Läsanvisningar

Börja med att läsa kapitlen i kursboken, därefter resurserna på Internet och till sist själva lektionsmaterialet.

Kursboken (8:e upplagan): Kapitel 10.9 och 17.8.

Internet: The Java Tutorial, Trail: Learning the Java Language: - Lambda Expressions (https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html)

Internet: The Java Tutorial, Trail: Learning the Java Language: - Method References (https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html)

Internet: Java SE 8: Lambda Quick Start (http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html)

Anonyma inre klasser

Från och med version 8 av Java är det möjligt att skriva kod som ser ut så här:

```
myJButton.addActionListener(e -> System.out.println("click"));
```

Det gulmarkerade uttrycket ovan är ett så kallat lambda-uttryck. Det vi kan göra med ett sådant uttryck är, väldigt förenklat sagt, att skicka en metod som ett argument i ett metodanrop. När det gäller lambda-uttryck pratas det dock inte om metoder utan om funktioner i stället.

Just det här lambda-uttrycket beskriver en funktion som har en parameter e och som skriver ut texten click. Funktionen skickas som en parameter till metoden addActionListener hos objektet myJButton (som är av typen JButton). Lägger vi till denna knapp i t.ex. en JFrame kommer texten click att skrivas ut i kommandofönstret varje gång användaren klickar på knappen.

Tack vare införandet av lambda-uttryck kan vi nu skriva betydligt mer kompakt och lättläst kod. I stället för lambda-uttrycket i exemplet ovan har vi hittills varit tvungna att antingen:

1. Låta klassen implementera gränssnittet ActionListener och överskugga dess metod actionPerformed enligt:

```
@Override
public void actionPerformed(ActionEvent e) {
  if (e.getSource() == myJButton) {
    System.out.println("click");
  }
}
```

2. Skapa en anonym inre klass enligt:

```
myJButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    System.out.println("click");
  }
});
```

Anonyma inre klasser erbjuder oss ett sätt att skriva klasser som bara kommer att användas en gång i ett program. Framför allt brukar det vara vanligt i applikationer, som använder Swing eller JavaFX, att skriva ett antal händelsehanterare för händelser som genereras av t.ex. tangentbord eller mus. Hellre än att skriva en särskild klass för varje händelse som ska hanteras, skriver vi anonyma inre klasser. Genom att skapa klassen på plats, där den behövs, blir det lite enklare att läsa koden. Koden blir dock inte särskilt elegant, eftersom det krävs en hel del kod bara för att definiera en enda metod.

Funktionsgränssnitt

Gränssnitt med bara just en enda (abstrakt) metod kallas från och med Java 8 för ett funktionsgränssnitt (eng. functional interface). Ett exempel på ett sådant gränssnitt är ActionListener som vi använt i exemplen ovan. Dess definition ser ut så här:

```
package java.awt.event;
import java.util.EventListener;

public interface ActionListener extends EventListener {
  public void actionPerformed(ActionEvent e);
}
```

Förutom att innehålla en enda abstrakt metod får funktionsgränssnitt även innehålla statiska metoder, default-metoder och metoder ärvda från klassen Object. Om vi vill att kompilatorn ska kontrollera att dessa krav uppfylls när vi skriver ett funktionsgränssnitt kan vi lägga till annotationen @FunctionalInterface på raden ovanför deklarationen av gränssnittet. I exemplet nedan skapar vi ett funktionsgränssnitt med namnet Formula med en metod calculate som tar två argument av typen double. Tanken med gränssnittet är att vi ska kunna utföra beräkningar på de två talen.

```
@FunctionalInterface
public interface Formula {
  public double calculate(double a, double b);
}
```

För att skapa ett nytt Formula-objekt som kan addera två tal med varandra kan vi använda en anonym inre klass enligt följande exempel:

```
Formula addFormula = new Formula() {
  public double calculate(double a, double b) {
    return a + b;
  }
};

double a = 4;
double b = 5;
double sum = addFormula.calculate(a, b); // 9

System.out.println("The sum of " + a + " and " + b + " is " + sum);
```

Att använda funktionsgränssnitt med anonyma inre klasser enligt ovan är ett vanligt designmönster i Java. Förutom ActionListener kan även gränssnitt som Runnable och Comparator användas på ett liknande sätt. Funktionsgränssnitt visar sig vara något väldigt centralt när det kommer till användning av lambda-uttryck.

