Lab 13 : Programming Contracts and Revision

Topics covered interfaces, abstract classes, revision

Tell us what you really think!

Please spend some time to complete the INFO1103 Unit of Study Survey (USS) survey found here:
http://sydney.edu.au/itl/surveys/complete/.

Any feedback you can give on the course is helpful. We read every single response! Please be specific about anything you think was good or bad about the course, and if you want to mention your tutor, remember to mention them by name. Your responses will be anonymous, and any feedback you can give (positive or negative) will help us in making the course better for future students.

Consider the following interface:

```
public interface Food {
   // returns whether or not the food is tasty
   public boolean isTasty();

   // weight of the food in kilograms
   public int getWeight();

   // when a food is consumed, its weight becomes zero
   // and it is no longer tasty.
   public void eat();
}
```

The following class represents your pet monster. The monster needs to be fed regularly, otherwise it will get angry and eat you.

```
public class Monster {
   private int weight; // weight in kilograms
   private boolean hungry;
   private boolean angry;

   // implement the constructor
   public Monster(int weight) { }

   // implement this method
   public void feed(Food food) { }

   public void pet() {
      if (hungry) {
         if (angry)
            System.out.println("Monster has eaten you");
         else
            System.out.println("Monster has bitten you");
      } else {
         if (angry)
            System.out.println("Monster is content");
         else
            System.out.println("Monster loves you");
      }
   }
}
```

Exercise 1: Implement the `Monster` constructor. A monster will start off not hungry or angry, and will have the given weight.

Exercise 2: Implement the `feed` method. When given an item of food, the following things happen in this order:

1. if the food is tasty, the monster is not angry
2. if the food weighs less than 10% of the weight of the monster, it is hungry
3. the food is eaten (`.eat()` called)
4. if the monster is hungry after feeding it becomes angry

Exercise 3: Write a class `Sandwich`, which `implements Food`. A sandwich has a weight (in kilograms – yes, these are monster-sized sandwiches), as well as an `age` in hours. A sandwich is tasty, unless it was made more than 48 hours ago.

Remember you'll need to implement the three methods from the `Food` interface for this to compile.

Try feeding the sandwich to the monster. See how it reacts to being fed the same sandwich twice.

Exercise 4: Write another class of your choosing to represent another type of food. This food should have different rules for when it is tasty (not age, like the sandwich).

Extension 1: Create an `ArrayList` of type `Food`, and add some food to it. Create a couple of sandwiches and a couple of your second food, and put them all in the `ArrayList`. Try writing a loop to iterate over the food and feed each item to the monster.

Extension 2: Is there any common code between your food classes? If so, consider how you would turn the `Food` interface into an `abstract` class, so that the common code is just written in one place.

Revision Summary

The following is a quick summary of most of the topics we have learned in this course. This list is not a comprehensive list of everything you need to know, but will cover many of the main topics.

Variables Variables store values in memory. The syntax for **declaring** a variable in Java is: `<type> <name>;` where `<type>` is the data type, and `<name>` is the name of the variable.

You need to **initialise** a variable before you use it; this means you give it an initial value. You can initialise a variable on the same line as declaration: `<type> <name> = <value>;`, or you can declare the variable and then initialise it later.

Examples:

```
//variable declarations:
int count;
String myName;
Object data;

//variable initialisation
char ch = 'd';
double sum = 0;
String category = "sample";
```

Choosing data types When creating a variable, it is important to make sure that the data type of the variable matches the kind of information you want to store. Consider each of the **primitive** data types, and whether they will be suitable to represent the information that you need to store.

Operators and expressions In order to perform ”operations” on variables, you need to use **operators**.

- Assignment: `=`
- Mathematical (performed on numbers; resulting in a number): `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), `%` (modulus)
- Relational (performed on numbers; resulting in a boolean value): `<` (less than), `<=` (less than or equal to), `>` (greater than), `>=` (greater than or equal to), `==` (equal to), `!=` (not equal to)
- Logical (performed on boolean values; resulting in a boolean value): `&&` (and), `||` (or), `!` (not)

Examples:

```
//mathematical operators:
int count = 4 + 3;
count = count - 1;
count = count * 2;
count = count / 3;
count = count % 5;

//relational operators:
int a = 5;
int b = 3;
boolean lt  = a < b;
boolean lte = a <= b;
boolean gt  = a > b;
boolean gte = a >= b;
boolean eq  = a == b;
boolean neq = a != b;

//logical operators:
boolean t = true;
boolean f = false;
boolean both = t && f;
boolean either = t || f;
boolean neither = !either;
```

`if` statements An if statement has the following syntax, where `<condition>` is a boolean expression:

```
if(<condition>) {
   // when condition is true
} else {
   // when condition is false
}
```

The `else` block is optional, and you can chain if statements like this:

```
if(<condition1>) {
   // when condition1 is true
} else if(<condition2>) {
   // when condition2 is true
} else {
   // when neither condition is true
}
```

Loops When you need to repeat the same code multiple times, you can use a loop. A `while` loop has the following syntax:

```
while(<condition>) {
   // keep doing this as long as condition is true
}
```

A `while` loop will continue to execute as long as the `<condition>` evaluates to true.

