EECS2030 (Sec. B & G) Fall 2025
Advanced Object Oriented Programming
Example Exam Questions: Solution

Caveat: These questions are **just examples** and **not** meant to be complete.
You should prioritize your time in studying all the covered materials.

1. Assume that a `Person` class is already defined, and it has an attribute `name` and a constructor that initializes the person's name from the input string. Consider the following fragment of Java code (inside some `main` method):

```
1 Person p1 = new Person("Heeyeon");
2 Person p2 = new Person("Jiyoon");
3 System.out.println(p1 != p2);
```

What happens when executing the above Java code?

Solution: One line output to the console:

```
true
```

2. Assume that a `Person` class is already defined, and it has an attribute `name` and a constructor that initializes the person's name from the input string. Consider the following fragment of Java code (inside some `main` method):

```
1 Person p1 = new Person("Heeyeon");
2 Person p2 = new Person("Jiyoon");
3 Person[] persons = new Person[2];
4 System.out.println(persons[persons.length()] != null);
```

What happens when executing the above Java code?

Solution: The above Java code does not compile at Line 4: `length()` is not a method defined in primitive arrays (instead, write `persons.length`).

3. Assume that a `Person` class is already defined, and it has an attribute `name` and a constructor that initializes the person's name from the input string. Consider the following fragment of Java code (inside some `main` method):

```
1 Person p1 = new Person("Heeyeon");
2 Person p2 = new Person("Jiyoon");
3 Person[] persons = new Person[2];
4 System.out.println(persons[persons.length] != null);
```

What happens when executing the above Java code?

Solution: An `ArrayIndexOutOfBoundsException` occurs from Line 4, as the maximum index for the `persons` array is `persons.length - 1`.

4. Assume that a `Person` class is already defined, and it has an attribute `name` and a constructor that initializes the person's name from the input string. Consider the following fragment of Java code (inside some `main` method):

```
1 Person p1 = new Person("Heeyeon");
2 Person p2 = new Person("Jiyoon");
3 Person[] persons = new Person[2];
4 System.out.println(persons[persons.length - 1] != null);
```

What happens when executing the above Java code?

Solution: One line output to the console:

```
false
```

5. Assume that a `Person` class is already defined, and it has an attribute `name` and a constructor that initializes the person's name from the input string. Consider the following fragment of Java code (inside some `main` method):

```
1 Person p1 = new Person("Heeyeon");
2 Person p2 = new Person("Jiyoon");
3 Person[] persons = new Person[2];
4 System.out.println(persons[persons.length - 1].name.equals("Jiyoon"));
```

What happens when executing the above Java code?

Solution: A `NullPointerException` occurs at Line 4, as `persons[0]` and `persons[1]` are both storing `null`.

6. Assume that a `Person` class is already defined, and it has an attribute `name` and a constructor that initializes the person's name from the input string. Consider the following fragment of Java code (inside some `main` method):

```
1 Person p1 = new Person("Heeyeon");
2 Person p2 = new Person("Jiyoon");
3 Person[] persons = {p1, p2};
4 p1 = p2;
5 System.out.println(persons[0] == p1);
```

What happens when executing the above Java code?

Solution: One line output to the console:

```
false
```

7. Assume that a `Person` class is already defined, and it has an attribute `name` and a constructor that initializes the person's name from the input string. Consider the following fragment of Java code (inside some `main` method):

```
1 Person p1 = new Person("Heeyeon");
2 Person p2 = new Person("Jiyoon");
3 Person[] persons = {p1, p2};
4 p1 = p2;
5 persons[0] = p2;
6 System.out.println(persons[0] == p1);
```

What happens when executing the above Java code?

Solution: One line output to the console:

```
true
```

8. Assume that a `Person` class is already defined, and it has an attribute `name`, a constructor that initializes the person's name from the input string, and a mutator method `setName` that changes the person's name from the input string. Consider the following fragment of Java code (inside some `main` method):

```
1 Person p1 = new Person("Heeyeon");
2 Person p2 = new Person("Jiyoon");
3 Person[] persons = {p1, p2};
4 p1 = persons[1];
5 persons[0] = p2;
6 p2.setName("Jihye");
7 System.out.println(p1.name);
```

What happens when executing the above Java code?

Solution: One line output to the console:

```
Jihye
```

9. Consider the following classes, where we use `print` to abbreviate `System.out.println`:

```
class A extends B {
  A() { }
}
```

```
class B extends C {
  B() { }
}
```

```
class C {
  C() { }
  void bm(){print("C.bm");}
}
```

```
class D extends C {
  D() { }
  void cm(){print("D.cm");}
}
```

```
class F extends D {
  F() { }
  void bm(){print("F.bm");}
  void em(){print("F.em");}
}
```

```
class E extends F {
  E() { }
  void dm(){print("E.dm");}
}
```

Now consider the following code in the `main` method of a tester class for the above classes:

```
1  D d1 = new C();
2  C d2 = new D();
3  d2.bm();
4  D e1 = new E();
5  d2 = e1;
6  d2.bm();
7  F f = e1;
8  e1.em();
9  B b1 = (A) d2;
```

(a) Explain if **Line 1** compiles.

> **Solution:** No. The new dynamic type `C` cannot fulfill the expectation of `d1`'s static type `D`, because `C` is not a descendant class of `D`.

(b) Explain if **Line 2** compiles.

