Generic type declarations

Objekt-orienterad programmering och design

Alex Gerdes, 2018

Polymorfism

- Polymorfism är värdefullt – ger hög grad av reuseability och extensibility.
  - You know this by now…
- I Java finns två sorters polymorfism:
  - **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 varit den enda och förhärskande formen.
  - **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).
- (I vissa språk finns även en tredje form:
  - Ad hoc polymorphism: Olika typer kan oberoende av varandra ge definitioner för samma namn/symbol. E.g. typklasser i Haskell, operator overloading i C#.)

Polymorfism – skillnader

- Subtype polymorphism handlar om att konkreta objekt uppträder på samma sätt (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.

Parametric polymorphism för 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*).

Parametric polymorphism för klasser

- Ni har redan sett och använt *instanser* av parameteriserade typer i former av Collections:
  - `Collection<T>`, `ArrayList<T>`, `Deque<T>`, etc.

- Parameteriserade typer har många fler användningar än för collections, e.g.:
  - Utility-gränssnitt som kan specialiseras till en viss typ (`Iterable<E>`, `Comparable<E>`, …)
  - Funktions-gränssnitt (`Function<T,R>`, `BinaryOperator<T>`, `Predicate<T>`, …)
  - Wrapper-klasser (`JLayer(V extends Component)`, …)
  - …

Definiera parameteriserade typer

Introducera en typ-parameter

Inom klassen, använd som om den vore en typ.

```
public class MyClass<T>{
   T t;
   public MyClass(T t){ this.t = t; }
   public T getIt(){ return t; }

}

MyClass<String> myString = new MyClass(”Get it?”);
String gotIt = myString.getIt();
```

Övning

- Börja från koden som finns att ladda ner från hemsidan.
- Färdigställ implementationen av klassen `Tuple`. Tanken är att den ska fungera som en tupel i e.g. Haskell eller Python. Den ska hålla två värden av (potentiellt) olika (godtycklig) typ, och ha metoderna `fst()` och `snd()` för att returnera första respektive andra komponenten.
- Titta på koden för `AnimalShelter` med tillhörande kring-klasser. Kika särskilt på `ShelterError`. Hur kan vi tänka om, så att det inte är möjligt att sätta hundar i ett katt-hem, och vice versa, utan att förlora code reuse eller extensibility? Vi vill få ett statiskt felmeddelande, inte ett fel (exception) vid runtime.
- Skapa en `Lists` klass och implementera en statisk metod `zip` som tar två listor (`List<A>` och `List<B>`) som argument och returnerar en ny lista med tupel av `A` och `B` (`List<Tuple<A, B>>`). Resultat listan har samma längd som minsta argument lista.
- För utmaning: färdigställ implementationen av interfacet `Function`. Tanken är att det ska representera funktioner från argument av någon typ `T` till resultat av någon typ `R`. Interfacet ska ha metoderna `compose`, som sätter ihop två funktioner av lämpliga argument- och retur-typer, samt `apply`, som applicerar funktionen på ett lämpligt argument. Hur kan ni göra metoderna så polymorfa som möjligt?
- För extra utmaning: färdigställ implementationen av klassen `Either<A, B>`. Tanken är att den ska fungera som typen `Either` i Haskell. Den ska representera *antingen* ett objekt av typen `A` (annoterat `Left`) *eller* ett objekt av typen `B` (annoterat `Right`), och ha metoderna `isLeft()` och `isRight()`. Den ska också ha metoden `either` som tar två parametrar med typen `Function` (`left`, `right`) och antingen exekverar `left` om objektet är en ‘left’ annars köra `right`. (Obs! Kräver kännedom om Lambdas!).
- För extra, extra utmaning: Ge en default-implementation av metoden `Function.compose` från ovan (Obs! Kräver kännedom om Lambdas!).