Lambda-uttryck

Ett lambda uttryck kan ses som ett block av kod som kan skickas runt så att koden kan exekveras vid ett senare tillfälle, antingen bara en gång eller flera gånger. Tittar vi återigen på första exemplet:

```
myJButton.addActionListener(e -> System.out.println("click"));
```

ser vi att vi gett ett lambda-uttryck som argument vid ett metodanrop. Det går även att tilldela ett lambda-uttryck till en referensvariabel eller returnera ett lambda-uttryck som ett resultat från en metod. Då krävs att referensvariabeln eller returtypen är ett funktionsgränssnitt. Exemplet ovanför skulle vi därför kunna dela upp i två steg enligt:

```
ActionListener l = e -> System.out.println("click");
myJButton.addActionListener(l);
```

Ett exempel på återanvändning av ett lambda-uttryck är:

```
ActionListener l = e -> System.out.println("click");
myJButton1.addActionListener(l);
myJButton2.addActionListener(l);
```

Nu kommer båda knapparna att skriva ut texten click när användaren trycker på knapparna.

Ett lambda-uttryck består av tre delar:

Parameterlista	Pilsymbol	Funktionskropp
`(int x, int y)`	`->`	`x + y`

Parameterlistan kan ha någon av följande former:

- `()`
  när funktionen saknar parametrar
- `(typ1 param1, typ2 param2, …)`
  vanlig fullständig parameterlista där både typ och namn på parametern eller parametrarna anges
- `(param1, param2)`
  när kompilatorn själv kan räkna ut vilka typer parametrarna har
- `param`
  när det finns en enda parameter och kompilatorn själv kan räkna ut dess typ

Funktionskroppen kan ha någon av följande former:

- `{ deklarationer och satser }`
  vanligt block av kod omgiven av klamrar och som kan innehålla en eller flera return
- `uttryck`
  motsvarar { return uttryck; }

Funktionskroppen kan alltså vara antingen ett enda uttryck eller ett block med deklarationer och satser. När kroppen består av ett enkelt uttryck evalueras helt enkelt uttrycket och resultatet returneras. När kroppen består av ett block exekveras rad för rad i kroppen, precis som i en metod, och ett return-uttryck återger kontrollen till anroparen, d.v.s. återigen precis som vanligt. Om

returtypen är void behöver vi inte skriva ut return. Nyckelorden break och continue är i funktionskroppen endast tillåtna att användas inuti loopar.

Nedan följer nu några exempel på lambda-uttryck tillsammans med korta kommentarer.

