Institiúid Teicneolaíochta, Trá Lí
INSTITUTE OF TECHNOLOGY TRALEE

SUMMER EXAMINATIONS AY 2013 - 2014

Object Oriented Programming 3

PROG61005
CRN 43852

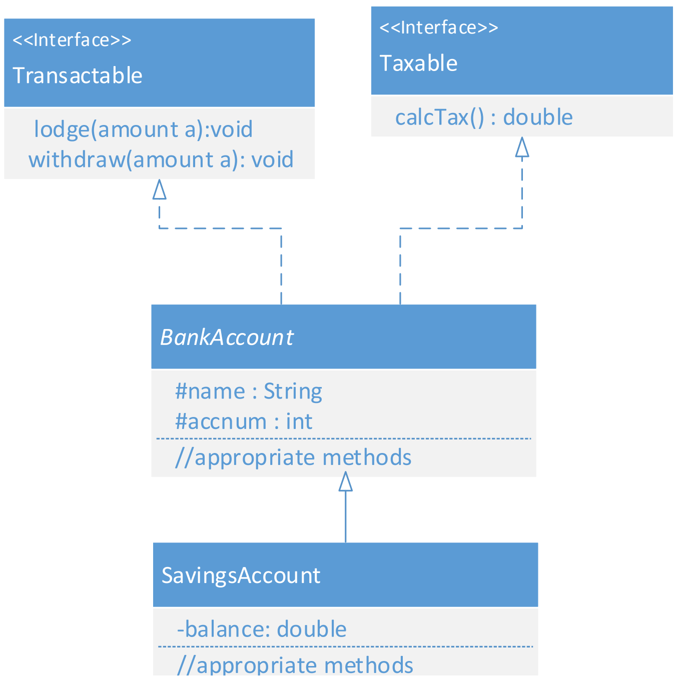
Internal Examiner:	Mr. John Walsh
External Examiner:	Mr Michael Godley

Duration of Exam: 2 Hours

Instructions to Candidates: **Answer Question 1 and one other question. Each question is 50 marks**

Question 1

In the VOPC diagram below, Transactable and Taxable are interfaces, BankAccount is an abstract class and SavingsAccount is a concrete class.



a) Explain the differences between an Interface and an Abstract Class.
(6 marks)

b) Write the full code for the *Transactable* and *Taxable* interfaces.
(7 marks)

c) The BankAccount Class includes three abstract methods appropriate to the VOPC diagram shown. Write the full code, including the class header, attributes and all appropriate methods for the BankAccount class.
(16 marks)

d) Write the full code, including the class header, attributes, a 3-argument constructor and methods for the SavingsAccount class. You can include stub methods where appropriate.
(12 marks)

e) What change(s) would you make to ensure that no class could inherit from the SavingsAccount class. (3 mark)

f) Explain where and why the Serializable marker interface is used in Java. Include an explanation of how it is implemented. (6 marks)

Question 2

a) Draw a hierarchical diagram of the Java Collection Framework including the various Interfaces(4) and Implementation classes(5). Do **not** include the Map interface and its implementation classes in your diagram. (8 marks)

b) Give a brief explanation of generic programming as used in the Java Collection Framework indicating two advantages of its use. You can include an example as part of your explanation. (8 marks)

c) The code below contains four java code statements. Examine each line and determine whether and why it will cause a compile error or not. (8 marks)

```
List [] myList1 =new List [5];
List myList2 = new List[5];
List myList3 = new List();
List myList5 = new Arraylist();
```

d) Considering the VOPC created in Question 1 and assuming appropriate constructor, accessor, mutator and toString methods, write a driver class that creates a generic collection of Savings Accounts. No duplicates are allowed in the collection. Any inputs are taken into the application via input dialog boxes. Use an appropriate structure to allow for multiple Savings Accounts to be input by the user. (14marks)

e) The application is required to print the collection of bank accounts available in order of the **name** attribute. One option to accomplish this is to write a new class based on the Comparator Interface to allow comparison of Savings Account objects. Write the code to accomplish this. (6 marks)

f) Write the additional statements required in part d) to print the collection of bank accounts available in order of the **name** attribute. (6 marks)

Question 3

The following additional method was included in the Banking system referred to in Question 1.

```
public void lodge(){
        String amount = JOptionPane.showInputDialog("Enter amount");
        if (validamount(amount))
        balance+=Double.parseDouble(amount);
};
```

a) There are now two lodge methods in the Banking System. Justify whether you would consider this an example of polymorphism or not. (8 marks)

b) Using the **? :** ternary operator (made up of three parts), write the full code, including the header, for the validamount(amount) method as used in the lodge method above. A valid amount is any amount greater than 0.

