CS2030S

AY24/25 S2 **Problem Set 3** 12-13 February 2025

1. Suppose we have the following types:

   - `SubR` <: `R` <: `SuperR`
   - `SubE` <: `E` <: `SuperE` <: `Exception`

   We have the following class `A`. The implementation of the method `foo` and other details in `A` are omitted.

   ```
   class A {
     R foo() throws E { ... }
   }
   ```

   Now, suppose we have a class `B` that inherits from `A`. `B` overrides the method `foo` in `A`. Consider the following method declaration of `foo` in `B`. Which would violate the substitutability of `A` by `B` and thus should not be allowed? Explain your answer in the context of the code snippet below:

   ```
   void bar(A a) {
     try {
       R r = a.foo();
       // use r
     } catch (E e) {
       // handle exception
     }
   }
   ```

   (a) `SubR foo() throws E { ... }`

   (b) `SuperR foo() throws E { ... }`

   (c) `R foo() throws SubE { ... }`

   (d) `R foo() throws SuperE { ... }`

   Comments:

   By LSP, we should be able to substitute `A` in the method `bar` with a `B`.

   (a) No violation. This implementation of `foo` returns a `SubR`, which can be assigned to `R` following a widening type conversion. So this is allowed in Java.

   (b) It violates substitutability. This implementation of `foo` returns a `SuperR`, which cannot be assigned to `R`. An explicit narrowing type conversion is needed. Further, the method `bar` might make use of methods in `R` that are not implemented in `SuperR`.

   (c) No violation. This implementation of `foo` throws a `SubE`, which can be caught by the `catch` statement. So this is allowed in Java.

   (d) It violates substitutability. This implementation of `foo` throws a `SuperE`, which is not caught by the `catch` statement.

2. Java provides an abstract class called `Number` that is the superclass of all primitive wrapper classes. `Number` is also the superclass of `BigInteger`, a class that supports arbitrary-precision integers. The primitive wrapper classes and `BigInteger` implement the `Comparable<T>` interface.

   Ah Beng first wrote the following method to convert an array of `BigInteger` to an array of `short` values. The method takes in a parameter `threshold`. Any value larger than the threshold is set to 0.

```
public static short[] toShortArray(BigInteger[] a, BigInteger threshold) {
  short[] out = new short[a.length];
  for (int i = 0; i < a.length; i += 1) {
    if (a[i].compareTo(threshold) <= 0) {
      out[i] = a[i].shortValue();
    }
  }
  return out;
}
```

As he continued to code, he realized that he also needed to convert an array of `Integer` and an array of `Double` to an array of `short`. He thus duplicated his method above and replaced `BigInteger` with `Integer` and `Double` respectively. He ended up with two more methods:

```
public static short[] toShortArray(Integer[] a, Integer threshold) {
  short[] out = new short[a.length];
  for (int i = 0; i < a.length; i += 1) {
    if (a[i].compareTo(threshold) <= 0) {
      out[i] = a[i].shortValue();
    }
  }
  return out;
}

public static short[] toShortArray(Double[] a, Double threshold) {
  short[] out = new short[a.length];
  for (int i = 0; i < a.length; i += 1) {
    if (a[i].compareTo(threshold) <= 0) {
      out[i] = a[i].shortValue();
    }
  }
  return out;
}
```

Soon, he realized that he needed to do this for all other wrapper classes. Instead of overloading the method `toShortArray` multiple times, he decided to write a single method that generalizes the above methods.

(a) His first few attempts below, however, did not work correctly. Explain why these attempts are not correct.

(i)

```
public static short[] toShortArray(Object[] a, Object threshold) {
  short[] out = new short[a.length];
  for (int i = 0; i < a.length; i += 1) {
    if (a[i].compareTo(threshold) <= 0) {
      out[i] = a[i].shortValue();
    }
  }
  return out;
}
```

(ii)

```
public static short[] toShortArray(Number[] a, Number threshold) {
  short[] out = new short[a.length];
  for (int i = 0; i < a.length; i += 1) {
    if (a[i].compareTo(threshold) <= 0) {
      out[i] = a[i].shortValue();
    }
```

```
  }
  return out;
}
```

(iii)

```
public static short[] toShortArray(Comparable[] a, Comparable threshold) {
  short[] out = new short[a.length];
  for (int i = 0; i < a.length; i += 1) {
    if (a[i].compareTo(threshold) <= 0) {
      out[i] = a[i].shortValue();
    }
  }
  return out;
}
```

Comments:

The elements of the array must support the `compareTo` method and the `shortValue` method. `Object` provides neither, while `Number` only provides `shortValue` and `Comparable` ony provides `compareTo`. Furthermore, `Comparable` is used as a raw type in the above and is not type-safe.

(b) Ah Beng discovered Java supports generics. Particularly, he found that a type parameter can have multiple bounds using the `&` symbol. For instance, `<T extends S1 & S2>` means that the type variable `T` is a subtype of both `S1` and `S2` [1].

Using generics with bounded type parameters, help Ah Beng to re-write all his methods into a single generic method.

Comments:

```
public static <T extends Number & Comparable<T>>
    short[] toShortArray(T[] a, T threshold) {
  short[] out = new short[a.length];
  for (int i = 0; i < a.length; i += 1) {
    if (a[i].compareTo(threshold) <= 0) {
      out[i] = a[i].shortValue();
    }
  }
  return out;
}
```

A good question was asked in a couple of recitation classes: What happens to the code after type erasure? Would `T` be erased to `Number` or to `Comparable`?

