Quiz for CS401 MPP – September 2015 (used as exercise questions)

1. For each of the lambda expressions below, do the following:
a. Assign an appropriate type (some functional interface)
b. Express it as a method expression

```
A. () -> Math.random()
B. (CheckoutRecord record) -> record.getCheckoutEntries()
C. (Long a, Long b) -> a.compareTo(b)
```

2. Which of the following lines would not compile? Why? (Suppose Employee is a super class of Manager)

```
List<? super Manager> list1 = new ArrayList<>();
List<Employee> list2 = new ArrayList<>();
List<Manager> list3 = new ArrayList<>();
list1 = list3;
list1 = list2;

List<? extends Manager> list4 = new ArrayList<>();
list4 = list2;
list4 = list3;
```

3. Consider the following generic class that represents a node in a singly linked list: True/False

```
public class Node<T> {

   private T data;

   private Node<T> next;

   public Node(T data, Node<T> next) }

      this.data = data;

      this.next = next;

   }

   public T getData() { return data; }

   // ...

}
```

Because the type parameter T is unbounded, the Java compiler replaces it with Object:

```
public class Node {

   private Object data;

   private Node next;
```

```
    public Node(Object data, Node next) {
        this.data = data;
        this.next = next;
    }
    public Object getData() { return data; }
    // ...
}
```

4. In the following example, the generic Node class uses a bounded type parameter: True/False

```
public class Node<T extends Comparable<T>> {
    private T data;
    private Node<T> next;
    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }
    public T getData() { return data; }
    // ...
}
```

The Java compiler replaces the bounded type parameter T with the first bound class, Comparable:

```
public class Node {
    private Comparable data;
    private Node next;
    public Node(Comparable data, Node next) {
        this.data = data;
        this.next = next;
    }
    public Comparable getData() { return data; }
```

// …

}

5. The Java compiler also erases type parameters in generic method arguments. Consider the following:

```
// Counts the number of occurrences of elem in anArray.
//
public static <T> int count(T[] anArray, T elem) {
    int cnt = 0;
    for (T e : anArray)
        if (e.equals(elem))
            ++cnt;
        return cnt;
}
```

Because `T` is unbounded, the Java compiler replaces it with `Object`. So the erased method should look like this True/False

```
public static int count(Object[] anArray, Object elem) {
    int cnt = 0;
    for (Object e : anArray)
        if (e.equals(elem))
            ++cnt;
        return cnt;
}
```

6. Will the following class compile? Why or why not?

```
public final class Algorithm {
    public static <T> T max(T x, T y) {
        return x > y ? x : y;
    }
}
```

7. Write a generic method to exchange the positions of two different elements in an array.

8. If the compiler erases all type parameters at compile time, why should you use generics?

9. What is the following method converted to after type erasure?

```
public static <T extends Comparable<T>>
    int findFirstGreaterThan(T[] at, T elem) {
    // ...
}
```

10. Will the following method compile? If not, why?

```
public static void print(List<? extends Number> list) {
    for (Number n : list)
        System.out.print(n + " ");
    System.out.println();
}
```

11. Write a generic method to find the maximal element in the range [begin, end) of a list.

12. Will the following class compile? If not, why? (we didn't talk about this but think about it)

```
public class Singleton<T> {
```

```
        public static T getInstance() {
            if (instance == null)
                instance = new Singleton<T>();

            return instance;
        }
        private static T instance = null;
    }
```

13. Given the classes in question 7 and the Node class below:

```
class Node<T> { /* ... */ }
```

Will the following code compile? Why or why not?

```
Node<Circle> nc = new Node<>();
Node<Shape>  ns = nc;
```

14. Consider this class:

```
class Node<T> implements Comparable<T> {
    public int compareTo(T obj) { /* ... */ }
    // ...
}
```

Will the following code compile? Why or why not?

```
Node<String> node = new Node<>();
Comparable<String> comp = node;
```

15. Explain what is a Stream? (in lesson 9) List 3 features of Streams.

16. Stream instances are no longer usable once a terminal operation is performed on it. True/False

17. What is the template of using streams as a Java programmer?

18. List the functional interfaces used in 3 Stream operations: filter(), map() and reduce().

19. Suppose you have an array int[] values = {1, 2, 3, 4, 5}; What does Stream.of(values) return?

20. In the following code, which would compile without errors (Mark all that applies).

```
public interface A {
    default void foo(){
       System.out.println("Calling A.foo()");
    }
}

public interface B {
    default void foo(){
       System.out.println("Calling B.foo()");
    }
}
```

[ ]

```
public class Clazz implements A, B {
}
```

[ ]

```
public class Clazz implements A, B {
    public void foo(){}
}
```

[ ]

```
public abstract class Clazz implements A, B {
   public abstract void foo();
}
```

[ ]

```
public class Clazz implements A, B {
    public void foo(){
       A.super.foo();
    }
}
```