CS2030S Cheatsheet AY23/24 —— @Jin Hang

Program and Compiler

1. Java → bytecode →(By JVM)→ machine code
   - `$ javac Hello.java`: `javac` is the Java **compiler**
   - `$ java Hello`: Invoke the JVM `java` and execute the **bytecode** contains in `Hello.class`
     JVM language(bytecode) → x86-64 machine language
2. **Interpreter** (`jshell`): Interprets `Hello.java` interpreted from Java **directly** to the x86-64 machine language.

Compiler: 只知道CTT,不知道RTT!! 不执行情况下检查

1. Translate source code into machine code or bytecode
2. Parse source code and check grammar → error if violated
3. Detect any syntax error **before** the program is run.
4. Violations to **access modifiers** are checked by the compiler. Trying to access, update, or invoke fields or methods with `private` modifier will give a **compilation error**.

Cannot tell

1. if a particular statement in the source code will be executed
2. what values a variable will take

(3) Behavior

- **Conservative:** report an error as long as there is a possibility that a particular statement is incorrect
- **Permissive:** If there is a possibility that a particular statement is correct, it does not throw an error, but rely on the programmer to do the right thing.

Heap and Stack: JVM manages the memory of Java programs while its **bytecode** instructions are interpreted and executed.
Stack for local variables and call frames. **Note that** instance and class fields are **not** variables → fields are **not** in the stack.

1. Variables are contained within the call frames (created when we invoke a method and removed when the method finished).
2. Use $\emptyset$ to indicate the variable is not initialized. When instance method is called, JVM creates a stack frame for it, containing
   - the `this` reference
   - the method arguments
   - local variables within the method, among other things
3. When a **class method** is called, the stack frame **does not** contain the `this` reference.
4. After method returns, the stack frame for it is **destroyed**.

Heap for storing dynamically allocated objects
Whenever you use the keyword **new**, a new object is created in the heap. An object in the heap contains the following information:

- Class name.
- **Captured variables**
- Instance fields and the respective values

1. Memory allocated in heap stays as long as ∃ a reference to it
2. You do not have to free the memory allocated to objects.
3. The JVM runs a garbage collector that checks for unreferenced objects on the heap and cleans up the memory automatically.

Heap and Stack Diagram

1. Can omit memory addresses stored in var. and of the object.
2. The intermediate call frames (e.g., Point constructor) can be omitted. Only the final effect matters.

Type Inference 选最specific的,并检查是否会造成Casting Error

- **Argument Typing:** Type of argument is passed to parameter.
- **Target Typing:** Return type is passed to variable.
- **Type Para.:** Declared type, especially for bounded type para..

Bound Constrain: No solution → Compilation Error

- Type1 <: T <: Type2, then T is inferred as Type1
- Type1 <: T2, then T is inferred as Type1
- T <: Type2, then T is inferred as Type2

Keywords

1. `null`: Any reference variable that is **not initialized** will have the special reference value `null`
   - In Java, uninitialized variables ≠ variables initialized to `null`
   - Uninitialized variables cannot be used
   - Uninitialized fields have default values(i.e. 0, 0.0, null...) but uninitialized var. not → compilation error
2. `this` reference variable refers to self, used to distinguish between two variables of the same name.
   - `x = x` VS `this.x = x` in constructor. First ⇔ assigning value of para. `x` to itself ⇒ ×update field ⇒ `this.x`←default value
   - **Automatically added** if no ambiguity referring to field.
   - Use `this(...)` at the **first line** to chaining constructor. It will invoke our original constructor.
   - **Cannot** have both call to `super(..)` and call to `this(..)`
3. `final` keyword can be used in three places:
   - In a class declaration to prevent inheritance.
   - In a method declaration to prevent overriding.
   - An **optional** modifier for the `main` method.
   - In a field declaration to prevent re-assignment.
4. `static`: `static` field/method ∈ class rather than instance ⇒ Can be accessed/updated or invoked without instantiating.
   - Java prevents using `this` in static method, otherwise → error
   - Non-static methods(including constructor) has `this`
5. `extends`
   - A class can only extend from one superclass, but it can **implement**(cannot extends) multiple interfaces.
   - An **interface** can **extend**(cannot implement) from one or more other interfaces, but cannot extend from another class.
6. `super` call the constructor of the superclass
   - has to appear as the first line in the constructor
   - If no call to `super`, default `super()` **automatically** added
   - No `super` – 1 error, while Non-first `super` – 2 error
   - Call overridden method in parent class using `super`
   - **Cannot** have both call to `super(..)` and call to `this(..)`
7. `instanceof` Assume `obj instanceof Circle`, it returns true if `obj` has a **run-time type** that is a **subtype** of `Circle`.

