

## TENTAMEN: Objektorienterad programutveckling, fk

### Läs detta!

- *Uppgifterna är inte ordnade efter svårighetsgrad.*
- Börja varje hel uppgift på ett nytt blad. Skriv inte i tesen.
- Ordna bladen i uppgiftsordning.
- Skriv din tentamenskod på varje blad (så att vi inte slarvar bort dem).
- **Skriv rent dina svar. Oläsliga svar r ä t t a s e j!**
- Programkod som finns i tentamenstesen behöver ej upprepas.
- Programkod skall skrivas i Java 5, eller senare version, och vara indenterad och renskriven.
- Onödigt komplicerade lösningar ger poängavdrag.
- Givna deklARATIONER, parameterlistor etc. får ej ändras.
- Läs igenom tentamenstesen och förbered ev. frågor.

I en uppgift som består av flera delar får du använda dig av funktioner klasser etc. från tidigare deluppgifter, även om du inte löst dessa.
--

*Lycka till!*

## Uppgift 1

I en viss applikation finns följande klass

```
public class Analyzer {  
    public boolean isGood(int x) {  
        //Code omitted  
    }  
    public void use(int x) {  
        //Code omitted  
    }  
}
```

I samma applikation finns klassen

```
public class Client extends Analyzer {  
    public void decide(int x) {  
        if ( isGood(x) )  
            use(x);  
    }  
}
```

Man har senare kommit fram till att `Client` även borde ärva från

```
public class SomeBaseClass { ... }
```

Förutsättningen är nu att `Analyzer` inte får ändras och eftersom Java som bekant inte tillåter multipelt implementeringsarv kan inte lösningen baseras på implementeringsarv. Uppgiften är att istället låta klientklassen utnyttja `Analyzer` genom delegering. Som ledning får du följande gränssnitt:

```
public interface Analyzable {  
    boolean isGood(int x);  
    void use(int x);  
}
```

Klassen `Client` skall efter refaktoreringen ha metoderna `isGood`, `use` samt `decide`.

(8 p)

## Uppgift 2

I ett program som utför prisjämförelser för restauranger finns följande klasser:

```
public class Menu {
    public int getNoOfDishes() { ... }
    public void add(Dish d) { ... }
    public Dish getDish(int i) { ... }
}

public class Dish {
    public String getDescription() { ... }
    public int getPrice() { ... }
}

public class BreakfastMenu extends Menu { ... }
public class LunchMenu extends Menu { ... }
public class DinnerMenu extends Menu { ... }

public class CustomerClient {
    public List<Dish> getDishesBelow(int hour,int priceLimit) {
        Menu menu = null;
        if ( hour >= 5 && hour < 11 )
            menu = new BreakfastMenu();
        else if ( hour < 15 )
            menu = new LunchMenu();
        else if ( hour < 21 )
            menu = new DinnerMenu();
        if ( menu != null ) {
            List<Dish> dishes = new ArrayList<>();
            for ( int i = 0; i < menu.getNoOfDishes(); i++ )
                if ( menu.getDish(i).getPrice() <= priceLimit )
                    dishes.add(menu.getDish(i));
            return dishes;
        }
        return null;
    }
}
```

a) CustomerClient bryter mot en viktig designprincip, vilken?

(1 p)

b) Refaktorera CustomerClient genom att tillämpa designmönstret *Factory Method*. Du får lägga till en ny klass.

(6 p)

### Uppgift 3

Studera klasserna Membership och Club.

```
public class Membership {  
    private String name;  
    private String email;  
    private int age;  
    public Membership(String name,String email,int age) {  
        ...  
    }  
    public int getAge() { return age; }  
}
```

```
public class Club {  
    private List<Membership> members = new ArrayList<>();  
    public void join(Membership m) {  
        members.add(m);  
    }  
    public List<Membership> getMembers() {  
        return members;  
    }  
    ...  
}
```

Klassen Club bryter mot principen att en klass inte bör exponera interna dataobjekt, vilket sker då metoden `get` returnerar en referens till listan med medlemmarna. Ett bättre sätt är att låta användaren inspektera listan via en iterator, en teknik som belysts i kursen. Eftersom listor är itererbara kan vi utnyttja det för att slippa att definiera en egen iterator-klass. Ett problem uppkommer då eftersom listklassens iterator har metoden `remove`, och vi vill inte att en användare som bara tillåts iterera igenom listan skall kunna ta bort medlemmar.

