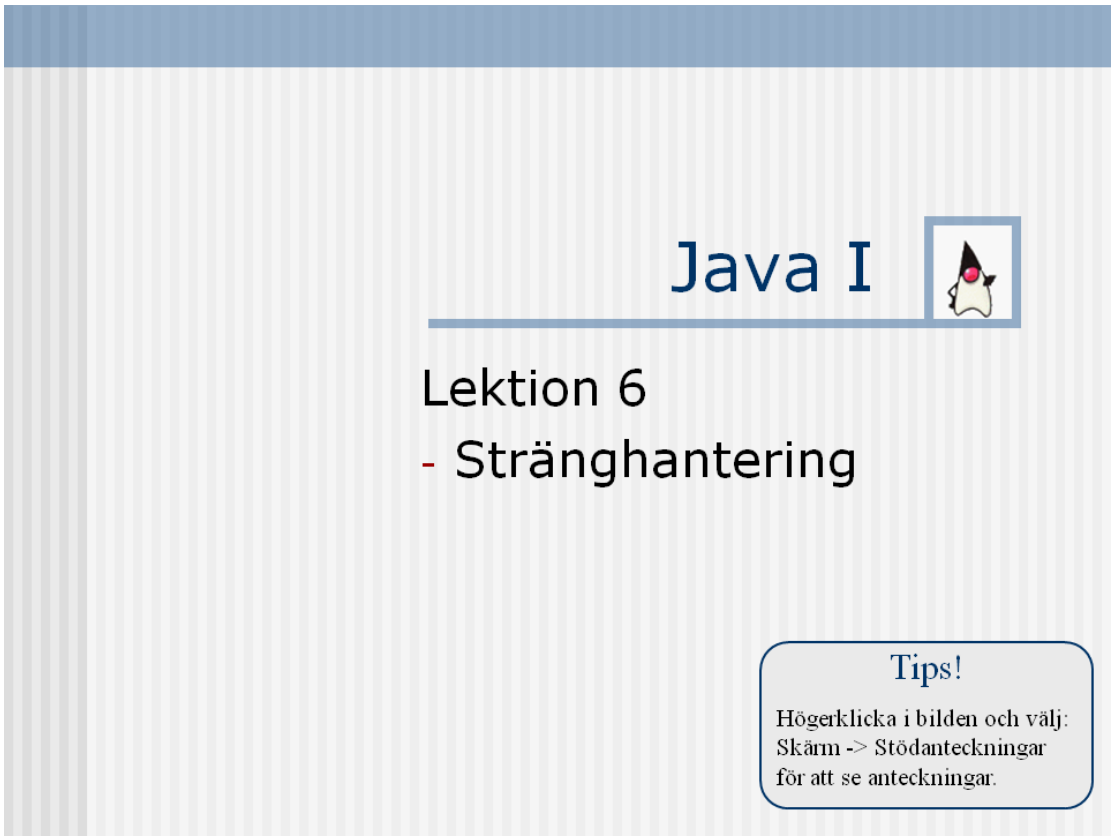
Mittuniversitetet
MID SWEDEN UNIVERSITY

Lektion 6

Datateknik GR(A), Java I, 7,5 högskolepoäng

Syfte:	Att kunna använda klasserna String, StringBuilder, StringTokenizer och dess metoder. Känna till hur vi kan formatera utskrifter med metoden format och hur inläsning av data från fil med klassen Scanner görs.
Att läsa:	Kursboken, Kapitel 2.12 (Standardklassen String) Kursboken, Kapitel 5.2 (Redigering av utskrifter) Kursboken, Kapitel 5.4 (Avkodning av indata) The Java™ Tutorial, Strings https://docs.oracle.com/javase/tutorial/java/data/strings.html The Java™ Tutorial, Scanning and Formatting https://docs.oracle.com/javase/tutorial/essential/io/scanfor.html



Vi har i tidigare lektioner använt oss av klassen `String` för att hantera textsträngar i våra applikationer. Vi ska i denna lektion titta närmare på klassen `String` och hur vi på olika sätt kan manipulera en sträng. Vi ska även titta på andra klasser för stränghantering. Dessa är `StringTokenizer`, `StringBuilder` och även `Scanner`.

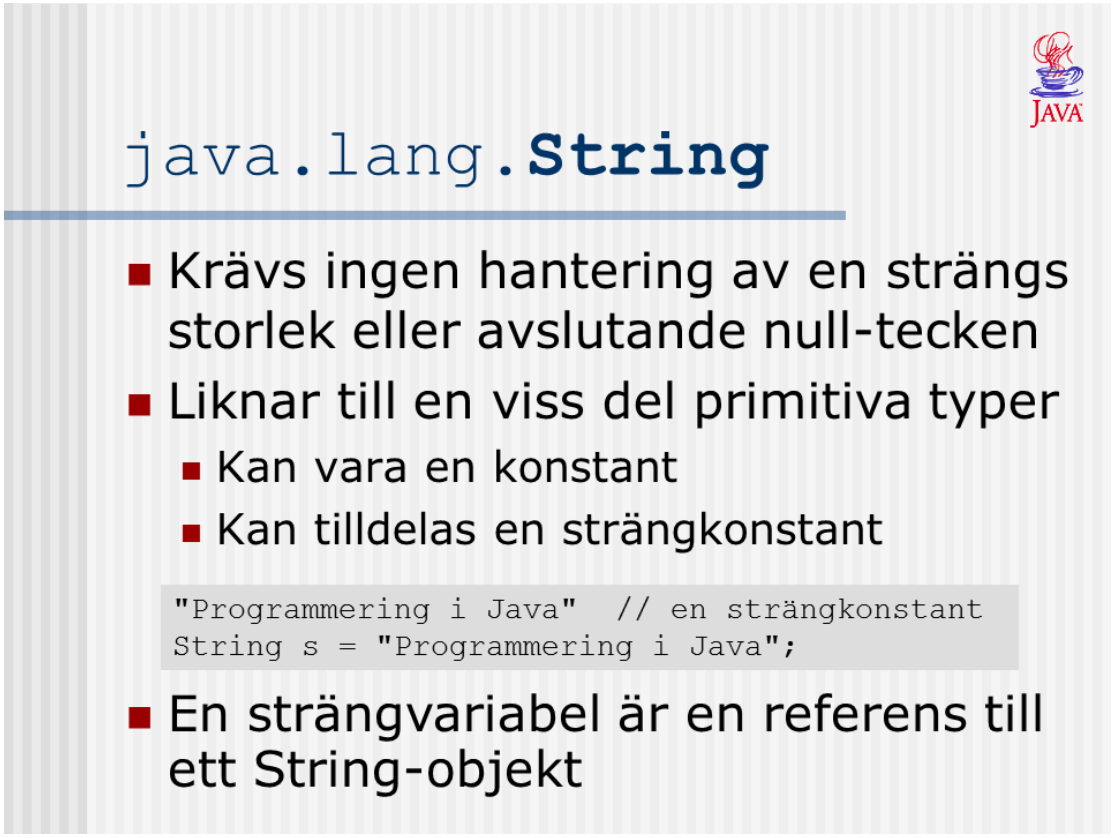
Rent generellt kan en sträng sägas bestå av ett antal tecken, som kan vara bokstäver, siffror, utropstecken m.m. Olika programmeringsspråk har olika sätt att implementera en sträng, men i Java hanteras strängar som objekt av klassen `String`.

`String`-klassen finns i paketet `java.lang` vilket innebär att vi inte behöver importera något för att kunna skapa och använda strängar (kom ihåg att alla klasser i paketet `java.lang` alltid finns tillgängliga). Man kan därför se `String`-klassen som en integrerad del i språket (på samma sätt som primitiva typer, kontrollsatser etc.) och som jag nämnt tidigare så liknar klassen `String` väldigt mycket de primitiva typerna.

En speciell egenskap med `String`-objekt är att när vi väl har skapat ett objekt av klassen `String` så kan innehållet i strängen inte förändras. D.v.s. vi kan inte lägga till eller ta bort tecken i en sträng utan att först skapa en ny sträng.

I och med att en sträng representeras av en klass så har den både instansvariabler och medlemsmetoder. Med metoderna kan vi t.ex. undersöka och söka efter innehåll i strängen, samt jämföra strängar med varandra. Som bilden ovan visar så representeras en sträng internt av en s.k. teckenarray (en array av `char`) samt en räknare som håller reda på hur många tecken som strängen innehåller. Arrayer behandlas i en senare lektion.

Vi kommer att i denna lektion se på olika sätt att använda de mest förekommande metoderna i klassen `String`.

