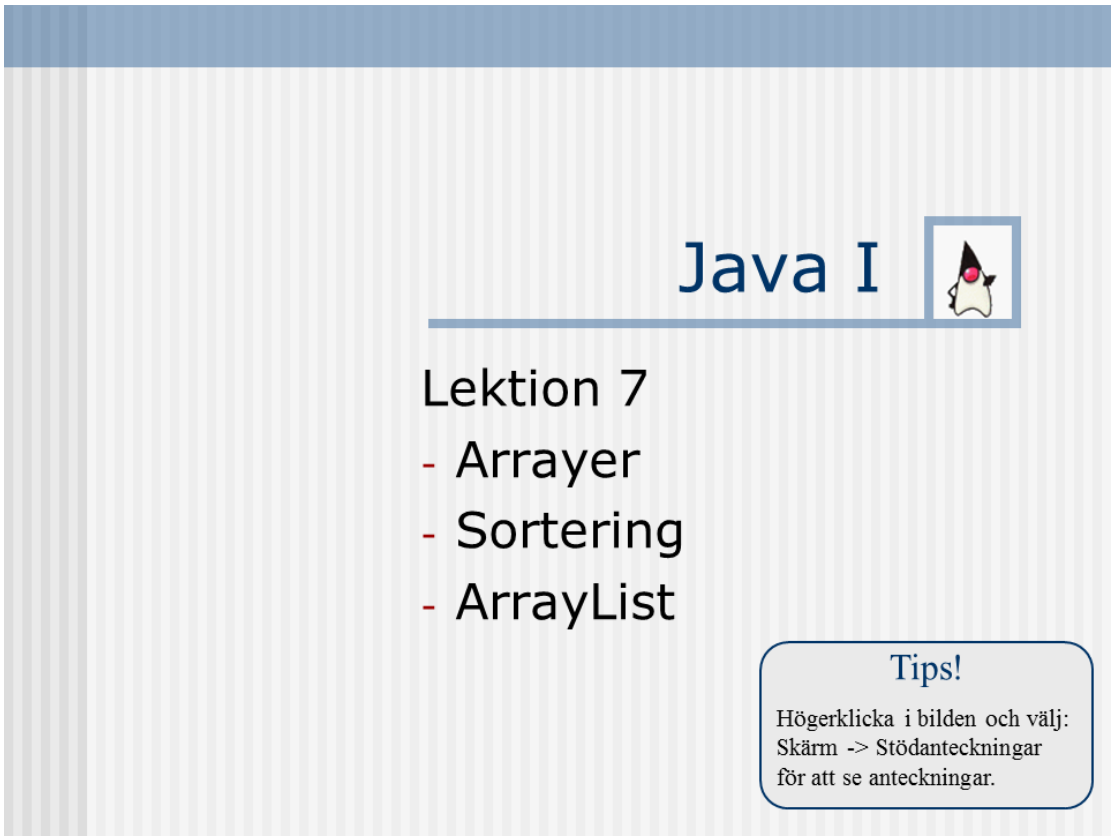
Mittuniversitetet
MID SWEDEN UNIVERSITY

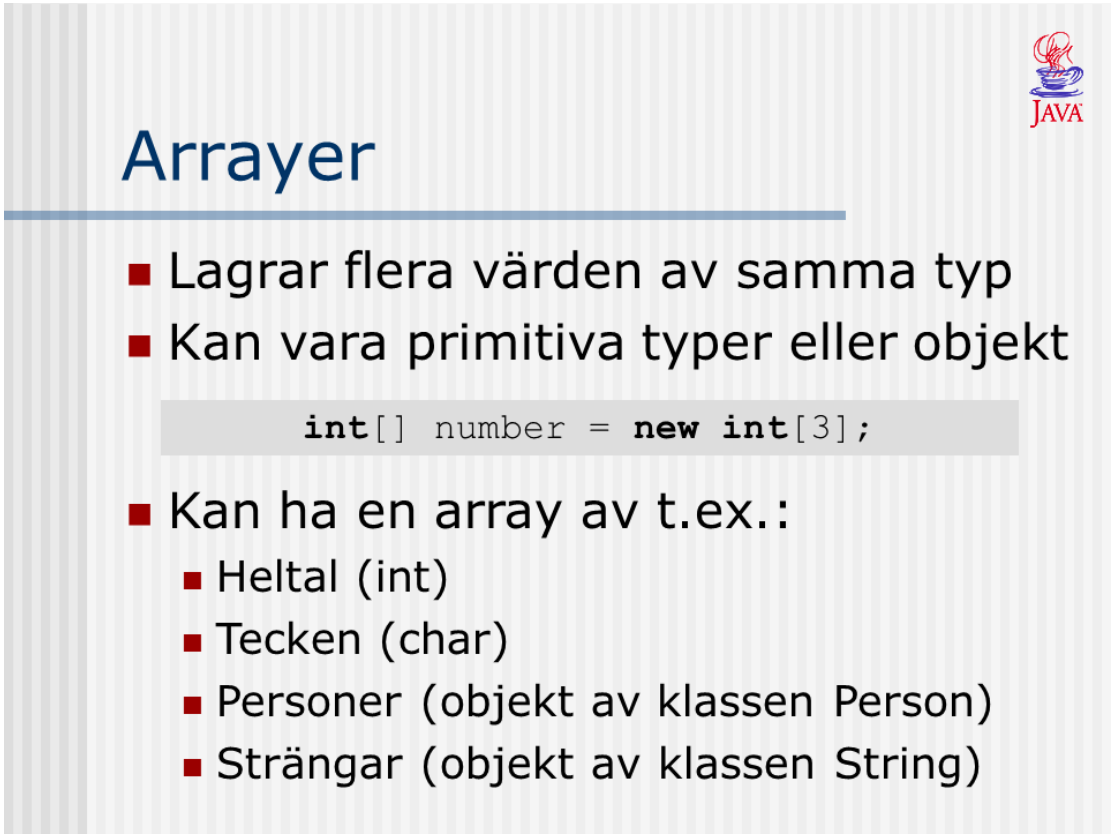
Lektion 7

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

Syfte:	Att kunna skapa och använda arrayer av både primitiva typer och egendefinierade klasser. Kunna skriva en egen enkel algoritm för sortering samt använda befintliga klasser i Javas API för att sortera. Känna till och kunna använda en ArrayList.
Att läsa:	Kursboken, Kapitel 3.8 (Arrayer med enkla komponenter) Kursboken, Kapitel 3.9 (Arrayer med referenser) Kursboken, Kapitel 9 (Mer om arrayer) The Java™ Tutorial, Arrays https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html



Hittills har vi i kursen enbart använt oss av variabler som kan lagra enstaka värden. I denna lektion ska vi titta på hur vi kan lagra flera värden i samma variabel. Vi ska även titta på hur vi på olika sätt kan sortera en mängd av värden, både primitiva typer och egendefinierade klasser.

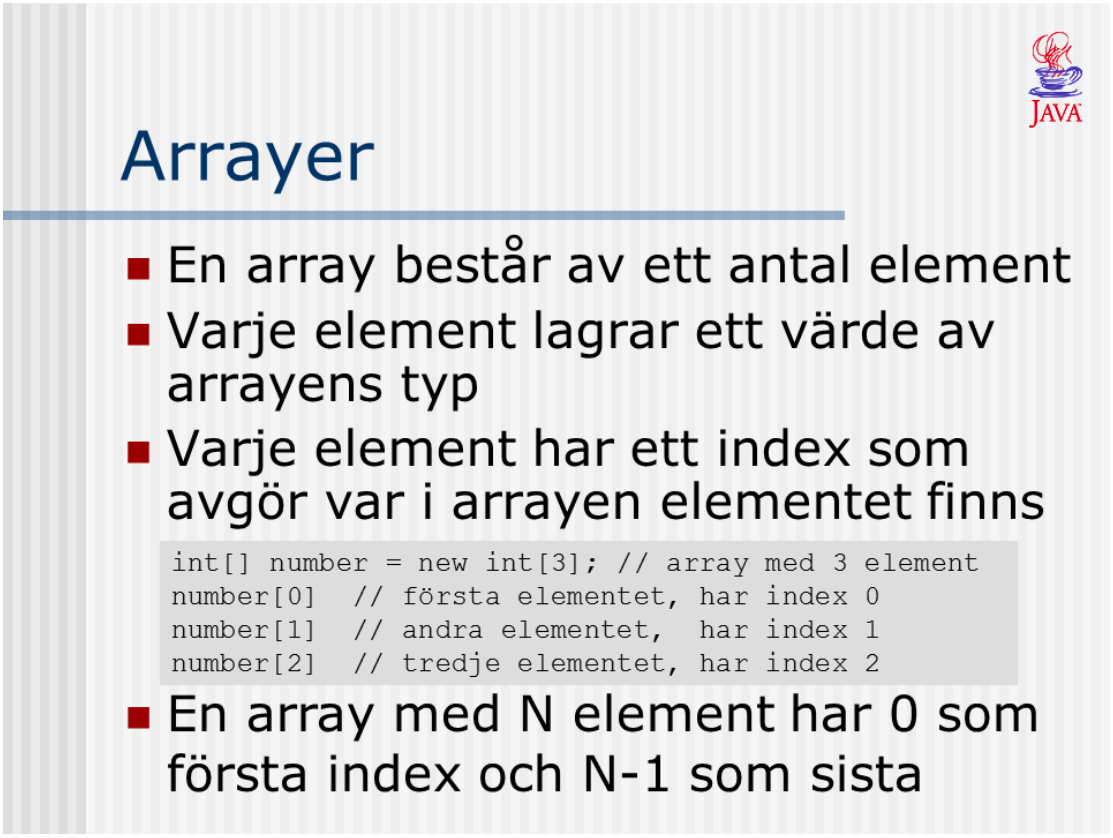


En array i kan man se som en variabel som kan lagra flera värden av samma typ. Man kan alltså se en array som en samling av variabler som finns lagrad på samma plats. I stället för att i ett program har tre separata heltalsvariabler kan man skapa ett heltals-array som kan lagra 3 heltal (se exemplet ovan hur vi skapar en sådan array). Man kan säga att variabeln number ovan är en slags behållare som rymmer tre tal av typen `int`.

Att arbeta med arrayer istället för med separata variabler är betydligt mer smidigt och effektiv, speciellt då vi hanterar stora datamängder (många tal t.ex.).

En array i Java kan bestå både av primitiva typer och av olika objekt. Vi kan med andra ord ha en array av heltal (`int`), en array av decimaltal (`double`), en array av personer (Person-objekt, en array av klassen Card (Card-objekt).

Det viktiga att komma ihåg är att en array endast kan lagra värden av samma typ. Vi kan alltså inte ha en array som innehåller både heltal och decimaltal, eller både Person-objekt och Card-objekt!

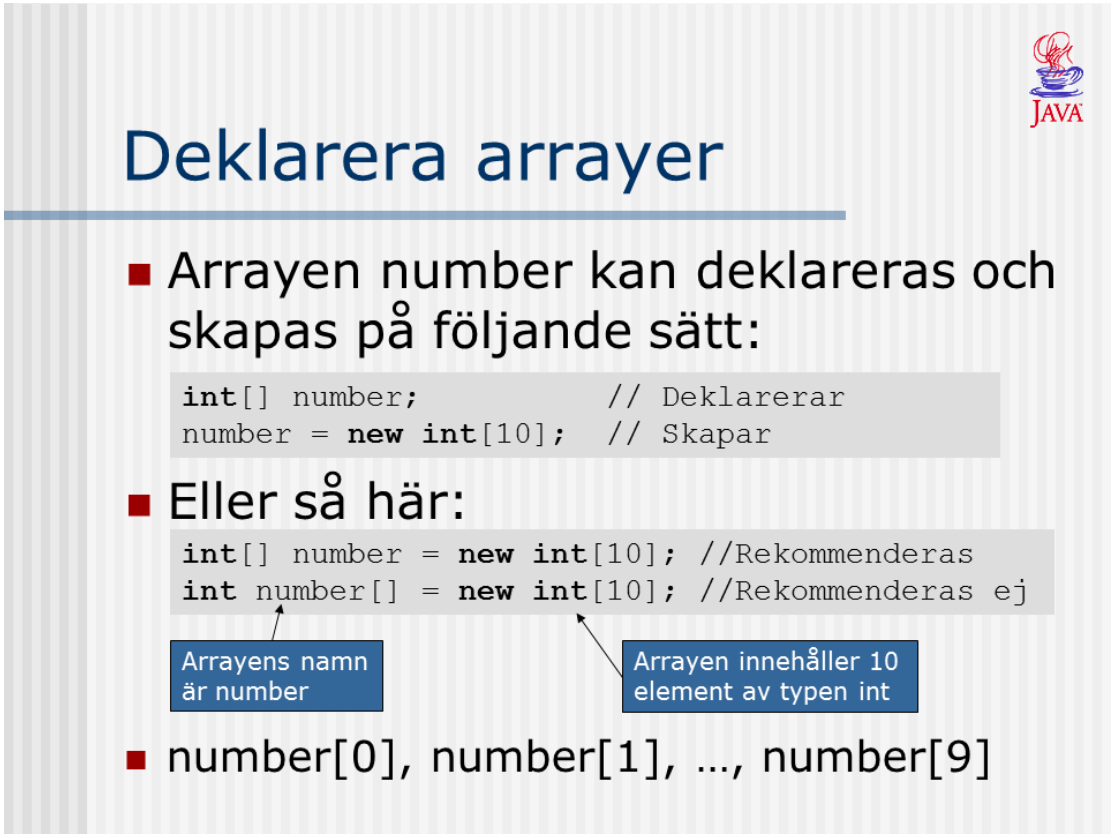


En array består alltså av ett antal värden av samma typ. Varje enskilt värde i en array kallas för arrayens element. Så arrayen innehåller ett antal element vars värde är av samma typ som arrayen.

