COMP132 – Computer Science II
Fall 2015

Homework #4
Inheritance and Polymorphism
Solutions

1. This question builds upon the `TextMessage` example from class and contained in the `132SampleCode` project (`comp132.examples.inheritance`). You will define a new class named `PaymentMessage` that represents a new type of text message that can be sent to a vendor to make a payment. This new type of message has text but also has a cash value.

a. Give an implementation of the `PaymentMessage` class. Because a `PaymentMessage` is everything a `TextMessage` is, plus some other stuff, it should be a sub-class of `TextMessage`. In addition to being a sub-class of `TextMessage` the `PaymentMessage` must:

i. have a field to keep track of the amount of the payment
ii. have an appropriate constructor
iii. have an accessor for the amount of the payment.
iv. not implement any other methods (yet).

```
public class PaymentMessage extends TextMessage {
    private float paymentAmount;

    public PaymentMessage(long from, long to, String msg, double amt) {
        super(from, to, msg);
        paymentAmount = amt;
    }

    public float getAmount() {
        return paymentAmount;
    }
}
```

b. List all of the methods that can be invoked on a `PaymentMessage` object as defined in part a. Remember a sub-class inherits methods from its super-class (which also inherits from its super-class!).

Defined in `PaymentMessage`:	Inherited from `TextMessage`:	Inherited from `Object`:
`getAmount`	`getMessageText`	`clone`
	`getRecipiantNumber`	`equals`
	`getSenderNumber`	`finalize`
	`getMessageLength`	`getClass`
	`getMessageType`	`hashCode`
	`toString`	`notify`
	`equals`	`notifyAll`
		`toString`
		`wait (3 versions)`

c. Consider the following snippet of code that uses a `PaymentMessage` object as defined in part a:

```
PaymentMessage pm = new PaymentMessage(
  7173456789L, 71798765432L, "Here ya go", 22.75);
System.out.println("pm.getMessageType(): " +
  pm.getMessageType());
```

When executed this code would produce the output:

```
pm.getMessageType(): Text Message
```

Which is clearly not the desired output. Explain as clearly and fully as you can why this snippet generates this output.

The class `PaymentMessage` inherited the `getMessageType` method from its super-class, `TextMessage`. Thus, when `getMessageType` was invoked on a `PaymentMessage` object, the implementation of that method from `TextMessage` was used. That implementation returns the string "`Text Message`".

d. Give the code that you would add to the `PaymentMessage` class to cause the snippet of code in part c produce the output:

```
pm.getMessageType(): Payment Message
```

```
public String getMessageType() {
  return "Payment Message";
}
```

e. Currently the length of a `PaymentMessage` would simply be the length of the text that it contains. However, storing the payment requires some space as well. Thus, the size of a `PaymentMessage` should be the length of its text plus 4 bytes of storage for the payment information (because it is a float). Give the code that you would add to the `PaymentMessage` class so that it computes the correct length.

```
public int getMessageLength() {
  int textLen = super.getMessageLength();
  int totalLen = textLen + 4;
  return totalLen;
}
```

2. The questions below make use of the following three classes:

```
class Ecks {
  private int x;

  public Ecks(int a)
{
    x = a;
  }

  public int bar() {
    return x + 1;
  }

  public int foo() {
    int b = bar();
    return x*b;
  }
}
```

```
class Why extends Ecks
{
  private int y;

  public Why(int b) {
    super(7);
    y = b;
  }

  public int bar() {
    int
  c=super.bar();
    return c + y;
  }

  public int qux() {
    return y + 3;
  }
}
```

```
class Zee extends Why
{
  private int z;

  public Zee() {
    super(5);
    z = 3;
  }

  public int bar() {
    int d = qux();
    return z * d;
  }
}
```

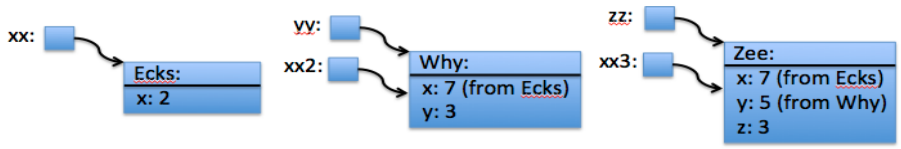
Assume that the following statements are executed before each of the questions below (i.e. the parts below do not build upon each other, each is started with a fresh set of objects and references).

```
Ecks xx = new Ecks(2);
Why yy = new Why(3);
Zee zz = new Zee();
Ecks xx2 = yy;
Ecks xx3 = zz;
```