OOP Principle

1. **Information Hiding:** Expose 尽可能少 fields/methods
   - Isolate the internal of a class using an abstraction barrier.
   - Constructors / The `this` Keyword
2. **"Tell, Don't Ask" principle:** The client should tell an object what to do, instead of asking to get the value of a field, then perform the computation on the object's behalf.
3. **Liskov Substitution Principle (LSP)** If class B is substitutable for parent class A → should pass all test cases of parent class A. Otherwise, not substitutable and violate LSP.
   - Any inheritance with overriding should not introduce bugs
   - Let $f(x)$ be a property provable about obj. x of type T. Then $f(y)$ should be true for obj. y of type S where $S <: T$.
   - LSP cannot be enforced by the compiler

LSP Template

- Yes. `X` changesthe behavior of foo(), so the property that `/property/` no longer holds for **subclass** `X`. Places in a program where `/SuperClass/` is used cannot be simply replaced by `X`.
- No. Any code written for `/SuperClass/` would still work if we substitute `/SuperClass/` with `X`

Definition

class field: static fields that are associated with a class

- A static class field needs not be final and it needs not be public.

class method: Static Method can be accessed through the class

- always invoked without being attached to an instance
- cannot access Non-Static fields/methods
- the reference `this` has no meaning within a class method.
- Possible to overload, Impossible to override!

Constructor: No return type and access modifiers can be omit, `this` **cannot** omit! When called by `new`:

- Allocate sufficient memory location to store all the fields of the class and assign this reference to the keyword this.
- Invoke the constructor and pass keyword `this` **implicitly**.
- When done, return the **reference** pointed to by this back.

Default Constructor: No constructor given ⇒ default constr.. is added **auto.**, takes no parameter and has no code in body
Fully Qualified Name: i.e. `Circle.this.r`.

- starts with a sequence of class names separated by a dot.
- If name refers to a field, FQN is then followed by `this`. Otherwise, there is no keyword `this`.
- Finally, the FQN is followed by the actual name used.

Composition HAS-A; **Inheritance** IS-A
Method Signature: i.e. `C::foo(B1, B2)`

- Method name
- Number of parameters
- (Optionally) class name
- Types of parameters
- Order of parameters

Method Descriptor: Signature + Return type. [`A C::foo(B)`]
始终牢记Override和Overload都和变量名没关系!!!
Override: Same **method descriptor** and accessv modifier
Consider S<:T, A2<:A1, E1<:E2, `A1 T::foo(B) throws E2` can be **overriden** in class S as `A2 S::foo(B) throws E1`

- Parameter must be same type, cannot use subtype to override

Overloading: Change type, order, and number of para. but keep method **name** identical. *`contains(double x, double y)` and `contains(double y, double x)` are not distinct ⇒ ×overloaded

- Overload the class constructor using `this(...)`

Abstract Class: General! cannot and should not be instantiated.

- has at least one abstract method ⇒ abstract class.
- An abstract class may have **no** abstract method.
- The subtype of abstract class inherits the abstract methods unless the method is **overridden**.
- may inherit from concrete class to prevent instantiation of class.
- ×final: final ⇔ ×inherited ⇒ cannot be used ⇒ ×compile.
- ×private: private method ⇔ ×accessible ⇒ ×overridden ⇒ × implementation ⇒ ×compile
- ×static: static ⇒ ×override ..(same with private)

Interface: The abstraction models what an entity can do.

- Can have var. or static ver. Cannot have fields!
- All methods in interface are **public abstract** by default.
- A class implementing an interface must be an abstract class. Or has to **override** all abstract methods from the interface.

