Subtyping, co- och contra-variance

Objekt-orienterad programmering och design

Alex Gerdes, 2018

Typer

- Java har två sorters typer – *primitiva typer* och *referens-typer*.
  - Primitiva typer är typer för *värden*: int, float etc.

- Java har två sorters referens-typer – *array types* samt *class or interface types*.
  - Arrayer är ett specialfall. De bor på heapen, vi arbetar med dem via referenser, de är i praktisk mening objekt – men typerna ser lite speciella ut.

- Vi pratar här om *statiska* typer – de som används av typ-checkaren under kompilering.

Subtyper

- En typ A (e.g. `Triangle`) är en subtyp till en annan typ B (e.g. `Polygon`), om alla element (värden eller objekt) av typ A (`Triangle`) kan användas som om de vore av typ B (`Polygon`).
- En klasstyp C är en subtyp till:
  - `Object`
  - Sin superclass (`extends`) och alla dess supertyper.
  - Alla de interfaces C implementerar (`implements`).
- En interfacetyp I är en subtyp till:
  - `Object`
  - Sitt superinterface (extends) och alla dess supertyper.
- Vi skriver A **<:** B för att ange att A är en subtyp till B.

Subtyper och polymorfism

- Eftersom `Triangle` **<:** `Polygon`, kan en referens till ett objekt av typ `Triangle`:
  - Tilldelas en variabel med deklarerad typ `Polygon`.
  - Ges som argument till en metod med parametertyp `Polygon`.
    - Metoden förväntar sig en `Polygon`, vilket en `Triangle` är.
    - Metodens body kan bara bero på gränssnittet för `Polygon`.
  - Returneras från en metod med deklarerad retur-typ `Polygon`.
    - Den kod som anropar metoden förväntar sig en `Polygon`, vilket en `Triangle` är.

Parameteriserade typer

- En *parameteriserad typ* förväntar sig en annan typ som argument för att bli ”hel”.
  - Utan sitt argument har vi en *typkonstruktor* (type constructor), inte en typ.
  - Vi kan bara skapa objekt av faktiska typer, inte typkonstruktorer.

- I Java finns:
  - Arrays: `[]` kan ses som en typkonstruktor för array-typer. Vi har inte en faktisk typ förrän vi anger typen för elementen, e.g. `String[]`.
  - Generic types: E.g. `ArrayList<_>` är en typkonstruktor. Vi måste ange en argument-typ för att vi ska ha en faktisk typ, som vi kan skapa objekt av, e.g. `ArrayList<String>`.
    - (Den icke-parameteriserade versionen `ArrayList` en bastard, en faktisk typ med implicit argument-typ `Object`.)

Subtypning för parameteriserade typer

- Vi vet att subtypning fungerar ”normalt” vad gäller *typkonstruktorn*:
  - `ArrayList<String>` **<:** `List<String>` **<:** `Object`

- Hur fungerar det för *typargumenten*?
  - `List<Triangle>` **<:** `List<Polygon>` ?

  - Svar: Nej. Varför inte?

Live code

- …

Quiz

Arrayer är ett specialfall:

```
Triangle[] <: Polygon[]
```

Vad kan gå fel?

Live code

- …

Quiz

Arrayer är ett specialfall:

```
Triangle[] <: Polygon[]
```

Vad kan gå fel?

```
Triangle[] triarray = new Triangle[1];
Polygon[] polyarray = triarray;

polyarray[0] = new Square(5,10);
Triangle t = triarray[0];
```

Fångas vid runtime:
`ArrayStoreException`

Variance för parameteriserade typer

- En typkonstruktors *varians* (variance) avgör hur subtypning hanteras.
- Antag att A <: B. Om vi då har:
  - T<A> <: T<B>, då säger vi att T är *covariant* i sin parameter.
    - Från ”co-” eller ”com” = med, subtypningen varierar med – åt samma håll som – den för parametern.
  - T<B> <: T<A>, då säger vi att T är *contravariant* i sin parameter.
    - Från ”contra” = mot, subtypningen varierar emot – åt motsatt håll från – den för parametern.
  - Inget av ovanstående, då säger vi att T är *invariant* i sin parameter.
    - Från ”in-” = inte, vi har ingen varierande subtypning alls.
- Vid flera parametrar kan parametrarna ha olika varians.
  - E.g. T<A,C> <: T<B,D> iff A <: B (covariant) och D <: C (contravariant).