(6 marks)

c) Explain, using an example, dynamic method binding as used in Java.
(10 marks)

d) Rewrite the lodge() method using a try….catch block so that a NumberFormatException and an IllegalArgument exception are dealt with if the user enters text or an amount value less than 0.
(12 marks)

e) Rewrite the code you developed for part d) above to further improve its robustness so that the system would continue to execute until a valid entry is supplied. (5 marks)

f) Rewrite the lodge() method so that a NumberFormatException and an IllegalArgument exception are propagated back to the calling method.
(3 marks)

g) Write a brief note the naming convection used in Java for Packages, Classes, Interfaces, Methods, Variables and Constants. (6 marks)

Appendix : Summary of Collection Interface methods

Method Summary	
`boolean`	**`add`**`(E e)` Ensures that this collection contains the specified element (optional operation).
`boolean`	**`addAll`**`(Collection<? extends E> c)` Adds all of the elements in the specified collection to this collection (optional operation).
`void`	**`clear`**`()` Removes all of the elements from this collection (optional operation).
`boolean`	**`contains`**`(Object o)` Returns `true` if this collection contains the specified element.
`boolean`	**`containsAll`**`(Collection<?> c)` Returns `true` if this collection contains all of the elements in the specified collection.
`boolean`	**`equals`**`(Object o)` Compares the specified object with this collection for equality.
`int`	**`hashCode`**`()` Returns the hash code value for this collection.
`boolean`	**`isEmpty`**`()` Returns `true` if this collection contains no elements.
`Iterator<E>`	**`iterator`**`()` Returns an iterator over the elements in this collection.
`boolean`	**`remove`**`(Object o)` Removes a single instance of the specified element from this collection, if it is present (optional operation).
`boolean`	**`removeAll`**`(Collection<?> c)` Removes all of this collection's elements that are also contained in the specified collection (optional operation).
`boolean`	**`retainAll`**`(Collection<?> c)` Retains only the elements in this collection that are contained in the specified collection (optional operation).
`int`	**`size`**`()` Returns the number of elements in this collection.
`Object[]`	**`toArray`**`()` Returns an array containing all of the elements in this collection.
`<T> T[]`	**`toArray`**`(T[] a)` Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.

Interface Comparable<T>
Type Parameters:

`T` - the type of objects that this object may be compared to

Method Summary

`int`	`compareTo(T o)` Compares this object with the specified object for order.

Method Detail

compareTo

`int compareTo(T o)`

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

The implementor must ensure `sgn(x.compareTo(y)) == -sgn(y.compareTo(x))` for all `x` and `y`. (This implies that `x.compareTo(y)` must throw an exception iff `y.compareTo(x)` throws an exception.)

The implementor must also ensure that the relation is transitive: `(x.compareTo(y)>0 && y.compareTo(z)>0)` implies `x.compareTo(z)>0`.

Finally, the implementor must ensure that `x.compareTo(y)==0` implies that `sgn(x.compareTo(z)) == sgn(y.compareTo(z))`, for all `z`.

It is strongly recommended, but *not* strictly required that `(x.compareTo(y)==0) == (x.equals(y))`. Generally speaking, any class that implements the `Comparable` interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."

In the foregoing description, the notation `sgn(`*expression*`)` designates the mathematical *signum* function, which is defined to return one of `-1`, `0`, or `1` according to whether the value of *expression* is negative, zero or positive.

Interface Comparator<T>
Type Parameters:
`T` - the type of objects that may be compared by this comparator

Method Summary

`int`	`compare(T o1, T o2)` Compares its two arguments for order.
`boolean`	`equals(Object obj)` Indicates whether some other object is "equal to" this comparator.

Method Detail

compare

```
int compare(T o1,
            T o2)
```

Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

In the foregoing description, the notation `sgn(`*expression*`)` designates the mathematical *signum* function, which is defined to return one of `-1`, `0`, or `1` according to whether the value of *expression* is negative, zero or positive.

The implementor must ensure that `sgn(compare(x, y)) == -sgn(compare(y, x))` for all `x` and `y`. (This implies that `compare(x, y)` must throw an exception if and only if `compare(y, x)` throws an exception.)

The implementor must also ensure that the relation is transitive: `((compare(x, y)>0) && (compare(y, z)>0))` implies `compare(x, z)>0`.

Finally, the implementor must ensure that `compare(x, y)==0` implies that `sgn(compare(x, z))==sgn(compare(y, z))` for all `z`.

It is generally the case, but *not* strictly required that `(compare(x, y)==0) == (x.equals(y))`. Generally speaking, any comparator that violates this condition should clearly indicate this fact. The recommended language is "Note: this comparator imposes orderings that are inconsistent with equals."

Parameters:
`o1` - the first object to be compared.
`o2` - the second object to be compared.
Returns:
a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.
Throws:
`ClassCastException` - if the arguments' types prevent them from being compared by this comparator.