CS2030 Summary Notes

Author: Lim Wei Liang

Semester Taken/Lecturer: AY20/21 Special Term 1 under Dr Boyd Anderson

Note: Content taught is **equivalent** to that of **2030S AY20/21 Semester 2**

References: (See here for more details)

CS2030 Programming Methodology II

AY 2020/21 Special Term Part 1 This website hosts the lecture notes, lecture slides, and other written guides about CS2030. You can explore the links on the menu on the left, or search for keywords in the search bar above. Other Critical Web Resources Piazza for Q&A and discussions LumiNUS for all other information (administration, logistic) and tools

https://nus-cs2030-2021-s3.github.io/notes/index.html

The Java™ Tutorials

Not sure where to start? See Learning Paths The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.See Java Language Changes for a summary of updated language features in Java SE 9 and subsequent

https://docs.oracle.com/javase/tutorial/index.html

Units 00 - 03: Introduction
- 1. Why CS2030?
- 2. Java Basics
- 3. Taming Complexity
- 4. Types

Units 04 - 16: Object-Oriented Programming (OOP)
- 1. Basic Concepts
- 2. Encapsulation and Abstraction
- 3. Inheritance
- 4. Polymorphism (through Method Overriding)
- 5. Further abstraction with Abstract Classes/Methods and Interfaces

Units 17 - 24: Types
- 1. More about Types (Type Casting & Variance of Types)
- 2. Exceptions
- 3. Generics

Units 25 - 33: Functional Programming
- 1. Immutability
- 2. Nested Classes
- 3. Achieving Side Effect Free Programming
- 4. Lazy Evaluation
- 5. **Infinite Lists/Streams**
- 6. Functors and Monads

Units 34 - 37: Parallel Programming
- 1. Parallel Streams

2. Asynchronous Programming
3. Thread Pools/Fork and Join
Summary

Units 00 - 03: Introduction

1. Why CS2030?

- Real-world code is complex, evolves continuously, and is the product of teamwork.
- Write 'better' code through programming paradigms, idioms, and language design. (e.g. Object-oriented Programming (OOP), and Functional Programming)
  - more human-friendly
  - easier to change and maintain
  - fewer crashes
- **Programming Paradigms**
  - **imperative** (how to deduce the solution, e.g. procedural, OOP)
  - vs **declarative** (assertion of truth e.g. logic, functional programming)
- **Not a module about:**
  - algorithms or efficiency, software engineering, Java language and libraries

2. Java Basics

- **What is Java?**
  - a verbose, compiled, cross-platform language
  - high-level (abstraction penalty?) and multi-threaded
  - statically typed, object-oriented and class-based inheritance general purpose programming language
- **Compiled vs Interpreted** Languages - **compiler is our friend**
  - Java Workflow: source code → compiler → compiled into java bytecode ahead of time→ just in time interpreted and run by the JVM
  - helps us to catch errors early, but can only detect errors based on the static code, not from running it (runtime errors)
  - prefer to catch errors in the compilation phase as compared to runtime phase
  - to run code more interactively, we can use jshell, which is a REPL (Read-Evaluate-Print-Loop) interpreter, runs off .java files
- **Java Memory Management**
  - **Stack** - for storing activation records of method calls, a method's local variables are stored here, is FILO
  - **Heap** - for storing Java objects upon invoking **new,** garbage collection done here

3. Taming Complexity

Complexity ***-*** *loosely defined as anything that increases the likelihood of bugs in a program.*

- Simple ways to tame complexity would be to:
  - comment code
  - use a coding convention
- **Use of Abstractions**

  > *"Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts." - The Abstraction Principle*

  - **Abstraction Barrier -** Distinction between **client** (which calls the function and doesn't need to know the implementation details, only how to interface with the functions through parameters and return values) & **implementer** (which is concerned with how the function will work)
    - a separation of concerns
    - modify computation behind abstraction barrier using inheritance, polymorphism, and closure
  - **The Variable** as an abstraction over data
  - **Functions:** Abstraction over Computation
    - **Compartmentalize computation** and its effects, reducing interactions (*pure functions, no side effects, also can use immutable objects)*
    - **hide implementation details** so changing the implementation doesn't affect the rest of the code, easier to communicate to other programmers how code works
    - **reduce repetition** and write more succinct code
  - **Further Abstraction** using Composite Data Types consisting of multiple primitive types **grouped together**
    - further extend by **creating methods** that operate on the composite data type, which will eventually shield us from having to know its implementation (same side of abstraction barrier)
  - Abstraction using **Classes -** bundled composite data types/fields & associated methods
    - only expose the methods you want others to use

4. Types

- **Type** of a variable decides the computation that can be performed on a variable
- **Type System:** A set of rules about types of variables, expression, functions, and their interactions.