1. `() -> {}`
   Inga parametrar, inget returneras.
2. `() -> 42`
   Inga parametrar, enkelt uttryck, 42 returneras.
3. `() -> null`
   Inga parametrar, enkelt uttryck, null returneras.
4. `() -> { return 42; }`
   Inga parametrar, enkelt uttryck, 42 returneras (samma som exempel 2).
5. `() -> { System.out.println("Hello!"); }`
   Inga parametrar, block av kod, inget returneras
6. Inga parametrar, komplext block av kod med flera return-satser.
```
() -> {
  if (true) return 10;
  else {
    int result = 15;
    for (int i = 1; i < 10; i++)
      result *= i;
    return result;
  }
}
```
7. `(int x) -> x+1`
   Fullständig parameterlista med en parameter av typen int, enkelt uttryck, ett heltal returneras.
8. `(int x) -> { return x+1; }`
   Samma som exempel 7.
9. `(x) -> x+1`
   Parameterlista med en parameter där kompilatorn själv listar ut vilken typ parametern har, enkelt uttryck, ett tal returneras.
10. `x -> x+1`
    Samma som exempel 9, parenteser behövs inte om endast en parameter och dess typ kan bestämmas av kompilatorn.
11. `(String s) -> s.length()`
    Fullständig parameterlista med en parameter av typen String, enkelt uttryck, strängens längd returneras.
12. `s -> s.length()`
    Parameterlista med en parameter där kompilatorn själv listar ut vilken typ parametern har, enkelt uttryck, parameterns längd returneras.
13. `(int x, int y) -> x+y`
    Fullständig parameterlista med två parametrar av typen int, enkelt uttryck, ett heltal returneras.
14. `(x,y) -> x+y`
    Parameterlista med två parametrar där kompilatorn själv listar ut vilken typ parametrarna har, enkelt uttryck, summan av parametrarna returneras.
15.
```
(int x, int y) -> {
  int sum = x+y;
  System.out.println("The sum of " + x + " and " + y + " is " + sum);
}
```
    Fullständig parameterlista med två parametrar av typen int, block av kod, inget returneras.

Varför använda lambda-uttryck?

Som vi redan har nämnt är den främsta anledningen till att använda lambda-uttryck att vi med dessa kan skriva mer kompakt och lättläst kod. Anta att vi med funktionsgränssnittet Formula vill skapa objekt som kan räkna enligt de fyra grundläggande räknesätten. Att göra detta utan att använda lambda-uttryck skulle se ut så här:

```
Formula additionFormula = new Formula() {
  public double calculate(double a, double b) {
    return a + b;
  }
};

Formula subtractionFormula = new Formula() {
  public double calculate(double a, double b) {
    return a - b;
  }
};

Formula multiplicationFormula = new Formula() {
  public double calculate(double a, double b) {
    return a * b;
  }
};

Formula divisionFormula = new Formula() {
  public double calculate(double a, double b) {
    return a / b;
  }
};
```

Vi kan nu ersätta dessa med lambda-uttryck istället genom att börja med att kopiera metoden calculate:s parameterlista och använda den som parameterlista i lambda-uttrycket.

```
(double a, double b)
```

Därefter följer pilsymbolen:

```
(double a, double b) ->
```

Till sist tar vi hela metodkroppen från calculate och använder som funktionskropp i lambda-uttrycket. Vi får då följande kompletta lambda-uttryck som vi kan tilldela en referensvariabel av typen Formula:

```
Formula additionFormula = (double a, double b) -> { return a + b; };
```

Eftersom tilldelningen sker till en typ av Formula kan kompilatorn själv räkna ut vilken typ parametrarna i parameterlistan ska ha. Detta genom att funktionsgränssnittet Formula:s enda abstrakta metod är calculate och den tar två double som parametrar. Vidare består funktionskroppen endast av ett enkelt uttryck. Vi kan därför förenkla lambda-uttrycket så här:

```
Formula additionFormula = (a, b) -> a + b;
```

Den kod som använde anonyma inre klasser och som upptog totalt 20 rader kan nu ersättas med följande fyra rader:

```
Formula additionFormula = (a, b) -> a + b;
Formula subtractionFormula = (a, b) -> a - b;
Formula multiplicationFormula = (a, b) -> a * b;
Formula divisionFormula = (a, b) -> a / b;
```

Vi kan snabbt konstatera att koden blivit betydligt kompaktare. Lika självklart kanske du inte håller med om att koden dessutom blivit mer lättläst. Det kan vara svårt så här i början att tolka och läsa lambda-uttryck, men med lite övning blir det lättare. Det gäller framför allt att ha koll på hur den abstrakta metoden i funktionsgränssnittet ser ut.

