Exempeltentamen TDA550 - LÖSNINGSFÖRSLAG

Uppgift 1.

a) Utskriften blir:

A.f
C.f

b)

i) anropet obj1.h() ger kompileringsfel eftersom den statiska typen Base inte har metoden h.

ii) anropet obj2.h() ger kompileringsfel eftersom den statiska typen Base inte har metoden h.

iii) anropet obj3.g() ger kompileringsfel eftersom den statiska typen Int inte har metoden g.

Uppgift 2.

a) Koden bryter mor *Dependency Inversion Principle* genom att den beror av specifika klasser och metodnamn snarare än ett gemensamt gränssnitt. Ett symptom är fallanalysen i metoden getMemoryUsage som måste skrivas om vid tillägg av nya minnesenheter.

b)

```
public abstract class MemoryUnit {
   public abstract long getMemoryUsage();
}

public class USBStick extends MemoryUnit {
   public USBStick(long x) { ... }
   public long getMemoryUsage() { ... }
}

public class DVDMemory extends MemoryUnit {
   public DVDMemory(long x) { ... }
   public long getMemoryUsage() { ... }
}

public class SSDUnit extends MemoryUnit {
   public SSDUnit (long x) { ... }
   public long getMemoryUsage() { ... }
}

public class Main {
   public static long getMemoryUsage(List<MemoryUnit> units) {
      long usage = 0;
      for (MemoryUnit u : units)
         usage += u.getMemoryUsage();
      return usage;
   }

   public static void main(String[] args) {
      List<MemoryUnit> units = new LinkedList<>();
      units.add(new USBStick(1000));
      units.add(new DVDMemory(80000));
      units.add(new SSDUnit(20000));
      System.out.println(getMemoryUsage(units));
   }
}
```

Uppgift 3.

a) Klassen BadMain bryter mot *Open/closed-principen* som säger att en klass skall vara öppen för tillägg men stängd för ändringar.

b) Inför först fabriksklassen

```
public class PriceCalculatorFactory {
    public static PriceCalculator createPriceCalculator(String country) {
        if ( PriceCalculator.isBrittish(country) )
            return new GBPCalculator(country);
        else if ( PriceCalculator.isEuropean(country) )
            return new EUROCalculator(country);
        else
            return new USDCalculator(country);
    }
}
```

och skriv om klassen BadMain till GoodMain:

```
public class GoodMain {
    public static void main(String[] arg) {
        String country = arg[0];
        PriceCalculator priceCalculator = PriceCalculatorFactory.createPriceCalculator(country);
        float subTotal = 12345.0F;  // USD (priset antas vara satt i dollar)
        float weight = 123.3F;
        System.out.println(priceCalculator.calculateTotalPrice(subTotal,weight,country));
    }
}
```

Vi ser att koden i main-metoden frikopplats från de valutaspecifika klasserna. Objekt av dem kan skapas utan att namnge deras klasser. Vid tillägg av nya valutor eller ändringar i implementeringsklasserna isoleras förändringarna till fabriksklassen.

Uppgift 4.

a) Kompileringsfel! Listan lef kan t.ex. vara av typen List<DinnerTable> och ett objekt av typen Table är ingen subtyp till Dinnertable.

b) Kompileringsfel! Listan lef kan t.ex. vara av typen List<DinnerTable> och ett objekt av typen Object är ingen subtyp till Dinnertable.

c) Ok! Listan lst kan vara av typen List<Table> eller List<Object>, och typen Table är subtyp till typen Object.

d) Kompileringsfel! Listan lst kan vara av typen List<Table> eller List<Object>, och typen Object är ingen subtyp till typen Table.

e) Kompileringsfel! Listan let kan vara av typen List<CoffeeTable>, och typen DinnerTable är ingen subtyp till typen CoffeeTable.

f) Ok! Typen Object är supertyp till alla andra typer.

g) Ok! Typen Furniture är supertyp till typen Table (och alla subtyper till Table).

h) Kompileringsfel! Listan let kan t.ex. vara av typen List<Table> och typen CoffeeTable är ingen subtyp till typen Table.

i) Kompileringsfel! Listan lst kan vara av typen List<Object> och typen Table är ingen supertyp till Object.

j) Ok! Typen Object är supertyp till alla andra typer.

