**CST Part IA: OOP, SV 2**
**Joe Yan**
**2016-11-14**

# 1 Example Sheet 1314 20A

The main difference between abstract classes and classes is that abstract classes can not be instantiated. So if we want to use an abstract classes, we must inherit from it first and provide implementations for all abstract methods. Note that abstracted classes do not necessarily contains abstract methods. However the class contains abstract methods must be abstract.
The interface is different from class and abstract class that it can only contains abstract methods. (on other words it needs to be inherit to gain implementations either.) And interfaces cannot contains any fields which is different from both non-abstract or abstract classes.(It is purely abstract)

# 2 Example Sheet 1314 25B

```
public interface OOPListQueueInterface {
        OOPListQueue normalise();

        boolean isEmpty();

        OOPListQueue enqueue(int x);

        OOPListQueue dequeue();

}

public class OOPListQueue implements OOPListQueueInterface {
        private OOPLinkedList listA;
        private OOPLinkedList listB;

        public OOPListQueue() {
                // TODO Auto-generated constructor stub
                listA = new OOPLinkedList();
                listB = new OOPLinkedList();
        }

        @Override
        public OOPListQueue enqueue(int x) {
                listB.add(x);
                return this;
        }

        @Override
        public OOPListQueue dequeue() {
                if (this.isEmpty()) {
                } else if (listA.isEmpty()) {
                        this.normalise();
                        listA.remove();
```

```
                } else {
                        listA.remove();
                }
                return this;
        }

        @Override
        public OOPListQueue normalise() {
                listA = listA.append(listB.reverse());
                listB.setEmpty();
                return this;
        }

        @Override
        public boolean isEmpty() {
                // TODO Auto-generated method stub
                return listA.isEmpty() && listB.isEmpty();
        }
}

public class OOPLinkedList {
        private OOPLinkedListElement head;

        public OOPLinkedList() {
                head = null;
        }

        public OOPLinkedList(OOPLinkedListElement h) {
                head = h;
        }

        protected OOPLinkedList setHead(OOPLinkedListElement e) {
                head = e;
                return this;
        }

        protected OOPLinkedListElement getHead() {
                return head;
        }

        public boolean isEmpty() {
                return head == null;
        }

        public void setEmpty() {
                head = null;
        }

        public OOPLinkedList add(int x) {
                head = new OOPLinkedListElement(x, head);
                return this;
        }

        public OOPLinkedList remove() {
```

```
            if (head != null) {
                    // int i = head.getElement();
                    head = head.getNext();
                    return this;
            } else {
                    return this;
            }
    }

    public int get() throws Exception {
            if (head == null)
                    throw new Exception("Empty OOPLinkedList.");
            else
                    return head.getElement();
    }

    public int length() {
            if (head == null)
                    return 0;
            int len = 1;
            OOPLinkedListElement p = head;
            while (p.getNext() != null) {
                    ++len;
                    p = p.getNext();
            }
            return len;
    }

    public OOPLinkedList reverse() {

            int len = this.length();
            OOPLinkedList reversedList = new OOPLinkedList();
            int hh;
            try {
                    for (int i = 0; i < len; ++i) {

                            hh = this.get();

                            reversedList.add(hh);
                            this.remove();
                    }
            } catch (Exception e) {
                    System.out.println("This should never happen.");
                    e.printStackTrace();
            }

            head = reversedList.head;
            return reversedList;

    }

    public OOPLinkedList append(OOPLinkedList tail) {

            if (this.isEmpty())
```

```
                return tail;
            int originHeadElement;
            try {
                originHeadElement = this.get();
                return (this.remove().append(tail)).add(originHeadElement);

            } catch (Exception e) {
                System.out.println("This should never happen.");
                e.printStackTrace();
            }
            return null;
        }
}
```

# 3 Example Sheet 1314 30C

- Checked exception is recoverable during runtime.
- It needs to be thrown in the method.
- It needs try and catch block when being called.

```
public class CheckedExcetion {
        public int[] mArray;

        public CheckedExcetion() {
            mArray = new int[5];
        }

        public void setNOnes(int i) throws ArrayIndexOutOfBoundsException {
            for (int j = 0; j < i; j++) {
                mArray[j] = 1;
            }
        }

        public static void main(String args[]) {
            CheckedExcetion a = new CheckedExcetion();
            int i = 6;
            try {
                a.setNOnes(i);
            } catch (ArrayIndexOutOfBoundsException e) {
                // TODO: handle exception
                a.mArray = new int[i];
            }
        }
}
```

- Unchecked exception is not recoverable.
- It is a programming error.

# 4 Example Sheet 1314 35C

a,b.
```
public class MyClass implements Cloneable {
        private String mName;
        private int[] mData;

        public MyClass(MyClass toCopy) {
                this.mName = toCopy.mName;
                this.mData = new int[toCopy.mData.length];
                for (int i : mData) {
                        this.mData[i] = toCopy.mData[i];
                }
        }

        public MyClass clone() throws CloneNotSupportedException {
                MyClass clone = (MyClass) super.clone();
                mData = mData.clone(); // However, if the mData is final,
                                       //a copy constructor must be used.
                return clone;
        }
}
```

c. If we already build a clone constructor in the class and then we extend the class to a child class. Now the problem comes because the constructor of the parent type will not "know" we created a child class so we need to tell the parent constructor which type the object actually is (such as using instanceof).
Clone function will not have such problem we just need to write clone function for the child and the dynamic polymorphism will automatically find the clone we need. (Use super.clone() to pass the information for parent clone)
Overall clone function is recommended for hierarchy OOP structure.

d. Like the annotation in the code said, if the field is final after we use the super.clone then that final field is final and can no longer be changed! Now we need a clone constructor to give the final field value at the first time.