Mittuniversitetet
MID SWEDEN UNIVERSITY

Lektion 2

Introduktion till Java

Java för C++-programmerare, 7,5 h

Syfte:	Få en förståelse för vad Java är och hur det har utvecklats fram till idag. Ge en snabb översikt över grundläggande begrepp och mekanismer. Presentera olika metoder för att läsa in data från tangentbordet.
Att läsa:	Java Direkt, kapitel 1 - 2 Java Complete, kapitel 1 - 5 Trail: Getting Started https://docs.oracle.com/javase/tutorial/getStarted/ Lesson: Object-Oriented Programming Concepts https://docs.oracle.com/javase/tutorial/java/concepts/

Vad är Java?

På den här kursen ska vi använda Java som programmeringsspråk och några grundläggande verktyg för att skriva ett antal mindre program. Java-plattformen är betydligt mer omfattande än detta. För att få en uppfattning om vilka teknologier som finns att tillgå du kan titta på http://www.oracle.com/technetwork/java/index.html.

Java…

- är ett objektorienterat programmeringsspråk.
- är plattformsoberoende genom teknik med exekvering i en ”virtual machine”. Den kod som genereras av ett Javaprogram kan exekveras på de flesta plattformar (Windows, Macintosh, UNIX m.fl.) via en JVM (Java Virtual Machine).
- stödjer exekverbart innehåll på Internet. Ett Javaprogram (en s.k. applet) kan exekveras som en del av en webbsida.
- har stöd för komponentteknik. Javas komponentteknik heter JavaBeans. En JavaBean är en självständig och återanvändbar programmodul som kan användas i en mängd olika applikationer och som kan manipuleras visuellt i en utvecklingsmiljö.
- har ett stort antal färdiga komponenter i ”packages”. Till Java finns det tillgängligt ett stort antal programbibliotek för vanliga uppgifter som användargränssnitt, nätverkskommunikation, filhantering, datasbashantering m.m. Det gör att en programutvecklare kan använda dessa färdiga paket vid applikationsutveckling.

Ursprungligen utvecklades Java som ett språk för att skriva kompakta och plattformsoberoende program för konsumentelektronik. Inom Sun fanns ett forskningsprojekt (projekt Green) för att enklare kunna programmera inbäddade system med enklare processorer.

De första försöken utgick från ifrån C++, men man fann snart att C++ inte var tillräckligt portabelt mellan olika processorer (inom konsumentelektroniksystem används många olika processorer). Därför utvecklades ett språk i C++:s anda, men som var mer portabelt och också mindre komplex.

Språket, som kallades Oak, hade James Gosling (på dåvarande Sun Microsystems) som sin huvudsakliga skapare och konstruktör. Men Oak var redan upptaget och därför döptes språket om till Java. Var namnet kom ifrån finns det många åsikter om. En del menar att det kom till efter ett besök i en lokal kaffebutik under en fikarast.

Sommaren 1992 skapade projektet sin första produkt, baserad på det nya språket, en extremt intelligent fjärrkontroll. Den var interaktiv, handhållen och kontrollerades genom en tryckkänslig skärm. I en demonstration James Gosling gjorda av Star7 sågs maskoten Duke springa runt och vinka m.m. medan han utförde de kommandon användaren bad om. Se följande film på Youtube där James Gosling pratar om och visar produkten: http://www.youtube.com/watch?v=Ahg8OBYixL0.

Produkten baserades på det nya språket Oak som var processoroberoende. Dock blev produkten ingen succé. Den verkade helt enkelt vara före sin tid.

I samband med att Internet fick allt större uppmärksamhet fick projektgruppen också en idé om att Java borde kunna användas för att utveckla plattformsoberoende program för Internet.

Tekniken prövades till att börja med genom att utveckla en egen webbläsare kallad HotJava, som skrevs helt i Java. Till skillnad från tidigare webbläsare så var Hotjava den första som på riktigt kunde visa animationer och dynamiska sidor. HotJava visades upp på en konferens och i början var det knappt några som rynkade på näsan åt HotJava. Men när James visade upp en sida med en molekyl som man kunde ta tag i och vrida och vända på gick det ett sus genom publiken – det röde på sig. I samband med detta började tekniken med Java att också skapa ett intresse utanför Sun.

Språket offentliggjordes och lanserades officiellt i november 1995, då även Netscape meddelade att man licensierat språket och avsåg att ge fullt stöd för Java i sin webbläsare. I och med detta började det stora genombrottet för Java.

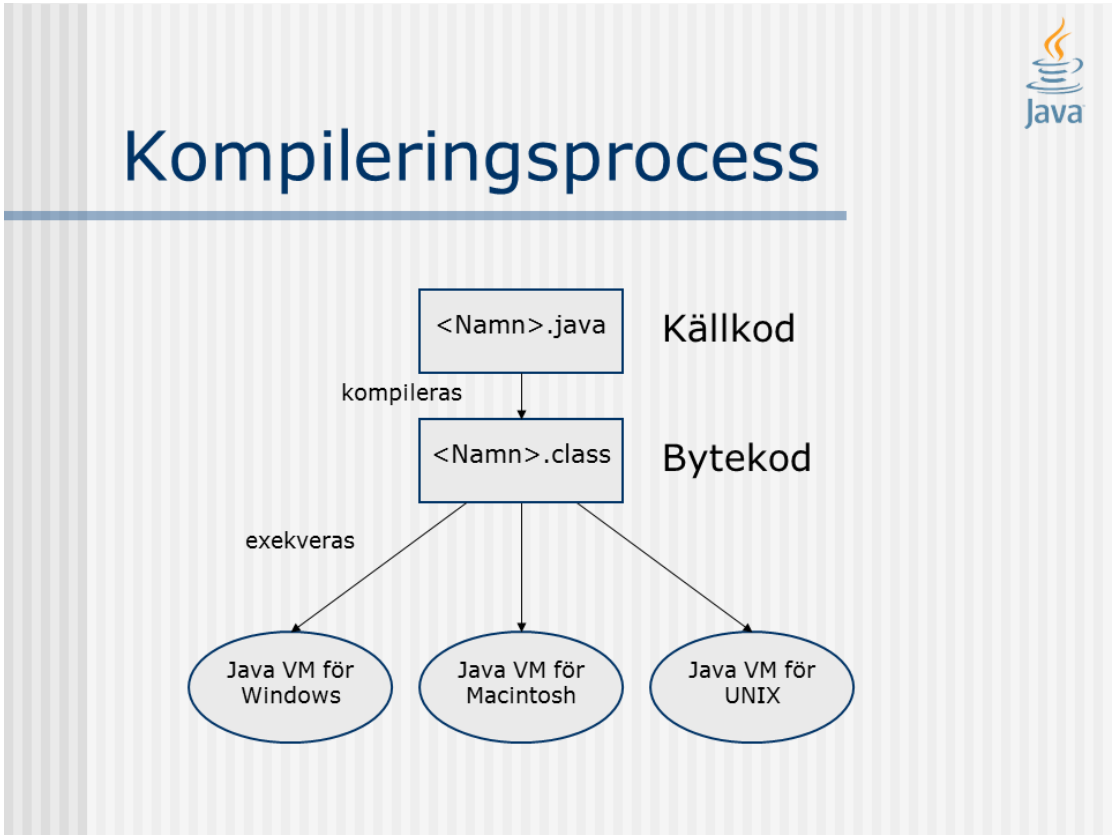
Tidigt bestämde man att Javasstandarden även skulle omfatta ett flertal bibliotek, så att vanliga uppgifter som filhantering m.m. utfärdas på samma sätt oavsett plattform. Genom att tidigt definiera klassbibliotek (paket) och låta dessa vara en del av Javastandarden (till skillnad mot t.ex. C++ som länge saknat standardbibliotek) så kan de flesta program utvecklas med dessa standardbibliotek. Hela tiden utvecklas dessa standardbibliotek och fler bibliotek läggs till i nya versioner av Java. 1997 släpptes v1.1, 1998 släpptes 1.2. I dagsläget är den mest utbredda version 8 (J2SE 8 eller Java 2 Platform Standard Edition 8). I oktober 2017 släpptes den versionen som är version 9.