För att komma åt ett visst värde i en array måste vi kunna avgöra var i arrayen detta värde finns. Därför har varje element i arrayen ett index. Detta index visar på vilken plats i arrayen ett element finns och används för att komma åt elementets värde.

Första elementet har alltid index 0 och därefter räknas index upp med 1 för varje nästkommande element. Så i vår heltals-array number har första elementet index 0, andra elementet index 1 och tredje (och sista) elementet har index 2. För att komma åt värdet i första elementet skriver vi då: `number[0]`.

En array av en viss storlek (med N antal element) har alltid 0 som första index och N-1 som sista index. En array med 5 element har 4 som sista index. En array med 100 element har 99 som sista index etc.



Att skapa en array görs i två steg (som kan kombineras i ett enda steg). Först måste man deklarera att man ska använda en array (på samma sätt som vi måste deklarera att vi ska använda en vanlig variabel i ett program).

När man deklarerar en array anger man arrayens typ, dvs vilken typ elementen har som ska lagras i arrayen, samt namnet på arrayen. För att skilja en vanlig variabeldeklaration mot en arraydeklaration används hakparenteser `[]` efter typen arrayen, för att visa att det är en array vi deklarerar. T.ex deklarerar vi en array av typen int och som vi kallar number med denna programsats:

```
int[] number;
```

För att sen skapa själva arrayen och ange hur många element arrayen ska rymma skriver vi följande programsats:

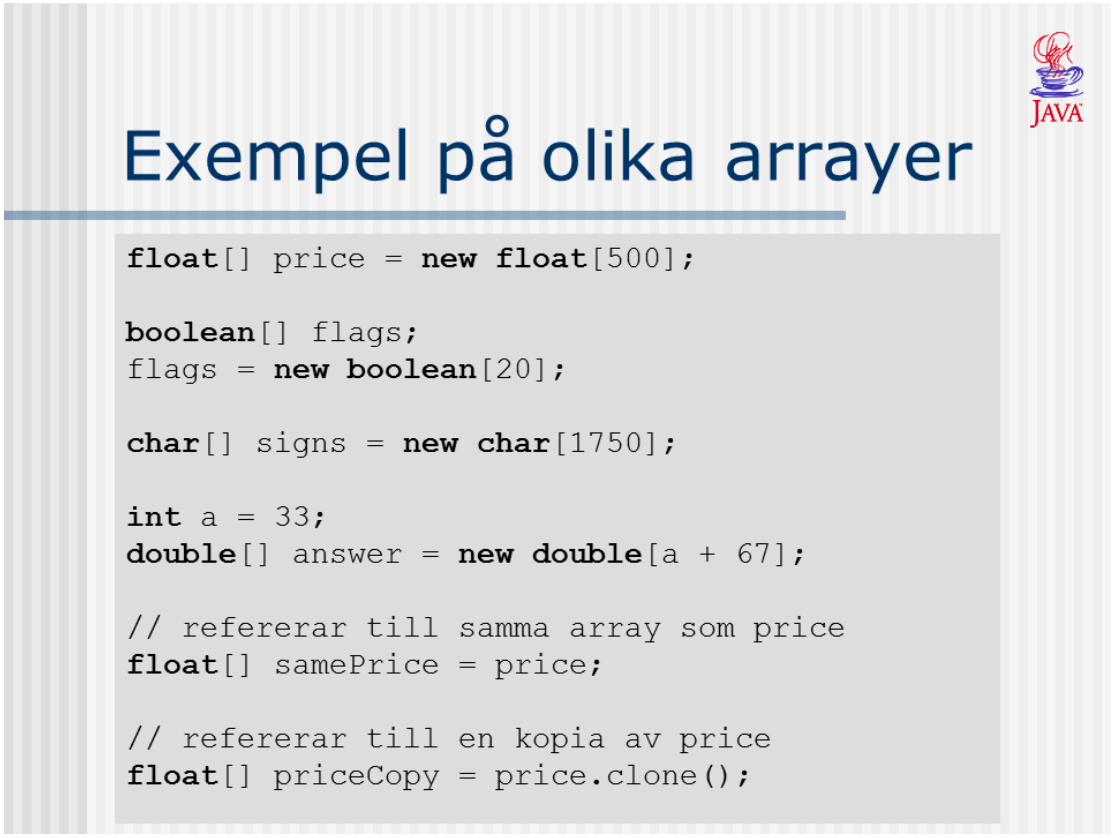
```
number = new int[10];
```

När man skapar arrayen, menas det med att man allokerar tillräckligt med utrymme i minnet för att kunna lagra så många element man angett av den aktuella typen. I vårt fall med arrayen number skapar vi ett ny array av typen `int` och där antalet element ska vara 10 (anges inom hakparenteserna).

Deklarationen och skapandet av arrayen kan kombineras på en enda rad vilket brukar vara det vanliga. Var man placerar hakparenteserna vid deklarationen kan man välja själv men det första alternativet (nedan) är vanligast i Java och rekommenderas därför:

```
int[] number = new int[10];
int  number[] = new int[10];
```

Båda dessa rader skapar ett heltals array med namnet number och som rymmer 10 element, där alla 10 element är av typen int. För att komma åt värdena i dessa element kan vi använda index 0, 1, 2, …, 9.



Bilden ovan visar olika sätt vi kan skapa arrayer på. Det viktiga är att vi använder hakparenteser för att visa att det är en array vi skapar och att vi innanför dessa hakparenteser anger hur många element vi vill kunna lagra i arrayen.

Första raden skapar en array med namnet price som rymmer 500 element (index 0 till 499) av typen `float`.

Andra raden deklarerar att vi vill använda en array av boolean med namnet flags. Tredje raden skapar denna array och sätter att den ska rymma 20 st `boolean`.

Antalet element som arrayen ska rymma behöver inte vara en heltalskonstant utan kan vara värdet av en annan variabel (dock av typen `int`). Det går även skapa en array där antalet element beräknas av ett uttryck.

Sista raden visar att vi kan skapa en array från ett redan existerande array. Arrayen `samePrice` och `price` kommer nu att referera till samma array. Vill vi i stället skapa en kopia av en array kan vi anropa metoden `clone`.

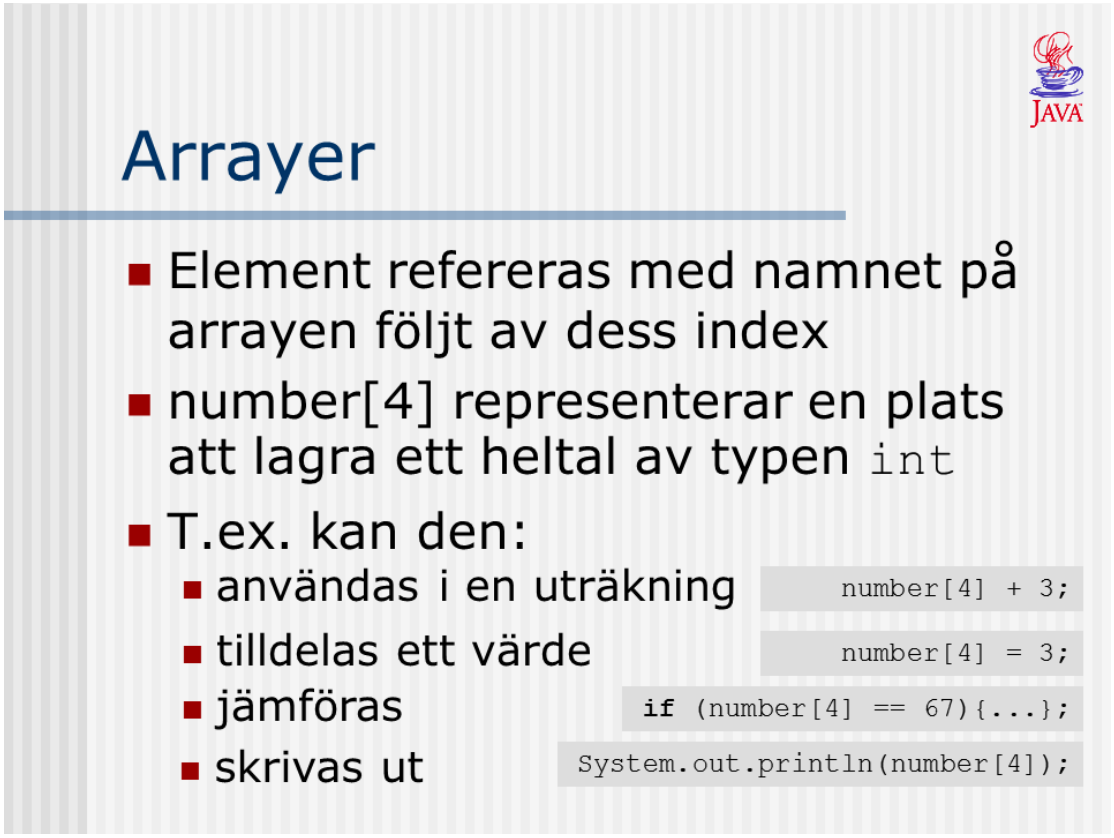
I java behandlas arrayer som objekt vilket gör att vi måste använda `new`-operatorn för att skapa en array.

I och med att en array behandlas som ett objekt gör att en array i Java även har medlemmar som kan användas. T.ex. finns instansvariabeln `length` som visar hur stor arrayen är (hur många element den max kan rymma).

När vi skapar t.ex. ett nytt Person-objekt så returnerar ju `new` en referens till det nyskapade objektet. Denna referens kan vi då tilldela till en variabel av typen Person (`Person p = new Person()`). På samma sätt fungerar det när vi använder `new` för att skapa en array. Precis som variabler av ett objekt är en referensvariabel så är en array-variabel också en referensvariabel, den refererar till den array som skapades med `new`. En array-variabel innehåller alltså inte alla element som ingår i arrayen, utan endast en referens till var elementen egentligen finns.

När arrayer används som argument vid metodanrop så är det en kopia på referensen som skickas och inte en kopia på de element som ingår i arrayen (kom ihåg att parametrar överförs via värdeanrop, dvs en kopia skickas). För primitiva typer är det en kopia på värdet som skickas men för objektreferenser är det en kopia på referensen som skickas). Det betyder att de förändringar som görs på arrayens element i en metod gäller för alla array-variabler som refererar till detta array.

Trots att en array ses som ett objekt i Java så är en array inte en instans av en Array-klass. Vi skapar alltså inte en array från en klass. Däremot finns det två klasser i Javas API som är nära besläktad med array. Dessa är klasserna `Array` och `Arrays`. Längre fram kommer vi att titta på hur vi kan använda klassen `Arrays` för att sortera innehållet i en array.