- **Static** (Java is statically typed, i.e .the types of variables are fixed at runtime, and this helps the compiler check type errors through their **Compile-Time Type**)
  - vs **Dynamically typed languages (e.g. Python),** where the type of a variable depends on the values currently stored
- **Strong vs Weakly Typed** (exists on a spectrum, essentially deals with how strictly the type system is enforced in the language, ensuring type safety)
- **Primitive Types** (which **do not share their values)**
  - each primitive type not initialized will have a default value, e.g. default value for int is 0.
- vs **Reference Types** (anything that isn't a primitive type) e.g. objects, strings, arrays
  - only a reference/pointer to the object/instance is stored, rather than a copy of the data itself like in primitive types.
  - aliases can occur, where we have multiple references to the same object in the heap.
  - if a reference type is declared but not initialized, it will have a special null reference value.
- **Subtypes and Supertypes**
  - A type T is a subtype of S (T <: S) if:
    - a piece of code written for variables of type S can also safely be used on variables of type T.
  - This relation is reflexive and transitive
  - **Type Relations amongst Primitive Types**
    - byte (8-bit) <: short <: int <: long (64-bit) <: float (16-bit) <: double (32-bit)
    - char (16-bit unicode char) <: int
    - boolean
  - **Widening Type Casting** during variable assignment - Java allows putting a value of type T into a variable of type S iff T <: S. (e.g. we can put an int into a long) (automatic!)
  - **Narrowing Type Casting -** not automatic, only do it when 100% sure (e.g. checking using instanceof beforehand), otherwise leads to runtime errors.

Units 04 - 16: Object-Oriented Programming (OOP)

where a program runs by instantiating objects of different classes. Objects interact with each other by calling each others' methods.

1. Basic Concepts

- **Classes** (the blueprint), nouns
- **Objects** (specific instances)
- **Methods** (aka functions, govern behaviour), verbs
- **Fields** (or attributes/characteristics)

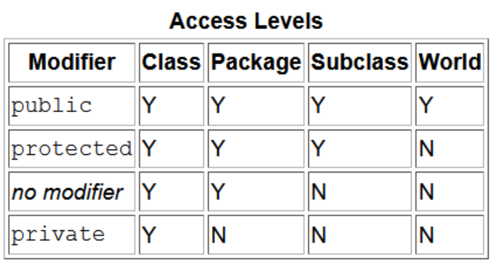
- **Constructors -** a method that initializes an object (e.g. internal fields), invoked automatically only when object instantiated.
    - Has same name as class and no return type (not even null)
    - otherwise, default constructor inherited from object will be used, initalizes fields to default values.
- **'this' keyword** - reference variable that refers back to self (the object/instance), used to distinguish between two variables of same name.
    - may not technically be 'necesssary' in non-ambiguous situations, but we enforce the use of this for clarity
- **Class Fields (contrast to Instance Fields)**
    - avoid hard-coding magic numbers, instead we can create global values associated only with a class rather than a specific instance (through the **static** keyword)
    - constants: 'public static final' - **final** means the value of the field cannot change after being initialized.
    - e.g. Math.PI (in java.lang.Math)
    - always accessed through class name for clarity
- **Class Methods**
    - similarly, class methods (defined with static) are associated with a class rather than a specific object.
    - class methods cannot access instance fields or instance methods (no keyword 'this')
    - preferably accessed through class (e.g. Circle.getNumOfCircles())
    - cannot be overridden due to static binding
- **Main Method**
    - All Java programs need to have a main method to start the program.
    - a type of class method, and must be defined like this: public static final void main(String[] args) { }
- **Overloaded Methods**
    - multiple methods with the same name in a class but with different arguments/order of arguments. you cannot, however, overload with different return types (compiler doesn't know which method to call)

2. Encapsulation and Abstraction

- **Encapsulation**: bundle related data (fields) and functions (methods) together to create a class

- Use **Information Hiding** to maintain the Abstraction Barrier (enforced by compiler) - **What Does the Client Need to Know?**

    - Utilize access modifiers to prevent access to internal details of the implementation meant to be hidden
    - For top-level class (only public, and default)
    - For members - all four can be used



Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

- **Private -** can only be accessed within the same class, only within the abstraction barrier. okay to access the private fields/methods of other instances of the same class.
- **Accessors and Mutators** are methods to either get or set the field values of an instance. **BUT,** breaks the encapsulation and exposes internal representation.
- **Solution:** "Tell, Don't Ask" principle should be adhered to - when we want a class to do something, implement a method to do it. (or tell the class to do it, rather than ask for the values to do it yourself)
- Ideally, don't define classes with accessors and modifiers to the private fields.
- **Further Abstractions using Composition -** one class containing another reference type as a field. Builds up layers of abstractions and construct sophisticated classes. improves abstraction and encapsulation.
    - Models a HAS-A relationship between two entities.
    - NOTE: aliasing can cause problems if not careful for reference types - one solution is to avoid sharing references as much as possible, but this results in suboptimal performance.
        - or, create **immutable** reference types

3. Inheritance

- models a IS-A relationship, uses 'extends' keyword
- Java only allows for single inheritance
- to avoid boilerplate forwarding methods from composition
- Used to create subtypes. Terms: superclass, subclass
- Only **public** fields and methods are inherited, **NOT private** fields and method.
- '**super**' keyword (e.g. super(params) to call the parent's constructor)
- inheritance has to **preserve the meaning of subtyping**, don't overuse it!
- Compile-time and run-time types - when S <: T, and we assign T name = new S. The **compile-time type** is T, but the **run-time type** is S.
    - Only methods/fields that can be called on a variable are those of its compile-time type.
- Every class in Java inherits from the class Object (java.lang.Object), which has some useful methods that are inherited:
    - **equals(Object obj)**, checks if two objects are equal to each other

