Mutability och State

Objekt-orienterad programmering och design

Alex Gerdes, 2018

Immutability

- Ett *icke muterbart* (immutable) objekt är ett objekt vars tillstånd inte kan förändras efter att det skapats.

Mutability

- Per default i Java arbetar vi med objekt som är *muterbara* (mutable), dvs kan förändras på ett eller annat sätt.

- Enklaste formen av förändring är att värdet på attribut förändras (via setters).
- Det finns dock många sätt som objekt kan förändras – låt se!

Live code

- `Scheduler`

Aliasing

- Om vi har två (eller fler) referenser till samma objekt kallas dessa för *alias*.
  - Formellt korrekt vore att säga att vi har två eller fler kopior av samma referens.
- Genom alias kan vi komma åt samma objekt, och dess metoder, från mer än ett ställe i koden. Om objektet är muterbart kan förändringar gjorda via ett alias överraska övriga som har referenser till samma objekt. Ex:

```
Point xy = new Point(1,2);

Polygon p = new Square(xy);

doSomethingWithPoint(xy);

xy.x = 3;
```

Vi skapar en Point och får en referens

Square kommer lagra referensen internt – ett alias – och skulle kunna uppdatera sin Point

Metoden får ett alias, och skulle kunna uppdatera via detta.

Vi har fortfarande kvar ursprungsreferensen och kan överraska övriga som fått den.

Live code

- `Scheduler (add event)`

Metoden `equals`

- I klassen `Object` har metoden `equals` följande utseende:

```
public boolean equals(Object obj) {
  return this == obj;
}
```

- Detta betyder att om vi har två objekt betraktas de som lika endast om de är *alias*
- Alla klasser behöver därför definiera vad som menas med att ett objekt är lika med ett annat objekt

Metoden `equals`

- `equals`-metoden används på många ställen i bl.a. Collection-klasserna. Här är ett typiskt exempel på användning av `equals`:

```
private Object[] elementData = ...;
...
public int indexOf(Object elem) {
  for (int i = 0; i < elementData.length; i++)
    if (elem.equals(elementData[i]))
      return i;
  }
  return -1;
}
```

- Alla klasser behöver definiera vad som menas med att ett objekt av klassen är lika med ett annat objekt.

Metoden `equals`

- Låt oss titta på en klass `Triangle`:

```
import java.awt.Point;

public class Triangle {
  private Point p1, p2, p3;

  public Triangle(Point p1, Point p2, Point p3) {
    this.p1 = p1;
    this.p2 = p2;
    this.p3 = p3;
  }
  …
}
```

Metoden `equals`

- Vid en första anblick kan det tyckas vara enkelt att avgöra likheten mellan två objekt av klassen `Triangle` – om samtliga tre hörnpunkter är lika i de båda trianglarna borde trianglarna vara lika. Detta leder således till att `equals`-metoden skulle få följande utseende:

```
//-- Felaktig --//
public boolean equals(Object otherObject) {
  if (!(otherObject instanceof Triangle))
    return false;

  Triangle other = (Triangle) otherObject;

  return p1.equals(other.p1) &&
         p2.equals(other.p2) &&
         p3.equals(other.p3);
}
```

Metoden `equals`

- Låt oss nu göra ett litet testprogram

```
Triangle t1 = new Triangle(new Point(0,0), new Point(1,1), new Point(1,0));
Triangle t2 = new Triangle(new Point(2,2), new Point(4,3), new Point(1,0));
Triangle t3 = new Triangle(new Point(0,0), new Point(1,1), new Point(1,0));
System.out.println(t1.equals(t2));
System.out.println(t1.equals(t3));
```

- Utskriften blir vad vi förväntar oss:

```
false
true
```

- d.v.s. att `t1` och `t2` är olika, samt att `t1` och `t3` är lika

Metoden `equals`

- Låt oss nu krångla till det hela genom att införa en subklass `ColoredTriangle` till klassen `Triangle`

```
import java.awt.Point;
import java.awt.Color;

public class ColoredTriangle extends Triangle {
  private Color color;

  public ColoredTriangle(Color color, Point p1, Point p2, Point p3) {
    super(p1, p2, p3);
    this.color = color;
  }
}
```