Wrapper Class: Primitive wrapper class obj. are **immutable**, ×change once created ⇒ less efficient than primitive types.
Autoboxing: `Integer i = 4;`

- **Single-step process:** `Double d = 2;`(2 steps) will not compile

(Auto-)Unboxing: not restrict in only one step: `double d = i;`
Variance: Producer convariant / Consumer contravariant may lead to **Run Time Error**! Please check the run time mismatch!

Type Checking

1. Compiler only using CTT for its Type checking!

```
class T { foo() {...} }
class S1 extends T { bar() {...} }
class S2 extends T { baz() {...} }
T x = new S1();
x.bar(); // Error, Type T has no method called bar
```

2. **Anti-symmetry:** Prevents **cyclic subtyping** relationship.

```
class A extends B { } class B extends A { } // Error!
```

3. **Nominal:** subtyping relationship has to be explicitly declared
Consider `a = (C) b;`

Compile Time Check: Find CTT(b)

1. 有无可能 RTT(b) <: C. 不可能→ compilation error
   - CTT(b) <: C: simply widening
   - C <: CTT(b): narrowing and requires run-time checks.
   - **C is an interface:** Let RTT(b)=B, it may have a subclass A s.t. A <: C, i.e. `class A extends B implements C`. RTT(b)=A ⇒ allowed at run-time.
2. Find CTT(a), 有无可能 C<:RTT(a), 不可能→compilation error
3. **Runtime Check** for RTT(b) <: C

Common Subtype Relation

1. **Type Casting:** Happen in Compile Time, check in Run Time
2. **Interface:** If a class C implements interface I, $C <: I$.
   `I i2 = (I) new A();` compiles, even A does not implement I
3. **Warp Class:** Integer <: Number → Integer[] <: Number[]
4. **Exception:** Exception a <: Exception b → catch(Exception a) cannot catch the Exception b

Method Invocation: If we want to invoke `obj.foo(arg)`
Compile Time: **CTT向上找, 放在一起后选最specific的**

1. Determine CTT(`obj`) and CTT(`arg`)
2. Determine all methods with name `foo` that are accessible in CTT(`obj`), including the parent of CTT(`obj`) and so on.
   - An abstract method is considered accessible although there is no method implementation.
   - RTT(`obj`) must be concrete → implements abstract method
3. Determine all methods from Step 2 accept CTT(`arg`).
   - Correct number of parameters.
   - Correct parameter types (i.e. **supertype of CTT(`arg`)**).
4. Determine the **most specific** method. If No most specific method, fail with compilation error.
   - Assume A<:B → `foo(A)` is more specific than `foo(B)`
   - Given S1 <: T and S2 <: T, `foo(S1)` is not more specific than `foo(S2)` and `foo(S2)` is not more specific than `foo(S1)`.

Run Time: **从RTT向上找第一个满足条件的!**
Determine `RTT(obj)` → Starting from `RTT(obj)`, find the first method that match the method descriptor.

Exception

1. **Unchecked Exceptions:** (<: RuntimeException)
   - Perfect Code should not have, i.e. `ClassCastException`
   - ×Explicitly caught/thrown ⇔ ×need `throws`/`try-catch`..
   - something wrong with program and cause **run-time errors**
2. **Checked Exceptions:** No control over, even perfect code!
   - A checked exception must be handled, or ×compiles
   - Must use `throws`, otherwise → Compile Eror
3. **Error Execution:** try → catch → finally(ALWAYS Executed)
   - Error find → all subsequent lines in `try` is **not** executed
   - look one-by-one from top to bottom for the first catch block
   - Consider ExceptionX <: ExceptionY, then the following code

```
catch(ExceptionY e){} -> catch(ExceptionX e) {}
```

⇒ compilation error since we will never catch `ExceptionX`

4. **Pokemon Exception Handling:** use `catch(Exception e)` **above** bolcks hands subclass of Exception → **Compile Error!**

Generics Consider `class Pair<S,T>`
- **type parameters:** `S,T`, **generic type:** `Pair<S,T>`
- Type arguments must be **reference types** (i.e. `Integer`)
- **Parameterized type:** Generic type instantiated.
- `A<T> extends B<String,T>`, `T` is **passed** from `A` to `B` class

Generic Methods
- `<T>` is added **before** return type and only scoped in its method
- Call generic method: `A.<String>contains(strArray, "s")`
- For `class D <T extends A & B>{}`, T will be erased to A(LHS) and then **casted** to B (type cast to implements B)

