Concrete Classes:

OOP Concepts:

1. Abstraction
    a. Client can just call the methods from the implementor, does not need to know how it is implemented, just need to know what to do with the result
2. Encapsulation
    a. Data hiding (private attributes and private methods)
3. Inheritance
    a. Note that Java only supports single inheritance
    b. Think of what we can abstract to the
4. Polymorphism
    a. A subclass of a parent class can be substitutable for its parent
    b. i.e. if $B <: A$ then we can let A a = new B()

Principles:

1. Tell Don't Ask
2. Liskov Substitutability Principle (LSP)

Types of Relationship:

1. HAS-A ()
2. IS-A (Inheritance) ()

Typing:

1. **Compile Time Type (Static Type):** Looks at the variable typing then see what are the methods that we can call using the overarching class. Looks at more of the overloaded methods to see what is the method signature that is required for the given arguments. This ensures that any object that is assigned to the object can call the method and therefore, there wouldn't be any issues
2. **Run Time Type (Dynamic Type):** Only when we are running and after the compile time type, we would have determined what is the method that we are going to call. Then we will look at the actual class that is assigned to that variable, it could be a subclass and therefore, there could be an overridden method and it will be called instead of the parent class.

Methods:

1. **Overloading**: Compile Time Type Thing
2. **Overriding**: Run Time Type Thing

Java Memory Model:

- Call Stack: Where the variables and functional calls are being stored and it is a stack so that when we call we can get the result of the inner parts and then return to the method that is calling it
- Heap Space: Basically the place where we store a new object. Once new is called, it will be there
- Meta Space: Basically where all the static variables are stored and can be shared across the same class

Interfaces:

- Similar to parent concrete classes, it defines what needs to be implemented
- Type of Arrow ()
- Let the client talk to the interface (Coding against the interface)
- For instance: List
- We can have multiple interface implementation per class (As compared to the single inheritance)

SOLID Principles:

1. Single Responsibility Principle
    a. Every Object is only responsibility for 1 thing that it needs to do
2. Open-closed principle
    a. Classes should be open for extension but closed for modification
    b. We should always be able to add subclasses afterwards and the previous reliance on the interface will not break. But the interface will not be modifiable because it will break some of the relationship that it can have with the client.
3. Liskov Substitution Principle
    a. Covers issues about typing and behaviour as well. (B substitute A then it needs to behave like an A as well)
    b. If S is a subtype of T (denoted $S <: T$), then an object of type T can be replaced by that of type S without changing the desirable property of the program.
4. Interface Segregation Principle
    a. For instance, if a class implements 2 interfaces, the first interface will only know that it has the methods that it requires, it does not know that it has implemented another interface. Vice versa for the other interface
5. Dependency Inversion Principle
    a. Program to an interface, not an implementation
    b. Always program to the interface because it ensures that the contract is met. Because the client programs to the interface and the implementor also depends on the interface. So that there will not be any issues

Single Abstract Method (SAM) Interface:

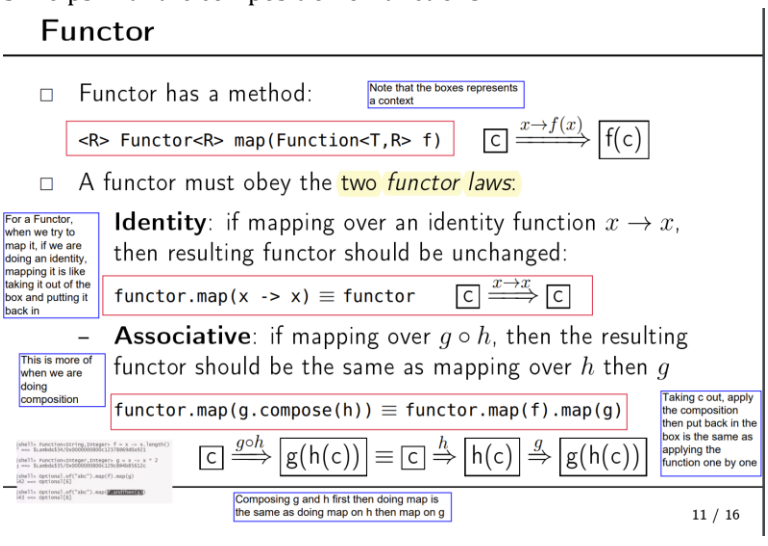
- Can be written as a lambda function
- Note that it is an actual implementation of the interface. It is an actual concrete class implementation, we can write it as an anonymous inner class
- Note the Method that it is writing, what it needs to take in and what it goes out
- Examples
    - Function
    - Predicate
    - Supplier (Helps to delay evaluation)
    - Consumer
    - Comparator
- Can be written as:
    - Concrete Class
    - Anonymous Inner Class
    - Lambdas (Preferred way of writing)

Abstract Classes:

