CECS 277 Midterm 1 Review

1.Implement a subclass

```
public class B extends A {

}
```

2. Abstract classes

Cannot instantiate an abstract class
the top base class in an inheritance

3. Subclass constructor

```
public class Circle extends Shape
{
  private double radius;
  public Circle(int r) {
    super();//call Shape()
    radius = r;
}

  public Circle(int x, int y, int r)
 {
    super(x,y); // calls Shape(x,y)
    radius = r;
  }
}
```

4. final

If a method is final, then the method cannot be overridden
in the subclass.

5. Overriding methods from a base class

Let Circle extend Shape

```
public class Circle extends Shape {
  .
  .
  .
  //every class has a toString() method.
  public String toString() {
    return super.toString();//super.toString() calls the method to
                            //toString in the base class
  }
}
```

6. Implementing an abstract method in a subclass

No key word abstract

7. private, protected, public

```
public class A {
  protected int a;
}

public class B extends A {
  public void add() {
    //subclasses have access to protected
    a = a + 5;
  }
}

public static void main() {
B b = new B();
System.out.println(b.a); // THIS RESULTS IN COMPILER ERROR
//SINCE 'a' IS PROTECTED. 'a' can only be seen by
//subclass implementations/same packages
}
```

8. Polymorphism

```
Shape[] s = new Shape[10];
s[0] = new Circle(__);
Rectangle r = new Rectangle(__);
s[1] = r;
.
.
.
s[9] = new Cylinder(__);
//Assume double computeArea() is an abstract method.
double totalArea = 0.0;
for (int i = 0; i <= s.length; i++) {
  totalArea += s[i].computeArea();
}
```

//using instanceOf to find the total area of all rectangles in array s.

```
double totalAreaofRectangles = 0.0;
for (int i = 0; i <= s.lengh; i++) {
  if (s[i] instanceof Rectangle) {
    //do something
    totalAreaofRectangles += s[i].computeArea();
  }
}
```

```
//using instanceOf to find the total volume of all rectangles in array s.
//Assume the method computeVolume is defined only in the class Cylinder

      double totalVolume = 0.0;
      for (int i = 0; i <= s.lengh; i++) {
        if (s[i] instanceof Cylinder)
      {
          //do something
          totalVolume += ((Cylinder) s[i]).computeVolume();
        }
      }
```

9. Composition

```
      class Book {

      }

      class BookOrder {
        private Book b; // <- composition
      }
```

      Note: Inheritance is an "is-a" relationship
            Composition is a "has-a" relationship

10. Copy constructor vs clone

      Copy constructors

```
        public class A {
          private int a1;
          private int a2;
          .
          .
          .
          public A(A a ) {
             a1 = a.a1;
             a2 = a.a2;
          }
        }
```

      Clone

        Shallow copy vs deep copy

          Shallow Copy

            No composition

```
            class A implements Cloneable {
              .
              .
              public Object clone() {
                  try {
                    return super.clone();
                  } catch (CloneNotSupportedException e) {
```

```
            return null;
          }
        }
      }
    Deep Copy
      class A implements Cloneable {

      }
      class B extends A implements Cloneable{
        private A a;
        public setA(A aa)
        { a = aa;}

        public Object clone() {
          try {
            B b = (B)super.clone();
            A a = (A)a.clone();)
            b.setA(a);;
            return b;
          } catch (CloneNotSupportedException e) {
            return null;
          }
        }
      }
```

11. UML Diagram
    -private
    +public
    #protected
12. Javadoc
```
    /**
      @param
      @return
    */
```
13. Interface
    Constant Variable
    abstract method declaration
14. Sorting
```
      public class Employee implements Comparable<Employee> {
        private int id;
        private String name;

        public int compareTo(Employee o) {
          //compare id
          return this.id - o.id;
        }
      }
   //Create another class to sort name
      public class SortByName implements Comparator {
        public int compare(Object o1, Object o2) {
```

```
      //compare name
    }
  }



  public static void main() {
    Employee[] e = new Employee[5];
    e[0] = new Employee(123,....);
    .
    .
    .
    e[4] = new Employee(423,....);

    //sorting by id since compareTo sorts by name.
    Arrays.sort(e);

    //sorting by name
    Arrays.sort(e, new SortByName());
  }
```