Note: Try to answer the following questions without compiling or running the code.

a. For each of the following statements, indicate if it is *legal* (i.e. will compile as written) or *illegal* (will generate a compiler error):

An object diagram showing the objects and references produced by the above statements will be useful for understanding the solution to all of the related questions:



i. `Ecks x1 = yy;` **Legal**

`yy` holds a reference to an object of type `Why`. The class `Why` is a sub-class of `Ecks`. Therefore the reference `x1` of type `Ecks` can refer to the object of type `Why`.

ii. `Why y1 = xx;` **Illegal**

`xx` holds a reference to an object of type `Ecks`. The class `Ecks` is not a sub-class of `Why`. Therefore the reference `y1` of type `Why` cannot refer to the object of type `Ecks`.

iii. `Why y2 = zz;` **Legal**

`zz` holds a reference to an object of type `Zee`. The class `Why` is a sub-class of `Zee`. Therefore the reference `y2` of type `Why` can refer to the object of type `Zee`.

iv. `Zee z2 = yy;` **Illegal**

`yy` holds a reference to an object of type `Why`. The class `Why` is not a sub-class of `Zee`. Therefore the reference `z2` of type `Zee` cannot refer to the object of type `Why`.

b. For each of the following statements, indicate if it is *legal* (i.e. will compile as written) or will generate a *compile time error* or a *runtime error*.

i. `Why y2 = xx3;` **Compile Time Error**

While `xx3` holds a reference to an object of type `Zee` and a reference of type `Why` can hold a reference to an object of type `Zee` the compiler cannot be sure this will be the case. It is possible that `xx3` refers to an object of type `Ecks`, `Why` or `Zee`. If it were the case that `xx3` referred to an object of type `Ecks`, then the reference `y2` of type `Why` could not legally refer to the object referred to by `xx3`. Thus, because a `TypeCastException` could occur a cast is necessary here and there will be a compile time error.

ii. `Why y3 = (Why)xx3;` **Legal**

As above `xx3` could hold a reference to an object of type `Ecks`, `Why` or `Zee`. If at runtime that object were of type `Ecks` this statement would cause a runtime exception. Thus, the cast is required to tells the compiler that we expect that `xx3` will hold a reference to an object that can be referred to by a reference of type `Why` (i.e. an object of type `Why` or `Zee` but not `Ecks`). When actually run, `xx3` refers to an object of type `Zee`, which can be referred to by a reference of type `Why`, so this is legal.

iii. `Zee z1 = (Zee)xx3;` **Legal**

The explanation here is very similar to the previous statement.

iv. `Zee z2 = (Zee)xx2;` **Run Time Error**

The explanation here goes along in a similar fashion to the previous two statements until the program is run. At runtime the reference `xx2` actually refers to an object of type `Why`. Because `Why` is not a sub-class of `Zee` (actually the opposite is true), the reference `z2` of type `Zee` cannot refer to the object of type `Why` and a runtime exception occurs.

v. `Why y4 = (Zee)xx3;` **Legal**

This one is legal, but intentionally confusing. In practice this would be written as in the second statement above (where `y3` is assigned). The difference here is that the reference in `xx3` is first cast to type `Zee`. Now anything that can be referred to by a reference of type `Zee` can also be referred to by a reference to type `Why` (because `Zee` is a sub-class of `Why`). So cast is required to copy the reference of type `Zee` into `y4`.

c. Give the output that would be produced by the following lines of code:

```
System.out.println("xx.bar() = " + xx.bar());
System.out.println("yy.bar() = " + yy.bar());
```

xx.bar() = 3

The variable `xx` refers to an `Ecks` object and the object determines the behavior that is performed. So the implementation of `bar` in the `Ecks` class is executed. That method returns the value of the field `x` (`2`) plus `1`, which is `3`.

yy.bar() = 11

In this statement the variable `yy` refers to a `Why` object. So the implementation of `bar` in the `Why` class is executed. That method returns the value returned by `super.bar()` (`7+1 = 8`) plus the value of the field `y` (`3`), which is `11`.

d. Give the output that would be produced by the following lines of code:

```
System.out.println("xx.foo() = " + xx.foo());
System.out.println("yy.foo() = " + yy.foo());
```

xx.foo() = 6

The variable `xx` refers to an `Ecks` object. So the implementation of `foo` in the `Ecks` class is executed. That method returns the result of calling `bar` (`2+1 = 3`) times the value of the field `x` (`2`), giving `3*2 = 6`.

