Lab 4 Notes

CS 61BL
Summer 2019

Lab 4
June 28, 2019

Name:

SID:

Section Number: 01 02 03 04 05 06 07 08 09 10 11 12

Write your name and login above. Please complete this worksheet during your lab, and turn it in to your TA by the end of your section. You are encouraged to work with your partners and neighbors collaboratively.

1 Fish and Salmon (optional)

1.1 Look at `Fish.java` and `Salmon.java`. You'll see that each have constructors, and that they are labeled as Constructors A, B, C, D, E.

Fill in the table below to indicate whether or not each combination compiles. Put a "Y" if that combination compiles, and "N" if that combination fails to compile. The rows indicate which constructors should be included for that scenario from `Fish.java` and the columns indicate which constructors should be included from `Salmon.java`. You can check your answers by commenting out all other constructors.

	None	C	D	E
None	Y	Y	Y	N/A
A	Y	Y	Y	N/A
B	N	N	N	Y
A and B	Y	Y	Y	Y

1.2 Choose a combination of constructors that compiles and comment the rest out. Finally, in the `main()` method of `Main.java`, add lines A and B from the lab spec.

(a) What happens / what does IntelliJ tell you?

does not work, cannot be applied to given types

(b) Why do you think this is?

fs is statically declared as a Fish, function wants a Salmon

(are all Fish Salmons? no)

1.3 Look at the `swim` functions provided in `Fish.java` and `Salmon.java`. Cross out any lines that would cause a compile-time error. Remember that we examine the static type at compile time.

```
1  // If any of these four lines don't compile,
2  // also cross out any line farther down that uses that instance.
3  Fish fish = new Fish();
4  Salmon salmon = new Salmon();
5  Fish bob = new Salmon();
6  Salmon tuna = new Fish();
7
8  fish.swim();
9  salmon.swim();
10 bob.swim();
11 tuna.swim();
12
13 fish.swim(5);
14 salmon.swim(5);
15 bob.swim(5);
16 tuna.swim(5);
```

Salmon subclass of Fish

↳ Fish does not have a swim method that takes in an int speed.

1.4 Now, write next to each line what is printed by each method call. If in the previous part you determined a line would not compile, you should ignore it here (you can just cross it out again). Remember that we examine the dynamic type at runtime.

```
1  fish.swim();
2
3  salmon.swim();
4
5  bob.swim();
6
7  tuna.swim();
8
9
10 fish.swim(5);
11
12 salmon.swim(5);
13
14 bob.swim(5);
15
16 tuna.swim(5);
```

splash

splish splash

splish splash

swimming at 5 mph

2 Identify Static Types

2.1 Consider this piece of code:

```
1 Point p;
```

What is the static type of `p`? (choose one)

a. Point

b. Object

c. Can't tell.

2.2 Suppose you define the following method inside the 'TracedPoint' class:

```
1 public static void printPoint(Point p) {
2     System.out.println(p);
3 }
```

Then you run the following code

```
1 TracedPoint tp = new TracedPoint(2, 3);
2 printPoint(tp);
```

What is the static type of the variable **inside the method** `printPoint`? (choose one)

a. Point

b. TracedPoint

c. Object

2.3 Assume that `TracedPoint extends Point`. Suppose you define the following method inside the `TracedPoint` class:

```
1 public void printX(){
2     System.out.println(this.x);
3 }
```

What is the static type of `this` inside the method? (choose one)

a. Point

b. TracedPoint

c. Can't tell

this, inside TP ⇒ type of that class

3 Playing with Puppers

3.1 Suppose we have the Dog and Corgi classes which are a defined below with a few methods but no implementation shown. (modified from Spring '16, MT1)

```
1  public class Dog {
2      public Dog(){ /* D1 */ }
3      public void bark(Dog d) { /* Method A */ }
4  }
5  public class Corgi extends Dog {
6      public Corgi(){ /* C1 */ }
7      public void bark(Corgi c) { /* Method B */ }
8      @Override
9      public void bark(Dog d) { /* Method C */ }
10     public void play(Dog d) { /* Method D */ }
11     public void play(Corgi c) { /* Method E */ }
12 }
```

statically sees this

uses this ~only override

For the following main method, at each call to play or bark, tell us what happens at **runtime** by selecting which method is run. Circle if there is a compiler error or runtime error if they occur.

```
1  public static void main(String[] args) {
2      Dog d = new Corgi();           Compile-Error  Runtime-Error  C1  D1
3      Corgi c = new Corgi();         Compile-Error  Runtime-Error  C1  D1
4      Dog d2 = new Dog();            Compile-Error  Runtime-Error  C1  D1
5      Corgi c2 = new Dog();          Compile-Error  Runtime-Error  C1  D1
6      Corgi c3 = (Corgi) new Dog(); Compile-Error  Runtime-Error  C1  D1
7      d.play(d);          Compile-Error  Runtime-Error  A  B  C  D  E
8      d.play(c);          Compile-Error  Runtime-Error  A  B  C  D  E
9      c.play(d);          Compile-Error  Runtime-Error  A  B  C  D  E
10     c.play(c);          Compile-Error  Runtime-Error  A  B  C  D  E
11     c.bark(d);          Compile-Error  Runtime-Error  A  B  C  D  E
12     c.bark(c);          Compile-Error  Runtime-Error  A  B  C  D  E
13     d.bark(d);          Compile-Error  Runtime-Error  A  B  C  D  E
14     d.bark(c);          Compile-Error  Runtime-Error  A  B  C  D  E
15     d.bark((int)c);     Compile-Error  Runtime-Error  A  B  C  D  E
16     c.bark((Corgi)d2); Compile-Error  Runtime-Error  A  B  C  D  E
17     ((Corgi)d).bark(c);Compile-Error  Runtime-Error  A  B  C  D  E
18     ((Dog) c).bark(c); Compile-Error  Runtime-Error  A  B  C  D  E
19     c.bark((Dog) c);   Compile-Error  Runtime-Error  A  B  C  D  E
20 }
```

Dog does not have play()

implicit call to superclass constructor

Compiles but we casted wrong !!

corgi play(), fit to arg type

} dynamic: corgi & fit to arg type

see above

d2 not a Corgi; d is actually a corgi same reason as green