Generics och polymorfism

Objekt-orienterad programmering och design

Alex Gerdes, 2017

Live code

- Pair
- Lists

Pair

```
public class Pair<A,B> {
  private A a;
  private B b;

  public A getFirst() { return a; }
  public B getSecond() { return b; }
}
```

Typ-parametrar och typ-argument

- `A` och `B` kallas *typ-parametrar*, och `Pair` är en *parameteriserad typ*.

```
public class Pair<A,B> {
  private A a;
  private B b;

  public A getFirst() { return a; }
  public B getSecond() { return b; }
}
```

- I en konkret användning, e.g. `Pair<String,Integer>`, är `String` och `Integer` *typ-argument*.
- Inuti class-bodyn kan A och B användas som om de vore konkreta typer, eftersom de vid varje instansiering kommer att vara det (e.g. `String` och `Integer` ovan).

Metod-parametrar och argument

- Jämför med parametrar och argument till en metod:

```
public void setSize(int x){
   this.width = x;
}

myRectangle.setSize(42);
```

- `x` är en *parameter* till metoden. I ett konkret anrop är e.g. `42` ett argument.
- I metod-bodyn kan vi använda `x` som ett konkret expression, eftersom `x` vid varje anrop kommer att ges ett konkret argument (e.g. `42` ovan).

Typer för metod-parametrar

- När vi anger metod-parametrar deklarerar vi deras typ:

```
public void setSize(int x){
  this.width = x;
}
```

- Detta gör att vi kan använda dem i vetskapen om att de har de egenskaper och metoder som vi kan förvänta oss av objekt eller värden av den angivna typen.

Typ-parametrar – utan ”typ”

```
public class Pair<A,B> {
  private A a;
  private B b;

  public A getFirst() { return a; }
  public B getSecond() { return b; }
}
```

- Quiz: Vilka metoder skulle vi kunna anropa på `a` eller `b`?
  - Svar: Enbart de som tillhör `Object`, eftersom `Object` är en supertyp till allt.

Typ-parametrar med ”typer”

- Vi kan ange så kallade *upper bounds* på typ-parametrarna:

```
public class HomogeneousPolygonGroup<P extends Polygon> {
  private List<P> polygons;

  public void paint(Graphics g) {
    for (P p: polygons) { p.paint(g); }
  }
}
```

- Nu kan vi göra fler antaganden om objekten, eftersom vi vet mer om deras typ.
- Att försöka skapa e.g. en `HomogeneousPolygonGroup<Integer>` ger ett statiskt fel.

Upper Bounds

- Ett *upper bound* på en typ-parameter ger oss en möjlighet att kontrollera vilka sorts typer vi kan ge som argument.
- Vi kan ange flera upper bounds med `&` mellan:

```
public class NumHelper<T extends Number & Comparable<? super T>>{
  public static int compareNumbers(T t1, T t2){
    return t1.compareTo(t2);
  }
}
```

- Om ett av våra bounds är en klass-typ måste denna anges först.
  - I have no idea why…
- Vi kan inte ange *lower bounds* (`super`) som vi kan med wildcards (`?`)

Aside: wildcards

- Notera skillnaden mellan en typ-parameter och ett wildcard:

```
public class NumHelper<T extends Number & Comparable<? super T>{
  public static int compareNumbers(T t1, T t2){
    return t1.compareTo(t2);
  }
}
```

- En typ-parameter (`T`) är en variabel som representerar en typ. Vi vet inte vilken typ detta är – det bestäms av det argument som ges vid instansiering, och kan vara olika typer olika gånger vi instansierar.
- Ett wildcard (`?`) är ett *typ-argument* som representerar en *okänd typ*. Vi kommer aldrig veta mer om denna typ än de bounds (upper och lower) som är givna.

Typ-parametrar 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

- Either
- Function

OPC: The Open-Closed Principle

> *Software modules should be open for extension, but closed for modification.*

- Vårt mål är reusability, extensibility, maintainability.
- OPC är kärnan i vår metodik för att uppnå detta.

- Två av våra mest värdefulla verktyg för att uppnå OPC är *polymorfism* och *code reuse*.

Polymorfism

- Generellt brukar man definiera tre olika ”sorters” polymorfism:
  - **Parametric polymorphism**: En typ (eller metod) kan vara parameteriserad över en annan typ. Den parameteriserade typen (metoden) definierar en struktur (algoritm) som är *oberoende* av den typ den tar som argument.
  - **Subtype polymorphism**: Ett objekt av en subtyp kan agera som om det vore ett objekt av en supertyp (superclass eller interface).
  - **Ad hoc polymorphism**: Olika typer kan oberoende av varandra ge definitioner för samma namn/symbol.

