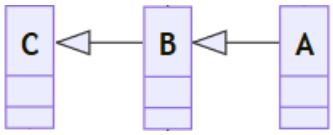
Problem Set 03 Worksheet

0 Exploration

Attempt the following on your own and try to explain the result.

0.a Method

We consider the following subtype relationship: `A <: B <: C`. The *simplified* class diagram is shown on the right. In the simplified version, we only care about the subtyping relationship but not the details of the class.



For each questions below, try to come up with a test code on your own. You can then check using Java compiler for the answer.

0.a.1 Fixed Method Descriptors

Consider the following method descriptor: `B foo(B)`. Select all the method invocation that does not cause any compilation error.

0.a.1.1 Parameter Type

```
a) foo(new A());
b) foo(new B());
c) foo(new C());
```

0.a.1.2 Return Type

```
d) A res = foo(new B());
e) B res = foo(new B());
f) C res = foo(new B());
```

0.a.2 Fixed Method Invocation

Consider the following method invocation: `B res = foo(new B());`. Select all the method descriptor that does not cause any compilation error.

0.a.2.1 Parameter Type

```
a) B foo(A a) { ... }
b) B foo(B b) { ... }
c) B foo(C c) { ... }
```

0.a.2.2 Return Type

```
d) A foo(B b) { ... }
e) B foo(B b) { ... }
f) C foo(B b) { ... }
```

0.a.3 PECS

Can you relate the two exercises above to the PECS principle? Connect the concepts below (*i.e., draw a line from the left to the corresponding concepts on the right*):

- ☐ *Producer*
- ☐ *Consumer*
- ☐ *Return Type and/or Value*
- ☐ *Parameter Type and/or Argument Value*

0.b Try-Catch-Finally

You may have learnt that the `finally`-clause is *always* executed. This may lead to quite a counter-intuitive result. Try running the following methods and try to understand the result. Is the `finally`-clause always executed in these cases?

```
void f(int x) throws Exception {
  if (x == 0) {
    throw new Exception();
  }
  System.out.println("inside f");
}
```

```
public static void main(String[] args) {
  System.out.println(g_(0));
  System.out.println(g_(1));
}
# Change g_ to g1 or g2
```

```
int g1(int x) {
  try {
    System.out.println("before f");
    f(x);
    System.out.println("after f");
    return 1;
  } catch(Exception e) {
    System.out.println("f error");
    return 2;
  } finally {
    System.out.println("before return?");
  }
}
```

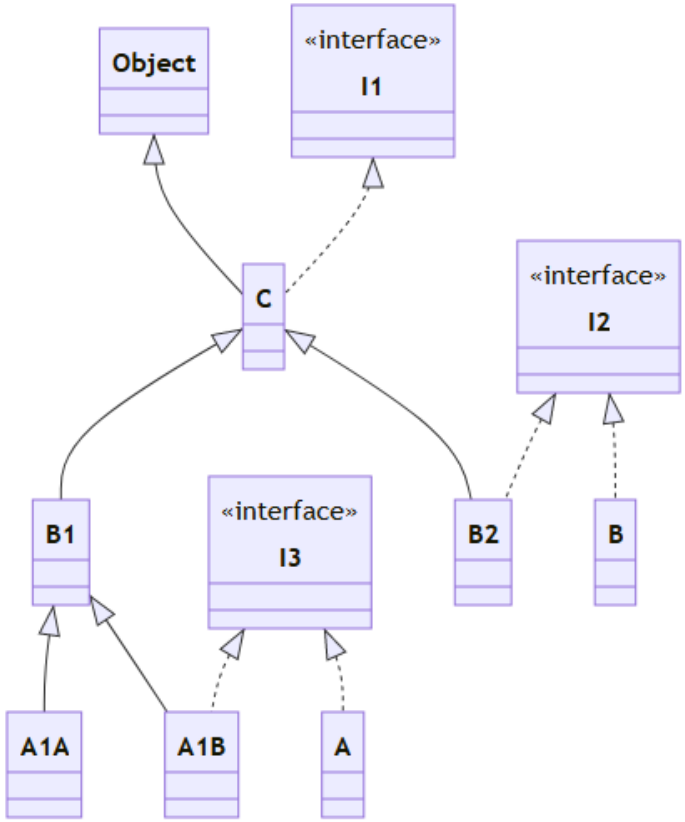
```
int g2(int x) {
  try {
    System.out.println("before f");
    f(x);
    System.out.println("after f");
    return 1;
  } catch(Exception e) {
    System.out.println("f error");
    return 2;
  } finally {
    return 3;
  }
}
```

0.c Bounded Generic

Consider the class diagram on the right. We define the following generic type:

```
class Generic<T extends □> { }
```

We will replace □ with some known types shown on the right. Your job is to determine the classes that can be used to instantiate `T`. If there is none, simply write the empty set ∅.



