Segment 1: Exercise in Exception

(a) Try the code below. Does it compile?

```
// Fix this
import java.io.File;
import java.util.Scanner;

class ExceptionDemo {
  public static void main(String[] args) {
    File f = new File("hello.txt");
    Scanner s = new Scanner(f);
  }
}
```

Look up the `Scanner` API and show that a `FileNotFoundException` might be thrown.

Fix the code so that it compiles.

```
// Answer
import java.io.File;
import java.util.Scanner;
import java.io.FileNotFoundException;

class ExceptionDemo {
  public static void main(String[] args) {
    File f = new File("hello.txt");
    try {
      Scanner s = new Scanner(f);
    } catch (FileNotFoundException e) {
      // do something
    }
  }
}
```

(b) What about:

```
// Fix this
import java.io.File;
import java.util.Scanner;

class ExceptionDemo {
  public static Scanner openFile(String filename) {
    File f = new File(filename);
    return new Scanner(f);
  }
  public static void main(String[] args) {
    Scanner sc = openFile("hello.txt");
  }
}
```

Now, you can either throw/catch inside `openFile`, or throw it from openFile for the main to catch. Write both versions.

```
// Answer
import java.io.File;
import java.util.Scanner;
import java.io.FileNotFoundException;

class ExceptionDemo {
  public static Scanner openFile(String filename) throws FileNotFoundException {
    File f = new File(filename);
    return new Scanner(f);
  }
  public static void main(String[] args) {
    try {
      Scanner sc = openFile("hello.txt");
    } catch (FileNotFoundException e) {
      // do something
    }
  }
}
```

```
// Fix this
import java.io.File;
import java.util.Scanner;
import java.io.FileNotFoundException;

class ExceptionDemo {
  public static Scanner openFile(String filename) {
    File f = new File(filename);
    try {
      return new Scanner(f);
    } catch (FileNotFoundException e) {
      return null;
    }
  }
  public static void main(String[] args) {
    Scanner sc = openFile("hello.txt");
    // need to handle null
  }
}
```

(c) Create a new checked exception `MyOwnException` and throws it from the method foo for main and catch it in main.

```java
// Fix this
class ExceptionDemo {
  public static void foo() {
    // throw MyOwnException
  }
  public static void main(String[] args) {
    foo();
  }
}
```

Answer:

```java
// Answer
class MyOwnException extends Exception {

}
class ExceptionDemo {
  public static void foo() throws MyOwnException {
    throw new MyOwnException();
  }
  public static void main(String[] args) {
    try {
      foo();
    } catch (MyOwnException e) {
      // do something
    }
  }
}
```

Segment 2: Exercise in Generics

(a) Declare a generic class with type parameter T and a single private field x of type T.

```java
// Answer
class A<T> {
  private T x;
}
```

(b) Now instantiate an instance of `A<T>` with type argument `Integer`

```java
// Answer
A<Integer> a = new A<Integer>();
```

Remember not to use raw types (`new A()`). This is one of the major sources of mark deductions in labs and PEs. From now on, we always use the `-Xlint:rawtypes` flags to compile our code.

(c) Declare a generic class `B<T>` that extends from `A<T>`, and a generic class `C<T>` that contains a field of type `A<T>`. Are the occurrences of `T` refers to the same `T`? (answer: yes)

```
// Answer
class B<T> extends A<T> {
}
class C<T> {
  private A<T> a;
}
```

(d) What is wrong with the following

```
// Answer
class F extends A<T> {
}
// F must extend an instantiated type of A
// In this case, A<T> is not an instantiated type since the
// type parameter T is not provided by F
//
```

Change the generic class `A<T>` to parameterized type with type argument `String`. Does it work now? (yes)

```
// Answer
class F extends A<String> {
}
```

(e) Write a generic method that copy from one array to another. Here is the skeleton:

```
// Skeleton
class A {
 public static ??? void copy(???  from, ??? to) {
   for (int i = 0; i < from.length; i++) {
     to[i] = from[i];
   }
 }
}
```

Here is how the generic method is supposed to work:

```
String s[] = new String[2];
String t[] = new String[2];
```

```
Integer i[] = new Integer[2];
Integer j[] = new Integer[2];

A.<String>copy(s, t); // ok
A.<Integer>copy(i, j); // ok
A.<String>copy(i, j); // error
A.<String>copy(s, j); // error
```

```
// Answer
class A {
 public static <T> void copy(T[] from, T[] to) {
   for (int i = 0; i < from.length; i++) {
     to[i] = from[i];
   }
 }
}
```

(f) (The next two are useful for Lab 3 -- don't skip)

Write a generic class `D<T>` that contains a field that is an array of type `T` with 10 elements. Instantiate that array in the constructor.

```
// Answer
class D<T> {
  T[] a;
  D() {
    @SuppressWarnings("unchecked")
    T[] tmp = (T[]) new Object[10];
    this.a = tmp;
  }
}
```

(g) Write a generic class `E<T extends Comparable<T>>` that contains a field that is an array of type `T` with 10 elements. Instantiate that array in the constructor.

This is the first time students see (i) use of raw type in the context of array allocation; (ii) use of @SuppressWarnings("rawtypes"); (iii) use of multiple attributes in @SuppressWarnings.

```
// Answer
class E<T extends Comparable<T>> {
  T[] a;
  E() {
    @SuppressWarnings({"unchecked", "rawtypes"})
    T[] tmp = (T[]) new Comparable[10];
    this.a = tmp;
  }
}
```

Note that this is another situation where raw types are OK (for now, until we teach wildcards). Here, we suppress the "rawtypes" warning in addition to unchecked warnings.