Participant code_____

Glacier Tutorial

Glacier is an extension to Java that allows programmers to specify which classes are immutable. Then, if an immutable class accidentally permits mutation of any state that the class references, Glacier reports an error at compile time. Glacier includes an annotation, `@Immutable`, that can be put on class declarations. `@Immutable` classes have no fields that can be modified after the class's constructor exits. For example:

```
@Immutable public class ImmutableRectangle {…}
```

means that the ImmutableRectangle class has no fields that can be modified outside the constructor. Glacier gives errors when it finds fields that are of mutable type and when it finds assignments to fields of immutable objects.

To use @Immutable, you need to:

```
import edu.cmu.cs.glacier.qual.Immutable;
```

If a class's declaration is annotated `@Immutable`, then every instance of the class is `@Immutable`; there's no need to specify the annotation elsewhere, such as on variable declarations.

If you don't annotate a class or interface, Glacier assumes that it might be mutable. You might see `@MaybeMutable` in error messages, corresponding to these types, which might or might not have mutable fields.

In an `@Immutable` class, all fields must be either primitives, such as `int`, or references to other `@Immutable` classes. Some JDK classes, such as `String`, are already annotated `@Immutable`.

There are exactly two lines of code (total) in the Person and PersonHeight classes that will cause Glacier to report errors. Circle them and explain (briefly) why:

```
class PersonHeight {
    int feet;
    int inches;
}

@Immutable public class Person {
    String name;
    PersonHeight height;

    public void setName(String name) {
        this.name = name;
    }
}
```

`@Immutable` classes can extend `@MaybeMutable` classes that do not have any mutable fields. They can also implement any interface; mutability is about fields, not about methods, and

Participant code_____

interfaces have no method implementations. However, `@Immutable` interfaces can only be implemented by `@Immutable` classes. Subclasses of `@Immutable` classes must be `@Immutable`.

In the code below, `Circle` is not annotated, but we want it to be immutable. Add the `@Immutable` annotation to the *minimum* number of places necessary to make sure `Circle` is immutable without having Glacier report any errors when compiling the code below.

```
interface HitTesting {
    boolean testIntersection(int x, int y);
}


public abstract class Shape implements HitTesting { }


public class Circle extends Shape {
    private final double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public boolean testIntersection(int x, int y) {
        // …
        return false;
    }
}

public class RedCircle extends Circle {}
```

Arrays are special because the whole array can have a separate annotation from the elements inside the array. Immutable arrays cannot have their elements reassigned. For example:

```
Circle @Immutable [] circleArray = …
circleArray[0] = new Circle(3); // ERROR: can't assign to elements of
immutable arrays
```

By putting the annotation before the [], the annotation applies to the array itself rather than the elements. The elements are mutable or immutable according to their type.

Once copied via clone() or copyOf(), an array can be either @Immutable or @MaybeMutable as needed, but once the new array is assigned to a variable, the array has whatever annotation the variable specifies. However, because copying is expensive, this should only be done when copying is required for other reasons.

Re-write the declaration below to add the right annotation(s) so that elements of the array will always be the same Date objects, even though those objects themselves may change:

```
Date [] someDates;
```