Instructions

This quiz is 80 minutes long. It contains 31 questions in 13 pages (including this page) for a total of 100 points.

Please check your copy to make sure that it is complete before you start. Turn in all pages, together, when you finish. Write your name on the top of every page. Please write neatly. No credit will be given if we cannot read what you write. Good luck!

<table>
<thead>
<tr>
<th>Question Name</th>
<th>Maximum Points</th>
<th>Question Numbers</th>
<th>Points Given</th>
</tr>
</thead>
<tbody>
<tr>
<td>True/False</td>
<td>30</td>
<td>1-15</td>
<td></td>
</tr>
<tr>
<td>Specifications</td>
<td>6</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>Module Dependency</td>
<td>9</td>
<td>17-19</td>
<td></td>
</tr>
<tr>
<td>Exceptions</td>
<td>9</td>
<td>20-22</td>
<td></td>
</tr>
<tr>
<td>Generics</td>
<td>10</td>
<td>23-24</td>
<td></td>
</tr>
<tr>
<td>Immutability</td>
<td>6</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>Visitor Pattern</td>
<td>15</td>
<td>26-28</td>
<td></td>
</tr>
<tr>
<td>Equality</td>
<td>15</td>
<td>29-31</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>100</strong></td>
<td><strong>1-31</strong></td>
<td></td>
</tr>
</tbody>
</table>
True/False [30 points, 2 points each]

Circle the correct answer:

1. **T / F**
   The representation invariant should hold at all times during the execution of a method. _Need only hold at the end and beginning._

2. **T / F**
   A precondition can simplify the implementation of a method.

3. **T / F**
   Methods of immutable types always have empty modifies clauses in their specifications. _Can modify arguments (‘this’ should not change)._  

4. **T / F**
   A Java interface type cannot be used in a typecast since every object’s type is a concrete class.

5. **T / F**
   The representation of an abstract data type (ADT) is exposed if one of its methods returns a mutable object. _Could copy before returning it._

6. **T / F**
   An immutable ADT can have a mutable representation. _Just don’t expose it!_

7. **T / F**
   If you can prove that every constructor produces a well-formed object that satisfies the representation invariant (RI) and that every mutator preserves the RI of the argument object, then RI assertions will never fail. _Rep exposure._

8. **T / F**
   The Model-View-Controller pattern can only used with composite view objects.

9. **T / F**
   In Java, an interface can ‘extend’ another interface.

10. **T / F**
    A hashCode() method that uses only the address of the object to compute the hash code for that object will always satisfy the Object Contract. _Not if you override equals!_
11. **T/F**
Is it possible that the code of `A.foo` is executed during the evaluation of `(new B()).foo()` in the following?

```java
class A {
    ...
    public foo() { ... }
}
class B extends A {
    ...
    public foo() { ... }
}
```

*Could call super.foo() in B.*

For questions 12-15 assume
- B extends A
- A has a defined `apply()` method and so does B
- a refers to an object whose compile-time type and run-time type is A
- b refers to an object whose compile-time and run-time type is B.

12. **T/F**
`b.apply(a)` can behave differently than `((B) b).apply(a)`.

13. **T/F**
`b.apply(a)` can behave differently than `((A) b).apply(a)`.

14. **T/F**
`b.apply(a)` can behave differently than `b.apply((B) a)`.

15. **T/F**
`b.apply(a)` can behave differently than `((B) b).apply((A) a)`.

**MANTRA:** Overriding == Runtime Type,
Overloading == Compile/Declared Type
Specifications [6 points]

16. Write a precondition or requires clause for the method `removeDuplicates`, so all duplicates from `List lst` are removed. Your precondition should be non-trivial, e.g., it cannot be that the input `lst` is a list without duplicates. Feel free to write your specification in plain English.

```java
public static void removeDuplicates(List lst) {
    if (lst == null || lst.size() == 0) return;

    List copy = new ArrayList(lst);
    Iterator elements = copy.iterator();
    Object pre = elements.next();

    while (elements.hasNext()) {
        Object nex = elements.next();
        if (pre.equals(nex)) lst.remove(nex);
        else pre = nex;
    }
}
```

Requires clause or precondition:

Requires input to be sorted

OR

Requires duplicates to be adjacent
Module Dependences [9 points]

Examine the Module Dependence Diagram (MDD) below and answer questions 17-19.