Java är ett generellt programmeringsspråk som kan användas för alla typer av applikationer.

Läser man om Java i pressen kan man få intrycket av att Java är ett specialskrivet språk för Internet och som endast kan användas i denna miljö, men så är inte fallet. Anpassningen för Internet består av ett fåtal klasser i standardbiblioteken. Det är möjligt att skriva helt vanliga applikationer i Java som inte har någonting alls med Internet att göra.

Java har influerats från många andra språk. James Gosling har sagt att Java är objektorienterat som Smalltalk, numeriskt som Fortran, systemspråk som C/C++, och distribuerat på ett sätt som inget tidigare språk. Syntaxmässigt är språket mest likt C++.

När ett Javaprogram kompileras genereras s.k. bytekod som är ett standardiserat instruktionsfomat för en virtuell Javamaskin. Det är denna virtuella Javamaskinen som är nyckeln till plattformsoberoendet för Java. En virtuell Javamaskin är ett program som är skrivet för en viss plattform (Windows, UNIX, Macintosh etc) och som tolkar bytekod och utför instruktionerna i denna bytekod. I och med att bytekodens format är standardiserat kan alltså en fil med bytekod exekveras på samtliga plattformar som har en virtuell Javamaskin, och bete sig på i princip samma sätt. Det är alltså samma bytekod exekveras men olika virtuella maskiner används på olika plattformar.



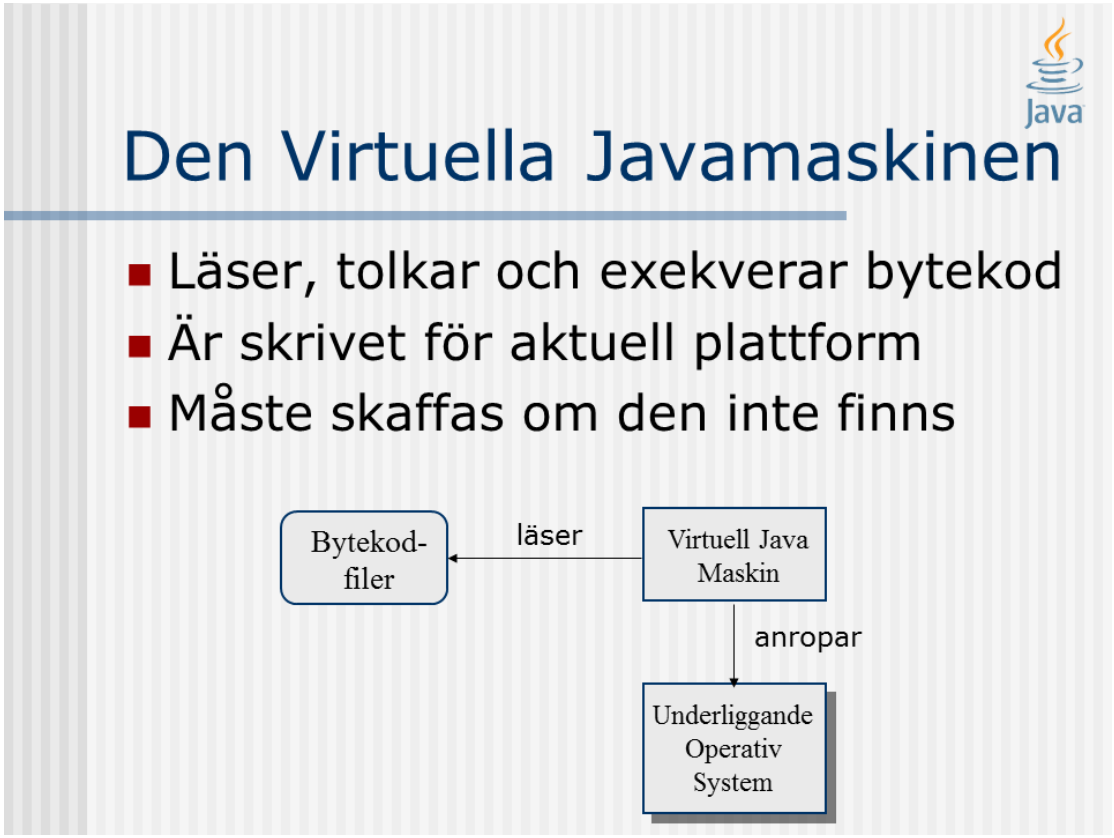
De instruktioner som skrivs sparas i en källkodsfil med filändelsen .java. Innan programmet kan köras måste källkoden kompileras till bytekod (med filändelsen .class). Samma bytekod kan sedan exekveras i olika virtuella maskiner för olika plattformar.

Den grundläggande utvecklingsmiljön för Java, och också den första som släpptes, är Java Development Kit (JDK). Det är en kommandobaserad miljö som innehåller den senaste versionen av Java, kompilator, virtuell maskin och standardbiblioteken för Java. Denna miljö är gratis och finns att laddas ner från www.oracle.com tillsammans med aktuell dokumentation.

Den kommandoradsbaserade kompilatorn i JDK anropas med **javac** och kompilerar java källkod till bytekod. Den virtuella maskinen som exekverar bytekoden anropas med **java** och används för att exekvera den kompilerade bytekoden.

I en kommandobaserad miljö skriver utvecklarna sina Javaprogram i en vanlig texteditor (Notepad, TextPad, UltraEdit). För att sen kompilera och exekvera sina program skriver man kommandon i t.ex kommandoprompten.

I en integrerad utvecklingsmiljö gör man oftast allting i samma program. Man skriver källkoden, kompilerar och exekverar den utan att växla till andra program. Många integrerade miljöer erbjuder även visuell kodning, vilket innebär att mycket av koden kan automatgenereras. För att bygga upp ett användargränssnitt är det i stort sett bara att dra-och-släppa komponenter till en rityta.



Den virtuella Javamaskinen är den komponent som läser, tolkar och exekverar bytekod. Normalt är denna komponent skriven för den aktuella plattformen (operativsystem och hårdvaruarkitektur) som läser bytekoden, tolkar den, och gör anrop mot det underliggande operativsystemet.

