FACULTY OF INFORMATICS

ADVANCED JAVA PROGRAMMING

Assignment 2

Author FEDERICO LAGRASTA

CLAUDIO MAGGIONI

October 23, 2022

A

1

The snippet would cause a compile time error at line 4 since it is not possible to call `get(...)` on an lower bounded `List` (i.e. a variable with `List<? super SOMETHING>` static type, where `SOMETHING` can be any class), in this case on `src`. This is due to the get-put principle. If the code snippet were legal, the following example would have been legal but not be type safe:

```
List<Object> objects = new ArrayList<>();
objects.put(new Object());

List<? super String> strings = objects;

// not type safe, as strings[0], i.e. objects[0] is an Object and not a String
String s = strings.get(0);
```

Listing 1: Type unsafe example of covariant access on a `List` with a lower bounded wildcard.

Additionally, the snippet would cause another compile time error at line 5, since it is also not possible to call `set(...)` on an upper bounded `List` (i.e. a `List<? extends SOMETHING>`). The following example would not be type safe otherwise:

```
List<Integer> ints = new ArrayList<>();
ints.put(42);

List<? extends Number> numbers = ints;
numbers.set(0, 50.0);

// not type safe, as numbers[0], i.e. ints[0] is a Double and not an Integer
Integer i = ints.get(0);
```

Listing 2: Type unsafe example of contravariant access on a `List` with an upper bounded wildcard.

2

The code does not compile.

B

1

No, as the compiler will trust our cast. However, an "unchecked cast" compiler warning will be reported.

2

Yes, as arrays can not be downcasted. Specifically, a `ClassCastException` will be thrown at line 8, where the `Object[]` instance would be needed to be downcasted to `String[]`. Note that the exception is not thrown inside `myArrayGenerator(...)` as `T` is erased to `Object` during compliation, making the explicit cast redundant, but in turn making the implicit cast added by the use of generics illegal.

C

1

The compiler would report 3 compile time errors (at lines 3, 5 and 7) as generic type argument `T` cannot be referenced in static contexts, i.e. in static field declarations, static method signatures or static method bodies. This compiler rule is a side effect of erasure.

2

The code does not compile.

D

1

The code will not compile, as the two methods included in the class have the same signature after erasure, since `Class<?>` is erased to the type `Class` and the types `T` and `U` will be erased to `Object`. Therefore, the class after erasure would look like this:

```
public class Couple {
  public Class getType(Object t) {
    return t.getClass();
  }

  public Class getType(Object u) {
    return u.getClass();
  }
}
```

Listing 3: Class `Couple<T, U>` after erasure.

thus making the two methods have an identical signature, which is illegal.

2

The code does not compile.

E

1

It will print `I am a class` since the keyword `super` prioritizes the parent class over interfaces. The implemented interface would have to be referred as `Second.super` (i.e. changing line 17 to `Second.super.doSomething();` would result in the `I am an interface` output).

2

It won't compile. `First.doSomething()` would be inherited by the child class `Third` over `Second.doSomething()` since parent classes have priority interfaces with default methods w.r.t. interfaces. However, `First.doSomething()` has default or "package-private" visibility while the interface `Second` requires `doSomething()` to be public (as method signatures in interfaces are by default public). Therefore, a compile-time error is reported due to assigning "weaker access privilegies" to the `doSomething()` method than required by interface `Second`.