Generiska typer är invarianta

- Subtypning för argumentet ger ingen subtypning för hela typen:
  - E.g. `List<Triangle>` <: `List<Polygon>` gäller inte!

- Quiz: Varför?

Arrays är covariant

- Subtypningen varierar *med* den för argumentet:
  - E.g. eftersom `Triangle` <: `Polygon`, så är `Triangle[]` <: `Polygon[]`

- Quiz: Varför, om saker kan gå fel?
  - Svar: För att Java innan Generics behövde den polymorfism det ger.

Live code

- …

Polymorfism för metoder

- Med generics kan vi ge typ-parametrar inte bara till klasser utan även till metoder:

```
public static <T> T head(T[] a){
   return a[0];
}
String[] strings = …
String x = MyClass.<String>head(strings);
```

Kan utelämnas nästan jämt, typcheckaren kommer hitta rätt typ utifrån användandet (*typinferens*).

Live code

- …

Covariance och collections

- Vid *covariance* skulle vi få problem med att *lägga till* saker i en lista:
  - Om `List<Triangle>` <: `List<Polygon>`, då skulle vi tillåta följande:

```
List<Triangle> trilist = new ArrayList<Triangle>();
List<Polygon> polylist = trilist;

polylist.add(new Square(5,10));
```

  - Att *läsa* från en lista är däremot inget problem.

```
List<Triangle> trilist = new ArrayList<Triangle>();
List<Polygon> polylist = trilist;

Polygon p = polylist.get(0);
```

Covariance och collections

- Vid *contravariance* skulle vi få problem med att *läsa* saker från en lista:
  - Om `List<Polygon>` <: `List<Triangle>`, då skulle vi tillåta följande:

```
List<Polygon> polylist = new ArrayList<Polygon>();
List<Triangle> trilist = polylist;

Triangle t = trilist.get(0);
```

  - Att *skriva till* en lista är däremot inget problem.

```
List<Polygon> polylist = new ArrayList<Polygon>();
List<Triangle> trilist = polylist;

polylist.add(new Square(5,10));
```

Generics och explicit varians

- För generiska typer kan vi explicit ange hur vi vill att subtyping ska hanteras:

Invariant

Covariant

Contravariant

```
List<Polygon> inv = new ArrayList<Polygon>();

List<? extends Polygon> cov = new ArrayList<Triangle>();

List<? super Triangle> contv = new ArrayList<Polygon>();
```

- `?` kallas för *wildcard.*

Existential quantification

- Notera: `List<? extends Polygon>` betyder **inte** att vi har en lista med objekt av *olika* typer, som *var och en* är av en subtyp till `Polygon`. (Det är faktiskt vad vanliga `List<Polygon>` innebär.)
- `List<? extends Polygon>` innebär att referensen pekar på ett objekt av typ `List<X>`, där allt vi vet om `X` är att X <: `Polygon`.

- ∃(X <: `Polygon`) sådant att referensen pekar på ett objekt av typen `List<X>`.

Explicit covariance

```
List<? extends Polygon> cov = …
```

- Vi kan läsa element ur listan: vad `?` än representerar, så vet vi att objekten i listan går att betrakta som `Polygon`.
- Vi kan inte skriva till listan. Antag att vi vill lägga till en `Triangle` – hur vet vi att det inte faktiskt är en lista av `Square`? Eller vice versa.

Explicit contravariance

```
List<? super Polygon> cov = …
```

- Vi kan lägga till element av typen `Polygon` (eller dess subtyper) till listan. Vad `?` än representerar vet vi att det är en supertyp till `Polygon`, vilket innebär att listan kan ta emot objekt av typ `Polygon`.
- Vi kan faktiskt läsa från listan – men bara som typen `Object` (eftersom ingenting kan existera som inte är en subtyp till `Object`).

Metoder och varians

- Samma princip gäller för metoder, även om vi inte behandlar dem som funktionstyper:
  - En metod som har deklarerad retur-typ `Polygon` får returnera en `Square`.

- En metod som gör `@Override` på en metod som i superklassen är deklarerad att returnera en `Polygon`, kan i en subklass deklareras till att returnera e.g. en `Square`.