Programming Concepts Using Java

Quiz 1 – Revision

W04:L01: Abstract classes and interfaces

- We can use the class hierarchy to group together related classes
- An abstract method in a parent class forces each subclass to implement it in a sensible manner
- Any class with an abtract method is itself abstract
  - Cannot create objects corresponding to an abstract class
  - However, we can define variables whose type is an abstract class
- Abstract classes can also describe capabilities, allowing for generic functions
- An interface is an abstract class with no concrete components
  - A class to extend only one parent class, but it can implement any number of interfaces

W04:L02: Interfaces

- An interface is a purely abstract class – all methods are abstract
- A class implements an interface – provides concrete code for each abstract function
- Classes can implement multiple interfaces – abstract functions, so no contradictory inheritance
- Interfaces express abstract capabilities
  - Capabilities are expressed in terms of methods that must be present
  - Cannot specify the intended behaviour of these functions
  - Another class only needs to know about these capabilities
- Java later allowed concrete functions to be added to interfaces
  - Static functions — cannot access instance variables
  - Default functions — may be overridden
- Reintroduces conflicts in multiple inheritance
  - Subclass must resolve the conflict by providing a fresh implementation
  - Special "class wins" rule for conflict between superclass and interface

W04:L03: Private classes

- An object can have nested objects as instance variables
- In some situations, the structure of these nested objects need not be exposed
- Private classes allow an additional degree of data encapsulation
- Combine private classes with interfaces to provide controlled access to the state of an object

```
public class LinkedList{
    private int size;
    private Node first;
    public Object head(){ ... }
    public void insert(Object newdata){
        ...
    }
    private class Node {
        public Object data;
        public Node next;
        ...
    }
}
```

W09:L04: Controlled interaction with objects

- Can provide controlled access to an object
- Combine private classes with interfaces
- External interaction is through an object of the private class
- Capabilities of this object are known through a public interface
- Object can maintain instance variables to track the state of the interaction

```
public interface QIF{
    public abstract int getStatus(int trainno, Date d);
}
public class RailwayBooking {
    private BookingDB railwaydb;
    public QIF login(String u, String p){
        QueryObject qobj;
        if (valid_login(u,p)) {
            qobj = new QueryObject();
            return(qobj);
        }
    }
    private class QueryObject implements QIF {
        private int numqueries;
        private static int QLIM;
        public int getStatus(int trainno, Date d){
        if (numqueries < QLIM){
            // respond, increment numqueries
        }
    }
}
```

W04:L05: Callbacks

- Callbacks are useful when we spawn a class in parallel
- Spawned object notifies the owner when it is done
- Can also notify some other object when done
  - `owner` in `Timer` need not be the object that created the `Timer`
- Interfaces allow this callback to be generic
  - `owner` has to have the capability to be notified

W04:L05: Callbacks (Cont.)


```
public interface Timerowner{
    public abstract
        void timerdone();
}
public class Myclass
        implements Timerowner{
    public void f(){
        ..
        Timer t = new Timer(this);
        // this object
        // created t
        ...
        t.start(); // Start t
        ...
    }
    public void timerdone(){...}
}
```

```
public class Timer
        implements Runnable{
    // Timer can be
    // invoked in parallel
    private Timerowner owner;
    public Timer(Timerowner o){
        owner = o; // My creator
    }
    public void start(){
        ...
        owner.timerdone();
        // I'm done
    }
}
```

W09:L04: Iterator

- Iterators are another example of interaction with state
  - Each iterator needs to remember its position in the list
- Export an object with a prespecified interface to handle the interaction
- Need the following abstraction

```
Start at the beginning of the list;
while (there is a next element){
    get the next element;
    do something with it
}
```

- Encapsulate this functionality in an interface called Iterator

```
public interface Iterator{
    public abstract boolean has_next();
    public abstract Object get_next();
}
```

W09:L06: Iterator (Cont.)

- Create an `Iterator` object and export it!

```
public class Linearlist{
    private class Iter implements Iterator{
        private Node position;
        public Iter(){...} // Constructor
        public boolean has_next(){...}
        public Object get_next(){...}
    }
    // Export a fresh iterator
    public Iterator get_iterator(){
        Iter it = new Iter();
        return(it);
    }
}
```

- The new Java `for` over lists implicitly constructs and uses an iterator

```
for (type x : a)
    do something with x;
```