CS2030S

AY25/26 Sem 1

Variables and Types

- **Statically typed**: variable can only hold values of the same type (assigned at compile time)
- **Strongly typed**: enforce strict rules in type system → catch type errors during compile time
- **Subtype**: a piece of code written for variables of supertype can be safely used on variables of subtype
  - **Reflexive**: For any type S, $S <: S$
  - **Transitive**: If $S <: T$ and $T <: U$, then $S <: U$
  - **Anti-symmetric**: If $S <: T$ and $T <: S$, then $S = T$
- **Widening type casting**: if $S <: T$, variable of type T can automatically hold value from variable of type S
- **Narrowing type casting**: if $S <: T$, explicit typecasting from T to S (else code won't compile)
  If runtime type of target is not the same as the cast type, then runtime error will occur

Encapsulation

- **Encapsulation**: keep data and functions related to a composite data type together
- **Access modifiers**:

Accessed from	`private`	`public`
Inside the class	✓	✓
Outside the class	×	✓

- **Information hiding**: protect abstraction barrier from being broken by explicitly specifying if a field/method can be accessed from outside abstraction barrier
  - Private fields: prevent arbitrary changes
  - Public methods
- **"Tell, Don't Ask" principle**:
  - **Don't ask** an object for its state (using getters/setters), then perform the task on its behalf
  - **Tell** an object what to do – a task that is performed only on the fields of a class should be implemented in the class itself

Composition

Composition: HAS-A relationship

Stack and Heap

- **Stack**: one method invoked ⇒ one **frame** created
  - Last-In First-Out
  - When method completes, frame is removed
- **Heap**: one `new` keyword ⇒ one new **object** created
- **Metaspace**: static fields of classes

Inheritance

Inheritance: IS-A relationship (`extends`)
Subclass **inherits** all accessible fields/methods

- Can access all public fields/methods of superclass
- Cannot access any private fields/methods of superclass

Overloading

- **Method signature** = method name + number, type, order of parameters `C::foo(B1, B2)`
- **Method descriptor** = method signature + return type `A C::foo(B1, B2)`
- **Method overloading**: multiple methods in the same class have **same name**, **different method signature**
- If two methods have same name & method signature → not overloaded → cannot compile

Polymorphism

- **Method overriding**: subclass defines instance method with **same method descriptor** as superclass
- `@Override` annotation: hint to compiler that a method is intended to override another method in superclass
- **Dynamic binding**: decide *instance* method invoked e.g. `curr.foo(arg)`

  Compile time step:

  1. Determine CTT(`curr`)
  2. In CTT(`curr`), find all methods with name `foo`
  3. Check `arg` can bind to which method
  4. Choose the most specific one (use subtyping)
     Record its **method descriptor** M

  Run time step:

  5. Determine RTT(`curr`)
  6. In RTT(`curr`), find M *exactly*
  7. Found it? Execute M
     No find? Repeat search in the superclass
- **LSP**: if $S <: T$, then an object of type T can be replaced by that of type S without changing the **desirable property** of the program
- `final`: used to prevent
  - field from being re-assigned
  - method from being overidden (**prevent overriding**)
  - classes from being inherited (**prevent inheritance**)
- **Abstract class**: cannot be instantiated
  - A class with at least one abstract method must be declared abstract
  - An abstract class may have no abstract method
- **Interface**: all methods `public abstract`, no fields
- **Inheritance rules**:
  - A class can **extend** at most one class, **implement** multiple interfaces
  - An interface cannot extend from another class, can **extend** multiple interfaces
- Casting using an interface: *possible* that a subclass `E` *could* extends `D` and implements `A`

Types

- **Wrapper class**: encapsulates primitive type
  Primitive wrapper class objects are immutable
- **Auto-boxing**: primitive to wrapper
  Unboxing: wrapper to primitive
- No subtyping relationship b/w wrapper classes
- Due to immutability, during every mutating operation, a new wrapper object is created
- Narrowing type conversion e.g. `a = (C) b`

  Compile-time check

  1. Check if it is *possible* for `RTT(b)` <: `C`
  2. Check if `C` <: `CTT(a)`

  Runtime check

  3. Check if `RTT(b)` <: `C`
- **Covariant**: if $S <: T$ then $C(S) <: C(T)$
  Contravariant: if $S <: T$ then $C(T) <: C(S)$
  Invariant: neither covariant nor contravariant
- Java arrays of reference types are **covariant**

Exceptions

- Handle exceptions: `try-catch-finally` block
  1. Check in the order they appear
  2. Check subtyping relationship: choose the first `catch` block that thrown exception can bind to