0.c.1 Simple Bound

Value of □	Allowed Value of T
`<T extends C>`	
`<T extends B1>`	
`<T extends B2>`	
`<T extends I2>`	
`<T extends I3>`	

0.c.2 Intersection Bound

Value of □	Allowed Value of T
`<T extends B1 & I2>`	
`<T extends C & I2>`	
`<T extends B1 & I3>`	

0.d Type Erasure

For each of the following generic class, write the code after type erasure.

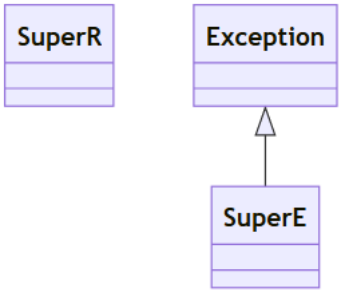
Before Type Erasure	After Type Erasure

`class C<T> {`
`  T t;`
`  C(T t) {`
`  this.t = t;`
`  }`
`  T getT() {`
`  return this.t;`
`  }`
`}`	

`class C<T extends Point> {`
`  T t;`
`  C(T t) {`
`  this.t = t;`
`  }`
`  T getT() {`
`  return this.t;`
`  }`
`}`	

`class C<S, T extends S> {`
`  S s;`
`  T t;`
`}`	

1 Question 1



Draw the *simplified* class diagram for the following subtyping relationship:

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

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

      :

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

Do the exercises in Exploration, especially 0.a.1 and 0.a.2 with the following cases:

- Fixed method descriptor: `R foo()`
- Fixed method invocation: `R r = a.foo();`

For now, ignore the exception (*i.e., assume there is no exception in the method*).

Now we consider only the exception. Select all exceptions that can be caught by the `catch`-clause where ___ indicate a *don't care* information (*i.e., the value can be anything as long as it is correct and not the cause of compilation error*).

```
a) ___ foo() throws Exception { ... }
b) ___ foo() throws SuperE    { ... }
c) ___ foo() throws E         { ... }
d) ___ foo() throws SubE      { ... }
```

The idea is now to mix-and-match the information:

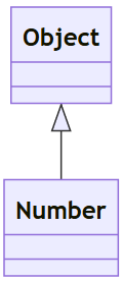
- Find all allowed parameter type (*in this case, nothing*)
- Find all allowed return type
- Find all exceptions

The combination of all of these should be safe to compile. A different way to check is to pass in an argument of type `B` to `bar`.

Method Definition	**Notes**
`SubR foo() throws E { ... }`	
`SuperR foo() throws E { ... }`	
`R foo() throws SubE { ... }`	
`R foo() throws SuperE { ... }`	

2 Question 2

To start with, draw the class diagram involving `Number`, `Comparable<T>`, `Integer`, `Double`, and `BigInteger` in relation to the `short shortValue()` and `int compareTo(T o)` method. In other words, you can ignore other classes/interfaces as well as other methods. However, if the method is non-generic, you need to write the non-generic method.



Consider the call x.compareTo(y). Complete the table below to show output of the method call. Simply write *-ve*, *0*, and *+ve* to indicate negative value, zero, and positive value respectively.

Ordering	`x.compareTo(y)` **result**
`x < y`	
`x == y`	
`x > y`	

2.a Question 2a

Determine the compile-time type of the expression `a[i]` for each code below.

Code	**CTT of `a[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;`
`}`	

`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;`
`}`	

`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;`
`}`	

Using the class diagram above, determine the methods available for each of the compile-time type of `a[i]`. Can you now explain the error message that you get?

2.b Question 2b

Let us invert the question from Q2a. Consider the following expressions below, write down the *most general type* (*i.e., the opposite of the most specific type*) that contains the *method descriptor* for the given method (*e.g., need not be the actual method definition*).

Expression	Most General Type
`a[i].compareTo(threshold)`	
`a[i].shortValue()`	

Given the most general type above, can you determine the possible generic type for the static method `toShortArray`? *Note: Using* `<T>` *alone does not work. Can you figure out why?*

3 Question 3

Note that when a generic type is created (*e.g.,* `class Generic<T> { }`), the type `T` is only known when the type is being parameterized. Multiple different `T` can be specified for different instance. For each of the cases below, first determine the code after type erasure. Then, determine if there are any compile time error or not using the type erasure and other information.

3.a Question 3a

Before Type Erasure	After Type Erasure

`import java.util.List;`
`class A {`
`  void foo(List<Integer> integerList) {}`
`  void foo(List<String> stringList) {}`
`}`	

Is the compilation successful or not? Explain.

3.b Question 3a

Before Type Erasure	After Type Erasure

`class B<T> {`
`  T x;`
`  static T y;`
`}`	

Is the compilation successful or not? Explain.

3.c Question 3a

Before Type Erasure	After Type Erasure

`class C<T> {`
`  static int b = 0;`
`  C() {`
`  this.b++;`
`  }`
`  public static void main(String[] args) {`
`  C<Integer> x = new C<>();`
`  C<String> y = new C<>();`
`  System.out.println(x.b);`
`  System.out.println(y.b);`
`  }`
`}`	

Is the compilation successful or not? Explain.