1. Consider the following code (Oracle):

```
public class Bucket{
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```

Which of the following is an implementation of a generic version of this class that doesn't have compiler warnings or errors?

a.
```
public class Bucket<> {

    private Object t;

    public void set(Object t) { this.t = t; }
    public Object get() { return t; }
}
```

b.
```
public class Bucket<T> {

    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

c.
```
public class Bucket<Object> {

    private Bucket<Object> t;

    public void set(Bucket<Object> t) { this.t = t; }
    public Bucket<Object> get() { return t; }
}
```

d.
```
public class Bucket<T> {

    private Object t;

    public void set(Object t) { this.t = t; }
    public Object get() { return t; }
}
```

e.
```
public class Bucket <T> {

    private <T> t;
```

```
    public void set(<T> t) { this.t = t; }
    public <T> get() { return t; }
}
```

2. Which of the following is an instantiation of the `Bucket` class from number 1 for a bucket of integers that will not lead to compiler warnings or errors?
   a. `Bucket<T> integerBucket = new Bucket<Integer>();`
   b. `Bucket<Integer> integerBucket = new Bucket();`
   c. `Bucket<Object> integerBucket = new Bucket<Integer>();`
   d. `Bucket<Integer> integerBucket = new Bucket<Integer>();`
   e. `Bucket<> integerBucket = new Bucket<Integer>();`

3. Which of the following is **not allowed** with Java generics?
   a.
   ```
   List<String> l1 = new List<String>();
   ArrayList<String> l2 = (ArrayList<String>)l1;
   ```
   b.
   ```
   public static void rtti(List<?> list) {
       if (list instanceof ArrayList<?>) {         // ...   }
   }
   ```
   c.
   ```
   List<Integer>[] arrayOfLists = new List<Integer>[2];
   arrayOfLists.add(1);
   ```
   d.
   ```
   public class Parser<T extends Exception> {
       public void parse(File file) throws T {     // ...   }
   }
   ```
   e.
   ```
   public static <E> void append(List<E> list, Class<E> cls) throws
   Exception {
       E elem = cls.newInstance();
       list.add(elem);
   }
   ```

4. Which of the following is a proper **explicit** invocation of the generic method `compare(Pair<K, V> p1, Pair<K, V> p2)`?
   a. `boolean same = Util.compare<Integer, String>(p1, p2);`
   b. `boolean same = Util.compare(p1, p2);`
   c. `boolean same = Util.<Integer, String>compare(p1, p2);`
   d. `boolean same = Util.<T, T>compare(p1, p2);`
   e.
   ```
   boolean same = Util.compare((Pair<Integer,String>)p1,
   (Pair<Integer,String>)p2);
   ```
5. When using type inference, which of the following will compile without warning or errors?
   a. `Map<String, List<String>> myMap = new HashMap();`

b. `Map<String, List<String>> myMap = new HashMap<>();`

c. `Map myMap = new HashMap<String, List<String>>();`

d. `Map<T, V> myMap = new HashMap<String, List<String>>();`

e. `Map<> myMap = new HashMap<String, List<String>>();`

6. What change(s) needs to be made to properly bind the generic type parameter `U` to `String` in the following code:

```
public <U> void inspect(U u){ ... }
```

a. `public <U> void inspect((String)U u){ ... }`
b. `public <U implements String> void inspect(U u){ ... }`
c. `public <U> void inspect(String u){ ... }`
d. `public <String> void inspect(U u){ ... }`
e. `public <U extends String> void inspect(U u){ ... }`

7. Consider the following code:

```
public class Node<T> {

    private T data;
    private Node<T> next;

    public Node setData(T data, Node<T> next) }
        this.data = data;
        this.next = next;
    }

    public T getData() { return data; }

    public static <T extends Bucket> void draw(T bucket) { ... }
}
```

Which of the following is how the code will look, to the Java compiler, after it is compiled?