En lösning på problemet är att skapa klassen `ReadOnlyIterator`. En iterator av denna typ ger i kombination med en annan iterator en som kastar `UnsupportedOperationException` då metoden `remove` anropas.

a) Implementera klassen `ReadOnlyIterator` genom att utnyttja designmönstret *Decorator*.

(5 p)

b) Ändra klassen `Club` så att den blir itererbar genom att erbjuda användaren en iterator av typen i a). Exempel på användning:

```
Club c = new Club();  
c.join(new Membership("Lisa", "lisa@coldmail.moc", 23));  
c.join(new Membership("Allan", "allan@foomail.bar", 21));  
c.join(new Membership("Emilia", "emil@spammail.urk", 22));  
Iterator<Membership> it = c.iterator();  
while ( it.hasNext() ) {  
    Membership m = it.next();  
    if ( m.getAge() > 21 )  
        ; // do something with m  
    else  
        it.remove(); // throws UnsupportedOperationException  
}
```

(2 p)

## Uppgift 4

Studera följande klasser:

```
public class Base {
    protected String txt;
    public Base() { txt = ""; }
    public Base(String txt) { this.txt = txt; }
    public String toString() { return "Base("+txt+")"; }
    public static void f(int x) { System.out.println("Base.f("+x+)"); }
    public static void f(int x,int y) { System.out.println("Base.f("+x+", "+y+)"); }
    public void g(int x) { System.out.println("Base.g("+x+)"); }
    public void g(int x,int y) { System.out.println("Base.g("+x+", "+y+)"); }
    public void h(Base ref) { System.out.println("Base.h("+ref+)"); }
    public void h(Sub ref) { System.out.println("Base.h("+ref+)"); }
}

public class Sub extends Base {
    public Sub() { super(); }
    public Sub(String txt) { super(txt); }
    public String toString() { return "Sub("+txt+")"; }
    public static void f(int x) { System.out.println("Sub.f("+x+)"); }
    public void g(int x) { System.out.println("Sub.g("+x+)"); }
    public void h(Sub ref) { System.out.println("Sub.h("+ref+)"); }
}
```

Vad skriver följande kod ut? Du får 0.5 p per rätt svar. Vilka principer bestämmer vilken metod som anropas (3 p)?

```
Base ref = new Base();
ref.f(1);           // 1
ref.f(2,3);        // 2
ref.g(4);          // 3
ref.g(5,6);        // 4

ref = new Sub();
ref.f(1);           // 5
ref.f(2,3);        // 6
ref.g(4);          // 7
ref.g(5,6);        // 8
...
```

```
Sub ref2 = new Sub();
ref2.f(1);          // 9
ref2.f(2,3);        // 10

ref = new Sub("apa");
ref.h(new Base("bepa")); // 11
ref.h(new Sub("cepa"));  // 12
```

(9 p)

## Uppgift 5

Studera följande gränssnitt:

```
public interface MinFindable1 {  
    /**  
     * Find a minimal element in a.  
     * @Pre True  
     * @return The smallest index of a minimal element in a.  
     * @throws IllegalArgumentException if a is null or refers to an empty array.  
     */  
    int indexOfMin(int[] a) throws IllegalArgumentException;  
}  
public interface MinFindable2 {  
    /**  
     * Find a minimal element in a.  
     * @Pre a != null && a.length > 0  
     * @return The index of some arbitrary minimal element in a.  
     */  
    int indexOfMin(int[] a);  
}
```

a) I vilket av gränssnitten har metoden `indexOfMin` starkast specifikation? Motivera svaret!

(2 p)

*Uppgift b) finns på nästa sida →*

- b) Ange för varje kombination av  $i$  och  $j$  om  $\text{MinFinder}_i$  kan vara en äkta subtyp till  $\text{MinFindable}_j$  (åtta fall). Motivera svaren!