        - by default, checks if they have the same reference
    - **toString()**, returns string representation of object as String object
        - invoked implicitly when concatenating Strings with other reference types
- **Liskov Substitution Principle (LSP)**
    - **Definition:** Let f(x) be a property provable about objects x of type T. Then f(y) should be true for objects y of type S such that S <: T.
    - We should not **violate the LSP** when creating new subclasses through inheritance. (i.e. changing agreed upon expected behaviour), subclass must be able to substitute parent class.
    - Less likely to create bugs/break our code when we adhere to the LSP.
- **Preventing inheritance (use keyword 'final'), usually to prevent critical classes/methods from being overriden**
    - On fields - makes them immutable after initial assignment
        - for instance final variables, must be initialized in the constructor
    - On methods - prevents overriding
    - On classes - prevents them from being subclassed
- Checking if something is an instance of/subtype of a class using **instanceof**

4. Polymorphism (through Method Overriding)

- alter behaviour of existing classes, without altering existing code
- **method signature** - method name, and number, type and order of its parameters (cannot have two methods with the same signature)
- **method descriptor** - method signature + return type
- **Overriding** occurs when subclass defines instance method with same method descriptor as instance method in parent class.
    - **always** use the **@Override** annotation for increased clarity, and to help the compiler catch mistakes. (does not affect the bytecode created)

Overriding Conditions

- Overriding method cannot have a more restrictive access level
- Static methods cannot be overridden
- Can only override what can be inherited (i.e. not private methods)
- Subtype of the return type is allowed in overriding method
- **Exceptions**
    - can throw new unchecked exceptions
    - cannot throw new/broader checked exceptions than the overridden method
    - cannot override constructors

How does the compiler decide which method to call?

- **Static Binding -** resolved at compile time, for **static, private, or final** methods where no overriding can possibly occur
- **Dynamic Binding -** which version of a method is involved is decided during run-time, depending on the **run-time type** of the object. That said, the specific method signature initially chosen depends on **compile-time type** (most specific one)
- Allows to write general and succinct code that is also automatically extensible.

5. Further abstraction with Abstract Classes/Methods and Interfaces

Abstract Classes and Methods

- a class that has been made into something so general that it **cannot and should not** be instantiated. (≥ 1 abstract method in the class)
- when we want to **further generalize our Classes,** so that we can make as-generic-as-possible methods that work for all these classes (e.g. getLargestArea works for all types of shapes)
- **abstract** keyword prevents a class from being instantiated on its own without being subclassed and defined.
- an abstract class can contain fields, and both **concrete** and **abstract** methods. **concrete class** cannot contain any **abstract** methods.
- abstract methods should not have a method body. (public abstract)

Interfaces

- models "can do" behaviour
- **implements** keyword instead of **extends**
- an interface is also a type, keyword **'interface'**
- when we want to make even more generic methods, and it doesn't make sense to subclass them (for classes across different class hierarchies)
- methods contained within are 'public abstract', should be contained in it's own class file and name should end with -able
- If a class C implements interface I, then C <: I. Implication - a type can have multiple supertypes
    - the concrete class has to override all abstract methods in the interface and provide an implementation to each

In Java:

- A class can extend at most one class
- A class can implement zero or more interfaces
- An interface can extend zero or more interfaces, but cannot extend from another class

Spectrum: (by extent of actual implementation)

- **Concrete Classes** (entirely implemented)

- **Abstract Classes** (≥1 abstract method and may contain fields)
- **(pure) Interfaces** (all abstract methods)

Note: for CS2030 we ignore the existence of impure interfaces, which contain default implementations due to compatibility reasons

Wrapper Classes - making code for reference types also usable on primitive types

- **A wrapper class** is a class that encapsulates a *type*, rather than fields and methods. Java has wrapper classes for all of its primitive types, making them behave like reference types.
    - e.g. int → Integer, double → Double
- Primitive wrapper class objects are **immutable.**
- **Auto-boxing and Unboxing -** Java has automatic implicit type conversion between primitive types and its wrapper class. (no need to do i.intValue() for e.g.)
- **Performance Concerns -** more overhead associated with using objects rather than primitives, although it might be hidden by the autoboxing and unboxing. Hence, methods in Java API e.g. array methods are overloaded - having one method each for each primitive type, and another for reference Object types.
    - situational

Units 17 - 24: Types

1. More about Types (Type Casting & Variance of Types)

Type Casting

- The need for narrowing type casting leads to fragile code - compiler cannot help ensure that run-time types are correct, only compile-time types. Leads to more responsibility on programmer to ensure correctness.