yy.foo() = 77

The variable `yy` refers to a `Why` object. So the implementation of `foo` in the `Why` class should be executed. But `Why` does not override `foo`, so the implementation inherited from `Ecks` is executed. That method will return the result of calling `bar` times the value of the field `x` (`7`). But notice that `Why` does override the method `bar` and that the call `bar()` is equivalent to `this.bar()`. The `this` reference refers to the `Why` object and thus the implementation of `bar` in `Why` will be used. The `bar` method executes as describe above in the second part of question c, returning `11`, which is multiplied by `x` (`7`) giving `77`.

e. Give the output that would be produced by the following lines of code:

```
System.out.println("zz.bar() = " + zz.bar());
System.out.println("zz.foo() = " + zz.foo());
```

zz.bar() = 24

The variable `zz` refers to a `Zee` object. So the implementation of `bar` in the `Zee` class is executed. This method will return the value of the field `z` (3) times the result returned by a call to `this.qux()`. `Zee` has inherited its implementation of `qux` from `Ecks` and that implementation returns the value of the field `y` (5) plus 3, or 8. In the end, the call to `bar()` returns `3*8 = 24`.

zz.foo() = 168

The variable `zz` refers to a `Zee` object. Note that the value of the `this` reference will also refer to that same `Zee` object when `foo` is invoked. Now, `Zee` does not implement `foo`, so `Zee` inherits the version in `Why`, which in turn was inherited from `Ecks`. That implementation of `foo` returns the value of `x` (7) times the result of invoking `this.bar()`. The invocation of `this.bar()` proceeds as just describe above and returns `24`. The end result of the call to `zz.foo()` is then `7*24 = 168`.

3. Imagine that a cell carrier wants to begin charging for sending multimedia messages. The cost of sending a multimedia message is 2 cents for each character of text (e.g. 0.02 times the length of the text) plus 3 cents for each byte in the multimedia file (e.g. 0.03 times the size of the file). Give a method `computeCost` as it would appear in the `MultimediaMessage` class that computes and returns the cost of sending the message.

```
public double computeCost() {
  double textCost = 0.02 * super.getMessageLength();
  double mediaCost = 0.03 * fileSize;
  return textCost + mediaCost;
}
```

4. This question uses the `TextMessageList` and `MultimediaMessage` classes from class and contained in the 132SampleCode project (`comp132.examples.inheritance`).

a. Consider the following snippet of code:

```
MultimediaMessage mm1 =
  new MultimediaMessage(3517654321L, 7171234567L, "WDYT?");
mm1.attachFile("skiTrip.jpg", 2000);
System.out.println(mm1);
```

This code displays the following output:

```
(From 3517654321 to 7171234567): WDYT?
```

A `MultimediaMessage` clearly has more information contained in it than was displayed (e.g. the file name and file size). Explain as clearly and as fully as you can why only the above information was displayed.

When an object is passed to the `println` method, that object's `toString` method is invoked and the `String` that is returned is printed. The `MultimediaMessage` class inherited its `toString` method from the `TextMessage` class. The implementation of the `toString` method returns a `String` containing only the sender's number, the recipiant's number and the text. Thus, when a `MultimediaMessage` is printed that is all that is displayed.

b. Modify the `MultimediaMessage` class so that when the snippet of code from part a is run the output is:

```
(From 3517654321 to 7171234567): WDYT?
    File: skiTrip.jpg (2000)
```

Give only the code that you added to the `MultimediaMessage` class as your answer to this question. Hint: Including escaped characters in a `String` can affect the way it is printed. Including a "`\n`" results in a new line and a "`\t`" results in a tab.

```
public String toString() {
  String tmStr = super.toString();
  String mmStr = tmStr + "\n\tFile: " + fileName +
    " (" + fileSize + ")";
  return mmStr;
}
```

c. Add a method named `listMessagesFrom` to the `TextMessageList` class that accepts a single parameter indicating a phone number and displays the basic information about each message that was received from that number. The basic information to be displayed about each object should be obtained by invoking its `toString` method.

```
public void listMessagesFrom(long fromNumber) {
  for (TextMessage tm : messageList) {
    if (tm.getSenderNumber() == fromNumber) {
      System.out.println(tm);
    }
  }
}
```

Recall that the `println` method automatically invokes the `toString` method on an object that is passed to it. This is always possible because all objects inherit a `toString` method from the `Object` class. In this case, the `TextMessage` class has overridden `toString`, ensuring that the proper output is generated.

