CS2030 AY18/19 SEM 2

WEEK 7 | 08 MARCH 19

TA GAN CHIN YAO

DISCLAIMER

Slides are made by me, unofficial, optional
Available to download at bit.ly/cs2030_gan
Slides (if any) will be uploaded on
Friday weekly

Q1.

Concepts:
Generics, Type erasure, wild cards

1. For each of the statements below, indicate if it is a valid statement with no compilation error. Explain why.

(a) List<?> list = new ArrayList<String>();

Yes, valid. ArrayList implements List and the wildcard type is bound to String.

Note: It is important to mention ArrayList implements List, as the following line is invalid:
List<?> list = new HashSet<String>(); since HashSet does not implements List

(b) List<? super Integer> list = new List<Object>();

No. Cannot instantiate an interface List.

Trick question. Take note that the following is valid: List<? super Integer> list = new ArrayList<Object>();

Q1.

1. For each of the statements below, indicate if it is a valid statement with no compilation error. Explain why.

(c) List<? extends Object> list = new LinkedList<Object>();

Yes, valid. LinkedList implements List and Object is upper bounded by ? extends Object.

(d) List<? super Integer> list = new LinkedList<>();

(d) Yes, valid. LinkedList implements List and Java infers the type to be Integer.

Q2.

Concepts:
Generics, raw type

2. Consider a generic class `A<T>` with a type parameter `T` having a constructor with no argument. Which of the following expressions are valid (with no compilation error) ways of creating a new object of type `A`? We still consider the expression as valid if the Java compiler produces a warning.

(a) `new A<int>()`

Error. A generic type cannot be primitive type. Java generic type must be referenced type.
Primitive: boolean, int, long, short, byte, char, float, double

(b) `new A<>()`

Ok. Java will create a new class replacing T with Object.

(c) `new A()`

Ok too. Same behaviour as above, but using raw type (for backward compatibility) instead. Should be avoided in our class. You will get a warning from compiler that raw type is used, but it will still compiled.

Note: Raw type is bad as it defeats the purpose of generics. Generics is created to allow code reuse, and to allow you to utilize compile time detection to write safer code. Using raw type disable this compile time detection.

Q3.

Concepts:
Wrapper class, auto boxing and unboxing

3. Given the following Java program fragment,

```
class Main {
    public static void main(String[] args) {
        double sum = 0.0;

        for (int i = 0; i < Integer.MAX_VALUE; i++) {
            sum += i;
        }
    }
}
```

you can determine how long it takes to run the program using the `time` utility

```
r-49-103-25-172:dd gan$ time java Main

real    0m2.996s
user    0m2.986s
sys     0m0.056s
r-49-103-25-172:dd gan$
```

Q3.

Now, replace `double` with the wrapper class `Double` instead. Determine how long it takes to run the program now. What inferences can you make?

Despite it's conveniences, there is an associated overhead in the use of autoboxing. In addition, due to immutability of `Integer`, many objects are created.

```
class Main {
    public static void main(String[] args) {
        Double sum = 0.0;
        for (int i = 0; i < Integer.MAX_VALUE; i++) {
            sum += i;
        }
    }
}
```

Each loop, Double auto unbox to double. At the end before the loop ends, double is auto box back to Double again. The cycle repeats until the loop break.

```
r-49-103-25-172:dd gan$ time java Main

real    0m7.567s
user    0m7.494s
sys     0m0.332s
```

As you can see, using Double class causes the problem to run slower (3x slower in my computer). Note that the speed varies from computer.

Primitive is always the fastest. Use int. Only use the wrapper class Integer when you need an object, e.g. for generics. Note that primitive is not an object. i.e. int is not an object, but Integer is an object

Q4.

Concepts:
Integer caching, == VS equals()

4. Recall that the `==` operator compares only references, i.e. whether the two references are pointing to the same object. On the other hand, the `equals` method is more flexible in that it can override the method specified in the `Object` class.

   In particular, for the `Integer` class, the `equals` method has been overridden to compare if the corresponding `int` values are the same or otherwise.