Variance of Types

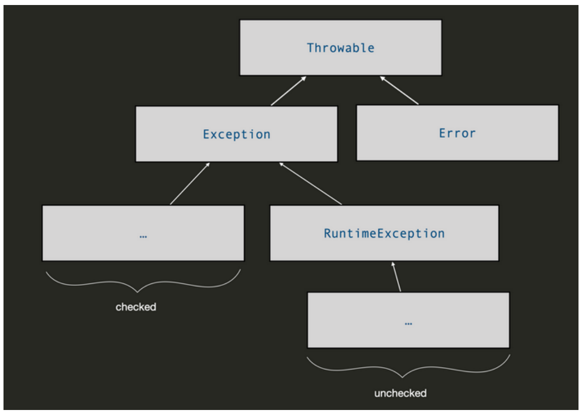
- Refers to how the subtype relationship between complex types relates to the subtype relationship between components.
- Let C(S) correspond to some complex type based on type S. (e.g. an array of type S)
    - it is **covariant** if S <: T implies C(S) <: C(T) (e.g. arrays, although this is an **unsafe** type system rule, could result in errors even without type casting)
    - it is **contravariant** if S<: T implies C(T) <: C(S)
    - **invariant** if it is neither covariant nor contravariant

2. Exceptions

- Java's exception construct - try, catch, finally and hierarchy of Exception classes.
- exceptions are instances that are a subtype of the Exception class, with information about an exception encapsulated within the instance and 'passed' to the catch block.

```
try {
  // do something
} catch (an exception parameter) {
  // handle exception
} finally {
  // clean up code
  // regardless of there is an exception or not or whether 'return' is called in try
}
```

- **Checked vs Unchecked Exceptions**
    - Unchecked exceptions are caused by programmer error (e.g. IllegalArgumentException). Usually not explicitly caught and thrown given how numerous they are. Causes runtime errors. (subclasses of the class RunTimeException) Left unchecked so the program fails and we can fix it.
    - Checked exceptions are those the programmer has no control over and can be recovered from during program execution. Should be **actively anticipated and handled.** e.g. (FileNotFoundException). We must always handle any checked exceptions, otherwise the program will not compile.
- **Error** class is for situations that are usually unrecoverable, so the program should terminate. (e.g. stack or heap is full)
- **Throwing Exceptions** - in method definition we add in 'throws AbcException', then have a line saying 'throw new AbcException(params);. This causes the method to immediately return
    - Note: All subclasses in Throwable that are not subtypes of Error and RunTimeException are checked exceptions and need to be specified in the throws clause of methods if not handled there.



- Exceptions can be passed up the call stack to the caller, and should be handled in the most suitable place. However, should not be passed all the way up to the user if it is a checked exception. (breaks abstraction barrier)
- finally block is always executed even when 'return' or 'throw' is called in a catch block.
- New exceptions can be created by inheriting from existing ones, but only when necessary e.g. to provide more information to exception handler. Remember to adhere to the LSP when inheriting!
- **Good Practices**
    - **Catch Exceptions to Clean Up,** exception can always be re-thrown upwards afterwards
    - Avoid Pokemon Exception Handling (catching everything and not doing anything) → fails silently and ends up ignoring all exceptions
    - Overreacting (immediately exit the program) → cannot free resources or clean-up

- Breaking Abstraction Barrier - we should handle implementation-specific exceptions within the abstraction barrier
- Do not use exceptions as a control flow mechanism → might not be correct + more inefficient

3. Generics

- When making general methods using Superclasses and Interfaces, we face problems with a **lack of compile-time type checking,** as we have to commonly do narrowing type casting (leading to run-time errors)
- **To reduce the risk of human error,** we use **generic types**, which ensure that type-checking takes place, while still having general classes and methods that are reusable. (basically polymorphism with increased type safety)
- **Generic types** take other types as **type parameters**
    - Pass in **type arguments** to make it a **parametrized type**
    - Type arguments can only be reference types
    - Generic types are **invariant**
- **Generic Methods**
    - type parameter is declared before the return type of the method in <>
    - Static methods will have no access to generic class' type parameters
    - Avoid using same name for class type parameter and static method type parameter (depends)
- **Bounded Type Parameters -** use keyword **'extends'** for subtypes, both superclasses and interfaces!
- **Type Erasure**
    - Java uses a **code sharing** approach for implementing generics. (type parameters and type arguments are erased during compilation after **type checking** is done)
        - alternative: code specialization (e.g. C++) where new class is generated for every new type argument
    - In other words, generics don't exist in the bytecode after compilation, code becomes more like polymorphism.
    - **Implications:**
        - Arrays and Generics cannot mix
        - Arrays are of a **reifiable type** - a type where full type information is available during run-time. Allows JRE to check for type mismatches.
        - Generics are **not reifiable** because of type erasure.
        - Consequence - **heap pollution** - variable of parametrized type refers to an object that is not of that parameterized type.
        - Hence, not allowed to create new arrays of generic type.

