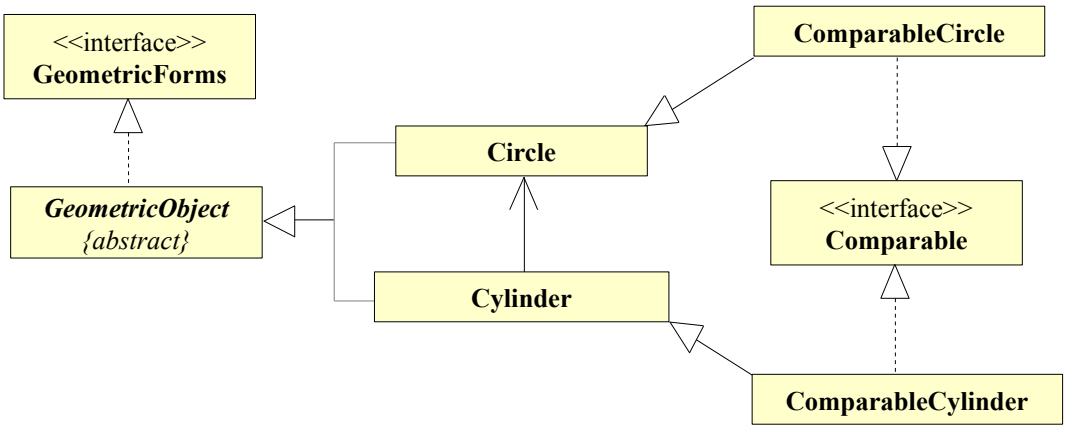
Tentamen 131217- LÖSNINGSFÖRSLAG

Uppgift 1.

a) *Liskov Substitution Principle* säger att en supertyp är utbytbar med sina subtyper. Med den design som gjorts innebär detta alltså att var som helst där vi förväntar oss en cirkel går det lika bra med en cylinder. Eller med andra ord gäller det att *en cylinder är en cirkel*, vilket naturligtvis är helt galet. Den som gjort designen verkar ha infört implementationsarvet mellan klasserna Cylinder och Circle enbart med återanvändning av kod som motiv, vilket inte är ett tillräckligt motiv.

b) Delegering är ett alternativ implementationsarv för att återanvända kod.



c) Synligheten i GeometricObject är **protected**. Synligheten bör dock vara **private**. Man skall alltid eftersträva att exponera så lite som möjligt av superklassens interna representation till dess subklasser. Orsaken är givetvis att om den interna representationen är exponerad och används av subklasserna, måste subklasserna förändras ifall superklassen byter representation. I vårt specifika fall tillhandahåller dessutom klassen GeometricObject publika access-metoder för samtliga instansvariabler, varför inga ytterligare förändringar av klassen behöver göras.

d) Det saknas konstruktorer för att ange var det skapade objektet skall placeras i det två-dimensionella rummet. Är det möjligt skall en instans ges sitt fullständiga tillstånd när instansen skapas.

Uppsättningen konstruktorer är inkonsekvent mellan de olika klasserna. Exempelvis borde ComparableCylinder ha samma uppsättning konstruktorer som Cylinder.

Ur konsekvenssynpunkt kan även ordningen på parametrarna till konstruktorerna ifrågasättas. Exempelvis har klassen Cylinder konstruktorerna Cylinder(radius, length) och Cylinder(radius, color, length), det hade varit lämpligare att den sistnämnda konstruktorn ersatts med Cylinder(radius, length, color).

e) Det hade varit lämpligare att avbilda en färg med exempelvis klassen java.awt.Color (eller en egendefinierd klass) än en String. Likaledes hade det varit lämpligare att avbilda en position med exempelvis klassen java.awt.Point (eller en egendefinierad klass) än som två **int** x och y.