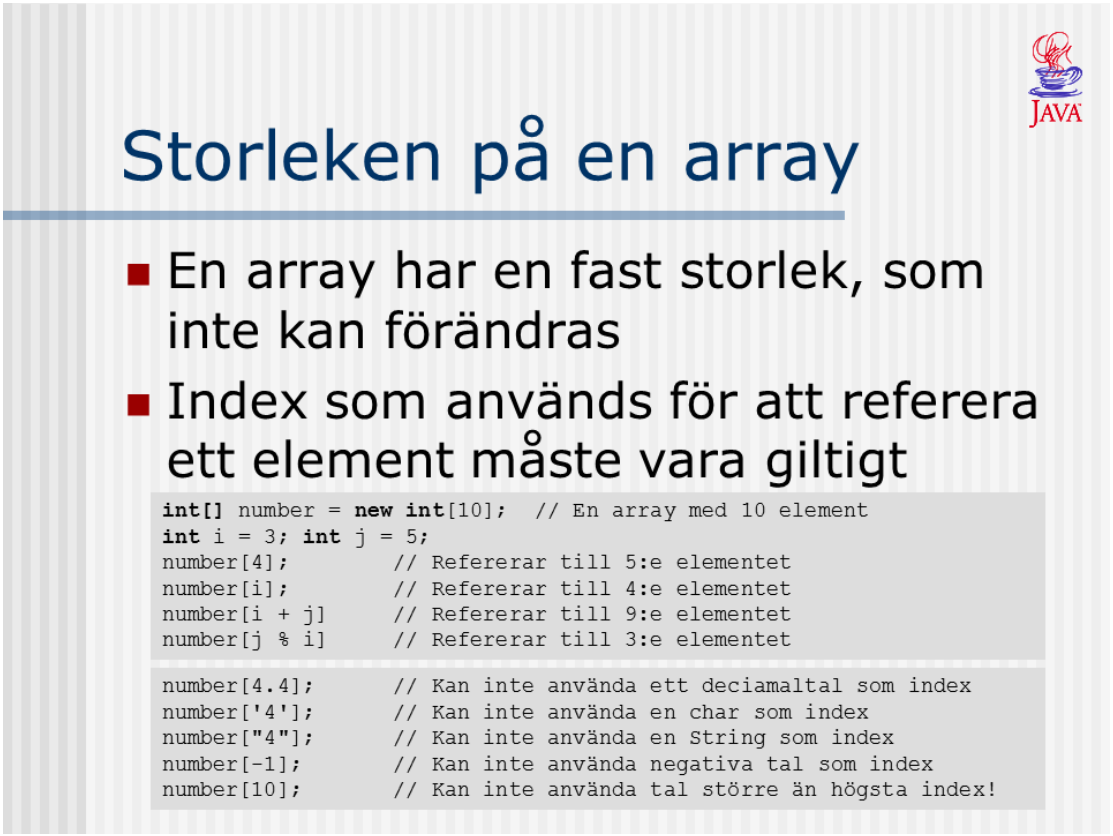


För att komma åt ett visst värde i en array använde vi elementets index. Ett element i en array refereras genom att ange namnet på arrayen samt elementets index inom hakparenteserna.

Uttrycket `number[4]` refererar alltså till värdet som elementet på index 4 har i arrayen `number`. `tal[4]` representerar en plats att lagra ett heltal av typen `int`, och kan användas överallt, och på samma sätt, som en vanlig variabel av typen `int` kan användas. T.ex. kan den användas i en uträkning, tilldelas ett värde och skrivas ut på skärmen.

För att tilldela elementet på index 4 värdet 3 skriver vi följande: `number[4] = 3;`

I exemplet **MyFirstArray.java** gör vi ett första försök med enkla arrayer.



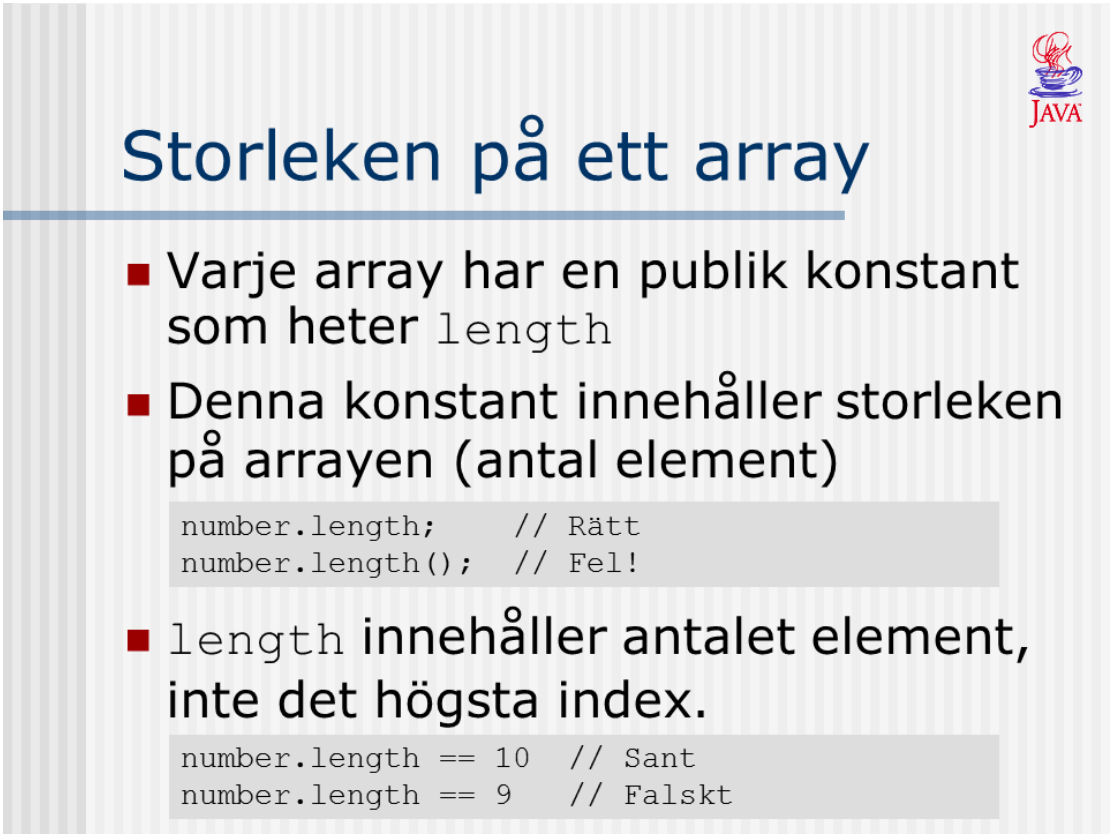
Alla arrayer i Java har en storlek som inte kan ändras efter att vi har skapat arrayen. Det betyder att har vi skapat en array som ska rymma 3 heltal kan vi senare i ett program inte öka storleken på arrayen så att den rymmer 5 heltal (vi måste då först skapa en helt nytt array).

Något annat man måste tänka på när man använder en array är att det index som används för att referera till ett visst element måste vara giltigt. Med giltigt menas framför allt att index måste vara av typen `int`. Sen om vi använder en heltalskonstant, värdet i en variabel eller värdet i ett uttryck, för att referera till ett visst element spelar mindre roll.

Däremot kan vi inte använda ett decimaltal, tecken eller sträng som index. Inte heller kan vi använda ett index som ligger utanför arrayens giltiga värden. Dvs vi kan inte ange ett negativt värde eller ett värde som är större än arrayens högsta index.

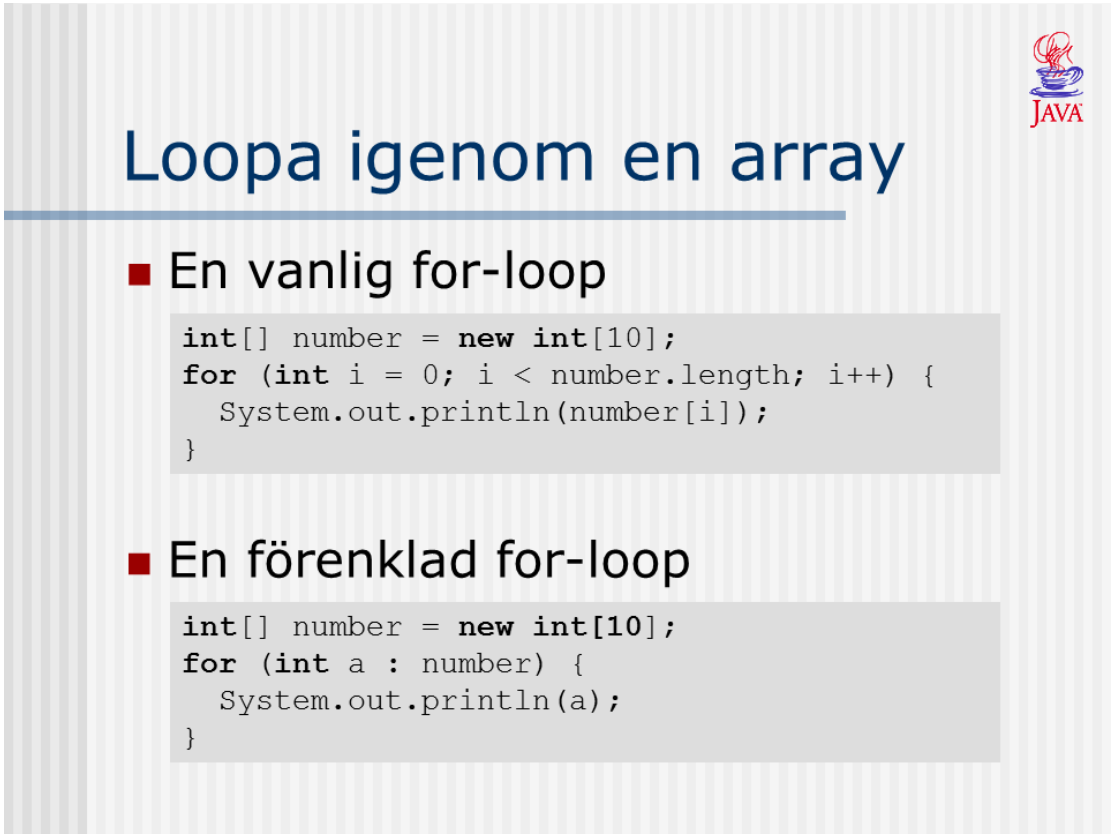
Just fallet att man anger ett index som är större än det som max är tillåtet brukar vara ett ganska vanligt fel (speciellt när vi använder arrayer i loopar). Det som händer vid dessa tillfällen är att den virtuella Javamaskinen genererar ett fel (`ArrayIndexOutOfBoundsException`).

Om vi t.ex. har följande array: `int[] number = new int[10]` så är det fel av oss att använda följande index: `number[10]` för att tilldela ett element ett värde.



Som nämnts tidigare så innehåller alla array i Java en medlemsvariabel som visar hur stor arrayen är, dvs hur många element som kan lagras i arrayen. Denna variabel heter `length` och är en publik konstant, vilket innebär att vi direkt kan komma åt värdet på variabel genom att med punktnotation ange arrayens namn följt av `length`. Observera att length INTE är en metod varför vi inte ska skriva `length()`.

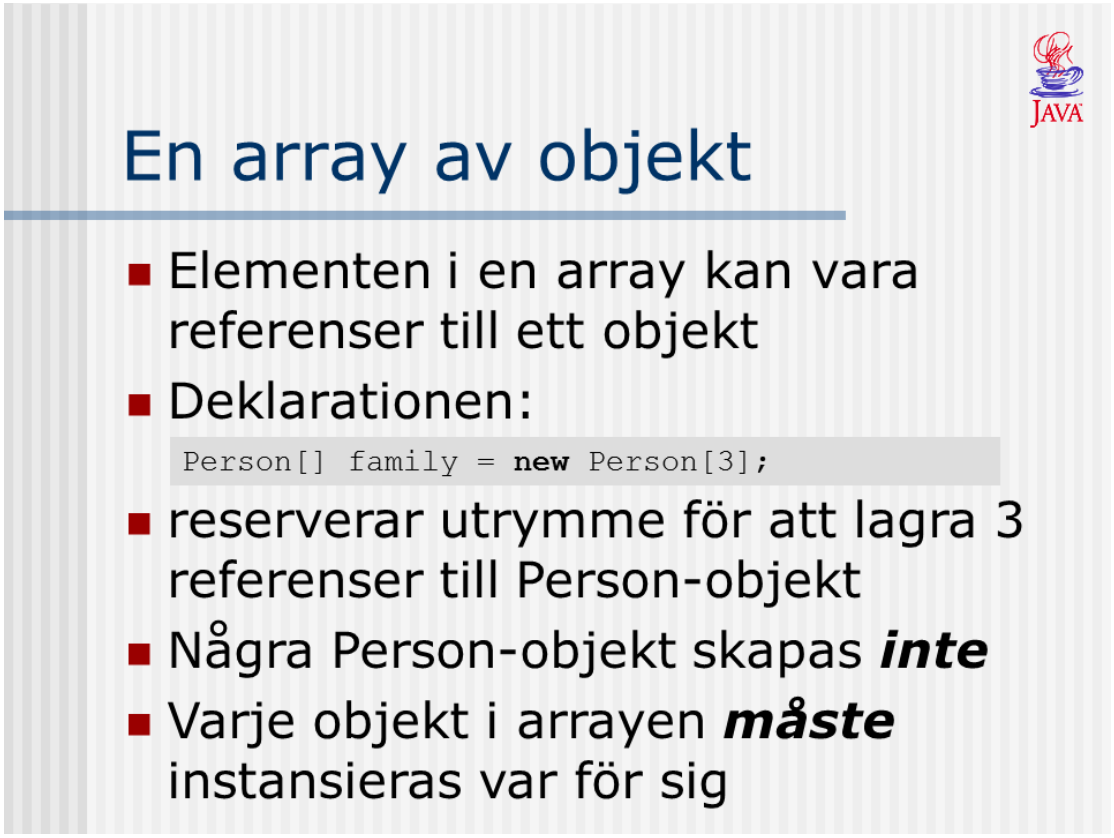
Har vi heltals-arrayen `int[] number = new int[10]` kommer anropet `number.length` att ge oss storleken på arrayen, vilket är 10. `length` innehåller alltså antalet element som arrayen max kan lagra och inte det högsta index som man kan använda eller hur många element som tilldelats värden. För vårt exempel kommer `length` att innehålla värdet 10 eftersom vi skapade en array med plats för 10 heltal.