Ytterligare exempel på lambda-uttryck

Låt säga att vi har en lista innehållandes strängar med namn på olika programmeringsspråk.

```
List<String> names = Arrays.asList("C++", "Java", "JSP", "Python", "Pascal",
                                   "C#", "Cobol");
```

Om vi vill sortera denna lista i sjunkande bokstavsordning (z – a) har vi tidigare börjat med att skriva en Comparator i en separat klass:

```
public class DescendingAlphabeticalComparator implements Comparator<String> {
  @Override
  public int compare(String a, String b) {
    return b.compareTo(a);
  }
}
```

Sedan har vi sorterat listan med denna Comparator:

```
Collections.sort(names, new DescendingAlphabeticalComparator());
System.out.println(names.toString());
```

Lite mer kompakt kod får vi om vi skapar en anonym inre klass i stället för en separat klass:

```
Collections.sort(names, new Comparator<String>() {
  @Override
  public int compare(String a, String b) {
    return b.compareTo(a);
  }
});
```

Tittar vi nu på definitionen av gränssnittet Comparator ser vi en intressant sak:

```
package java.util;

@FunctionalInterface
public interface Comparator<T> {
  int compare(T o1, T o2);

  // samt en del default- och statiska metoder som jag inte tar med här
}
```

Det är definierat som ett funktionsgränssnitt och innehåller en enda abstrakt metod som heter compare. Tack vare detta kan vi ersätta den anonyma inre klassen med ett lambda-uttryck. Vi börjar med att kopiera metoden compare:s parameterlista från den anonyma klassen och använder den som parameterlista i lambda-uttrycket.

```
(String a, String b)
```

Vi lägger till pilsymbolen

```
(String a, String b) ->
```

Och till sist tar vi hela metodkroppen från compare och använder som funktionskropp i lambda-uttrycket. Vi får då följande kompletta lambda-uttryck:

```
(String a, String b) -> { return b.compareTo(a); }
```

Eftersom vi deklarerat listan över programmeringsspråken att innehålla objekt av typen String kan kompilatorn räkna ut vilken typ parametrarna i parameterlistan ska ha. Vidare består funktionskroppen av ett enkelt uttryck och vi kan därför förenkla lambda-uttrycket enligt följande:

```
(a, b) -> b.compareTo(a)
```

Vi får nu en väldigt kompakt och lättläst kod för att sortera listan i sjunkande bokstavsordning:

```
Collections.sort(names, (a, b) -> b.compareTo(a));
```

Kommer vi senare på att vi i stället vill sortera listan på ett annat sätt, t.ex. i strängarnas längd, är det enkelt att bara ändra på lambda-uttrycket:

```
Collections.sort(names, (a, b) -> Integer.compare(a.length(), b.length()));
```

Inbyggda funktionsgränssnitt

För att använda ett lambda-uttryck måste den typ som används vara ett funktionsgränssnitt. Vi har tidigare skrivit att lambda-uttryck kan:

- Ges som argument till en metod. Metodens parameter måste då vara en typ av ett funktionsgränssnitt. Som exempel har vi följande kod:

  ```
  myJButton.addActionListener(e -> System.out.println("click"));
  ```

  Tittar vi på metoden addActionlistener ser dess huvud ut så här:

  ```
  public void addActionListener(ActionListener l)
  ```

  Den har en parameter l som är av typen ActionListener. ActionListener är i sin tur ett gränssnitt med bara en abstrakt metod, d.v.s. ett funktionsgränssnitt. Därför är det ok i det här fallet att ge ett lambda-utryck som argument till metoden.

- Tilldelas till en referensvariabel. Denna variabel måste då ha ett funktionsgränssnitt som dess typ. Som exempel har vi haft följande kod:

  ```
  Formula additionFormula = (a, b) -> a + b;
  ```

  Här är referensvariabeln additionFormula av typen Formula som i sin tur är ett funktionsgränssnitt som vi själva skapade.