- **Solutions:**
    - Use data structures from Java Collections such as an ArrayList, or,
    - make our own Array<T>
- **Unchecked Warnings**
  - When compiler warns us that a piece of code e.g. a cast, may not be entirely type-safe (it cannot check because of type erasures)
  - Can be overridden using @SuppressWarning("unchecked")
    - This must be used at the **most limited** scope, only if we are **certain**, and we must **add a comment explaining** why this is safe
    - Only can be applied to **declaration**, not assignment.
- **Raw Types**
  - When a generic type is used **without type arguments**
  - Mainly for backwards compatibility reasons and is how the code looks after type erasure. But results in **no type checking taking place at all at compile time!**
  - Do not use unless really necessary - e.g. as part of instanceof, or making Comparable[] like in the lab
- **Wildcards - ?**
  - **Upper-bounded Wildcards** <? extends TypeName>
    - S <: T ⇒ A<? extends S> <: A<? extends T> (**covariant)**
    - For any type S, A<S> <: A<? extends S>
  - **Lower-bounded Wildcards** <? super TypeName>
    - S <: T ⇒ A<? super T> <: A<? super S> (**contravariant**)
    - For any type S, A<S> <: A<? super S>
  - **PECS - Producer Extends, Consumer Super** (w.r.t. variables of type T)
  - **Unbounded Wildcards** e.g. Array<?>
    - supertype of all generics e.g. Array<T>, Array<? super T>, could be any type
    - Restrictive because we don't know what the actual type is
- **Type Inference**
  - when Java attempts to infer the type argument of a generic method and generic type which lacks explicit type annotations. (which can also be called a *type witness*)
  - always selects the **most specific** possible type that passes the type checks (allows the code to compile) e.g. parameters for generic methods, type of the expression (**target typing**)
  - type inferencing can lead to **unexpected consequences,** so be careful!
  - **Syntax:**
    - **for generic types, use empty <>. e.g. new Pair<>();**

        - for generic methods, call like a non-generic method

Units 25 - 33: Functional Programming

Misc:

- **Varargs** - to pass in an array of items to a method, we can also use T... items, which can then be used like this - of(1, 2, 3, 4)
- **@SafeVarargs** to tell compiler it's safe to use a generic array in this case.

1. Immutability

- Another way to reduce complexity - by **avoiding change** altogether
- Immutable objects cannot have any **visible changes** outside its abstraction barrier (any of its methods must always **behave the same way**)
- Make fields **'final'**, make immutable classes **final** where possible to disallow inheritance and method overriding, and have methods **return new objects** rather than mutating the current object

Benefits:

- Avoid issues with aliasing and shared references, or the solution to that which involves creating numerous objects on the heap. Instead, immutable classes allow for shared references until modifications need to be made. (**copy-on-write semantic**)
    - Caveat: might not always be faster e.g. what if we keep modifying a class - lots of extra overhead as we need to constantly create new objects
- **Ease of Understanding** - objects do not change at all unless reassigned
- **Enables Safe Sharing of Objects** - e.g. making static factory methods for the client that reuses objects
- **Enables Safe Sharing of Internals** - e.g. ImmutableArray<T>, a wrapper around a regular array, can safely share subarrays of itself.
- **Enables Safe Concurrent Execution**

2. Nested Classes

Nested Class

a class defined within another containing class

- Group **logically relevant** classes together
- Usually not used outside container class, can also be made to act as a 'helper' class
- Considered a field of the containing class, and can access fields and methods of the container class (including private ones), even if two-layers of container classes! (and **vice-versa)**
- Declare as private if no need to expose to client.
- Should belong to **same encapsulation** as container class.

- **Static Nested Class** - associated with containing class, can only access its static fields and methods
- **Non-Static Nested Class (Inner Class) -** associated with a specific instance, can access all fields and methods
  - shouldn't have static methods and fields in this (makes no sense + compiler has some degree of enforcement - only allows constants)
- **qualified 'this'** - e.g. A.this.x to access 'this' of container class

Local Class

a class within a function (think: local variables)

- similar to nested classes, can access fields and methods of enclosing class AND can access local variables of enclosing method.
- e.g. creating implementation of Comparator interface within a sort method.
- **Variable Capture**
  - when a method returns, all **local variables of the method are removed from the stack**, but an instance of a local class might still exist
  - Hence, local class always **makes a copy of the local variables it accesses inside itself**. (**captures** the local variables)
  - **Effectively final -** To avoid confusion with variable capture, Java only allows local classes to access local variables that are **explicitly declared final or effectively final** (doesn't change after initialization)

Anonymous Class

where a class is declared and instantiated in a single statement - thus lacking a name

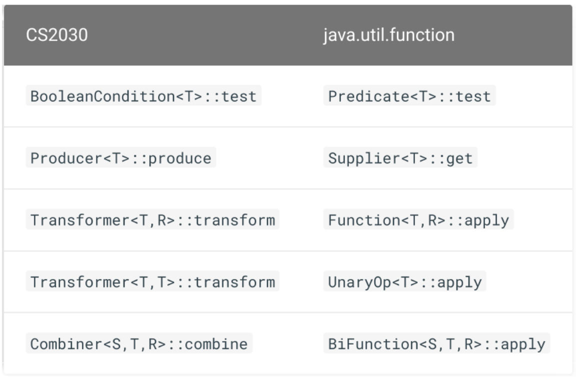
- **Format**: **new X (arguments) { body }**
  - **X** is a class that the anonymous class extends or an interface that is implemented, and cannot be empty.
    - Cannot extend + implement at the same time, or implement >1 interface.
  - **arguments** are those to be passed into the constructor (non-existent for those implementing an interface)
  - **no constructor allowed** for anonymous classes (has to inherit one)
- can be passed around like any other object
- Similar to local class in terms of variable **capture** and **access of the enclosing scope**

