CS2030 Programme Methodology II

AY2021/22 Semester 1

1. Object-Oriented-Programming (OOP)

OO Principles

(1) Abstraction

- Data Abstraction + Functional Abstraction
- Implementor defines the data/functional abstractions using lower-level data and processes.
- Client uses high-level data types and methods.
- Interaction between two objects is viewed as communication across an abstraction barrier.

(2) Encapsulation

- Packaging: Package related data and behaviour in a self-contained unit.
- Information Hiding: Hide information/data from the client and allow access only through methods provided.

Good OOP Degisn

(1) Tell-Don't-Ask Principle

- Tell an object what to do; Don't ask an object for data (e.g. accessors).

(2) Immutability of Objects

- Make all instance fields private final to encapsulate data and prevent mutation.

(3) Avoid Cyclic Dependencies

2. Inheritance

Inheritance

(1) "is-a" Relationship

(2) `super` Keyword

- `super(…)` to access parent's constructor.
- `super.x` refers to parent's property.
- `super.foo()` refers to parent's method.

(3) `protected` Modifier

- `protected` gives access to properties/methods to all other classes (including sub-classes) within the same package.

(4) Override a Method

- Explicitly redefining a method in a sub-class overrides the same method from its super-class.
- The annotation `@Override` indicates to the compiler that the method overrides the same one in the parent class.

(5) Overload a Method

- Methods of the same name can co-exist if their method signatures (number, type, order of arguments) are different.

3. Polymorphism

Liskov Substitution Principle

If S is a sub-class of T, then an object of type T can be replaced by that of type S without changing the desirable property of the programme.

Polymorphism

(1) Compile-Time Type VS Run-Time Type

- Consider the code `A a = new AA();`:
  The compile-time type of `a` is `A`, so it can call all methods of `A`; The run-time type of a is `AA`, so it will run all methods of `AA`.

(2) Static Binding in Overloading and Dynamic Binding in Overriding

4. Abstract Class and Interface

Concrete Class, Abstract Class and Interface

(1) Concrete class is the actual implementation.

(2) Interface is a contract specifying the abstraction between what the client can use and what the implementor should provide.

(3) Abstract class is a trade-off between the two, typically used as a base class.

SOLID Principles in OO Design

(1) Single-Responsibility Principle: A class should have only one reason to change.

(2) Open-Closed Principle: Open for extension; Closed for modification.

(3) Liskov Substitution Principle

(4) Interface Segregation Principle: Clients should not know of methods they do not need (which interface should be visible).

(5) Dependency Inversion Principle: Program to an interface, not an implementation.

5. Java Collections

Java Collection: `ArrayList<T>`

(1) OO Principles

- Abstraction: Methods that organise, store and retrieve data.
- Encapsulation: How data is being stored is hidden.

(2) `ArrayList<A>` is called a parametrised type.

(3) Auto-boxing and Unboxing

- Passing primitive types into a collection causes it to be auto-boxed.
- Assigning boxed types to a primitive type causes it to be unboxed.

(4) `ArrayList<AA>` is not a sub-type of `ArrayList<A>`, while `AA[]` is a sub-type of `A[]`.

Interface: `List<T>`

(1) `List<T>` extends `Collection<T>`.

(2) Converting from an Array to a List:

- `List<Integer> lst = Arrays.asList(arr);`
- Converting from an primitive array to a list requires every element to be boxed:
  `Arrays.stream(arr).boxed().collect(Collectors.toList());`
  Or convert the array to a boxed array first:
  `Arrays.stream(arr).boxed().toArray(Integer[]::new);`

(3) Converting from a List to an Array:

- `lst.toArray()` returns an array of Object.
- `lst.toArray(new Integer[0])` returns an array of Integer.

(4) Sorting

- Natural Order: `Comparable<T>`
- Comparator: `Comparator<T>`

6. Java Keywords, Exception Handling and Assertions

<u>Keywords</u>

(1) `static`

- Define constants
- Define aggregated data
- `static` methods belong to the class instead of an object.
- No overriding since `static` methods resolved at compile time.
- `static` fields/methods should be called through the class instead of instances.

(2) `enum`

- An `enum` is a special type of class used for defining constants.