Många gånger vill man loopa igenom alla element i en array för att t.ex. tilldela elementen värden eller avläsa värdena. Absolut enklast är att använda en for-loop vars initieringsvariabel sätts till 0 (eftersom första index i en array är 0). I villkorsdelen anger vi att initieringsvariabeln ska vara lägre än antalet element i arrayen (`length`). I loopen använder vi sen initieringsvariabeln för att referera till arrayens element.

Från och med JDK version 1.5 finns det en variant av for-loopen som vi kan använda när vi vill avläsa de värden en array har. Denna variant är praktisk när vi vill skriva ut alla värdena. Utgår vi från exemplet i bilden kommer variabeln `a` att tilldelas vart och ett av de värden de olika elementen i arrayen number har. Viktigt att tänka på är att denna variant av for-loopen gör en kopia på värdet i varje element. Vi kan alltså inte tilldela en array sina värden med denna for-loop när arrayen innehåller någon primitiv datatyp.

I exemplet **RandomArray.java** visar vi på hur man kan slumpa tal till en array. Programmet frågar efter hur många tal som ska slumpas, inom vilket intervall slumptalen ska ligga och skriver därefter ut alla talen samt summa och medelvärde av talen.



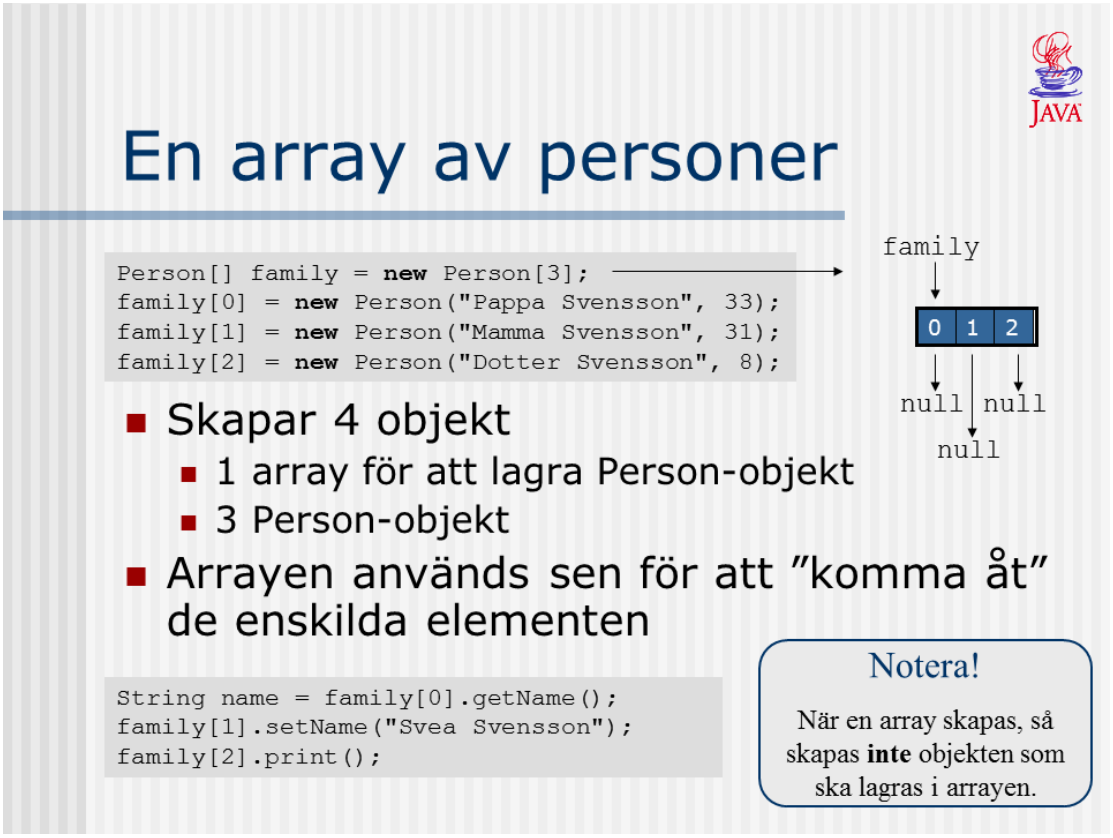
En array kunde som sagt även innehålla referenser till objekt. Vi kan alltså skapa array för att ”lagra” olika typer av objekt. T.ex. kommer följande deklaration:

```
Person[] family = new Person[3];
```

att skapa en array där vi kan lagra olika objekt av klassen Person. Det är inte själva objektet som lagras i arrayen utan det är en referens till objektet som lagras i arrayen. Ovanstående deklaration skapar alltså en array som rymmer 3 referenser till Person-objekt.

Vi kan skapa array för alla typer av objekt (klasser), egendefinierade eller sådana från javas klassbibliotek (API), dock inte en array som lagrar objekt av olika klasser.

Det viktiga är att när vi skapar en array för att hålla objekt så skapas det INTE några egentliga objekt i samband med skapandet av arrayen. I vårt fall ovan med arrayen family (som rymmer referenser till 3 Person-objekt) skapas det inte några faktiska objekt av Person. Utan varje objekt i arrayen måste vi skapa (instansiera) efteråt.



Följande exempel i bilden ovan skapar fyra nya objekt. Först en array för att spara tre Person-objekt (minns att en array ses som ett objekt). Därefter skapas tre Person-objekt som vi tilldelar de olika elementen i arrayen famiyj. Första objektet i index 0, andra i index 1 och sista objektet i index 2.

Första programsatsen skapar alltså en array med namnet family för att lagra tre Person-objekt. Än så länge har det inte skapats några objekt av klassen Person, så innehållet i arrayen family kommer att vara s.k. null-referenser. Dvs varje element i arrayen family "pekar" mot null (dvs ett icke skapat objekt).

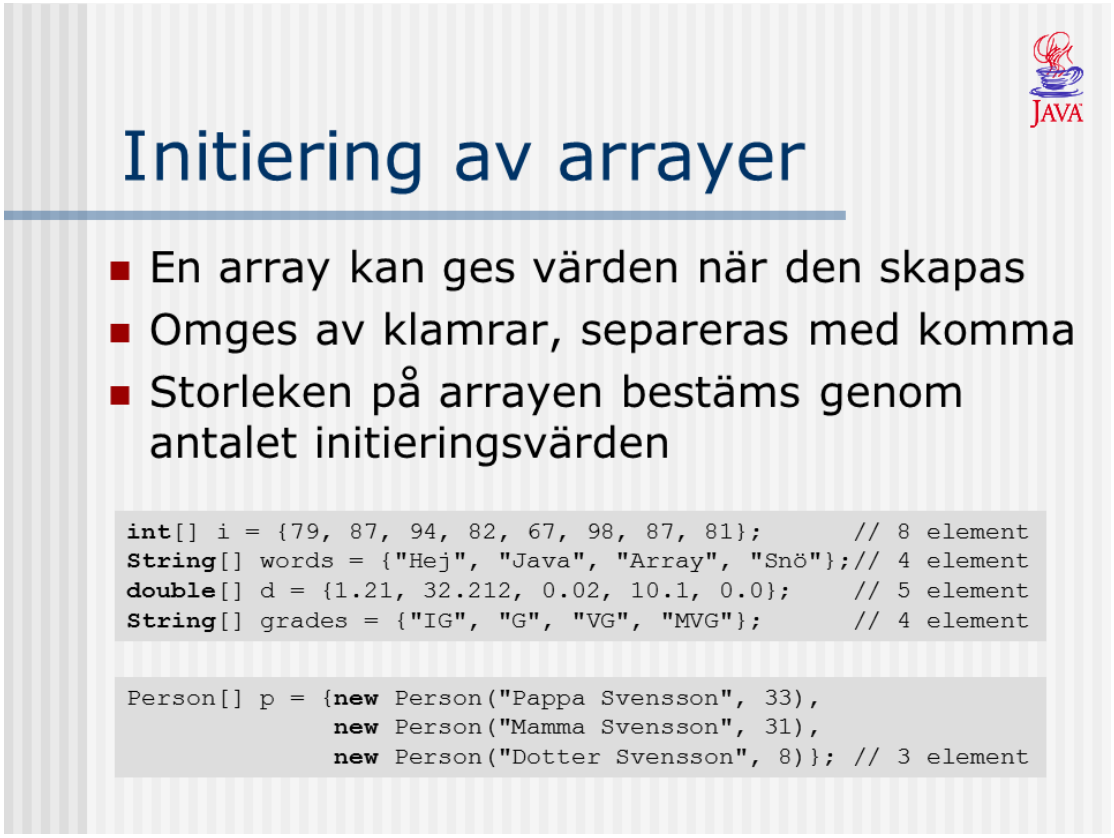
De följande tre programsatserna skapar sen de individuella objekten och tilldelar dem till elementen i arrayen. Så efter att dessa rader har körts kommer första elementet i arrayen family att innehålla en referens till Person-objeket med namnet "Pappa Svensson" och åldern 33.

När vi tilldelat Person-objekt till arrayen kan vi använda arrayens namn för att komma åt de individuella objekten. En referens till personen "Pappa Svensson" finns i `family[0]`, en referens till personen "Mamma" finns i `family[1]` etc. Dessa kan vi använda för att anropa metoder på objektet, t.ex. för att skriva ut objektens namn. T.ex. kommer programsatsen `family[0].print()` att anropa `print`- metoden och skriva ut innehållet i objektet för pappan.

Om vi redan skulle ha Person-objekten innan arrayen deklareras kan vi helt enkelt tilldela ett element i arrayen denna referens.

```
Person p1 = new Person("Sture Svensson", 65);
family[0] = p1;
```

Ta en titt på exemplet **PersonArray.java** där en array skapas för att rymma referenser till Person-objekt.



Det vanliga sättet använda en array är att först skapa arrayen och därefter tilldela varje element i arrayen ett värde (oftast i en loop för att underlätta). Vi kan också i samband med att vi skapar en array ge elementen i arrayen värden (jämför lite med konstruktorn som ger ett objekt värden i samband med att vi skapar objektet).

