, `Iterable` (good to know for exams), and `Comparator`.

4.2 Inheritance

- Classes **extend** (inherit from) only one parent/superclass (e.g., `class coolerObject extends Object`).
- **Substitutability (LSP):** A child class should be replaceable with its parent. If $S <: T$, then every T can actually be an S without changing the expected behavior.
- `super()`: Calls the parent's constructor.
- **Forwarding:** Using `object.getArea()` instead of `object.circle.getArea()` to delegate method calls.

4.3 Object Class Methods

Every class inherits from the root `Object` class.

- `equals(Object obj)`: Usually compares memory location. Should be overridden to check for the same type (

4.4 Abstract Classes

- Cannot be instantiated by itself.
- Can have properties inherited by child classes.
- Can have some methods defined (unlike interfaces).

5 Advanced Concepts: Contexts, Streams, and Concurrency

5.1 Against null and Optional

- `null` is a subtype of every non-primitive type and should not be used as an intended return type.
- `Optional<T>` (instance of `Type type`) and field equality.
- `toString()`: Usually returns `Object@Location`. Should be overridden to provide a meaningful string representation.
- **Method Override**: Redefining a parent's method in the child class (using the `@Override` annotation for compiler hints). The original can still be called using `super.toString()`.

5.2 Abstract Classes

- Cannot be instantiated by itself.
- Can have properties inherited by child classes.
- Can have some methods defined (unlike interfaces).

6 Advanced Concepts: Contexts, Streams, and Concurrency

6.1 Against null and Optional

- `null` is a subtype of every non-primitive type and should not be used as an intended return type.
- `Optional<T>` (or a custom `Maybe<T>`) represents a value that may contain 1 T or nothing.
- **Creation**: `Optional.of(T value)`, `Optional.empty()`, or `Optional.ofNullable(T value)`.
- **High-Level Operations** (preferred):
 - * **filter**: Takes a `Predicate<T>` (Boolean Predicates). Returns the same `Optional` if the predicate passes, or `Optional.empty()` otherwise.
 - * **map**: Takes a `Function<T, R>`. Transforms the content from T to R , or returns `empty()`.

- * `orElse(T other)`: Eager evaluation (terminal operation). Will evaluate the default value regardless of whether the `Optional` is empty.
- * `orElseGet(Supplier)`: Lazy evaluation (intermediate/source operation). Invoked only when the `Optional` is empty.
- **Low-Level Operations** (`get()`, `isPresent()`) should be avoided.

6.2 Java Streams

Streams provide a functional way to process sequences of elements.

- **Structure**: Data Source \rightarrow Intermediate Operations \rightarrow Terminal Operation.
- **Data Source**: `IntStream.rangeClosed(1, 10)` or `range` (exclusive end). Also `iterate(Seed, f)`.
- **Intermediate Operations**: Return a new stream. Includes `map`, `flatMap`, `filter(func)`, `sorted(comp)`, `limit(long)`, `skip(long)`, and specialized mapping (`mapToInt`, `mapToDouble`, etc.).
- **Terminal Operations**: Reduce the stream to a single value or produce a side effect. Includes `sum()`, `min(comp)`, `max(comp)`.
 - * `reduce(identity, op)`: Reduces stream elements using a binary operator, starting with `identity`.
 - * **Boolean**: `allMatch(pred)`, `anyMatch(pred)`, `noneMatch(pred)`. `;/itemize`
 - * **Parallelization**: `.parallel()` and `.sequential()`. When parallel and reducing, the function must be associative.
 - * **Correctness**: Do not modify stream data; streams should be stateless.

6.3 Error Handling

- * **Try-Catch Block**: Used to handle exceptions (e.g., `try {...} catch (FileNotFoundException e) { ... }`). Exception handling is considered a side effect. The `finally` block executes regardless.
- * **Checked Exceptions**: Must be declared using `throws`.
- * **Self-Created Exceptions**: Can extend built-in exception types (e.g., `class IllegalArgumentException extends RuntimeException`).
- * **Throwing Constraint**: A child class cannot throw a more general exception than its parent.

6.4 Asynchronous Programming

- * **Threads:** Uses the `Thread` class with a `Runnable` object (which has a `void run()` method).
- * **Synchronization:** `t.start()` executes the thread. `t.join()` blocks the code until the thread `t` is finished.
- * **CompletableFuture<T> (Callbacks):** Used for asynchronous tasks:
 - **Source:** `supplyAsync(Supplier()), runAsync(Runnable())`.
 - **Map (thenApply):** `thenApply(Function())` or `thenApplyAsync(Function())`.
 - **FlatMap (thenCompose):** `thenCompose(Function())`.
 - **Combine:** `thenCombine(CompletionStage<? Extends U>, BiFunction(T, U, V))`.
 - **Side Effect:** `thenAccept(Consumer<? Super T> Action)`.

7 Software Design Principles (SOLID)

- * **Single Responsibility Principle (SRP):** Each class should have only one responsibility.
- * **Open/Closed Principle (OCP):** Software should be **open for extension** but **closed for modification**.
- * **Liskov Substitution Principle (LSP):** Subclasses should be able to function as their base classes without changing it.
- * **Interface Segregation Principle (ISP):** Clients should not depend on interfaces they do not need. Large interfaces should be split.
- * **Dependency Inversion Principle (DIP):** High-level modules should not depend on low-level modules; both should depend on abstractions.

8 Memory Management

Variable storage in Java:

- * **Stack:** Used for method calls (LIFO).
- * **Heap:** Used for objects (those called with the `new` keyword).
- * **Metaspace:** Used for static variables and methods.