OLLSCOIL NA hÉIREANN
THE NATIONAL UNIVERSITY OF IRELAND, CORK

COLÁISTE NA hOLLSCOILE, CORCAIGH
UNIVERSITY COLLEGE, CORK

Mock Exam 2010

Second Year Computer Science

CS2500: Software Development

Dr Mock

INSTRUCTIONS: Answer **all 8** questions for full marks (160). Each question carries equal marks (20). There is **no need** to write whole classes *except where this is explicitly stated*. There is **no need** to write `import` statements. Use meaningful identifier names. Pay attention to the layout of your code and make sure your coding style adheres to the Java coding conventions. There is **no** need to add comments unless this is explicitly requested.

3 hours

Question 1: Basics. *(20/160 marks)*

Question 1.a. *(10 marks)*
Provide a `main( )` method that uses the enhanced `for` loop to print its arguments to standard output.

Question 1.b. *(10 marks)*
Provide a method that creates a `Scanner` to read from standard input. The method should use the `Scanner` to read all `int`s on standard input and print their sum to standard output.

Question 2: Classes. *(20/160 marks)*

Question 2.a. *(5 marks)*
What is an object and how does it relate to a class?

Question 2.b. *(15 marks)*
Implement a class called `Pet`. The class is used to represent the details of pet animals in a pet shop. Each `Pet` has a name, which is a `String`, an ID, which is an `int`, and a price, which is a `double`. The `Pet`'s name and its ID are fixed but its price may vary over time. Your class should include a constructor and relevant getter and setter methods.

Make sure your class is implemented using good object oriented and software engineering standards.

Question 3: Inheritance. *(20/160 marks)*

Question 3.a. *(5 marks)*
Explain the notions of inheritance and polymorphism.

Question 3.b. *(5 marks)*
Explain the purpose of the 'IS-A Test'.

Question 3.c. *(5 marks)*
Explain the purpose of the '`@Override`' annotation. For sake of simplicity there is no need to mention abstract classes and interfaces in your answers.

Question 3.d. *(5 marks)*
Explain the notion of 'overloading' and provide a meaningful example.

Question 4: Class Design. *(20/160 marks)*

Fota Wildlife Park has several `Animals`. Each `Animal` has *eating* and *roaming* behaviour. In addition each `Animal` has a specific *noise* behaviour. An `Animal` eats either grass or meat. Currently Fota have a `Hippo`, a `Lion` (`Feline Animal`), and a `Wolf` (`Lupine Animal`). *However, they are* expecting *a* `Dog`*, a* `Tiger`*, a* `Cat`*, and many more* `Feline` *and* `Lupine` `Animals`*.*

The `Hippo` eats grass and roams alone. The `Feline Animals` roam alone and eat meat. The `Canines` roam in packs and also eat meat.

Model the behaviour of Fota's `Animals` **using abstract classes in the higher levels and concrete classes at the final level of the class hierarchy**. The noise behaviour should be implemented by printing the name of the `Animal` that makes the noise. **Write each class and abstract class on a separate sheet in your answerbook.**

Keep in mind that your class design should support the addition of new `Animals` with as little coding effort as possible.

Question 5: Threads. *(20/160 marks)*

Question 5.a. *(10 marks)*

Explain the notion of a race condition, provide an example, and explain how they may be avoided.

Question 5.b. *(10 marks)*

Explain the notion of deadlock and provide an example.

Question 6: Event Handlers. *(20/160 marks)*

Provide a class `MockExamButton`. The class should have a `main( )` method, which displays a window with a button on in. The initial size of the window should be 200×300 pixels. Initially the button should display the text '`click me`'. However, when the user clicks on the button the text should change to '`Number of clicks =` n', where n is the number of times the button has been clicked.

Table 1 on Page 5 lists some useful (and not so useful) methods and constants which you may use to answer this question. Notice that the table is not exhaustive and lists some red herrings.

Question 7: Generics and Iterators. *(20/160 marks)*

Question 7.a. *(10 marks)*

What is a generic class and why are they useful?

Question 7.b. *(10 marks)*

Describe the `Iterable` interface and explain how it is used.

Question 8: Enumerated Types. *(20/160 marks)*

Question 8.a. *(10 marks)*

Provide a critical comparison of `int enums` and `Java enums`.

