Programmering 2
Övningsuppgift 2 - tillägg

Version: 1.0
Uppdaterad: 2021-04-07, se dokumenthistoriken sist i dokumentet för info.
Författare: Patrick Wentzel

Det här dokumentet är ett tillägg till instruktionen för övningsuppgift och är tänkt att komplettera föreläsningarna med information som är relevant för uppgiften.

Innehållsförteckning

Inledning

I uppgiften ska vissa klasser och metoder som inte har presenterats på föreläsningar användas. Det här dokumentet innehåller information om delar av klasserna Collections och Optional samt några metoder i datastrukturen TreeMap.

Det finns också en del som handlar om hur man kan komma igång med övningsuppgiften och ett exempel på stegen för att lösa ett problem.

Klassen Collections

I uppgiften ingår två krav som är tänka att hanteras med hjälp av metoder från klassen Collections:

- "Samlingar som returneras ska inte kunna modifieras (byta innehåll). Klassen `Collections` har flera användbara hjälpmetoder för det."

- "Metoder som returnerar samlingar ska returnera tomma samlingar om det inte finns något av det som efterfrågas."

Klassen är en del i Java Collections Framework[1] och beskrivs i dokumentationen som en klass som innehåller "static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection, and a few other odds and ends." - lite gott och blandat med andra ord. Det kan vara värt att ta en titt på listan över metoden i klassen - det finns mycket användbart i den som kan vara bra att känna till.

[1] Bra tutorial för Java Collections Framework

Oföränderliga/omodifierbara samlingar

Det första av de två kraven ovan är tänkt att förhindra att den som använder metoden och tar emot samlingen som returneras inte ska ha någon möjlighet att kunna ändra på innehållet.

Sättet vi kan göra det är genom att använda oss av några av metoderna i Collections som är tänkta för just det; metoderna heter `unmodifiableXYZ` där `XYZ` är namnet på den typ av samling[2] som de returnerar (vilket påverkar vilka metoder som finns tillgängliga och prestanda etc).

Metoderna tar emot en samling som parameter och returnerar en oföränderlig vy till samlingen, t.ex.:

```
public static <T> Set<T> unmodifiableSet (Set<? extends T> s)
```

som skulle kunna användas enligt nedan för att returnera en mängd med hundobjekt som heter dogs:

```
return Collections.unmodifiableSet(dogs);
```

[2] Bl.a. List, Set, Map, SortedMap, SortedSet

Tomma samlingar

Det andra kravet, att returnera tomma samlingar ifall värde saknas, finns för att den som använder klassen ska slippa anpassa kod för att hantera att en metod kan returnera null. I många fall kan det vara mer praktiskt att ge en tom samling[3].

Vi skulle kunna lösa problemet att returnera tomma samlingar genom att kontrollera att t.ex. returvärdet från ett anrop till en avbildnings metod get inte är null och om det är det skapa en tom samling av rätt sort som vi returnerar, men med hjälp av befintliga metoder i Map och Collections kan vi göra det mycket enklare.

I Collections finns ett flertal metoder som ger tillbaka tomma samlingar av olika typer som är precis vad vi vill ha; metoderna heter `emptyXYZ` där `XYZ` är typen av samling[4].

Utöver den "vanliga" metoden `get` som vi använder för att hämta värden i avbildningar (t.ex. HashMap och TreeMap), finns det en annan metod, `getOrDefault`, med signaturen `getOrDefault (Object key, V defaultValue)` som ger oss möjligheten att specificera vad som ska returneras ifall nyckeln inte är kopplad till ett värde.

För att lösa kravet att inte returnera `null` utan ge tillbaka en tom samling istället kan vi kombinera metoderna till följande för en metod som ska returnera ett objekt från en HashMap:

```
return booksByAuthor.getOrDefault(author, Collections.emptyList());
```

och kombinerat med det första kravet:

```
return Collections.unmodifiableList(booksByAuthor.getOrDefault(author,
Collections.emptyList()));
```

[3] Eller ett tomt objekt istället för null. Det finns ett etablerat designmönster som delvis handlar om samma sak: Null Object Pattern.

[4] Bl.a. List, Set, Map, SortedMap, SortedSet.

Klassen Optional

Optional som infördes i Java 8 är en klass som fungerar som en container för andra objekt som kan vara null med syftet att minska problem med null (som ju kan orsaka katastrofala fel och behöver hanteras helt korrekt), genom att "paketera" det som kan vara null och indikera om ett värde finns paketerat eller inte via metoderna `isEmpty` och `isPresent`.

För att använda värdet kan man sen använda olika metoder, t.ex. get (som kastar ett undantag om värdet saknas), eller orElse som ger ett annat värde tillbaka om Optional-objektet är tomt. Klassen innehåller flera olika metoder för att göra olika saker med värdet om det finns.

För att lösa kravet i uppgiften behöver vi dock bara veta hur vi kan skapa nya Optional-objekt som vi kan returnera, och precis som med de omodifierbara samlingarna ovan har klassen Optional ett par statiska metoder som "paketerar" värdet vi vill returnera `Optional.of(T value)` och `Optional.ofNullable(T value)` där den förra ger en NullPointerException ifall värdet value är null, och den senare istället returnerar ett tomt objekt, vilket är det som vi är intresserade av.

Exempel i en metod:

```
Optional<Owner> getOwner(){
  return Optional.ofNullable(owner);
}
```

För den som vill veta mer om den här lilla men användbara klassen finns det en bra introduktion[5] och förstås dokumentationen.

[5] och här är en till https://stackify.com/optional-java/

SortedMap & TreeMap

I uppgiften ingår att plocka fram data före eller efter ett årtal, eller mellan två årtal. För att kunna göra det på ett effektivt sätt behöver nycklarna i avbildningen vara sorterade, vilket kan göras genom att använda t.ex. en TreeMap.

