Övning vecka 3.

Denna vecka ska vi titta på polymorfism, dynamisk kontra statisk bindning, implementationsarv, equals samt något om klassinvarianter.

Uppgift 1 Liskovs substitutionsprincip (LSP)

Liskovs substitutionsprincip (LSP) säger att alla objekt av en subtyp U ska kunna fungera som substitut för objekt av vilken som helst av U:s supertyper.

a) Betrakta klasserna IntegerClass och ObjectClass nedan.

```
public class IntegerClass {
    public static Integer square(int x) {
        return new Integer(x * x);
    }
}//IntegerClass
```

```
public class ObjectClass {
    public static Object square(int x) {
        return new Integer(x * x);
    }
}//ObjectClass
```

Vi ser att klasserna är så gott som identiska. Är det möjligt att göra en ny design där vi låter den ena klassen vara en subklass till den andra klassen? Om så är fallet, spelar det någon roll om det är IntegerClass som är subklass till ObjectClass eller tvärtom?

b) Betrakta klasserna SuperClass och SubClass nedan.

```
public class SuperClass {
    private final int value;
    public SuperClass(int value) {
        this.value = value;
    }
    public int getValue() {
        return value;
    }
}//SuperClass
```

```
public class SubClass extends SuperClass {
    public SubClass(int x) {
        super(x);
    }
    private int getValue() {
        return super.getValue();
    }
    public int getSquare() {
        return getValue() * getValue();
    }
}//SubClass
```

När vi använder objekt av klassen SubClass vill vi inte att man ska kunna komma åt objektets värdet direkt utan bara dess kvadrat. Detta har vi försökt åstadkomma genom att deklarera metoden getValue i SubClass som **private**. Varför går det inte att kompilera klassen SubClass?

Uppgift 2 Polymorfism och dynamisk typ

a) Betrakta klasserna Binding1, S, A och B nedan:

```
public class Binding1 {
   public static void printValue(A v) {
      System.out.println("This was an A: " + v.toString());
   }
   public static void printValue(B v) {
      System.out.println("This was a B: " + v.toString());
   }
   public static void printValue(S v) {
      System.out.print("That was an S");
      if (v.getClass() != S.class) {
         System.out.print(", subclassed by " + v.getClass() + ": ");
      } else {
         System.out.print(", not a strict subclass of S." + ": ");
      }
      System.out.println(v.toString());
   }
   public static void main(String[ ] args) {
      S spa = new S();
      A apa = new A();
      B bepa = new B();
      S apalt = apa;
      A bepalt = bepa;
      printValue(spa);
      printValue(apa);
      printValue(bepa);
      printValue(apalt);
      printValue(bepalt);
   }
}//Binding1

public  class S {
   public String toString() {
      return "I am an S!";
   }
}//S

public  class A extends S {
   public String toString() {
      return "I am an A!";
   }
}//A

public  class B extends A {
   public String toString() {
      return "I am a B!"; }
}//B
```

Klassen Binding1 innehåller metoder för utskrift av objekt av typerna S, A och B.

Alla klasser i Java är subklasser till klassen Object som innehåller metoden toString(), och klasserna S, A och B *överskuggar* denna metod. I klassen Binding1 är metoden printValue *överlagrad* och de olika versionerna skiljer sig åt med avseende på typen hos den formella parametern.

Vad kommer utskriften att bli när vi kör Binding1.main? Vilka metoder i vilka klasser kommer att exekveras vid körningen av Binding1.main och framförallt; varför anropas just dessa metoder?

b) Betrakta klassen Binding2 nedan:

```
public class Binding2 {
    public static void main(String[ ] args) {
        S[ ] objs1 = new S[ ] { new A(), new B(), new S() };
        for (S o : objs1) {
            Binding1.printValue(o);
        }
        A[ ] objs2 = new A[ ] { new A(), new B() };
        for (S o : objs2) {
            Binding1.printValue(o);
        }
        for (A o : objs2) {
            Binding1.printValue(o);
        }
    }
}//Binding2
```

Antag att klasserna A, B och S är de samma som definierades i deluppgift a). Vad blir utskriften när Binding2.main körs? Varför?

