Åtkomstmodifikatorer, instantiering, referenser, identitet och ekvivalens, samt klassvariabler

Inkapsling – tumregler

1. Man skall inte behöva veta detaljer i implementationen av en klass för att kunna använda den.
2. En klass bör kunna garantera att dess status (dvs dess värden på attributen) är konsistenta. För att detta skall vara möjligt så måste alla ändringar av status kontrolleras.

För att uppnå detta måste man förhindra okontrollerad access till attributen utifrån klassen själv.

Det anses vara en god stil att göra *alla* attribut privata och använda *selektorer* (“get-metoder”) för att avläsa dem och *mutatorer* (“set-metoder”) för att sätta dem men

- ingen regel utan undantag och
- man måste se till att mutatorerna sätter rimliga värden.

`public` och `private`

- En metod som är **`public`** får anropas från alla metoder i alla klasser.
- Ett attribut som är **`public`** får avläsas och ändras från alla metoder i alla andra klasser.
- En metod som är **`private`** får bara anropas av metoder i den egna klassen.
- Ett attribut som är **`private`** får bara avläsas och ändras från metoder i den egna klassen.

Obs: **`private`** *inte* skyddar mot access från andra objekt i samma klass.

Anm: Det finns två skyddnivåer till: **protected** och *package* (som inte har något nyckelord) – mer om dessa senare.

Skapande av objekt

Objekt skapas från klassen med operatorn **`new`**.

När objektet skapas sker följande:

1. Alla attribut får *defaultvärden* (0, 0.0, '\0', `false`, `null`)
2. Eventuella tilldelningar i deklarationen av attributen utförs i deklarationsordning
3. En konstruktor körs

Om det *inte* finns någon konstruktor i en klass tillhandahåller java en *parameterlös default-konstruktor*.

Om det finns en eller flera konstruktorer väljes den som matchar argumenten i antal och typ. I detta fall tillhandahåller *inte* systemet någon parameterlös konstruktor.

Exempel: klassen Point

```
// Punkt i planet
public class Point {
    private double x;
    private double y;

    public Point(double x) { // Konstruktor
        this.x = x;
    }

    public Point(double x, double y) { // Konstruktor
        this.x = x;
        this.y = y;
    }

    public double getx() { return x; } // Selektor

    public double gety() { return y; } // Selektor
}
```

Frågor om klassen `Point`

- Vad händer om vi skriver `Point p = new Point(1.0)`?
- Vad händer om vi skriver `Point p = new Point()`?
- Skriv en mutator `void moveTo(double x, double y)` som flyttar punkten till en ny position. Vad är konsekvensen av att göra detta tillägg?
- Vad skulle vi förlora på att göra attributen publika?

Exempel: klassen `Circle`

```
public class Circle {
   private Point center;
   private double radius;

   public Circle(Point center, double radius) {
      this.center = center;
      this.radius = radius;
   }

   public Circle(double radius) {
      center = new Point(0.0, 0.0);
      this.radius = radius;
   }

   public void scale(double sf) { radius *= sf; }

   public void moveTo(double x, double y) { center.moveTo(x,y); }

   // Samt diverse andra metoder ...
}
```

Synpunkter på klassen `Circle`

- Kontrollerar inte att det är vettiga värden på attributen `radius`
- Vad händer vid följande kod?

```
Point p = new Point(0.5, -3.5);
Circle c1 = new Circle(p, 1.0);
Circle c2 = new Circle(p, 2.0);
c1.moveTo(-1.0, -2.0);
```

Gör så att `Point` äger sitt eget data

Skriv om konstruktorn för `Circle`-klassen:

```
public Circle(Point center, double radius) {
   this.center = new Point(center.getx(), center.gety());
   this.radius = radius;
}
```

eller

```
public Circle(Point center, double radius) {
   this.center = center.copy();
   this.radius = radius;
}
```

med metoden `copy` i klassen `Point`;

```
public Point copy() { return new Point(x,y); }
```

Sammanfattning om referenser

- När ett objekt skapas med **`new`** returneras en *referens* (dvs ett *handtag* – logiskt sett motsvarande objektets *adress*) till det skapade objektet.
- En variabeldeklaration, t.ex.

  ```
  Die t;
  ```

  skapar inte en ny `Die`, utan bara en variabel `t` som kan hålla en *referens* till en `Die`
- Referenser kan tilldelas till referensvariabler (av rätt typ) och jämföras med == och `!=`
- Instansvariabler av referenstyp initieras till `null`
- **`this`** är en referens till det ”egna” objektet

Sammanfattning om referenser forts

- Flera referenser kan peka till samma objekt (aliasering). T.ex. blir resultatet av tilldelningen

  ```
  t1 = t2;
  ```

  alltid att `t1` och `t2` är alias för samma objekt (el. `null`).

