CS2030S
AY23/24S2 Final Examination
By: github.com/zeepheru

"Java"

Final
- Class - cannot be inherited.
- Method - cannot be overwritten.
- Field - cannot be reassigned.

Static
- Fields - class fields.
- Methods: Accessed through class, cannot access non-static fields.

Abstract
- Abstract class cannot be instantiated.
- Only **one** method needs to be abstract.
- ≥ 1 method is abstract $\implies$ class HAS to be abstract!

Interface
- All methods in an interface are `public abstract` by default.
- `I i2 = (I) new A()` would always compile - there is a possibility that during **run-time**, `a` is an instance of class that **extends** `A` that implements `I`.
- If `I` was a generic interface - `I<T>; I<String>`, then a warning - cannot check if `A` is an instance of `I<String>` - **type-erased** to `I`.
- If `A` was `final` - then the run-time type of `a` is always `A` - won't implement the interface! - doesn't compile.

NOTE: `private` fields/methods are **not** inherited by children classes.

Primitives
- `byte <: short <: int <: long <: float <: double`
- `char <: int`

Signatures, Overriding/loading

Method Signature:
- **YES**: name, number of args, type of args, order of args
- **NO**: name of args, return type, exceptions
- Note: generics make a signature different.
- **Method Descriptor**: method signature & return type.

An **overridding** method:
- NOT REQUIRED to throw the same exceptions.
- CANNOT be private if parent method is public.
- CAN return a subtype.

You cannot overload with the same arg types **after type erasure**.

```
// Same signature after type erasure!
void foo(A<String> a) {...}
void foo(A<Integer> a) {...}
```

OOP Principles

Encapsulation
- Composite data types
- Abstraction barrier
- private fields/attributes, public methods

Inheritance - "is-a"
Composition - "has-a"

Polymorphism
- Method overriding - change how existing code behaves w/o changing the code itself

Information Hiding
- *"Abstraction barrier", "publicly accessible"*

Tell Don't Ask
- basically, never ask for the internals for a computation or operation
- Eg: an `Interval` and a `Time` → Don't ask for the secs and ms from `Time` to compute differences, get `Time` **to do it itself**.

Liskov Substitution Principle
- The current **DESIRED PROPERTY** is important. Often the property that is being modified in a superclass being changed in a subclass.
- Let $\varphi(x)$ be a property provable about objects x of type T. Then $\varphi(y)$ should be true for objects y of type S where $S <: T$.
- A subclass should not break the expectations set by the superclass. If a class B is substitutable for a parent class A, then it should be able to pass all test cases of the parent class A. If it does not, it is not substitutable → LSP is violated.
- Any code written for A would still work if we substitute A with B.

Misc-1

Dynamic Binding
1. At compile-time, find matching object descriptor in CTT.
2. At run-time, find a method with the same method descriptor in RTT.

Varargs
- `void foo(T... args)` - `args` will be `T[]`.
- Use `@SafeVarargs` for when `T` is generic.

Variances
- **Covariant**: $S <: T \implies C(S) <: C(T)$
  - E.g. `arrays`
- **Contravariant**: $S <: T \implies C(T) <: C(S)$
- **Invariant**: Neither.
  - E.g.`generic classes`

Type Inference
The **most specific** type is taken.

Generics, Wildcards
Wildcards can be in return types.
Generics are generally invariant.

Upper-bounded Wildcard
- `Seq<? extends Shape>`.
- For any type `S`, `A<S>` $<:$ `A<? extends S>`
- Covariance: if `S` $<:$ `T`, then `A<? extends S>` $<:$ `A<? extends T>`.
- So in practice, an argument `Seq<? extends T>` would match `Seq<T>` and `Seq<S>`, where `S` $<:$ `T`.

Lower-bounded Wildcard
- `Seq<? super Shape>`
- For any type `S`, `A<S>` $<:$ `A<? super S>`
- Contravariance: if `S` $<:$ `T`, then `A<? super T>` $<:$ `A<? super S>`.

PECS
Producer `extends`; Consumer `super`.
Not needed if question didn't ask for it.

Type Erasure

```
Integer i = A<String>.foo();
// becomes
Integer i = (Integer) A.foo();
```

Hence why you can't do a `instanceof A<String>`.

Generics and Arrays
arrays are reifiable - full type information should be available at runtime - now type erasure complicates matters - the generic is gone.
`new A<?>[10]` works because it is reifiable!

