

## 1 Athletes

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

```
1 class Person {
2     void watch(Athlete other) { System.out.println("wow"); }
3     void speakTo(Athlete other) { System.out.println("kudos"); }
4 }
5
6 class Athlete extends Person {
7     void speakTo(Person other) { System.out.println("take notes"); }
8     void play(Athlete other) { System.out.println("game on"); }
9 }
10
11 class SoccerPlayer extends Athlete {
12     void speakTo(SoccerPlayer other) { System.out.println("howdy"); }
13     void speakTo(Person other) { System.out.println("hmph"); }
14 }
```

- (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.

```
1 Person itai = new Person();
2
3 SoccerPlayer shivani = new Person();
4
5 Athlete sohum = new SoccerPlayer();
6
7 Person jack = new Athlete();
8
9 Athlete anjali = new Athlete();
10
11 SoccerPlayer chirasree = new SoccerPlayer();
12
13 itai.watch(chirasree);
14
15 chirasree.watch(itai);
16
17 jack.speakTo(anjali);
18
19 anjali.speakTo(sohum); // this one is really tricky!
20
```

2 *Inheritance*

```
21 sohum.speakTo(jack);
22
23 jack.play(chirasree);
24
25 ((SoccerPlayer) jack).speakTo(chirasree);
26
27 ((SoccerPlayer) sohum).speakTo(itai);
28
29 chirasree.speakTo((SoccerPlayer) sohum);
30
31 ((Person) chirasree).speakTo(itai);
```

- (b) Some of the errors above can be resolved by **adding/removing casting**. For this part, resolve those errors, and write the modified function calls below. If multiple casts would suffice, choose the most permissive one. Note: you cannot resolve a compile error by creating a runtime error!

## 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)

```

1  public class DMSList {
2      private IntNode sentinel;
3      public DMSList() {
4          sentinel = new IntNode(-1000, _____);
5      }
6      public class IntNode {
7          public int item;
8          public IntNode next;
9          public IntNode(int i, IntNode h) {
10             item = i;
11             next = h;
12         }
13         public int max() {
14             return Math.max(item, next.max());
15         }
16     }
17     public _____ {
18
19     _____
20
21     _____
22
23     _____
24
25     _____
26
27     _____
28
29     _____
30
31     _____
32
33     _____
34     }
35     /* Returns 0 if list is empty. Otherwise, returns the max element. */
36     public int max() {
37         return sentinel.next.max();
38     }
39     public void insertFront(int x) { sentinel.next = new IntNode(x, sentinel.next); }
40 }

```

### 3 Challenge: A Puzzle

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

```

1  public class A {
2      public static void main(String[] args) {
3          ___ y = new ___();
4          ___ z = new ___();
5      }
6
7      int fish(A other) {
8          return 1;
9      }
10
11     int fish(B other) {
12         return 2;
13     }
14 }
15
16 class B extends A {
17     @Override
18     int fish(B other) {
19         return 3;
20     }
21 }
```

Note that the only missing pieces of the classes above are static/dynamic types! Fill in the **five** 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)`