Type Erasure: Erase type parameters and type arguments during **compilation**⇒代表所有实例化的Generic⇒不用recompile
- If the type parameter is **bounded**, it is replaced by the **bounds**

Generics and Array
- **Heap pollution:** A var. of a para. type refers to an obj. that is not of that para. type. (`ArrayStoreException`)
- Array is **reifiable type**, where full type info. available in **RT.**
- Java generics is **not** reifiable due to **type erasure**.
- Generic array can be declared but not **instantiate**
- Array is **covariant** || Generics are **invariant** → no subtyping relationship → preventing the possibility of heap pollution

Unchecked Warnings: Use `@SuppressWarning`, annotation that suppresses warning messages from **compilers**.
- Use annotation **to most limited scope** to avoid suppressing warnings that are valid concerns from the compiler.
- Suppress a warning only if sure it won't cause type error later.

Raw type: A generic type used **without type arguments**
- Without a type argument → the compiler can't do type checking → uncertainty (`ClassCastException`)
- Mixing raw types with parameterized types ⇒ errors.
- **Only** use it as an operand of the `instanceof` operator.

Wildcards: `A.<Object,Object>foo(circles, c)` **won't compile**

Unbounded Wildcards: `Array<?>` :> `Array<T>`
- `Array<?>` is an array of objects of some specific unknown type;
- `Array<Object>` is an array of Obj. instances, with type checking
- `Array` is an array of **Obj. instances**, **without** type checking.

Reifiable type: A type where no type information is lost during compilation. `Comparible<?>` is **reifiable**.

Nested Class Use nested class only if it ∈ 相同封装 Otherwise, container class would leak implementation details to nested class

Access: Static: Only Static; **Non-Static:** Both!

Static Class: No captured variables, only access static var.

Local Class:
- Though accessible, local class makes a copy of local var. inside Only captures local var. Fields can be accessed, don't capture
- Use dashed line to separate the fields and captured var.
- Captured var. are **NOT** part of fields, cannot accessed with dot operator(`this.y`)
- `B.this` is **captured** by convention
- var. (even in `g()`) are captured when nested class is instantiated.

Anonymous Class: `new X (arguments) { body }`
- X is a class that the anonymous **class extends** or an **interface** that the anonymous class **implements**.
- 不能同时 extend 和 implement, 不能 implement > 1 interface
- Argument: argus. pass to constructor. X=interface ⇒ **no constructor in body**, but still **need ()**
- like local class, capture var. as well.

Lambda Expression

Main-effect return value without modifying any of the input.

Side-effect: (No∼: computes → returns)
- Printing to the screen
- I/O write operation.
- Modify the value of a field
- Mutating input arguments.
- Invoking side-effect func.
- Throwing exceptions.

Referential transparency: Let $f(x) = a \Leftrightarrow f(x)$ and a can substitute each other everywhere they appears.
- Absence of side-effect ⇏ referential transparency [return time]
- Side-effect ⇒ Not referential transparency [`a.get(0)` and 5]

Pure functions: side-effect free and referentially transparent.
- **Deterministic:** same input → same output (**ensures** ref. trans.)
- Immutable class ⇒ methods have NO side effects ⇒ pure

Functional Interface: An **interface** with **exactly one** abstract method (either declared in the interface or inherited)

Lambda Expression
- only one abstract method to overwrite, we don't have to write `@Override public Integer transform(..) { .. }`.
- type of parameter is **redundant** as the type argument already

Method Reference: Use `::` to refer to
1. **static method** in a class, i.e. `A::f // x -> A.f(x)`
2. **constructor** of a class, i.e. `A::new // x -> new A(x)`
3. **instance method** of class or interface, i.e. `a::g // x -> a.g(x)`
- When compiling, **type inferences** to find the method.
- **Multiple matches** or Ambiguity in matches ⇒ Compile **Error**
- `A::h // (x, y) -> x.h(y) or (x, y) -> A.h(x, y)` same expression can be interpreted in two different ways (depends on para. h takes and h is class/instance method)
- Use **reference**, when the obj. is modified, func(). `->` old one. ⇒ not effectively final; Use **Lambda** ⇒ Effectively Final

