CS2030S Cheatsheet AY23/24 —— @Jin Hang

Program and Compiler

Java programs can be excute in 2 ways:

1. Java *arrow* bytecode $\xrightarrow{\text{JVM}}$ machine code
   - `$ javac Hello.java`: `javac` is the Java **compiler** `*.java` → `*.class`
   - `$ java Hello`: Invoke the JVM `java` and execute the **bytecode** contains in `Hello.class` JVM language(bytecode) → x86-64 machine language
2. Interpreted by the Java interpreter(`jshell`) `jshell` interpreter `jshell` interprets `Hello.java` interpreted from Java **directly** to the x86-64 machine language.

Compiler: 只知道CTT,不知道RTT!!

(1) Job

1. Translate source code into machine code or bytecode
2. Parse the source code written and check grammar and return error if grammar is 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.

(2) 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. The memory allocated on the heap stays there as long as there is 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. We can omit both memory addresses stored in the variable 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

- 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 but not uninitialized variables.
2. `this` is a reference variable that refers back to self, and is used to distinguish between two variables of the same name.
   - Expected behavior of `x = x` rather than `this.x = x`? Do nothing! The field has not been updated with the value of parameter `x`.
   - `this` can be **automatically added** if there is no ambiguity that we are referring to the 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`: a field or method with modifier `static` belongs to the class rather than the specific instance. In other words, they can be accessed/updated or invoked without even instantiating the class. **Note that**
   - Java actually prevents the use of this from any method with static modifier, or it will throw an error.
   - On the other hand, non-static methods including constructor has the keyword `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 an interface cannot extend from another class.
   - `extends` from abstract class or interface must override or declare `abstract`, or it will throw an error!
6. `super` call the constructor of the superclass
   - has to appear as the first line in the constructor
   - if there is no call to `super`, Java will automatically add the default `super()`
   - No `super` – 1 error, while Non-first `super` – 2 error
   - The methods that have been overridden can be called with the `super` keyword.
   - **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`.
8. `throws`: `RuntimeException` not add `throws`
9. `throw`: Actual throwing of exceptions.
10. `try/catch/finally`: group statements that check/handle errors together making code easier to read.

OOP Principle

1. **Information Hiding:** Expose as few fields/methods as possible
   - Isolate the internal representation 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 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, then it is not substitutable and the LSP is violated.
   - Ensure that any inheritance with method overriding does not introduce bugs to existing code
   - 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
   - A subclass should not break the expectations set by the superclass.

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:

- always invoked without being attached to an instance
- cannot access its instance fields or call other of its instance methods.
- the reference this has no meaning within a class method.
- should be accessed through the class

Constructor: No return type and access modifiers can be 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: If there is no constructor given, then a default constructor is added **automatically**. It takes no parameter and has no code written for the body.

Fully Qualified Name: i.e. `Circle.this.r`.

- starts with a sequence of class names separated by a dot.
- If the name refers to a field, the FQN is then followed by the keyword 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: i.e. `A C::foo(B1, B2)`

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

始终牢记Override和Overload都和变量名没关系!!!

Override: A **subclass** defines an instance method with the same **method descriptor** as an instance method in the parent class Generally, consider S<:T, A2<:A1, then `A1 T::foo(B,C)` can be **overriden** in class S as `A2 S::foo(B,C)`

Overloading: We create an overloaded method by changing the type, order, and number of parameters of the method but keeping the method name identical. **Note that** `contains(double x, double y)` and `contains(double y, double x)` are not distinct methods and are not overloaded

- Overload the class constructor using `this(...)`
- It is possible to overload static class methods

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**.
- an abstract class may inherit from concrete class to prevent instantiation of the class.

Interface: The abstraction models what an entity can do.

- 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: All primitive wrapper class objects are **immutable**, once create an object, it cannot be changed. ⇒ less efficient than primitive types.
Autoboxing: `Integer i = 4;`
- **Single-step process:** `Double d = 2;` will not compile

(Auto-) Unboxing: not restrict in only one step: `double d = i;`
Variance:

```
Object[] obj <-- intArray // ok
Integer[1] = "Hello"
// ok, as String <: Object. But will have
    runtime error, as integer array <- String!
```

Type Checking

1. **Only use CTT information for its type checking.**
2. View of Compiler is CTT!

```
class T { foo() {...} }
class S1 extends T { bar() {...} }
class S2 extends T { baz() {...} }
T x = new S1();
x.bar();
// Error, because view of compiler is CTT.
    Since type T has no method called bar
```

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

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

4. **Nominal:** subtyping relationship has to be explicitly declared

Consider `a = (C) b;`

Compile Time Check

1. Find CTT(b)
2. Determine whether there is a possible RTT(b) <: C, if impossible → compilation error
3. Some possible cases
   - CTT(b) <: C: simply widening
   - C <: CTT(b): narrowing and requires run-time checks. Consider C <: B: If CTT(b) = B and RTT(b) = C (or subtype of C), then it is allowed at run-time. If CTT(b) = B and RTT(b) = C (or other subtype of B that is not C), then it not allowed at run-time. Since there is a possibility, the compiler will add codes to check at run-time.
   - **C is an interface:** Let RTT(b) = B. Then it may have a subclass A such that A <: C (重点是看A和C之间有没有交集)
     `class A extends B implements C ..`
     If RTT(b) = A, then it is allowed at run-time.
4. Find CTT(a)
5. Determine whether there is a possible C <: RTT(a), if impossible → compilation error
6. Runtime Check for RTT(b) <: C

