# Lösningsförslag

### Lambda-uttryck

U 1. I detta exempel skrivs sex personer ut, nämligen de som fyllt 18 år:

```
Jonsson, Camilla (23)
Hermansson, Lena (38)
Lundström, David (21)
Björk, Stefan (20)
Andersson, Gun (55)
Svensson, Ulf (47)
```

U 2. `print(persons,  p -> p.getAge() >= 18);`

Klassen `AgeCondition` behövs inte längre, eftersom lambdauttrycket fungerar som ett `PersonCondition`-objekt.

U 3. `print(persons,  p -> p.getName().startsWith("Lund"));`

Det är inte heller här nödvändigt att införa någon ny klass för att implementera interfacet `PersonCondition`. Lambdauttrycket är allt vi behöver.

U 4. `persons.removeIf(p -> p.getName().startsWith("B"));`

I en fotnot i uppgiften nämns den lite komplicerade typparametern i `removeIf`. Denna spelar bara roll då man *inte* använder lambdauttryck. En längre kommentar finns i samband med U 6 nedan; en liknande typparameter återfinns nämligen där i `List.sort`.

U 5. `Comparator` är ett funktionellt interface eftersom det innehåller exakt en abstrakt metod. Det kan därför användas för att beskriva en parameter, där ett lambdauttryck är tillåtet som argument. Den följande uppgiften demonstrerar detta.

U 6. `persons.sort((p1, p2) -> p1.getName().compareTo(p2.getName()));`

#### Kommentar: om typparametern (för dig som är särskilt intresserad)

I `List` är metoden deklarerad med lite annorlunda typparameter än i uppgiften, nämligen så här:

```
void sort(Comparator<? super E> c);
```

Det betyder att när vi vill sortera en lista med objekt av klassen `E`, så går det bra att använda en `Comparator` som kan jämföra objekt av en *superklass* till `E`. Det visar sig främst spela roll om man inte använder lambdauttryck. Anta att vi exempelvis inför klassen `Student`, som subklass till `Person`:

```
public class Student extends Person {
    // ... attribut, konstruktor, metoder...
}
```

Anta vidare att vi redan har en `Comparator`-klass som kan jämföra `Person`-objekt, så här:

```
public class PersonComparator implements Comparator<Person> {
    public int compare(Person p1, Person p2) { ... }
}
```

Då betyder typparametern för `sort(Comparator<? super E> c)` att en lista av typ `List<Student>` inte nödvändigtvis måste sorteras av en `Comparator<Student>`, utan att det går lika bra med vår tidigare `Comparator<Person>`:

```
List<Student> students = ...; // skapa listan, fyll med Student-objekt
students.sort(new PersonComparator());
```

När vi använder lambdauttryck har detta mindre betydelse. Om vi sorterar `Student`-objekt med hjälp av deras `Person`-egenskaper, men använder ett lambdauttryck istället för `PersonComparator`, får vi något i den här stilen (om vi sorterar studenter efter ålder):

```
List<Student> students = ...;
students.sort((s1, s2) -> s1.getAge() - s2.getAge());
```

Här räknar Java automatiskt ut typerna på `s1` och `s2`, och vi får automatiskt en `Comparator` av rätt typ. Därför fungerar lambdauttrycket precis lika bra oavsett om `sort` vore deklarerad som

```
void sort(Comparator<E> c);
```

eller

```
void sort(Comparator<? super E> c);
```

U 7. `persons.sort((p1, p2) -> p1.getAge() - p2.getAge());`

Om man inte kommer på tricket med subtraktion ovan kan man använda ett lite längre lambdauttryck:

```
persons.sort((p1, p2) -> {
    if (p1.getAge() < p2.getAge()) {
        return -1;
    } else if (p1.getAge() > p2.getAge()) {
        return 1;
    } else {
        return 0;
    }
});
```

Om man ändå vill skriva alltsammans som ett enda uttryck, och slippa skapa ett block med satser som ovan, så kan man använda en kombination av villkorsuttryck. Det blir emellertid ganska kryptiskt.

```
persons.sort((p1, p2) -> (p1.getAge() < p2.getAge()) ? -1
    : ((p1.getAge() > p2.getAge()) ? 1 : 0));
```

**Kommentar: om overflow (för dig som är särskilt intresserad)**

Differensen i det första lösningsförslaget ovan, `p1.getAge() - p2.getAge()`, får fel tecken om termerna är *väldigt* stora (i storleksordningen `Integer.MAX_VALUE` eller `MIN_VALUE`), *och* har olika tecken. Problemet kallas *overflow*, och har att göra med att datatypen `int` har ett begränsat maximalt antal siffror (32 bitar, binära siffror).

Om vi exempelvis subtraherar åldrarna $a = 2^{31} - 1$ och $b = -2^{31}$ får vi $a - b = 2^{31} - 1 - (-2^{31}) = 2^{32} - 1$, som inte ryms i ett `int`-värde. Termen $2^{32}$ faller nämligen då bort, och resultatet blir negativt ($-1$) istället för positivt. (Det funkar ungefär som när apotekets könummer, som är begränsat till två siffror, slår över från 99 till 00; hundratalet faller då bort.)

I vårt fall behöver vi knappast bekymra oss, eftersom människors åldrar är positiva tal, och då kan overflow inte uppstå i subtraktionen. Ingen människas ålder är heller i närheten av `Integer.MAX_VALUE` $= 2^{31} - 1 \approx 2.1 \cdot 10^9$ år.

Overflow, när det ändå uppstår, kan vara klurigt att hantera. Det klokaste är oftast att välja en större datatyp, så att man helt slipper trassla in sig i problemet. Om vi exempelvis byter `int` till `long` kan vi hantera åldrar upp till $2^{63} - 1 \approx 9.2 \cdot 10^{18}$ år.

**Att hantera overflow**

Vill man skydda sig mot overflow, och korrekt hantera `int`-heltal som ligger nära `int`-intervallets gränser, kan man använda den statiska metoden `Integer.compare`:[a]

```
persons.sort((p1, p2) -> Integer.compare(p1.getAge(), p2.getAge()));
```

(De båda andra lösningsförslagen ovan, if-satsen och villkorsuttrycken, undviker overflow eftersom ingen differens beräknas.)

[a] `https://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html#compare-int-int-`

U 8. Namnet på metoden kan vi välja fritt. Här har vi valt att kalla den `compute`.

```
public interface Func {
    double compute(double x);
}
```

U 9. Det namn vi valt ovan, `compute`, är också det vi anropar i `deriv`.

```
public static double deriv(Func f, double x) {
    final double h = 1e-6;
    return (f.compute(x + h) - f.compute(x)) / h;
}
```

U 10. Vårt `Func` motsvaras av `Function<Double,Double>`. I det senare interfacet råkar metoden heta `apply`.

```
public static double deriv(Function<Double,Double> f, double x) {
    final double h = 1e-6;
    return (f.apply(x + h) - f.apply(x)) / h;
}
```

Exempelkoden i uppgiften förblir oförändrad, även om vi använder `Function`:

```
double yp = deriv(x -> Math.sin(x),  0.3);
```

U 11. Returvärdet är en funktion, närmare bestämt ett lambdauttryck.

```
public static Func deriv(Func f) {
    final double h = 1e-6;
    return x -> (f.compute(x + h) - f.compute(x)) / h;
}
```