MACQUARIE University

Department of Computing

COMP125 Fundamentals of Computer Science
Workshop - Classes and Objects - 2

Learning outcomes

Following are this week's learning outcomes,

a. Implement and use `compareTo()` method

b. References

c. Creating and using arrays of objects

Questions

1. Consider the following class definition for a `Cube`. A cube is a 3-dimensional object, enclosed by 6 equal squares. A cube is characterised by the length of each side (all sides have equal length).

```
public class Cube {
        private double side;
        public double getSide() {
                return side;
        }
        public void setSide(double side) {
                this.side = Math.max(0, side);
        }
        public Cube(double side) {
                setSide(side);
        }
        public double volume() {
                return side*side*side;
        }
}
```

(a) **`compareTo` method definition:** Define the method `compareTo` that when passed another `Cube` object, returns

- 1, if the calling object has a larger side than that of the parameter object
- -1 if the calling object has a smaller side than that of the parameter object
- 0 if the calling object and the parameter object have the same sides.

Solution:

```
public class Cube {
```

```
        //other stuff

        public int compareTo(Cube other) {
                if(this.side > other.side)
                        return 1;
                if(this.side < other.side)
                        return -1;
                return 0;
        }
}
```

(b) **Calling `compareTo` method:** Complete the following client code that performs the following operations,

a. create a `Cube` object `c1` with side of 1.5
b. create a `Cube` object `c2` with side of 3.5
c. create a `Cube` object `c3` with side of 1.5
d. creates a `Cube` object `c4` with side of 0.5
e. Store the outcome of calling `compareTo()` on `c1`, passing `c2` to the method, in a variable `flag1`.
f. Store the outcome of calling `compareTo()` on `c3`, passing `c1` to the method, in a variable `flag2`.
g. Store the outcome of calling `compareTo()` on `c4`, passing `c3` to the method, in a variable `flag3`.

Solution:

```
public class Client {
        public static void main(String[] args) {
                Cube c1 = new Cube(1.5);
                Cube c2 = new Cube(3.5);
                Cube c3 = new Cube(1.5);
                Cube c4 = new Cube(0.5);
                int flag1 = c1.compareTo(c2);
                int flag2 = c3.compareTo(c1);
                int flag3 = c4.compareTo(c3);
        }
}
```

(c) **Tracing `compareTo` execution:** What are the values of `flag1, flag2, flag3` after the code is executed?

Solution: flag1 = 1 flag2 = 0 flag3 = -1

(d) **Understanding references:** Consider the following client code,

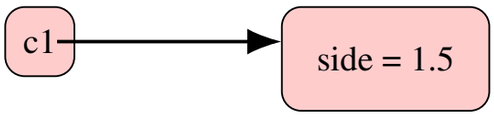
```
public class Client {
        public static void main(String[] args) {
                Cube c1 = new Cube(1.5);
```

```
                    Cube c2 = new Cube(3.5);
                    Cube c3 = c2;
                    double s = c1.getSide() - 1;
                    c2.setSide(s);
            }
}
```

Draw a memory diagram to illustrate the storage of these objects in the memory.

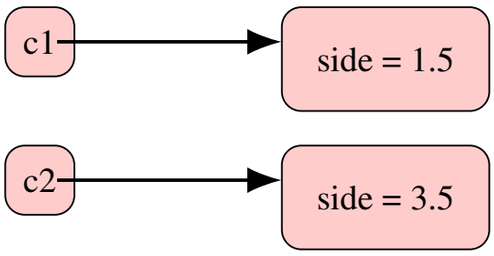
Solution: After the statement

```
Cube c1 = new Cube(1.5);
```



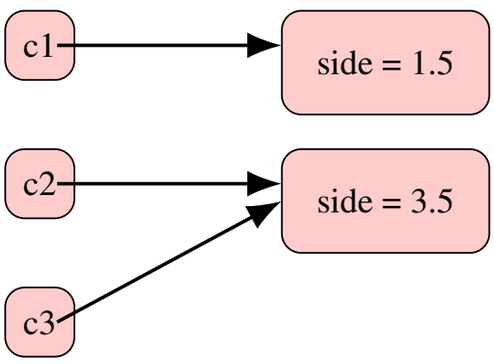
After the statement

```
Cube c2 = new Cube(3.5);
```