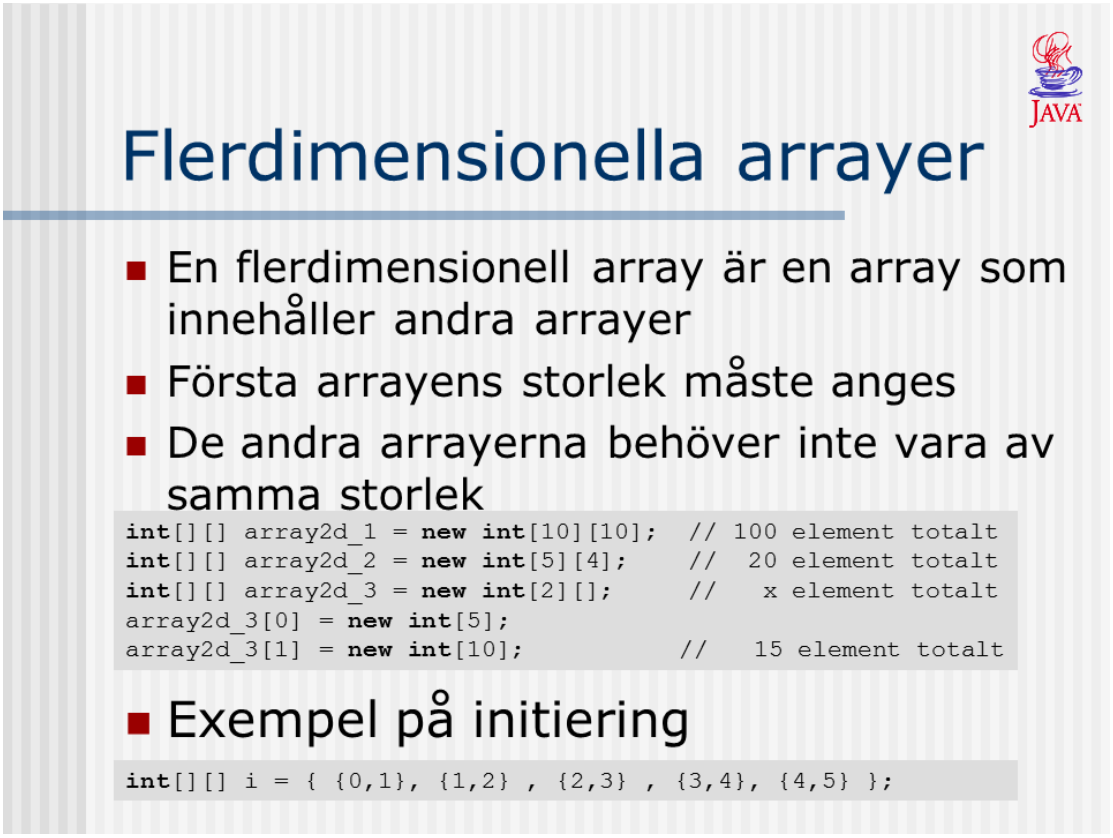
För att tilldela en array värden i samband med skapandet anger vi värdena i en kommaseparerad lista omsluten av klamrar {}. Storleken på arrayen kommer att bestämmas genom hur många värden vi anger inom klamrarna.

T.ex. kommer följande programsats:

```
int[] number = {2, 4, 6, 8, 10};
```

att skapa en array av heltal som kommer att få storleken 5 (eftersom vi angav 5 värden innanför klamrarna). Första elementet (index 0) kommer att ha värdet 2, andra elementet (index 1) kommer att ha värdet 4 etc.

Vill vi skapa en array av objekt (t.ex. Person-objekt) gör vi på samma sätt som för de primitivatyperna. Vi skapar Person-objekt i en kommaseparerad lista. Exemplet i bilden ovan kommer alltså att skapa en array som rymmer referenser till 3 Person-objekt.



Det är även möjligt att skapa en array som innehåller arrayer, dvs en array vars element även de är arrayer. Sådana array kallas vanligtvis för flerdimensionella array (eller 2D-arrayer). Att skapa en 2D-array är inte mer komplicerat än att skapa en vanlig (1D) array. Man använder bara 2 st hakparanteser efter varandra. Ex:

```
int[][] a = new int[5][3];
```

Ovanstående deklaration skapar en 2d-array med totalt 15 element (5*3=15). En 2D-array var som sagt en array av arrayer och i detta fall skapar vi array som innehåller 5 element. Varje element innehåller i sin tur en array som innehåller 3 element.

För att fylla första arrayens element skriver vi:

```
a[0][0] = 11; a[0][1] = 22; a[0][2] = 33;
```

För att fylla andra arrayens element skriver vi:

```
a[1][0] = 44; a[1][1] = 55; a[1][2] = 66;
```

etc.

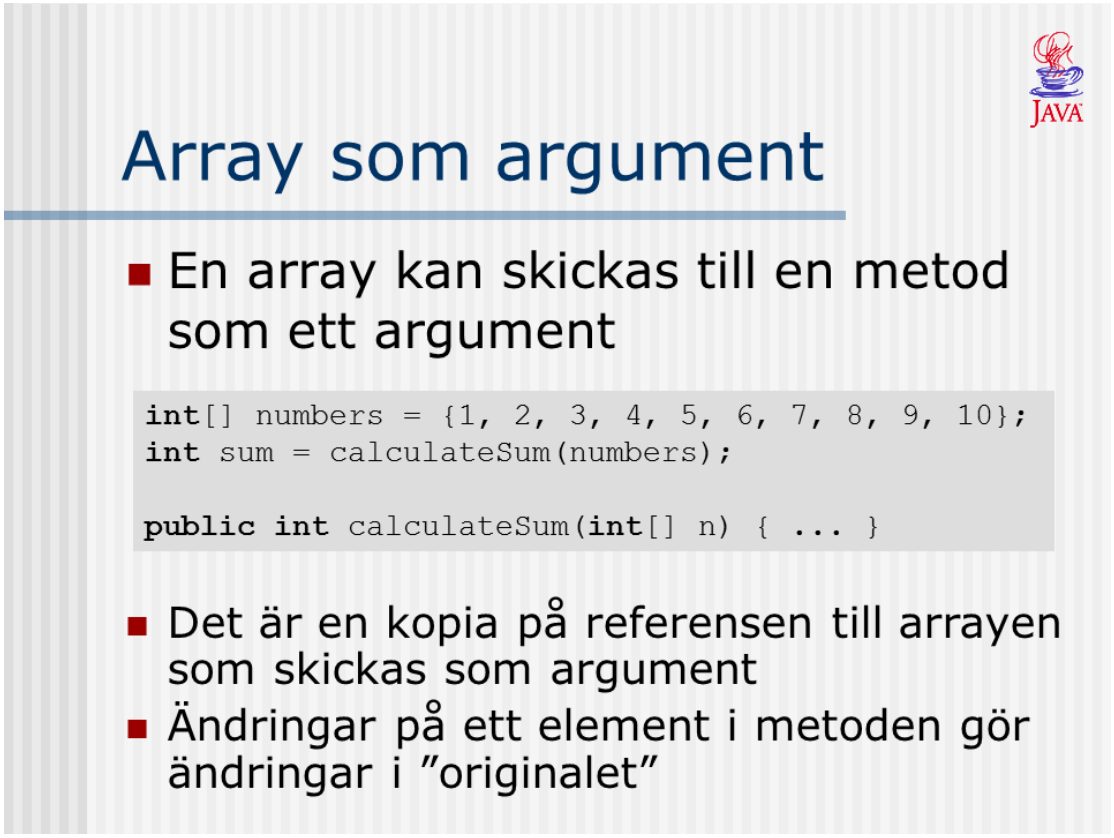
När vi skapar en 2D-array måste vi alltid ange första arrayens storlek. Däremot är det inte nödvändigt att ange de andraarrayens storlek. Innan vi kan börja använda en array skapat på detta vis, måste vi själva skapa ett nytt array för varje element. Nu behöver inte varje array vara av samma storlek heller. Se exempel i bilden för arrayen array2d_3.

Vi kan direkt initiera en 2D-array när vi deklarerar den. Precis som när vi initierar en 1D-array så anger vi elementens värden med en kommaseparerad lista inom klamrar {}. Storleken bestäms då genom antalet värden vi skriver.

Exemplet i bilden ovan skapar en 2D-array som motsvarar följande deklarering:

```
int i[][] = new int[5][2];
```

I exemplet **Multiplication.java** visar vi på ett exempel där ett flerdimensionellt array används. Användaren får skriva in vilken multiplikationstabell som ska visas.

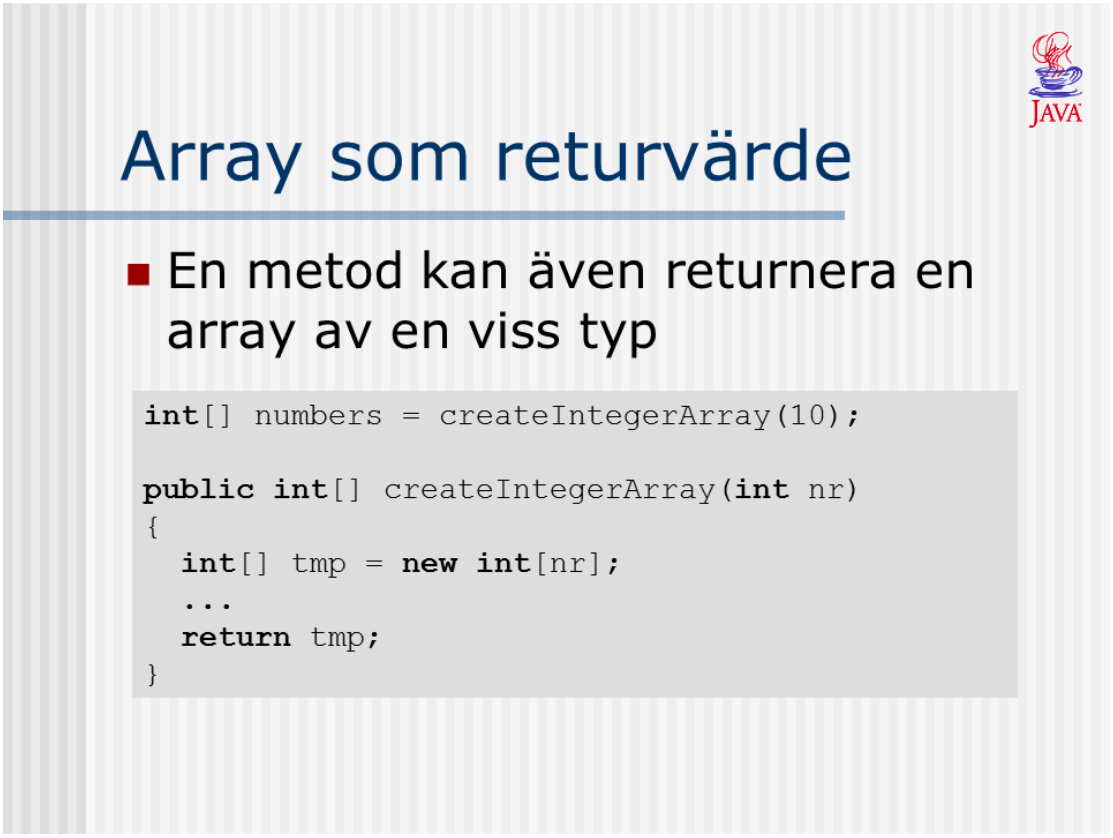


Precis som vi kan skicka en vanlig variabel (antingen av primitivtyp eller av ett objekt) som ett argument till en metod, kan vi även skicka en array som argument till en metod.

För att skicka en array som argument räcker det med att ange arrayens namn i metodanropet, dvs vi behöver inte använda hakparenteserna när vi skickar arrayen. Däremot måste vi i metod-deklarationen i den metod som ska tar en array som parameter använda hakparenteserna (se exempel i bilden ovan, metoden calculateSum).

Kom ihåg att en array i Java ses som ett objekt, så en arrayvariabel är en referens till var arrayens element finns. Själva variabeln innehåller inte några värden. Så när vi skickar arrayen som ett argument är det en kopia på referensen till arrayen som skickas och inte en kopia på varje element i arrayen.

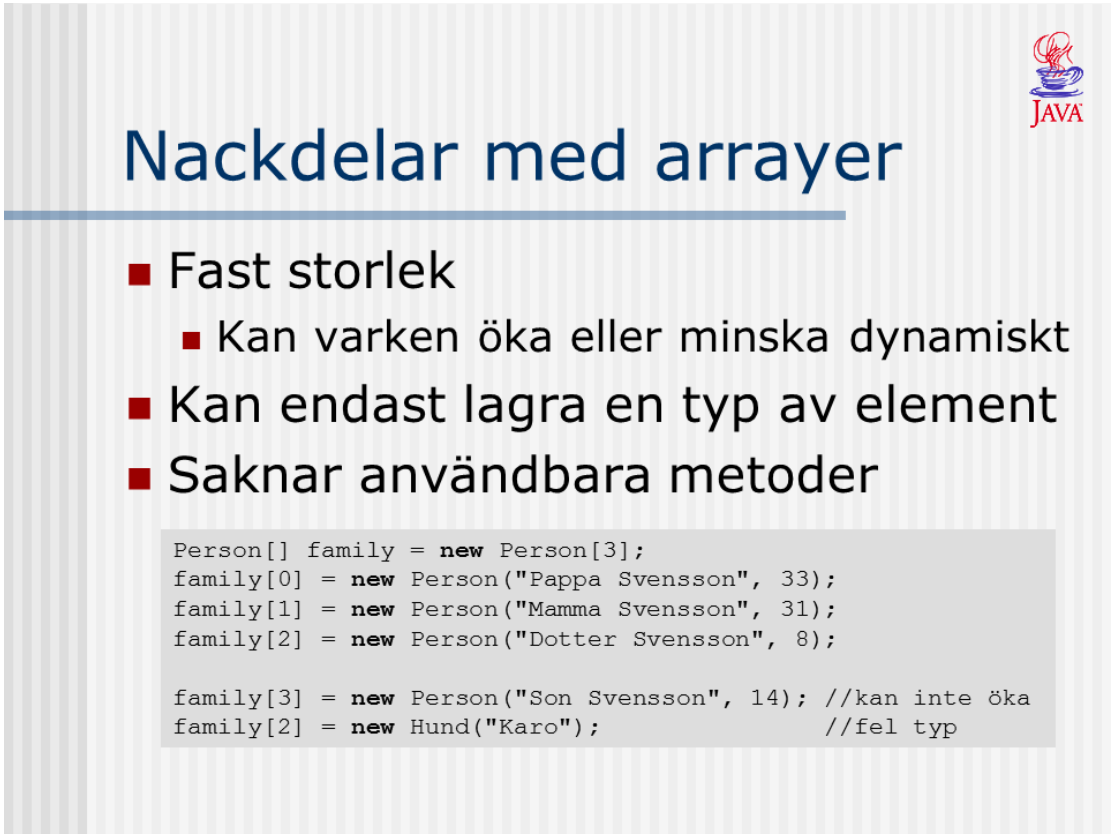
Detta innebär att om vi modifierar något av arrayens element inuti metoden görs dessa förändringar även på ”originalet” (gäller både för array av primitiva typer och array för objekt).