Den virtuella maskinen är alltså i högsta grad plattformsspecifik och måste skaffas om den inte redan finns inbyggd i aktuell plattform (i en webbläsare kan man installera en Javamaskin via ett så kallat plugin).

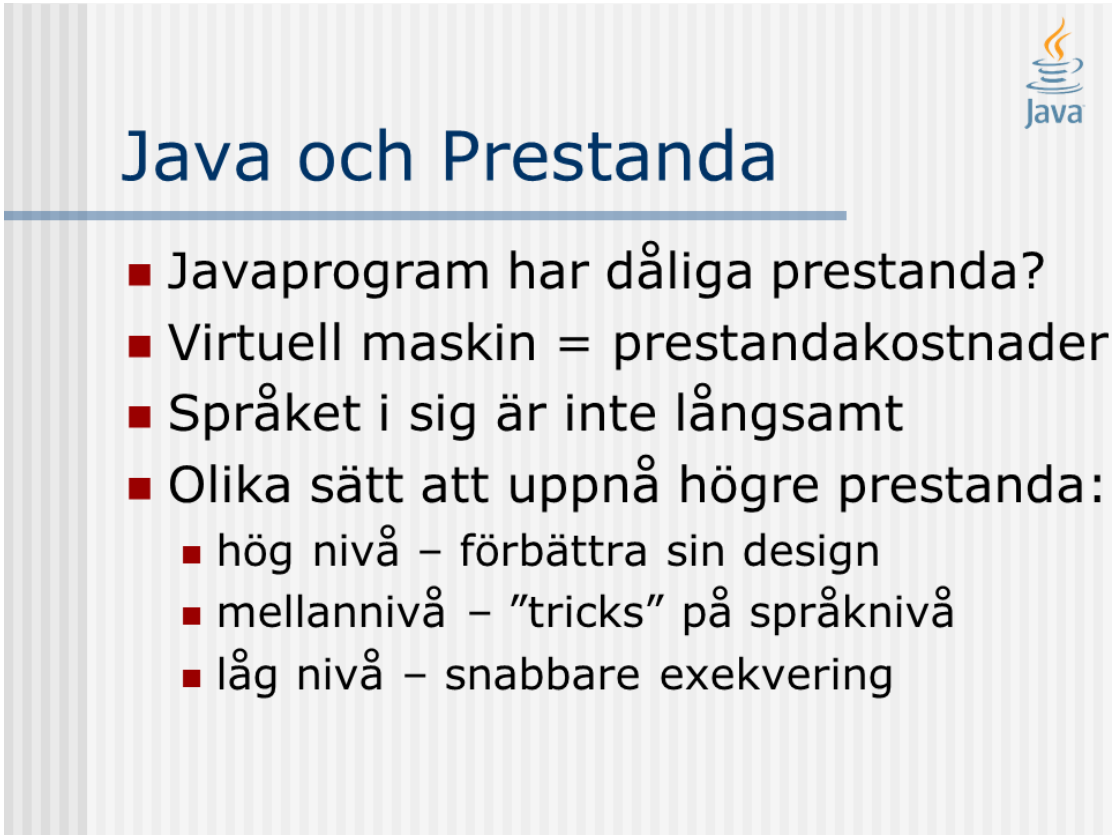
Specifikationen av den virtuella maskinen och formatet på bytekoden är mycket lik en ”virtuell” processor så den virtuella maskinen upprätthåller begrepp som programräknare, stack och cache etc. Den enda skillnaden är att den virtuella maskinen exekverar som ett program.

Både specifikationen av formatet på bytekoden och den virtuella maskinen är helt öppen, och det är teoretiskt möjligt att skriva en bytekod-kompilator för ett annat språk än Java, men detta har visat sig svårare än många trott.

Den virtuella Javamaskinen laddar dynamiskt in ”.class”-filer allt eftersom de behövs i programmet. För att exekvera en applikation anges namnet på den på en klass, varpå dess ”.class”-fil med bytekod läses in och dess main-metod exekveras av den virtuella maskinen. Allteftersom nya klasser refereras i applikationen kommer de att läsas in dynamiskt.

När en ”.class”-fil (Applet) läses från Interner så kommer koden i filen först att passas till en verifierare. Verifieraren är en speciell modul som går igenom filen och konstaterar att koden är ”riktig” och att inga farliga operationer utförs i koden (som att läsa filer på användares dator).

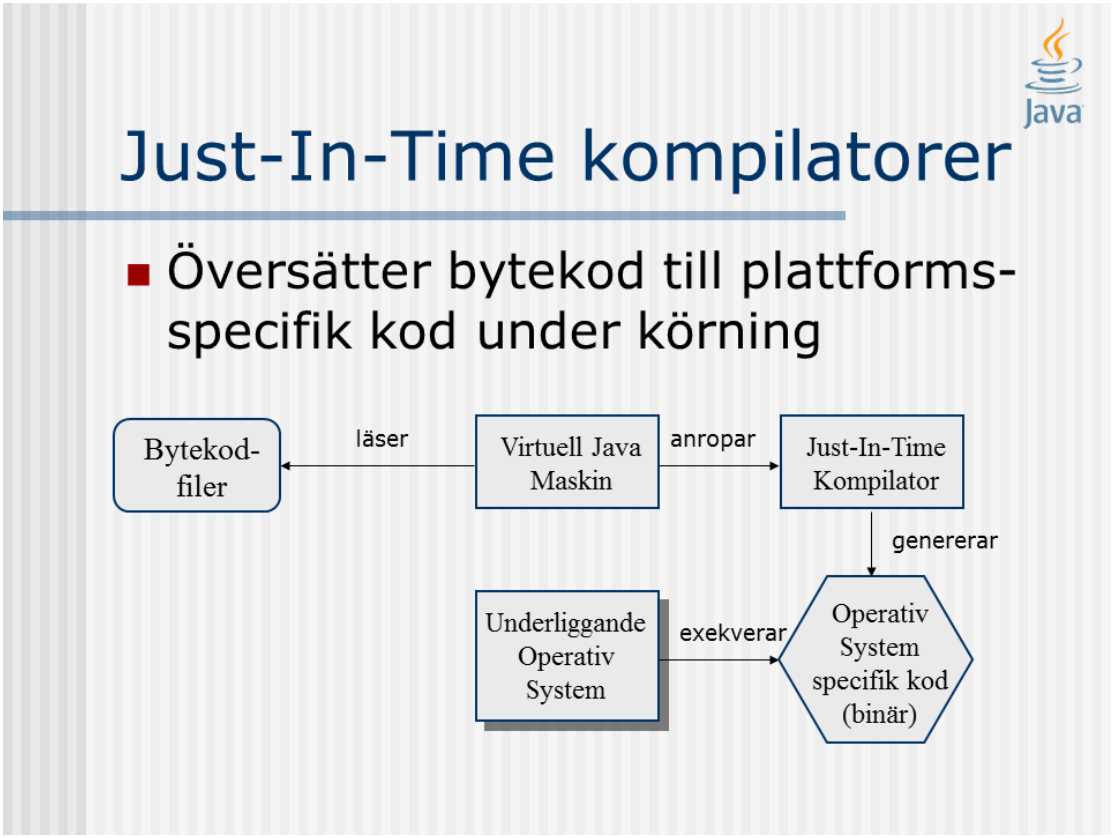
Det finns i JDK (den utvecklingsmiljö vi använder i kursen) ett program, javap, som är en disassembler av bytekod och som visar bytekoden på ett för människor läsbart format. Detta program kan tillsammans med specifikationen av den virtuella maskinen rent teoretiskt användas för att lära sig bytekod-formatet (varför man nu skulle vilja det?). Observera att man med javap INTE får tillbaka den källkod som finns i java-filen. Prova gärna att använda `javap` på någon av de class-filer du gjort för att se vad som händer.