  Superclass of exception appear before subclass → second catch will never be executed → prevented with compilation error
- Throw exception:
  1. Declare method that throws exception with `throws`
  2. Create a new `IllegalArgumentException` object and throw it to the caller with `throw` keyword
- **Checked exception**: must throw and catch
- **Unchecked exception**: not caught

Generics

- Generics are **invariant**: no subtyping relationship
- **Generic method**: parametrise a method with type parameters, without being in a generic class
- **Bounded type parameters**: use `extends`
  If `T` <: `GetAreable`, then `T` must have `getArea()`
- **Type erasure**: type information of generics missing during runtime
  1. Remove **angle brackets**, replace type parameters with their **bounds**
  2. If generic type is instantiated and used, insert **type casts** to preserve type safety
  3. Generate **bridge methods** to preserve polymorphism in extended generic types
- Generic array declaration is ok, but instantiation is not
  - Arrays are **reifiable**: full type information available during runtime (checks if items in array match array type → `ArrayStoreException` if mismatch)
  - Generics are **not reifiable**: due to type erasure, type information missing during runtime
  - **Heap pollution**: a variable of parameterised type refers to an object not of that parameterised type → retrieving leads to `ClassCastException`
- **Unchecked warning**: message from the compiler that there could be runtime error that it cannot prevent, due to type erasure
  - `Seq` is generic with type parameter `T`, no upper bound
  - After type erasure, `T` is replaced with `Object`
  - Compiler can't verify at runtime that `Object[]` will only contain `T` objects, since erasure removes the type info needed for this check
  - Hence the cast from `Object[]` to `T[]` is unchecked
- `@SuppressWarning("unchecked")` annotation: suppress warning messages from compiler & assure compiler that the type operation is deemed to be safe
  - Since `array` is private, only way to put sth is `set`
  - `set` only accepts `T`, so objects in `array` of type `T`
  - Casting `Object[]` to `T[]` is type-safe
- **Raw type**: generic type used without type arguments → use **type erased version**
- **Unbounded**: `C<T>` <: `C<?>`
  Upper-bounded: `C<T>` <: `C<? extends T>`, `C<? extends S>` <: `C<? extends T>`
- **Lower-bounded**: `C<T>` <: `C<? super T>`
  `C<? super T>` <: `C<? super S>`

- **Producer Extends; Consumer Super** (PECS)
- **Diamond operator**: infer type as **declared** type (CTT)
- **Local type inference algorithm**:
  1. Write down all local type constraints
     - Target typing
     - Argument typing
     - Type parameter bound
  2. Solve type constraints
  3. Choose most specific one (mentioned or superclass)

Immutability

Immutable class: instance cannot have any visible changes outside its abstraction barrier

- Make fields `final` to avoid re-assignment

  Remove any assignments to the fields (compilation error due to `final`)
- Make class `final` to disallow inheritance
- Change setter from `void` to return a new instance

Nested Class

- **Static nested class**: access **static** fields, methods
- **Inner class**: access **all** fields, methods
- Qualified `this` reference: `<container>.this.<variable>`
- **Local class**: declared inside a method

  Access variables of enclosing class through qualified `this` reference & local variables of enclosing method
- **Variable capture**: captured **local variables** must be **declared / effectively final**
- **Anonymous class**: `new X(arg) { body }`

Lambda Expressions

- **Pure function**:
  1. **No side effects**: print to screen, write to files, throw exceptions, modify fields, modify arguments
  2. **Referentially transparent** (**deterministic**)
- Stack and heap: anonymous function represented as anonymous class - type name is compile-time type name & the lambda expression

  E.g. `Producer () -> a.get()`
- **Method reference**: if a lambda expression does nothing but call an existing method, it is clearer to refer to the existing method by name
  1. Static method: `ContainingClass::staticMethodName`
  2. Instance method of a particular object: `containingObject::instanceMethodName`
  3. Instance method of an arbitrary object of a particular type: `ContainingType::methodName` (first argument is an instance of `ContainingType`)
  4. Constructor of a class: `ClassName::new`
- **Higher-order function**: takes in a single argument, returns another function
- Lambda is a **closure**: captured variables must be **declared / effectively final**

Lazy Evaluation

- **Lazy evaluation**: lambda expressions are only evaluated when parameters are supplied → delay execution of the lambda body until it is needed
- **Memoisation**: cache the value of a function after evaluating → method body only runs once

Infinite List

- Distinguish b/w **invocation** (`this.tail()`) & **field access** (`this.tail`) - see argument type of method, check that the types make sense

  `Lazy<T> head, Lazy<InfList<T>> tail`

  `T head(), InfList<T> tail()`
- Use **recursion** to write new methods

```
public static <T> InfList<T> weave(InfList<T
    > l1, InfList<T> l2) {
  return new InfList<>(
    l1.head, Lazy.of(() -> InfList.weave(l2,
        l1.tail()))  );  }
```