The type erasure process erases `T` to be the first bound, which in this case is `Number`. It then type casts the variable to `Comparable` so that `compareTo` can be called.

3. Consider the generic class `A` and the different attempts to create a subclass `B` of `A` below.

```
class A<T> {
  public void fun(T x) {
    System.out.println("A");
  }
```

[1] A class bound must be specified first before an interface bound

```
}

// (i)
class B extends A<String> {
  public void fun(String i) {
    System.out.println("B");
  }
}

// (ii)
class B extends A<String> {
  public void fun(Object i) {
    System.out.println("B");
  }
}

// (iii)
class B extends A<String> {
  public void fun(Integer i) {
    System.out.println("B");
  }
}
```

(a) For each of the attempts, does it compile? Explain why in terms of overloading and overriding.

Comments:

(i) The code compiles.

Is this an example of method overloading or overriding? From the code, it appears that `B::fun(String)` overrides `A::fun(String)` inherited from `A<String>`. However, since `A::fun(String)` undergoes type erasure and becomes `A::fun(Object)` at runtime, does this mean that `B::fun(String)` is actually overloading `A::fun(Object)`, despite the programmer's intent to override?

To ensure proper method overriding after type erasure, Java introduces a bridge method in `B` that internally calls `B::fun(String)`. Bridge methods are special compiler-generated methods that cannot be invoked directly by programmers; they exist solely for runtime consistency in dynamic method binding.

A bridge method is always generated when (i) a type extends/implements a parameterized type and (ii) type erasure changes the signature of one or more inherited method.

The erased `A` and `B` look like this:

```
class A {
  public void fun(Object o) {
    System.out.println("A");
  }
}

class B extends A {
  public void fun(Object o) { // Bridge method
    this.fun((String) o);
  }

  public void fun(String i) {
    System.out.println("B");
  }
}
```

`fun(String)` overloads `fun(Object)`; `B::fun(Object)` overrides `A::fun(Object)`.

(ii) The code does not compile. Due to the implicitly created bridging method in `B`, there are two `B::fun(Object)` methods. This is not allowed in Java.

(iii) The code compiles. The erased `B` looks like this:

```
class B extends A {
  public void fun(Object o) { // Bridge method
    super.fun((String) o);
  }

  public void fun(Integer i) {
    System.out.println("B");
  }
}
```

`fun(Integer)` overloads `fun(Object)`.

(b) What is the output if the following code is run for each of the compilable implementations of `B` above?

```
A<String> a = new B();
a.fun("2");
```

Comments:

For (i), `B::fun` is invoked and it prints "`B`".

Although this may seem intuitive, the actual behavior is more subtle due to generics and type erasure. During compilation, the compiler examines class `A` (the compile-time type of the target) to determine which methods can be invoked. Since there is only one such method, the erased method descriptor `void fun(Object)` is stored.

At runtime, Java checks class `B` (the runtime type of the target) for a method matching this descriptor. It finds the bridge method `void fun(Object)` in `B` and invokes it. The bridge method then calls `B::fun(String)`, which prints "`B`".

For (iii), the process is the same and the bridge method in class `B` is matched. The bridge method, however, calls `A::fun(Object)` instead and prints "`A`".

Past Year Questions

These questions are provided here for discussion among yourselves (e.g., on Ed). We will not discuss these during the recitations. All questions are taken from **Midterm 2020/21 Semester 2.**

4. Consider the following generic class:

```
class Wrapper<U extends Comparable<U>> {
  U value;
}
```

After type erasure, what will the type of `value` be?

A. `Object`

B. `Comparable<U>`

C. `Comparable`

D. `Wrapper`

> **Comments:**
>
> C

5. Consider the following:

```
interface I {
}

abstract class A<T> {
}

class C extends A<Integer> implements I {
}
```

For each statement below, indicate if it compiles without any error or warning. Please provide a rationale for your answer.

(a) `I i = new A<Integer>();`

> **Comments:**
>
> Compilation error. `A` does not inherit from `I`. `A` is abstract anyway.

(b) `I i = new C();`

> **Comments:**
>
> Compiles without warning or error. `C` is a subtype of `I`.

(c) `A<String> a = new C();`

> **Comments:**
>
> Compilation Error. `C` is a subtype of `A<Integer>` and cannot be assigned to `A<String>`.

6. Consider the following classes `Main` and `SSHClient`, where:

`PasswordIncorrectException` <: `AuthenticationException` <: `Exception`

```
class Main {
  void start() {
    try {
      SSHClient client = new SSHClient();
      client.connectPENode();
    } catch (Exception e) {
      System.out.println("Main");
    }
  }
}

class SSHClient {
  void connectPENode() throws Exception {
    try {
      // Line A (Code that could throw an exception)
    } catch (AuthenticationException e) {
```

```
      System.out.println("SSHClient");
    }
  }
}
```

After calling:

```
new Main().start()
```

(a) What would be printed if an `Exception` is thrown from Line A of `connectPENode`?

> **Comments:**
>
> `Main`

(b) What would be printed if an `AuthenticationException` is thrown from Line A of `connectPENode`?

> **Comments:**
>
> `SSHClient`

(c) What would be printed if a `PasswordIncorrectException` is thrown from Line A `connectPENode`?

> **Comments:**
>
> `SSHClient`