En av de vanligaste invändningarna mot Java är att program har dåliga prestanda. Invändningarna har varit och är fortfarande i vissa fall berättigad. Och det är naturligtvis arrangemanget med en virtuell maskin som i huvudsak skapar dessa prestandakostnader. Den virtuella maskinen kan ju ses som en interpretator som lägger ett extra skal över den slutliga exekveringsmiljön. Språket Java som sådant har inga konstruktioner som gör att det skulle vara långsammare än andra språk.

Att uppnå högre prestanda i ett program, oavsett språk eller miljö, kan angripas på flera olika sätt:
Hög nivå: genom att förbättra sin design och använda mer effektiva datastrukturer och algoritmer. Ofta kan betydande prestandavinster uppnås genom detta, eftersom dålig grunddesign nästan alltid leder till program som är långsamma.
Mellan nivå: genom att använda diverse ”tricks” på språknivån som att deklarera variabler som konstanta (static), klasser och metoder som inte får ärvas eller omdefinieras (final), skapa få och stora klasser. Eller genom att använda shift-operatorerna vid multiplicering och division.
Låg nivå: genom att se till att den genererade koden kan optimeras och exekveras snabbare, dvs. att den fysiska exekveringen av programmet utförs snabbare på den aktuella plattformen.

De nivåer som rekommenderas i första hand är den höga och den låga. Att använda tekniken på mellan nivån bryter mot grundläggande principer för god programmering, och ger program som är svåra att underhålla, som inte är lika utbyggbara och lätta att förändra.



Vi ska nu titta på hur Java fått bättre prestanda genom att på låg-nivå förändra hur Javaprogram exekveras.

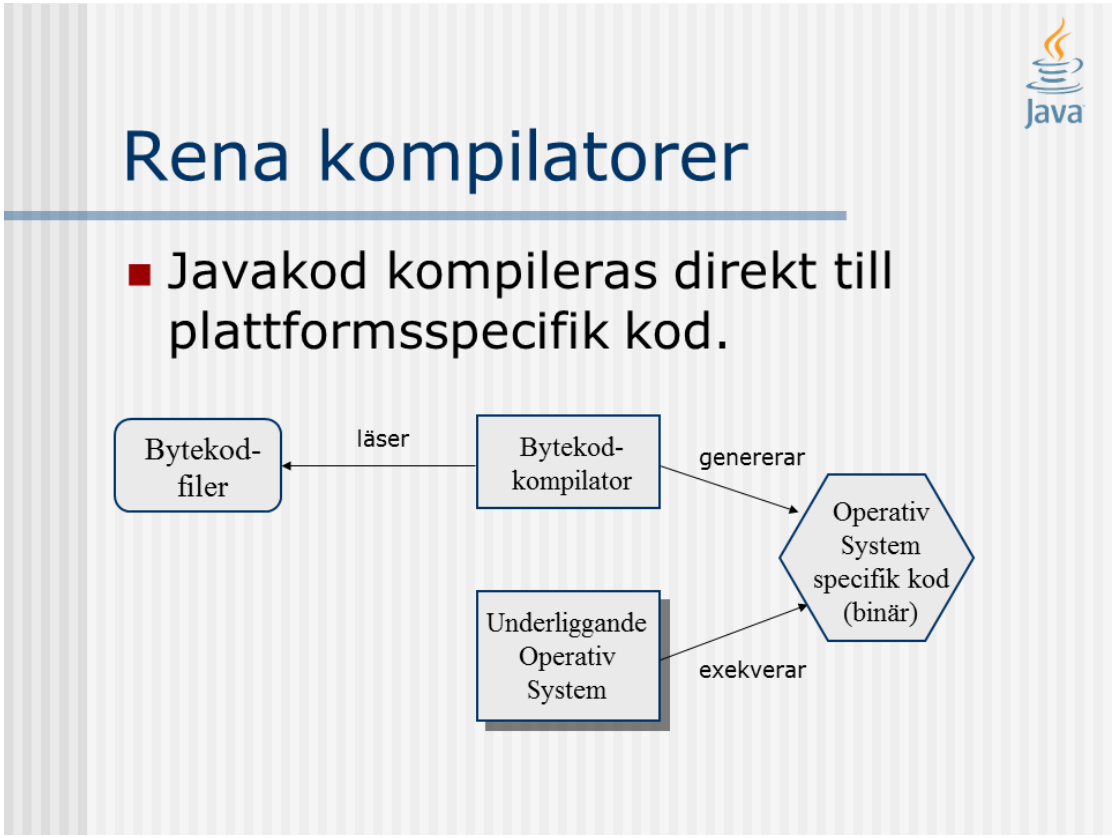
En Just-In-Time (JIT) kompilator är en kompilator som översätter bytekoden till plattformsspecifik kod som sedan exekveras på aktuell plattform (likt en kompilator av exempelvis C++).

Kompileringen görs dock dynamiskt vid exekvering av ett program, där JIT-kompilatorn kopplas på den virtuella Javamaskinen. Det är alltså bytekoden som kompileras, inte den ursprungliga Javakällkoden.

Den virtuella Javamaskinen är inte medveten om JIT-kompilatorn, och dess vanliga säkerhetskontroller av bytekoden kommer fortfarande att utföras.

JIT har diverse tricks för sig för att snabba upp exekveringen. Bl.a. kommer den att spara genererad kod för metoder så att själva översättningen från bytekod till plattformskod endast sker en gång. Den gör enkla former av optimering av den genererade koden. Den kan tyvärr inte göra mer avancerad optimering eftersom den helt enkelt inte har tid – översättningen sker ju i samband med att koden ska exekveras (just in time).

