CS 61B Spring 2021

Inheritance

Exam Prep Discussion 4: February 8, 2021

1 Athletes

Suppose we have the Person, Athlete, and SoccerPlayer classes defined below.

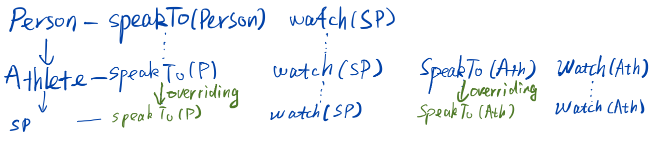
```java
class Person {
    void speakTo(Person other) { System.out.println("kudos"); }
    void watch(SoccerPlayer other) { System.out.println("wow"); }
}

class Athlete extends Person {
    void speakTo(Athlete other) { System.out.println("take notes"); }
    void watch(Athlete other) { System.out.println("game on"); }
}

class SoccerPlayer extends Athlete {
    void speakTo(Athlete other) { System.out.println("respect"); }
    void speakTo(Person other) { System.out.println("hmph"); }
}
```

(a) For each line below, write what, if anything, is printed after its execution. Write CE if there is a compiler error and RE if there is a runtime error. If a line errors, continue executing the rest of the lines.

```java
Person itai = new Person(); ✓

~~SoccerPlayer shivani = new Person()~~; CE

Athlete sohum = new SoccerPlayer(); ✓

Person jack = new Athlete(); ✓

Athlete anjali = new Athlete(); ✓

SoccerPlayer chirasree = new SoccerPlayer(); ✓

itai.watch(chirasree);        P -> watch    "wow"

jack.watch(sohum);            P -> watch    CE

itai.speakTo(sohum);          P -> speakTo  "kudos"

jack.speakTo(anjali);         P -> speakTo  "kudos"
```

P
↓
Ath
↓
SP

	dynamic	static
itai	Person	Person
~~shivani~~	~~Person~~	~~SP~~
sohum	SP	Ath
jack	Ath	Person
anjali	Ath	Ath
chirasree	SP	SP



```
anjali.speakTo(chirasree); Ath→ speakTo (Ath) "take notes"

sohum.speakTo(itai);  Ath→ speakTo (P) → SP→speakTo(P) "hmph"

chirasree.speakTo((SoccerPlayer) sohum); SP→speakTo(Ath) "respect"

sohum.watch(itai);  Ath→ watch(P)  CE    No fix

sohum.watch((Athlete) itai); Ath→ wath(Ath)  RE  No fixing

((Athlete) jack).speakTo(anjali); Ath→speakTo(Ath) "take notes"

((SoccerPlayer) jack).speakTo(chirasree); RE   Removing casting can fix

((Person) chirasree).speakTo(itai); Person→ speakTo → SP→speakTo  "hmph"
```

(b) You may have noticed that jack.watch(sohum) produces a compile error. Interestingly, we can resolve this error by **adding casting**! List two fixes that would resolve this error. The first fix should print wow. The second fix should print game on. Each fix may cast either jack or sohum.

1. jack.watch((SoccerPlayer) sohum) ;
2. ((Athlete) jack). watch (sohum);

	Dynamic	Static
jack	Ath	Person
Sohum	SP	Ath

Person.watch (Ath)

(c) Now let's try resolving as many of the remaining errors from above by **adding or removing casting**! For each error that can be resolved with casting, write the modified function call below. Note that you cannot resolve a compile error by creating a runtime error! Also note that not all, or any, of the errors may be resolved.

2 Dynamic Method Selection

Modify the code below so that the max method of DMSList works properly. Assume all numbers inserted into DMSList are positive, and we only insert using `insertFront`. You may not change anything in the given code. You may only fill in blanks. You may not need all blanks. (Spring '16, MT1)

```
public class DMSList {
    private IntNode sentinel;
    public DMSList() {
        sentinel = new IntNode(-1000, ____________________);
                                      new IntNodeEnd()
    }
    public class IntNode {
        public int item;
        public IntNode next;
        public IntNode(int i, IntNode h) {
            item = i;
            next = h;
        }                                   Problem is "next" might be null
        public int max() {
            return Math.max(item, next.max());
        }
    }
    public class IntNodeEnd extends IntNode {

    public IntNodeEnd(){

        super(0, null);

    }

    public int max() {

        return item;

    ____________________________________________________________

    ____________________________________________________________

    }
    }
    /* Returns 0 if list is empty. Otherwise, returns the max element. */
    public int max() {
        return sentinel.next.max();
    }
    public void insertFront(int x) { sentinel.next = new IntNode(x, sentinel.next); }
}
```

3 Challenge: A Puzzle

Consider the **partially** filled classes for A and B as defined below:

```
public class A {
    public static void main(String[] args) {
        A y = new B();
        B z = new B();
    }

    int fish(A other) {
        return 1;
    }

    int fish(B other) {
        return 2;
    }
}

class B extends A {
    @Override
    int fish(B other) {
        return 3;
    }
}
```

A.fish(A) = 1

A.fish(B) = 2

B.fish(A) = 1

B.fish(B) = 3

Note that the only missing pieces of the classes above are static/dynamic types! Fill in the **four** blanks with the appropriate static/dynamic type — A or B — such that the following are true:

1. y.fish(z) equals z.fish(z)
2. z.fish(y) equals y.fish(y)
3. z.fish(z) does not equal y.fish(y)