Metoden `equals`

- Låt oss nu göra ett nytt testprogram:

```
Triangle t1 = new Triangle(new Point(0,0), new Point(1,1), new Point(1,0));
Triangle t2 = new Triangle(new Point(2,2), new Point(4,3), new Point(1,0));
ColoredTriangle t3 = new ColoredTriangle(Color.GREEN,
                                 new Point(0,0), new Point(1,1), new Point(1,0));

System.out.println(t1.equals(t2));
System.out.println(t1.equals(t3));
```

- Utskriften blir

```
false
true
```

- d.v.s. att `t1` och `t3` är lika. Men är verkligen ett objekt av klassen `Triangle` lika med ett objekt av klassen `ColoredTriangle`?
- Vi måste i metoden `equals` ta hänsyn till vilka *typer* som objekten i jämförelsen har.

Metoden `equals`

- I klassen `Object` finns metoden `getClass()` som returnerar vilken runtime-klass ett objekt har. Denna metod kommer nu väl till pass

```
//-- Fortfarande felaktig --//
public boolean equals(Object otherObject) {
  if (otherObject.getClass() != this.getClass())
    return false;
  Triangle other = (Triangle) otherObject;
  return p1.equals(other.p1) && p2.equals(other.p2) && p3.equals(other.p3);
}
```

- Testkör vi nu med samma exempel som tidigare får vi med denna variant av `equals`-metoden utskriften

```
false
false
```

- Detta resultat är vad vi vill ha. Dock är `equals`-metoden fortfarande inte korrekt!

Metoden `equals`

- I *The Java Language Specification* anges att `equals`-metoden skall ha följande egenskaper:
  - Skall vara *reflexiv*:
    - För varje icke-`null` referens `x` skall det gälla att `x.equals(x)` returnerar `true`
  - Skall vara *symmetrisk*:
    - För alla referenser `x` och `y` skall det gälla att `x.equals(y)` returnerar `true` om och endast om `y.equals(x)` returnerar `true`
  - Skall vara *transitiv*:
    - För alla referenser `x`, `y` och `z` skall gälla att om `x.equals(y)` returnerar `true` och `y.equals(z)` returnerar `true` så skall också `x.equals(z)` returnera `true`
  - Skall vara *konsistent*:
    - Om objekten till vilka `x` och `y` refererar inte har förändrats skall upprepade anrop av `x.equals(y)` returnera samma värde
  - För alla icke-`null` referenser `x` skall gälla att `x.equals(null)` skall returnera `false`

Metoden `equals`

- I `equals`-metoden för klassen `Triangle` måste vi ta hand om fallet då objektet som jämförelsen utförs mot är `null`

```
public boolean equals(Object otherObject) {
  if (this == otherObject)
    return true;
  if (otherObject == null)
    return false;
  if (otherObject.getClass() != this.getClass())
    return false;
  Triangle other = (Triangle) otherObject;
  return p1.equals(other.p1) && p2.equals(other.p2) && p3.equals(other.p3);
}
```

Hashcode

- En *hash code* är ett (litet) enskilt värde som beräknas från en (stor) struktur, för att ge ett effektivt sätt att identifiera strukturen.
- Används i många effektiva datastrukturer som nycklar.
- I Java ärver alla objekt metoden `public int hashCode()` från `Object`. I `Object` implementeras denna (i vissa VM) till att returnera ett värde baserat på referensens värde(!).
- Muterbara objekt *bör* göra override på `hashCode()`, (och `equals()`) så att olika hash codes returneras för olika tillstånd.
- Om `x.equals(y)`, så är `x.hashCode() == y.hashCode()`

Live code

- `(explain hashmap)`
- Date (`equals`, `hashCode`)

Quiz

- Vad innebär immutabilitet?

- Svar: Ett *icke muterbart* (immutable) objekt är ett objekt vars tillstånd inte kan förändras efter att det skapats.