En metods returtyp kan också vara en array av någon typ, dvs vi kan returnera en array från en metod. När vi anger metodens returtyp i metoddeklarationen måste vi använda hakparanteser för att visa att det är en array som ska returneras. När vi däremot returnerar själva arrayen i metoden så anger vi bara namnet på arrayen (utan hakparenteser). Se exemplet i bilden ovan.

Den array som returneras kan vi tilldela till en annan array av samma typ. Vi behöver inte ange storleken på denna array utan den bestäms av storleken av arrayen som returnerades. Vi kan skapa en helt ny array (som exemplet i bilden), men även tilldela returvärdet till ett befintligt array (som då mister sitt tidigare innehåll).

Ta en titt på exemplet **PassAndReturn.java** där vi har metoder som tar arrayer som parametrar och som returnerar arrayer. Några metoder innehåller kod som jag inte går igenom vad den exakt gör (skapar en tabell som visar innehållet i arrayen).



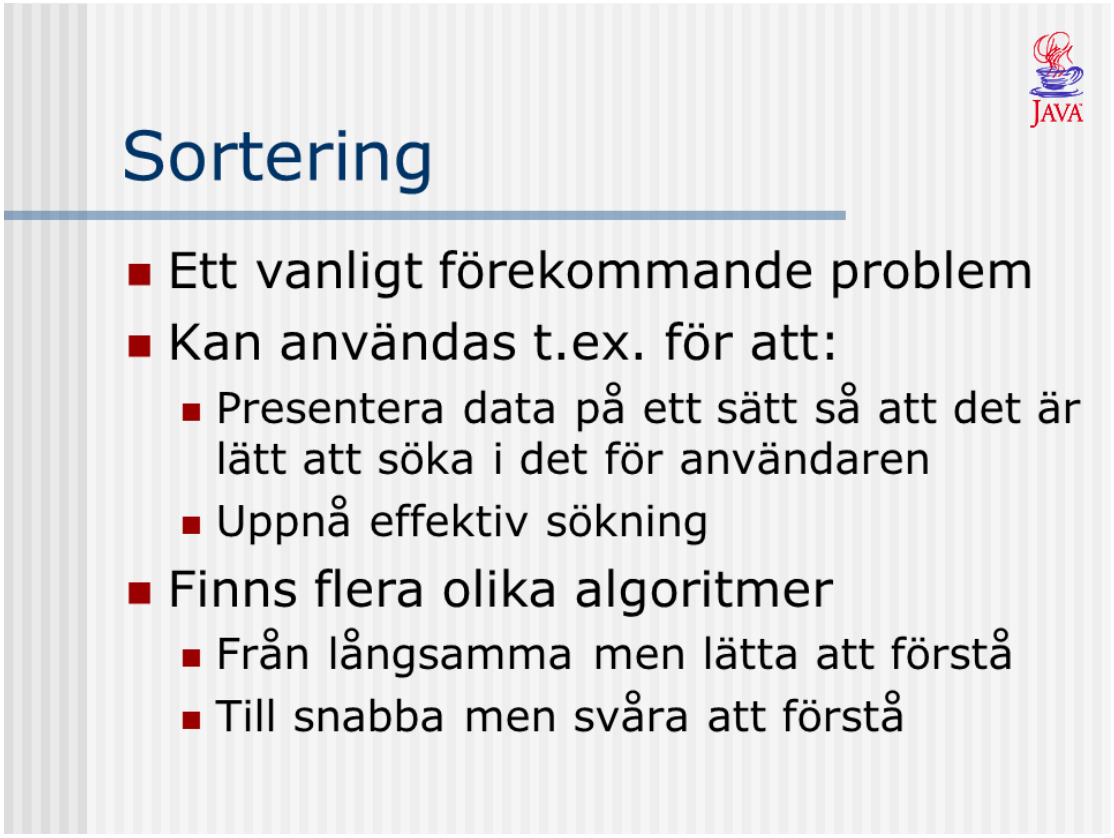
Arrayer kan vara väldigt smidiga och användbara, men de har ett par nackdelar. Den största är kanske att när vi väl har skapat arrayen är storleken fast och vi kan varken öka eller minska den efter behov. Har vi satt storleken till 5 så ökas den inte automatisk när vi försöker lägga till ett 6:e element (vi får då ett felmeddelande).

En annan nackdel är att en array bara kan lagra värden av en och samma typ. Det är med andra ord inte möjligt att lagra både `int` och `double` i en array, eller Person-objekt och Card-objekt i samma array.

En tredje nackdel med array är att de saknar metoder för att manipulera innehållet i arrayen. Ibland hade det varit smidigt att kanske sätta alla element i en array till ett visst värde, eller haft något sätt att kontrollera om ett visst värde finns i arrayen etc.

Det finns en klass i Javas standardbibliotek (`java.util.ArrayList`) som åtgärdar dessa nackdelar och denna klass tittar vi på lite senare i lektionen.

I klassen **ExpandArray.java** visas ett exempel på vad som måste göras för att utöka antalet element i en array.

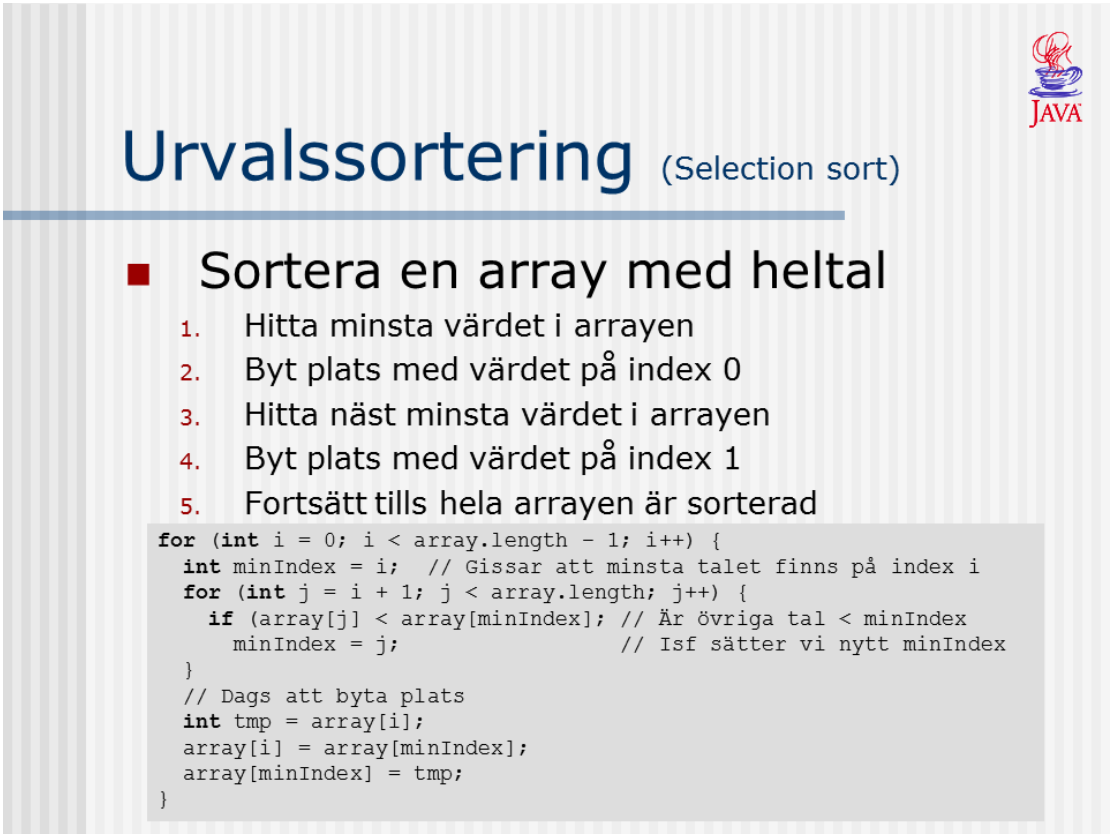


Sortering är ett vanligt förekommande problem inom datalogin. Dels finns det en enorm mängd av applikationer som presenterar sorterat data för användaren, t.ex. adresslistor, innehållsförteckningar och kalendrar. Dels finns det många andra tillämpningar av sortering.

Antag till exempel att vi ska avgöra om en array med n antal element innehåller några dubbletter. En enkel lösning är då att jämför alla element med alla andra element. Detta ger två nästlade loopar, dvs $O(T(n)) = n^2$ antal jämförelser. Antag istället att vi först sorterar sekvensen. Dubbletter kommer då att ligga bredvid varandra och vi kan hitta dem genom att helt enkelt gå igenom hela sekvensen en gång, vilket ger $O(T(n)) = n$. Går det att sortera snabbare än n^2 blir denna lösning bättre än den föregående.

Eftersom sortering är så vanligt finns det en mängd välkända sorteringsalgoritmer med olika egenskaper. Vi ska nu titta både på en enkel, långsam och på en mer komplicerade men snabbare.

Det vanligaste är att ett antal poster ska sorteras efter en viss nyckel (eller efter flera nycklar) som endast utgör en del av hela posten. I exemplen här består posterna för enkelhets skull endast av en nyckel (som är en `int`).



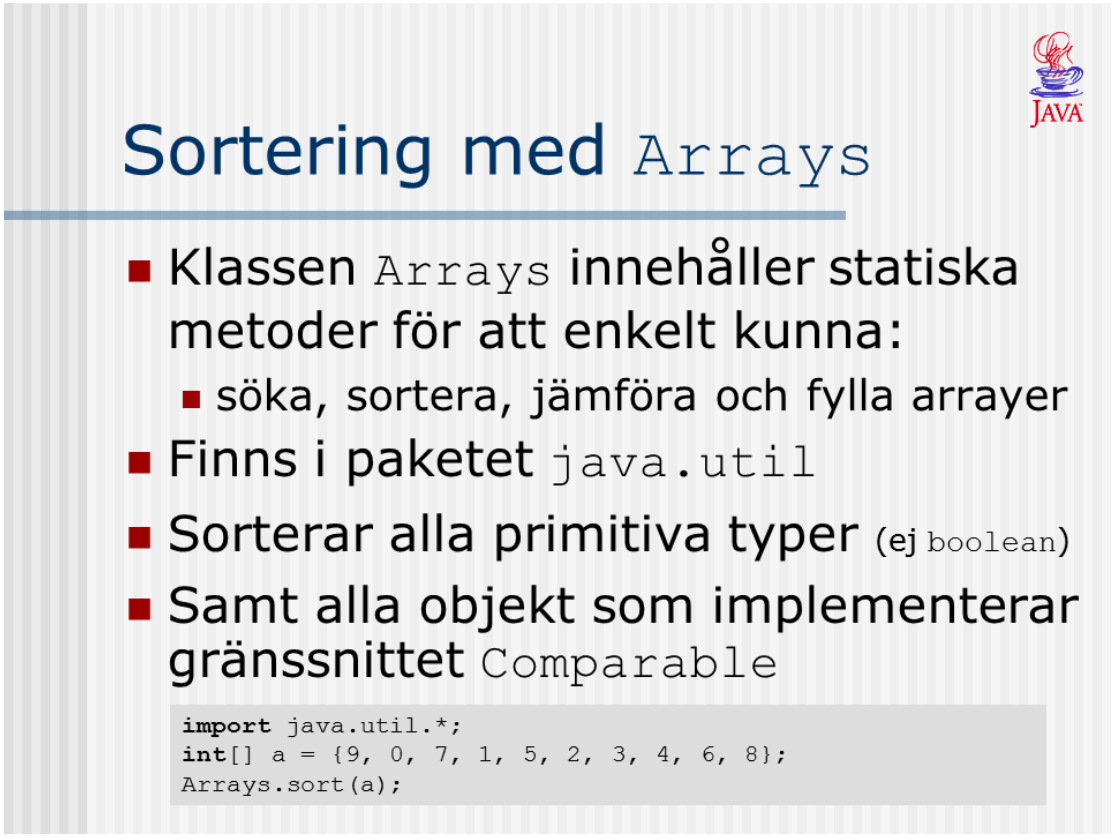
Urvalssortering (selection sort på engelska) är väldigt enkel sorteringsalgoritm. Den fungerar som följer: Hitta minsta värdet i arrayen som ska sorteras och byt plats med värdet på första index (0). Hitta därefter det näst minsta värdet i arrayen och byt plats med värdet på andra index (1). Fortsätt så här till dess att hela arrayen är sorterat.