Exceptions

Potential Issues
1. Using exceptions as *control flow*.
2. Catching **all** possible exceptions - not advisable.
3. Anticipated exceptions should be a **checked exception**.

Nesting
- **Inner** class: non-static class in `class`.
  - Consider a B in an A.
  - `A.B b = a. new B()` is valid initialization code, where `A a = new A();`
  - `this` in `B` would refer to `B`.
  - To access the fields of `A` use the *qualified reference* `A.this`.
  - Not using a *qualified reference* is fine.
  - The **same name** can be used inside and outside an inner class - the closest innermost one will be used.
- **STATIC nested** class - Operates similarly to a static method. Can only access static fields and methods in outer class.
- **Local** class: class in `method`.
  - Variable capture:
    - Captures the fields from outer method **that it needs**.
    - Field has to be **final** or **implicitly final** - no reassignment after first initialisation.
    - Always captures the outer class of the method: `B.this`

Anonymous Class
Extending class:

```
new Book("name") {
    @Override
    public String foo() {
        ...
    }
}
```

Implementing interface:

```
new Runnable() {
    @Override
    public void run() {
        ...
    }
}
```

Monads
`map` - $X \to Y$ || `flatMap` - $X \to Monad(Y)$

Left Identity Law
- $id \circ f = f$
- `Monad.of(x).flatMap(y -> G(y)) = G(Y)`

Right Identity Law
- $f \circ id = f$
- `m.flatMap(x -> Monad.of(x)) = m`

Associative Law
- `m.flatMap(x -> F(x)).flatMap(y -> G(y))`
- `= m.flatMap(x -> F(x).flatMap(y -> G(y)))`

Preventing violating of laws
- The `flatMap` method should **NOT add anything on its own** - would violate associative law
  - *eg add some string to the log*
  - ***It should just combine/operate on previous values/side info***
- The `of` method should **NOT add anything either**
  - *eg its own startup message*

Functors

Lambdas can be applied sequentially to a value; does not carry side info.

- Supports a `map` and `of`.
- **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))`.

Rec. Note: Monads are functors. You can write `map` using `flatMap`!

Lambdas

`ff -> x -> ff.transform(x);` is correct syntax.
In fact, this, $f : X \to Y \to Z$, returning a sequence of n unary functions, is called **"currying"**.

Funtional Interface

- `@FunctionalInterface`
- Exactly 1 **abstract** method. (can contain other stuff)

Immutability

- Fields `final`; always `return` **new instance** of obj.
  - **Ease of understanding** - guarantees obj being passed around is the same.
  - **Safe sharing of objects** - eg an origin is always the same.
  - **Safe sharing of internals**
  - **Safe concurrent exec**

Parallel/Async

- **Concurrency** - divides computation into subtasks called threads.
- **Parallelism** - multiple subtasks/threads running at the same time.
- parallelism $\subseteq$ concurrency

Threads

New `Thread().start()`'s are async.

```
// A and B in any order
new Thread(() -> printRepeat("A")).start();
new Thread(() -> printRepeat("B")).start();
// Also
System.out.println(Thread.currentThread().getName());
```

CompletableFuture

Chaining `cF.thenApplyAsync().thenApplyAsync()...` will be done **in order**! Also note the necessity to `join()` at the end. Otherwise not all will be printed.

ForkJoin

Always preferable to `fork` then `join` in a different order. Highest level of parallelism.

```
right.fork(); left.fork();
return mid.compute() + right.join() + left.join();
```

ForkJoinPool

- Each thread has a deque of tasks.
- When a thread is idle, it checks its deque of tasks. If the deque is not empty, it picks up a task at the head of the deque to execute *(e.g., invoke its `compute()` method)*.
- Otherwise, if the deque is empty, it picks up a task from the *tail* of the deque of another thread to run. The latter is a mechanism called **work stealing**.
- When `fork()` is called, the caller adds itself to the *head* of the deque of the executing thread. This is done so that the most recently forked task gets executed next, similar to how normal recursive calls.
- When `join()` is called, several cases might happen. If the subtask to be joined hasn't been executed, its `compute()` method is called and the subtask is executed.
  - If the subtask to be joined has been completed (some other thread has stolen this and completed it), then the result is read, and `join()` returns.
  - If the subtask to be joined has been stolen and is being executed by another thread, then the current thread either finds some other tasks to work on from its local deque, or steals another task from another deque.

Stack and Heap Examples