

Generic and Exception

Preliminary

The following questions are not going to be discussed. Answers is given without explanation. You are encouraged to discuss them on Ed.

1. Consider the following classes.

   Code Snippet

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

   (a) What is the corresponding code after type erasure[1]?

   (b) Does the code compile? Explain your answer.

2. Consider the following exception hierarchy: `WrongPassException` <: `AuthException` <: `Exception`. Further consider the following `Main` and `SSHClient` classes.

   Main.java

   ```
   public class Main {
     void start() {
       try {
         SSHClient client = new SSHClient(); // (1)
         client.connectPENode();             // (2)
       } catch (Exception e) { // (A)
         System.out.println("Main");         // (3)
       }
     }
   }
   ```

   SSHClient.java

   ```
   public class SSHClient {
     void connectPENode() throws Exception {
       try {
         // Line A (Code that could throw an exception) // (4)
       } catch (AuthException e) { // (B)
         System.out.println("SSHClient");              // (5)
       }
     }
   }
   ```

We want to know the behavior of the program when the user is running the following code snippet.

Code Snippet

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

We will consider the different exceptions thrown from `// Line A` of `connectPENode`. For each exception, answer the following questions.

(i) Is the exception caught in (A)?
(ii) Is the exception caught in (B)?
(iii) What is the sequence of execution using the label (1) to (5)?
(iv) What is printed?

(a) Consider `Exception` is thrown in `//Line A`.
(b) Consider `AuthException` is thrown in `//Line A`.
(c) Consider `WrongPassException` is thrown in `//Line A`.

Discussion

The following questions are problem sets to be discussed during recitation. You are expected to attempt them before coming to the recitation. You may be randomly called to present your answer during recitation. You are encouraged to discuss them on Ed.

1. Consider the following subtyping relationships.
   - `SubR` <: `R` <: `SuperR`
   - `SubE` <: `E` <: `SuperE` <: `Exception` [2]

   Further consider the following implementation of class `A` with a method `foo` below. Other details in class `A` are omitted.

   Class A

   ```
   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 `bar` in `B` involving `try` [3] and `catch` [4].

   Method Bar

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

   Which of the following would violate the substitutability of `A` by `B` and thus should not be allowed? Explain your answer in the context of the program fragment below.

   (a) `SubR foo() throws E { /* .. */ }`
   (b) `SuperR foo() throws E { /* .. */ }`
   (c) `R foo() throws SubE { /* .. */ }`
   (d) `R foo() throws SuperE { /* .. */ }`

2. Java provides an abstract class called `Number` [5] that is the superclass of all primitive numeric wrapper classes. `Number` is also the superclass of `BigInteger` [6], a class that supports arbitrary-precision integers. The primitive numeric wrapper classes and the `BigInteger` class implement the `Comparable<T>` [7] interface.

   Ah Beng first wrote the following method to convert an array of `BigInteger` to an array of primitive `short` values. The method takes in a parameter called `threshold` such that any value greater than `threshold` is set to `0`.

   Initial Code

```
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 primitive `short` values. He thus duplicated his method above and replaced `BigInteger` with `Integer` and `Double` respectively. He ended up with two more methods below.

   Additional Code 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;
}
```

   Additional Code for 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;
}
```

   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 are shown below. However, they did not work correctly. What are the issues with the attempts?

Using Object

```
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;
}
```

Using 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;
}
```

Using 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;
}
```

(b) Ah Beng discovered Java supports bounded generics[8]. In particular, 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`. Note that a class bound must be specified first before an interface bound.

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

3. Consider the generic class `A` below.

A.java

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

We have 3 different attempts to create a subclass `B` of `A` below. Consider each attempt independently.

Attempt 1

```
class B extends A<String> {
  public void fun(String i) {
```

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

Attempt 2

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

Attempt 3

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

(a) For each attempts, does the code compile? Explain the reason for compilation or error in terms of overriding and overloading.

Hint: You may be interested in reading up on type erasure and bridge method[9].

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

Code Snippet

```
A<String> a = new B();
a.fun("CS2030S");
```

Hint: The bridge method (if any) has to be compiled separately. Thus, the method descriptor to be matched is fixed.

Exercise

The following questions are to be discussed on Ed as no answer will be given. ***Use only concepts that you have learnt****. You are expected to attempt them before the next lecture as the concepts may be used and/or mentioned in the lecture. The concept may also be tested for assessment.*

1. We will be using the following `Box` class as our base.

Box.java

```
public class Box<T> {
  private T val;

  public Box(T val) {
    this.val = val;
  }

  public T getVal() {
    return this.val;
  }
}
```

(a) Implement the following two methods. In each cases, determine if the method is generic or non-generic. Additionally, determine the type of the parameter.

(i) `void copyFrom(Box box)` that takes in a `box` of type `Box`. The method copies the content of the given `box` (i.e., `box.val`) to the content of the current box.

(ii) `void copyTo(Box box)` that takes in a `box` of type `Box`. The method copies the content of the current box to the content of the given `box` (i.e., `box.val`).

(a) Consider the following variable declarations.

Code Snippet

```
Box<String> box1 = /* ... */;
Box<Object> box2 = /* ... */;
```

Can your code work for the following test cases? If not, can you modify your code so that it will work for all the test cases below or explain briefly the reason it will never work?

(i) `box1.copyTo(box2);`

(ii) `box1.copyFrom(box2);`

(iii) `box2.copyTo(box1);`

(iv) `box2.copyFrom(box1);`

Bibliography

[1] Oracle, “Type Erasure.” [Online]. Available: https://docs.oracle.com/javase/tutorial/java/generics/erasure.html

[2] Oracle, “Java 25 API: Exception.” [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/lang/Exception.html

[3] Oracle, “The try Block.” [Online]. Available: https://docs.oracle.com/javase/tutorial/essential/exceptions/try.html

[4] Oracle, “The catch Block.” [Online]. Available: https://docs.oracle.com/javase/tutorial/essential/exceptions/catch.html

[5] Oracle, “Java 25 API: Number.” [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/lang/Number.html

[6] Oracle, “Java 25 API: BigInteger.” [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/math/BigInteger.html

[7] Oracle, “Java 25 API: Comparable<T>.” [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/lang/Comparable.html

[8] Oracle, “Bounded Type Parameter.” [Online]. Available: https://docs.oracle.com/javase/tutorial/java/generics/bounded.html

[9] Oracle, “Bridge Methods.” [Online]. Available: https://docs.oracle.com/javase/tutorial/java/generics/bridgeMethods.html