

1. Use this code to answer the questions below

```
public class Animal {
    public void speak( ) { System.out.println("Grrr"); }
    public boolean likes(Object obj) { return obj == this; }
}
public class Dog extends Animal {
    public void speak() { System.out.println("Woof"); }
    public boolean likes(Animal other) { return (other instanceof Dog); }
}
```

And suppose we have these objects and references:

```
Animal a = new Dog();
Dog d = new Dog();
```

What is printed or returned in each of these cases:

	a.speak()
	a.likes(d)
	d.likes(a)

2. Consider this class hierarchy:

```
public class Arithmetic {
    private Number value;

    protected Number add(Number a, Number b) { /* add two numbers */ }
}
public class SubArithmetic extends Arithmetic {
    _____(d)_____ (e)_____ value;
    @Override
    _____(a)_____ (b)_____ add(_____(c)_____ a, _____(c)_____ b) { /* add */ }
}
```

In the **add()** method in class SubArithmetic, what values can be used? Circle all correct answers:

- (a) **public, protected, nothing (package scope), private**
- (b) **Number, Double, double, void, Integer**
- (c) **Number, Double**

For the **value** attribute in the subclass, what values are allowed. Write your own answer:

- (d) _____
- (e) _____

3. What is the primary purpose of creating and using an interface (circle one answer).

- (a) increase cohesion between methods in a class
- (b) decrease coupling between classes
- (c) make it easier to write the program
- (d) separate the specification of behavior from its implementation

4. In addition to the above, how does an interface enable us to use polymorphism?

5. Name these methods that are defined in the `Object` class. Write *parameters* and *return* types, for example: `int foo(String)`. If a method does not have return value, write `void`.

- (a) _____ returns a `String` representation of the object
- (b) _____ test if the *values* of two objects are the same
- (c) _____ return the actual type (`Class`) of an object
- (d) Which of the above methods *cannot* be overridden by any subclass of `Object`? _____

6. The `LocalDate` class has methods like this:

```
public static LocalDate of(int year, int month, int day);
public static LocalDate now( );
```

- (a) What is the purpose of these methods?
- (b) What is the general name for this *kind* of method (another example is `Calendar.newInstance()`)?

7. We have a `Person` class written in 2003 (before Java 7) with a birthday:

```
public class Person {
    private String name;
    private Date birthday;

    /** Initialize a new Person */
    public Person(String name, Date bday) {
        this.name = name;
        this.birthday = new Date(bday); // Part (a)
    }
    public int getAge( ) { /* return Person's current age */ }
    public Date getBirthday() { return (Date) birthday.clone(); }
    // other methods omitted
}
```

- (a) In the constructor, why create a new `Date` instead of just writing `"this.birthday = bday;"`?
- (b) We'd like to change the birthday attribute to be a **`LocalDate`** object (because its easier to work with). But many other parts of the program are using the **`Person`** class and its methods (like `getBirthday` and `getAge`). Can we change birthday to **`LocalDate`** without changing any other part of the program? Explain how.

- (c) The answer to part (b) makes use of which of the *fundamental characteristics of OOP*?

Here is partial code for a MusicPlayer that can play songs and podcasts. Some methods are not shown.

```
public class Song {
    private Media media; // a JavaFX Media object for the song
    public Song(Media song) { this.media = song; ... }
    public void play( ) { /* play the song */ }
    public void pause( ) { /* stop playing */ }
}
```

```
public class Podcast {
    private URL url; // a URL to get the podcast
    public Podcast(URL url) { ... }
    public void play( ) { /* download from url and play the podcast */ }
    public void pause( ) { /* stop playing */ }
}
```

```
public class MusicPlayer {
    // playlist can contain Songs and Podcasts
    List<Object> playlist = new ArrayList<Object>( );
    // This is the item now being played or null if nothing is playing
    Object nowplaying = null;
    /** add a new Song to playlist */
    public void add( Song song ) { playlist.add( song ); }
    /** add a new Podcast to playlist */
    public void add( Podcast pod ) { playlist.add( pod ); }

    /** Play items in playlist. Remove each item when it finishes. */
    public void play( ) {
        while( playlist.size() > 0 ) {
            // play first item in list. Remove it when finished.
            nowplaying = playlist.get( 0 );
            if ( nowplaying instanceof Song ) {
                Song song = (Song) nowplaying;
                song.play( );
            }
            else if ( nowplaying instanceof Podcast ) {
                Podcast pod = (Podcast) nowplaying;
                pod.play( );
            }
            else System.out.println("Skipping unrecognized item");
            playlist.remove( nowplaying );
            nowplaying = null;
        }
    }

    /** stop playing the current song or podcast. */
    public void stop( ) {
        if (nowplaying == null) return;
        if (nowplaying instanceof Song) ((Song)nowplaying).pause( );
        else ((Podcast)nowplaying).pause( );
    }

    /** Return the current song. or podcast. Is null if nothing is playing. */
    public Object getCurrentSong() { return nowplaying; }
}
```

8. Write Java code for an interface named *Playable* to represent anything the *MusicPlayer* can play. *Playable* should specify the behavior that the *MusicPlayer* must use to play a Songs, Podcasts, etc.