a.
```
public class Node {

        private Object data;
        private Node next;

        public Node setData(Object data, Node next) { ... }

        public Object getData() { return data; }

        public static void draw(Bucket bucket) { ... }
    }
```

b.
```
public class Node {

    private Object data;
    private Node next;

    public Node setData(Object data, Node next) { ... }

    public Object getData() { return data; }

    public static void draw(Object bucket) { ... }
}
```

c.
```
public class Node<> {

    private Object data;
    private Node<> next;

    public Node setData(Object data, Node<> next) { ... }

    public Object getData() { return data; }

    public static <Object extends Bucket> void draw(Object
bucket) { ... }
}
```

d.
```
public class Node {

    private ? data;
    private Node<?> next;

    public Node setData(? data, Node<?> next) { ... }

    public ? getData() { return data; }

    public static <? extends Bucket> void draw(? bucket) { ... }
}
```

e. It looks the same. The code doesn't change until runtime.

8. "Is a" relationships in Java mean that because `Integer` *is a* `Object` you can assign an `Integer` to an `Object` (`someObject = someInteger;`). The same can be said for the relationship between `Integer` and `Number`.

   Now, consider the following code:

```
public void shapeTest(Shape<Number> n) { /* ... */ }
```

Based on the signature of this method, could you pass in `Shape<Integer>` and/or `Shape<Object>`?

a. Yes, you could pass in both.
b. No, you can't pass in either.
c. You can pass in Shape<Integer> but not Shape<Object>.
d. You can pass in Shape<Object> but not Shape<Integer>.
e. You can pass in either if you cast it to Shape<Number>.

9. Consider the following code:

```
List list = new ArrayList<>();
Collections.sort(list);
```

Upon compilation, you get the following warning:
Type safety: Unchecked invocation max(List) of the generic method max(Collection<? extends T>) of type Collections.

What change needs to be made to resolve this warning and prevent others?

a.
```
List list = new ArrayList<String>();
Collections.sort(list);
```
b.
```
List list = new ArrayList();
Collections.sort(list);
```
c.
```
List<> list = new ArrayList<>();
Collections.sort(list);
```
d.
```
List<String> list = new ArrayList();
Collections.sort(list);
```
e.
```
List<String> list = new ArrayList<String>();
Collections.sort(list);
```

10. Consider the following code:

```
StringBuilder myText = new StringBuilder();
List<String> myList = new ArrayList<String>();
boolean containsMyText = myList.contains(myText);
```

After running static analysis, you get the following notification:

Bug: StringBuilder is incompatible with expected argument type String in new util.Configuration(String, String)

This call to a generic collection method contains an argument with an incompatible class from that of the collection's parameter (i.e., the type of the argument is neither a supertype nor a subtype of the corresponding generic type argument). Therefore, it is unlikely that the collection contains any objects that are equal to the method argument used here. Most likely, the wrong value is being passed to the method.

In general, instances of two unrelated classes are not equal. For example, if the Foo and Bar classes are not related by subtyping, then an instance of Foo should not be equal to an instance of Bar. Among other issues, doing so will likely result in an equals method that is not symmetrical. For example, if you define the Foo class so that a Foo can be equal to a String, your equals method isn't symmetrical since a String can only be equal to a String.

In rare cases, people do define nonsymmetrical equals methods and still manage to make their code work. Although none of the APIs document or guarantee it, it is typically the case that if you check if a Collection<String> contains a Foo, the equals method of argument (e.g., the equals method of the Foo class) used to perform the equality checks.

Which of the following will **not** resolve the notification?

a.
```
String myText = “my text”;
...
List<String> myList = new ArrayList<String>();
boolean containsMyText = myList.contains(myText);
```
b.
```
StringBuilder myText = new StringBuilder();
...
List<String> myList = new ArrayList<String>();
boolean containsMyText = myList.contains(myText.substring(0));
```
c.
```
StringBuilder myText = new StringBuilder();
...
List<String> myList = new ArrayList<String>();
boolean containsMyText = myList.contains(myText.toString());
```
d.
```
StringBuilder myText = new StringBuilder();
...
List<StringBuilder> myList = new ArrayList<StringBuilder>();
boolean containsMyText = myList.contains(myText);
```
e.
```
StringBuilder myText = new StringBuilder();
...
List<String> myList = new ArrayList<String>();
boolean containsMyText = myList.contains((String)myText);
```