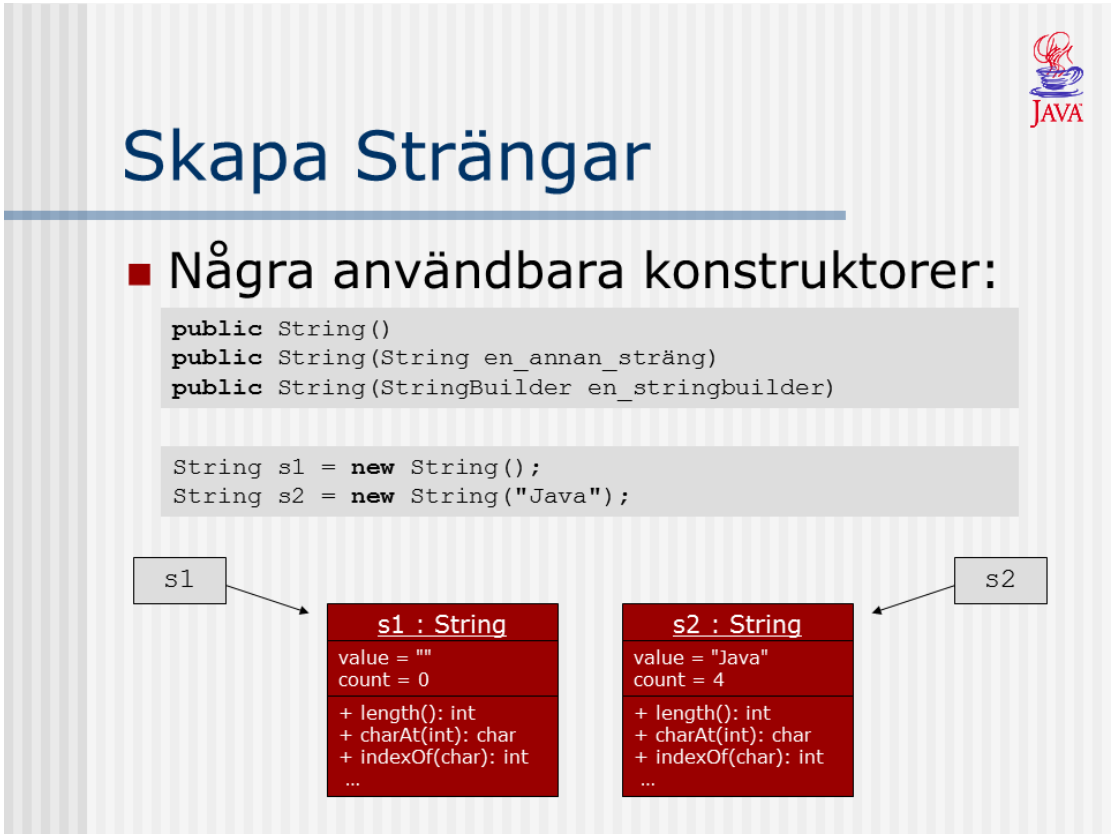


I och med att en sträng i Java endast hanteras av en speciell klass behöver vi inte ta hänsyn till att hålla reda på strängens storlek eller att avsluta den med `null`-tecken (som man t.ex. måste göra i C++ när vi håller på med tecken-vektorer).

Som redan nämnts många gånger tidigare så liknar `String` till en viss del de primitiva typerna så till vida att en sträng kan vara en konstant (egentligen en strängkonstant). Med det menas att vi i våra program kan skapa/använda en sträng genom att omsluta de tecken som tillhör strängen med citationstecken (`""`).

En strängkonstat kan t.ex. vara `"Java"` eller `"Hej hopp i lingonskogen!"`. Till ett sträng-objekt kan vi tilldela en strängkonstant på samma sätt som vi kan tilldela ett heltal till en variabel av typen `int` (`int a = 44`) eller ett falskt värde till en `boolean` (`boolean f = false`). Vi kan alltså skapa en sträng genom att tilldela den en strängkonstant.

I och med att strängar är objekt av klassen `String` så är en strängvariabel en s.k. objektreferens, det vill säga en referens till ett sträng-objekt. Minns att en variabel av en primitiv typ innehåller själva värdet (t.ex. `int i = 10` här innehåller variabeln i värdet 10), men att en variabel av ett objekt innehåller en referens till objektet och inte själva objektets innehåll (`Person p = new Person("Kalle");` här innehåller p en referens till ett `Person`-objekt).



Förutom att skapa en sträng genom att tilldela den en strängkonstant så kan man också skapa strängar som man skapar vilket annat objekt som helst: nämligen genom att använda `new`-operatorn och en konstruktor.

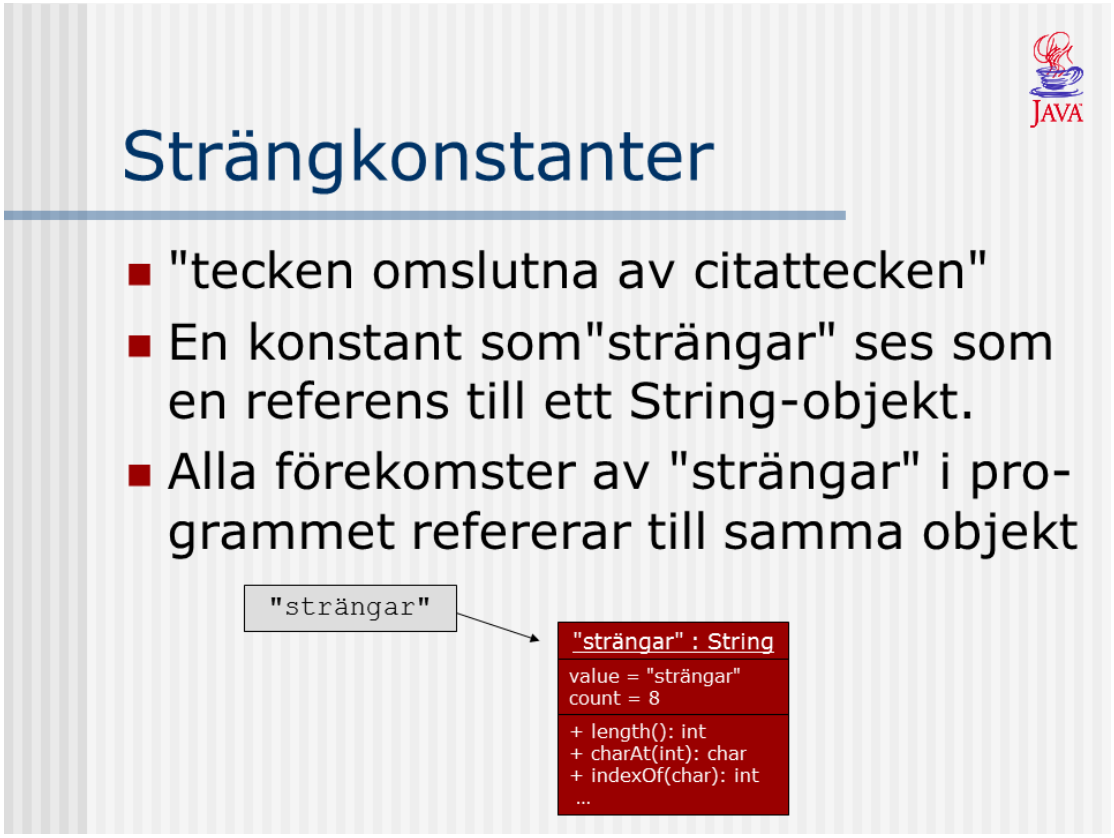
`String`-klassen innehåller flera olika konstruktorer som tillåter att man anger det initiala värdet på strängen, från ett antal olika källor. Det finns konstruktor som skapar en ny "tom" sträng med längden noll och värdet "" (dvs. en tom sträng). Det finns en annan konstruktor som tar som parameter en annan sträng, dvs. man kan skapa en sträng från en annan redan existerande sträng.

Det går att skapa en sträng utifrån ett objekt av klassen `StringBuilder` (som vi går igenom längre fram i lektionen) samt från en array av `char` (en tecken-array).

När vi skapar ett nytt sträng-objekt genom att använda den "tomma" konstruktorn (som i fallet med `s1` ovan) kommer det i Java att skapas ett nytt sträng-objekt och `s1` kommer att vara en referens till detta objekt. Objektet kommer att ha värdet "" och ha längden 0.

Skapar vi ett nytt sträng-objekt genom att använde den konstruktor som tar en sträng som parameter skapas ett helt nytt objekt (skiljt från det som skickades som argument). I fallet ovan (`s2`) så skapar vi ett nytt sträng-objekt av strängkonstanten `"Java"`. `s2` kommer nu att innehålla en referens till ett sträng-objekt som har värdet Java och längden 4.

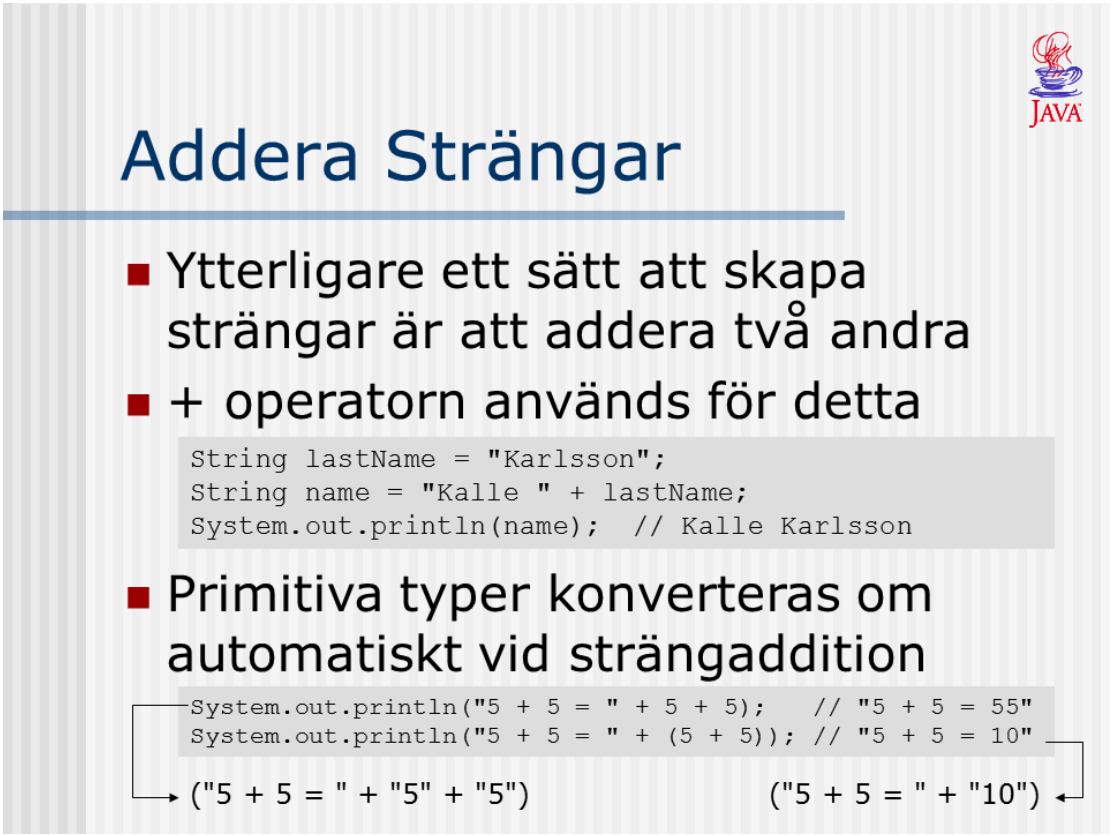
På båda dessa objekt (`s1` och `s2`) kan vi nu anropa någon av medlemsmetoderna för att t.ex. ta reda på strängens längd eller ta reda på en viss bokstav på en viss plats i strängen.



