Polymorphism Mystery

1. You are given below code

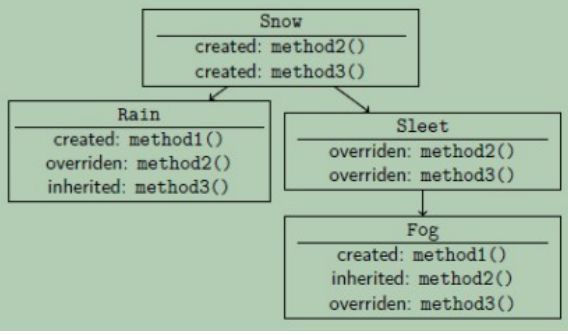
```java
public class Snow {
    public void method2() {
        System.out.println("Snow 2");
    }
    public void method3() {
        System.out.println("Snow 3");
    }
}
```

```java
public class Rain extends Snow {
    public void method1() {
        System.out.println("Rain 1");
    }
    public void method2() {
        System.out.println("Rain 2");
    }
}
```

```java
public class Sleet extends Snow {
    public void method2() {
        System.out.println("Sleet 2");
        super.method2();
        this.method3();
    }
    public void method3() {
        System.out.println("Sleet 3");
    }
}
```

```java
public class Fog extends Sleet {
    public void method1() {
        System.out.println("Fog 1");
    }
    public void method3() {
        System.out.println("Fog 3");
    }
}
```

Class Diagram



(Note that the arrows have the opposite direction of typical UML diagram)

2. Here are some vocabs.
   D_____ type == Runtime type
   S_____ type == Compile-time type

- (True/False) At compile time, Java knows the static type of an object. Thus can determine what is legal or illegal.
- (True/False) At compile time, Java knows the dynamic type of an object.

- When calling a method (which happens in run-time), the version called is always of the _____ type. What is the name of the method in Java Object class to view this actual type? _____

3. Keep the following rules in mind

1) **[Compile Time]**

- The static type dictates the legal action for the object

```
Shape s = new Triangle();
s.getArea(); //legal?
s.doTriangleThing(); //legal?
```

- Both casting _____ and _____ the tree is ok static type. This means subclass can be casted into superclass and vice versa. But casting in any other direction in the tree or outside the tree (complete foreigner) is invalid.

```
void foo(Object o){
    Shape s = (Shape) o; //legal?
    s.getArea();
}

void bar(Triangle t){
    Shape s = (Shape) t; //legal?
    s.getArea();
}
```

2) **[Run Time]**

- Happens only when no error in step 1)
- The dynamic type dictates which _____ will be called.
- Only casting _____ the tree is ok with dynamic type. This means only subclass can be casted up to superclass in run-time. Also casting in any other direction in the tree or outside the tree (complete foreigner) is invalid.

** Why? Run-time is when all the value gets evaluated thus, we cannot allow any _____.

```
class Point {
   int x; int y;
   …
}

class Point3D
   extends Point{
      int z;
}
```

```
Point p = new Point (3, 5);
Point3D p2 = (Point3D) p; //What could go wrong in run-time?
```

Such _____ is prevented by Java throwing a run-time _____.

** Why legal in compile time? Can't we check earlier in compile time?

For example,

```
void foo(Object o){ //formal parameter
    ((Point) o).translate(3,5);
}
```

Below is the actual invocation code of foo in run-time

```
foo(new Point(10,20)); //foo called with actual parameter
foo(new Point3D(10,20,30));
foo("hello");
```

The first two cases are certainly valid run-time type casting, but the latter is not. Since compiler doesn't know the actual type, so he wisely does not judge and throws no error and defer that judgement to run-time.
(* Mini life lesson: We should also NOT judge others as often times we do not have a full-knowledge of situation.)

As a good programmer, we should care about both compile-time and run-time. Best coding practice is check the run-time type and only if it is valid, do the typecasting!

```
void foo(Object o){
   if(o _____ Point) //ensures no ClassCastException will be thrown
    ((Point) o).translate(3,5);
}
```

How about we use
`if(o.getClass() == class.Point)` instead?

What is a difference?

4. **[Think-Pair-Share]** Discuss with your buddy what the results for below cases would be. It's ok to be wrong. Remember all thinking is good thinking. Go figure smarties!

1) Mystery #1

```
Snow var1 = new Sleet();
var1.method2();
```

2) Mystery #2

```
Snow var2 = new Rain();
var2.method1();
```

3) Mystery #3

```
Snow var3 = new Rain();
((Rain)var3).method1();
```

4) Mystery #4

```
Snow var4 = new Rain();
var4.method2();
```

5) Mystery #5

```
Snow var5 = new Rain();
((Sleet)var5).method2();
```

6) Mystery #5

```
Snow var6 = new Fog();
((Sleet)var6).method2();
```

7) Mystery #7

```
Snow var7 = new Sleet();
((Fog)var7).method1();
```

8) Mystery #8

```
Snow var8 = new Sleet();
System.out.println(((Object)var8).getClass());
```