Samlingar som implementerar gränssnittet SortedMap (t.ex. TreeMap) har några extra metoder som ger tillbaka vyer till samlingen:

`SortedMap<K, V> headMap (K toKey)`
Returns a view of the portion of this map whose keys are strictly less than toKey.

`SortedMap<K, V> tailMap (K fromKey)`
Returns a view of the portion of this map whose keys are greater than or equal to fromKey.

`SortedMap<K, V> subMap (K fromKey, K toKey)`
Returns a view of the portion of this map whose keys range from fromKey, inclusive, to toKey, exclusive.

Så om vi till exempel har en TreeMap<Price, Collection<House>> som representerar olika fastigheter kopplade till pris och vi vill ha ut allt objekt inom ett visst intervall skulle vi kunna göra:

```
SortedMap<Price, Collection<House>> houses = housesByPriceMap.subMap(1000, 3000);
```

vilket skulle skulle ge oss en vy som utgör en del av originalsamlingen. Om vi var intresserade av objekt med ett pris under eller över ett visst värde skulle vi kunna göra:

```
housesByPriceMap.headMap(1000); // x < 1000
```

respektive

```
housesByPriceMap.tailMap(1000); // x >= 1000
```

Det finns också ett ytterligare gränssnitt, NavigableMap som utökar SortedMap med ytterligare metoder för att bl.a. hitta nästa högre eller lägre nyckel osv.

För mängder (Set) finns det motsvarande gränssnitt SortedSet och NavigableSet.

Tips för att komma igång med uppgiften

Det rekommenderade sättet att lösa uppgiften är att utgå från metoderna och använda en "top-down"-ansats och utgå från det övergripande målet och dela upp det i mindre delar och sen lösa varje mindre del (eller dela upp de ytterligare).

Tanken med uppgiften är att man i princip ska utföra tre olika steg (för varje metod):

1. Fundera på hur metoden skulle kunna lösas (algoritmen), och vilken datastruktur som skulle kunna underlätta för lösningen.

2. Lägg till datastrukturen i klassen (som en medlemsvariabel).

3. Fyll datastrukturen med data i klassens konstruktor genom att loopa igenom samlingen med data som kommer in som parameter.

Det är möjligt att flera metoder kan arbeta med samma datastrukturer, fast på olika vis, om de behöver liknande data eller kunna hämta data på samma sätt.

Hur kan man välja datastrukturer?

Det finns flera olika aspekter att tänka på och metodernas namn och returvärden kan ge ledtrådar.

Det första valet är om det handlar om en mängd eller en avbildning - om man behöver kunna ta fram ett värde baserat på en nyckel så är en Map rätt, annars är nog en mängd rätt val[6];

Det andra valet är om vi behöver ha saker sorterade eller inte. Om saker ska vara sorterade är någon av de trädbaserade samlingarna rätt (t.ex. TreeMap och TreeSet). Här gäller det att tänka på att de sorterande samlingarna ställer särskilda krav på möjligheter att jämföra.

För avbildningar är nästa fråga vad som ska vara nyckel och vad som ska vara värde, och det beror på hur användningsfallet ser ut och hur nyckel och värde relaterar till varandra. Om vi vill kunna hitta personer utifrån personnummer skulle vi vilja ha Map<PersonNummer, Person> eftersom varje pnr bara är kopplat till en person. Om vi däremot skulle vilja kunna hitta alla som deltar i en viss kurs skulle vi vilja ha Map<KursKod, Collection<Student>> - en samling av samlingar kallas ofta för multimap och ställer vissa krav på hur man arbetar med den (se föreläsning 6 eller boken om det).

För uppgiften finns det ledtrådar i namn på metoderna, och returvärdena som i många fall är samma som det som borde vara värdet i en Map eller hur ett Set borde vara definierat.

---

[6] Eftersom listor är förbjudna i uppgiften… annars skulle listor absolut kunna vara rätt val också i flera fall.

Ett exempel

Om vi har en metod (vi använder böcker som exempel) som heter `getBooksByPublisher(String name)` och som ska returnera en `Collection<Book>` så kan ger returvärde en indikation om vad värdet borde vara, och att det handlar om en Map eftersom vi vill hitta något via ett attribut (namnet på ett förlag). Också det faktum att Books står i plural indikerar att värdet ska vara en samling, och alltså är det vi är intresserade av en `Map<String, Collection<Book>>`.

En metod som den ovan behöver egentligen inte mycket mer än definition av datastrukturen och sen ett metodanrop för att hämta ut data för att kunna lösas.

En del andra metoder kräver att man gör saker i flera steg och kanske använder tillfälliga samlingar för att mellanlagra resultat under bearbetning - det är viktigt att fundera på algoritmen först.

En del av metoderna är tänkta att lösa med hjälp av mängdoperationer.

Man kan bygga flera "nivåer" av samlingar för att underlätta operationer (men det kan också krångla till saker genom ökad komplexitet), så om vi t.ex. skulle vilja snabbt kunna ta fram alla studenter som har läst en viss kurs ett visst år skulle man kunna tänka sig en Map<Student, Map<Year, Set<Course>>> som skulle låta oss göra saker som:

```
Set<Course> coursesTakenIn2012 = map.get("Patrick").get(2012);
```

Eller om det var en TreeMap som värde:

```
Collection<Set<Course>> coursesFrom2012To2013 = map.get("Patrick").subMap(2012,
2014).values();
```

Det är bara fantasin, behovet och omdömet som sätter gränser.

Dokumenthistorik

Version	Datum	Ändringar
1.0	2021-04-06	Första version.