(4 p)

```
public class MinFinder1 implements MinFindableX {
    public int indexOfMin(int[] a) {
        int minPos = 0;
        int minElement = a[0];
        for ( int i = 1; i < a.length; i++ )
            if ( a[i] < minElement ) {
                minElement = a[i];
                minPos = i;
            }
        return minPos;
    }
}
```

```
public class MinFinder2 implements MinFindableX {
    public int indexOfMin(int[] a) {
        if ( a == null || a.length == 0 )
            return -1;
        int minPos = 0;
        int minElement = a[0];
        for ( int i = 1; i < a.length; i++ )
            if ( a[i] <= minElement ) {
                minElement = a[i];
                minPos = i;
            }
        return minPos;
    }
}
```

```
public class MinFinder3 implements MinFindableX {
    public int indexOfMin(int[] a) {
        if ( a == null || a.length == 0 )
            throw new IllegalArgumentException();
        int minPos = 0;
        int minElement = a[0];
        for ( int i = 1; i < a.length; i++ )
            if ( a[i] < minElement ) {
                minElement = a[i];
                minPos = i;
            }
        return minPos;
    }
}
```

```
public class MinFinder4 implements MinFindableX {
    public int indexOfMin(int[] a) {
        int minPos = 0;
        int minElement = a[0];
        for ( int i = 1; i < a.length; i++ )
            if ( a[i] <= minElement ) {
                minElement = a[i];
                minPos = i;
            }
        return minPos;
    }
}
```

## Uppgift 6

En *ValueMap* är en associativ datasamling som associerar nycklar med samlingar av värden. T.ex. kan en sådan samling användas för att lagra temperaturavläsningar för årets månader.

```
public interface ValueMap<K,V> {  
    /**  
     * Adds value to the collection associated with key.  
     */  
    void addValue(K key,V value);  
    /**  
     * Returns the collection of values associated with key  
     * and null if key is unknown.  
     */  
    Collection<V> getValues(K key);  
    /**  
     * Returns the set of keys associated with a collection containing value  
     * and null if there is no such collection.  
     */  
    Set<K> getKeys(V value);  
}
```

Exempel:

```
ValueMap<String,Integer> monthTemps = new SortedValueMap<>(); // Uppgift b  
monthTemps.addValue("January",2);  
monthTemps.addValue("January",-1);  
monthTemps.addValue("February",-1);  
monthTemps.addValue("March",8);  
monthTemps.addValue("February",4);  
monthTemps.addValue("March",2);  
monthTemps.addValue("January",2);  
  
monthTemps.getValues("January") → [2,-1,2]  
monthTemps.getKeys(2) → {"January","March"} // Bokstavsordning
```

- a) Konstruera klassen *SortedValueMap*. Den skall förstås implementera gränssnittet *ValueMap*. Mappen skall vara ordnad med avseende på nycklarna.

(4 p)



b) Data lagras ofta i s.k. CSV-filer (Comma Separated Values). CSV-filer är textfiler.

Ex. Filen data.txt:

```
January,7,6,2,5,3,9,2,2,4  
February,1,2,0,8,4,4,7,1  
March,5,3,10,8,4,6,9  
January,3,0,7,1
```

I fallet ovan är alltid första strängen en nyckel, följd av ett antal kommaseparerade värden.

Konstruera metoden readCSVFile:

```
public static ValueMap<String,Integer> readCSVFile(String fileName)  
throws IOException,NumberFormatException {  
    /* TODO */  
}
```

Metoden skall läsa textfilen med angivet namn och bygga en SortedValueMap med innehåll baserat på innehållet i filen. Läs filen radvis.

(3 p)

## Uppgift 7

Nedan finns utdrag ur två klasser för hantering av geometriska objekt.

```
public class Point {  
    private int x,y;  
    public Point(int x,int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public int getX() { return x; }  
    public int getY() { return y; }  
    public void setPosition(int x,int y) {  
        this.x = x;  
        this.y = y;  
    }  
    // other methods omitted  
}
```

```
public class GeometricalObject {  
    private Point position;  
    public GeometricalObject(Point position) {  
        this.position = position;  
    }  
    public Point getPosition() {  
        return position;  
    }  
    // other methods omitted  
}
```

- a) Visa med ett kodexempel att klassen GeometricalObject är muterbar trots att den saknar muterande metoder.