Run Time Check: Check if RTT(b) <: C

Common Subtype Relation

1. **Type Casting:** $A <: B$ → B b = a will compile
2. **Interface:** If a class C implements interface I, $C <: I$. Specifically, `I i2 = (I) new A();` compiles even though 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

1. Determine CTT(`obj`) and CTT(`arg`)
2. Determine all the methods with the name `foo` that are accessible in CTT(`obj`), including the parent of CTT(`obj`), grand-parent of CTT(`obj`), and so on. **(CTT向上找)**
   - An abstract method is considered accessible although there is no method implementation.
   - RTT(`obj`) must be a concrete class
     → must have an implementation of the abstract method
     → the implementation of the abstract method must override the 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 there is 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

1. Determine `RTT(obj)`
2. Starting from `RTT(obj)`, find the first method that match the method descriptor (continue up the class hierarchy)

Exception

1. The `try/catch/finally` keywords group statements that check/handle errors together making code easier to read.
2. Exceptions are instances of a subtype of the Exception class.
3. **`finally` is ALWAYS Executed!**
4. **Throwing Exception**
   - Declare that the construct is throwing an exception, with the `throws` keyword.
   - Create a new Exception object and throw it to the caller with the `throw` keywords. (`throws` ! = `throw`)
   - throw statement causes the method to **immediately return (the following part will not happen!!!)**
5. **Unchecked Exceptions:** (Perfect Code should not have) Not explicitly caught or thrown → something is wrong with the program and cause **run-time errors** (unchecked exceptions <: RuntimeException) i.e. `ClassCastException`

   $$\text{Unchecked Exceptions} \rightarrow \begin{cases} \text{caught} \\ \text{Not caught} \rightarrow \text{Error message} \end{cases}$$
6. **Checked Exceptions:** program **will not compile**. A checked exception must be handled.
7. **Normal Execution:** try → finally
8. **Error Execution:** try → catch → finally
   - Error find → all subsequent lines in `try` is **not** executed
   - look one-by-one from top to bottom for the first catch block
   - continues from the catch block to the finally block
   - Consider ExceptionX <: ExceptionY, then the following code

   ```
   catch(ExceptionY e) {// handle ExceptionY}
   catch(ExceptionX e) {// handle ExceptionX}
   ```

   will cause a compilation error since we will never catch `ExceptionX` because it will already be caught by `catch(ExceptionY e)`
   - **Pokemon Exception Handling:** use `catch(Exception e)` **above** other catch bolcks that hands the subclass of Exception → **Compilation Error!**
9. **Overriding:** Override a method that throws a checked exception → the overriding method must throw only the **same**, or a **more specific** checked exception, than the overridden method.**(By LSP)**. The caller of the overridden method cannot expect any **new** checked exception beyond what has already been "promised" in the method specification.

Generics Consider `class Pair<S,T>`

- **type parameters:** `S,T`, **generic type:** `Pair<S,T>`
- To instantiated a generic type, pass in **type arguments** and only **reference types** (i.e. `Integer`) can be used
- Generic type $\xrightarrow{\text{instantiated}}$ parameterized type
- For `class A<T> extends B<String,T>`, `T` is **passed** from the `A` class to the `B` class. ⇔ declare a generic type `T` in class `A<T>` and **use** the generic type `T` in `.. extends B<.., T>`.

Generic Methods

- `<T>` is added **before** the return type of the method.
- `<T>` only scoped in its own method
- To call a generic method, we need to pass in the type argument placed **before** the 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 Parameter Confusion: Declare type parameter with the same name but are actually different type i.e. **`T#1, T#2`**
Type Erasure: Java erase the type parameters and type arguments during **compilation**, resulting compiled code **does not** to be recompiled when encountering new types.
- If the type parameter is **bounded**, it is replaced by the **bounds**

Generics and Array

- **Heap pollution:** A variable of a parameterized type refers to an obj. that is not of that para. type. (`ArrayStoreException`)
- Array is **reifiable type**, a type where full type information is available during **run-time**.
- 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**.
- Always use **`@SuppressWarning` to the most limited scope** to avoid unintentionally suppressing warnings that are valid concerns from the compiler.
- Suppress a warning only if we are sure that it will not cause a type error later.
- Always add a comment to explain why safely suppressed.

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.
- Raw types must not be used in your code, ever **except** using it as an operand of the `instanceof` operator.

Wildcards: `A.<Object,Object>foo(circles, c)` won't compile
- Generics is **invarient** → `Seq<Circle>` <: `Seq<Object>`
- We have to contrain array type to be **same** in Generics
- inveriant → Avoided the possibility of **run-time errors** by avoiding covariance arrays

Upper-Bounded Wildcards: covariance
- S <: T ⇒ `A<? extends S>` <: `A<? extends T>`
- `A<S>` <: `A<? extends S>`

Lower-Bounded Wildcards: contravariance
- S <: T ⇒ `A<? super T>` <: `A<? super S>`
- `A<S>` <: `A<? super S>`

PECS: "Producer Extends; Consumer Super"
Unbounded Wildcards: `Array<?>` :> `Array<T>`
`Array<?>, Array<Object>, and Array`
- `Array<?>` is an array of objects of some specific unknown type;
- `Array<Object>` is an array of Object instances, with type checking by the compiler;
- `Array` is an array of **Object instances**, **without** type checking.

Revisiting Raw Types We can use unbounded Wildcards like 1. `a instanceof A<?>` 2. `new Comparable<?>[10];`
Reifiable type: A type where no type information is lost during compilation. `Comparible<?>` is **reifiable**. Since we don't know what is the type of ?, no type information is lost during erasure!
Common Mistake: interface cannot instantiate

```
void foo(List<? super Integer> list){ }
foo(new List<Object>) // Error!
```