3. Achieving Side Effect Free Programming

- Mathematical functions (solely map input → output) have:
  - **No Side Effects** e.g. print, throw exception, write files etc.
  - **Referential Transparency** (outcome is deterministic for same arguments)
    - i.e. if f(x) = a we can always replace f(x) with a and vice-versa

- By writing code that meets these criteria, we can then reason about it using mathematical reasoning techniques.
- Functions that meet both criteria are known as **pure functions**
- Immutable classes will also be needed to meet criteria
- Can then build program solely from pure functions - known as **Functional Programming**
- Java is an OO language, so we can only do **functional-style programming,** where we try to adhere to these rules as much as possible (e.g. immutability, no side effects)

- Functions as **first-class citizen** - can be assigned to variable, pass as a parameter, return a function from another function etc. (just like an int)
  - Realized in Java as an instance of a (local) anonymous class extending a **Functional Interface**
- **Functional Interfaces** are interfaces with exactly one abstract method. (annotate with @FunctionalInterface)



CS2030	java.util.function
`BooleanCondition<T>::test`	`Predicate<T>::test`
`Producer<T>::produce`	`Supplier<T>::get`
`Transformer<T,R>::transform`	`Function<T,R>::apply`
`Transformer<T,T>::transform`	`UnaryOp<T>::apply`
`Combiner<S,T,R>::combine`	`BiFunction<S,T,R>::apply`

- **Lambda Expressions** are syntactic sugar for conveniently writing these local anonymous classes.
  - Syntax: (parameters) → {computation + return value};
  - Only need to provide type info, brackets, 'return' etc, when necessary
- To use existing methods **(method reference)**, use Class/Instance::methodName
  - For constructor, method name will be 'new'
  - Compilation error is thrown if there is ambiguity about which method to use
- **Currying -** translating a general n-ary function to a sequence of n unary functions
- **Curried Functions -** higher order functions (function that returns another function) used such that that we can generalize to multiple arguments
  - allows partial application of a function
- Similar to local and anonymous classes, lambda expressions will capture variables in the enclosing scope that are needed, and these must be **effectively final.**
- **Lambdas are Closures -** constructs that store a function together with the enclosing environment.
  - this is extremely useful - can delay execution, pass less parameters around, reduce duplication, separate logic more easily etc.
- Extremely general abstractions can be made using classes like Box<T>
  - allow **manipulation of internal data across abstraction barrier** while **maintaining certain rules** through methods that accept certain functions as parameters

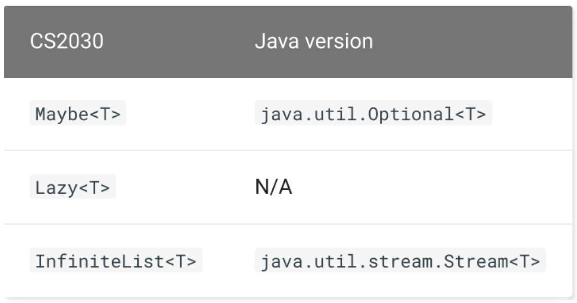
    - e.g. Map, Filter
- Maybe<T> helps us to automatically take care of the situation where output is null and maintain pureness of functions by having their codomain include null. (i.e. codomain will be a Maybe type)
- **Note: enum type,** same as assigning constant integers to variables but safer and less prone to errors. Also a class of its own, supports declaring fields and methods. (see Java documentation)

4. Lazy Evaluation

- **Procrastinate up to the last moment**
    - Lambda expressions are not invoked upon declaration - allows us to delay computation until needed
- **Never repeat yourself**
    - Cache (or **memoize)** the value that we compute after invoking a function so that we don't have to compute it again
    - Caveat: function must be **pure** for this to make sense
- Implemented in Lab 6 - Lazy<T>, can find implementations in external functional programming libraries

5. Infinite Lists/Streams

- **InfiniteList<T>** is a generic class that uses lazy evaluation to create a list of infinite length (see Lab 7)
- In-built streams are also **lazy**
- **Stream Pipeline** (Data Source (i.e. stream creation) → Intermediate Operations → Terminal Operation)



CS2030	Java version
`Maybe<T>`	`java.util.Optional<T>`
`Lazy<T>`	N/A
`InfiniteList<T>`	`java.util.stream.Stream<T>`

- **Stream Creation**
    - **Static factory methods**: Stream.generate(), iterate(), of(1, 2, 3)
    - Convert other objects into streams - Arrays::stream, List::stream
- **Intermediate Operations** - lazy operations that return another Stream
    - **Manipulating Internal Data** - map, filter, flatMap etc.
    - **Truncation** to finite stream - limit(int n), takeWhile(predicate), range(start, non-inclusive end)
    - peek(Consumer) - to get an idea of the Stream contents at some point
    - **Stateful** operations - need to keep track of some states to operate e.g. sorted, distinct (contrast to Stateless)

