# CS1102: Data Structures and Algorithms

## Tutorial 1 – Java and Problem Solving (Solution)

(28, 29 January 2010)

1. [**Concept recap**] What is the difference between the following keywords and concepts? Please provide examples to help to explain.

   a. `throw` and `throws`

   **Answer**:
   `throw` triggers the exception.
   `throws` declares that the method would possibly throw exceptions.

   This is how you create your own exception (see also slide 44 in Lecture 1). In this example, the `DivideByZeroException` is a more specialized case of `Exception` (hence the word extends). The keyword super calls the constructor of class `Exception` with the desired message.

```
class DivideByZeroException extends Exception{
     public DivideByZeroException(String message){
          super(message);
     }
}
…

static int divide (int a, int b) throws
DivideByZeroException {
if (b == 0)
     throw new DivideByZeroException("My own exception");
     return a/b;
}
```

   b. ArrayList and Vector

   **Answer**:
   There are several java classes that seem to do the same thing: Array, ArrayList, Vector. ArrayList is an ordered collection, pretty much like a Vector. An ArrayList is better than an Array if you do not have knowledge in advance about element number. Both ArrayList and Vector classes keep an inner Array with an initial size of 10.

   A Vector is a synchronized, making it safe to use in a threaded program. ArrayLists are not synchronized, making them not thread safe. However, **using synchronization will incur a performance hit**. As such, unless synchronization is needed, the use of ArrayList is recommended.

When a Vector array fills with elements, it will increase its available size by doubling the size of its inner Array. An ArrayList will increase its array size by 50%. If you do not know what data size you have, but you know its growth rate, Vector has a slight advantage in that you can set the rate with which it grows. It is always good practice to set the object's initial capacity to the largest capacity that your program will need.

c. Auto-boxing and auto-unboxing. Which one of lines 1, 2, 3, and 4 are examples of auto-boxing and auto-unboxing?

```
import java.util.*;

// Prints a frequency table of the words on the command line
public class Frequency {
   public static void main(String[] args) {
    HashMap<String, Integer> m = new HashMap<String, Integer>();  //1
      for (String word : args) {
          Integer freq = m.get(word);                             //2
          m.put(word, (freq == null ? 1 : freq + 1));             //3
            System.out.println(freq);                             //4
      }
      System.out.println(m);
   }
}
```

**Answer:**
For each of the primitive types in Java, there exist object equivalents to them, such as Integer – int , Float – float, Double – double, etc.
Auto-boxing means converting from a primitive type to its object equivalent. For example when you write
`Integer i1 = 5;` instead of
`Integer i1 = new Integer(5);`

Auto-unboxing is the reverse of auto-boxing, meaning the conversion from the object to its primitive equivalent, like when you write:

```
int i = 0;
i = new Integer(5);
```

The above program computes and prints an alphabetized frequency table of the words appearing on the command line.

Line 1 contains the definition of a HashMap, which is a collection in which <key,value> pairs can be stored, with an unique key.

Line 2 retrieves the number of occurences of the word key in the HashMap.

Line 3 contains both auto-unboxing and auto-boxing. At first, freq is auto-unboxed to add 1 to it – `freq + 1`. However, in order to put the computed value of the expression `freq == null ? 1 : freq + 1` which results in an `int` value, the value must be auto-boxed into an `Integer` object.

Line 4 is not an example of auto-unboxing. On this line, the `toString` (lecture 1, slide 117) method of object freq is called.