   What do you think is the outcome of the following program fragment?

```
Integer x = 1;
Integer y = 1;
System.out.println(x == y);

x = 1000;
y = 1000;
System.out.println(x == y);
```

True

False

Why do you think this happens?
Hint: check out Integer caching

- We would expect the top fragment to be false since we are comparing object references. Since integers within a small range are very often used, it makes sense for the Integer class to keeps a cache of Integer objects within this range (-128 to 127) such that autoboxing, literals and uses of Integer.valueOf() will return instances from that cache instead.
- Rather than worry over the effects of caching or otherwise, the bottomline is to always use equals() to compare two reference variables. (Which is why you have been overriding equals())

Q4.

Concepts:
Type erasure

5. Compile and run the following program fragments and explain your observations.

(a)
```
import java.util.List;

class A {
    void foo(List<Integer> integerList) {}
    void foo(List<String> StringList) {}
}
```

Compile error. After type erasure, the class becomes:

```
class A {
   void foo(List integerList) {}
   void foo(List StringList) {}

}
```

So you can see that the 2 methods have the same signature, therefore clash. You are not overloading methods. Hence compile-time error

Q5.

Concepts:
Static T in generics

(b)
```
class B<T> {
    T x;
    static T y;
}
```

Compile error. Recall that static variable only has 1 copy for any number of class instances created. Compile error because the 'T' in static is ambiguous. Imagine this:
B<Integer> b1; // here Integer is acting as 'T', for compile time type enforcement
B<String> b2; // here String is acting as 'T', for compile time type enforcement

Then, what is static T y? Is it Integer or String? Since static only has 1 copy, clearly we can see there is a problem here since static T y can be treated as Integer or String.

Q5.

(c)
```
class C<T> {
    static int b = 0;
    T y;

    C() {
        this.b++;
    }

    public static void main(String[] args) {
        C<Integer> x = new C<>();
        C<String> y = new C<>();

        System.out.println(x.b);
        System.out.println(y.b);
    }
}
```

Compile ok. Output:
2
2
Although it seems there are two different classes, C<Integer> and C<String>, there is still only one class C.
As such, there is only one copy of the static variable b.

Q6.

6. Which of the following code fragments will compile? If so, what is printed?

(a)
```
List<Integer> list = new ArrayList<>();
int one = 1;
Integer two = 2;

list.add(one);
list.add(two);
list.add(3);

for (Integer num : list) {
    System.out.println(num);
}
```

Autobox int 1 to Integer(1)

No autoboxing occurs. Integer(2) just get added inside list

Autobox int 3 to Integer(3)

Important: No auto unboxing happens. Calling System.out.println(Object obj) method. Then, printing Integer.toString() method of each num

Compile ok. Output:

1
2
3

Q6.

(b)
```
List<Integer> list = new ArrayList<>();
int one = 1;
Integer two = 2;

list.add(one);
list.add(two);
list.add(3);

for (int num : list) {
    System.out.println(num);
}
```

Autobox int 1 to Integer(1)

No autoboxing occurs. Integer(2) just got added

Autobox int 3 to Integer(3)

Important:
list now contains: Integer(1), Integer(2), Integer(3)
Step 1: Integer(1) auto unbox to become primitive int 1
Step 2: Primitive int 1 is assigned to num.
Step 3: System.out.println(int i) method is called
Step 4: Repeat step1-3 for Integer(2) and Integer(3)

Compile ok. Output:
1
2
3

Q6.

(c)
```
List<Integer> list = Arrays.asList(1, 2, 3);

for (Double num : list) {
      System.out.println(num);
}
```

Compile error.

Integer cannot be converted to Double.

double x = (int) 5; // ok , widening conversion

Double y = new Integer(1); // NOT OK, Integer cannot be converted to Double. Both Integer and Double are classes on its own, not associated.

Q6.

(d)
```
List<Integer> list = Arrays.asList(1, 2, 3);

for (double num : list) {
    System.out.println(num);
}
```

list now contains: Integer(1), Integer(2), Integer(3)

Step 1: Integer(1) auto unbox to primitive int 1
Step 2: primitive int 1 gets widening conversion to become double 1.0

Step 3: 1.0 is now assigned to num
Step 4: Call System.out.println(double num) method
Step 5: Repeat step1-4 for Integer(2), Integer(3)

Compile ok. Output:
1.0
2.0
3.0

Q6.