Question 8.b. *(10 marks)*

Implement an `enum` class which represents a mini solar system. The solar system has two planets: Earth and Mercury. Each planet has a mass and a radius. The mass of Earth is 5.975×10^{24} kg and its radius is 6.378×10^{6} m. The mass of Mercury is 3.303×10^{23} kg and its radius is 2.439×10^{6} m.

Class/Interface	Methods and Constants
JButton	void addActionListener(ActionListener listener)
JButton	void addChangeListener(ChangeListener listener)
JButton	void doClick( )
JButton	void doClick(int pressTime)
JButton	void fireActionPerformed(ActionEvent event)
JButton	void fireItemStateChanged(ItemEvent event)
JButton	void fireStateChanged( )
JButton	void setText(String text)
JFrame	static int EXIT_ON_CLOSE
JFrame	Container getContentPane( )
JFrame	void setDefaultCloseOperation(int operation)
JFrame	void setSize(int x, int y)
JFrame	void setVisible(boolean visibility)
JFrame	void update(Graphics g)
Component	void add(Component comp)
Component	void addContainerListener(ContainerListener listener)
Component	void addPropertyChangeListener(PropertyChangeListener listener)
Component	void addPropertyChangeListener(String propertyName, PropertyChangeListener listener)
ActionListener	void actionPerformed(ActionEvent event)
ChangeListener	void stateChanged(ChangeEvent event)
ItemListener	void itemStateChanged(ItemEvent event)

Table 1: Useful methods and constants. Some of the `JButton` methods are inherited from `AbstractButton`. Some of these methods are not needed: they're red herrings.

1 Answers to Questions

1.1 Mock Exam 2010 (Answers)

Answer 1.a. The following is a possible answer.

```
public static void main( String[] args ) {
    for (String arg : args) {
        System.out.println( arg );
    }
}
```

Answer 1.b. The following is a possible answer.

```
public static void computeSum( ) {
    Scanner scan = new Scanner( System.in );
    int sum = 0;
    while (scan.hasNextInt( )) {
        sum += scan.nextInt( );
    }
    System.out.println( sum );
}
```

Answer 2.a. An object is an instance of a given class. The class is a blueprint for the object. It describes the object's instance variables and instance methods and how the object should be constructed.

Answer 2.b. The following is possible answer.

```
public class Pet {
    private final String name;
    private final int    id;
    private       double price;

    public Pet( String name, int id ) {
        this.name = name;
        this.id   = id;
    }
    public String getName( ) {
        return name;
    }
    public int    getId( ) {
        return id;
```

```
    }
    public double getPrice( ) {
        return price;
    }
    public void   setPrice( double price ) {
        this.price = price;
    }
}
```

Answer 3.a. In Java subclasses may extend a unique superclass. When a subclass extends its superclass, the subclass inherits all `public` attributes and methods from the superclass. This means the subclass can access any inherited attribbute and has the (default) method behaviour when calling an inherited method. The term polymorphism refers to the ability of a subclass instance to appear as an instance of its superclass.

Answer 3.b. The 'IS-A Test' is used to decide if a given class can extend (be a subclass of) another given class. Specifically, if A and B are classes and if 'A IS-A B' makes "sense" then A extends B.

Answer 3.c. A class which extends another class may override the inherited methods from the superclass. Overriding a method is done by providing a new definition for the method. The '`@Override`' annotation is meant to make the overriding more explicit and helps detect/prevent errors such as typos in the name of the overridden method.

Answer 3.d. If two methods from the same class have the same name, the same visibility, but different types for their parameter list then the methods are said to *overload* each other. The following two methods overload each other.

```
public void f( int a ) { }
public void f( int a, int b ) { }
```

Answer 4. The following is a possible answer.

```
public abstract class Animal {
    private boolean eatsGrass = false;
    public void roam( ) {
        System.out.println( "Roaming alone" );
    }
    public abstract void makeNoise( );
    public final void eat( ) {
        System.out.println( "Eating " + (eatsGrass ? "grass" : "meat") );
    }
    public setEatsGrass( boolean eatsGrass ) {
        this.eatsGrass = eatsGrass;
    }
```

```
    public boolean getEatsGrass( ) {
        return eatsGrass;
    }
}
```

```
public abstract class Feline extends Animal {
}
```

```
public abstract class Lupine extends Animal {
    @Override
    public void roam( ) { System.out.println( "Roaming in packs." ); }
}
```

```
public class Lion extends Feline {
    @Override
    public void makeNoise( ) { System.out.println( "I'm a lion" ); }
}
```

```
public class Wolf extends Lupine {
    @Override
    public void makeNoise( ) { System.out.println( "I'm a wolf" ); }
}
```

```
public class Hippo extends Animal {
    public Hippo( ) { setEatsGrass( true ); }
    @Override
    public void makeNoise( ) { System.out.println( "I'm a hippo" ); }
}
```

Answer 5.a. A race condition is an error in a concurrent program by which the output is ill-defined and depends on the right sequence or timing of other events.

