CS2030S Recitation

Week 4: Problem Set 2

brian

2025-09-10

National University of Singapore

Recap

Recap: Type safety

- What is type safety?

Recap: Type safety

- What is type safety?
  - Will not have type errors

Recap: Type safety

- What is type safety?
  - Will not have type errors
  - Cannot use values in an invalid or unintended way according to their type
- How does the compiler achieve type safety?

Recap: Type safety

- What is type safety?
  - Will not have type errors
  - Cannot use values in an invalid or unintended way according to their type
- How does the compiler achieve type safety?
  - Use compile time type to determine what methods are available

Recap: Type safety

- What is type safety?
  - Will not have type errors
  - Cannot use values in an invalid or unintended way according to their type
- How does the compiler achieve type safety?
  - Use compile time type to determine what methods are available
  - Only allow assigning subtype to supertype (methods are guaranteed to be there)
    - Because the subtype (if it does not override the method) would inherit the implementation from supertype

Recap: Type safety

- What is type safety?
  - Will not have type errors
  - Cannot use values in an invalid or unintended way according to their type
- How does the compiler achieve type safety?
  - Use compile time type to determine what methods are available
  - Only allow assigning subtype to supertype (methods are guaranteed to be there)
    - Because the subtype (if it does not override the method) would inherit the implementation from supertype

Q1a

Consider the following subtyping relationship

`subR <: R <: SuperR`

`subE <: E <: SuperE <: Exception` and a class `A`

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

Let `B <: A` and `B` overrides `foo()`

Q1a

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

Which implementations of `foo` in `B` violate substitutability of A with B

(a) `SubR foo() throws E`

(b) `SuperR foo() throws E`

(c) `R foo() throws SubE`

(d) `R foo() throws SuperE`

Q1a

(a) Is ok. `SubR <: R` so it can bind to `R`

Q1a

(a) Is ok. `SubR <: R` so it can bind to `R`

(b) Not ok. `SuperR </: R`

- Let's say we allow the bind to happen. What if we do `r.g()` where `g` is found in `R` but not in `SuperR`

Q1a

(a) Is ok. `SubR <: R` so it can bind to `R`

(b) Not ok. `SuperR </: R`
  - Let's say we allow the bind to happen. What if we do `r.g()` where `g` is found in `R` but not in `SuperR`

(c) Is ok. `SubE <: E` so it can bind to `E`
  - Catching exception is like binding the exception thrown to the one declared in the catch block

Q1a

(a) Is ok. `SubR <: R` so it can bind to `R`

(b) Not ok. `SuperR </: R`

- Let's say we allow the bind to happen. What if we do `r.g()` where `g` is found in `R` but not in `SuperR`

(c) Is ok. `SubE <: E` so it can bind to `E`

- Catching exception is like binding the exception thrown to the one declared in the catch block

(d) Not ok. `SuperE <:/ E` (Similar reason to (b))

Q2 background

Java provides an abstract class called `Number`

This is the superclass of all primitive *numeric* wrapper classes

`BigInteger` is A class which supports arbitrary-precision integers (giant numbers)
`BigInteger` implements the `Comparable<T>` interface
Therefore,

- `BigInteger <: Number`
- `BigInteger <: Comparable<T>`

Q2 background

My best friend Ah Beng wrote a method to convert an array of `BigInteger` to an array of primitive `short` values

```java
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;
}
```

He realised he needed to do the same method for `Integer` and `Double`

Q2 background

So Ah Beng wrote the following code

```java
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;
}
```

Q2 background

Ah Beng scored A+ for CS1010X he realised he’s repeating code. So he wanted to generalize the methods he has written

Q2a

This was his first attempt. What's wrong? What kind of error do we get?

```java
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;
}
```

Q2a

- Compile error

Q2a

- Compile error
  - `a` has compile time type of `Object[]`

Q2a

- Compile error
  - `a` has compile time type of `Object[]`
  - `a[i]` has compile time type of `Object`

Q2a

- Compile error
  - `a` has compile time type of `Object[]`
  - `a[i]` has compile time type of `Object`
  - `Object` does not have `compareTo` or `shortValue` method

Q2a

Being a persistent student, Ah Beng tried another approach.

```java
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;
}
```

Does it work now?

Q2a

- Still compile error

Q2a

- Still compile error
  - Even though now we have access to `shortValue`
  - We don't have access to `compareTo`

Q2a

Ah Beng tries again

```java
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;
}
```

“See! I’m the best Java programmer” exclaimed Ah Beng.

Q2a

Ah Beng tries again

```java
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;
}
```

"See! I'm the best Java programmer" exclaimed Ah Beng. Why is Ah Beng not the best Java programmer and will fail CS2030S?

Q2a

- Still compile error

Q2a

- Still compile error
  - We gained access to `compareTo`
  - but lost access to `shortValue`

Q2a

- Still compile error
  - We gained access to `compareTo`
  - but lost access to `shortValue`
  - We neeed access to both

Q2a

- Still compile error
  - We gained access to `compareTo`
  - but lost access to `shortValue`
  - We neeed access to both, is all hope lost for my bestie?

Q2b

Ah Beng discovered (by being told by Brian) that Java supports generics

Q2b

Ah Beng discovered (by being told by Brian) that Java supports generics

A type parameter can have multiple bounds using the `&` symbol

Q2b

Ah Beng discovered (by being told by Brian) that Java supports generics

A type parameter can have multiple bounds using the `&` symbol

`<T extends S1 & S2>` (Only the first thing can be a class rest must be interfaces)

Q2b

Ah Beng discovered (by being told by Brian) that Java supports generics

A type parameter can have multiple bounds using the `&` symbol

`<T extends S1 & S2>` (Only the first thing can be a class rest must be interfaces)

Help Ah Beng rewrite the method with generics

Q2b

```java
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++) {
      if (a[i].compareTo(threshold) <= 0) {
        out[i] = a[i].shortValue();
      }
    }
    return out;
}
```

Q2b

```java
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++) {
      if (a[i].compareTo(threshold) <= 0) {
        out[i] = a[i].shortValue();
      }
    }
    return out;
}
```

- What happens after type erasure?

Q2b

```java
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++) {
      if (a[i].compareTo(threshold) <= 0) {
        out[i] = a[i].shortValue();
      }
    }
    return out;
}
```

- What happens after type erasure?
  - Erase to first bound
  - Cast to other bounds when need to access the method (injected by compiler)

Q3

We have `PasswordIncorrectException <: AuthenticationException <: Exception`

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

Q3

and

```java
class SSHClient {
  void connectPENode() throws Exception {
    try {
      // Line A (Code that could throw an exception)
    } catch (AuthenticationException e) {
      System.out.println("SSHClient");
    }
  }
}
```

if we run

```java
new Main().start();
```

What would be printed based on the exceptions thrown in Line A

Q3

(a) Exception
- Exception not caught in `catch` clause in `SSHClient` because `Exception </: AuthenticationException`
- It would be caught in the `catch` clause in `Main` (after stack unwinding) since subtyping is reflexive
- "Main" printed

(b) AuthenticationException
- Exception is a subtype of itself so will be caught in `catch` clause in `SSHClient`
- "SSHClient" printed

(c) PasswordIncorrectException
- Exception is a subtype of `PasswordIncorrectException` so will be caught in `catch` clause in `SSHClient`
- "SSHClient" printed

The End

bye!