Parametric polymorphism

- En typ (eller metod) kan vara parameteriserad över en annan typ. Den parameteriserade typen (metoden) definierar en struktur (algoritm) som är *oberoende* av den typ den tar som argument.
  - Historiskt brukar detta inom den funktionella sfären (e.g. Haskell) helt enkelt kallas polymorfism.
  - Java Generics ger parametric polymorphism till Java (sen Java 5, 2004).
- Ger oss möjlighet att återanvända samma struktur för många olika typer:

```
Pair<String,Integer> p1 = …
Pair<? extends Number, Polygon> p2 = …
```

- Ger automatisk code reuse genom att vi återanvänder koden för strukturen.

Subtype polymorphism

- Ett objekt av en subtyp kan agera som om det vore ett objekt av en supertyp (superclass eller interface).
  - Historiskt brukar detta inom den objekt-orienterade sfären helt enkelt kallas polymorfism, eftersom det länge varit den enda och förhärskande formen.
- Ger oss möjlighet att betrakta objekt av olika typer som om de hade samma typ, genom att enbart titta på ett gemensamt publikt gränssnitt:

```
for (Polygon p : polygons) { p.paint(g); }
```

- Ger code reuse via direkt arv (subclassing), eller genom implementation av interface via delegering.

Ad hoc polymorphism

- Olika typer kan oberoende av varandra ge definitioner för samma namn/symbol.
  - Exempel: Operator overloading i C#, type classes i Haskell.
- Betrakta följande funktion (pseudokod):

```
add(x,y) = x + y;
```

- Vilka argument kan ges till `add`? Om språket har ad hoc polymorphism för operatorn +, kan helt enkelt värden av alla typer som ger en definition av + ges som argument:

```
<T implements +> T add(T x, T y) = x + y;
```

OBS! Inte korrekt Java-kod

- Ger *ingen* code reuse – hela poängen är att operatorn eller metoden definieras separat för varje typ.
- Java har inte ad hoc polymorphism.

Subtype vs Ad hoc polymorphism

- Subtype polymorphism och Ad hoc polymorphism har klara likheter.
  - Betänk följande interface i Java:

```
public interface Adder {
  public add(Adder x);
}
```

  - Alla typer som implementerar interfacet ovan gör det separat. Kod som använder metoden `add` kan användas för alla dessa typer.
  - Det som gör att vi betraktar detta som subtype polymorphism, och inte ad hoc, är att vi samtidigt definierar en *typ* `Adder`. Skillnaden är närmast filosofisk, men har konkret påverkan på olika språk-features:
    - E.g. vi kan inte skapa listor av ”objekt av typer som implementerar `+`” i C#, eller listor av ”värden av typer som implementerar `Eq`” i Haskell.

Subtype vs parametric polymorphism

- Parametric polymorphism och subtype (och ad hoc) polymorphism har stora olikheter:
  - Subtype polymorphism handlar om att konkreta objekt uppträder på samma sätt (cf. Liskov Substitution Principle), och därför kan användas i varandras ställe. Code reuse åstadkoms genom att subklasser ärver från superklasser.
  - Parametric polymorphism handlar om att definiera kod som är *oberoende* av en underliggande typ. Denna kod kan då återanvändas i alla konkreta instansieringar av typen. Olika instansieringar av samma parameteriserade typ kan *inte* användas i varandras ställe.

- Slutsats: De olika formerna av polymorfism har helt olika användningsområden. De samspelar, men fyller olika funktion, därav att Java (och många andra språk) har båda formerna.

Live code

- Map

Polymorfism och variance

- Konceptet variance (se tidigare föreläsning) uppstår i samspelet mellan subtyping och parameteriserade typer.
  - E.g. om A <: B, hur förhåller sig då P<A> till P<B>?
- Metodsignaturer är …
  - E.g. om CatShelter är en subtyp till AnimalShelter, hur förhåller sig då putAnimal och adoptAnimal i CatShelter till motsvarande i AnimalShelter?

```
public class AnimalShelter {
  public void putAnimal(Animal a){…}
  public Animal adoptAnimal(){…}
}
```

  - Vilka typer är ok för putAnimal i CatShelter att ta som argument? Vad är ok för CatShelter att definiera som returtyp för adoptAnimal?

Metoder och variance

- Generellt i Java gäller: metoder är *covarianta* i sin returtyp, och *invarianta* i sina argumenttyper.
  - Vi skulle kunna tillåta *contravarianta* argumenttyper – vissa språk gör det – men Java (och e.g. C#) betraktar det som overloading om vi varierar argumenttypen åt något håll.

```
public class CatShelter extends AnimalShelter {
  public void putAnimal(Cat a){…}
  public Cat adoptAnimal(){…}
}
```

Ok: Cat <: Animal, dvs covariance

Detta ger overloading: putAnimal kan inte vara covariant i sin argumenttyp

Live code

- …

What’s next

Block 4-1:

Dependencies –
Can’t live with ’em, can’t live without ’em