1. Consider the following program fragment.

```
1  class A {
2    int x;
3
4    A(int x) {
5      this.x = x;
6    }
7
8    public A method() {
9      return new A(x);
10   }
11 }
12
13 class B extends A {
14   B(int x) {
15     super(x);
16   }
17
18   @Override
19   public B method() {
20     return new B(x);
21   }
22 }
```

Does it compile? What happens if we switch the `method` definitions between class `A` and class `B` instead? Give reasons for your observations.

Suggested Guide:

There is no compilation error in the given program fragment. Any existing code that invokes `A`'s `method` prior to being inherited would still work as this existing code would just invoke `B`'s overridden method instead (as `B` inherits from `A`).

When we switch the method definitions, `A`'s `method` now returns a reference to a `B` object. Therefore when overriding it with a method that returns `A`, we can not guarantee that the returned object is a `B` object as `A` is not a subtype of `B`. Or to put it another way, the return type of `B`'s `method` cannot not be a supertype of the return type of A's `method`. Thus this overridden method is not allowed and results in a compilation error.

Now suppose Java does allows the `method()` of class `A` and `B` to be swapped. Consider the following code fragment, where `g()` is a method defined in class `B` (but not in class `A`).

```
1 void f(A a) {
2   B bNew = a.method();
3   bNew.g();
4 }
```

Say someone else calls `f(new B())`. `a.method()` on Line 2 will invoke the `method()` defined in `B`, which returns an object of class `A`. So now, `bNew` which has a compile-time type of `B` is referencing an instance of `A`. In Line 3, `bNew.g()`

invokes a method `g()`, which is defined only in `B`, through a reference of (run-time) type `A`. But since `bNew` is referencing to an object with run-time type `A`, this object does not have a defined method `g()`!

2. Consider a generic class `A<T>` with a type parameter `T` with a default constructor. Which of the following expressions are valid (*i.e.*, *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>();`

   (b) `new A<>();`

   (c) `new A();`

Suggested Guide:

(a) **Error**. A generic type cannot be a primitive type. You need to use a wrapper class, as covered in the lectures.

(b) **Valid**. Java will create a new class replacing `T` with `Object`.

(c) **Valid**. Same behaviour as above, but using *raw type* (*for backward compatibility*) instead. Should be avoided in our class!

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

   (a) Program A

```
1  import java.util.List;
2
3  class A {
4    void foo(List<Integer> integerList) {}
5    void foo(List<String> stringList) {}
6  }
```

   (b) Program B

```
1  class B<T> {
2    T x;
3    static T y;
4  }
```

   (c) Program B

```
1  class C<T> {
2    static int b = 0;
3    C() {
4      this.b++;
5    }
```

```
6     public static void main(String[] args) {
7       C<Integer> x = new C<>();
8       C<String> y = new C<>();
9
10      System.out.println(x.b);
11      System.out.println(y.b);
12    }
13  }
```

Suggested Guide:

(a) **Compilation error**. This is because after type erasure our two methods `foo` have the same method signature:

```
1  class A {
2    void foo(List integerList) {}
3    void foo(List stringList) {}
4  }
```

(b) **Compilation error**. For the field declaration `T x`, the type of `x` is bounded to the type argument `T`. This is fine for instance field. Unfortunately, for class fields (*i.e.*, *using* `static` *keyword*), there is only one copy of `y`. Which type argument should it be bounded to? This ambiguity is why the Java compiler does not permit this.

(c) **Output**.

```
1  2
2  2
```

Although it seems there are two different classes (*i.e.*, `C<Integer>` *and* `C<String>`), there is still only one class `C`. There is only one copy of the class variable `b`.