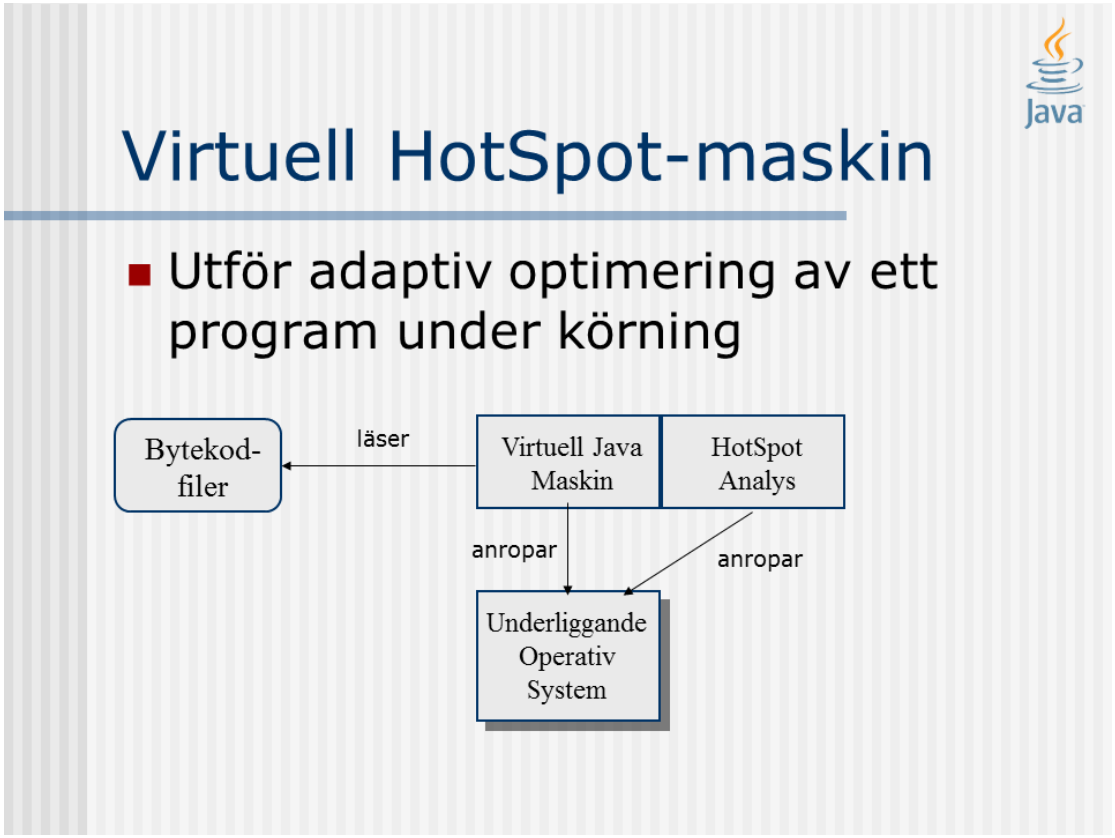
Mätningar har påvisat stora effekter på exekvering av Javaprogram. Exakta måtten varierar, men alltifrån:
3-4 ggr snabbare när användargränssnittskod exekveras
till 40-50 ggr snabbare för beräkningar, loopar och upprepade metodanrop.



Ingenting förhindrar att Javakod kompileras direkt till plattformsspecifik kod, dvs. att en ”ren” kompilator av källkod till exekverbar fil på aktuellt plattformsformat. Olika leverantörer av utvecklingsmiljöer har också lanserat sådana kompilatorer. En utvecklare kan då välja att antingen få bytekod genererad eller att få en plattforms-specifik exekverbar fil (i Windows en exe-fil).

Med en ren kompilator kan också mer avancerad optimering göras eftersom denna optimering då görs i samband med kompileringen, och det finns då hur mycket tid som helst till detta. Vissa hävdar att sådana kompilatorer ger program med jämbördiga prestanda som exempelvis kompilerad C++ kod.

Vi avråder dock kraftigt mot denna lösning eftersom den bryter mot Javas plattformsoberoende.



HotSpot är en teknik för att implementera virtuella maskiner som bygger på något som kallas adaptiv optimering.

Den virtuella maskinen analyserar varje Javaprogram när det exekverar, och använder denna information till att omedelbart optimera de kritiska s.k. ”hot spots” som ett program har (de områden i programmet där mest datorresurser förbrukas). Genom att optimera endast dessa kritiska områden frigörs tid att göra mer avancerade optimeringar.

Med adaptiv optimering kan bättre optimeringar göras än med en ”ren” kompilator, eftersom mer information finns om programmets exekvering (en ”ren” kompilator studerar endast källkoden före programmets exekvering).

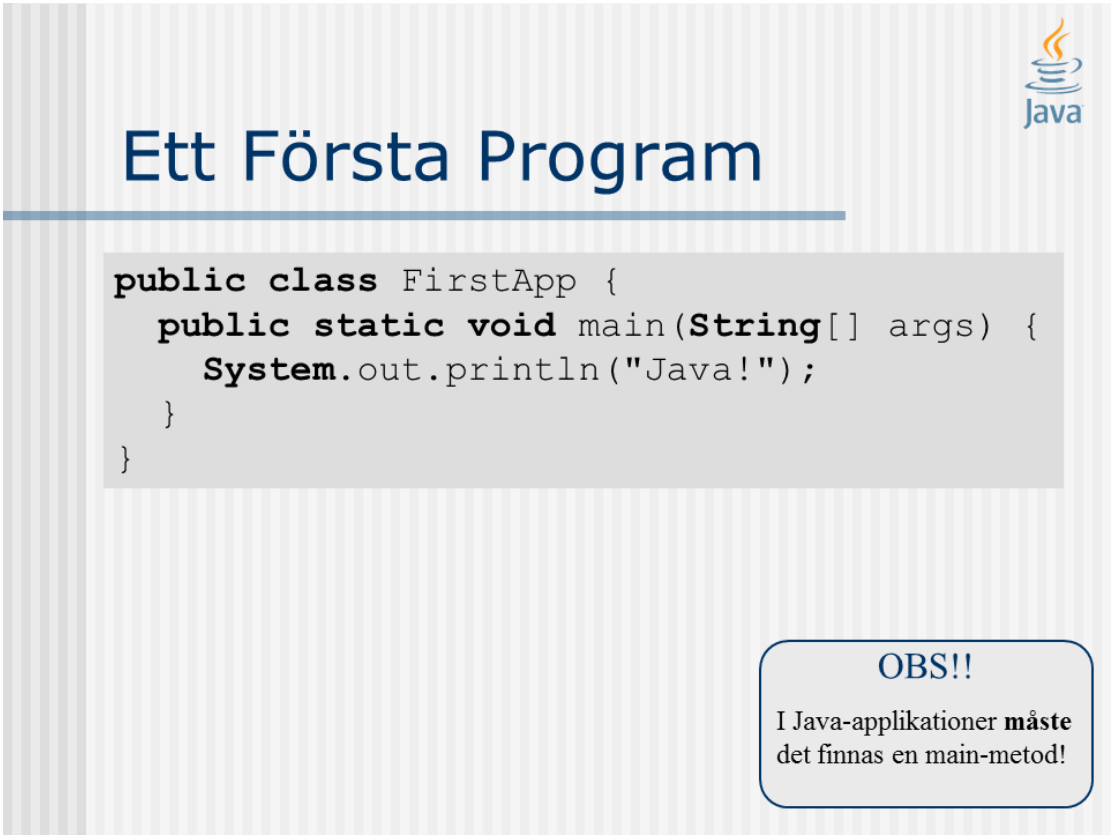
Ett Javaprogram definieras i en eller flera källkodsfiler, som sedan kompileras till s.k. bytekod som den virtuella Javamaskinen kan exekvera.