f) Genom att implementera gränssnittet Comparable<E> kommer kompilatorn att kontrollera typtillhörighet, dvs att endast objekt av samma typ jämförs. Går ett anrop av metoden compareTo igenom kompileringen vet vi att objekten som jämförs är av samma typ och ingen explicit typomvandling eller felhantering behövs (i metoden compareTo eller i anropande metod).

```
public class ComparableCircle extends Circle implements Comparable<ComparableCircle> {
    //kod för konstruktorerna utelämnad
    public int compareTo( ComparableCircle cc) {
        if (getRadius() > cc.getRadius()) {
            return 1;
        } else if (getRadius() < cc.getRadius()) {
            return -1;
        } else {
            return 0;
        }
    }//compareTo
}//ComparableCircle
```

Uppgift 2.

a) Utskriften blir:

9
16

Förklarning:

När konstruktorn Sub() anropas är det första som händer att konstruktorn i superklassen Super() anropas, varvid instansvariabeln i sätts till värdet 3. Därefter ökas i med 1 i konstruktorn Sub() och erhåller värdet 4. Anropet s.m() innebär att m() i klassen Sub anropas eftersom s har den dynamiska typen Sub. Metoden ökar i med 5, varvid i erhåller värdet 9. Detta värde utgör första utskriften.

Anropet s.n() innebär att i ökas med 2 och för värdet 11. Sedan anropas metoden m() i klassen Sub eftersom det aktuella objeket s har den dynamiska typen Sub. Denna metod ökar i med 5 varvid i får värdet 16. Detta värde utgör andra utskriften.

b) Utskriften blir:

Base1
Sub1
Sub1

Vilken metod som skall väljas (bland ett antal överlagrade metoder) bestäms vid kompileringen med utgångspunkt för de statiska typerna. Vid exekveringen börjar sökningen efter den valda metoden i klassen som det anropande objekt tillhör. Den dynamiska typen på anropande objekt bestämmer alltså var sökningen börjar.

Uppgift 3.

a)

Information Expert Pattern: Det objekt som har den information som behövs för att utföra en arbetsuppgift skall uttföra arbetsuppgiften.

Dependency Inversion Principle: Programmera mot ett gränssnitt, inte mot en konkret klass.

Open-Closed Principle: En programenhet skall vara öppen för utökningar, men stängd för modifieringar.

I vårt exempel: Punkt är mest lämpad att beräkna sitt avstånd till en annan punkt. Segment använder den konkreta klassen Point. Skall i stället använda ett interface.Klassen Segment är inte stängd för modifieringar. Om även 3D-punkter kommer att införas i framtiden måste klassen modifieras.

b)

```
public interface Point {
  public double distanceTo(Point other);
}//Point

public class Point2D implements Point {
    private double x, y;
    //konstruktorn utelämnad
    public double distanceTo(Point point) {
        Point2D other = (Point2D) point;
        double dx = this.x - other.x;
        double dy = this.y - other.y;
        return Math.sqrt(dx * dx + dy * dy);
    }
}//Point2D

public class Segment {
    private Point point0, point1;
    //konstruktorn utelämnad
    public double length() {
        return point1.distanceTo(point0);
    }
}//Segment
```

Uppgift 4.

a) les.add(s); ger kompileringsfel

b) les.add(c); ger kompileringsfel

c) c = les.get(0); ger kompileringsfel

d) s = les.get(0); korrekt

e) lss.add(s); korrekt

f) lss.add(c); korrekt

g) s = lss.get(0); ger kompileringsfel

h) o = lss.get(0); korrekt

Uppgift 5.

a) Specifikation B är starkast, eftersom den har svagare förvillkor.

b) Ja! En subklass får ha ett starkare eftervillkor på en överskuggad metod än vad superklassen har.