Uppgift 5.

a) false => 0 > i && i < 10 => i < 20 => 40 < i || i < 30 => true

b) ArraySearchImpl1 har starkare specifikation än ArraySearch eftersom ArraySearchImpl1 har svagare förvillkor och starkare eftervillkor. Vi kan därför använda ArraySearchImpl1 i klienter som är kompatibla med ArraySearch.

ArraySearchImpl2 däremot har svagare specifikation än ArraySearch eftersom ArraySearchImpl2 har starkare förvillkor och svagare eftervillkor. ArraySearchImpl2 kan därför inte användas i en klient som förväntar sig en implementering kompatibel med ArraySearch.

Uppgift 6.

```
public class SparseVectorMap implements SparseVector {
  private Map<Integer, Double> map;
  private int size;

  public SparseVectorMap(int size) {
      this.size = size;
      map = new HashMap<Integer, Double>();
  }

  public SparseVectorMap(double[] v) {
      this.size = v.length;
      map = new HashMap<Integer, Double>();
      for (int i = 0; i < v.length; i++) {
          if (v[i] != 0) {
              map.put(i, v[i]);
          }
      }
  }

  public void put(int index, double val) {
      if (index < 0 || index >= size)
          throw new IllegalArgumentException();
      if (val == 0) {
          map.remove(index);
      } else {
          map.put(index, val);
      }
  }

  public double get(int index) {
      if (index < 0 || index >= size)
          throw new IllegalArgumentException();
      Double d = map.get(index);
      if (d == null) {
          return 0.0;
      } else {
          return d;
      }
  }

  public int size() {
      return size;
  }

  public double dot(SparseVector v) {
      if (size != v.size())
          throw new IllegalArgumentException("Vector lengths are different");
      double sum = 0;
      for (Map.Entry<Integer, Double> e : map.entrySet()) {
          sum += e.getValue() * v.get(e.getKey());
      }
      return sum;
  }
}
```

Uppgift 7.

Med användning av Observer och Observable:

```
import java.util.Observable;
public class Clock extends Observable {
    private Time time;

    public void tick() {
        time.increase();
        setChanged();
        notifyObservers();
    }

    public Time getTime() {
        return time;
    }

    //more methods and constructors not shown here
}
```

```
import java.util.Observer;
import java.util.Observable;
public class Display implements Observer {
    private Clock clock;

    public Display(Clock clock) {
        this.clock = clock;
        clock.addObserver( this );
    }

    private void show(Time time) {
        ...
    }

    public void update(Observable obs, Object o) {
        show(clock.getTime());
    }
}
```

Klassen Console redovisas här med en alternativ lösning:

```
import java.util.Observer;
import java.util.Observable;
public class Console implements Observer {

    public Console(Clock clock) {
        clock.addObserver(this);
    }

    private void print(Time time) {
        ...
    }

    public void update(Observable obs, Object o) {
        print(((Clock) obs).getTime());
    }
}
```

Med användning av PropertyChangeSupport och PropertyChangeListener:

```
import java.beans.PropertyChangeSupport;
import java.beans.PropertyChangeListener;
public class Clock {
  private Time time;
  private final PropertyChangeSupport pcs = new PropertyChangeSupport(this);

  public void addPropertyChangeListener(PropertyChangeListener listener) {
      this.pcs.addPropertyChangeListener(listener);
  }

  public void removePropertyChangeListener(PropertyChangeListener listener) {
      this.pcs.removePropertyChangeListener(listener);
  }

  public void tick() {
      time.increase();
      pcs.firePropertyChange("", true, false);
  }

  public Time getTime() {
      return time;
  }

  //more methods and constructors not shown here

}
```

```
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeEvent;
public class Display implements PropertyChangeListener {
  private Clock clock;

  public Display(Clock clock) {
      this.clock = clock;
      clock.addPropertyChangeListener(this);
  }

  private void show(Time time) {
      ...
  }

  public void propertyChange(PropertyChangeEvent e) {
      show(clock.getTime());
  }
}
```