Många former av förändring

- Objekt kan vara muterbara för att:
  - Attribut förändras (via setters)
  - För attribut som håller referenser som pekar på andra muterbara objekt, så innebär en förändring av dessa också implicit en förändring av objektet som håller referensen.
    - E.g. om objekt A har ett attribut som är en array, och objekt B har tillgång till en referens till samma array, så kan B uppdatera innehållet i arrayen och därigenom förändra tillståndet för A.

Defensive copying

- För att garantera att inga alias finns till eventuella muterbara objekt vi har i attribut behöver vi skapa så kallade *defensive copies*:
  - Om vi exponerar dessa objekt via getters.
  - Om vi i instansmetoder skickar dessa objekt som argument till externa metoder.
  - Om vi tar in nya värden för attributen, via constructor eller via setters.

- **OBS!** Notera att detta inte bara gäller när vi strävar efter immutability!
  - Även om det objekt som har attributen inte är tänkt att vara immutable, så vill vi fortfarande skydda oss mot *oväntade* förändringar via alias.

Live code

- SchedulerEvent

Mutate-by-copy

- Istället för att ändra objektets tillstånd, skapa en kopia med det nya tillståndet och returnera denna istället.

- Detta ursprungliga objektet ändras inte, och andra som håller referenser till det kan inte bli överraskade av ”alias updates”.

- Kan å andra sidan leda till att många nya objekt skapas.
  - Kan bli ineffektivt.

Live code

- Date

Quiz

- När bör man göra classes immutable?

- Svar: Så ofta – och i så hög grad – som möjligt!

> Classes should be immutable unless there's a very good reason to make them mutable. If a class cannot be made immutable, limit its mutability as much as possible. (Joshua Bloch)

Att mutera eller icke mutera

- Tumregel: Immutable per default – men låt objekt vars syfte är att förändras ofta vara mutable.

- Även om valet faller på mutable:
  - Se till (via e.g. defensive copying) att inga alias-problem uppstår.
  - Bero internt på immutable objects så mycket som möjligt.

Quiz

- Hur bör vi tänka kring arv av immutable classes?

- Svar: Per default är svaret ”nej”.
    - Vi har inga garantier för att en subclass också är immutable, vilket kan sägas bryta mot Liskov Substitution Principle.

Immutable classes

- En class är helt immutable om:
  - Alla attribut är `final`
    - Även privata attribut – krävs för trådsäkerhet, mer senare.
  - Klassen i sig är `final`
    - Dvs kan inte ärvas, så ingen risk för muterbara subclasses
  - Alla attribut som har referenser till muterbara objekt:
    - … är `private`.
    - … returneras aldrig från metoder, vare sig direkt eller indirekt.
    - … håller den garanterat enda referensen till objektet i fråga (defensive copying).
    - … ändrar inte tillstånd på objekten de refererar till efter initialisering (doh).

  - Referensen `this` ”does not escape constructor”
    - Krävs för trådsäkerhet – mer senare, och i senare kurs.

Värdesemantik

- Immutable objects har i grunden *värde-semantik* – dvs de uppför sig som ren data, e.g. `int` eller `String` (som är immutable i Java).

- Vi behöver inte använda defensive copying för attribut som håller primitiva värden – och inte heller för attribut som refererar till immutable objects.
  - Ingen annan kan göra något dumt med dem oavsett!

Vinster med immutability

- Inga oväntade alias-uppdateringar – det är säkert att bero på tillståndet hos immutable objects.
- Inget behov av defensive copying av objekt som är immutable.
- Det är mycket lättare att resonera och bevisa egenskaper för kod som arbetar med immutable objects.
- Immutable objects (done right) är alltid automatiskt *trådsäkra*, och lämpar sig bra för *parallella beräkningar*.
- Ingen risk att objekt kan hamna i *inkonsekventa tillstånd* (inconsistent states).
- Högre maintainability – alias-buggar är bland de allra jobbigaste att spåra och lösa.

Nackdelar med immutability

- Kan sägas bryta mot OCP – om vi senare inser att vi behöver kunna mutera ett objekt krävs förändringar av koden.
  - Å andra sidan handlar OCP mycket om just att förutsäga förändringar (av kod). Vi bör alltid tänka efter om ett objekt lämpar sig för att vara immutable.