Som sagt är denna typ av sortering väldigt enkel att förstå och fungerar ganska bra om arrayen som ska sorteras innehåller få element. Vid större antal element är denna typ av sortering inte att föredra.

Algoritmen implementeras genom att en for-loop går från första (0) till näst sista elementet (array.length-1). Det är elementen denna loop refererar till som vi jämför med övriga för att finna minsta värdet. För varje varv i denna loop antar vi att aktuellt index innehåller det minsta värdet. Därefter följer ytterligare en for-loop som går igenom alla övriga element i arrayen. Om värdet i aktuellt index är mindre än värdet i det index vi antog innehåll det minsta värdet, sätter vi detta till nytt minsta värde.

När vi jämfört alla element med varandra är vi säkra på vilket index som innehåller det minsta värdet. Vi byter då plats på värdet med det index som yttre-loopen refererar till.

I exemplet **SelectionSort.java** har jag implementerat algoritmen för urvalssortering. Ta en titt på detta exempel. Prova gärna även att ändra i källkoden så att elementen sorteras i sjunkande ordning.



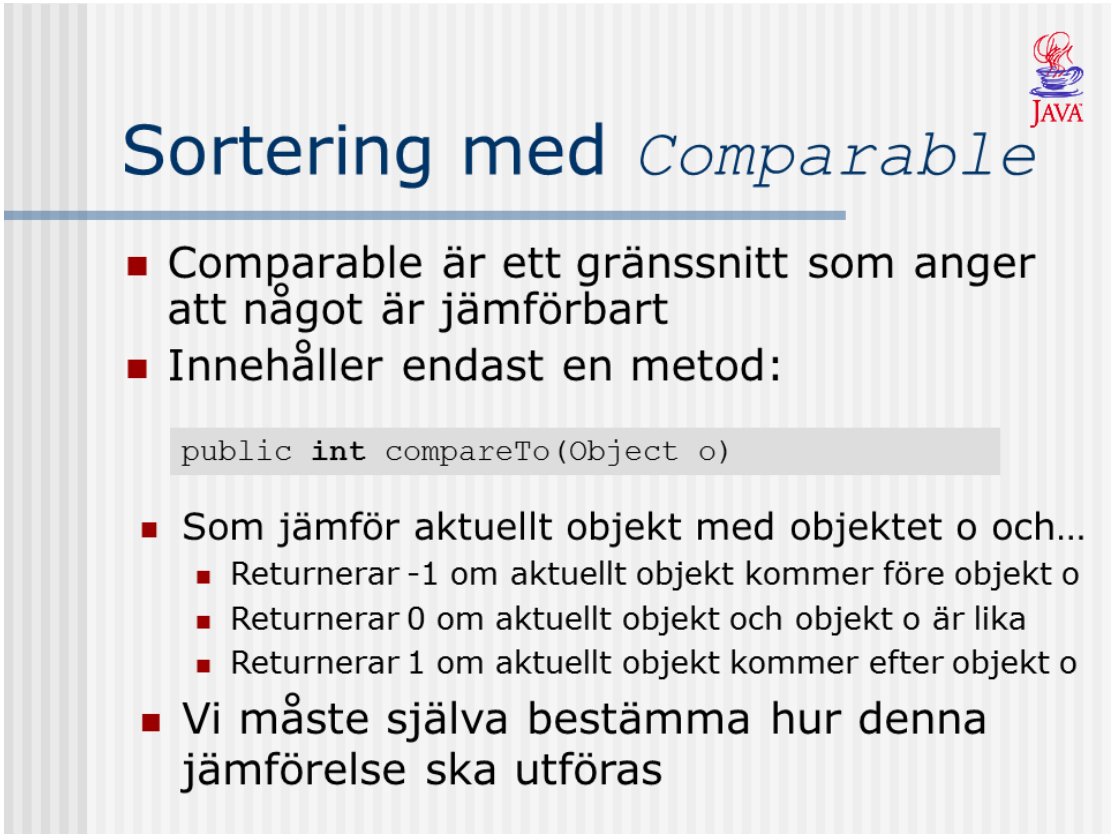
I paketet `java.util` finns klassen `Arrays` med vilken vi på olika sätt kan manipulera en array av primitiva typer eller av objekt. Framför allt kan vi använda klassen för att söka bland elementen i en array efter ett visst värde, sortera elementen i arrayen, jämföra olika array med varandra och fylla arrayens element med värden. Med `Arrays` kan vi väldigt enkelt (och snabbt) sortera arrayer av primitiva typer (ej av typen `boolean`), samt alla objekt som implementerar gränssnittet `Comparable`.

Alla metoder i klassen `Arrays` är statiska vilket innebär att vi inte behöver skapa något objekt av klassen utan direkt kan anropa metoderna med klassnamnet. För att sortera en array gör vi följande anrop:

```
Arrays.sort(namn_på_arrayen);
```

Metoder som sorterar arrayer av primitiva typer implementerar en variant av ’Quick Sort’ som är en väldigt snabb sorteringsalgoritm (men som är väldigt komplicerad att implementera och förstå). Metoder som sorterar objekt implementerar en variant av ’Merge Sort’.

Exemplet **ArraysSort.java** är i stort sett exakt lika som **SelectionSort.java**, men att vi istället använder `Arrays.sort` för att sortera elementen.
I exemplet **SortTime.java** visar jag på hur otroligt mycket snabbare sortering med `Arrays.sort` är jämfört med urvalssortering (för arrayer med många element).

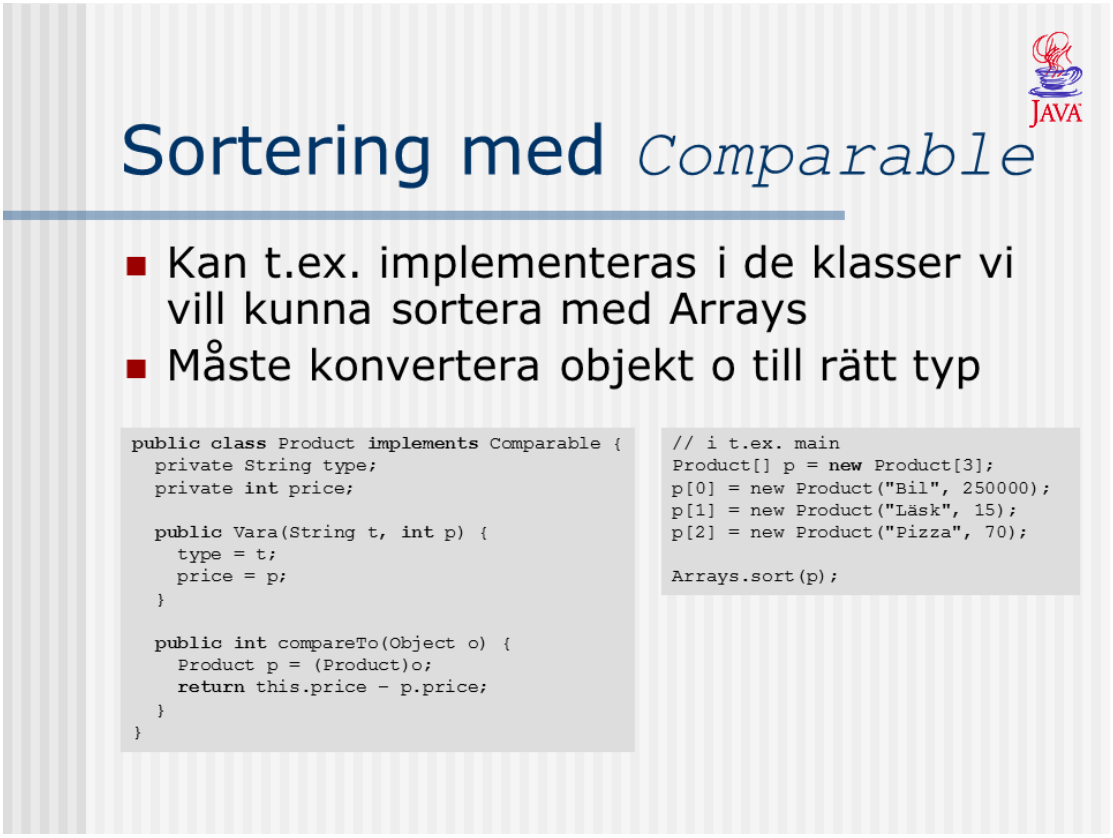


När vi utnyttjar metoden `sort` i klassen `Arrays` på de primitiva typerna så är den ordning elementen ska sorteras efter redan bestämd. För tal gäller att de naturligt sorteras i storleksordning och där det tal med lägsta värdet kommer för ett tal med högre värde. När det gäller sortering av objekt finns det ingen s.k. naturlig ordning angiven eftersom olika typer av objekt kan/ska sorteras på olika sätt.

För att vi ska kunna sortera egendefinierade objekt med hjälp av `Arrays.sort` måste vi själva ange den ordning objekten ska sorteras i. Detta gör vi genom att vi låter klassen implementera gränssnittet `Comparable` som anger att något är jämförbart med något annat (av samma klass). Detta gränssnitt innehåller endast metoden `compareTo`.

Den tar som parameter ett objekt av klassen `Object` (eller en subklass till `Object`) och ska returnera ett negativt tal om aktuellt objekt anses komma före det objekt som skickas som argument, 0 om båda objekten anses lika, eller ett positivt tal om aktuellt objekt anses komma efter skickat objekt.

Hur denna jämförelse ska gå till måste vi själva bestämma och koden för detta implementeras i `compareTo`.



Vill vi sortera en array som innehåller objekt av något slag måste klassen implementera `Comparable`. Observera att metoden tar ett `Object` som parameter vilket innebär att vi i metodens implementation måste typkonverteras till rätt typ för att kunna komma åt metoderna för aktuellt objekt.

Exemplet i bilden har vi en klass som representerar en produkt med sitt namn och pris. Vi vill kunna sortera flera produkter med hjälp av `Arrays.sort` och klassen implementerar därför gränssnittet `Comparable` (`implements Comparable`). I klassen `Product` måste vi nu tillhandahålla metoden `compareTo` och i denna skriva kod som jämför en produkt med en annan. I exemplet jämför vi produkternas pris så att dessa ska kunna sorteras prismässigt (vara med lägre pris först). Vi hade givetvis kunnat skriva kod som jämför namnet på varorna för att sortera varor i bokstavsordning.

Det först vi gör i metoden är att konvertera argumentet som skickas till metoden till rätt typ (från `Object` till `Product`). Här borde vi naturligtvis först göra en kontroll så att aktuellt objekt verkligen går att konvertera till en produkt, men vi antar att så är fallet. Därefter returnerar vi skillnaden mellan priset på aktuell produkt och produkten som skickades till metoden. Är aktuellt pris lägre kommer ett negativt tal att returneras (skillnaden i hur mycket lägre priset är). Är aktuellt pris högre än priset på varan som skickades till metoden returneras ett positivt tal (skillnaden i hur mycket dyrare aktuell produkt är). Noll returneras om båda produkterna kostar lika mycket.

Vi kan nu skapa en array av typen `Product` och sen enkelt sortera arrayens element efter deras pris med `Arrays.sort`. Det svåra är att implementera metoden `compareTo` så att den på ett smidigt och effektivt sätt jämför objekten på ett korrekt sätt.