Som nämnts tidigare så är en strängkonstant (strängliteral) i Java ett antal tecken (noll eller fler) omslutna av dubbelfnuttar (""). När Java stöter på en ny strängkonstant i programmet så skapas ett nytt objekt för denna strängkonstant.

Om t.ex. vårt program innehåller strängkonstanten `"strängar"`, skulle Java skapa ett objekt för denna och sen använda själva strängkonstanten som en referens till objektet (se bild ovan).

Det betyder att strängkonstanten `"strängar"` är en referens till ett objekt vars innehåll är strängar och har längden 8. Överallt i vårt program där vi sen använder strängkonstanten `"strängar"` igen, så är det till detta objekt Java kommer att referera (det skapas inga fler nya objekt med värdet strängar med längden 8).



Ett annat sätt att bygga strängar på i Java är att addera två andra strängar. För att göra detta används additions-operatorn, som i det här fallet är överlagrad. Överlagring innebar ju att samma sak kunde ha olika betydelse. Överlagrar vi två metoder har vi två metoder med samma namn, fast de utför olika saker. På samma sätt gäller det vid operatorer. Operatoröverlagring innebär i det här fallet att + har en betydelse för t.ex. heltal och en annan om + används vid strängar. Vid heltal utför operatorn + heltals addition, men med strängar utför operatorn + strängaddition.

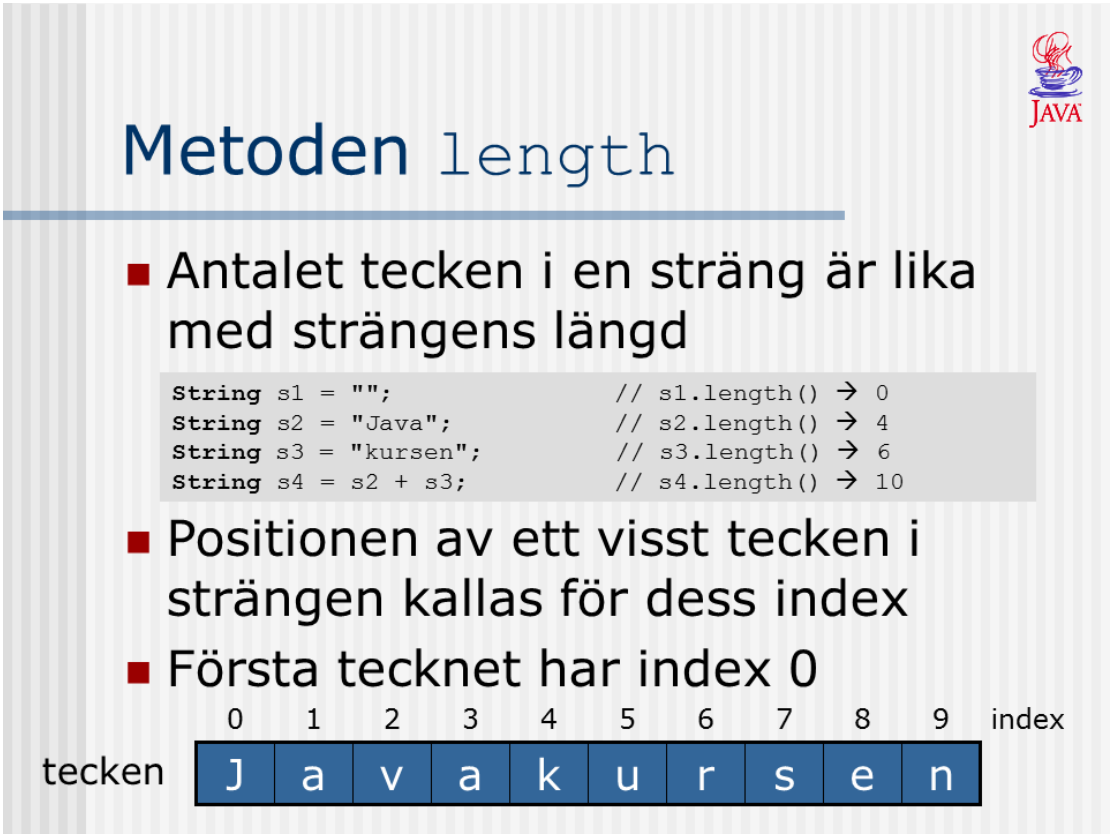
Använder vi + operatorn för att addera en sträng med en primitiv typ så kommer den primitiva typen automatiskt att konverteras till en sträng (du kommer kanske ihåg att när vi använder någon form av operator är det viktigt att de båda operanderna som operatorn jobbar med är av samma typ).

Har vi följande uttryck:

```
String s1 = "70" + 1;
```

Så kommer heltalet 1 att konverteras till en sträng innan additionen utförs. Värdet i objektet som s1 refererar till kommer efter additionen att vara "701" och inte "71".

Vill vi vara säker på att det t.ex. är en heltalsaddition som ska utföras i samband med sträng addition så kan vi använda parenteser för att visa vilken del som ska adderas först (minns att parenteser alltid hade högst prioritet och det som stod innanför parenteser utförs innan något annat). Se sista exemplet i bilden ovan.

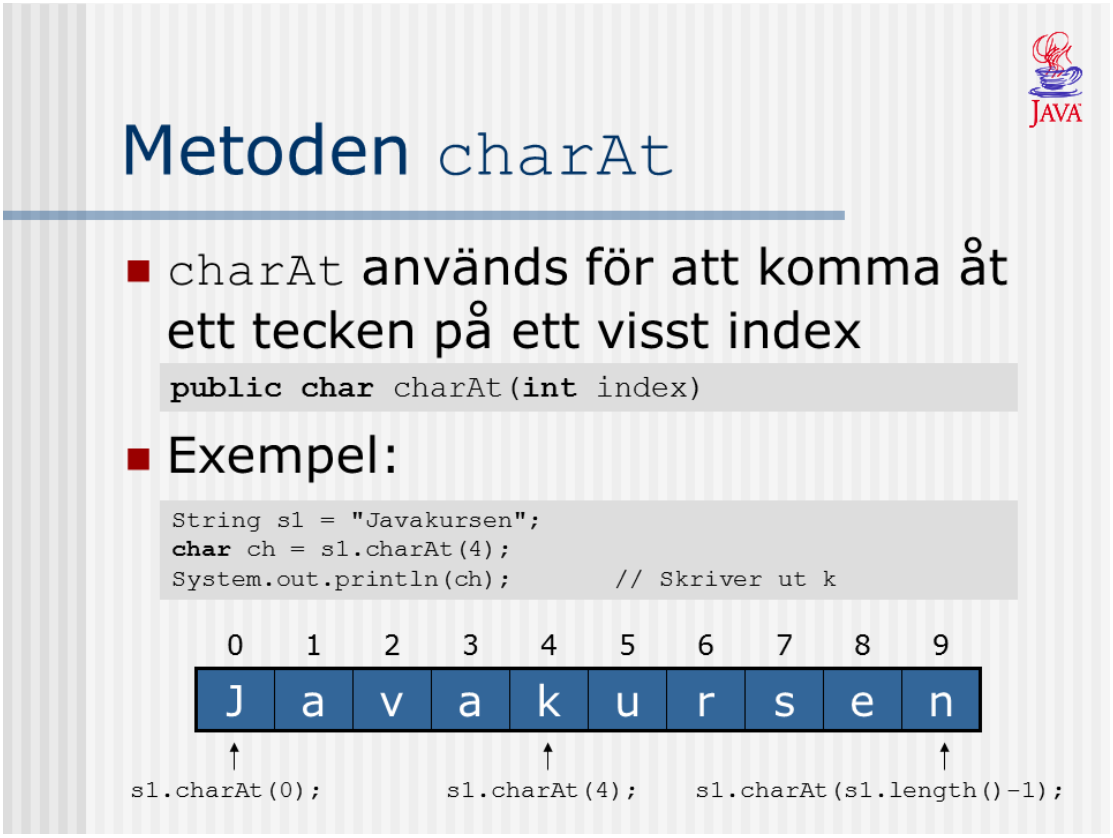


Antalet tecken i en sträng kallas strängens längd (length). Medlemsmetoden `length` returnerar ett heltal (`int`) som anger strängens längd. T.ex. kommer nedanstående deklaration ge variabeln l värdet 4.

```
String s2 = "Java";
int l = s2.length();
```

Positionen av ett speciellt tecken i en sträng kallas dess index. Med hjälp av detta index kan vi komma åt enskilda tecken i en sträng. I exemplet i bilden kommer `s4` att ha värdet "Javakursen" (eller egentligen objektet som `s4` refererar till).

Alla strängar i Java är noll-indexerade, dvs. att det första tecknet i strängen har index 0. 'J' ligger på index 0, 'a' på index 1 osv. fram till 'n' som ligger på det sista indexet 9. Med andra ord så är det högsta index i en sträng lika med längden på strängen minus 1 (`length() - 1`).



