— Solution notes —

COMPUTER SCIENCE TRIPOS Part IA – 2014 – Paper 1

3 Object-Oriented Programming (RKH)

Encapsulation

(*a*) (*i*) Explain the purpose of access modifiers in OOP languages. [2 marks]

Answer: Access modifiers permit data or information hiding and encapsulation, encouraging clean interfaces and hidden implementation. To get both marks, the candidate had to explain what the modifiers allowed (control who can access sata and methods) and why this was good (mention encapsulation/data hiding/etc).

(*ii*) Copy and complete the table below to show the access restrictions for the four access modifiers in Java. [2 marks]

Access Modifier				
Defining class Class in same package Subclass in different package Non-subclass in different package				

Answer:

The marks were awarded here for correct values for `protected` and `package` (the others being trivial). A surprising number of candidates could not remember the name 'package'. If table entries were correct, this was not penalised.

	public	protected	package	private
Defining class	Can access	Can access	Can access	Can access
Classes in same package	Can access	Can access	Can access	
Subclass in different package	Can access	Can access		
Non-subclass in diff. package	Can access			

(*b*) A Java game designer wishes to store all the game preferences (e.g., player name, screen size, music volume, etc.) within a custom `Preference` class.

Class definition, Collections

(*i*) Assuming each preference is stored as a unique `String` key mapping to a `String` value, give a simple implementation of `Preference` that allows for efficiently setting or updating preferences and retrieving previously set ones. Your implementation should define an exception that is thrown when a preference key is requested but not present. [5 marks]

Answer: An illustrative definition is provided below. Marks were awarded for a sensible class definition using private state and public accessors **(1)**; choice of `HashMap` or similar to store keys and values **(1)**; correct use of generics in collections **(1)**; correct exception definition **(1)**; correct exception usage **(1)**.

```
public class NotFoundException extends Exception {}

public class Preference {
  private Map<String,String> mPrefs =
           new HashMap<String,String>();
  public void setPreference(String key, String val) {
    mPrefs.put(key,val);
```

```
  }
  public String getPreference(String key) throws
       NotFoundException {
    if (!mPrefs.containsKey(key)) throw new
       NotFoundException();
    return mPrefs.get(key);
  }
}
```

Some candidates created classes with explicit state and getters for each preference (e.g. `mName` with `getName()`). In this case, one mark was awarded for good encapsulation, and the two marks related to exceptions were still available. However, the maximum mark for this approach was three. Many candidates chose to define their *class* as throwing an exception as well as their method (e.g. `public class X throws Y`). This was a common error that, whilst not penalised here, was surprising.

Singleton

(*ii*) It is important that only one `Preference` object exists in a running game. Show how to apply access modifiers and the Singleton design pattern to ensure this. Your implementation should lazily instantiate the object. Is it necessary to make your class `final` or `Cloneable`? Explain your answer. [6 marks]

Answer: Sample code below (with previous code stripped out for clarity). Marks awarded for: private constructor **(1)**; private static instance **(1)**; static getter() **(1)**; correct lazy instantiation **(1)**.

```
public class PreferenceSingleton {

  private static PreferenceSingleton sInstance = null;

  private PreferenceSingleton() {}

  public static PreferenceSingleton getInstance() {
    if (sInstance==null) sInstance = new PreferenceSingleton();
    return sInstance;
  }
}
```

With regards `final` or `Cloneable`: they are not *needed* since there is a `private` constructor that prevents extension **(2)**. The majority of candidates attempted to answer this by explaining why these two modifiers might or might not be a good idea to apply, which is not what the question asked.

(*c*) The designer also implements other Singleton classes in the game and proposes to create a `SingletonBase` base class from which all such classes would inherit the singleton behaviour. By providing example Java code, explain why this is not viable. [5 marks]

Answer: The notion of an abstract singleton class is an attractive one, but it is not possible to make anything useful. We might start with something like:

```
public abstract class SingletonParent {
```

```
  private static SingletonParent sInstance = null;

  public SingletonParent getInstance() {
    return sInstance;
  }

  protected SingletonParent() {}
}
```

However, there is no way to initialise `sInstance` to be an instance of e.g. `Preference` because static methods are not overridden and (worse) all subclasses would share the same `sInstance`! Hence we would be forced to reimplement the pattern within each subclass so having a base class is pointless **(3)**. The last two marks were awarded for illustrative code (which was rarely provided).

Some candidates asserted that it was impossible due to the `private` constructor. Whilst this is strictly true (and attracted one mark), the conversion to `protected` was implicit and should have been considered.