- **Bounded** operations - should only be called on finite streams (again, sorted and distinct are examples of this)
- **Terminal Operations** - these trigger the eager evaluation of the stream
  - forEach(Consumer<T> consumer) e.g. System.out::println
  - reduce(identity, accumulation function) - returns a single value
  - Element Matching against a predicate using noneMatch, allMatch, anyMatch
  - count()
- Note that streams can **only be operated on once** to avoid an IllegalStateException. (not true for InfiniteList!)
- Specialized streams exist for primitives to avoid the inefficient use of wrapper classes (e..g IntStream)
- Streams are useful in functional programming as they are more **declarative** than loops, making our code more **concise and less error-prone**
  - Caveat - not all loops can be replaced by streams elegantly. (e.g. double-nested, triple-nested)

6. Functors and Monads

- flatMap allows us to take in metadata alongside the value (useful in say, Loggable<T>)
- A recurring pattern:
  - Generic classes that encapsulate a value, along with some **side information** (of to initialize, and flatMap to update)
  - e.g. Lazy<T> (has the value been evaluated or not), Maybe<T> (is the value there or not), Loggable<T> (side info of a log describing operations done on the value)
  - These **abstractions** need to obey certain rules to behave as intended
- **Monads has two methods 'of' and 'flatMap' and obey three laws (I**f we obey these laws, we can safely write methods that receive a monad from others, operate on it in any order, and return it to others)
  - **Identity Laws** - 'of' and 'flatMap' methods **should not do anything extra** to the encapsulated value and side information
    - Note: Monad is the type, monad is an instance of it
    - **Left Identity Law**
      - Monad.of(x).flatMap(x → f(x)) must be the same as f(x)
    - **Right Identity Law**
      - monad.flatMap(x → Monad.of(x)) must be the same as monad.
  - **Associative Law -** no matter how we group the calls to flatMap, their behaviour **must be the same**
    - monad.flatMap(x → f(x)).flatMap(x → g(x)) must be the same as monad.flatMap(x → f(x).flatMap(y → g(y)))

        - Similar to Math: (x+y)+z = x+(y+z)
- **Functors have two methods 'of' and 'map' and obey two laws**
    - Simpler than monads, only ensures lambdas can be **applied sequentially** to the value (nothing to do with side information) i.e. **supports the map operation**
    - **Preserving Identity**: functor.map(x → x) is the same as functor
    - **Preserving Composition**: functor.map(x → f(x)).map(x → g(x)) is the same as functor.map(x → g(f(x))
- A class can be both a functor and a monad e.g. Lazy<T>, Maybe<T>

Units 34 - 37: Parallel Programming

Just an introduction to how this can be achieved, but barely scratches the surface of this big field.

- **Sequential execution** - only one instruction is carried out at a time
- **Concurrency -** when we constantly switch between different processes to make it appear like they are running at the same time. Can be achieved for a program by making it multi-threaded. (e.g. separating unrelated tasks like I/O and UI) Prevents a single task from stalling the entire program.
- **Parallelism -** when multiple subtasks are actually running at the same time. Achieved via computer with multiple cores/threads, or a processor that can execute multiple instructions at the same time
- All parallel programs are concurrent, but not all concurrent programs are parallel. (although nowadays nearly all computers have >1 core)

1. Parallel Streams

- Java Streams support parallel computation using .parallel(), a **lazy** operation. (counterpart is .sequential())
    - Only marks the stream to be processed in parallel, hence doesn't matter where you includeit
    - Order of output may be lost, can use forEachOrdered at the expense of some overhead
    - or use .parallelStream() from the outright
- **Embarassingly Parallel Computations** - When a task is stateless & without side effects, it's very easy to parallelize e.g. number of primes between x and y
- That said, **parallelizing a stream does not always improve performance.** Creating threads incurs overhead.

Conditions for a Task to be Parallelized

- **No Interference**: **Stream operations must not interfere with the stream data,** causes ConcurrentModificationException. Can't be done even in non-parallel streams.
- Most of the time must be **stateless**

    - **Stateful lambda** - result depends on any state that might change during the stream execution.
    - Otherwise, need to ensure that state updates are visible to all parallel subtasks
- **Side-effects kept to a minimum** to avoid problems e.g. when modifying non thread-safe data structures
    - note: can avoid using .collect() or thread-safe data structure
- **Associativity** - e.g. to run **reduce()** in parallel
    - combiner.apply(identity, i) must be equal to i
    - combiner and accumulator must be associative
    - combiner and accimulator must be compatible - i.e. combiner.apply(u, accumulator.apply(identity, t)) must be equal to accumulator.apply(u, t)
- **Ordered vs Unordered Stream (e.g. from List vs from Set)**
    - or iterate (ordered) vs generate (unordered)
    - **Stable** operations maintain encounter order e.g. sorted, distinct, limit, findFirst
    - Call unordered() to make parallel operations more efficient - subtasks no longer need to coordinate to maintain the order

2. Asynchronous Programming

- **Synchronous Programming -** when methods *block* until they return something. (entire program waits around for the method to complete before continuing execution)
    - not efficient - e.g. what if we are getting files from a slow web server?

