Notes for CS2030

Courtesy of TA Gan Chin Yao

This is a note compiled by myself when I was taking the module CS2030 as a student. It consists of subtle details about Java that are often the source to tricky questions. These details are often not taught explicitly in lectures, and are often discovered through practices and experiences. These knowledges may or may not be applicable, depending on the professor setting the paper, but it is definitely part of your syllabus curriculum. In general, these knowledges can help you get through trick questions. Take this note as a supplement. Read only if you have time.

- When inheriting interface, note that the methods are by default ***public***. Therefore, when overriding, you must put ***public***, otherwise you are assigning weaker access modifier to overridden method (i.e. compile error).
- ***static*** method in ***interface*** must have a body
- Cannot override a ***static*** method belonging to superclass in subclass with non-static. Compile time error.
- If you have a ***static*** method in superclass, and the same name ***static*** method in subclass, you are not overriding, but instead your subclass static method is totally not related to superclass at all. Therefore if A a = new B(); a.f(); => calling static in A, NOT in B.

  e.g. class A {

  ```
      public static void method() {}
  ```

  }

  class B extends A {

  ```
      public static void method() {} // this method is not overriding superclass
  ```

  }
- Subclass overridden method can either don't ***throws***, or ***throws*** more specific exceptions.

  e.g. class A {

  ```
      public void method() throws IllegalArgumentException {}
  ```

  }

  class B extends A {

```
    @Override
    public void method() throws Exception {} //compile error, throwing more
general exception
}
```

- Local variables do not get initialized. Only instance variable get assigned default values. Therefore, may result in "Variables might not have been initialized" compile time error.

```
e.g. class A {
    public void method() {
        int k; // local variables do not get initialized
        System.out.println(k); // compile error
    }
}
class B {
    int k; // get assigned default value of 0.
    public void method() {
        System.out.println(k); // ok
    }
}
```

- For downcasting, if it is possible, no compile time error, but we still must put explicit cast, otherwise won't compile.

```
e.g. interface J {}
J j = …
class A implements J {} => A a = (A) j; //must cast, becos j could be object of other
type.
J j = new A(); // no need cast
```

- unreachable statement WON'T even compile! BUT if you call a method that throw an error, no way for the compiler to know, therefore next line can still compile.

```
E.g. void f() {
    throw new Exception();
    // unreachable here
    int d = 5; // wont compile
```

```
}
void f2() {
f();
// though f() throw exception, to compiler it is still REACHABLE here.
int d = 5; // will compile
}
```

- the following wont compile:

```
static void f() throws IllegalArgumentException {
  try {
    throw new Exception();
  } catch (IllegalArgumentException e) {
    System.out.println("Caught in f");
  }
```

becos throws is not handling Exception() !! -> Unhandled exception

- Compile error will occur if you catch a more general exception (i.e. superclass), and then the next catch is a more specific one(i.e. subclass), stating that the specific exception has already been caught.
- Works: List<A> a = new ArrayList<>();

```
a.add(new A());
a.add(new B());    // assume B extends A
a.add(new C());    // assume C extends A or B
```

becos a B is also an A, a C is also an A.

- Double a = 8; // compile error, expecting Double, found int
- Double a = 8.0; // ok, auto box
- When autowrapping, must be of same type, i.e. int -> Int, cannot int -> Double also, no matter if its widening or narrowing.
- If subclass is ***Integer*** method and super class is ***int*** method, it does not override superclass, therefore superclass version is call and no polymorphism is at play
- - For variable capture, it is the local variable inside the method that cannot be change.

```
e.g. class A {
    int j = 5; // this can be changed
```

```
void m() {
    int k = 5; // it is this that can't be change
    class B {
        int l = 5; // this can be changed also
    }
}
```

- Careful of things like List a = new HashSet(); // wrong as HashSet does not implement List
- Array of primitive type is invariant. Array of reference type is covariant. Therefore can't cast intArr = doubleArr, but can A_array = B_array (Where A and B are classes).
- For instance final variable, you must explicitly initialized it when declaring, or latest in the constructor only. Method assigning is not allow.

```
e.g. class A {
    final int k; // error, becos did not initialize, and no constructor to initialize
    void m() {
        final int k; // ok in method as a local variable
        k = 5; // ok
        k = 10; // not ok cos its final
    }
}
```

- All instance variable, primitive or reference, in interface are implicitly public static final
- Declaring ***final static*** does not prevent subclass from declaring a static method with same signature (no overriding is at play, becos 2 static methods are totally unrelated).
- ***static*** method in ***interface*** can ONLY be called via the interface class name, unlike instance variable of normal class can call static method.
- ***interface*** cannot ***implements***, but can ***extends*** multiple ***interface***. Cannot extends normal class, only interface

```
e.g interface K extends L, M {} // ok, assuming L and M are interface
```

- interface: 2 unrelated interfaces can cast each other, becos it is possible

```
e.g interface J {}
```

interface K {}

class A implements J {}

J j = new A();

K k = (K) j; // will succeed. This is becos even though in this case class A does not implements K at all, but it could be possible that there could be another class that implements both J and K, and be referenced to by J j. Hence, that would cause the cast to succeed. Generally in Java, as long as something is possible (even though in your programme it is not possible, but you can write a program that makes something possible), then java will allow it to compile.

- enum cannot be generic
- enum is implicitly final class
- enum cannot have extends clause, because it already extends Enum<T>
- enum can implements interface
- no top level static class. static class must be inner class
- <T extends(NO SUPER) Number> void printPositiveBytess(List<T> list) //ok
  when writing in parameter, no extends
  (List<T extends/super ...>) wrong. only List<? extends/super ...> ok
  But in class definition its ok. e.g. class Cat<T extends ..> ok
  NO concept of Cat<T super ..>, only extends
- Whenever you call static method of class with generic, example class A<T> {}
  it should never be A<Integer>.staticMethod()
  it should be A.staticMethod() becos <T> is useless in static context (in static method, no instance object is initialize. Therefore we cannot tell what type of object T is referring to, hence T is useless in static context), therefore A<Integer> makes no sense.
- In abstract class, method without explicitly declaring abstract is NOT abstract, but method in interfaces are implicitly abstract even if you never explicitly write abstract.
- List<Integer> list = new LinkedList<>(); Right hand side can be empty (i.e. inferred)
  List<?> list = new LinkedList<>(); valid also
  List<Integer> list = new LinkedList<?>(); RHS cannot have question mark '?'

- only local primitive and reference to object stay on stack, others on heap

  e.g. class A {

  int y; // instance var belonging to the object store on HEAP

  void m() {

  int k; // local var stores on stack

  A a = new A(); // reference to obj store on stack

  }

  }
- constructor cannot take generic param
- Call fork() as early as possible because fork() does not block. Once you call fork(), other worker can take the task and do the job. If you call fork() later, the time spent to execute other code could be parallelly used by other worker thread to compute
- Call compute() first before you call join(). join() will block and wait for a result before proceeding. It is helpful to see compute() as a fixed time that has to be spent no matter where you call it. So if you call compute() first, while the time is spent on executing compute(), the forked worker thread can continue to calculate parallelly, thereby reducing the amount of block time left when you call join()
- Call join() in reverse order for best performance. i.e. f1.fork(); f2.fork(); f3.fork(); f3.join();f2.join(); f1.join();
- Do not ever call join() without first calling fork(). It will compile, but you will end up in infinite wait.
- Do not call fork and not call join(). It will compile, but it is pointless to call fork() alone.