Example:

```
int count = 0;
while(count < 10) {
   System.out.println(count);
   count = count + 1;
}
```

If you need to perform an action a certain number of times, you might use a `for` loop with the following syntax:

```
for(<initialisation>; <condition>; <update>) {
   <body>
}
```

This is exactly the same as the following `while` loop:

```
<initialisation>;
while(<condition>) {
   <body>
   <update>;
}
```

Example:

```
for(int count = 0; count < 10; count = count + 1) {
   System.out.println(count);
}
```

Using the `break` keyword inside either loop will exit the loop early, and resume execution after the end of the loop. Using the `continue` keyword will exit *this iteration* of the loop, and will jump back to the top of the loop for the next iteration.

Methods A method consists of a method **prototype** or **header**, and a **body**. The syntax is as follows:

```
<modifiers> <type> <name>(<parameters>) {
   <body>
}
```

Here, `<modifiers>` is a list of keywords. They may be an access specifier (`public` or `private`), or the `static` keyword. A static method is one that does not need an instance of a class in order to be called; hence is called using the class name (e.g. `ClassName.someStaticMethod()`).

`<type>` is the return type of the method. If the return type is `void`, the method does not return any information, and is likely designed to just perform an action (or set of actions). If the return type is anything else (for example `int`), you need to include a `return` statement inside the method body, indicating that the result of the method is the value returned.

Remember: most methods are designed to calculate and return a value, rather than print out something. If a method has a non-`void` return type, it is likely that the method should not print anything.

`<name>` is the name of the method, and `<parameters>` is a comma-separated list of variable declarations.

Examples:

```
public static int getSum(int a, int b) {
   return a + b;
}

public static void printSquareRoot(double value) {
   double root = Math.sqrt(value);
   System.out.println(root);
}

public static String getYesNoString(boolean test) {
   if(test) {
      return "yes";
   }
   return "no";
}
```

Arrays The syntax for creating a blank array is `<type>[] <name> = new <type>[<size>];`, where `<type>` is the data type of the array, and `<size>` is the number of positions in the array (i.e. its length).

When you create an array like this, the smallest index you can access is 0, and the largest is `<size> - 1`. If you try to access an index outside this range, you get an `ArrayIndexOutOfBoundsException`.

Examples:

```
// creating arrays
String[] values = new String[5]; // values.length is 5
int[] data = {1, 2, 3}; // data.length is 3

// setting array content
values[4] = "hello";
data[0] = 3;

// accessing array content
String firstValue = values[0];
System.out.println(data[1] + 3);
```

Black box testing is testing of a method or section of code where you do not know the inner workings of the code. Values should be tested to ensure that given a certain input, you get the expected output (as this is generally all you know).

White box testing is where you do know the inner workings of the code you are testing. Values should be tested to follow each possible execution path, with a focus on **edge cases** and **border cases**.

Regression testing is where you create a set of tests that are run every time you make a change to the code. This ensures that changes to the code do not break previously-working sections of code.

Preconditions are conditions or assumptions that must be true `before` a section of code is executed. If a precondition is not met before the code is execution, the code is likely to have an error or throw an exception.

Postconditions are conditions or assumptions that must be true `after` a section of code is executed.

Class invariants are facts about a class that are true at all times. These are often just logical facts, such as knowing that a person cannot at any point have a negative height.

Exceptions Any time that a problem occurs, and that problem needs to be dealt with (or it isn't possible to recover from), an exception is thrown. To throw an exception: `throw new Exception("message");`

Exceptions must be handled. They can either be thrown further up the stack, or they can be dealt with using a `try/catch` block:

```
public static double calculateAverage(double[] values)
   throws ArithmeticException {

   if(values.length == 0} {
      throw new ArithmeticException("Cannot find average of 0 values");
   }
   // More code for calculating average...
}

public static void main(String[] args) {
   double data = new data[0];
   try {
      double average = calculateAverage(data);
   } catch (ArithmeticException e) {
      System.out.println(e.getMessage());
   }
}
```

In this example, the `ArithmeticException` in the `calculateAverage` method is thrown. It must either be dealt with immediately (which would be a bit pointless), or thrown further up the execution stack for "someone else" to deal with.

To do the latter, `throws ArithmeticException` is added to the method prototype. Now it must be handled by any code that calls the `calculateAverage` method; in this case, that is our main method.

File Input and Output Reading from a file is easiest using a `Scanner`. The three main steps to reading a file are:

1. Open the file for reading with a `Scanner`.
2. Read the contents of the file using the `next___()` methods of the scanner.
3. Close the file.