- Kan, naivt använt, leda till (extrem) ineffektivitet.
  - Att modellera något som ska förändras mycket och ofta (t ex animeras) med hjälp av immutable objects och mutate-by-copy leder till att väldigt många nya objekt skapas, vilket äter både minne och processortid.
  - Å andra sidan kan det, rätt använt, förhindra (extrem) ineffektivitet.

Quiz: Immutable = pure?

- Antag att vi har en class `C` som uppfyller alla krav för att vara immutable, och följande kodsegment:

```
C c = new C();
int x = c.getSomeValue();
```

- Kan vi vara säkra på att vi alltid får samma värde i `x`?

- Svar: Nej! Våra krav för immutability säger ingenting om att metoder inte kan bero på yttre, globalt tillgängliga, variabler.

Globalt tillgängligt

- Alla klasser och object har tillgång till en mängd saker per default.
  - De använda allt som är `public` och `static` från alla paket och klasser som finns i scope.
  - De kan skapa nya objekt genom att anropa publika konstruktorer (eller publika statiska Factory Methods).

- Vi har inga garantier för att alla dessa är icke-muterbara, eller alltid returnerar samma saker när vi använder dem.

Singleton Pattern

Restrict instantiation of a class to a single object. Whenever an instance of the class is requested, return that one object.

- Begränsa skapandet av instanser av en klass, så att det bara finns en enda instans, och det är denna instans som returneras närhelst en sådan efterfrågas.

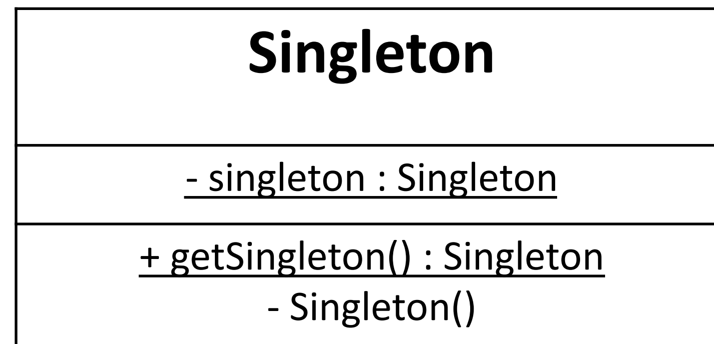
Singleton pattern

- Namnet *Singleton Pattern* kommer sig ur mängdlära, där en *singleton* är en mängd med enbart ett element i sig.

- Singleton Pattern kan användas för att representera:
  - Det kontext i vilket en komponent eller applikation körs.
    - E.g. config-filer, system-specifika detaljer, etc
  - Specifika resurser som applikationen använder:
    - E.g. Database drivers, mobilens kamera, etc.

- Används på tok för ofta, vilket har gett detta pattern (delvis oförtjänt) dåligt rykte.

Singleton Pattern



Göm
konstruktorn!

Live code

- Logger

Global state

- Ett alltför vanligt, men väldigt dåligt, användningsområde för Singleton Pattern är att använda det för att definiera objekt som håller *globala tillståndsvariabler* (global state) – alltså muterbara variabler som kan ändras och läsas från var som helst i koden.
  - Jätte-jätte-jättedåligt!
  - Don’t do it!
  - Really don’t!
- Kod som använder global state skapar beroenden som är extremt svåra att överblicka och underhålla.
  - Om du verkligen behöver state, skicka då det objekt som representerar detta state explicit till de klasser som behöver bero på och uppdatera det. Detta ger explicita beroenden, som fortfarande är starka, men ger bättre överblick.

Summering

- Mutability är bra där det behövs – men kan ställa till problem.
- Begränsa mutability så mycket det bara går – men inte mer.
- Använd inte muterbar global state!
- Defensive copying är bra att använda som regel – tänk alltid tanken!
- Implementera `equals` och `hashCode` metoder

What’s next

Block 6-2:
Fler design patterns