To avoid this, we can make use of the following:

1. **Threads (java.lang.Thread)**
    - A **thread** is a single flow of execution in a program.
    - Initialize as an object, takes in a **Runnable** which takes in no parameter and returns void
    - Run the Thread using instance**.start()**, which is non-blocking, and can only be done once
    - instance.getName() gives the name of a thread
    - Thread.currentThread() gets the reference of the current running thread.
    - By default, we only have a single **main** thread.
    - Thread.sleep(time) pauses current execution thread for a given period.
    - instance.isAlive() checks if thread is still running
    - Program exits only after all threads created run to completion.
    - **Limitations**
        - not a high-level enough abstraction - a lot of details we must take care of to write a complex multi-threaded program e.g. dependencies, inter-thread communication, exceptions, overhead (because of not reusing Thread instances)

2. **Java's CompletableFuture monad (java.util.concurrent.CompletableFuture)**
    - Encapsulates a value that is either there or not there yet. (a promise)

    Initialization

    once created, immediately starts computation asynchronously

    - completedFuture(value) - creating a task that is already completed and returns a value
    - runAsync(runnable), returns a CompletableFuture<Void>
    - supplyAsync(supplier<T>), returns CompletableFuture<T>
    - allOf, anyOf - to have a CompletableFuture depend upon other CompletableFuture instances,takes in CompletableFuture<?>...

    Chaining Computations

    - thenApply (same as map)
    - thenCompose (same as flatMap)
    - thenCombine(another cf, biFunction) (same as combine)
    - These run in the same thread as the caller. If we want to make them run in a different thread - use the Async versions

    Getting the Result (synchronous/blocking calls)

    Call these only in the final step of the code. Blocks until they return a value.

    - **get() -** throws some checked exceptions that must be handled
    - **join()** - only throws unchecked exceptions

    Dealing with Exceptions

    CompletableFuture<T> stores any exceptions and passes them down the chain until join()/get() is called. If join(), throws a CompletionException that contains information on the original exception.

    - can always do try-except-finally, OR any of the following:
    - exceptionally
    - whenComplete
    - handle(BiFunction taking in value and exception)
        - useful for substituting another value if an exception occurs (i.e. e != null)

3. Thread Pools/Fork and Join

Thread Pools

- Creating and destroying threads is not cheap → achieve thread reuse using a thread pool
- Consists of:
    - A collection of threads, each waiting for a task to execute
    - A collection of tasks to be executed

- One type implemented by Java is the ForkJoinPool, meant for a parallel divide-and-conquer model of computation.
    - implement using abstract class RecursiveTask<T>, and methods compute(), fork() and join()
    - **long story short**: threads always look for something to do and they cooperate to get as much work done as possible
    - Order of forking should be reverse of the order of joining. **Reason** - most recently forked tasks more likely to be executed next (put to front of queue)

Summary

CS2030 teaches you to write "**better**" code, where **"better"** means:

- more human-friendly
- easier to change and maintain
- fewer crashes

through programming paradigms, idioms, and language design.

Examples: (Majority are applicable to other programming languages as well)

- **Reduce boiler-plate code** (via lambdas, inheritance etc.) → more succinct code, changes only need to happen in one place
- **Abstraction Principle** - each significant piece of functionality implemented in just one place
    - e.g. via variables, methods, generics, methods with lambda inputs, Optionals etc.
- **Writing Extensible Code**
    - Composition
    - Polymorphism
    - Abstract Classes & Interfaces
- **Minimizing the effects of changing an implementation**
    - Client-Implementer separation, maintaining abstraction barrier on implementation details
- **Liskov Substitution Principle** (just because a class inherits from another does not mean it is substitutable i.e. does not mean LSP was adhered to), don't change a desirable property of the parent class
    - Subset of **SOLID Principles** (more in CS2103)
- **Generics** - makes code re-usable while ensuring type safety (no need to always cast)
    - avoid raw types! (legacy feature)
    - **PECS** for maximum compatibility
    - variance, type erasure (and consequences - e.g. cannot mix with arrays, static variables, same method signatures)

- **The compiler is your friend!** Catch errors at compile time instead of run-time! (of course, compiler cannot catch everything due to static program analysis)
  - **ensures type safety (by type checking and type inference)**
  - Note: at run-time, we can make use of things like Unit Tests and Assert statements
- **Reducing Complexity**
  - **Isolate the moving parts**
    - Encapsulation
    - Information hiding
    - Tell, don't ask e.g. Map, Filter
  - **Minimize the moving parts**
    - Immutability - no aliasing problems, easier to reason
    - Pure Functions

Misc Paradigms and Topics:

- Exceptions
- Asynchronous Programming (CompletableFuture to make pipelines of asynchronous computation)
- Streams (concentrate on what needs to be computed, don't worry about handling loops)

Overarching Thoughts

- OOP (polymorphism: easy to extend) or FP (referential transparency: easy to reason)? Well, pick the best one for the situation. Doesn't have to be mutually exclusive either.
- Java is a nice OOP language, but a clunky FP language (and has lots of historical baggage)
  - But if you have no choice, there are libraries with immutable collections and other monadic constructs
  - Other language types - Parallel/Concurrent, Functional, OO Functional

What's Next

- Software Engineering Modules: CS2103, ...
- Programming Language Modules: CS2104, ...
- Parallel Computing Modules: CS3210, CS3211, ...

THE END
