

## 1 Playing with Puppies

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

```

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

```

overload

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

```

1 public static void main(String[] args) {
2     Dog d = new Corgi();
3     Corgi c = new Corgi();
4
5     d.play(d);    Compile-Error  Runtime-Error  A  B  C  D  E
6     d.play(c);    Compile-Error  Runtime-Error  A  B  C  D  E
7     c.play(d);    Compile-Error  Runtime-Error  A  B  C  D  E
8     c.play(c);    Compile-Error  Runtime-Error  A  B  C  D  E
9
10    c.bark(d);     Compile-Error  Runtime-Error  A  B  C  D  E
11    c.bark(c);     Compile-Error  Runtime-Error  A  B  C  D  E
12    d.bark(d);     Compile-Error  Runtime-Error  A  B  C  D  E
13    d.bark(c);     Compile-Error  Runtime-Error  A  B  C  D  E
14 }

```

I am not sure

the inspiration can be drawn from `d.play`; the compiler will firstly check whether the methods exist in static type's class, if not, compilation error; otherwise, it then uses dynamic method selection during runtime, **by checking whether there is an alternative overriding method in the subclass**. It will not directly do into the subclass actually pointed at, and check all its methods.

## 2 Cast the Line

Suppose Cat and Dog are two subclasses of the Animal class and the Tree class is unrelated to the Animal hierarchy. All four classes have default constructors. For each line below, determine whether it causes a compilation error, runtime error, or runs successfully. Consider each line independently of all other lines. (extended from Summer '17, MT1)

```

1 public static void main(String[] args) {
2     Cat c = new Animal(); Compilation error
3     Animal a = new Cat();
4     Dog d = new Cat(); Compilation error
5     Tree t = new Animal(); Compilation error
6
7     Animal a = (Cat) new Cat();
8     Animal a = (Animal) new Cat();
9     Dog d = (Dog) new Animal(); may have runtime error
10    Cat c = (Cat) new Dog(); runtime errorCompile error
11    Animal a = (Animal) new Tree(); runtime errorCompile error
12 }
```

Casting that is evidently impossible is also Compile error

### 3 SLList Vista

(Slightly adapted from Summer 2017 MT1) Consider the SLList class, which represents a singly-linked list. A heavily abridged version of this class appears below:

```

1 public class SLList {
2     ...
3     public SLList() { ... }
4     public void insertFront(int x) { ... }
5
6     /* Returns the index of x in the list, if it exists.
7        Otherwise, returns -1 */
8     public int indexOf(int x) { ... }
9 }

```

You think to yourself that the behavior of `indexOf` could be a bit confusing, so you decide it should throw an error instead. In the space below, write a class called `SLListVista` which has the same exact functionality of `SLList`, except `SLListVista`'s `indexOf` method produces a `NoSuchElementException` in the case that `x` is not in the list.

Since we have not covered exceptions yet, the following line of code can be used to produce a `NoSuchElementException`:

```

1 throw new NoSuchElementException();

```

```

1 import java.util.NoSuchElementException;
2
3 import java.util.NoSuchElementException;
4 public class SLListVista extends SLList{
5     1 usage
6     @Override
7     public int indexOf(int x) {
8         int index = super.indexOf(x);
9         if (index == -1) {
10             throw new NoSuchElementException();
11         }
12         return index;
13     }
14 }

```

## 4 Dynamic Method Selection

Totally lost

Modify the code below so that the max method of DMSList works properly. Assume all numbers inserted into DMSList are positive. You may not change anything in the given code. You may only fill in blanks. You may not need all blanks. (Spring '17, MT1)

```

1  public class DMSList {
2      private IntNode sentinel;
3      public DMSList() {
4          sentinel = new IntNode(-1000, null new LastIntNode());
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 class LastIntNode extends IntNode
18     public LastIntNode() {
19         super(0, null);
20     }
21     @Override
22     public int max() {
23         return 0;
24     }
25
26 }
27 public int max() {
28     return sentinel.next.max();
29 }
30 }
```