- Användas som ett resultat från en metod. Då krävs det att returtypen är ett funktions-gränssnitt. Som exempel utgår vi från följande funktionsgränssnitt som vi skapar själva:

```
@FunctionalInterface
public interface Drawable {
  public void draw(java.awt.Graphics g);
}
```

För att skapa ritbara linjer kan vi definiera följande metod vars returtyp är ovanstående funktionsgränssnitt:

```
public Drawable getDrawableLine(int x1, int y1, int x2, int y2) {
  return g -> { g.drawLine(x1, y1, x2, y2); };
}
```

I de två sista exemplen har vi skapat egna funktionsgränssnitt, men det är inte så vanligt att vi behöver göra det. Det finns flera inbyggda gränssnitt, så kallade standardgränssnitt, som är funktionsgränssnitt och alltså kan användas som typer för lambda-uttryck. Många gånger räcker dessa funktionsgränssnitt till för att skapa de lambda-uttryck vi behöver. Gränssnitten är definierade i paketet java.util.functions och finns tillgängliga från version 8 av Java.

Gränssnitten i det här paketet är generella funktionsgränssnitt som används av JDK. Samtidigt har de gjorts tillgängliga för att användas av användarkod skapad av utvecklare som dig. Även om de inte utgör en komplett uppsättning av funktionsgränssnitt för alla typer av lambda-uttryck, är de ofta tillräckliga för att täcka in de mest vanliga användarfallen. Några av de funktionsgränssnitt paketet innehåller listas i tabellen nedan.

Gränssnittets namn	Funktionens namn	Funktionens profil
`Predicate<T>`	`test`	`T -> boolean`
`Consumer<T>`	`accept`	`T -> void`
`Function<T, V>`	`apply`	`T -> V`
`Supplier<T>`	`get`	`() -> T`
`UnaryOperator<T>`	`apply`	`T -> T`
`BinaryOperator<T>`	`apply`	`(T, T) -> T`

Många av dessa funktionsgränssnitt finns även i versioner som arbetar med primitiva typer som IntPredicate, BooleanSupplier och DoubleConsumer m.fl. Utöver den abstrakta metoden innehåller gränssnitten även default-metoder och statiska metoder. Vi ska nu titta närmare på några av dessa gränssnitt.

Predicate<T>

Detta funktionsgränssnitt representerar en funktion som tar ett argument av godtycklig typ T och som returnerar en boolean. Ett exempel på användning är:

```
Predicate<Integer> greaterThanTen = i -> i > 10;

System.out.println(greaterThanTen.test(14)); // true
System.out.println(greaterThanTen.test(7));  // false
```

Ett Predicate kan nästlas med andra Predicate via dess default-metoder and, or eller negate. Nästa exempel är ett enkelt exempel på nästling, men man kan åstadkomma väldigt komplexa regler genom att nästla olika Predicate med varandra.

```
Predicate<Integer> greaterThanTen = i -> i > 10;
Predicate<Integer> lowerThanTwenty = i -> i < 20;

greaterThanTen.and(lowerThanTwenty).test(15);  // true
greaterThanTen.and(lowerThanTwenty).negate().test(15); // false
```

Givetvis kan ett Predicate skickas som argument till en metod:

```
// Skriver ut: Number 10 passed the test!
process(10, i -> i > 7);

public void process(int number, Predicate<Integer> predicate) {
  if (predicate.test(number)) {
    System.out.println("Number " + number + " passed the test!");
  }
}
```

Som ett sista exempel på användning av Predicate ska vi titta på hur vi med hjälp av ett sådant kan filtrera ut vissa element ur en lista. Vi utgår från följande klass:

```
public class User {
  private String name;
  private String role;

  public User(String namne, String role) {
    this.name = name;
    this.role = role;
  }

  public String getRole() {
    return role;
  }
}
```

Vi skapar nu en lista och lägger till några användare:

```
List<User> users = new ArrayList<>();
users.add(new User("Robert", "admin"));
users.add(new User("Sara", "member"));
users.add(new User("Kalle", "member"));
users.add(new User("Fia", "admin"));
```