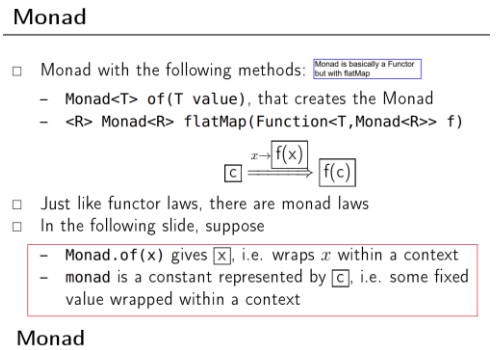
- In between Concrete Classes and Interfaces
- When we have some attributes and methods that we want to push to the parent but we want to have some abstract methods that are not defined then we use abstract classes.
- Write anonymous inner classes to prevent cyclic dependencies between the child class and parent class

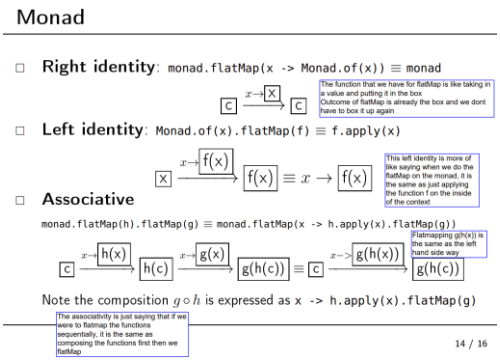
Context:

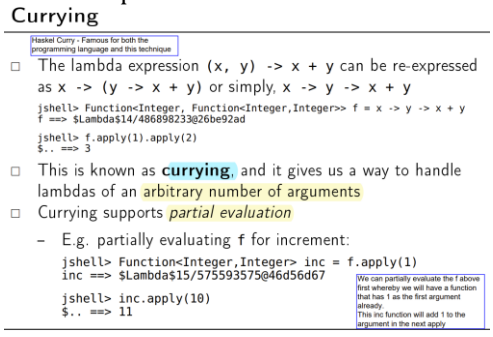
- Normally a generic type (At least a <T>)
- Allows the client to call the Context by passing in a **lambda** (any implementation of a SAM) via a certain method (filter, map, flatMap). By passing in lambda, we will just help to tell the implementor how to manipulate the data
- Communication between the Client and the Implementor. Cross Barrier Manipulation
    - Client tells the client how to do by passing in the SAM. How to map, how to filter etc.
    - Passing in functionalities into the context
- Normally, the return value will always be a Context. The pipeline will just be within a whole context until we do a terminating condition to get it out. (Kind of a reduction)
- Pipeline
    - Data Source
    - Intermediate Operations
    - Terminal Operations
- For Example:
    - Optional (Maybe)
    - Lazy
    - Stream
    - CompletableFuture
- **Type of Transformations**:
    - **map**: Takes out the value, put a function application and then put it back into the box
        - Note that the function needs to take in T and then return without the context
        - **Functors**: Helps with the composition of functions



        - **flatMap**: Takes out the value, put a function application and then the result is actually already within the context.
            - Note that the function needs to take in T and return the context instead for this
            - Note that we can just think of flatMap and map as ways to go inside the box and operate then we box it back up.
            - **Monads**: Helps with the composition of functions

- 

- 

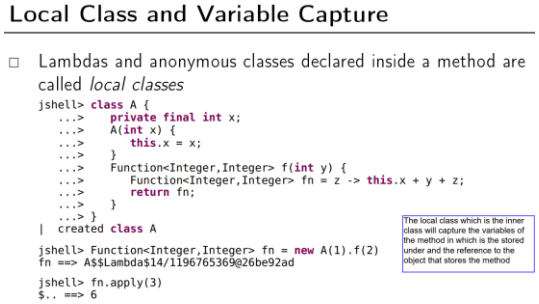
- Things to take note of:
  - **Pure Functions**
    - Takes in arguments and returns a deterministic value
    - Has no other side effects
  - **Function Composition**
    - compose
    - andThen
  - **Currying**
    - Gives us a way to handle lambdas of an arbitrary number of arguments
    - There are partial evaluations of a function
    - 

Generic Typing:

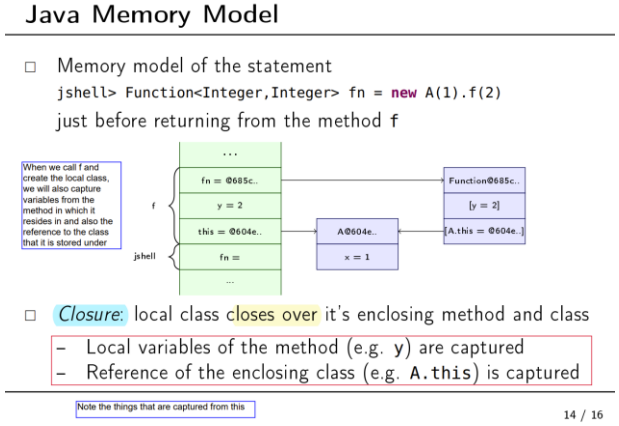
- Example: Box<T>(of T t)
- Bounded and Unbounded Wildcards
  - For example (map(Function<? super T, ? extends R>))
    - For the function, we want it to be able to take in anything that is T which is our object or it can just take in any part inside of T which is part of its super class. We just want to make sure that the function can operator on our T
    - For the function, we want to be able to give out anything that extends R because for the output of the map, we know that we can get an R object, we can have a subclass and within it there will still be an R and therefore, we can have ? extends R as the return type.