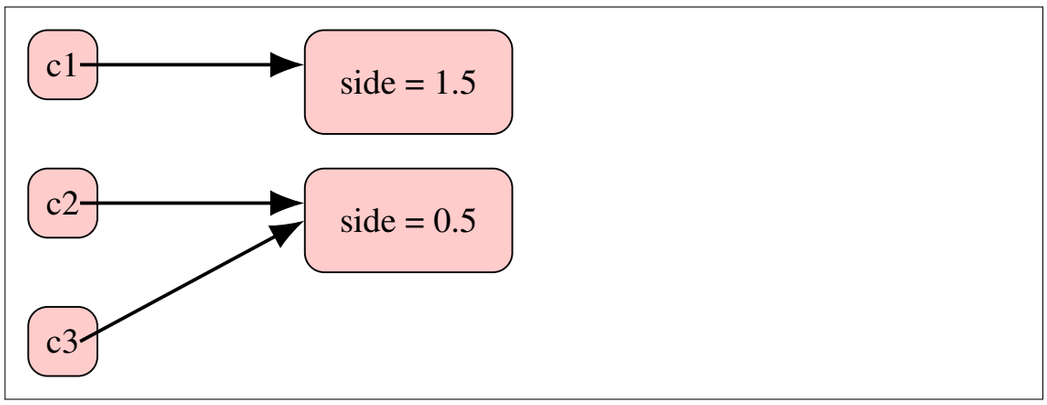
After the statement

```
Cube c3 = c2;
```



After the statements

```
double s = c1.getSide() - 1;
c2.setSide(s);
```



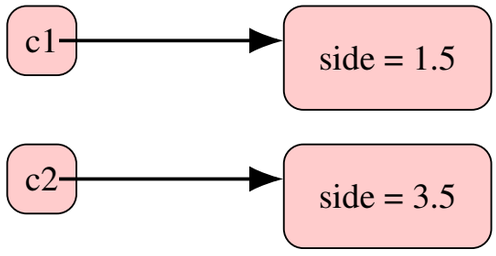
(e) **Understanding passing of objects to methods:** Consider the following client code,

```
public class Client {
        public static void main(String[] args) {
                Cube c1 = new Cube(1.5);
                Cube c2 = new Cube(3.5);
                int flag = c1.compareTo(c2);
        }
}
```

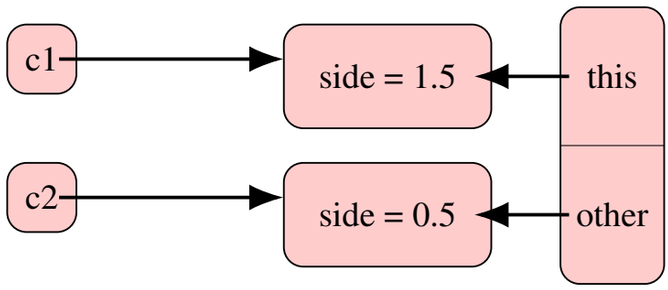
Draw a memory diagram to illustrate the memory manipulations that are caused by the statements of the above code. Specifically, the memory scope during the method call `c1.compareTo(c2))`.

Solution: After the statements

```
Cube c1 = new Cube(1.5);
Cube c2 = new Cube(3.5);
```



During the call `c1.compareTo(c2)`,



2. **Class containing an array**:

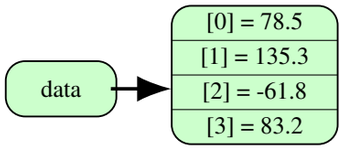
   Consider the following class definition,

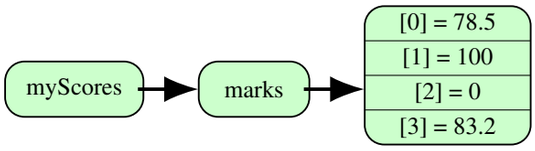
```
public class Marks {
        private double[] marks;
        //in this case, setter should be called only
        //from the constructor, hence private

        private void setMarks(double[] m) {
                if(m == null)
                        return;
                marks = new double[m.length];
                for(int i=0; i < m.length; i++)
                        setMark(i, m[i]);
        }

        public void setMark(int idx, double mark) {
                if(idx < 0 || idx >= marks.length)
                        return; //out of bounds
                if(mark < 0)
                        marks[idx] = 0;
                else if(mark > 100)
                        marks[idx] = 100;
                else
                        marks[idx] = mark;
        }

        public double getMark(int idx) {
                if(idx < 0 || idx >= marks.length)
                        return 0; //out of bounds
                return marks[idx];
        }

        public Marks(double[] m) {
                if(m == null)
                        return;
                setMarks(m);
        }
}
```

Consider the following client code:

```
public class Client {
        public static void main(String[] args) {
                double[] data = {78.5, 135.3, -61.8, 83.2};
                Marks myScores = new Marks(data);
        }
}
```

Discuss the following memory diagram in pairs. Call your tutor for assistance if neither of you understand what's going on. If you clearly understand it, please help others who are having difficulty.