(2 p)

- b) Addera lämplig metod i Point och modifiera koden i GeometricalObject så att denna ej längre är muterbar.

(4 p)

## Uppgift 8

Följande klasser utgör en datamodell för aritmetiska uttryck som t.ex. 42, X, X+23, -(y+foo\*bar). Uttryck kan alltså innehålla heltalskonstanter, variabler samt operationerna addition, multiplikation samt negation (unärt minus). För att kunna räkna ut värdet på uttryck som innehåller variabler brukar man använda en s.k. *omgivning* (eng. *environment*) vilket är en tabell med värdetilldelningar för de variabler som förekommer i uttrycket. Omgivningar specificeras av gränssnittet Environment.

```
public interface Environment {  
    void assign(String variable,int value);  
    int getValue(String variable) throws  
    UndefinedVariableException;
```

```
public class UndefinedVariableException extends RuntimeException  
{  
    public UndefinedVariableException(String message) {  
        super(message);  
    }  
}
```

```
public class Expression {  
    private String type;  
    private int value;  
    private String variable;  
    private String operator;  
    private Expression leftOperand;  
    private Expression rightOperand;  
    private Expression unaryOperand;  
    public Expression(int value) {  
        type = "CONSTANT";  
        this.value = value;  
    }  
    public Expression(String variable) {  
        type = "VARIABLE";  
        this.variable = variable;  
    }  
    public Expression(String operator,Expression unaryOperand) {  
        type = "UNARYEXPRESSION";  
        this.operator = operator;  
        this.unaryOperand = unaryOperand;  
    }  
    public Expression(String operator,Expression leftOperand,Expression rightOperand)  
    {  
        type = "BINARYEXPRESSION";  
        this.operator = operator;  
        this.leftOperand = leftOperand;  
        this.rightOperand = rightOperand;  
    }  
    public String getType() { return type; }  
    public int getConstantValue() { return value; }  
    public String getVariable() { return variable; }  
    public String getOperator() { return operator; }  
    public Expression getUnaryOperand() { return unaryOperand; }  
    public Expression getLeftOperand() { return leftOperand; }  
    public Expression getRightOperand() { return rightOperand; }  
}
```

*forts. på nästa sida*

Här skapas några uttryck m.h.a. klasserna ovan och värdet beräknas med metoden `getValue`:

```
public class Main {
    public static void main(String[] arg) {
        Environment env = new SomeEnvironmentImplementation();
        env.assign("A",10);
        env.assign("B",20);
        env.assign("C",30);
        Expression e1 =
            new Expression("ADDITION",new Expression(42),new Expression("A"));
        Expression e2 =
            new Expression("MULTIPLICATION",new Expression("B"),new Expression("C"));
        Expression e3 =
            new Expression("NEGATION",new Expression("ADDITION",e1,e2));
        System.out.println(getValue(e3,env));
    }
    private static int getValue(Expression e,Environment env) {
        switch ( e.getType() ) {
            case "CONSTANT":
                return e.getConstantValue();
            case "VARIABLE":
                return env.getValue(e.getVariable());
            case "UNARYEXPRESSION":
                switch ( e.getOperator() ) {
                    case "NEGATION":
                        return -getValue(e.getUnaryOperand(),env);
                    default:
                        throw new IllegalStateException("Illegal operator found: "
                            + e.getOperator());
                }
            case "BINARYEXPRESSION":
                switch ( e.getOperator() ) {
                    case "ADDITION":
                        return getValue(e.getLeftOperand(),env) +
                            getValue(e.getRightOperand(),env);
                    case "MULTIPLICATION":
                        return getValue(e.getLeftOperand(),env) *
                            getValue(e.getRightOperand(),env);
                    default:
                        throw new IllegalStateException("Illegal operator found: "
                            + e.getOperator());
                }
            default:
                throw new IllegalStateException("Illegal expression type found: "
                    + e.getType());
        }
    }
}
```

*forts. på nästa sida*

Koden bryter mot flera designprinciper som behandlats i kursen, vilka? Refaktorera koden enligt designmönstret *Strategy* och använd gränssnitten:

```
public interface Expression {  
    int getValue(Environment env);  
}
```

```
public interface UnaryOperator {  
    int unOp(int operand);  
}
```

```
public interface BinaryOperator {  
    int binOp(int leftOperand,int rightOperand);  
}
```

Betr. operationerna räcker det om du behandlar addition och negation. Koduplicering skall givetvis undvikas.