Metoden `charAt` är en medlemsmetod som returnerar ett visst tecken i strängen. Det tecken som ska returneras är det tecken vars index man skickar som argument till metoden. Har vi följande sträng:

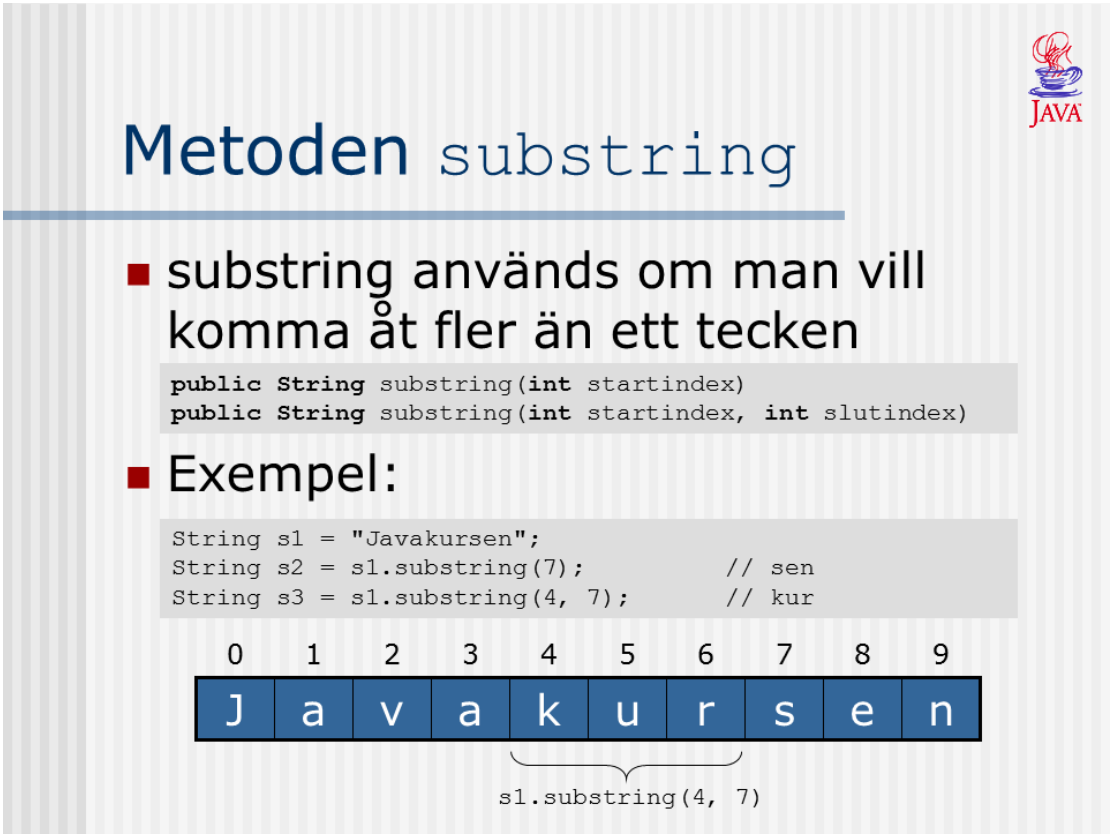
```
String s1 = "Javakursen";
```

Kommer `s1.charAt(0)` returnera första tecknet i strängen (vilket är 'J') medan `s1.charAt(s1.length()-1)` returnerar sista tecknet i strängen (vilket är 'n'). Observera att det är ett tecken (`char`) som returneras av metoden. Vill vi lagra det tecken som returneras måste vi tilldela returvärdet till en variabel av typen `char`:

```
char ch = s1.charAt(4);  // ch har värdet 'k'
```

För att komma åt sista tecknet måste man använda strängens längd minus 1 eftersom längden på strängen är 10 men sista indexet är 9. Gör vi inte det så kommer ett fel att genereras (StringIndexOutOfBoundsException).

I exemplet **StringReverser.java** har jag använt mig av metoderna `length` och `charAt` för att skriva ut en given sträng i bakvänd ordning. Ta en titt på det exemplet.



Metoden `charAt` returnerar alltså ett enda tecken från en sträng. Vill man istället komma åt mer än ett tecken från en sträng kan man använda metoden `substring`. Metoden finns i två olika varianter (överlagrad):

Den första metoden tar en parameter och returnerar en sträng som innehåller alla tecken från angivet startindex till slutet av strängen.

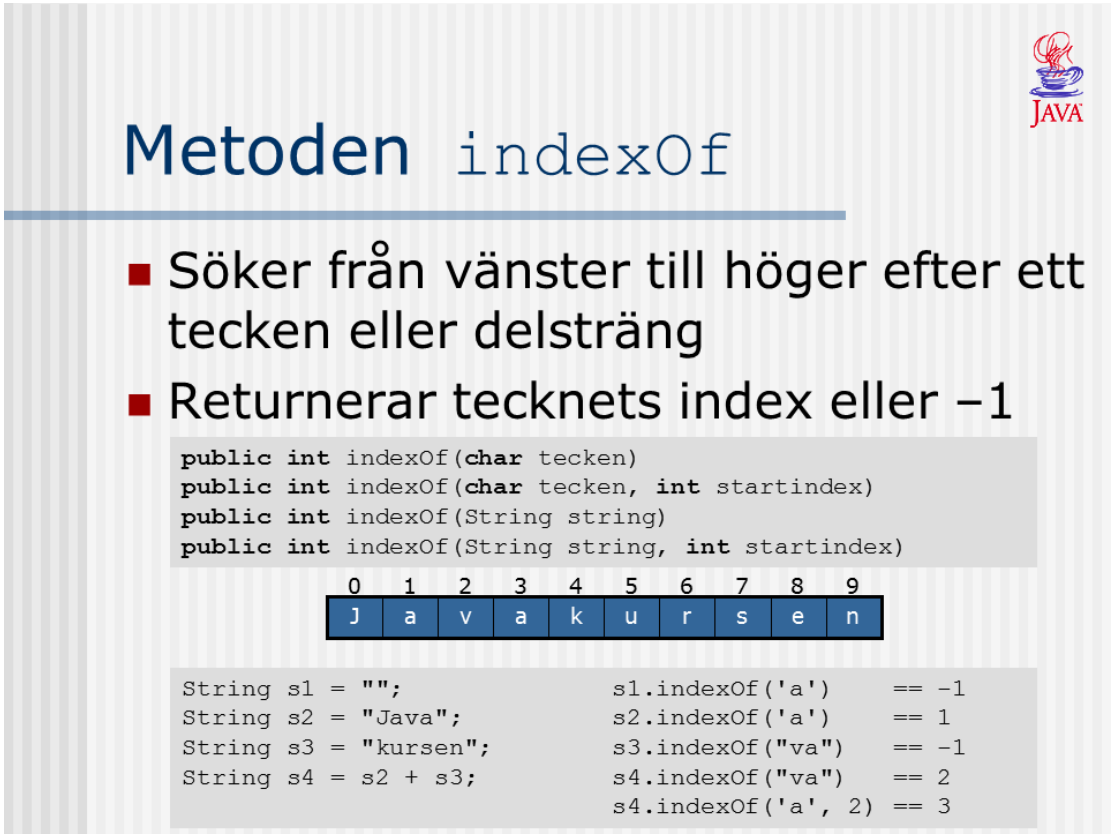
Den andra metoden tar två parametrar och returnerar en sträng som innehåller alla tecken från angivet startindex till angivet slutindex – 1. Viktigt att notera är att slutindex alltid pekar till bokstaven efter den sista bokstaven du vill ha i strängen.

Har vi följande sträng:

```
String s1 = "Javakursen";
```

Så kommer anropet: `s1.sustring(7);` att returnera en ny sträng (en delsträng av strängen s1) som innehåller alla tecken från och med index 7 till slutet av strängen. Med andra ord som kommer strängen som returneras att vara "sen".

Den andra metoden tog två parametrar och gör vi följande anrop:
`s1.substring(4, 7)` kommer en ny sträng returneras som innehåller de tecken i `s1` som börjar från och med index 4 till och med index 6 (slutindex -1). Dvs "kur" kommer att returneras. Notera att det andra argumentet (slutIndex) ska peka ett steg bortanför det sista tecknet som ska returneras.



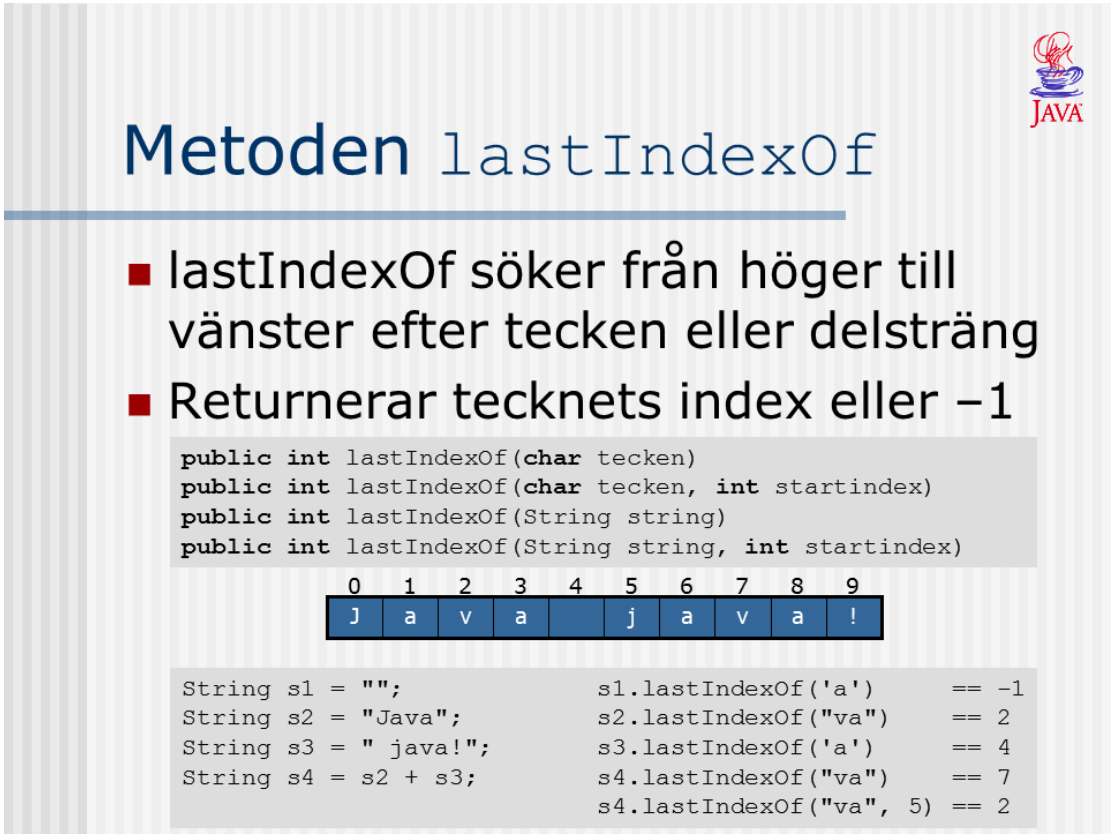
Ibland kan det vara användbart att kunna söka i en sträng för att se om ett visst tecken eller sträng förekommer i aktuell sträng. Till detta finns det två medlemsmetoder i String, metoderna `indexOf` och `lastIndexOf`. Dessa metoder är överlagrade och finns i ett antal olika varianter. Metoden som bara tar en parameter söker från början av strängen medan man i den metod som tar två parametrar kan ange var i strängen sökningen ska påbörjas.