(e)
```
List<Integer> list = new LinkedList<>();
list.add(5);
list.add(4);
list.add(3);
list.add(2);
list.add(1);

Iterator<Integer> it = list.iterator();
while (it.hasNext()) {
    System.out.println(it.next());
}
```

Compile ok. Output:
5
4
3
2
1

list now contains: Integer(5), Integer(4), Integer(3), Integer(2), Integer(1)

Step 1: Integer(5) get called in hasNext()
Step 2: Call System.out.println(Object obj) method
Step 3: Execute Integer.toString() method
Step 4: Repeat step1-3 for Integer(4), Integer(3), Integer(2), Integer(1)

Note: No auto unboxing happens

SUMMARY CONCEPTS

- Generics
- Type erasure
- Wild cards
- Raw type
- Wrapper class
- Auto boxing, auto unboxing
- Integer caching
- == vs equals()

Guide for wild cards

Assuming:
class A
class B extends A
class C extends B
class D extends C
class E extends D

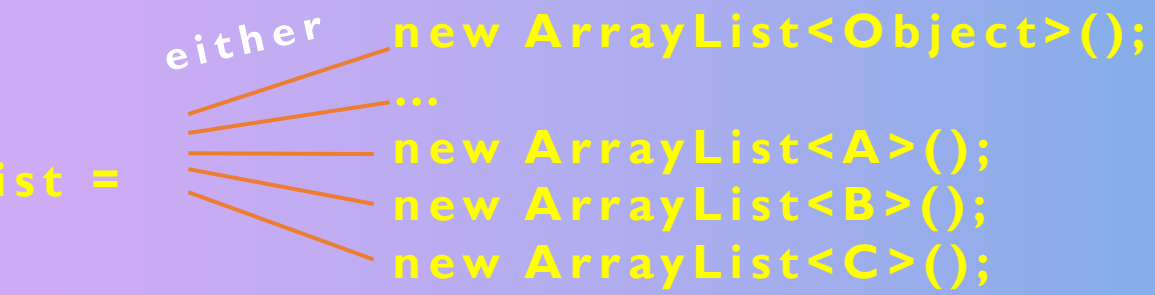
ArrayList<? extends C> list = either
- new ArrayList<C>();
- new ArrayList<D>();
- new ArrayList<E>();
- ...

- We cannot add anything except null.
- list.add(null); // ok, because any class can refer to null. E.g. E e = null; // ok
- list.add(new C()); // error because list could be referring to new ArrayList<D>(); Then, we are doing a subclass reference D to a superclass object C, which is not allowed.
- list.add(new D()); // error because list could be referring to new ArrayList<E>();
- list.add(new E()); // error because even though here we only have up till class E, it is entirely possible to have more classes extending class E. i.e. we know that Object is the top class for all classes, but there is no bottom class since we can always inherit.

- When we call list.get(), we must use C, or any superclass of C, e.g. B or A.
  i.e. assuming list contains some items beforehand:
- C c = list.get(0); // ok, because it is valid to have things like C c = new E(); C c = new D();
- B b = list.get(0); // ok
- A a = list.get(0); // ok
- Object obj = list.get(0); //ok
- D d = list.get(0); // not ok, because it could be referring to new ArrayList<C>(); Then, we are doing a subclass variable D referencing a superclass object C.
- E e = list.get(0); // not ok, because list could be referring to new ArrayList<C or D>();
- This is because superclass variable can reference subclass object. We know that <? extends C> can only refer to an actual ArrayList<C or subclass of C>. Therefore, by using C or any superclass of C as the reference, we are guaranteed safe since superclass variable can reference subclass object.

Guide for wild cards

Assuming:
class A
class B extends A
class C extends B
class D extends C
class E extends D



- When we add, we can only add object C or its subclass, then we are guaranteed safe.
- Assume we pick ArrayList<? extends C> list = new ArrayList<A>();
- list.add(new C()); // ok, because superclass variable can reference subclass object
- list.add(new E()); // ok, A a = new E() is valid
- list.add(new B()); // not ok. Even though it is possible to get ArrayList<? super C> list = new ArrayList<B>(); realised that it is also entirely possible to get list referencing ArrayList<A>();
- Hence, if you add anything other than C or its subclass, there is a chance that list is not compatible.