Enligt Javadefinitionen är det teckensnitt som används Unicode, vilket innebär att tecken som å, ä, ö (t.o.m. specialtecken) är tillåtna i namn på identifierare som variabler och klasser. Trots detta uppstår ofta problem när åäö används i identifierare, framför allt när källkoden flyttas mellan olika plattformar. Därför rekommenderas du att undvika åäö i namn på klasser och variabler. Det rekommenderade språket att använda för att namnge sina klasser, metoder och variabler är engelska.

Ett Javaprogram finns i två olika former: applikationer och applets.

- En Java-applikation är ett program som exekveras fristående ifrån andra program (men som naturligtvis kräver en virtuell Javamaskin på den aktuella plattformen).

- En Java applet är ett program som alltid exekverar inifrån ett program, normalt en webbläsare som Firefox, Chrome eller Internet Explorer. Java applets är formen för att distribuera Javaprogram över Internet eller ett intranät.



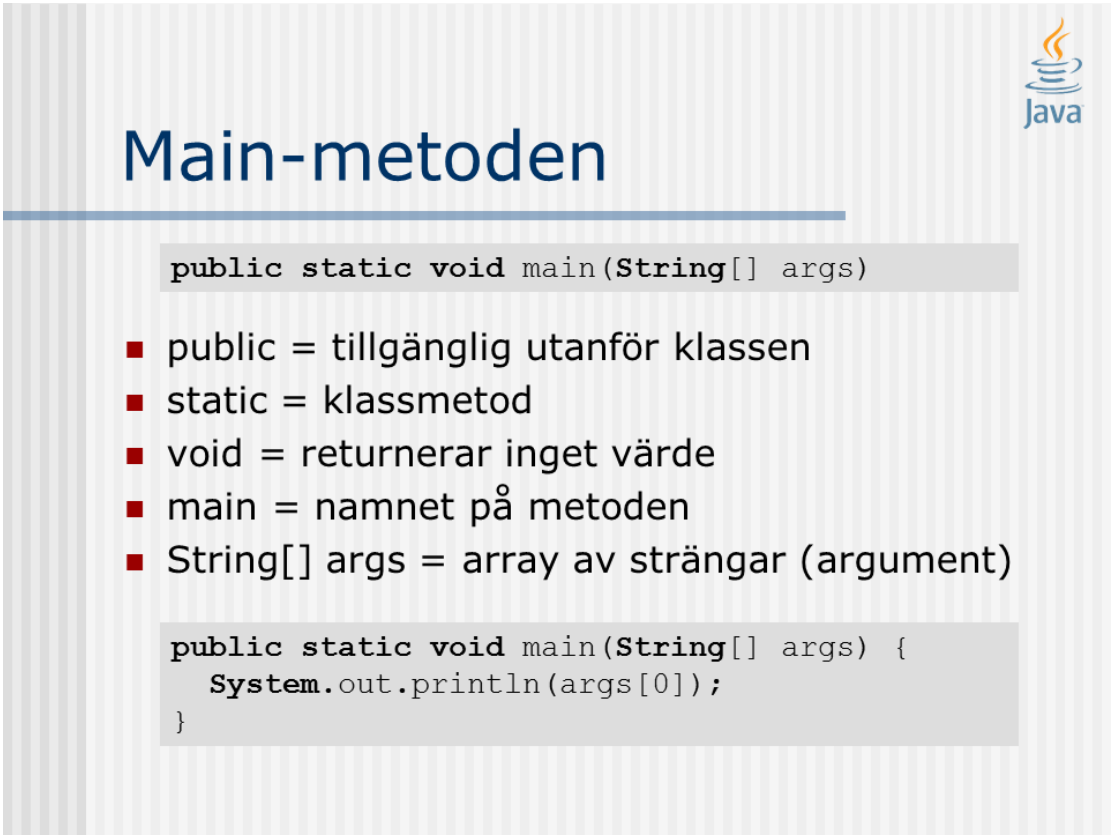
Alla Javaprogram bygger på klasser så även detta minimala program innehåller en klass. Klassen heter FirstApp och den fil i vilket källkoden ligger heter då FirstApp.java (Java skiljer på små och STORA bokstäver vilket är viktigt att komma ihåg!).

Programmet börjar med en klassdeklaration i form av nyckelorden public class följt av namnet på klassen. Klassdeklarationen är innesluten i matchande klammertecken { och } inom vilka klassens medlemmar (instansvariabler och metoder) finns. I detta exempel har klassen inga instansvariabler utan endast en metod, den s.k. Main-metoden som är den metod i vilket exekveringen av en Java-applikation startar.

Denna main-metod (eller metod deklarationen) är precis som klassen innesluten i matchande klammertecken. I det här fallet innehåller main endast en sats, och här används en klass System från paketet java.lang. Objektet out är av klassen PrintStream vilket är en klass för att skriva meddelanden. På out anropas metoden println som skriver ut en variabel (i detta fall en sträng). Detta kan låta krångligt vilket är förståeligt. Det räcker för tillfället att veta att en programsats av typen System.out.println skriver till konsolfönstret (standard ut).

När main-metoden är slut är också programmet slut, och programmets exekvering avslutas.

Vill du prova detta programexempel finner du källkod och bytekod i exemplen som tillhör lektionen.



Main-metodens definition kan till en början se väldigt komplex ut. De olika delarna kommer vi att gå igenom allt eftersom. Nu från början så är det här ytterligare en sak du bara får acceptera att så här skriver man.