Den första metoden, `indexof`, söker från vänster till höger i en sträng efter förekomsten av ett visst tecken eller delsträng (dvs sökningen börjar från index 0). Finns det tecken som söks returneras dess index (som ett heltal) och finns inte tecknet returneras –1. Är det en sträng som eftersöks så är det första index där denna sträng finns som returneras.

Har vi en sträng:
```
String s4 = "Javakursen";
```

kommer anropet: `s4.indexOf('p');` att söka från vänster till höger i `s4` efter tecknet 'p'. Eftersom detta tecken inte finns i strängen kommer metodanropet att returnera –1. Om vi däremot gör anropet: `s4.indexOf('a');` kommer metoden att returnera 1 eftersom det första a finns på index 1 i strängen. Anropet `s4.indexOf('a', 2)` kommer också att söka efter tecknet ’a’ men med början på index 2 i `s4`. Därför kommer värdet 3 att returneras eftersom detta är första a efter index 2.

Vi kan även som sagt söka efter delsträngar i en sträng med `indexOf`. Gör vi anropet `s4.indexOf("kurs");` kommer 4 att returneras. Det är det index första bokstaven i delsträngen finns på som returneras.



Metoden `indexOf` sökerfrån vänster till höger inom en sträng efter antingen ett tecken eller en substring. Det finns också en metod som söker från höger till vänster. Denna metod heter `lastIndexOf` och finns i samma varianter som `indexOf`.

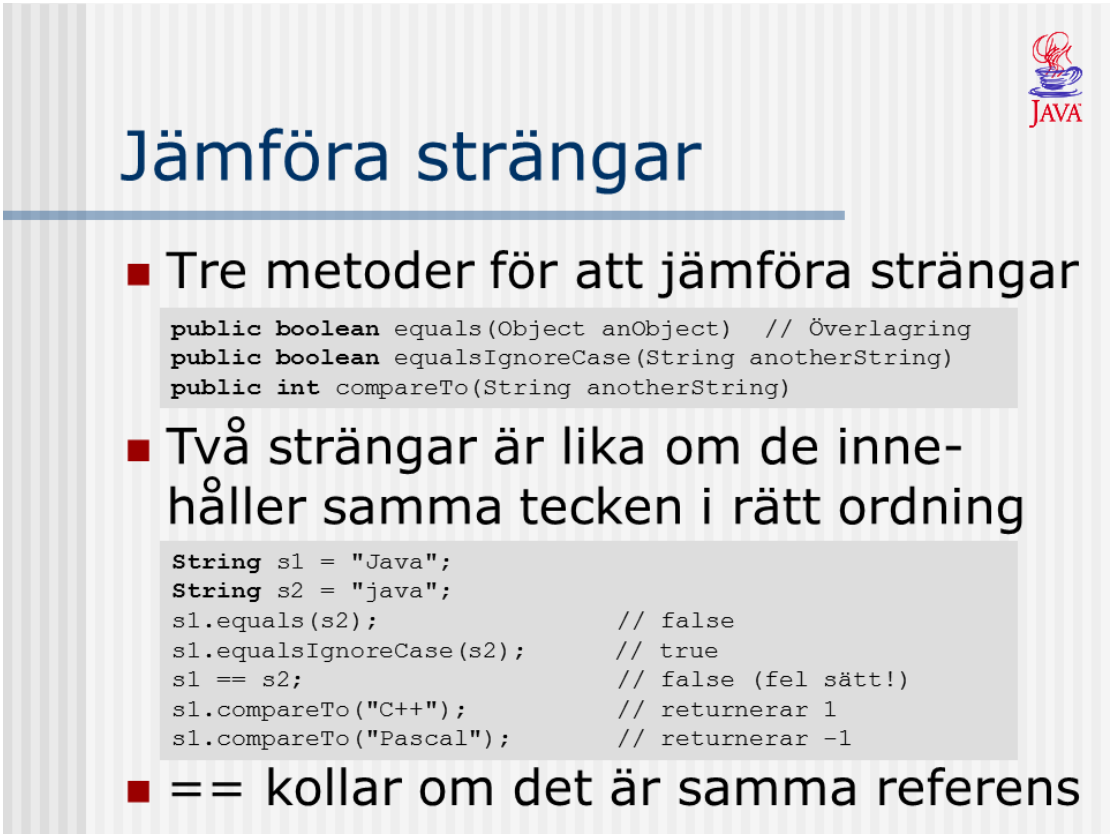
Om tecknet eller delsträngen påträffas returneras ett heltal innehållande tecknets eller delsträngens index i strängen. Hittas inget tecken eller delsträng i strängen så returneras värdet –1.

Som standard kommer metoderna med bara en parameter att börja söka vid respektive ände (vänster, höger) av strängen. Metoderna med två parametrar tillåter att man specificerar varifrån (vilket index) i strängen sökningen ska påbörjas.

Har vi en sträng:
```
String s4 = "Javakursen";
```

kommer anropet: `s4.lastIndexOf('p');` att söka från höger till vänster i s4 efter tecknet 'p'. Eftersom detta tecken inte finns i strängen kommer metodanropet att returnera –1. Om vi däremot gör anropet: `s4.lastIndexOf('a');` kommer metoden att returnera 3 eftersom första a från höger finns på index 3 i strängen. Anropet `s4.lastIndexOf('a', 2)` kommer också att söka med början på index 2. Värdet 1 kommer att returneras eftersom detta är första a från höger efter index 2.

I exemplet **FileName.java** har jag använt mig av metoderna `lastIndexOf` och `substring` för att dela upp ett filnamn som användaren skriver in.



Java erbjuder tre olika metoder för att jämföra strängar med varandra. Den första som vi redan använt är:

`equals` – som kontrollerar om två strängar innehåller exakt samma tecken i exakt samma ordning. Denna metod tar hänsyn till stora och små bokstäver. Är strängarna exakt lika returneras true. Om strängarna inte är lika returneras false.

Metoden `equals` överlagrar alltså metoden i klassen `Object` som jämför om två objekt är lika. Notera att denna metod anser att "Java" och "java" inte är samma strängar eftersom ena strängen har stort J och den andra ett litet j.

En variant på metoden `equals` är:
`equalsIgnoreCase` – som fungerar på samma sätt som `equals` men den bryr sig inte om stora och små bokstäver. Här anses alltså strängarna "Java" och "java" vara samma strängar.

Metoden `compareTo` jämför två strängar för att avgör vilken som kommer först i alfabetisk ordning. Metoden returnerar en `int` som är 0 (noll) om de båda strängarna är lika. Mindre än noll returneras om s1 kommer före s2, och större än 0 returneras om s2 kommer före s1. Observera att denna metod inte tar hänsyn till de svenska tecknen å, ä och ö vid jämförelsen. Denna metod kan vara bra att använda om man vill sortera strängar i bokstavsordning.

Observera att vi inte kan använda likhetsoperatorn == för att jämföra om innehållet i två strängar är lika. Eftersom String är en klass kommer ==, som används på två objekt av `String`, att returnera sant om de båda objektreferenserna refererar till samma objekt (falskt om inte).

I exemplet **CompareTo.java** används metoden `compareTo` för att avgöra vilken av två strängar som kommer först i bokstavsordning.

Fler Metoder

- Klassen String innehåller även bland annat dessa metoder

```
boolean endsWith(String suffix)
boolean startsWith(String prefix)
String toUpperCase()
String toLowerCase()
String trim()
String replace(char oldChar, char newChar)
```

```
String s1 = "Javakursen";
boolean end = s1.endsWith("sen");        // true
boolean start = s1.startsWith("Java");   // true
s1 = s1.toUpperCase();                   // "JAVAKURSEN"
s1 = s1.toLowerCase();                   // "javakursen"
s1 = " Javakursen   ".trim();            // "Javakursen"
s1.replace('a', 'o');                    // "Jovokursen"
```

Utöver de metoder som redan tagits upp finns det ytterligare några metoder som kan användas (används dock inte lika ofta som de tidigare). På grund av att strängar är ”read-only” kommer de metoder som returnerar en sträng att returnera ett nytt strängobjekt. Om man vill använda en av dessa metoder för att konvertera en sträng måste man tilldela resultatet tillbaka till ”originalsträngen”.

`endsWith` – kontrollerar om slutet på strängen är det samma som strängen som skickas som argument
`startsWith` – kollar om strängen börjar med samma sträng som skickas som argument
`toUpperCase` – returnerar en ny sträng där strängen konverterats till bara STORA bokstäver
`toLowerCase` – returnerar en ny sträng men med enbart små bokstäver
`trim` – tar bort eventuella ”white space” i början och slutet av strängen (dvs mellanslag m.m.)
`replace` – ersätter alla tecken (första argumentet) i strängen med det tecken som anges i andra argumentet

Ta en titt på exemplet i bilden ovan för att se hur metoderna kan användas som. Ta även en titt i Javas API för klassen `String` och gå igenom de metoder som finns. Kan du komma på fler metoder som kan vara användbara?

