Subtyping och variance

Objekt-orienterad programmering och design

Niklas Broberg, 2023

Typer

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

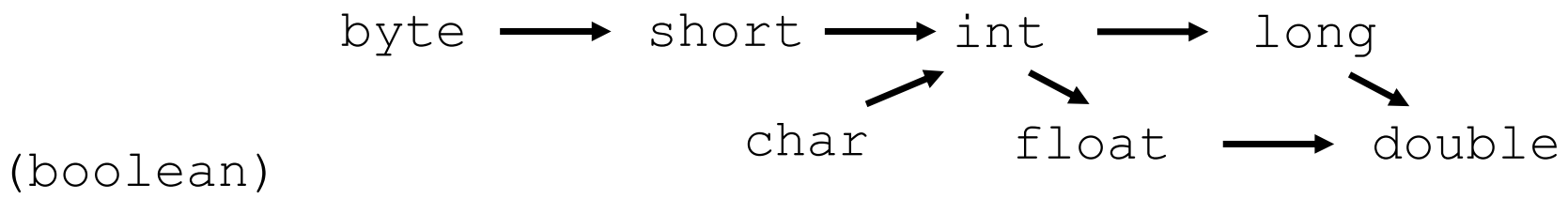
- Java har två sorters referens-typer – *array types* samt *class or interface types*.
  - Arrayer är ett specialfall. De bor på heapen, vi arbetar med dem via referenser, de är i praktisk mening objekt – men typerna ser lite speciella ut.

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

Subtyper

- En typ A är en subtyp till en annan typ B, om alla element (värden eller objekt) av typ A kan användas som om de vore av typ B.
- En klasstyp C är en subtyp till
  - `Object`
  - Sin superclass (`extends`) och alla dess supertyper.
  - Alla de interfaces C implementerar (`implements`).

- För primitiver finns en hierarki:



Type conversion

- Typer kan på olika sätt *konverteras*:
  - (Värden med) primitiva typer kan konverteras till (värden av) andra primitiva typer.
  - (Värden med) primitiva typer kan konverteras till och från (objekt av) deras representativa klass-typer, e.g. `int` till `Integer`.
  - En referenstyp A kan konverteras till en supertyp B, e.g. `Triangle` till `Polygon`.
  - En referenstyp B kan *castas* till en subtyp A:
    - `Triangle t = (Triangle) polygons.get(0);`
    - Notera att detta enbart påverkar hur typen behandlas *statiskt*. Vi ber typcheckaren lita på oss – och skulle objektet vi arbetar med vid runtime ha fel typ får vi ett exception.
    - (Casts är nästan alltid ett tecken på feltänkt design.)

Conversions

- En konvertering till en ”bredare” typ kallas *widening conversion* och kommer alltid att fungera felfritt.
- En konvertering till en ”snävare” typ kallas *narrowing conversion*.
  - Med primitiver kommer viss loss of precision att ske, e.g. `long` till `int`.
  - Med referenstyper får vi fel vid runtime om objektet inte redan hade rätt *dynamisk* typ.

- En konvertering från e.g. int till Integer kallas *autoboxing*. En konvertering från e.g. Integer till int kallas *unboxing*.

Subtypning för parameteriserade typer

- Java har två sorters polymorfism: subtypning, samt parametrisk polymorfism.
  - Vi vet ifrån tidigare block hur dessa funkar, var och en för sig. Men hur hänger de ihop?
  - Specifikt, hur fungerar subtypning för parameteriserade typer?

  - För *typkonstruktorn* fungerar subtypning precis som vanligt.
    - E.g. `ArrayList<T>` <: `List<T>`

  - Den lurigare frågan är, hur fungerar det för typargumenten?
    - Vad gäller för subtypsförhållande mellan e.g. `ArrayList<Polygon>` och `ArrayList<Triangle>`, där `Triangle` <: `Polygon`?

Parametriserade typer: Array vs ArrayList

- För parameteriserade typer gäller generellt:
  - En generisk typ `T<A>` är *inte* en subtyp till typ `T<B>`, även om `A` är en subtyp till `B` – generiska typer är ***invarianta*** (invariant).
  - Typen för elementen kommer *inte* att minnas vid runtime – det behövs inte då den statiska typcheckaren är mer strikt.
    - Nästan sant: alla parameteriserade typer är (tyvärr) subtyper till sin helt oparameteriserade version, e.g. `ArrayList<T>` är en subtyp till `ArrayList` – dåligt, beror enbart på att Java innan version 5 inte hade parameteriserade typer, och de ville vara bakåtkompatibla. Använd ALDRIG de oparameteriserade varianterna, det är att betrakta som en bugg. IntelliJ ger oss alltid en varning, e.g. ”Raw use of parameterized class ’ArrayList’”.

- För arrays (specialfall) gäller:
  - Alla arraytyper är subtyper till `Object`.
  - En arraytyp `A[]` är en subtyp till typ `B[]` om `A` är en subtyp till `B` – arrayer är ***covarianta*** (covariant).
  - Java minns typen för elementen vid runtime, och vi får exceptions om vi försöker göra dumt.

Parametriserade typer och explicit varians

- Vi kan dock ge variabler statiska typer som explicit är co- eller contra-varianta, med hjälp av **wildcards** samt **upper** och **lower** bounds.

Invariant

Covariant

Contravariant

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

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

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

- Mer om varians (variance) och dess teori på föreläsningen på fredag.

Övning - preamble

- Utgå från DrawPolygons sen tidigare, länk till github finns som vanligt på Canvas.

- Ni hittar också en ny klass `TestSubtyping`, med en metod `testSubtyping`. Gå igenom denna metod steg för steg, och lös uppgifterna.

- Steg 0: Titta på de variabler som deklareras i början av `testSubtyping`, och notera deras typer. Vi kommer använda dessa i diverse tester genom resten av övningen.

- Steg 1: Testa vad som går att göra med respektive variabel.
  a) Lägga till: Vilken typ av objekt får ni lägga in i respektive array eller lista?
     - Spelar den dynamiska typen roll?
  b) Plocka ut: Vilken typ av objekt kan ni plocka ut ur respektive array eller lista?
  c) Vilka av era variabler kan ni tilldela till varandra? Vad säger detta om subtypningen?
- Steg 2: Titta på metoden `paintAll(Graphics,List<Polygon>):void` som går igenom listan den får som argument och anropar `paint`-metoden på varje element.
  a) Vilka av listorna kan ni ge som argument till `paintAll`?
  b) Kan ni bredda parametertypen för `paintAll` så att den kan ta emot fler argument?
- Steg 3: Titta på metoden `addAll(List<Polygon,List<Polygon>):void` som går igenom sitt första argument och lägger till varje element i det andra argumentet.
  a) Vilka av listorna kan ni ge som argument till `addAll`?
  b) Kan ni bredda båda parametertyperna så att metoden kan ta emot fler argument?
- Steg 4: Förklara varför addAll, containsAll, och removeIf – alla från Java's standardbibliotek – har de parametertyper de har.

För mer utmaning

- Steg 5: Arrays vs ArrayList, varför har de olika beteende, och vilka konsekvenser får det?

- Steg 6: Förra övningen skapade vi ett interface `Function` med metoderna `compose` och `apply`; detta interface finns nu i repot ni har laddat ner.
  - Parametertypen för compose är i nuläget inte så generell som möjligt, vilket begränsar vår möjlighet att använda den. Hur kan ni göra den så bred som möjligt?