Lexical `this`: Using lambda don't think its presence in anony. class
- Code compiles in Lambda **may not** compile in Anonymous Class
- lambda as a **syntactic sugar** for anonymous class

Curried Functions: from **right-to-left**. Take `x` first, then take `y`.
- `Trans<U, Trans<U, U>> add = x -> y -> (x + y);`
- `x -> y -> (x + y)` ⇔ `x -> (y -> (x + y))`
- `add.trans(1).trans(1)` ⇒ `incr.trans(1)` `Trans<U,U> incr = add.trans(1);`
- `incr.trans(1)` ⇒ `(y -> y + 1)(1)` ⇒ 2

Closure A lambda expression stores **function** to invoke and data from the environment it defined. ⇒ Stores a function together with the enclosing environment is a **closure**.

Manipulators pass in lambda expressions behind the abstraction barrier and modify the internals arbitrarily

Box and Maybe: `Maybe<T>` is an `Optional` type.

Lazy Evaluation: If we have computed a value before, memoize it, and **won't compute it again**.

```
Logger.lazyLog(Logger.LogLevel.INFO, () -> "User " + System.
      getProperty("user.name") + " has logged in");
```

Infinite List and Stream
1. Java implementation of Stream is an infinite list (Lazy)
2. **Terminal operation:** triggers evaluation of stream, i.e. `forEach`
3. **Bounded operations:** (`distinct` and `sorted`) should only be called on a finite stream.
4. Unlike Infinite List, stream can only be **operated once**. Otherwise, throw `IllegalStateException` (i.e. `s.count()`)
- `limit(int n)`: returns a stream containing the first n elements
- `takeWhile(pred.)`: returns a stream containing elem. of stream, until predicate becomes false.(Do not include first false in stream!)
- `peek(consumer)`: 提取数据流经"管道"某一点时的值⇒不同点提取到的值是不同的，因此反映了变量x在不同状态下的值
- `reduce: Stream.of(1, 2, 3).reduce(0, (x, y) -> x + y);`

Monad: 1. well-behaved 2. value + side-information.
Sth. be a monad, first be a Functor, satisfy properties for `map`
- **Identity Laws:** `flatMap` do nothing more to value and side info
  1. **Left**∼: `Monad.of(x).flatMap(x -> f(x))` ⇔ `f(x)`
  2. **Right**∼: `monad.flatMap(x -> Monad.of(x))` ⇔ `monad`

  If both left and right ∼ elem. exists, it must be same value.
- **Associative Law**: Diff. group that calls `flatMap`, same result `monad.flatMap(x -> f(x)).flatMap(x -> g(x))` ⇔ `monad.flatMap(x -> f(x).flatMap(y -> g(y)))`

Functor: updates the value but changes nothing to the side info.
1. **Preserving identity:** `functor.map(x -> x)` ⇔ `functor`
2. **Preserving composition:** `functor.map(x -> f(x)).map(x -> g(x))` ⇔ `functor.map(x -> g(f(x))`

`map` don't change context into identity context created using `of()`

Parallel: A single-core processor only execute one instruction at one time ⇔ one process can run at any one time

Concurrent: divide computation into subtasks called **threads**
1. separate unrelated tasks into threads, and write separately
2. ↑ utilization of the processor

Parallelism
1. **Prerequisite:** Multiple cores/processors → dispatch instru.
2. All parallel programs are concurrent, but not all concurrent programs are parallel.

Parallel Stream: Divde into subseq. and run parallel
- `parallel`: lazy operation, merely marks stream to be processed in parallel, insert **before** the terminal operation.
- Same set of elements with different order → Fix this using `forEachOrdered` but lose some benefit of parallel
- **Stateless** and don't produce any side effects
- Parallelizing a stream **doesn't** always improve the performance Overhead of creating too many thread > Benefits
- If original order not important ⇒ `unordered()` ↑ efficient.
- `findFirst`, `limit`, `skip`: expensive on ordered stream (needs to coordinate between the streams to maintain the order)

Embarrassingly parallel: Each element is processed individually without depending on other elements.

Stateful Lambda: Result depends on any **state** that might change during execution of the stream.
- `generate` and `map` operations below are stateful!
- Parallelizing stateful lambda may lead to incorrect output.

