Veckoövning V5: Djurparken 2.0

I veckans föreläsning har vi tittat närmare på *abstrakta klasser*, *interfaces* ("gränssnitt" i kursboken) och *listor*. I den här övningen kommer du att använda de koncepten för att förbättra djurparken från förra veckans övning.

Om du inte gjorde förra veckans övning så går det utmärkt att göra den här övningen mer teoretiskt: skriv ned vilka ändringar du hade gjort för att få förra veckans övning att även uppfylla kraven för den här övningen.

Om du gör övningen med IntelliJ IDEA kan det vara en god idé att använda debuggern för att stega igenom ditt program och undersöka vad som händer under körning, speciellt för delarna 4-6.

Notera att denna övning är fullständigt frivillig, och inte kommer betygsättas.

1. Förberedelser

Gå igenom veckans föreläsning och läsanvisningar. Läs igenom förra veckans veckoövning.

2. Abstrakta djur

Arv används för att låta en uppsättning *subklasser* dela alla (publika) metoder och fält från sin superklass. Det är inte sällan som detta resulterar i att superklassen *i sig* inte representerar någonting konkret. Vad för sorts djur representerar till exempel klassen `Animal`?

I sådana situationer väljer vi ofta att markera superklassen som `abstract`. Detta förhindrar den som använder våra klasser från att skapa objekt av den abstrakta superklassen. Hen måste istället skapa objekt av någon av dess icke-abstrakta subklasser, till exempel `Cat`, `Dog` eller `Wallaby`.

Gå igenom klasserna du skapade i förra veckans övning. Vilka av dem kan göras abstrakta? Vilka av dem *bör* göras abstrakta? Markera de klasser som både kan och bör göras abstrakta som `abstract`.

3. Djuriska egenskaper

Även om vi kan modellera djurs genetiska indelning med hjälp av arv, så stöter vi på problem när det gäller deras beteenden. Det bästa exemplet på detta är delfiner: de lever i havet, simmar som fiskar, äter samma saker som fiskar, men är i själva verket däggdjur. Detta modellerar vi bäst med hjälp av `interface`s.

Gå igenom klasserna du skapade i förra veckans övning, och försök identifiera några olika egenskaper som delas mellan två eller flera av dina klasser. Uttryck dessa beteenden i form av interfaces, och låt alla djur som delar ett visst beteende implementera det interface som representerar beteendet. Tänk på att en klass kam implementera flera olika interfaces.

Ett av dessa interfaces bör vara `Hunter`, som tilldelas djur som äter andra djur. Ett annat bör vara `Prey`, som tilldelas djur som *blir ätna* av andra djur.

Notera att om en superklass implementerar ett visst beteende, så gör även alla dess subklasser det. Detta innebär att om du har ett interface `CanSwim` som implementeras av klassen `Fish`, så kommer även dess subklasser `Salmon`, `Cod` och `Carp` implementera `CanSwim`.

4. Listor

I förra veckans övning implementerade du en main-metod som bland annat innehöll en array av djur. Ersätt denna array med tre listor av djur:

- `List<Animal> animals` som innehåller *samtliga* djur,
- `List<Hunter> hunters` som innehåller alla rovdjur, och
- `List<Prey> prey` med alla djur som riskerar att bli uppätna.

Modifiera även funktionaliteten för att interagera med djuren i din djurpark så den fungerar med dessa tre listor. För att lista alla djur räcker det med att använda `animals`, men om djur av någon anledning tas bort ur

5. Matdags!

Lägg till funktionalitet för att "öppna dörren" mellan `hunters` och `prey`. När detta sker väljs ett slumpmässigt rovdjur ut som jägare, och ett slumpmässigt bytesdjur som byte. Låt sedan jägaren och bytet slå varsin sexsidig tärning; om jägaren slår högre än bytet blir bytet uppätet och tas därmed bort ur listan `prey`. Ett uppätet bytesdjur behöver även markeras som uppätet på annat vis, antingen genom att alla djur ges ett fält `boolean alive` eller dylikt, eller genom att uppätna djur även tas bort från listan `animals`.

Tips: använd klassen `Random` för att välja ut slumpmässiga djur och simulera tärningsslag. Skapa gärna hjälpmetoder, som t ex `Hunter chooseHunter(List<Hunter> hunters)` och `int rollDice(int sides)`.

6. Fler listor!

På samma sätt som `hunters` innehåller en delmängd av `animals`, så kan djur med andra beteenden även placeras i andra listor. Om man kan simma med fiskar och delfiner på djurparken kan detta t ex modelleras med en lista `List<Swimmer> swimmers`, och lämplig funktionalitet för att låta användaren simma med ett djur ur den listan.

Skapa en eller flera dylika listor, och funktionalitet som låter användaren interagera med dem. Tänk på hur listorna interagerar med varandra. Exempelvis kan det (beroende på hur du löste del 5) vara nödvändigt att ta bort djur ur dessa listor när de blir uppätna.