CS2030S Recitation Problem Set 3

Brian Cheong

Recap

Covariance vs Contravariance vs Invariant

- Only related for complex types
  - example: List<Integer>, Map<String, Integer>
- Covariance
  - $S <: T \land C \text{ is a complex type} \implies C(S) <: C(T)$
- Contravariance
  - $S <: T \land C \text{ is a complex type} \implies C(T) <: C(S)$
- Invariant
  - $C(T) \text{ and } C(S)$ has no subtyping relationship
- Java complex types (generic classes) are invariant (without wildcards)

Generics

- Used to make classes more flexible
  - Don't have to write the same class over and over for different types
- few ways to declare type parameters
  - At the class declaration `class Name<T>`
  - at method level `public <T> void name()`
  - Usual scoping rules apply (the 2 T would be different)
- Bound by some object `T <: GetAreable`
  - Expose methods of the bound
- Type erasure
  - change all T to upper-bound

Exceptions

- Checked exceptions
  - Errors that can be anticipated and recovered from
  - Eg. Opening a file that may not exist
- Unchecked exceptions (runtime exceptions)
  - Errors that cannot really be recovered from and should not happen
  - Eg. Dividing by 0

Exceptions

- Checked exceptions are part of the declaration of the method (`throws` keyword)
- Tells the compiler to check that this exception is handled somewhere
- If you are a method and you invoke something that can throw an error,
  - Either you handle it (try-catch)
  - or you throw it yourself too
  - Eventually some method needs to handle it
- no need to declare that `runtime` exceptions are thrown

Question 1: background

```
class A { // SubR <: R <: SuperR | SubE <: E <: SuperE <: Exception
  R foo() throws E { ... }
}
```

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

Question 1a:

Does this code compile?

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

- Yes
  - $SubR <: R$
  - SubR can bind to R

Question 1b:

Does this code compile?

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

- No
  - $SuperR <: R$
  - SuperR cannot bind to R
  - `bar` might use methods that is in `R` but not in `SuperR`

Question 1c:

Does this code compile?

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

- Yes
  - $SubE <: E$
  - SubE can bind to E

Question 1d:

Does this code compile?

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

- No
  - $SuperE <: E$
  - SuperE cannot bind to E

Discussion points

- What is the compiler doing? Relate to a principal we know
  - Compiler is actually helping you with LSP
  - Wherever you put `A` you can put `B`
  - Ensures that the methods in `B` produce types that preserve type safety w.r.t `A`

Question 2: background

- Java has an abstract class `Number`
- `BigInteger` is a subtype of `Number` and also implements `Comparable<T>` interface

Question 2: background

- Ah Beng implemented this method using `BigInteger`

```
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;
}
```

Question 2: background

- He realised he needed to create methods for `Integer`

```
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;
}
```

Question 2: background

- and `Double`

```
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;
}
```

Question 2ai:

- Having gotten $A+$ for CS1101S he knew repeating code like this is bad so he wanted to refactor all the methods into just one

```
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;
}
```

- This doesn't work. Why would Ah Beng not get A for CS2030S?

Question 2aii:

- Realising his mistake, Ah Beng changed `Object` to `Number`

```
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;
}
```

- This doesn't work. Why would Ah Beng **still** not get A for CS2030S?

Question 2aii:

- Realising his mistake, Ah Beng changed `Number` to `Comparable`

```
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;
}
```

- This **still** doesn't work. Why? Is there any hope left for Ah Beng?

Question 2b:

- As a mugger, Ah Beng found out that type parameters can have multiple bounds
- `<T extends S1 & S2>`
- Fix his code for him so that he can get that A
- *Brian fixes code live*
- What would the type erasure be? Would it be S1 or S2?

Question 3:

We have this class `A`

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

Question 3i:

Will this compile?

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

- `B::fun(String)` appears to override `A::fun(String)`
- But after type erasure `A::fun(Object)`
- So is it overloading or overriding?

Question 3i:

- But Java is built to meet people's expectations
- we would expect it to be overriding from the outside point of view (programmers view)

Question 3i:

```
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");
  }
}
```

`B::fun(String)` overloads `B::fun(Object)`, `B::fun(Object)` overrides `A::fun(Object)`

Question 3ii:

Will this compile?

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

Question 3ii:

- This cannot work. Think of how the bridge method would look like
- There would be 2 B::fun(Object)

Question 3ii:

```
class A {
  public void fun(Object o) {
    System.out.println("A");
  }
}
class B extends A {
  public void fun(Object o) { // Bridge method
    this.fun((Object) o);
  }
  public void fun(Object i) {
    System.out.println("B");
  }
}
```

- This leads to a compile error

Question 3iii:

Does this compile?

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

Question 3iii:

- Yes. Bridging method is used again.

```
class A {
  public void fun(Object o) {
    System.out.println("A");
  }
}
class B extends A {
  public void fun(Object o) { // Bridge method
    super.fun((String) o);
  }
  public void fun(Integer i) {
    System.out.println("B");
  }
}
```

Question 3b:

i:

- `void fun(Object)` is stored during compilation
- `B::fun(Object)` would be invoked in turn invokes `B::fun(String)` which print "B"

iii:

- `void fun(Object)` is stored during compilation
- `B::fun(Object)` is invoked which invokes `A::fun(Object)` which prints "A"

Thank you

bye