Streams

- `of`, `generate`, `iterate`
- `filter`, `map`, `flatMap`, `sorted`, `distinct`, `limit`, `takeWhile`
- `forEach`, `reduce`, `count`, `allMatch`, `anyMatch`, `noneMatch`
- A stream can only be **evaluated once** (terminal operation invoked) → iterating through a stream multiple times throws run-time error `IllegalStateException`

Monads and Functors

Functor: value, no context

- **Identity**: `fct.map(x -> x)` ≡ `fct`
- **Composition**: `fct.map(x -> f(x)).map(x -> g(x))` ≡ `fct.map(x -> g(f(x)))`

Monad: value and context

- **Left identity**: `Monad.of(x).flatMap(x -> f(x))` ≡ `f(x)`
- **Right identity**: `mn.flatMap(x -> Monad.of(x))` ≡ `mn`
- **Associative**: `mn.flatMap(x -> f(x)).flatMap(y -> g(y))` ≡ `mn.flatMap(x -> f(x).flatMap(y -> g(y)))`

Write down LHS and RHS, see if equal

Parallel Streams

- **Parallel stream**: `parallel()`
  1. Breaks down stream into subsequences
  2. Runs operations for each subsequence in parallel
  3. Whichever parallel tasks that complete first will output the result to screen first
- **Parallel reduce**: identity e, accumulator f, combiner g
  1. **Identity**: $f(e,x) = x$
  2. **Associative**: $f(x, f(y,z)) = f(f(x,y),z)$ $g(x, g(y,z)) = g(g(x,y),z)$
  3. **Compatible**: $g(x, f(e,y)) = f(x,y)$

  (If `reduce` accepts two arguments, then accumulator is the combiner.)
- Operations that attempt to **preserve encounter order** can get expensive → need to coordinate between the streams to maintain the order

Asynchronous Programming

- **Thread**: a single flow of execution in a program

  `Thread(Runnable task)`: constructor

  `start()`: execute the given lambda expression

  (`Runnable` is a functional interface with a method `run()` that takes in no parameter, returns `void`)
- Different **interleaving** of executions → no control over which thread executed first, when it switches to another
- A `CompletableFuture` **completes** when the operation (lambda expression) that you pass in has finished execution and returns a value

  Non-blocking operation: does not wait for `this` to complete → do other tasks in the meantime

  Blocking operation: waits for `this` to complete
- `runAsync`: create a new `CompletableFuture`
- `allOf`: completes when all the input complete
- `anyOf`: completes when any one of the input completes
- `thenApply`: analogous to `Stream::map`
- `thenCompose`: analogous to `Stream::flatMap`

  `op` is logically executed after `this` completes

  Operation is non-blocking
- `thenCombine`: analogous to `Stream::combine`

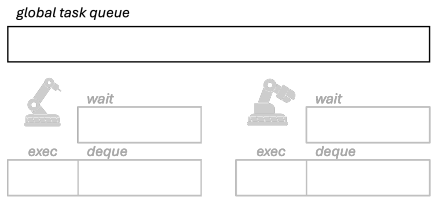
  `op` is logically executed after `this` and `other` complete

  Operation is non-blocking
- `thenRun`: executes `action` after `this` is completed
- `runAfterBoth`: executes `action` after `this` and `other` are completed
- `runAfterEither`: executes `action` after `this` or `other` is completed
- `join()`: get the result

  Operation is **blocking**
- **Dependency diagram** b/w tasks: figure out what gets printed by looking at all possible orders of execution

Fork and Join

- **Thread pool**: (i) a collection of threads, each waiting for a task to execute, and (ii) a collection of tasks (in a **shared queue**) - idle thread picks up a task from the shared queue to execute
- `RecursiveTask<T>`
  - `fork()`: divide a problem (submits a smaller version of the task for execution)
  - `join()`: retrieve the result (waits for the smaller tasks to complete and return)
  - `compute()`: define the task's logic - if the task is small enough, solve directly; else split into subtasks, fork them to run in parallel, then join their results
- `ForkJoinPool`:



Main thread:

- `fork()`: adds the forked task to the back of the **global task queue**
- `join()`: puts the current task in the waiting area, waits for the result from the joined task to be available

Worker thread:

- Idle thread: checks its deque of tasks
  - If not empty → pick up a task at the head of its deque → execute it (using `compute()`)
  - If empty → pick up a task from the tail of deque of another thread (**work stealing**) → execute it
- `fork()`: adds the forked task to the head of **local task deque**
- `join()`: puts the current task in the waiting area
  - If the joined task is in the deque, execute the joined task by calling `compute()`
  - Otherwise, wait for the result from the joined task to be available - may execute other tasks in the meantime

- **Palindrome**: `join()` in reverse order of `fork()`