Arrow means “depends on.” Which parts need to be examined and possibly changed (answer with A, B, C, D, and/or E) if

17. [3 points] Only the specification of B.method1() is changed?
   
   A ( B )

18. [3 points] Only the implementation of E.method1() is changed?
   
   None ( E )

19. [3 points] Interface D is replaced with a new part F?
   
   A, B, C, E
Exceptions [9 points]

For each snippet of code in questions 20-21, indicate whether the exception being thrown should be a checked exception or unchecked and explain briefly why.

20. [3 points]
   if (!checkRepInv())
       throw new AException();

   *Unchecked exception. Programmer cannot be expected to fix this.*

21. [3 points]
   if (fileNotFound())
       throw new BException();

   *Checked exception. Programmer should handle this exception.*

22. [3 points]
   // requires: x >= 0
   public void insert(int x) {
       if ( x < 0 ) throw new CException();
   }

   *Unchecked exception. Programmer cannot be expected to fix this.*
Generics [10 points]

Consider the generic classes below, and answer the questions 22-23.

```java
class Basket<E> {
    private E element;  // only 1 element
    public void setElement(E x) {
        element = x;
    }
    public E getElement() {
        return element;
    }
}
class Fruit { ... }
class Apple extends Fruit { ... }
```

23. [6 points] Which lines will compile without errors?

A. `Basket<Fruit> b2 = new Basket<Fruit>();`
B. `Basket<Apple> b3 = new Basket<Fruit>();`
C. `Basket<Apple> b6 = new Basket<?>();`

*Only A will compile without errors.*

24. [4 points] Does the code below compile? If so, explain any Java warnings, and explain what happens when it runs?

```java
Basket<Fruit> basket = new Basket<Fruit>();
basket.setElement(new Apple());
Apple apple = basket.getElement();
```

*Line 3 will not compile.*
Immutability [6 points]

Ben Bitdiddle wants to make an immutable list which takes in a List and stores it in a final variable as shown in the code below:

```java
public final class ImmutableList<T>{
    private final List<T> backingList;

    public ImmutableList(List<T> list){
        backingList = list;
    }
    public boolean equals(Object o) {
        return backingList.equals(o);
    }
    public int hashCode() {
        return backingList.hashCode();
    }
    public T get(int index) {
        return backingList.get(index);
    }
    public int size() {
        return backingList.size();
    }
}
```

Unfortunately, his Immutable list actually is mutable. Allysa P. Hacker suggests he change his constructor to:

```java
public ImmutableList(List<T> list){
    backingList = new ArrayList<T>(list);
}
```

25. Is Ben’s list now immutable? Explain why or why not. If it is not, suggest a fix to make the list immutable.

Ben’s list is still mutable. T, i.e., the elements of the list themselves could be mutable. We need to make a deep copy.
Visitor Pattern [15 points]

Ben is doing Lab 2 for 6.005. He is given an Expr class, and its subclasses Const, Plus and Minus:

```java
public abstract class Expr {
    /* Empty Class */
}
public class Const extends Expr {
    private final double value;
    public Const(double value){
        this.value = value;
    }
    public double getValue() {
        return value;
    }
}
public class Plus extends Expr {
    private final Expr left, right;
    public Plus(Expr left, Expr right){
        this.left = left;
        this.right = right;
    }
    public Expr getLeft() {
        return left;
    }
    public Expr getRight() {
        return right;
    }
}
```

Minus has the exact same internal definitions as the one shown above for Plus.

Ben decides to implement the visitor pattern for expressions. He creates the following visitor interface:

```java
public interface ExprVisitor<T>{
    public T visit(Const c);
    public T visit(Plus plus);
    public T visit(Minus minus);
    public T visit(Expr expr);
}
```

Instead of implementing the `accept(...)` method in each of the subclasses of Expr, Ben decides to save time and implement the `accept(...)` method in the top-level Expr class. This way, all the subclasses of Expr don’t need to have an `accept(...)` method. Here is his implementation:
public abstract class Expr {
    public <T> T accept(ExprVisitor<T> visitor) {
        return visitor.visit(this);
    }
}

26. [5 points] To get this method to compile, Ben has already added one extra visit(...) method to his ExprVisitor interface that would normally not be there. Which one is it, and why did he need to add it?

He added the visit(Expr) method.

The declared type of `this’ in Expr is Expr; The Java compiler checks for type safety and requires that a visit(X) method be present in the visitor interface, where X is Expr or a super-type of Expr.