Reading from a file example:

```
String filename = "input.txt";
Scanner scan = null;
try {
   scan = new Scanner(filename);
} catch(FileNotFoundException e) {
   System.out.println("Could not find file.");
   System.exit(0);
}

while(scan.hasNext()) {
   String line = scan.nextLine();
   System.out.println(line);
}

scan.close();
```

Writing to a file is easiest using a `PrintWriter`. The three main steps to writing to a file are:

1. Open the file for writing with a `PrintWriter`.
2. Print content to the file using the `print()` methods of the writer.
3. Close the file.

Writing to a file example:

```
String filename = "output.txt";
PrintWriter writer = null;
try {
   writer = new PrintWriter(filename);
} catch(FileNotFoundException e) {
   System.out.println("Could not write to file.");
   System.exit(0);
}

for(int i = 0; i < 10; i = i + 1) {
   writer.println(i);
}

writer.close();
```

Classes define a "blueprint" for what an object is and what it can do. A class is used to make an **instance** of that class. The new instance is called an **object**. A class definition generally consist of four main parts:

Instance variables are variables outside of any methods, but inside the class. They should be declared as either `public` or `private`, depending on whether any other classes need direct access to the variables.

Instance variables – just like any other variables – need a data type appropriate to what information they are going to represent.

Constructor(s) are special "methods" that are called when the `new` keyword is used. Syntactically, they must have the same name as the class, and do not have a return type (not even `void`); however are otherwise exactly the same as normal methods. The constructor is usually used for passing the instance its starting instance variable values.

Two standard constructor types are the "default constructor", which takes no arguments and initialises each of the instance variables to some default value; and a constructor that takes in one value for each of the assignable instance variables, and stores those values in the appropriate instance variables.

Get and set methods to provide access to any private instance variables.

Other methods can be added that allow special actions or functions to be performed by the instance. These methods can be marked `static` to imply that all instances of that class will behave in exactly the same way; i.e. that no matter what instance you call the method on, it will give you the same result.

Non-static methods (i.e. instance methods) must be invoked by having an instance of the class already, and then using the syntax `instance.someInstanceMethod()`. Static methods (i.e. class methods) do not require an instance, and are invoked through the class: `TheClass.someStaticMethod()`.

A simple class:

```
public class Person {
   // instance variables
   private String name;
   private int age;

   // constructor
   public Person(String initName, int initAge) {
      this.name = initName;
      this.age = initAge;
   }

   // set and get method for name
   public void setName(String newName) {
      this.name = newName;
   }
   public String getName() {
      return this.name;
   }

   // get method for age
   public int getAge() {
      return this.age;
   }

   // extra method for getting older
   public void incrementAge() {
      this.age = this.age + 1;
   }
}
```

An example of using the `Person` class:

```
public class PersonTester {
   public static void main(String[] args) {
      Person p1 = new Person("Barry", 42);
      p1.setName("Baz");
      p1.incrementAge();

      int age = p1.getAge(); // gets 43
      String name = p1.getName(); // gets "Baz"

      Person p2 = p1;
      p2.setName("Cheryl");

      name = p2.getName(); // gets "Cheryl"
      name = p1.getName(); // also gets "Cheryl"
   }
}
```

Idioms A very common problem is to iterate through an array:

```
int[] data = {1, 2, 3, 4, 5};
for(int i = 0; i < data.length; i++) {
   int value = data[i];
   // do something with value
}
```

This is often to search for something. If you are searching for a specific item, then you can return the item straight away, as soon as you find it:

```
public static Person findCheryl(Person[] people) {
   for(int i = 0; i < people.length; i++) {
      Person thisPerson = people[i];
      if(thisPerson.getName().equals("Cheryl") {
         // found Cheryl!
         return thisPerson;
      }
   }
   // didn't find Cheryl...
   return null;
}
```

If you are searching for the largest/smallest/some other non-exact item, you will need to look through the whole array, and remember the most appropriate answer you've found so far:

```
public static Person findOldest(Person[] people) {
   Person oldestPerson = null;
   Person highestAge = -1;
   for(int i = 0; i < people.length; i++) {
      Person thisPerson = people[i];
      if(thisPerson.getAge() > highestAge) {
         // found an older person, so maintain highest age so far,
         // and remember that this was the oldest person
         highestAge = thisPerson.getAge();
         oldestPerson = thisPerson;
      }
   }
   return oldestPerson;
}
```

Inheritance Inheritance is used when you need two or more classes to share some similar code, but also have some differences. These differences may be in what variables are stored in the class, or they may be the way that certain actions are performed by the class.

When one class `Child` `extend`s another class `Parent`, `Child` can then store and do anything that `Parent` can do, but you can then build on `Child` so that it has "extra features".

```
public class Parent {
   public int foo;
}
```

```
public class Child extends Parent {
   public int bar;
}
```

In this example, the `Parent` class only has a single `foo` instance variable. The `Child` class has two instance variables: `foo` and `bar`.

Any non-`private` instance variables and methods will be carried over to the child class, and treated as if they had been written in the child class itself.