9. How do you make the *Song* class be *Playable*? Write the line(s) of *Song* that must be changed.

10. Rewrite the *MusicPlayer* class to use polymorphism.

a) *MusicPlayer* should not depend on *Song* or *Podcast*.

b) Eliminate duplicate code.

c) Change the code so we don't need to use "instanceof" or a *cast*.

```
public class MusicPlayer {
    _____ playlist = _____;
    _____ nowplaying = null;
    //TODO Write add method
    public void add( _____ ) {

    }

    //TODO Complete this method
    public void play( ) {
        while( playlist.size() > 0 ) {
            nowplaying = playlist.get( 0 );

            playlist.remove( nowplaying );
            nowplaying = null;
        }
    }
    //TODO Complete this method
    public void stop( ) {
        if (nowplaying == null) return;

    }

    public _____ getCurrentSong() { return nowplaying; }
}
```

11. Draw a UML class diagram of the new MusicPlayer design. Use proper UML notation.

- show attributes, methods, data types, and their visibility.
- show relations between classes and interfaces, including *implements*, *association*, and *inherits* using proper UML notation.
- show multiplicities of associations (1, *, etc).

Don't show dependencies on Java API classes like String or ArrayList.

12. The MusicPlayer sets **nowplaying = null**; to indicate no current song. This means that we have to be careful to always test `if (nowplaying != null) ...` before invoking any of its methods. For complex code this is error-prone (the programmer might forget).

A solution is called the *Null Object Pattern*: define a special object (or a subclass) that does nothing! Use that object instead of null to indicate nothing to play. Then we will never get a NullPointerException by calling **nowplaying.play()** when there is no current song.

Define a static final attribute of MusicPlayer named NO_SONG that is an **anonymous class** that implements *Playable*. The methods of NO_SONG do nothing.

```
public class MusicPlayer {
    /** A Playable object that means there is no current item. */
    //TODO write an anonymous class
    public static final _____ NO_SONG =

    // we can use NOSONG instead of null
    private Playable nowplaying = NO_SONG;
```

13. Fill in the blanks to show how to use an *Iterator* to sum a *Collection* of coins.

```
public double sum( Collection<Coin> coins ) {  
    _____ iterator = coins._____;  
    double sum = 0.0;  
    while ( _____ ) {  
        sum += _____ . _____ . _____ ;  
    }  
    return sum;  
}
```

14. We have an e-commerce application with a list of **Sale** records. The **Sale** class has a **getTotal()** method that returns total value of the sale. We want to count how many sales are small (total < 100.0), medium (100 <= total < 1000), and large (total > 1000).

```
// counters for small, medium, and large sales  
int small = 0;  
int medium = 0;  
int large = 0;  
/**  
 * Count how many sales have total value < 100, 100-1000, or >1000.  
 * @return array of 3 counts.  
 */  
public int[] count( List<Sale> sales ) {  
    for(int k=0; k < sales.size(); k++) {  
        if (sales.get(k).getTotal() < 100.0) small++;  
        else if (sales.get(k).getTotal() >= 100.0  
                && sales.get(k).getTotal() < 1000.0) medium++;  
        else if (sales.get(k).getTotal() >= 1000.0) large++;  
    }  
    // return an array containing the 3 counts  
    int[] x = new int[] { small, medium, large };  
    return x;  
}
```

Rewrite this code by applying lessons from this course: make it DRY, avoid magic numbers, and any other changes to improve correctness and readability.

15. What is encapsulation in object-oriented programming? Describe it.

16. Give an example of polymorphism using at most 3 Java statements and using only classes and/or interfaces in the Java API. Your example should be complete enough to show that polymorphism is really occurring. It should be something we can perform in BlueJ Codepad or Eclipse scrapbook.

Indicate which statement(s) shows polymorphism is occurring.

17. Designing, writing, and maintaining good software has these problems:

- *change*: requirements change, technology changes, user needs change
- *complexity*: real-world applications tend to be complex. This makes development slower, harder to maintain, and increases the number of bugs (errors).
- *cost*: software tends to expensive to develop and maintain.¹

To cope with these problems, we want software that is:

- a) "easy" to modify or extend. "easy" means that changes are localized. Changing or extending one part of an application does not require changing other parts. Parts don't depend on too many other parts.
- b) decomposable. Reduce complexity by dividing the application into smaller, self-contained parts.
- c) reusable, so we can use existing, tested components in new applications (reduces cost and bugs).

Explain how each of the 3 fundamentals of OOP help achieve these goals. Be concrete and give an example, preferably using the OOP labs as example.

¹This is why you should study in depth (read books and internet articles, study open-source code). To get a high paying job as programmer or software engineer you need in-depth knowledge. Basics like we cover in OOP are not enough. Would you trust a doctor who studied medicine by reading Powerpoint slides?