*Tips:* Som ledning får du ett fragment av den nya main-metoden:

```
public static void main(String[] arg) {  
    Environment env = new SomeEnvironmentImplementation();  
    env.assign("A",10);  
    env.assign("B",20);  
    env.assign("C",30);  
    Expression e1 = new Addition(new Constant(42), ...  
    Expression e2 = ...  
    Expression e3 = ...  
    System.out.println(e3.getValue(env));  
}
```

(10 p)

## Lösningsförslag till tentamen

## *P r e l i m i n ä r*

**Kurs**  
**Tentamensdatum**  
**Program**  
**Läsår**  
**Examinator**

**Objektorienterad programutveckling, fk**  
**2019-01-18**  
**TKIEK-2,TKTFY-3,TKTEM-3**  
**2018/2019, lp 2**  
**Uno Holmer**

---

### Uppgift 1 (8 p)

```
public class Client extends SomeBaseClass implements Analyzable {
    private Analyzer analyzer;
    public Client() {
        analyzer = new Analyzer();
    }
    @Override
    public boolean isGood(int x) {
        return analyzer.isGood(x);
    }
    @Override
    public void use(int x) {
        analyzer.use(x);
    }
    public void decide(int x) {
        if ( isGood(x) )
            use(x);
    }
}
```

### Uppgift 2 (1+6 p)

a)

Klassen `CustomerClient` bryter mot principerna DIP och OCP. Klassen beror av specifika menyklasser och måste modifieras om till exempel måltiderna brunch, mellanmål eller nattvickning tillkommer. Lösningen nedan kan dock inte hantera mellanmål eftersom sådana är notoriskt svåra att tidsbestämma. Huruvida klassen även bryter mot Law of Demeter är kanske ett gränsfall.

b)

```
public class MenuFactory {
    public static Menu createMenu(int hour) {
        if ( hour >= 5 && hour < 11 )
            return new BreakfastMenu();
        else if ( hour < 15 )
            return new LunchMenu();
        else if ( hour < 21 )
            return new DinnerMenu();
        else
            return null;
    }
}
public class CustomerClient {
    public List<Dish> getDishesBelow(int hour,int priceLimit) {
        Menu menu = MenuFactory.createMenu(hour);
        if ( menu != null ) {
            List<Dish> dishes = new ArrayList<>();
            for ( int i = 0; i < menu.getNoOfDishes(); i++ )
                if ( menu.getDish(i).getPrice() <= priceLimit )
                    dishes.add(menu.getDish(i));
        }
    }
}
```

```
        return dishes;
    }
    return null;
}
}
```

### Uppgift 3 (5+2 p)

a)

```
public class ReadOnlyIterator<E> implements Iterator<E> {
    private Iterator<E> it;
    public ReadOnlyIterator(Iterator<E> it) {
        this.it = it;
    }
    @Override
    public boolean hasNext() {
        return it.hasNext();
    }
    @Override
    public E next() {
        return it.next();
    }
    @Override
    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

b)