2. [**Common issues**] What is the output of the following code?

```
a. int x = 1;
   Integer y = 1;
   x = y++ + ++y;
   System.out.println(x);   // (Answer: 4)

   int z = x > y++ + ++y ? x++ + ++x : ++x + y;

   System.out.println(x);   // (Answer: 5)

   System.out.println(y);   // (Answer: 5)

   System.out.println(z);   // (Answer: 10)
```

**Note**: the equivalent java statements for `x = y++ + ++y` are:

```
        int y1 = y++;   // y1 = 1 and y = 2
        int y2 = ++y;   // y2 = 3 and y = 3;
        x = y1 + y2;    // x = 1 + 3 = 4
```

and the equivalent java statements for

```
   int z = x > y++ + ++y ? x++ + ++x : ++x + y;
```

are:

```
        int y1 = y++;   //y1 = 3 and y = 4;
        int y2 = ++y;   // y2 = 5 and y = 5;
        int y3 = y1 + y2;    // y3 =  8 and y =5;
        int z = 0;


        if (x > y3){    //if (4 > 8)
        int x1 = x++;   //x1 = 4 and x = 5;
        int x2 = ++x;   //x2 = 6 and x = 6;
        z = x1 + x2;    //z = 10 and x = 6 and y = 5;
}
        else{
        int x1 = ++x;   //x1 = 5 and x = 5;
        z = x1 + y;              //z = 10 and x = 5 and y = 5;
}
```

We should avoid overly complicated expressions that might have subtle side effects. This is something not often mentioned as a good programming practice, but it is illustrated in this question. Furthermore, even if a portion of code is shorter, it does not necessarily mean that it is easier to read.

b.
```
Integer eye = 42;
Double d = 42.0;
int i = 42;
double dd = 42.0;
System.out.println(i == eye);       // (Answer: true)
System.out.println(i == d);         // (Answer: true)
System.out.println(eye == dd);      // (Answer: true)
System.out.println(d == dd);        // (Answer: true)
System.out.println(eye.equals(d));  // (Answer: false)
```

**Note**:
When comparing a primitive value (e.g. `int`, `double`) with objects of the wrapper class (e.g. `Integer`, `Double`), `==` evaluates the **value** equality. So, the first four answers are true.

The `equals` method of `Integer` class first checks whether the passed object is not null and is an `Integer` object, and then it checks whether they have same value. `eye.equals(d)` returns false because `d` is not an `Integer` object.

If you attempt `eye == d`, this will cause a compile error because they are objects of different types and thus are not comparable using ==. For objects of same type, == compares reference identity (whether they refer to the same object). Have a look at object comparison (Comparable interface), Lecture 1, slide 107.

c.
```
int []array = {1,2,3,4};
for ( int i : array ) {
   array[i] = 0;
}
for ( int i : array ) {
   System.out.print(i + “ “);
}
```

**Answer**: 0 0 3 0
The execution of the first for loop is:

```
                 // array is {1, 2, 3, 4}
i = array[0];    // i = 1;
array[i] = 0;    // array[1] = 0; and array is {1, 0, 3, 4}
i = array[1];    // i = 0;
array[i] = 0;    // array[0] = 0; and array is {0, 0, 3, 4}
i = array[2];    // i = 3;
array[i] = 0;    // array[3] = 0; and array is {0, 0, 3, 0}
i = array[3];    // i = 0;
array[i] = 0;    // array[0] = 0; and array is {0, 0, 3, 0}
```

We should be careful about modifying a collection when traversing it.

3. [**Static, scope**] Consider the following program. pack.java is stored in a directory *packed*, which has the same name as the package. test.java is stored in the directory that contains *packed*.

```
// ./packed/pack.java
package packed;
public class pack
{
     public int x1 = 1;
     protected static int x2 = 2;
     int x3 = 3;
     private int x4 = 4;
}
// end of file

// ./test.java
import packed.pack;
class test {
     private int x1 = 1;
     static int x2 = 2;

     public static void main( String args[] )
     {
          pack p = new pack();
          System.out.println( p.x1 );        //1
          System.out.println( p.x2 );        //2
          System.out.println( p.x3 );        //3
          System.out.println( p.x4 );        //4

          test t = new test();
          test t1 = new test();
          t.printSum(t1, p);
     }
     void printSum(test t, pack p)
     {
          System.out.println(this.x1 + t.x1);     //5
          System.out.println(this.x2 + p.x2);     //6
          System.out.println(test.x2 + pack.x2);  //7
               }
}
// end of file
```

Which ones of the seven marked lines are illegal? Why?