Vi vill därefter skapa nya listor som bara innehåller medlemmar vars roll är admin respektive member. Ett första dåligt försök vore att skapa två metoder som tar medlemslistan som argument och som filtrerar ut bara admin och member enligt följande:

```
public List<User> getAdminList(List<User> users) {
  List<User> result = new ArrayList<>();

  for (User user : users) {
    if (user.getRole().equals("admin")) {
      result.add(user);
    }
  }

  return result;
}
```

```
public List<User> getMemberList(List<User> users) {
  List<User> result = new ArrayList<>();

  for (User user : users) {
    if (user.getRole().equals("member")) {
      result.add(user);
    }
  }

  return result;
}
```

Som sen kan anropas på följande sätt:

```
List<User> admins = getAdminList(users);
List<User> members = getMemberList(users);
```

Som vi ser blir det väldigt mycket upprepning av kod i de båda metoderna. En annan nackdel med detta tillvägagångssätt är att om vi kommer på att vi senare vill filtrera listan på ett annat sätt, måste vi lägga till en ny metod. En bättre lösning är att använda sig av Predicate på följande sätt:

```
List<User> admins = getUserList(users, user -> user.getRole().equals("admin"));
List<User> members = getUserList(users, user -> user.getRole().equals("member"));

public static List<User> getUserList(List<User> users, Predicate<User> predicate)
{
  List<User> result = new ArrayList<>();

  for (User user : users) {
    if (predicate.test(user)) {
      result.add(user);
    }
  }

  return result;
}
```

Function<T, V>

Detta funktionsgränssnitt representerar en funktion som tar ett argument av godtycklig typ T och som producerar ett resultat av typen V av detta argument. Båda typerna kan vara av samma typ. Ett första exempel är en funktion som tar en sträng som argument och returnerar ett heltal (Integer) som innehåller strängens längd.

```
Function<String, Integer> stringLength = s -> s.length();
System.out.println(stringLength.apply("Lamdas are fun!")); // 15
```

Precis som med Predicate kan en Function nästlas med en annan Function. Då använder vi gränssnittets default-metod andThen. Metoden andThen returnerar en sammansatt funktion som först tillämpar den funktion metoden anropas på och sedan tillämpar den funktion som ges som argument till metoden. Den funktion som returneras kan i sin tur nästlas genom att återigen anropa andThen o.s.v.

```
Function<String, Integer> stringLength = s -> s.length();
Function<Integer, Double> half = i -> i / 2d;
System.out.println(stringLength.andThen(half).apply("Lamdas are fun!")); // 7.5
```

Notera hur det andra typargumentet Integer i funktionen stringLength blir det första typargumentet i funktionen half. Med andra måste resultatet av den första funktionen var ingångsvärde i den funktion den ska nästlas med.

Som ett sista exempel ska vi titta på hur vi med hjälp av en Function kan konvertera vår lista med User till en sträng som innehåller all information om användaren i formen namn|roll. Vi börjar med att skapa en metod som tar listan över användare och en Function<User, String> som anger hur konverteringen ska göras:

```
public List<String> getUsersAsString(List<User> users,
                                      Function<User, String> function) {
  List<String> result = new ArrayList<>();

  for (User user : users) {
    String s = function.apply(user);
    result.add(s);
  }

  return result;
}
```

Därefter kan vi anropa metoden med följande Function:

```
List<String> userStrings = getUsersAsString(users,
  user -> user.getName() + "|" + user.getRole());

System.out.println(userStrings.toString());
// [Robert|admin, Sara|member, Kalle|member, Fia|admin]
```

Vill vi senare ha strängen formaterad på ett annat sätt kan vi bara anropa metoden med en ny Function:

```
List<String> userStrings = getUsersAsString(users,
  user -> user.getRole().toUpperCase() + ";" + user.getName());

System.out.println(userStrings.toString());
// [ADMIN;Robert, MEMBER;Sara, MEMBER;Kalle, ADMIN;Fia]
```