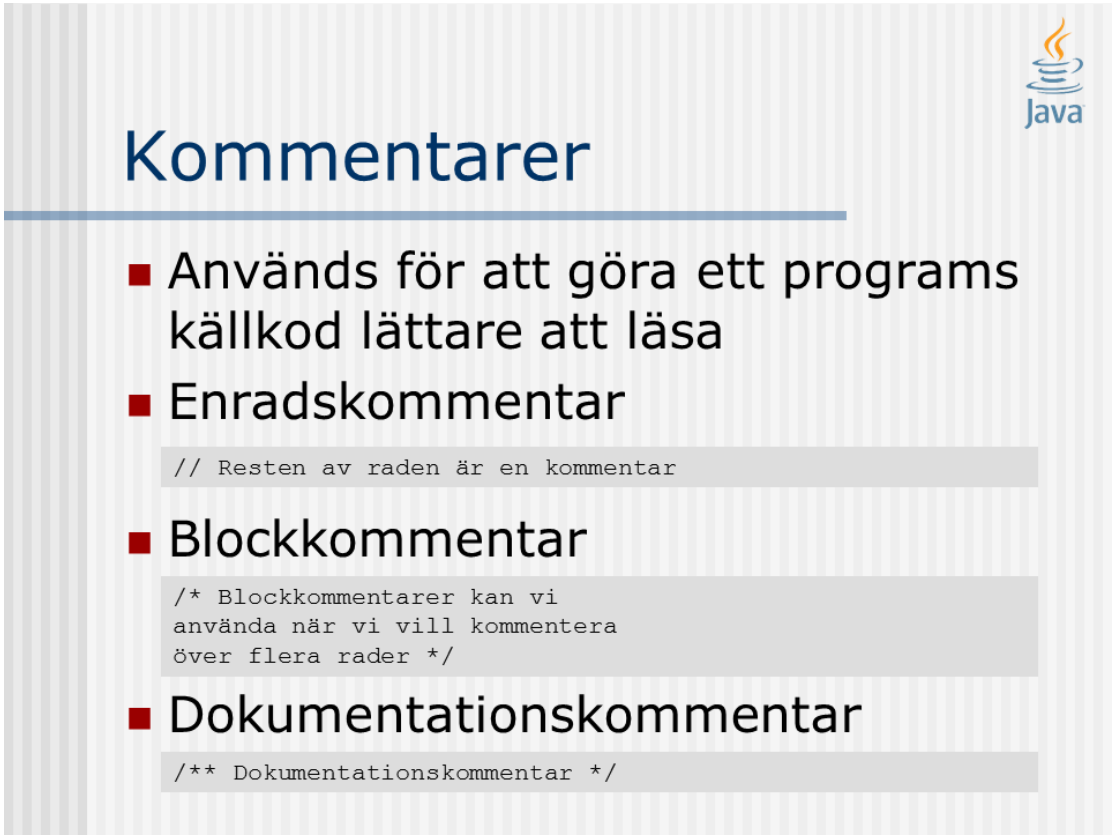
public innebär att metoden är tillgänglig utanför klassen, vilket main-metoden naturligtvis måste vara om programmet ska kunna köras igång.

static innebär att det är en statisk s.k. klassmetod, en metod som kan anropas direkt på klassen utan att något objekt av klassen har skapats. Att main-metoden är statisk är naturligt eftersom något objekt inte kan skapas före main-metoden körs.

void innebär att main-metoden inte returnerar något returvärde. Normalt så returnerar en metod ett värde när den har exekverat alla satser som tillhör metoden. När main metoden är klar så avslutas programmet och det finns därför ingen anledning till att returnera något. main är namnet på metoden.

String[] args är en array av strängar döpt till args. Denna array kommer att innehålla de eventuella argument som angetts på kommandoraden till programmet. Man kan alltså skicka in vissa värden som programmet kan tänkas behöva när vi startar det. För att skriva ut det första argumentet som skickats in till programmet kan koden längst ner användas. Startar vi programmet enligt följande: java MittProgram hejsan så är det hejsan som skrivs ut på skärmen. Har vi fler argument så kommer vi åt dessa genom args[1], args[2] osv.

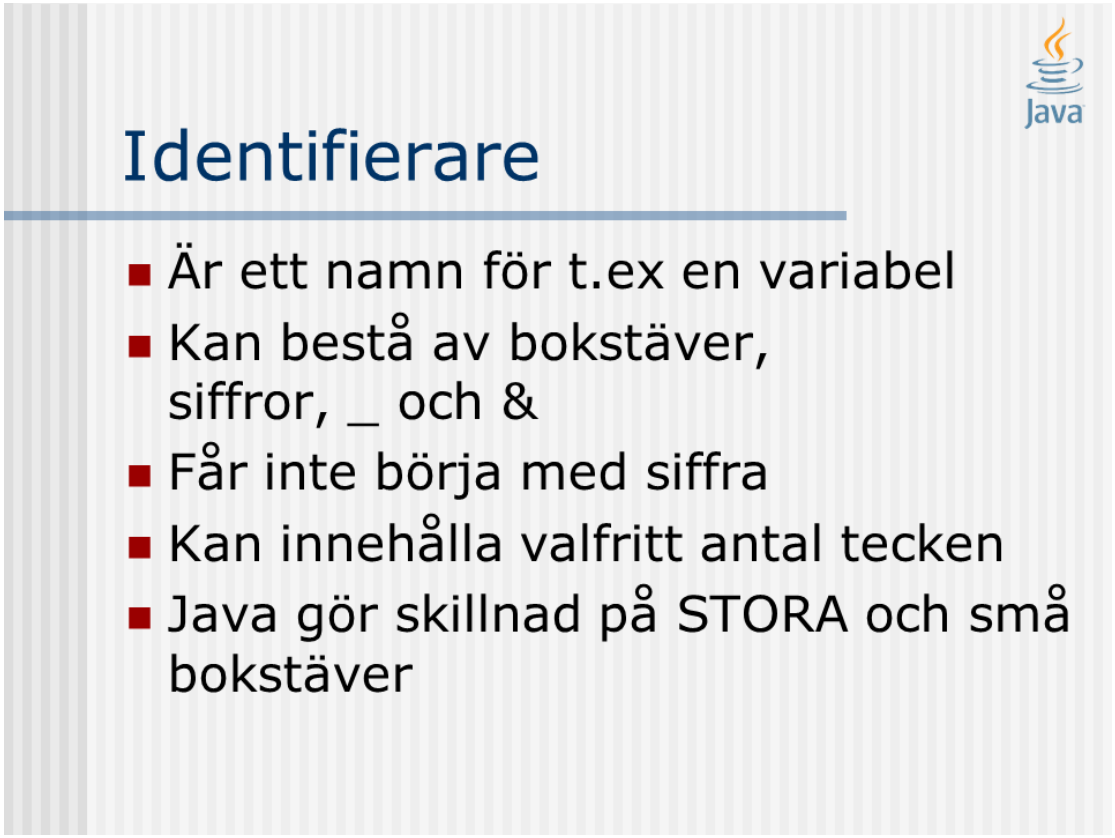
Vill du prova detta exempel kan du kompilera och köra Argument.java (se exemplen).



Oavsett vilket programmeringsspråk som används måste ett program kommenteras för att vara läsbart. Kommentarer kan också vara bra för att man som programmerare själv ska komma ihåg vad man har gjort för något. Det kommer visa sig att du glömmer otroligt fort vad det är du har gjort.

I Java finns det tre olika typer av kommentarer:

- Enradskommentarer som börjar med // och löper till radens slut.
- Blockkommentarer som kan löpa över flera rader. Börjas med /* och avslutas med */. Är väldigt användbart när man vill kommentera bort vissa rader i koden vid felsökning m.m.
- Dokumentationskommentarer används för att automatgenerera dokumentation i Java. Verktyget javadoc används för detta ändamål och genererar dokumentationen i HTML-format. Mer om javadoc kommer senare i kursen.



En identifierare kan ses som ett namn på ”saker” där saker är variabler, metoder eller klasser.

I Java (och många andra språk) kan en identifierare bestå av en blandning av bokstäver, siffror, understreck (_) och tecknet &.

I Java finns det ingen begränsning vad gäller antalet tecken, dock får en identifierare inte börja med en siffra.

Viktigt att tänka på är Java skiljer på STORA och små bokstäver. T.ex. är identifieraren antal inte samma som identifieraren Antal.

Exempel på identifierare vi hittills har stött på är FirstApp som identifierar vår första klass. Dessutom har vi stött på main som identifierar metoden med samma namn i klassen FirstApp.