Uppgift 6.

```
public class Temperature implements TemperatureI{
    private State celcius = new Celcius();
    private State fahrenheit = new Fahrenheit();
    private State state = celcius;
    public void setCelcius() {
        state = celcius;
    }
    public void setFahrenheit() {
        state = fahrenheit;
    }
    public double convert(double v) {
        return state.convert(v);
    }
    public boolean freezing(double v) {
        return state.freezing(v);
    }
}//Temperature

public interface State {
    public double convert(double v);
    public boolean freezing(double v);
}
```

```
public class Celcius implements State {
    public double convert(double v) {
        return (v - 32.0) * 5.0 / 9.0;
    }
    public boolean freezing(double v) {
        return v <= 0.0;
    }
}

public class Fahrenheit implements State {
    public double convert(double v) {
        return 32 + 9.0 * v / 5.0;
    }
    public boolean freezing(double v) {
            return v <= 32.0;
    }
}
```

Uppgift 7.

```
public int hashCode() {
    return id;
}//hashCode
```

Uppgift 8.

Med användning av klassen Thread:

```
import java.io.FileNotFoundException;
public class SearcherThread extends Thread {
    private FileSearcher fs;
    private String pattern;
    public SearcherThread(FileSearcher fs, String pattern) {
        this.fs = fs;
        this.pattern = pattern;
    }//constructor
    public void run() {
        try {
            fs.search(pattern);
            if (fs.found())
                System.out.println(fs.getFileName());
        }
        catch (FileNotFoundException e) {}
    }//run
}// SearcherThread

public class FileSearch {
    public static void main(String[]arg) throws Exception {
        String pattern = arg[0];
        for(int i = 1; i < arg.length; i++) {
            FileSearcher fs = new FileSearcher(arg[i]);
            SearcherThread searcher = new SearcherThread(fs, pattern);
            searcher.start();
        }
    }//main
}//FileSearch
```

Med användning av interfacet Runnable:

```
import java.io.FileNotFoundException;
public class SearcherThread implements Runnable {
    private FileSearcher fs;
    private String pattern;
    private Thread searcher = new Thread(this);
    public SearcherThread(FileSearcher fs, String pattern) {
        this.fs = fs;
        this.pattern = pattern;
         searcher.start();
    }//constructor
    public void run() {
        try {
            fs.search(pattern);
            if (fs.found())
                System.out.println(fs.getFileName());
        }
        catch (FileNotFoundException e) {}
    }//run
}// SearcherThread

public class FileSearch {
    public static void main(String[]arg) throws Exception {
        String pattern = arg[0];
        for(int i = 1; i < arg.length; i++) {
            FileSearcher fs = new FileSearcher(arg[i]);
            SearcherThread searcher = new SearcherThread(fs, pattern);
        }
    }//main
}// FileSearch
```

Uppgift 9.

a)

```
import java.util.*;
public class Person extends Observable implements Observer, Comparable <Person> {
    private NameType name;
    private PhoneNumber phonenumber;
    private Map<Person, PhoneNumber> phonelist = new TreeMap<Person, PhoneNumber>();

    public Person(NameType name, PhoneNumber phonenumber) {
        super();
        this.name = name;
        this.phonenumber  = phonenumber;
    }

    public NameType getName() {
        return name;
    }

    public void setPhoneNumber(PhoneNumber phonenumber) {
        this.phonenumber = phonenumber;
        setChanged();
        this.notifyObservers(this.phonenumber);
    }

    public void update( Observable p, Object obj) {
        if ((p instanceof Person) && (obj instanceof PhoneNumber)) {
            Person person = (Person) p;
            PhoneNumber phone = (PhoneNumber) obj;
            phonelist.put(person, phone);
        }
    }

    //more fields and methodes not shown here
}//Person
```

b)

Eftersom namnet skall skrivas ut sorterade i alfabetisk ordning skall en TreeMap väljas. Deklarationen blir alltså

```
    private Map<Person, PhoneNumber> phonelist = new TreeMap<Person, PhoneNumber>();


    public void printPhoneList() {
        for (Map.Entry<Person, PhoneNumber> e : phonelist.entrySet()) {
            System.out.println(e.getKey() + ": " + e.getValue());
        }
    }
```