Consumer<T>

Detta funktionsgränssnitt representerar en funktion som tar ett argument av godtycklig typ T och som inte returnerar något resultat (void). Ett exempel på användning är att skriva ut argumentet med System.out:

```
User user = new User("Robert", "admin");

Consumer<User> printName = u -> System.out.println("Name: " + u.getName());
printName.accept(user); // Name: Robert
```

Precis som med Function är det möjligt att nästla två Consumer med varandra genom att använda default-metoden andThen.

```
Consumer<User> printRole = u -> System.out.println("Role: " + u.getRole());
printName.andThen(printRole).accept(user); // Name: Robert
                                           // Role: admin
```

Supplier<T>

Detta funktionsgränssnitt representerar en funktion som inte tar något argument men som returnerar ett objekt av godtycklig typ T när vi anropar metoden get. Några enkla exempel på användning:

```
Supplier<Integer> supply42s = () -> 42;
System.out.println(supply42s.get()); // 42

Supplier<User> newUserSupplier = () -> new User();
User user = newUserSupplier.get();
user.setName("Kalle");

Supplier<LocalDate> todaysDateSupplier = () -> LocalDate.now();
System.out.println(todaysDateSupplier.get()); // 2015-09-24
```

Metodreferenser

Precis som namnet antyder är en metodreferens (eng. method reference) en referens till en metod. Metodreferenser låter oss återanvända existerande metoddefinitioner och skicka runt dessa på samma sätt som vi kan göra med lambda-uttryck. Om ett lambda-uttryck bara skickar vidare sina parametrar till en annan metod (statisk eller instans), utan att göra något med parametrarna, kan en metodreferens ersätta lambda-uttrycket. Metodreferenser kan på så sätt göra koden ytterligare lite mer kompakt och lättläst.

En metodreferens består, precis som ett lambda-uttryck, av tre delar:

Målreferens	Avgränsare	Metod
`String`	`::`	`length`

Metodreferenser kan användas på följande sätt:

- Som en statisk metod (Klassnamn::metodnamn)
- En instansmetod för ett specifikt objekt (referensvariabel::metodnamn)
- En metod från en superklass (super::metodnamn)
- En instansmetod på godtyckligt objekt med en specificerad typ (Klassnamn::metodnamn)
- En konstruktor-referens (Klassnamn::new)
- Allokering av en array (Typnamn[]::new)

Alla metodreferenser kan ersättas med ett lambda-uttryck. Det omvända gäller däremot inte. D.v.s. det är inte möjligt att ersätta alla lambda-uttryck med en metodreferens. I exemplet nedan visar vi hur metodreferensen i exemplet ovan kan ersättas med ett lambda-uttryck. För att visa att de båda motsvarar varandra tilldelar vi dem till det funktionsgränssnittet Function:

```
Function<String, Integer> test1 = String::length;
Function<String, Integer> test2 = s -> s.length();
String s = "Lamdas are fun!";

System.out.println(test1.apply(s)); // 15
System.out.println(test2.apply(s)); // 15
```

När vi anropar metoden apply på de båda Function-objekten kommer de båda att returnera 15.

I fallet med String::length heter den abstrakta metoden apply (i funktionsgränssnittet Function). Parametern s i metoden apply är av typen String och på denna anropas då metoden length (som är den metod som pekas ut av metodreferensen).