Uppgift 3 Klassinvarianter & aliasing

Betrakta klassen Period nedan:

```
/**
* @invariant getStart() is before or at the same time as getEnd()
* @invariant getStart() consistently returns the same value after object creation
* @invariant getEnd() consistently returns the same value after object creation
*/
import java.itil.Date;
public final class Period {
    private final Date start;
    private final Date end;
    /**
    * @param start the beginning of the period
    * @param end the end of the period; must not precede start
    * @pre start <= end
    * @post The time span of the returned period is positive.
    * @throws IllegalArgumentException if start is after end
    * @throws NullPointerException if start or end is null
    */
    public Period(Date start, Date end) {
        if (start.compareTo(end) > 0) {
            throw new IllegalArgumentException(start + " after " + end);
        }
        this.start = start;
        this.end = end;
    }
    public Date getStart() {
        return start;
    }
    public Date getEnd() {
        return end;
    }
}//Period
```

Klassen Period representerar ett datumintervall genom att innehålla ett startdatum och ett slutdatum. I kommentarerna till klassen visas dessutom vilket kontrakt implementationen av klassen måste uppfylla ("Programming by Contract"). En *invariant* är ett villkor som alltid måste vara uppfyllt, ett *precondition* utgör krav som måste vara uppfyllda innan en metod anropas och *postcondition* visar samband som gäller efter att metoden körts.

Några klassinvarianter för Period innebär att avsikten är att klassen skall vara *icke-muterbar*. Vilka? Att en klass är icke-muterbar innebär att när ett objekt av klassen väl instansierats så kommer objektet under resten av sin livslängd att ha samma värden som då det skapades.

Har programmeraren gjort sitt jobb eller finns det sätt att bryta mot klassens invarianter? Om så är fallet, vad blir konsekvensen och hur kan vi åtgärda felen?

Uppgift 4 Likhet (equals)

Java Language Specification säger att metoden equals skall uppfylla följande relationer;

Reflexivitet:	a.equals(a)
Symmetri:	a.equals(b) ⇒ b.equals(a)
Transitivitet:	a.equals(b) && b.equals(c) ⇒ a.equals(c)

a) Betrakta klasserna CaseInsensitiveString och MyString nedan:

```
public class MyString {
   private final String s;
   public MyString(String s) {
      if (s == null) {
         throw new NullPointerException();
      }
      this.s = s;
   }
   @Override
   public boolean equals(Object o) {
      if (o instanceof MyString) {
         MyString os = (MyString) o;
         return s.equals(os.s);
      }
      return false;
   }
   public String asString() {
      return s;
   }
}//MyString
public final class CaseInsensitiveString extends MyString {
   public CaseInsensitiveString(String s) {
      super(s);
   }
   @Override
   public boolean equals(Object o) {
      if (o instanceof MyString) {
         MyString os = (MyString) o;
         if (os instanceof CaseInsensitiveString) {
            return asString().equalsIgnoreCase(os.asString());
         }
         return super.equals(o);
      }
      return false;
   }
}
```

Finns det några omständigheter under vilka de tre önskvärda relationerna inte gäller för klassen? Vad händer om man jämför med en MyString? Med en CaseInsensitiveString? Spelar ordningen någon roll? Om det finns något problem, vad borde man gjort istället?

b) Betrakta klasserna Point och NamedPoint nedan:

```
public class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    @Override
    public boolean equals(Object o) {
        if (!(o instanceof Point)) {
            return false;
        } else {
        Point p = (Point) o;
        return p.x == x && p.y == y;
        }
    }
}//Point

public class NamedPoint extends Point {
    private final String name;
    public NamedPoint(int x, int y, String name) {
        super(x, y);
        this.name = name;
    }
    @Override
    public boolean equals(Object o) {
        if (!(o instanceof Point)) {
            return false;
        }
        // If o is a normal Point, do a name-blind comparison
        if (!(o instanceof NamedPoint)) {
            return o.equals(this);
        }
        // o is a NamedPoint; do a full comparison
        return super.equals(o) && ((NamedPoint) o).name == name;
    }
}// NamedPoint
```

Håller relationerna reflexivitet, symmetri och transivitet för equals för dessa två klasser? Om inte, vilken eller vilka relationer håller inte? Om någon av relationerna inte håller, hur borde man implementerat equals istället?