For an example, lets assume a multi-threaded program. Each thread is supposed to increment a shared variable. If the threads don't cooperate (synchronise) their actions then the final result (the variable's value) may be ill-defined. For example, lets assume two threads. Furthermore, assume the value of the variable is 0. The following may happen:

1. Two threads may simultaneously get the value of the variable;
2. Both threads will "think" the value is 0.

3. Both threads assign 1 to the variable.

Here the overall result is 1. With a different order of events the result of the program may be different. For example:

1. Thread 1 gets the value of the variable (it is 0);
2. Thread 1 assigns 1 to the variable;
3. Thread 2 gets the value of the variable (it is 1);
4. Thread 2 assigns 2 to the variable.

Race conditions may be avoided by using `synchronized` methods. Making a method `synchronized` guarantees exclusive thread access to the method. This allows us to synchronise thread actions and implement exclusive access to shared resources.

Answer 5.b. Deadlock occurs when two or more threads are in possession of a resource and are waiting for each other to release their resource. An easy example is a program with two threads and two `synchronized` methods `void a( int arg )` and `void b( int arg )`. The methods are mutually recursive. If the first thread calls `a( )` and the second thread calls `b( )` and if neither of them is blocked then the program will be deadlocked as the first thread cannot call `b( )` and the second cannot call `a( )`.

Answer 6. The following is a possible answer.

```
public class MockExamButton extends JButton
                        implements ActionListener {
    int clicks;

    public static void main( String[] args ) {
        JFrame frame = new JFrame( );
        MockExamButton button = new MockExamButton( );
        frame.getContentPane( ).add( button );
        frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        frame.setSize( 300, 300 );
        frame.setVisible( true );
    }

    private MockExamButton( ) {
        super( "click me" );
        addActionListener( this );
    }

    @Override
    public void actionPerformed( ActionEvent event ) {
        setText( "Number of clicks = " + (++ clicks) );
```

```
    }
}
```

Answer 7.a. A generic class is a class which is parameterised over one or more type parameters. For example, the class `Comparable` is generic, so you may write `Comparable<T>` to indicate that the instances of this class are instances of `T`. The advantage of generic class is that they provide more safety, help you prevent certain runtime errors, and avoid the need for casting at compile time and type checking at runtime.

Answer 7.b. `Iterable` is a generic interface. It is used to define the requirements for collection classes which allow you to iterate over what's in the collections. Instances of classes which implement the `Iterable` interface can be used in the generalised `for` loop notation:

```
Iterable<Class> list = // Construct it.
for (Class c : list) {
  // Use c.
}
```

To implement `Iterable<T>` you have to implement `public Iterator<T> iterator( )`. This usually involves an anonymous class. The following is an example:

```
public Iterator<T> iterator( ) {
    return new Iterator<T>( ) {
        public boolean hasNext( ) { ... }
        public T next( ) { ... }
        public void remove( ) { ... }
    };
}
```

Answer 8.a. `Java enums` and `int enums` both provide a way to define a collection of "constant" symbols. `Java enums` overcome all the disadvantages of `int enums`. `Java enums` are **easy to use**. For example, translating to `String` is automatic and iterating over all `enum` constants is easy. With `int enums` this is not the case: converting to `String` typically requires a `switch` statement and iterating is difficult if the constants are not contiguous. `Java enums` help you **maintain your code**. For example, rearranging `enum` constants, removing, or adding constants does not break code. With `int enums` rearranging may break code if typos result in duplicate constants. Likewise, removing and adding constants breaks the code for iterating.

Answer 8.b. The following is a possible implementation.

```
public enum Planet {
    EARTH  ( 5.975e+24, 6.378e6 ),
```

```
    MERCURY( 3.303e+23, 2.439e6 );

    private final double mass;
    private final double radius;

    private Planet( double mass, double radius ) {
        this.mass   = mass;
        this.radius = radius;
    }

    public double getMass( )    { return mass; }
    public double getRadius( )  { return radius; }
}
```