I `String` finns en metod med vilken vi har möjlighet att formatera innehållet i strängen på olika sätt. Metoden heter `format` och tar två eller fler argument. Den första parametern innehåller en s.k. formatsträng som innehåller både den text som ska skrivas ut och olika formatspecificerare som anger hur innehållet ska formateras. De övriga parametrarna (en eller flera) innehåller de värden som ska formateras. För varje formatspecificerare ska det normalt finnas en övrig parameter. Metoden format kan även användas i en `System.out` för att formatera en utskrift till kommandofönstret.

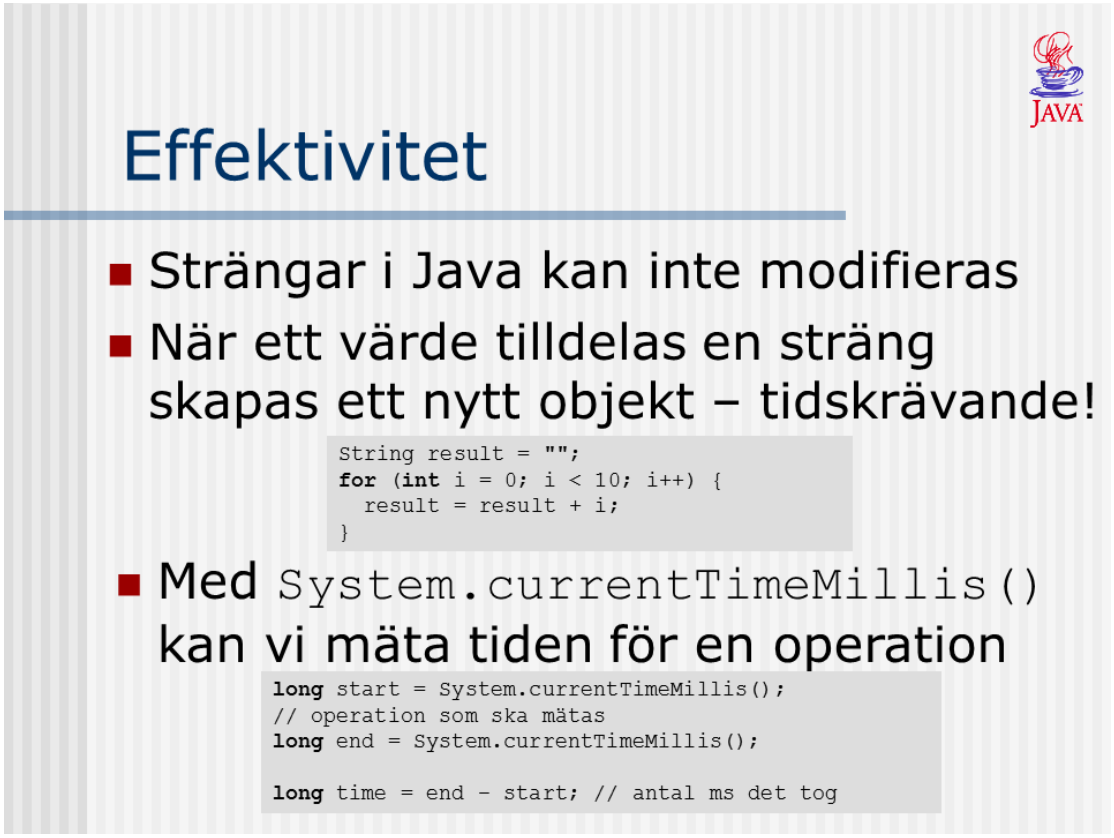
I formatsträngen börjar varje ny formatspecificerare med tecknet % därefter följer eventuellt en flagga som t.ex. kan ange om vänsterjustering ska användas, om talen ska grupperas, om talets tecken alltid ska visas (+ eller -) om utfyllnad ska ske med blanksteg eller siffran 0 etc.

Efter flaggan kan man ange en minsta bredd (minsta antal tecken som alltid ska skrivas ut). Om ett tal som ska formateras endast innehåller två positioner (t.ex. talet 10) men att man vill att bredden på utskriften alltid ska ta upp 4 positioner, kan man som bredd skriva 4. Beroende på vilken flagga som används sker eventuell utfyllnad med blanksteg eller siffran 0.

Det finns ett flertal olika typer man kan sätta. Mest vanligt är d för heltal och f för decimaltal. Används f som typ (decimaltal) anges bredden i formen total_bredd.antal_decimaler (t.ex. 5.2). Första delen anger att det totalt ska vara minst 5 positioner (inklusive decimalpunkten) i utskriften och andra delen säger att det max får vara två decimaler.

I exemplet i bilden formateras finns tre formatspecificerare i formatsträngen. %02d anger att det är heltal som ska formateras och att utskriften minst ska uppta två positioner. Eventuell utfyllnad ska göras med siffran 0. De tre övriga argumenten är de heltal som ska formateras (formatsträngen innehåller tre formatspecificerare).

Jag hänvisar till kursboken kapitel 5.2 för mer läsning om metoden format.



Många gånger är klassen `String` både bekväm och enkel att använda, men ett problem med klassen `String` är att den inte är särskilt effektiv. Problemet är att en `String` i Java är ett ”read-only” objekt. Det betyder att när ett objekt väl har instansierats kan innehållet i strängen inte förändras. Man kan inte sätta in nya tecken eller ta bort existerande tecken från strängen (det är i och för sig inte ofta vi behöver göra det, så problemet är kanske inte så stort).

Varje gång ett nytt värde tilldelas en sträng måste Java skapa ett nytt sträng-objektet och man kan därefter tilldela den nya referensen (till den nya strängen) till den gamla referensen. Ex:

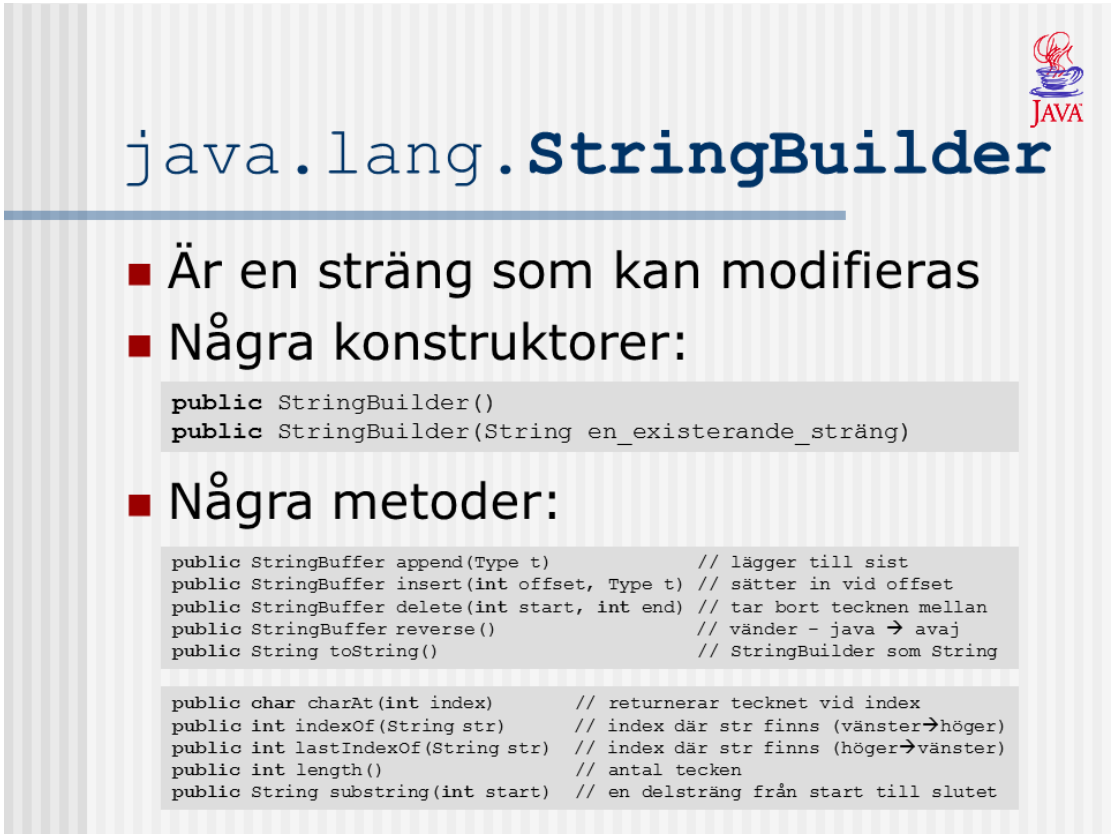
```
String s1 = "Java";
s1 += "kursen";
```

Vi börjar med att skapa en sträng `s1` och tilldelar den värdet `"Java"`. Därefter evaluerars uttrycket på högersida om `+=` och Java skapar först ett nytt sträng-objekt av strängkonstanten `"kursen"`. Sen skapas ytterligare ett nytt objekt eftersom vi adderar `s1` med strängkonstanten `"kursen"`. Referensen till detta nya objekt (s1 + "kursen") tilldelar vi tillbaka till variabeln `s1`. Detta resulterar i ett objekt (objektet som s1 refererade till förut) inte längre har några referenser utan ligger och skräpar i minnet. Tack vara javas automatiska skräpinsamlare städas referensen bort om minnet skulle behövas.

Skulle vi göra ovanstående i en loop betyder det att flera nya objekt skapas och vi kommer att få mängder med objekt som behöver skräpsamlas senare. I Java är skapande av objekt en relativt tids- och minneskrävande operation, vilket gör att sådana loopar är lite slösaktig med resurser. Men i och med Javas automatiska skräpsamling ”tar tillbaka” minnet efter en tid så är ingen större skada skedd annat än att det tar lite längre tid.