```
public class Club implements Iterable<Membership> {
    private List<Membership> members = new ArrayList<>();
    public void join(Membership m) {
        members.add(m);
    }
    public Iterator<Membership> iterator() {
        return new ReadOnlyIterator(members.iterator());
    }
}
```

### Uppgift 4 (9 p)

Det är tre principer som bestämmer vilken metod som anropas i fallen 1-12 (nästa sida):

1. *Statisk typ, statisk bindning.* För klassmetoder bestämmer referensvariabelns statiska typ vilken metod som anropas. I första hand väljs metoden (vid kompileringen) i den statiska typen om den finns där, annars i den närmast ovanliggande basklassen till denna som definierar metoden.
2. Kompilatorn utgår från referensvariabelns statiska typ för att hitta en lämplig metods signatur som matchar parametertyperna i anropet. I första hand väljs metoden (vid kompileringen) i den statiska typen om den finns där, annars i den närmast ovanliggande basklassen till denna som definierar metoden.
3. *Polymorfism, dynamisk typ, dynamisk bindning.* När kompilatorn matchat anropet mot möjliga metods signaturer och bestämt vilken metod som är tillämpbar börjar sökningen (under exekveringen) efter en lämplig metod i referensvariabelns dynamiska typ (det utpekade objektets typ). Hittas inte metoden där fortsätter sökningen uppåt i klasshierarkin tills metoden hittas.

Utskrifterna blir:

1. Base.f(1)
2. Base.f(2,3)
3. Base.g(4)
4. Base.g(5,6)
  
5. Base.f(1)
6. Base.f(2,3)
7. Sub.g(4)
8. Base.g(5,6)
  
9. Sub.f(1)
10. Base.f(2,3)
11. Base.h(Base(bepa))
12. Sub.h(Sub(cepa))

### Uppgift 5 (2+4 p)

- a) Metoden `indexOfMin` har starkast specifikation i `MinFindable1` eftersom den har ett svagare förvillkor och ett starkare eftervillkor.
- b)
1. `MinFinder1` är inte en äkta subtyp till `MinFindable1`  
Specifikationen är tydlig om vad som skall hända om fältreferensen `a` är `null` eller pekar på ett tomt fält, men om `a` är `null` kastar metoden `NullPointerException` och om `a` är tomt kastar den `ArrayIndexOutOfBoundsException`.
  2. `MinFinder2` är inte en äkta subtyp till `MinFindable1`.  
Metoden returnerar det *största* indexet för det minsta elementet, inte det minsta indexet som specifikationen kräver. Metoden kastar inte det undantag som specifikationen kräver. Istället returnerar den `-1` som kan resultera i ett indexfel hos klienten, vilket inte är acceptabelt.
  3. `MinFinder3` är en äkta subtyp till `MinFindable1`.  
Den strikta olikheten `<` i villkorssatsen garanterar att det minsta indexet för det minsta elementet returneras.
  4. `MinFinder4` är inte en äkta subtyp till `MinFindable1`.  
Samma brister som `MinFinder1` för `null` eller tomt fält och liksom `MinFinder2` returnerar den det största indexet.
  5. `6,7,8. MinFinder1, ..., MinFinder4` är äkta subtyper till `MinFindable2`.  
Metoderna ger i samtliga fyra fall rätt resultat om förvillkoret är uppfyllt eftersom de returnerar det minsta eller det största indexet för det minsta elementet och specifikationen tillåter vilket som helst av de minsta. Är förvillkoret inte uppfyllt är det ospecificerat vad metoden skall göra och klienten får då ta konsekvenserna.



**Uppgift 6** (4+3 p)

a)

```
public class SortedValueMap<K,V> implements ValueMap<K,V> {
    private TreeMap<K,Collection<V>> map;
    public SortedValueMap() {
        map = new TreeMap<>();
    }
    public void addValue(K key,V value) {
        if ( ! map.containsKey(key) )
            map.put(key,new ArrayList<V>());
        map.get(key).add(value);
    }
    public Collection<V> getValues(K key) {
        return map.get(key);
    }
    public Set<K> getKeys(V value) {
        Set<K> result = new HashSet<K>();
        for (Map.Entry<K,Collection<V>> e : map.entrySet() )
            if ( e.getValue().contains(value) )
                result.add(e.getKey());
        if ( result.isEmpty() )
            return null;
        else
            return result;
    }
}
```

b)