Local Classes/ Variable Capture/ Lambda Closures:

- Local class makes a copy of variables of the enclosing method and reference to the enclosing class.
    - Capture the variables that are in the method
    - Captures the reference of the object that is enclosing the method



- **Closure**:
    - Local class closes over its enclosing method and class



- Note that Java only allows a local class to capture variables that are explicitly declared final or effectively (implicitly) final
    - An effectively final variable is one where value does not change after initialization. Note that we should not be reassigning the value that we have received under the local method so that the capture does not change.
    - This is also why we cannot modify states out of the context. For instance if we have a stream, we cannot change the variables that are outside of it
- This normally needs to be captured when we try to write out an implementation of a SAM (which can be a lambda or anonymous inner class). Therefore, there will be a need for the captures

CompletableFuture

Methods:

supplyAsync: Factory method to wrap the value into a CompletableFuture

- Needs a Supplier that gives the return value of the object itself. i.e. CompletableFuture.<A>supplyAsync(() -> a.incr().decr()); We just need to have the A inside the return of the Supplier
- Spawns a new thread to compute this

thenRun: Once we are done, just runs the action

thenAccept: After we are done with the process, we will let the consumer consume the final value

runAsync: Gives a Completable Future<Void> after completing an action

anyOf: It runs whenever any of the given CFs complete and will be assigned the value of the cf that is completed first

runAfterBoth: Returns a new CompletionStage that, when this and the other given stage both complete normally, executes the given action.

runAfterEither: Returns a new CompletionStage that, when either this or the other given stage complete normally, executes the given action.

whenComplete: Returns a new CompletionStage with the same result or exception as this stage, that executes the given action when this stage completes.

exceptionaly: Returns a new CompletionStage that, when this stage completes exceptionally, is executed with this stage's exception as the argument to the supplied function.

thenApply: This is like a map() method. Where we can take out the value of the previous CompletableFuture and then apply the function onto the value and then rewrap it back

- Needs a function that helps to map the value inside to another value and rewraps it back into a CompletableFuture
- The computation will be done on the same thread since it needs to wait for the previous value to be computed first before we can apply the function on the result. The same thread will be used for both computation since it can be the same thread
- Because of the nature of the chain, therefore it is the same thread

thenApplyAsync: Similar to the *thenApply* method

- Needs a function that helps to map the value inside to another value and rewraps it back into a CompletableFuture
- However, as compared to the *thenApply*, it could be another thread that is used for the second computation. It could also be the same thread, since the thenApply needs the previous value first, therefore, we could use the same thread.
- Will be useful when we want to use the previous value for 2 separate computations, we can use this so that we can split up the computations to be Async and 2 things can be done concurrently with the previous value.

thenCompose: Similar to the idea of a flatMap

- Needs a function that helps to map the value inside to another value and then returns a CompletableFuture
- Useful when we want to chain up operations that returns us a CompletableFuture

thenCombine: Combines the value of the result of the computations that comes before it

completedFuture: Useful when we already have the value already.

- Just needs the value of the completed value
- When we already have the value and does not need to compute anything
- This allows us to not require to spawn another thread. Since it is already completed, we don't even need to compute it also.

allOf: Helps to call all of the CompletableFuture together. Enumerating all of the computations

- Takes in any number of CompletableFuture of any return Types
- Returns a CompletableFuture<Void> since the return Type could have any type, therefore, we need this void to ensure that for any that does not have any return value to be accounted for also
- When we do a .join(), it will just return null. There is no return value since it is a void

handle: Helps with exception handling

- Takes in a bifunction that takes in a T result and also a Throwable.
- Note that if it is an exception, then the result will be null.
- It can helps us to check if the value from our CompletableFuture computation has an error or not
- Let handle() handle the exception

Join: It is like our normal reduce but with a blocking wait

Pointers from AY2021/2022 Sem 2 Finals

- Make use of how to use the various functions and SAMs and the other methods they have
- Function -> andThen thenCompose
- Predicate -> and, or
- map and flatMap what we are taking in and if the context has like 2 parameters, take note of what are the ones that we want to take in
- When we want to lazily evaluate, make use of supplier and note that if we want to take it out again just use the .get() method
- Program to the interface (SOLID Principles) if we want to segregate stuff
- To remove cyclic dependencies, can look into using anonymous inner classes, definition go and look at it but basically it is to create new Class<T>() { definition of new class }
- Alternating signs, can make use of pow and look at how to make it positive or negative, but ensure power is not negative or 0
- Take note of Stream stuff
- Always draw out the things and ensure that we do not miss out on anything, make sure we have checked at least 2 layers of the test case before proceeding