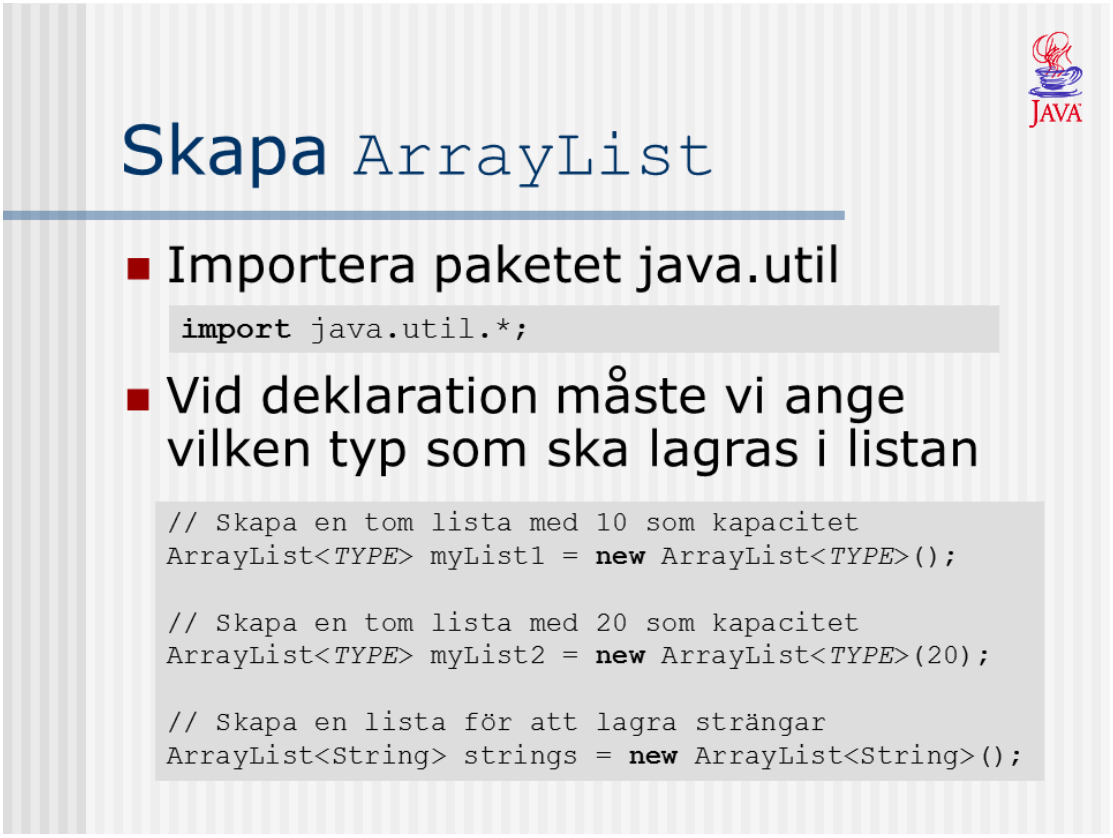
I exemplet **Person2.java** implementerar klassen gränssnittet `Comparable` och metoden `compareTo` jämför två personer med avseende på åldern. I exemplet **PersonSort.java** provar vi sen att skapa en array av typen Person och sorterar den med `Arrays.sort`.

En av nackdelarna med arrayer var att när de väl är skapade har det en viss storlek som inte går att ändra. Detta kan vara ett problem om vi i ett program vill använda en array, men vi från början inte vet hur många element vi kommer att behöva lagra. Antingen får vi ta i med en storlek som vi vet säkert kommer att räcka eller, vid behov, skapa en ny och större array. Det vill säga att utöka arrayen med fler element så som vi visade i exemplet **ExpandArray.java**. Som tur är finns det i Javas API ett antal klasser som automatiskt kan utöka innehållet. En av dessa är `ArrayList` som vi nu ska kika närmare på.

Med en `ArrayList` behöver vi inte ange någon storlek på antal element den ska innehålla. Inte heller behöver vi utöka antalet element utan det sker automatiskt när behovet uppstår. Eftersom `ArrayList` är en klass har den möjlighet att tillhandahålla metoder med vilka vi kan manipulera listan. Som vi tittar på lite senare finns bland annat metoder för att lägga in nya element i listan.

Tillskillnad från en array, som kan lagra både element av primitiva typer och av objekt (dock inte samtidigt), kan en `ArrayList` endast lagra objekt. Det innebär att vi inte direkt kan lagra primitiva typer i en `ArrayList`. Nu är det inget egentligt problem eftersom vi i stället för de primitiva typerna istället kan använda dess motsvarande omslagsklasser. Det vill säga om vi till exempel vill lagra heltal i en `ArrayList` använder vi omslagsklassen `Integer`.

Internt i en `ArrayList` används faktiskt en array för att lagra elementen i listan. Den automatiska utökningen av antalet element utförs på snarlikt sätt som i exemplet **ExpandArray.java**.



För att vi ska kunna använda `ArrayList` i en klass måste vi först importera klassen genom att skriva någon av följande programsatser:

```
import java.util.*; // Alla klasser i paketet
import java.util.ArrayList; // Enbartklassen ArrayList
```

Därefter kan vi skapa en `ArrayList` genom att använda någon av dess konstruktorer. En `ArrayList` har både en storlek och en kapacitet. Storleken anger hur många element listan innehåller och kapaciteten är hur många element listan maximalt kan innehålla innan den måste utökas. Det vanligaste sättet att skapa en `ArrayList` är att använda den parameterlösa konstruktorn. Kapaciteten sätts då till 10, vilket innebär att när vi lägger in det 11:e elementet kommer listans kapacitet automatiskt att utökas. Kapaciteten utökas då med ca en faktor av 1,5.

Även om jag skrivit ganska mycket om listans kapacitet är det något vi normalt inte behöver bry oss om. När kapaciteten är slut utökas den som sagt automatiskt. Denna utökning innebär en kopiering av den interna arrayen och allokering av en ny array. Detta tar givetvis lite tid och vet vi av att vi ska lagra väldigt många element i listan kan det vara en fördel att i förväg specificera listans kapacitet. Vi använder då konstruktorn som tar ett heltal som argument. Detta heltal specificerar listans initiala kapacitet.

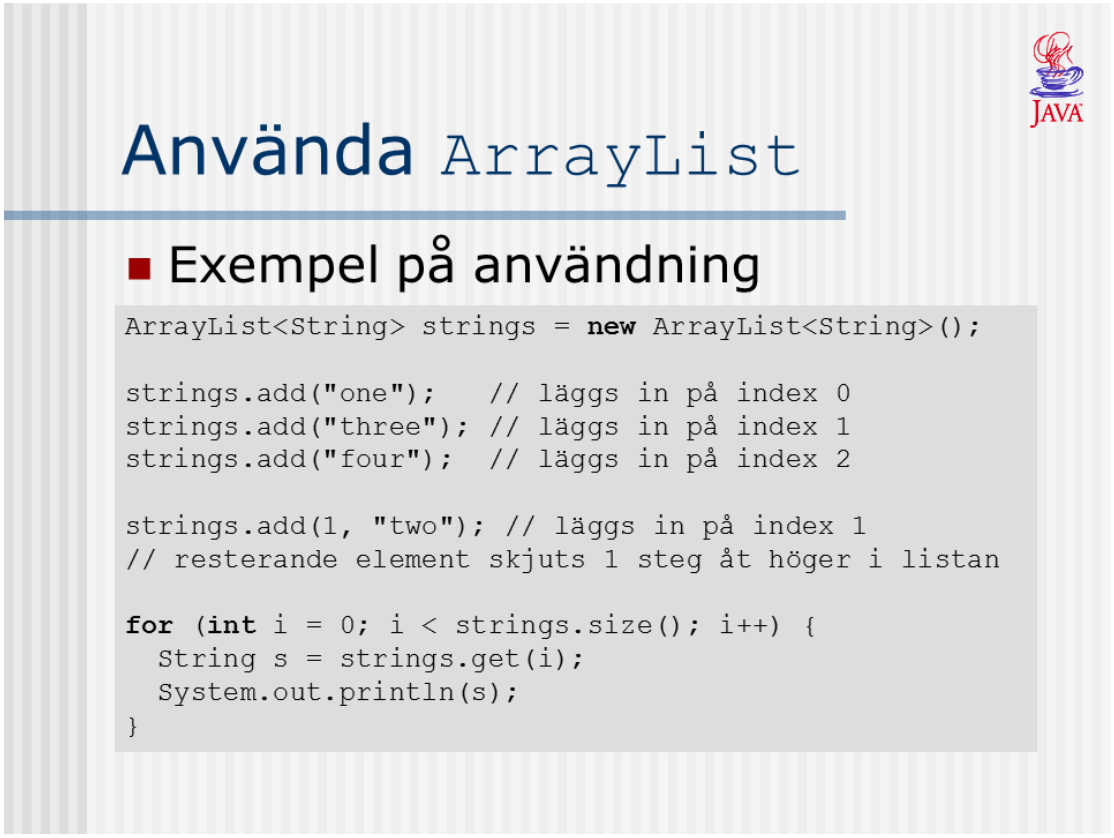
Något viktigt att tänka på när vi skapar en ArrayList är att vi måste ange vilken typ av element vi kommer att lagra i listan. Detta görs genom att ange elementets typ innanför < och >.

Klassen `ArrayList` har ett antal olika metoder med vilka vi kan manipulera listan och dess innehåll. Den mest använda metoden är `add`, vilken vi använder för att lägga till ett nytt element i listan. Metoden finns överlagrad där den första varianten (i bilden ovan) lägger till elementet sist i listan. Med den andra varianten kan vi specificera vilket index i listan vi vill att elementen ska läggas in på (elementen ligger i en `ArrayList`, precis som i en array, på olika index där första elementet i listan har index 0). Om det index vi anger finns mitt i listan någonstans kommer alla element, vars index är lika med eller högre än det vi angav, att skjutas ett steg åt höger i listan (de får sitt index ökat med 1).

För att hämta ett element i listan anropar vi metoden `get` och som argument anger vi det index elementet har i listan. Precis som med en array är det viktigt att vi här anger ett index som är giltigt.

Vill vi ta bort ett element ur listan anropar vi metoden `remove`. Som argument anger vi ett index och elementet på detta index tas då bort helt från listan. Eventuella element till höger om detta element skjuts ett steg åt vänster i listans (de får sitt index minskat med 1). Har vi en objektreferens kan vi även skicka den som ett argument till metoden remove. Om detta objekt finns som ett element i listan kommer elementet att tas bort.

Utöver dessa metoder finns även andra metoder som vi kan anropa för att bland annat ta reda på hur många element listan innehåller (metoden `size`), för att ta bort alla element ur listan (`clear`) och för att skapa en array av elementen i listan (`toArray`).



Exemplet i bilden visar på hur en `ArrayList` kan användas för att lagra element av typen String. Eftersom det är strängar som ska lagras i listan är det viktigt att vi anger typen när listan skapas:

```
ArrayList<String> strings = new ArrayList<String>();
```

Detta är ett annorlunda sätt att skapa objekt på. Vi går inte in mer på varför vi skriver som vi gör i denna kurs utan detta förklaras mer ingående i kursen Java II.

För att sen lägga in några strängar i listan anropas metoden `add`. Första anropet leder till att strängen läggs in på index 0 i listan, andra anropet till att strängen läggs in på index 2 och så vidare. Därefter lägger vi till strängen `"two"` på index 1 i listan. Detta leder till att elementet som redan låg på index 1 skjuts ett steg åt höger och hämnar på index 2. Det element som låg på index 2 hamnar på index 3 och så vidare.

För att loopa igenom alla element anropar vi metoden `size` för att ta reda på hur många element som vi ska loopa igenom. Med anrop till metoden get hämtar vi sen element för element ur listan och skriver ut dem. Notera att i just det här fallet hade vi lika gärna kunna använda en förenklad for-loop för att skriva ut alla elementen:

```
for (String s : strings) {
  System.out.println(s);
}
```

I det sista exemplet skapar vi en `ArrayList` som ska innehålla element av typen `Person`. Vi lägger till tre `Person`-objekt i listan genom att anropa metoden `add` på listan. Därefter används den förenklade for-loopen för att stega igenom alla element i listan. I loopen sätter vi ny ålder på alla personerna (alla blir ett år äldre).

För att ta bort elementet som objektet `dad` refererar till skickar vi `dad` som ett argument till metoden `remove`.

Om vi behöver omvandla listan till en array kan vi anropa metoden `toArray`. Först deklarerar vi då en array av samma typ som den typ vår `ArrayList` innehåller (nämligen `Person`).

Till sist tömmer vi hela listan på dess innehåll genom att anropa metoden `clear` (listans storlek blir då 0).