**Answer**:
Line 1 is legal because public variables can be accessed from outside the package.
Line 2, 3, 4 are illegal because protected, default and private member variables (i.e. p.x2, p.x3, and p.x4 resp.) cannot be accessed from outside package.
Line 2 would be legal if class test were a subclass of class pack and still be outside the package `packed`.
Line 5 is **legal** because x1 can be accessed within class test: both by `this.x1` and by `t.x1`. Although we are talking about two objects, `t` and `this`, we are still in a single class, `test`. Notice how the keyword `this` has been used to remove ambiguities. A sample use of this is the following

```
class test{ …
     private int x1 = 1;

     public void printSum(int x1){
          System.out.println(x1 + this.x1);
     }
     …
}
```

Line 6 and 7 are both illegal because x2 is declared as protected, so p.x2 and pack.x2 cannot be accessed from outside package. Nonetheless, it is recommended that you employ the practice shown in line 7, by accessing the static variable through the *class* identifier rather than the *object* identifier.

Note: If we add "package packed;" at the beginning of the file test.java and store it in the same directory, i.e. ./packed/test.java (so that pack and test are now in the same package) then only line 4 is illegal (since only x4 is declared as private).

Recap:

Table 1 shows the accessibility of data fields with different access modifiers, from different places.

Table 1. Accessibility of data fields

| Access Modifier | Same class | Same package | Subclass | Other package |
|---|---|---|---|---|
| public | Y | Y | Y | Y |
| protected | Y | Y | Y * | N |
| no modifier | Y | Y | N | N |
| private | Y | N | N | N |

Note: Y means accessible and N means not accessible.

`static` is a use modifier that can be applied to data fields. It indicates that only one such data field is available for all instances of this class. Without this modifier, each instance has its own copy of a data field.

4. [**Generics**] Consider the following classes:

```
public class AnimalHouse<E> {
     private E animal;

     public void setAnimal(E x) {
          animal = x;
     }

     public E getAnimal() {
          return animal;
     }
}

public class Animal{
}

public class Cat extends Animal {
}

public class Dog extends Animal {
}
```

For the following code snippets, identify whether the code fails to compile, compiles with a warning, generates an error at runtime, or none of the above. If there is an error or warning, explain why.

a. `AnimalHouse<Animal> house = new AnimalHouse<Cat>();`

b. `AnimalHouse<Dog> house = new AnimalHouse<Animal>();`

c.
```
AnimalHouse house = new AnimalHouse();
house.setAnimal(new Dog());
```

d.
```
AnimalHouse<?> house = new AnimalHouse<Cat>();
house.setAnimal(new Cat());
```

e.
```
AnimalHouse<Cat> house = new AnimalHouse<Cat>();
house.setAnimal(new Cat());
```

**Answer**:

**Line a**. fails to compile: `Cat` and `Animal` are incompatible types, even though `Cat` is a subtype of `Animal`.

**Line b**. fails to compile: `Dog` and `Animal` are incompatible types, even though `Dog` is a subtype of `Animal`.

**Line c.** compiles with a warning. The compiler does not know what type house will contain. It will accept the code but it warns that there might be a problem when setting the animal attribute to an instance of Dog.

A generic type without any type arguments is called a *raw* type. Using a generic type as a raw type (that is, removing the generics declaration) might be a way to work around a particular compiler error, but you lose the type checking that generics provides, so *it is not recommended*. This can happen when using an older API that operates on raw types.

**Line d**. fails to compile: The first line will compile – using the wildcard ?, it declares an instance of an unknown type. However, the type of the animal stored in house is unknown and hence the setAnimal method cannot be used.

**Line e** is correct and executes. This is because Cat is a subtype of Animal. However, you cannot assign a `AnimalHouse<SubtypeofAnimal>` to a `AnimalHouse<Animal>` because the two instantiations are unrelated and thus incompatible.