Med metoden `System.currentTimeMillis` kan vi mäta tiden en viss operation tar att utföra. Se exempel ovan hur vi kan mäta en operation. Metoden `currentTimeMillis` returnerar ett långt heltal (`long`) innehållandes nuvarande tid (sedan 1 januari 1970) i millisekunder. Genom att lagra tiden just innan den operation som ska utföras, och tiden när operationen är gjord, kan vi räkna hur lång tid det tog. Seden JDK 5.0 finns även metoden `nanoTime` som returnerar en mer precis tid, men bara om underliggande plattform stödjer det. Läs mer i Javas API för klassen `System`.

I exemplet **StringTest.java** mäter jag tiden det tar att skapa 100 000 strängar i en loop. I senare versioner av JDK har den vanliga strängklassen `String` blivit mycket effektivare när det kommer till strängaddition.



Om man vet att en sträng kommer att behöva modifieras efter att den är skapad bör man alltid använda klassen `StringBuilder` i stället. Denna klass ligger också i paketet `java.lang` vilket innebär att vi inte behöver importera något för att använda klassen. `StringBuilder` är i stort sett uppbyggd på samma sätt som en sträng (har ett värde och längd) men har även en kapacitet som anger hur många tecken en `StringBuilder` kan innehålla (ökas automatiskt om det behövs).

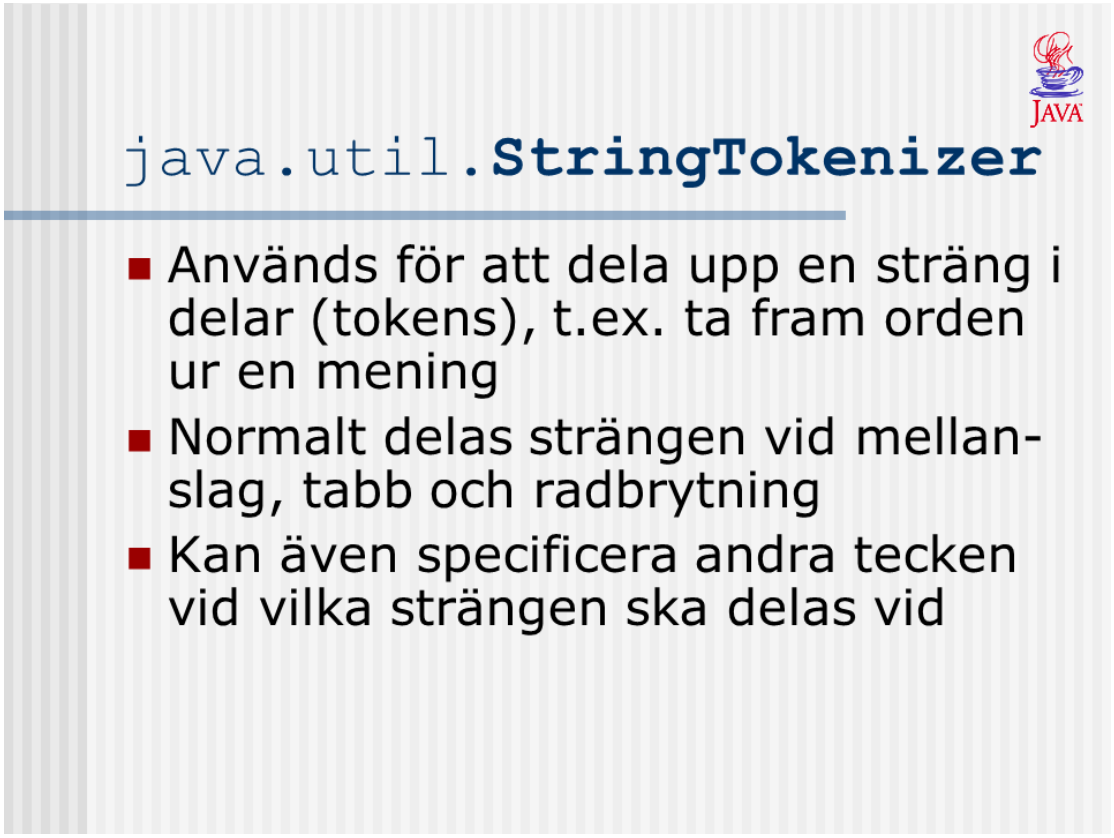
För att skapa en `StringBuilder` kan man antingen använda den tomma konstruktorn som skapar en `StringBuilder` med längden 0, innehållet `""` och kapaciteten 16 tecken. Det finns även en konstruktor som tar en sträng (`String`) som argument vilket innebär att det är lätt att skapa en `StringBuilder` från en redan existerande sträng (ett sätt att konvertera en sträng till en `StringBuilder`).

När vi har skapat en `StringBuilder` kan vi anropa metoder för att sätta in strängar, tecken, tal m.m. sist eller vid ett visst index i `StringBuilder`. Det finns även metoder för att vända innehållet i en `StringBuilder` ("java" blir "avaj") och för att ta bort delar ut `StringBuilder`. Det finns även metoder som vi känner igen från `String`, bl.a `charAt`, `length`, och `indexOf`.

Vet vi av att vi kommer att modifiera en sträng många gånger och då framför allt i en loop av något slag är det mer effektivt använda klassen `StringBuilder` i stället för `String`.

Konstruktorn `StringBuilder(String)` gör det lätt att konvertera en sträng till en `StringBuilder`. Och med hjälp av metoden `toString` är det lika lätt att konvertera en `StringBuilder` till en sträng.

I exemplet **StringBuuilderTest.java** gör vi samma typ av operation som i StringTest.java, men med en `StringBuilder` i stället. Notera den stora skillnaden i tid mellan StringTest och StringBuilderTest.



Den tredje och sista klassen som vi ska titta på som hanterar strängar är `StringTokenizer`. Med en `StringTokenizer` kan man dela upp en sträng i olika delar, s.k. tokens. Detta kan användas för att t.ex. dela upp en mening i de ord den innehåller.

När man skapar en `StringTokenizer` delas strängen som standar upp efter de s.k. whitespace (mellanslag, tabbar, nyrad etc). Men man kan även skapa en `StringTokenizer` som delar upp en sträng efter tecken som man själv anger.

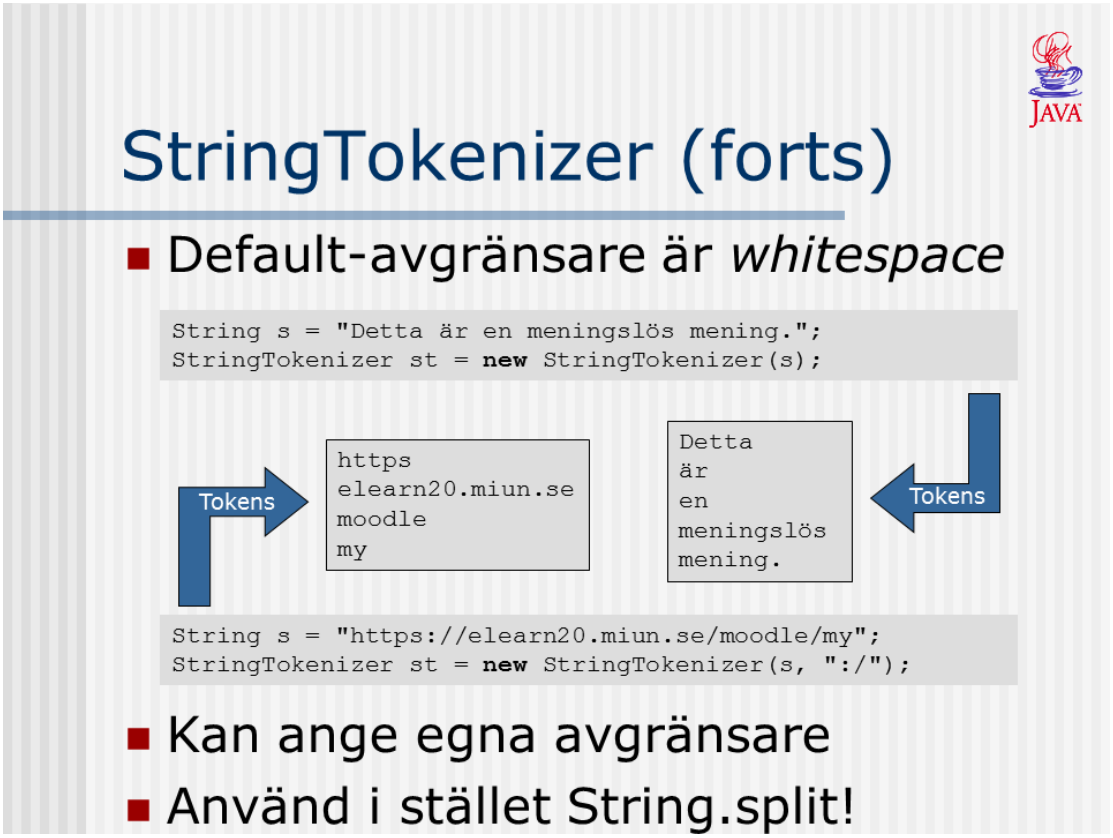
För att kunna använda klassen måste man importera paketet `java.util` där klassen `StringTokenizer` ligger.

För att skapa ett `StringTokenizer`-objekt kan man göra detta på två sätt (egentligen tre). Den första konstruktorn delar upp en sträng efter whitespaces:
`StringTokenizer(String str)`
– där str är strängen som ska delas upp.

Den andra konstruktorn delar upp strängen efter de avgränsare som skickas som andra argumentet:
`StringTokenizer(String str, String skiljetecken)`
– där skiljetecken anger vilka tecken str ska delas vid.

För att sen kontrollera t.ex. hur många tokens strängen delades upp i, eller returnera nästa token kan man anropa olika metoder i `StringTokenizer`:

`hasMoreTokens` – returnerar `true` om det finns några tokens kvar att plocka fram ur strängen.
`nextToken` – returnerar nästa token som en sträng. Vid första anropet returneras första token, vid andra anropet returneras andra token etc.
`countTokens` – räknar ut hur många tokens det finns kvar i strängen (minskar för varje gång vi anropar `nextToken`.



