Immutability

Objekt-orienterad programmering och design

Alex Gerdes och Sólrún Halla Einarsdóttir, 2018

Mutability

- Per default i Java arbetar vi med objekt som är *muterbara* (mutable), dvs kan förändras på ett eller annat sätt.
  - Medlemsvariabler (attribut) kan 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.
  - (Obs: Tillstånd = ”state”, inte tillstånd = ”permission”)

Immutability

- Att ha *icke muterbara* (immutable) objekt har flera fördelar:
  - Ingen alias-problematik: dvs om vi har två referenser till samma objekt, så kan förändringar via den ena referensen överraska den som försöker läsa från den andra.
  - Det är mycket lättare att resonera och bevisa saker för immutable objects.
  - Immutable objects är alltid *trådsäkra*, och lämpar sig bättre för *parallella beräkningar* (mer om detta senare).
  - Beroenden på immutable objects är säkrare, ”ingenting kan gå fel”.

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

Övning

- Börja från koden som finns att ladda ner från hemsidan.
  - Vår gamla kod från förra veckan är uppdaterad enligt vad vi diskuterade om på föreläsningen: Vi har nu separata Model, View och Controller, som var och en gör det som förväntas av dem. Nästan.
- Uppföljning från gamla ämnet: Animeringen fungerar inte. Vad är fel? Rätta felet.
  - Hint: Det saknas en enda rad kod.
- För resten av övningen, fokusera på package `tda551.shapes`. Vi struntar för den här övningen i hur det sen ska användas – du har nu hatten av maintainer för detta paket, och vet ingenting om `DrawPolygons`.
  - Klassen `Shape` har ett väldefinierat gränssnitt. Denna klass är definitivt muterbar (än så länge). Men på grund av att `Point` också är muterbar, kan en `Shape` flyttas indirekt på annat sätt än som är tänkt via gränssnittet. Förstå hur detta ”trick” går till. Fundera över olika sätt vi skulle kunna lösa det.
  - Ett (eller egentligen två) sätt är att definiera en egen, immutable, `Point`-klass. Detta kan antingen göras helt från scratch, eller genom att skapa en immutable wrapper runt den existerande `Point`-klassen. Testa båda dessa lösningar.
  - Lös problemet även utan att definiera en egen `Point`-klass, och utan att förändra gränssnittet för `Shape`. Jo, det går!
  - (Lite svårare) Även `Shape` i helhet kan göras immutable, och ändå tillåta translation, skalning och rotation. Sättet vi gör det på är att vi, istället för att uppdatera det egna objektet, returnera en ny kopia med de nya värdena. Retur-typerna för de tre ”muterande” metoderna blir alltså `Shape`, istället för `void` som nu.