I avsnittet ”Ytterligare exempel på lambda-uttryck” använde vi följande Comparator för att sortera en lista av strängar:

```
Collections.sort(names, (a, b) -> b.compareTo(a));
```

Denna Comparator går att ersätta med följande metodreferens:

Comparator<String> cmp = String::compareTo;

Eftersom typargumentet till funktionsgränssnittet Comparator är en String innebär det att den abstrakta metoden compare förväntar sig två argument också av typen String. Det första argumentet används som det objekt på vilken metoden compareTo anropas. Det andra argumentet skickas vidare som argument till metoden compareTo.

```
int result = cmp.compare(left, right);
                         |     |
  /----------------------/     |
  |              /--------------/
  |              |
left.compareTo(right);
```

Anledningen till att det blir så här är för att metodreferensen är av typen ”En instansmetod på godtyckligt objekt med en specificerad typ” (Klassnamn::metodnamn). Klassnamn::metodnamn kan nämligen betyda två olika saker beroende på om metodnamn är en statisk metod eller en instansmetod.

1. Om metodnamn är en statisk metod då är Klassnamn::metodnamn ett lambda-uttryck som skickar vidare alla sina argument till den metoden:

   (a, b) -> Klassnamn.metodnamn(a, b);

2. Om metodnamn är en instansmetod då är Klassnamn::metodnamn ett lambda-uttryck som använder det första argumentet som instansen (objektet) på vilken metoden anropas:

   (a, b) -> a.metodnamn(b);

   Här är a av typen Klassnamn.

Konstruktorreferenser

Det sista vi ska titta på lektionen är hur metodreferenser används för att skapa nya objekt. Vi kan skapa en referens till en existerande konstruktor, en så kallad konstruktorreferens, genom att använda dess namn och nyckelordet new på följande sätt:

```
Klassnamn::new
```

Det fungerar på liknande sätt som med referenser till statiska metoder. Som exempel ska vi titta på

klassen User och dess parameterlösa konstruktor. Vi kan då göra följande:

```
Supplier<User> f1 = User::new;
User u1 = f1.get();
```

Eftersom en metodreferens är ett lambda-uttryck måste vi tilldela den till ett funktionsgränssnitt. Ett lämpligt sådant är Supplier som vi tittat på tidigare. Via dessa metod get får vi sen ett objekt av User skapad av den parameterlösa konstruktorn. Konstruktorreferensen ovan motsvarar följande lambda-uttryck:

```
Supplier<User> f1 = () -> new User();
```

Om vi har en konstruktor som har en parameter, som User(String name), kan vi använda funktionsgränssnittet Function på följande sätt:

```
Function<String, User> f2 = User::new;
User u2 = f2.apply("Robert");
```

Konstruktorreferensen User.:new refererar nu till konstruktorn User(String name). Detta eftersom det första typargumentet till funktionsgränssnittet Function är av typen String. Kom ihåg att det andra typargumentet till funktionsgränssnittet var vilket resultat metoden apply skulle returnera. Konstruktorreferensen ovan motsvarar följande lambda-uttryck:

```
Function<String, User> f2 = name -> new User(name);
```

Om vi har en konstruktor som har två parametrar, som User(String name, String role), kan vi använda funktionsgränssnittet BiFunction på följande sätt:

```
BiFunction<String, String, User> f3 = User::new;
User u3 = f3.apply("Robert", "admin");
```

BiFunction har vi inte tittat på tidigare, men det fungerar på samma sätt som Function, men med den skillnaden att metoden apply har två parametrar. Konstruktorreferensen User.:new refererar nu till konstruktorn User(String name, String role). Detta eftersom det första och det andra typargumentet till funktionsgränssnittet BiFunction är av typen String, vilket matchar konstruktorns parametrar. Det sista typargumentet till funktionsgränssnittet är vilket resultat metoden apply ska returnera. Konstruktorreferensen ovan motsvarar följande lambda-uttryck:

```
BiFunction<String, String, User> f3 = (name, role) -> new User(name, role);
```

Hur ska vi göra om konstruktorn har fler är 2 parametrar? Det finns nämligen inga fler inbyggda funktionsgränssnitt som kan användas. Detta lämnar vi åt dig att klura ut själv. Hör av dig till kursens ansvariga för rätt lösning om du inte kommer på det.