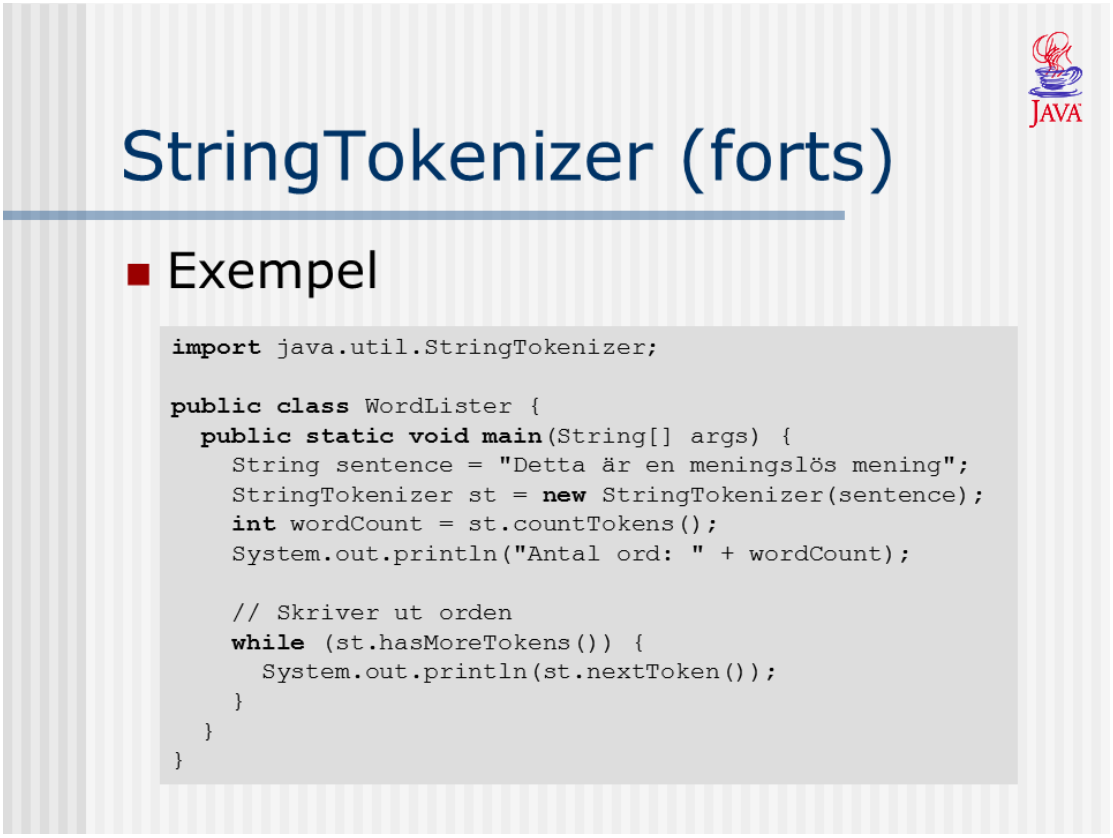
Det första exemplet i bilden ovan visar vilka tokens vi får om vi skapar en `StringTokenizer` med den konstruktor som enbart tar en sträng som argument. Skulle vi i en loop anropa metoden `nextToken` så länge som det finns fler tokens så skulle vi få utskriften:

```
Detta
är
en
meningslös
mening.
```

Observera att som standard är punkt, kommatecken, utropstecken m.m. inte en avgränsare så därför kommer dessa att ingå i de tokens som returneras.

Det andra sättet att skapa en `StringTokenizer` är att själv ange vilka avgränsar som ska användas. I exemplet har vi satt avgränsarna till tecknen **:** och **/** så att vi på ett enkelt sätt kan dela upp en Internetadress i olika delar.

Observera att klassen `StringTokenizer` är en gammal klass som endast finns kvar för bakåtkompabilitetens skull. Skriver vi ny kod rekommenderas vi att i stället använda oss av klassen `String` och dess metod `split`. Med om den metoden kommer i lektionen som behandlar arrayer. Fram till dess är det ok att du använder dig av `StringTokenizer` när du löser dina inlämningsuppgifter.



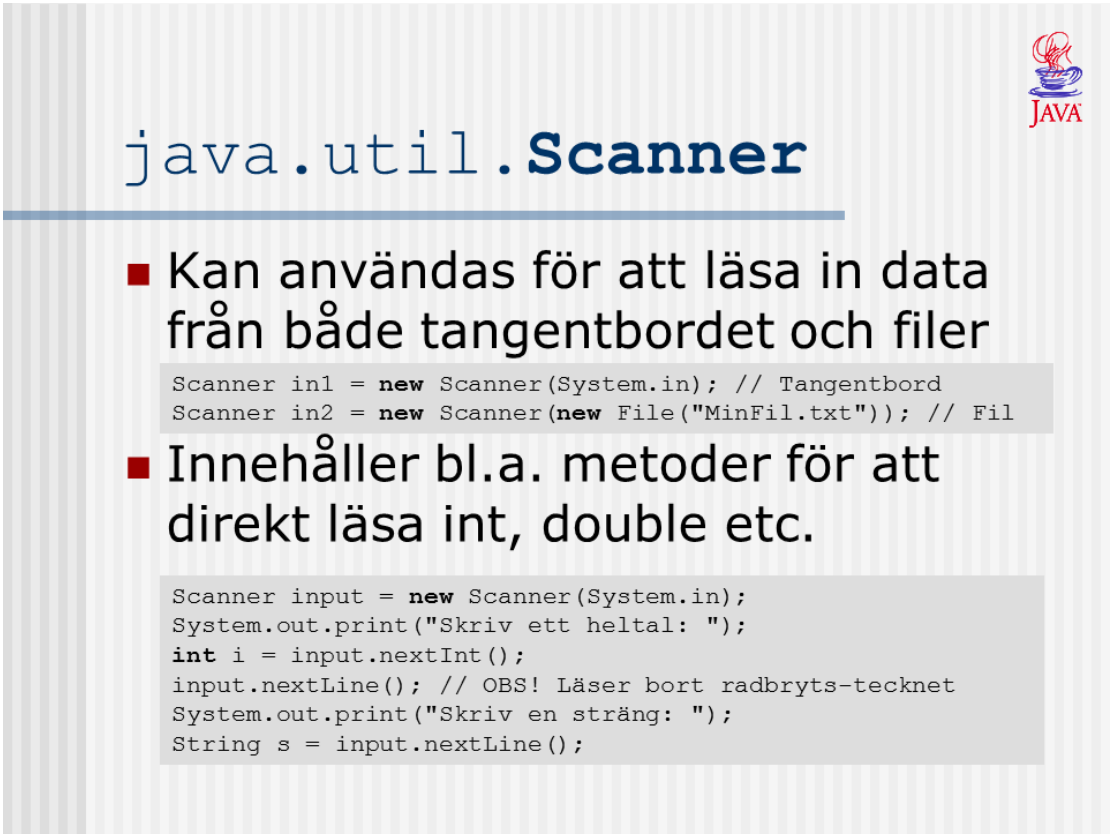
Exemplet ovan visar på hur vi kan använda `StringTokenizer` i ett program för att räkna antalet ord i en mening samt att skriva ut de enskilda orden. Detta exempel kan man utöka för att t.ex. läsa in ett text-dokument från hårddisken för att räkna hur många ord dokumentet innehåller.

Det vi gör i `main`-metoden är att skapa en sträng som innehåller texten ”Detta är en meningslös mening.”. Denna sträng använder vi sen på raden under för att skapa ett `StringTokenizer`-objekt som vi kallar för `st`. På objektet `st` anropar vi därefter metoden `countTokens` som returnerar ett heltal med antalet tokens som finns i strängen.

Med hjälp av en while-loop skriver vi sen ut alla tokens, vilket är alla ord i strängen. Metoden `hasMoreTokens` kommer att returnera true så länge som det finns fler tokens att ”plocka ut” ur st. Med andra ord kommer while-loopen att snurra så länge som det finns fler tokens.

I while-loopen anropar vi sen `nextToken` som returnerar nästa token, som vi skriver ut på skärmen. Anropet till `nextToken` innebär att antalet tokens som är kvar i st minskar med ett och när vi har anropat `nextToken` tillräckligt många gånger kommer metoden `hasMoreTokens` att returnera `false` och loopen avbryts.

I exemplet **Palindrom.java** testar vi om ett ord är ett palindrom eller inte



En annan klass som ingår i paketet `java.util` är klassen `Scanner`. Denna klass introducerades i version 1.5 vilket innebär att om man har en tidigare version av JDK än 1.5.0 kan man inte använda klassen. Med `Scanner` kan vi läsa in data på ett något enklare sätt än med `BufferedReader`. För att använda klassen `Scanner` i våra program måste vi importera paketet `java.util`. Vi måste dessutom importera paketet `java.io` om vi ska använda `Scanner` för att läsa från en fil.

I `Scanner` finns det ett flertal olika metoder med vilka vi direkt kan läsa in bl.a. heltal och decimaltal utan att första behöva konvertera med wrapper-klasserna `Integer` eller `Double`. Det finns givetvis även metod för att läsa en hel rad med tecken så som `readLine` i `BufferedReader`. Heltal kan vi läsa genom att anropa metoden `nextInt`, decimaltal genom att anropa metoden `nextDouble` och en hel textrad genom att anropa metoden `nextLine`.

Något att vara uppmärksam på är att det kan uppstå problem om vi i ett program använder `Scanner` för att först läsa ett heltal följt av en textrad. När vi trycker ner enter genererar det ett radbrytstecken. Eftersom ett tal inte kan innehålla tecken annat än siffror läses inte radbrytstecknet av `Scanner`. Därför måste du alltid göra ett extra anrop till `nextLine` när du läst ett tal och direkt efter vill läsa en sträng.

Ta nu en titt på exemplet **ScannerTest.java** där jag använder `Scanner` för att både läsa data från tangentbordet och från fil. Använder även metoden `format` för att formatera min utskrift.