27. [5 points] Ben argues that his implementation works, because the visit(Expr) method will never get called. He asserts that the accept method for Plus would call the visitor’s visit(Plus) method automatically, that the accept method for Minus would call the visitor’s visit(Minus) method automatically, and so on. Is Ben’s reasoning correct? Explain.

Ben is wrong.

visit(Expr) is always called. This is because Java decides which overloaded method to call at compile-time; it thus uses the declared type of the arguments(s) to decide which method gets called. `this’ refers to an Expr declared type in Expr, so we always call visit(Expr) in an accept(...) method.
28. [5 points] Now, assume that Ben has fixed the issues (if any) in his implementation of the visitor pattern for expressions, and he wants to create an expression and apply a visitor on it. Also assume that Ben has already implemented EvaluateExprVisitor. See the code excerpt below:

```java
Expr e = new Plus (new Const(1.0), new Const(2.0));
ExprVisitor<Double> evaluator = new EvaluateExprVisitor();

Double result = /* Your Code Here */
```

Complete the code here by using `evaluator` to set `result` to be equal to the computed value of the expression `e`.

```
e.accept(evaluator);

evaluator.visit(e) fails because e is declared to be Expr.

Evaluator.visit((Plus) e) fails to use the elegance of double dispatch; typecasts in general should be/can be avoided with visitor patterns, and this typecast depends on e being a Plus instance.
```
Equality and Comparable [15 points]

This is a difficult question. Attempt it last.

The Comparable<T> interface imposes a total ordering on the objects of each class the implements it; it allows them to be sorted by Arrays.sort and Collections.sort, and be used as elements in a SortedSet, such as TreeSet.

The interface specifies only one method: int compareTo(T o);
This method should return a negative integer, zero, or a positive integer if this is less than, equal to, or greater than the object o.

TreeSet uses compareTo to order its elements. It is strongly recommended, but not required, that this ordering be consistent with equals(); sorted sets do make this assumption when checking whether they contain an element. Consider the following class:

```java
public final class Card implements Comparable<Card> {
    private final int rank;
    private final int suit;
    // requires rank between 2 and 14 inclusive
    // requires suit between 0 and 3 inclusive
    public Card(int rank, int suit) {
        this.rank = rank;
        this.suit = suit;
        assert 2 <= rank && rank <= 14;
        assert 0 <= suit && suit <= 3;
    }
    @Override
    public int compareTo(Card o) {
        return rank - o.rank;
    }
    @Override
    public String toString() { ... }
}
```

29. [5 points] Imagine that you instantiate 52 card objects, for all possible ranks and suits, and add them to a set. What will be the set size if the set is a TreeSet? What will be the set size if it is a HashSet? Why? Recall HashSet uses hashCode internally.

For a TreeSet, the size is 13; for a HashSet, 52.
For a TreeSet, once one card of a rank has been added to the set, cards with the same rank will compareTo as 0, and not be added; there will only be 1 card of each rank.
For the HashSet, default implementations of equals and hashCode are used, making all cards not equals, and adding all of them.
30. [5 points] Create a short equals method that is consistent with the compareTo method above. What are the set sizes now? Why?

```java
public Boolean equals(Object o) {
    if (!(o instanceof Card)) return false;
    return rank == ((Card) o).rank;
}
```

_for TreeSet still 13; nothing has changed.
_for HashSet, likely 52, but possibly any value between 13 and 52.
If there are no hash collisions, each card has a distinct hashCode value,
and all of them will be added, even if some are equals. If a hash collision happens between 2 cards of the same rank only one of them will be added._

31. [5 points] Write or modify any required methods in Card to make the set hold all 52 cards, for either set, without violating any specifications.

2 possible solutions.

_A. Write compareTo to sort by rank and suit, use default equals and hashCode. This is not recommended, and could allow a HashSet to keep 2 cards of the same rank and suit, but it is valid._

_B. Write compareTo to sort by rank and suit, use equals that compares rank and suit, override hashCode._

**Sample implementations:**

```java
@override
public int compareTo(Card o) {
    return (4 * rank + suit) - (4 * o.rank + o.suit);
}

@override
public Boolean equals(Object o) {
    if (!(o instanceof Card)) return false;
    return compareTo((Card) o) == 0;
}

@override
public int hashCode() {
    return rank;
}
```