5. Consider the following snippet of code:

```
ArrayList<TextMessage> tmList = new ArrayList<TextMessage>();

TextMessage tm1 =
  new TextMessage(7171234567L, 3517654321L, "Yo Joe!");
TextMessage tm2 =
  new TextMessage(7171234567L, 2159876543L, "Hi Kim!");

tmList.add(tm1);
tmList.add(tm2);

MultimediaMessage mm1 =
  new MultimediaMessage(7171234567L, 3517654321L, "Yo Joe!");
mm1.attachFile("joe.jpg", 2000);

System.out.println("contains mm1: " + tmList.contains(mm1));
```

Using the implementations of `TextMessage` and `MultimediaMessage` given in class, the last line above will print out:

```
contains mm1: true
```

This is clearly not the desired behavior. This happens because the `contains` method passes each object in the list to the `equals` method of `mm1` and `mm1` inherited its `equals` method from the `TextMessage` class. Thus, when `tm1` is passed to `mm1`'s `equals` method it returns `true` and the `ArrayList` concludes that `mm1` is contained in the list. To obtain the desired behavior we must override the `equals` method in the `MultimediaMessage` class.

Give an implementation of `equals` for the `MultimediaMessage` class that overrides the one inherited from `TextMessage` and leads to the expected behavior. Keep in mind that the objects passed to the `equals` method can be any type of object that could be in `tmList`. Hints: 1. A `MultimediaMessage` object can only be equal to another `MultimediaMessage` object. 2. The `instanceof` operator can be used.

```
public boolean equals(Object o) {
   if (o instanceof MultimediaMessage) {
      MultimediaMessage msg = (MultimediaMessage) o;

      if (this.getSenderNumber() == msg.getSenderNumber() &&
           this.getRecipientNumber() == msg.getRecipientNumber() &&
           this.getMessageText().equals(msg.getMessageText()) &&
           fileName.equals(msg.getFileName()) &&
           fileSize == msg.fileSize) {

           return true;
      }
      else {
         return false;
      }
   }
   else {
      return false;
   }
}
```

Bonus #1: Given your solution to Bonus #1, consider the output that would now be generated by the following snippet of code:

```
ArrayList<TextMessage> tmList = new ArrayList<TextMessage>();

MultimediaMessage mm1 =
   new MultimediaMessage(7171234567L, 3517654321L, "Yo Joe!");
mm1.attachFile("joe.jpg", 2000);
TextMessage tm2 =
   new TextMessage(7171234567L, 2159876543L, "Hi Kim!");

tmList.add(mm1);
tmList.add(tm2);

TextMessage tm1 =
   new TextMessage(7171234567L, 3517654321L, "Yo Joe!");

System.out.println("contains tm1: " + tmList.contains(tm1));
```

Explain why the above code does or does not operate correctly given your implementation of the `.equals` method.

The issue here is that the object on which the `.equals` method will be invoked is a `TextMessage` object. Therefore, the implementation of `.equals` will be the one from the `TextMessage` class. That implementation still compares only the sender and recipient numbers and the contents of the text. Thus, the output of the above snippet of code will be:

```
contains tm1: true
```

This is the case even though the `ArrayList` does not contain the object referred to by `tm1`.

Note: Your answer may differ if you produced different code in Bonus #1.

Bonus #2: It turns out writing a correct `.equals` methods in the presence of inheritance is tricky. Research how the `.equals` method can be correctly implemented in this situation and give an implementation for the `TextMessage` class that operates correctly in the above situation.

Here is a solution given by eclipse. On the Source menu there is an option for Generate `hashCode` and `equals`. Selecting this item will generate the following code in `TextMessage`:

```
@Override
public boolean equals(Object obj) {
  if (this == obj) {
    return true;
  }
  if (obj == null) {
    return false;
  }
  if (getClass() != obj.getClass()) {
    return false;
  }

  TextMessage other = (TextMessage) obj;
  if (messageText == null) {
    if (other.messageText != null) {
      return false;
    }
  } else if (!messageText.equals(other.messageText)) {
    return false;
  }

  if (recipientNumber != other.recipientNumber) {
    return false;
  }

  if (senderNumber != other.senderNumber) {
    return false;
  }

  return true;
}
```