CS2030S

AY23/24 S2 **Lab Sheet for Week 5** February 2024

1. **Exception**

   (a) Try to compile the code below and note the compilation error.

   Look up the `Scanner` API to find out more about the exception thrown.

   Fix the code so that it compiles.

```
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);
  }
}
```

Notes for Tutors:

```
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) The code below does not compile as well.

```
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");
  }
}
```

Make the code compile in two different ways: (i) handle the exception inside the method `openFile`; and (ii) throw the exception to `main` and handle the exception in `main`.

Notes for Tutors:

```
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
  }
}

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
    }
  }
}
```

(c) Create your own checked exception called `MyOwnException` and throw it from the method `foo` to `main`. Handle the exception in `main`.

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

Notes for Tutors:

```
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
    }
  }
}
```

2. **Generic Type**

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

Notes for Tutors:

```
class A<T> {
  private T x;
}
```

(b) Instantiate an instance of `A<T>` with type argument `Integer`.

Notes for Tutors:

```
A<Integer> a = new A<Integer>();
```

(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>`. Do the occurrences of `T` refer to the same `T`?

Notes for Tutors:

```
class B<T> extends A<T> {
}
```

```
class C<T> {
  private A<T> a;
}
```

(d) Continuing with the classes defined above. Is there anything wrong with the following?

```
class F extends A<T> {
}
```

What if we replace `T` with `String`?

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

Notes for Tutors:

The type argument `T` is not defined (it was previously declared but only within the scope of the class it is declared with). Replacing it with `String` is OK since `String` is an existing class and we are passing `String` as a type argument.

3. **Generic Method**

Write a generic method that copies from one array to another.

```
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
```

Notes for Tutors:

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

4. (This question is particularly helpful for Exercise 3)

   (a) 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.

   Notes for Tutors:

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

   (b) Write a generic class `E<T>` that contains a field that is an array of type `T` with 10 elements, but `T` must be a subtype of `Comparable<T>`. Instantiate that array in the constructor.

Note that, until you learn about wildcards, you need to initialize the array as a rawtype. For now, you can use `@SuppressWarnings({"unchecked", "rawtypes"})` to suppress both warnings.

Warning: This is a legit case for using the annotation `@SuppressWarnings`. You must not abuse this annotation in other situations.

Notes for Tutors:

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