- To get, we can only assign the top level reference, i.e. Object
- Object obj = list.get(0); // ok, assuming there is content to get
- A a = list.get(0); // not ok, because list can be referencing ArrayList<Object>(); Then, we are doing A a = new Object(); which is not valid
- C c = list.get(0); // not ok, because list can be referencing ArrayList<B/A/any superclass/Object>();
- E e = list.get(0); // not ok, same reason as above
- Therefore, the only way to be assured that we can refer to the object, we need to go all the way to the top in class hierarchy, which is Object.

Guide for wild cards

Assuming:
class A
class B extends A
class C extends B
class D extends C
class E extends D

ArrayList<?> list =

either

new ArrayList<Object>();
...
new ArrayList<>();
new ArrayList<B>();
new ArrayList<String>();
new ArrayList<Integer>();
new ArrayList<Basically any class>();

- We cannot add anything except null because list can refer to any class.
- list.add(null); // ok, because any class reference can refer to null, i.e. A a = null; // ok
- list.add(new A()); // not ok, list could be referring to new ArrayList<Integer>();
- list.add("String"); // not ok, list could be referring to new ArrayList<A>();
- Hence, we can easily see why we cannot add anything, except null.

- When we get, only possible reference is Object, because Object can be referring to anything.
- Object obj = list.get(0); // ok, assume there is content. This is because following Objects are valid:
- Object obj = new Integer(3); // valid
  Object obj = new A(); // valid
- We can see that obj can refer to object of any class.
- String string = list.get(0); // not valid, because list can be referring to ArrayList<Integer>();
- Then this would mean String string = Integer(3); // Not valid. (Integer(3) is just an example)
- Hence, we can see why we can only use Object to reference a .get() item, since list can point to anything.

No wild card

Assuming:
class A
class B extends A
class C extends B
class D extends C
class E extends D

ArrayList<C> list = only new ArrayList<C>();

- We can add anything that is C or subclass of C, or null.
- list.add(new C()); // ok
- list.add(new D()); // ok, same idea as C c = new D(); which is valid
- list.add(new E()); // ok
- list.add(new B()); // not ok, because C c = new B(); is not valid
- list.add(null); // ok, because C c = null; is valid
- list.add(new Object()); // not ok

- When we get, we can use C reference, or any of C superclass as reference. This is because superclass variable can reference subclass object.
- C c = list.get(0); // ok, assume there is item in the list
- B b = list.get(0); // ok, because B b = new C(); is valid
- A a = list.get(0); // ok, because A a = new A(); is valid
- Object obj = list.get(0); // ok, because Object is superclass of C
- D d = list.get(0); // not ok, because D d = new C() is not valid
- E e = list.get(0); // not ok, same reason as above
- String s = list.get(0); // not ok, C and String are not related at all

The gist

- Superclass variable can refer to subclass object
- Subclass variable cannot refer to superclass object
- As long as there is a chance of ambiguity, Java compiler will not let you compile.
  i.e. Sometimes work, sometimes don't work => cannot compile
  100% will work => can compile

Assuming:
class Animal
class Cat extends Animal

Variance

- Covariant: Preserved. Cat[] is an Animal []
  Hence, Animal [] b = new Cat[3]; // valid
  - Array is covariant in java.
  - Method return is covariant in Java (You can override with more specific type)
  - List<? extends Cat> is covariant
- Contravariant: Reversed. Animal [] is a Cat[]
  We cannot write Cat[] cat = new Animal[3]; // not valid in Java
  - Array is not contravariant in Java (some languages are)
  - List<? super Cat> is contravariant
- Invariant: NIL. Cat[] is not an Animal[]. Animal[] is not a Cat[]
  - Java is invariant argument type. (i.e. method overloading)
  - List<Cat> is invariant to List<Animal>, vice versa
- Bivariant: Both covariant and contravariant applies.
  - Array is not Bivariant in Java.

PECS:

- Use covariance for methods which return a generic type
- Use contravariance for methods which take a generic type
- Use invariance for methods which both accepts and returns a generic type

QUESTIONS?