3. **JUnit test (Time permitting)**

   Run the JUnit test `testVolume` in JUnit test class `SphereTest`. Correct the method `volume` in class `Sphere` so that the test passes. The description for the method's requirements is provided as method comment.

Solution:

```
public double volume() {
  return 4 / 3.0 * Math.PI * radius * radius * radius;
}
```

4. **(Time permitting)** Consider the following class definition,

```
public class Line {
        private double x1, y1, x2, y2;
        //assume getters, setters, constructors

        public double getLength() {
                double dx = (x1 - x2);
                double dy = (y1 - y2);
                return Math.sqrt(dx*dx + dy*dy);
        }
}
```

   (a) Write a JUnit test case to ensure the correctness of `getLength()`. You will need the assertion

   `assertEquals(expectedValue, valueReturnedByMethodUnderTest, tolerance)`

```
@Test
public void testGetLength() {
        //to be completed
}
```

(b) Define instance method `compareTo` that when passed an object of class `Line`, returns 1 if the length of the calling object is more than that of parameter object, -1 if the length of the calling object is less than that of parameter object, and 0 if the length of the calling object and parameter object are the same.

Solution:

```
public int compareTo(Line other) {
        if(getLength() > other.getLength())
                return 1;
        if(getLength() < other.getLength())
                return -1;
        return 0;
}
```

Supplementary exercises (take-home exercises)

1. **Bank account**

Define a class representing a bank account with methods to check the balance, to deposit and withdraw funds and to test whether the account is overdrawn.

In package `comp125`, there is an outline implementation of the class along with a set of tests for the different methods. Use this as the basis for your implementation. Read also the in-code comments for the details of what each method is expected to do.

The first thing you need to do is decide how the `BankAccount` class is going to store its data – the bank balance. You need to define an instance variable of the appropriate type in the class. (Label your instance variable `private`.)

Then you need to write the body of each of the methods. At each point, run the tests to see if what you've written works. Try to work on the tests from top to bottom, getting one working after another. You can use the debugger to walk through your program if it's not doing what you expect – remember to place a breakpoint in your class to stop execution when running under the debugger.

When you've passed all the tests, complete the `public static void main(String[] args)` method in `BankAccountClient` to make use of the bank account class. Make a bank account, add $42.5 to the account, then withdraw $80.6 from it, display if it's overdrawn (`true`) or not (`false`) and display the balance. You can include such a `main` method inside your `BankAccount` class.

Please note that `BankAccountTest` uses an `assertEquals` assertion that has three parameters. The syntax is;

```
assertEquals(expectedDouble, returnedDouble, tolerance);
```

The assertion is successful (passes) if the `expectedDouble` and `returnedDouble` differ by at most `tolerance`. That is, $Math.abs(expectedDouble - returnedDouble) \leq tolerance$.

Solution:

```
public class BankAccount {
        public BankAccount() {
                setBalance(0);
        }

        /**
         * @return the balance of the account
         */
        public double getBalance() {
                return balance;
                // You might want to change this statement.
        }

        /**
         * since balance can be negative or postive (or zero), there is really
         * no validation rule here
         * @param bal
         */
        public void setBalance(double bal) {
                balance = bal;
        }

        /**
         * Increase the balance by the amount specified.
         * @param amount
         */
        public void deposit(double amount) {
                setBalance(balance+amount);
```

```
        }

        /**
         * Decrease the balance by the amount specified. It's OK if the resulting
         * balance is negative (overdrawn).
         * @param amount
         */
        public void withdraw(double amount) {
                setBalance(balance-amount);
        }

        /**
         * @return true if the balance is negative, false otherwise
         */
        public boolean overdrawn() {
                return balance < 0;  // You might want to change this statement.
        }
}
```