15. Method .equals() to compare content (value) of objects

```
== compare reference
Circle c1 = new Circle(__);
Circle c2 = c1;
//c2 is just a pointer to where c1 is pointing.
//c1==c2 is true
//To actually compare content, you must override the equals() method
public Circle {
  .
  .
  .
  public boolean equals(Object o) {
    if (o instanceOf Circle) {
      return radius == ((Circle)o).radius;
    }
 //check if radius of c1 equals c2
if (c1.equals(c2)) {

}
```

16. Upcasting and downcasting

What is the difference between up-casting and down-casting?

Casting in java means converting from type to type. When It comes to the talking about **upcasting** and **downcasting** concepts we are talking about converting the objects references types between the child type classes and parent type class. Suppose that we have three classes (A,B,C). Class B inherit from class A. Class C inherit from class B. As follows:

```
interface A

{
  void display();
}
class B implements A
{
    public void display() {
        System.out.println("Am in class B");
    }

}
class C extends B
{
    @Override
    public void display() {
        System.out.println("Am in class C");
    }
}
```

Note that we have declared the class A as interface but this will not make any changes in illustrating the upcasting/downcasting ideas.. However, It only restricts you from creating an object instance directly from type A.

Applying Up-casting

The upcasting is casting from the child class to base class. The upcasting in java is implicit which means that you don't have to put the braces(type) as a indication for casting. Below is an example of upcasting where we create a new instance from class C and pass it to a reference of type A. Then we call the function display.

```
public static void main(String[] args) {

        // Upcasting from subclass to super class.
        A aRef=new C();

        aRef.display();//Am in class C


    }
```

Applying Down-casting

The downcasting is the casting from base class to child class. Below we continue on the previous upcasting snippet by adding to lines for downcasting where we down cast the aRef reference from type A to type B. Downcasting is explicit note the usage of braces(type) in the example below.

```
public static void main(String[] args) {

        // Upcasting from subclass to super class.
        A aRef=new C();

        aRef.display();//Am in class C
        //Downcasting of reference to subclass reference.
        B bRef=(B) aRef;
        bRef.display();//Am in class C

    }
```

The output of the code is :

```
Am in class C
Am in class C
```

Note the display function of class C is called because the type of object is class C

Functions and variables after casting

After casting, you will have access only to the current reference type class members even your object is from type C and your reference of type A. Any unique class member in class C will not be visible to you.For instance:

```
public class ExampleClass {


    public static void main(String[] args) {

        // Upcasting from subclass to super class.
        A aRef=new C();
        aRef.setX(43);// Compile Error

    }
}

interface A
{
  void display();
}

class B implements A
{

    public void display() {
        System.out.println("Am in class B");
    }

}

class C extends B
{
```

```
    private int x;
    @Override
    public void display() {
        System.out.println("Am in class C");
    }

    public void setX(int x) {
        this.x = x;
    }
}
```

Common mistake in casting usage:

In downcasting you can't down cast to a inheritance hierarchy level less than your object instance level at creation time. For instance :

```
public class ExampleClass {


    public static void main(String[] args) {

        // Upcasting from subclass to super class.
        A aRef=new B();

        aRef.display();//Am in class C
        //Downcasting of reference to subclass reference.
        C bRef=(C) aRef; //ERROR
        bRef.display();
    }
}

interface A
{
  void display();
}

class B implements A
{

    public void display() {
        System.out.println("Am in class B");
    }

}

class C extends B
{
    @Override
    public void display() {
        System.out.println("Am in class C");
    }

}
```

In case above your code will compile correctly but you will get runtime error because a created object instance of type B is passed to a reference

of type C which is less than B in level. This means that in down-casting we are limited to the original object instance class type while in up-casting we don't have such similar situations.