Klassen Console redovisas här med en alternativ lösning:

```
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeEvent;
public class Console implements PropertyChangeListener {

  public Console(Clock clock) {
      clock.addPropertyChangeListener(this);
  }

  private void print(Time time) {
      ...
  }

  public void propertyChange(PropertyChangeEvent ev) {
      print(((Clock2) ev.getSource()).getTime());
  }
}
```

Uppgift 8.

```
public class EnemyRobotAdapter implements EnemyAttacker {
  private EnemyRobot theRobot = new EnemyRobot();
  public EnemyRobotAdapter(EnemyRobot newRobot){
      theRobot = newRobot;
  }
  @Override
  public void fireWeapon() {
      theRobot.smashWithHand();
  }
  public void driveForward() {
      theRobot.walkForward();
  }
  public void assignDriver(String driver) {
      theRobot.reactToHuman(driver);
  }
}
```

Uppgift 9.

a) Metoden equals är korrekt skriven.

Det första problemet med implementeringen av klassen är att klassinvarianten inte är uppfylld. För det första bör undantaget IllegalStateException kastas i konstruktorn om q är noll eftersom nämnaren inte kan vara noll i ett rationellt tal.

För det andra tillåter klassen oändligt många olika representationer av samma rationella tal. T.ex. kan 1/3 lagras som 2/6, 15/45, -1/-3, o.s.v. En lösning är att låta konstruktorn reducera bråket genom att förkorta det så långt det går, samt bestämma att negativa tal alltid representeras med negativ täljare och positiv nämnare. Då uppfylls dessutom klassinvarianten.

Skriv alltså om konstruktorn:

```
public Rational(int p, int q) {
    this.p = p;
    if ( q == 0 )
        throw new IllegalStateException("Zero denominator");
    this.q = q;
    normalize();
}
```

och utför förkortningen av bråket med gcd, samt justera tecknet:

```
private void normalize() {
    int gcd = gcd(p,q);
    p /= gcd;
    q /= gcd;
    if ( q < 0 ) {
        p = -p;
        q = -q;
    }
}
```

Ex gcd(20,28) är 4 och bråket kan förkortas till 5/7.

b) Klassen måste överskugga hashCode-metod:

```
public int hashCode() {
    return 31*p + q;
}
```

c) Deklarera först

public class Membership **implements** Cloneable

och inför en clone-metod

```
public Membership clone() {
    Membership result = null;
    try {
        result = (Membership)super.clone();
        result.someObjects = (LinkedList<SomeClass>)someObjects.clone();
        for (int i = 0; i < someObjects.size(); i++ )
            result.someObjects.set(i, someObjects.get(i).clone());
    }
    catch ( CloneNotSupportedException e) {
        throw new InternalError();
    }
    return result;
}
```

Uppgift 10.

a) Sub2 och Sub3 är typkorrekta.

I Sub1 deklarerar f ett undantag av vidare typ än basklassens f. En sådan överskuggning är inte tillåten.

I Sub4 kastar f ett undantag av vidare typ än den deklarerar och det är förstås inte tillåtet.

b) Man kan få vilken som helst av utskrifterna (men bara en):

- g succeeded
- catched E2
- catched E4
- catched E1

Base.g måste inte kasta ett undantag men kan om den vill, kasta vilket som helst av E1, E2, E3 och E4.

Undantagen E2 och E3 fångas av **catch** (E2 e), E4 av **catch** (E4 e), samt E1 av den yttre **catch** (E1 e).