```
public class CVSReader {
    public final static String FIELD_SEPARATOR = ",";
    public static ValueMap<String,Integer> readCSVFile(String fileName)
    throws IOException,NumberFormatException
    {
        ValueMap<String,Integer> map = new SortedValueMap<>();
        Scanner sc = new Scanner(new FileReader(fileName));
        while ( sc.hasNextLine() )
            addValues(map,sc.nextLine());
        sc.close();
        return map;
    }

    private static void addValues(ValueMap<String,Integer> map,String line) {
        String[] fields = line.split(FIELD_SEPARATOR);
        for ( int i = 1; i < fields.length; i++ )
            map.addValue(fields[0],Integer.parseInt(fields[i]));
    }
}
```

**Uppgift 7** (2+4 p)

- a) Punktojektet i ett geometriskt objekt kan muteras via en sparad referens till objektet som gavs till konstruktorn, eller via en referens returnerad av en accessmetod:

```
Point p = new Point(10,20);
GeometricalObject g = new GeometricalObject(p);
p.setPosition(30,40);
Point q = g.getPosition();
System.out.println("q.x:"+q.getX()+" ,q.y:"+q.getY()); // q.x:30,q.y:40
q.setPosition(50,60);
Point r = g.getPosition();
System.out.println("r.x:"+r.getX()+" ,r.y:"+r.getY()); // r.x:50,r.y:60
```

- b)

```
public class Point implements Cloneable {
    ...
    public Point clone() {
        try {
            return (Point)super.clone();
        }
        catch (CloneNotSupportedException e) {
            throw new InternalError();
        }
    }
}

public class GeometricalObject {
    private Point position;
    public GeometricalObject(Point position) {
        this.position = position.clone();
    }
    public Point getPosition() {
        return position.clone();
    }
    // other methods omitted
}
```

### Uppgift 8 (10 p)

Designen bryter mot principerna *DIP* och *OCP*. Såväl klassen `Expression` som metoden `getValue` måste skrivas om vid tillägg av ytterligare operatörer. Designen bryter också mot principen *Information Expert*. Metoden `getValue` borde tillhöra klassen `Expression` och inte `Main`. Vi refaktorerar båda klasserna och tillämpar *Strategy*-mönstret. Problemet kan lösas på några olika sätt. Här eliminerar klasserna `BinaryExpression` och `UnaryExpression` onödig kodduplicering i `Addition` m.fl. genom att de lagrar operanderna och är parametriserade med avseende på operatören, som ges till konstruktorn och lagras i klassen. Jämför med hur t.ex. `TreeSet` kan ta ett jämförarobjekt av typen `Comparator` som parameter till konstruktorn (*Strategy*).

```
public class Constant implements Expression {
    private int value;
    public Constant(int value) {
        this.value = value;
    }
    public int getValue(Environment env) { // Skönhetsfläck
        return value;
    }
}
public class Variable implements Expression {
    private String name;
    public Variable(String name) {
        this.name = name;
    }
    public int getValue(Environment env) {
        return env.getValue(name);
    }
}
public class BinaryExpression implements Expression {
    private BinaryOperator operator;
    private Expression leftOperand, rightOperand;
    public BinaryExpression(BinaryOperator operator, Expression leftOperand,
        Expression rightOperand)
    {
        this.operator = operator;
        this.leftOperand = leftOperand;
        this.rightOperand = rightOperand;
    }
    @Override
    public int getValue(Environment env) {
        return
            operator.binOp(leftOperand.getValue(env),
                rightOperand.getValue(env));
    }
}
public class Addition extends BinaryExpression {
    public Addition(Expression leftOperand, Expression rightOperand) {
        super(new AddOp(), leftOperand, rightOperand);
    }
}
public class UnaryExpression implements Expression {
    private UnaryOperator operator;
    private Expression operand;
    public UnaryExpression(UnaryOperator operator, Expression operand) {
        this.operator = operator;
        this.operand = operand;
    }
    @Override
    public int getValue(Environment env) {
```

```
        return operator.unOp(operand.getValue(env));
    }
}
public class Negation extends UnaryExpression {
    public Negation(Expression operand) {
        super(new NegOp(), operand);
    }
}
public class AddOp implements BinaryOperator {
    @Override
    public int binOp(int leftOperand, int rightOperand) {
        return leftOperand + rightOperand;
    }
}
public class NegOp implements UnaryOperator {
    @Override
    public int unOp(int operand) {
        return -operand;
    }
}
```