> **Solution:** Yes. The new dynamic type `D` can fulfill the expectation of `d2`'s static type `C`, because `D` is a descendant class of `C`.

(c) Explain if **Line 3** compiles. If yes, write down and explain how the output is printed. When tracing, consider only the earlier lines that compile.

> **Solution:**
> Yes, because method `bm` is declared in `d2`'s static type `C`.
> The dynamic type of `d2` is `D`, which means the version of method `bm` inherited from class `C` is called: output is `C.bm`.

(d) Explain if **Line 5** compiles. If yes, what are the static type and dynamic type of `d2` after `Line 5` is executed? When tracing, consider only the earlier lines that compile.

> **Solution:**
> Yes. `e1`'s static type `D` can fullfil `d2`'s static type `C`, because `D` is a descendant class of `C`.
> The static type of `d2` remains as `C`.
> The dynamic type of `d2` changes to `E` (i.e., the dynamic type of `e1`).

(e) Explain if **Line 6** compiles. If yes, write down and explain the output. When tracing, consider only the earlier lines that compile.

> **Solution:**
> Yes, because method `bm` is declared in `d2`'s static type `C`.
> The dynamic type of `d2` is `E`, which means the redefined/overridden version of method `bm` inherited from class `F` is called: output is `F.bm`.

(f) Explain if **Line 7** compiles.

> **Solution:** No. `e1`'s static type `D` cannot fullfil `f`'s static type `F`, because `D` is not a descendant class of `F`.

(g) Explain if **Line 8** compiles. If yes, write down and explain the output. If no, suggest a fix using type casting, then write down and explain how the output is printed. When tracing, consider only the earlier lines that compile.

> **Solution:**
> No, because `e1`'s static type `D` does not have the method `em` declared.
> We can use a type cast: `((E) e1).em()`, which performs a downward cast on `e1` and creates an alias of static type E, so we can call the method `em`.
> Since e1's dynamic type is `E`, the version of the method `em` in class E, which is inherited from class `F`, will be called: output is `F.em`.

(h) Explain why **Line 9** compiles.
But **Line 9** is problematic at runtime. Explain why and how we can extend the code to avoid it. When tracing, consider only the earlier lines that compile.

> **Solution:**
> - The static type of `d2` is `C`, so `(A) d2` is a valid downward cast, given that `A` is a descendent of `C`. The type cast then creates an alias of static type `A`, which can fullfil the expectation on the static type of `b1` (which is `B`), because `A` is a descendent of `B`.
> - A `ClassCastException` will occur at runtime, because the dynamic type of `d2` (which is `E`) cannot fullfil the expectation of the cast type `A`, given that `E` is not a descendent class of `A`.
> - To avoid this, do a runtime check:
>
> ```
> if (d2 instanceof A) {
>   B b1 = (A) d2;
> }
> ```

10. Consider the following classes, where we use `print` to abbreviate `System.out.println`:

```
interface I {
  void mi();
}
```

```
class A implements I {
  void mi() {
    println("A.mi"); }
}
```

```
class B implements I {
  void mi() {
    println("B.mi"); }
}
```

```
1  class Collector {
2    A[] as; int numberOfAs;
3    B[] bs; int numberOfBs;
4    Collector() {
5      as = new A[10]; bs = new B[10]; }
6    void addA(A a) {
7      as[numberOfAs] = a; numberOfAs++; }
8    void addB(B b) {
9      bs[numberOfBs] = b; numberOfBs++; }
10   void callAll() {
11     for(int i = 0; i < numberOfAs; i ++)
12     { as[i].mi(); }
13     for(int i = 0; i < numberOfBs; i ++)
14     { bs[i].mi(); }
15   }
16 }
```

```
1  class Tester {
2    static void main(String[] args) {
3      I i = new I();
4      B b = new B(); A a = new A();
5      Collector c = new Collector();
6      c.addB(b); c.addA(a);
7      c.callAll();
8    }
9  }
```

(a) Explain if the assignment `as[numberOfAs] = a` in **Line 7** of the above `Collector` class compiles.

Solution:
Yes, because `a`'s static type `A` is a descendant class of `as[i]`'s static type `A`.

(b) Explain if the method call `as[i].mi()` in **Line 12** of the above `Collector` class compiles.

Solution:
Yes, because `as[i]`'s static type `A` has the method `mi` defined.

(c) Explain if **Line 3** of the above `Tester` class compiles.

Solution:
No, because `I` being an interface cannot be used as a dynamic type.

(d) Write and Explain the console output from **Line 7** of the above `Tester` class.

Solution:
Output is:

```
A.mi
B.mi
```

- The first `for` loop in method `callAll` will call the version of method `mi` implemented in class `A`. So the output is: `"A.mi"`.
- The second `for` loop in method `callAll` will call the version of method `mi` implemented in class `B`. So the output is: `"B.mi"`.

(e) The above `Collector` class does not make use of *polymorphism*, which results from the fact that classes `A` and `B` implement a common interface `I`. Rewrite the above `Collector` class, such that there is only one array attribute and one `add` method, and that the `callAll` method contains just a single loop.

Solution:

```
1  class Collector {
2    I[] is; int numberOfIs;
3    Collector() {
4      is = new I[10]; }
5    void addI(I i) {
6      is[numberOfIs] = i; numberOfIs++; }
7    void callAll() {
8      for(int i = 0; i < numberOfIs; i ++)
9      { is[i].mi(); }
10   }
11 }
```