The output:

```
Am in class B

Exception in thread "main" java.lang.ClassCastException: B cannot be cast to C

   at ExampleClass.main(ExampleClass.java:21)
```

More casting

An object is automatically upcasted to its super class type. You need not to mention class type explicitly. But, when an object is supposed to be downcasted to its sub class type, then you have to mention class type explicitly. In such case, there is a possibility of occurring class cast exception. In most of time, it occurs when you are trying to downcast an object explicitly to its sub class type.

Try to run below program.

```
package com;
class A
{
    int i = 10;
}

class B extends A
{
    int j = 20;
}

class C extends B
{
    int k = 30;
}

public class ClassCastExceptionDemo
{
    public static void main(String[] args)
    {
        A a = new B();   //B type is auto up casted to A type - LINE 1
        B b = (B) a; //A type is explicitly down casted to B type.LINE 2
        C c = (C) b;    //Here, you will get class cast exception
        System.out.println(c.k);
    }
}
```

You will get ClassCastException. Below is the sample of the error.

Exception in thread "main" java.lang.ClassCastException: com.B cannot be cast to com.C
at com.ClassCastExceptionDemo.main(ClassCastExceptionDemo.java:23)

In the above example, Class B extends Class A and Class C extends Class B. In the main method, Class B-type object is created (Line 1). It will be having two non-static fields. one field (int i) is inherited from class A and another one is its own field (int j). 'a' is Class A-type reference variable which will be pointing to this newly created object. In the next statement (Line 2), reference variable 'a' is assigned to 'b' which is Class B-type reference variable. After execution of this statement, 'b' will also be pointing to the same object to which 'a' is pointing. In the third statement, 'b' is assigned to 'c' which is Class C-type reference variable. So, 'c' will also be pointing to same object to which 'a' and 'b' are pointing. While executing this statement, you will get run time exception called Class Cast Exception.

The real use of instanceof keyword by the example given below.

```
interface Printable{}
class A implements Printable{
public void a()
{System.out.println("a method");}
}
class B implements Printable{
public void b()
{System.out.println("b method");}
}

class Call{
void invoke(Printable p){//upcasting
if(p instanceof A){
A a=(A)p;//Downcasting
a.a();
}
if(p instanceof B){
B b=(B)p;//Downcasting
b.b();
}

}
}//end of Call class
```

```
class Test4{
public static void main(String args[]){
Printable p=new B();
Call c=new Call();
c.invoke(p);
}
}
```

Run time error and compiler time error

```
public class Animal {
    public void walk()
    {
        System.out.println("Walking Animal");
    }
class Dog extends Animal {
    public void walk()
    {
        System.out.println("Walking Dog");
    }
    public void sleep()
    {
        System.out.println("Sleeping Dog");
    }
}
class Demo {
    public static void main (String [] args) {
      Animal a = new Animal();
      Dog d = new Dog();
      a.walk();
      d.walk();
      d.sleep();

      //upcasting
        Animal a2 = (Animal)d;
        a2.walk();
        //a2.sleep();  error

        //downcasting
        Animal a3 = new Dog();
        //Dog d2 = a3;  //compile time error
        Dog d2 = (Dog)a3;
        d2.walk();
        d2.sleep();

        //Run time error: Animal cannot be cast to Dog
        Animal a4 = new Animal();
        //Dog d3 = (Dog)a4;
```

```
        //d3.walk();
        //d3.sleep();
    }
```

Output:

```
Walking Animal
Walking Dog
Sleeping Dog
Walking Dog
Walking Dog
Sleeping Dog
```

17. Object oriented design

1. CRC Method (Classes, Responsibilities, and Collaborators)
2. Cohesion
3. Low coupling versus high coupling
4. Relationship between classes
    a. Inheritance (is-a relationship
    b. Interface
    c. Aggregation versus Composition (Has-a relatiohip)
    d. Dependency
5. Multiplicities – Aggregation relationship
6. UML Diagram