- Observera att relationsoperatorerna == och != jämför om två referenser avser samma objekt, inte om objekten "ser likadana ut"

- Ett objekt som ingen refererar till är skräp som städas undan automatiskt av *skräpsamlaren* (GC)

  ```
  Die d = new Die(12);
  d = null;
  ```

  I Java är minneshantering = att komma ihåg att sätta variabler till null

Sammanfattning om referenser forts

- Variabler kan hålla referenser eller primitiva datatyper – aldrig hela objekt
- Objekt skickas som argument till metoder med “referenssemantik” (pass by reference)
- Om ett argumentobjekt förändras av en metod är det synligt för alla so har en referens till metoden (sidoeffekter)
- Man kan säga att objekten existerar globalt och är åtkomliga överallt där man har en referens till dem
- En metod kan returnera referenser som returvärde

Jämförelser av objekt – igen

- Igen: Relationsoperatorerna `==` och `!=` jämför bara med avseende på identitet
- *Alla* objekt har en metod `equals()` som jämför objekt för *strukturell* likhet
- Varje klass bör definiera en egen `equals()`-metod
- Referenser kan inte jämföras storleksmässigt men man kan naturligtvis definiera egna metoder för att jämföra objekt

Exempel på tänkbara jämförelsemetoder i `Circle`

```
public class Circle {

  private Point center;
  private double radius;

  // ... en massa metoder

  public boolean equals(Circle c) {
    return Math.abs(radius-c.radius) < 1.e-10;
  }


  public boolean equals2(Circle c) {
      return (radius==c.radius) && center.equals(c.center);
  }
}
```

Exempel på jämförelsemetoder forts

```
public int compareTo(Circle c) {
  if (radius == c.radius)
    return 0;
  else if (radius < c.radius)
    return -1;
  else
    return 1;
}
```

```
public boolean lessThan(Circle c) {
  return radius < c.radius;
}
```

Klassvariabler och klassmetoder

- Hittills har alla dataattribut varit *instans*-variabler dvs varje objekt har sin egen upplaga av dessa
- Man kan också ha *klass*-variabler som ligger i klass-objektet och därigenom är gemensamma för alla objekt i klassen (Anges i Java med `static`)
- En klassvariabel kan referas på samma sätt som vanliga attribut men vanligen genom *klassnamn.variabelnamn*
- Det går också att ha *klass*-metoder som alltså kan användas frikopplat från objekten (ex `Math.sin()` och `main()`)
- Ett objekts metoder har automatisk åtkomst till dess klass' metoder och attribut, men inte det omvända

```
public class Circle {
   private Point center;
   private double radius;
   private int id; // "cirkelnummer"
   private static int nCircles = 0;

   public Circle( Point center, double radius ) {
      this.center = center;
      this.radius = radius;
      id = ++nCircles;
   }

   public static void report() {
      System.out.println( "Antal skapade cirklar: " + nCircles );
   }

   public static void main(String [] args) {
      Circle c;
      for ( int i = 1; i<=10; i++ )
         c = new Circle( new Point( 10*i, 15*i ), 5*i );
      report();
   }
}
```

Övningar

1. Skriv en klass `Person` med en konstruktor som tar namn och personnummer som indata, och som håller reda på hur många personer som har instantierats sedan programmet startades. Det sistnämnda görs lämpligen med en *privat klassvariabel*. När skall den räknas upp? Hur kan man garantera att den alltid räknas upp? Skriv även en instansmetod (vanlig metod) **`int`** `getCount()` som returnerar klassvariabelns värde.
2. Skriv personklassen så att utomstående inte har direkt åtkomst till ett personobjekts namn och personnummer. Namn skall gå att byta, men inte personnummer.
3. Utöka personklassen ovan så att *klassen* person har en lista över samtliga personer som skapats i systemet. Modifiera `getCount()` till att returnera denna listas längd istället för att ha en räknare. Finns det några problem med denna typ av design? Vad får det för effekt på minneshantering?

Övningar (forts)

4. Utöka personklassen med en **`boolean`** `equals(Object)`-metod som returnerar **`true`** vid jämförelse av två personobjekt med samma personnummer, annars **`false`**.
5. Utöka personklassen så att det inte går att skapa två personer med samma personnummer. Med denna garanti i systemet blir implementationen av `equals`-metoden nu trivial. Vilken är den minsta möjliga implementationen av `equals` man behöver i personklassen och varför?