(3) `final`

- Explicitly prevent overriding

<u>Exception Handling</u>

(1) `throw` the Exception Out

(2) Handle the Exception (`try… catch… finally…`)

(3) Checked Exception VS Unchecked Exception

- A checked exception is one that the programmer should actively anticipate and handle.
- An unchecked exception is one that is unanticipated, usually the result of a bug.
- Unchecked exceptions are sub-classes of `RuntimeException`. All `Error` are also unchecked.

<u>Assertions</u>

(1) Exceptions are used to handle user mishaps, while assertions are used to identify bugs during programme development.

(2) Expression

- `assert boolean_expression;`
- `assert boolean_expression : string_expression;`

(3) `-ea` Flag

7. Generics

<u>Defining Functionilty</u>

(1) Concrete Class

(2) Lambda Expression

- `(parameterList) -> (Statements)`
- Variables used can be from class properties (`static`), instance properties and final or effectively final local variables.

(3) Anonymous Class from Functional Interface

<u>Wildcards</u>

(1) Unbounded Wildcards

- `ImList<?>` can refer to all types of `ImList`.

(2) Bounded Wildcards

- PECS: Producer `extends`; Consumer `super`.

(3) Variance of Types

- Covariant:
  `C <: S` → `C[] <: S[]`
  `C <: B` → `ImList<C> <: ImList<? Extends B>`
- Invariant:
  `C <: S` ↛ `ImList<C> <: ImList<S>` or `ImList<S> <: ImList<C>`
- Contravariant:
  `ArrayList <: List` → `ArrayList<C> <: List<C>`
  `B <: F` → `ImList<F> <: ImList<? Super B>`

8. Declarative Programming

<u>`Optional` to Manage Missing/Null Values</u>

(1) Common Methods

- `empty()`
- `filter(Predicate<? super T> pred)`
- `flatMap(Function<? super T, ? extends Optional<? Extends U>> mapper)`
- `ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction)`
- `map(Function<? super T, ? extends U> mapper)`
- `ofNullable(T value)`
- `orElseGet(Supplier<? Extends T> supplier)`

(2) Tell-Don't-Ask Principle: Avoid using `get()`, `isPresent()`, `isEmpty()`

<u>`Stream` to Manage Iteration</u>

(1) Stream elements within a stream can only be consumed once.

(2) Data Source (lazy evaluation)

- `IntStream.range()`
- `Stream<Integer>.iterate()`

(3) Operations

- Terminal Operation: Reduce the stream of values into a single value (eager evaluation).
- Intermediate Operation: Specify tasks to perform on a stream's elements (lazy evaluation).
- Lazy evaluation allows us to work with infinite streams (e.g. `iterate(T seed, Function<T, T> next)`, `generate(Supplier<T> supplier)`). Intermediate operations can be used to restrict the total number of elements inside the stream (e.g. `limit()`).

(4) Parallelism

- Avoid parallelising trivial tasks because they creates more work in terms of parallelising overhead.
- Stream operations must not interfere with stream data.
- Stream operations are preferably stateless with no side effects.

(5) Associative Accumulating Function

- `<U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner)`
- Rules to follow when parallelising:
  (a) `combiner.apply(identity, i)` must be equal to `i`.
  (b) combiner and accumulator must be associative.

(c) combiner and accumulator must be compatible, i.e. `combiner.apply(u, accumulator.apply(identity, t))` must be equal to `accumulator.apply(u, t)`.

9. Lazy

Caching

(1) `Supplier` to Handle Delayed Data

(2) `Optional` to Store Cache Value

LazyList

(1) `head`: `() -> value`

(2) `tail`: `() -> LazyList`

10. Asynchronous Programming

Fork and Join

(1) If Task A and B does not produce side effects, we can fork Task A to execute at the same time as Task B and join back Task A later.

(2) Callback: A callback is any executable code that is passed as an argument to other code so that the former can be called back after the latter completes.

`CompletableFuture`

(1) Static Constructors

- `RunAsync` with `thenRun` for `Runnables`
- `SupplyAsync` with `thenApply` for `Suppliers` with returned values

(2) Callback Methods

- `thenAccept(Consumer<? super T> action)`
- `thenApply(Function<? super T, ? extends U> fn)`
- `thenCompose(Function<? super T, ? extends CompletionStage<U>> fn)`
- `thenCombine(CompletionStage<? extends U> other, BiFunction<? super T, ? super U, ? extends V> fn)`

(3) `join` Method

- Returns the result when execution completes.