Associativity: `reduce(id., accumulator, combiner)`
1. `accumulator` with identity apply to every elem.
2. `combiner` apply for all elem. like accumulate in 1101S
- The combiner must be **associative**(order of applying don't matter)
- accumulator **not necessary** to be associative!
- The combiner and the accumulator must be **compatible** `combiner.apply(u, accumulator.apply(identity, t))` ⇔ `accumulator.apply(u, t)`

Synchronous Programming: Only after method returns can the execution of our program continue.

Thread: [`new Thread(..)`] a single flow of execution in a program

```
new Thread(() -> { : } /*Runnable*/).start();
```

- `start()` returns immediately.
- `Thread.currentThread().getName()` cur. running thread name
- `Thread::sleep` Pause execute current thread for a given period.
- Creation of Thread takes up some resources → Reuse A.P.!
- Thread **cannot** handle exception!

CompletableFuture Monad: whether value it promises is ready
- `completedFuture`: Create an already completed task and return.
- `runAsync(Runnable)` Complete when lambda expression finish
- `supplyAsync(Supplier<T>)` same with (2)
- `allOf(Com..)`: Complete when all `Com..` completes
- `anyOf(Com..)`: Complete when any `Com..` completes
- `thenApply` ∼ `map`
- `thenCompose`∼`flatMap`
- `thenCombine`∼`combine`
- `thenRun(Runnable)` Execute Runnable after cur. stage complete
- `runAfterBoth(Com.,Run.)` Execute `Runnable` after current stage & `Com..` complete
- `runAfterEither(Com.,Run.)` Execute `Runnable` after current stage / `Com..` complete

`get()` **Method**:
- **Synchronous**, blocks until the `CompletableFuture` completes
- maximize concurrency → call at the final step in our code
- Throws a couple of **checked exceptions**:
  1. `InterruptedException`: Thread has been interrupted
  2. `ExecutionException`: Errors/exceptions during execution

Handling Exceptions CompletableFuture can handle exceptions.
1. Store exception and passing it down in chain, until `join()` is called → `join()` throw Exception → who calls `join()` handle this exception.
2. `handle(BiTransformer<T, U, R>)` to chain despite exception

```
Comp..<Integer> ith = Comp...supplyAsync(() -> f(i));
Comp..<Integer> jth = Comp...supplyAsync(() -> f(j));
Comp..<Integer> r = ith.thenCombine(jth, (x, y) -> x - y).join()
```

Thread Pool: Contains
- a collection of threads, each waiting for a task to execute
- a collection of tasks to be executed

Fork and Join:
1. **Fork**: Divide problem into smaller identical problems
2. **Join**: Solve them recursively, then combine the results

Work Flow: Big Problem → two smaller problem `left` and `right` → `left.fork()` add `left` to the pool s.t. one thread can call `compute()` on left → `right.compute()` → `left.join()` to sum up all the result.

Fork Join Pool
1. When thread is idle and its deque is
   - **not empty**: picks up task at head of deque to execute
   - **empty**: picks up task from tail of deque of another thread
2. When `fork()` is called, adds caller to the head of deque of the executing thread
3. When `join()` is called and subtask to be joined is
   - **Not executed**: call `compute()` and the subtask is executed.
   - **completed**: read result, and `join()` returns.
   - **stolen and being executed**: current thread finds some other tasks to work on either in its local deque or steal another task from another deque
4. **Stealing** is always done from the back
5. Order tasks are added is from the head of the deque.
6. Tasks at back is expected to have more unfinished computation compared to tasks at front of the deque. ⇒ minimizes number of task stealing needed

Order of Fork and Join
1. Most recently forked task is likely to be executed next, `join()` most recent `fork()` task first.
2. `fork()`, `compute()`, `join()` order 应形成回文 and **no crossing**.
3. At **most** a single `compute` in the middle of the palindrome.
4. When a thread is operating on its deque, the thread has to finish its operation before another thread can operate on

-*-*-*-*-*-*- PLEASE DELETE THIS PAGE! -*-*-*-*-*-*-

Information
Course: CS2030/S
Type: Final Cheat Sheet
Date: May 7, 2024
Author: QIU JINHANG
Link: https://github.com/jhqiu21/Notes

-*-*-*-*-*-*- PLEASE DELETE THIS PAGE! -*-*-*-*-*-*-