Enligt Javas kodkonventioner för namngivning av paket, klasser, metoder och variabler (http://www.oracle.com/technetwork/java/codeconventions-135099.html) ska följande följas:

Klasser inleds alltid med stor bokstav. Efterföljande ord inleds också med stor bokstav. Metoder inleds alltid med liten bokstav. Efterföljande ord inleds med stor bokstav. Variabler skrivs på samma sätt som metoder.

Samtliga standarbibliotek ligger i olika paket och har namn som java.lang (klasser för de mest grundläggande funktioner i Javaspråket), java.util (stödklasser för listor, hashtabeller, datum, tid m.m.), java.io (klasser för fil och I/O-hantering), java.net (klasser för nätverkskommunikation och Internetklasser), javax.swing (klasser för att skapa grafiska användargränssnitt).

En utvecklare kan också skapa egna klassbibliotek och lägga in dem i paket. Paket lagras i en katalogstruktur baserad på paketets namn.

När klasser ifrån ett paket ska användas så importeras klasser ifrån paketet genom en import-sats. Antingen kan alla klasser i paketet importeras, eller så kan endast de klasser som verkligen ska användas importeras. En import av ett paket gör inte vår klass storleksmässigt större, utan en import gör det möjligt för oss att komma åt och använda klasser i paketet som importeras.

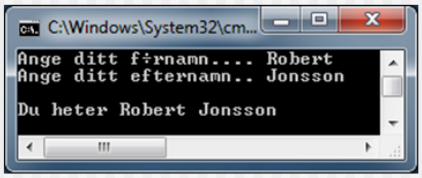
Speciella synlighetsregler finns som bestämmer vad som ska vara åtkomligt inom och utanför ett paket. Man kan alltså mer eller mindre bestämma vilka (tolka inte vilka som vilka olika användare) som ska kunna använda klasserna i paketet.

Läsa Från Tangentbord

- Alternativ 1
  - från kommandoprompten
- Använder ett objekt av klassen BufferedReader

```
BufferedReader input = new BufferedReader(new
  InputStreamReader(System.in));

String answer = input.readLine();
```



Lugn!
Det är inte meningen ni ska behöva förstå denna kod (än)

I stort sett alltid måste program ha indata av något slag som programmet sen ska bearbeta. Hittills i våra program har vi endast behandlat utdata med hjälp av `System.out.println`. När ett program behöver indata av någon form är det vanligtvis användaren som får mata in data från tangentbordet. Det kan t.ex. ske genom att mata in text i olika textfält. De två sätt vi kommer att jobba med i denna kurs är genom att mata in text direkt från kommandofönstret och genom dialogrutor.

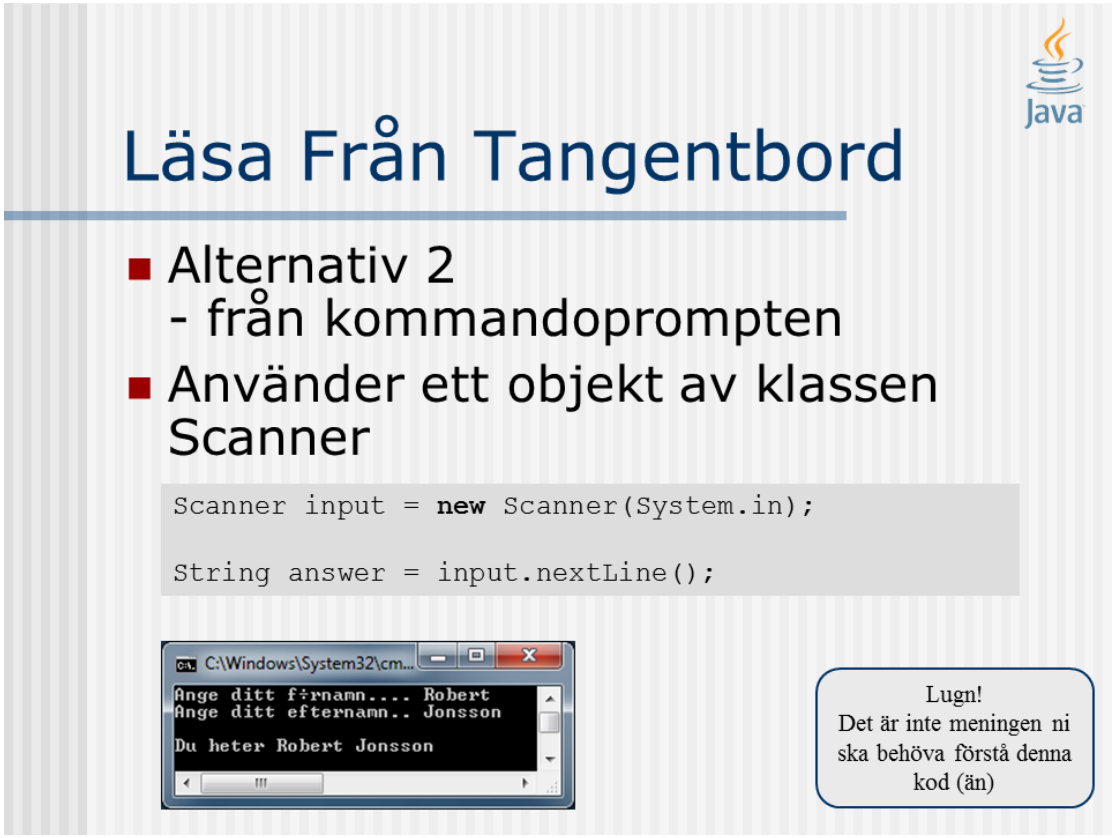
Oavsett vilket sätt vi använder oss av så är det en komplicerad process. Att exakt förklara hur koden som används fungerar är något som vi inte klarar av med vad vi hittills gått igenom. Detta är ytterligare exempel på kod som du får acceptera som den är.

Det första alternativet är genom att använda ett objekt av klassen `BufferedReader` som läser de tecken som matas in från tangentbordet. All in- och utmatning (I/O) i Java kan generera fel av något slag, så när vi använder denna metod måste vi antingen fånga felet (catch) eller kasta det vidare (throw). Detta är exempel på s.k. felhantering som vi inte kommer att gå igenom riktigt än.

Javas hantering av undantag är hämtad från C++ så du kommer att känna igen dig väl. En skillnad i exception-hanteringen mellan C++ och Java är dock att i Java är man **tvungen** att skriva try/catch –block om man anropar en metod som kan generera ett exception. **Alternativt** kan man kvalificera metoden där man gör anropet med nyckelordet throw. Detta innebär att ett exception pga. av ett anrop i den aktuella metoden inte hanteras där utan skickas vidare till nästa metod uppåt i anropskedjan.

Med hjälp av identifieraren input kan vi nu läsa in i programmet det användaren skriver. Detta gör vi genom att anropa metoden `readLine`. Resultatet sparar vi i en sträng som vi gett namnet svar.

Ta en titt på exemplet **Input1.java** för att testa inmatning med BufferedReader från kommandofönstret.

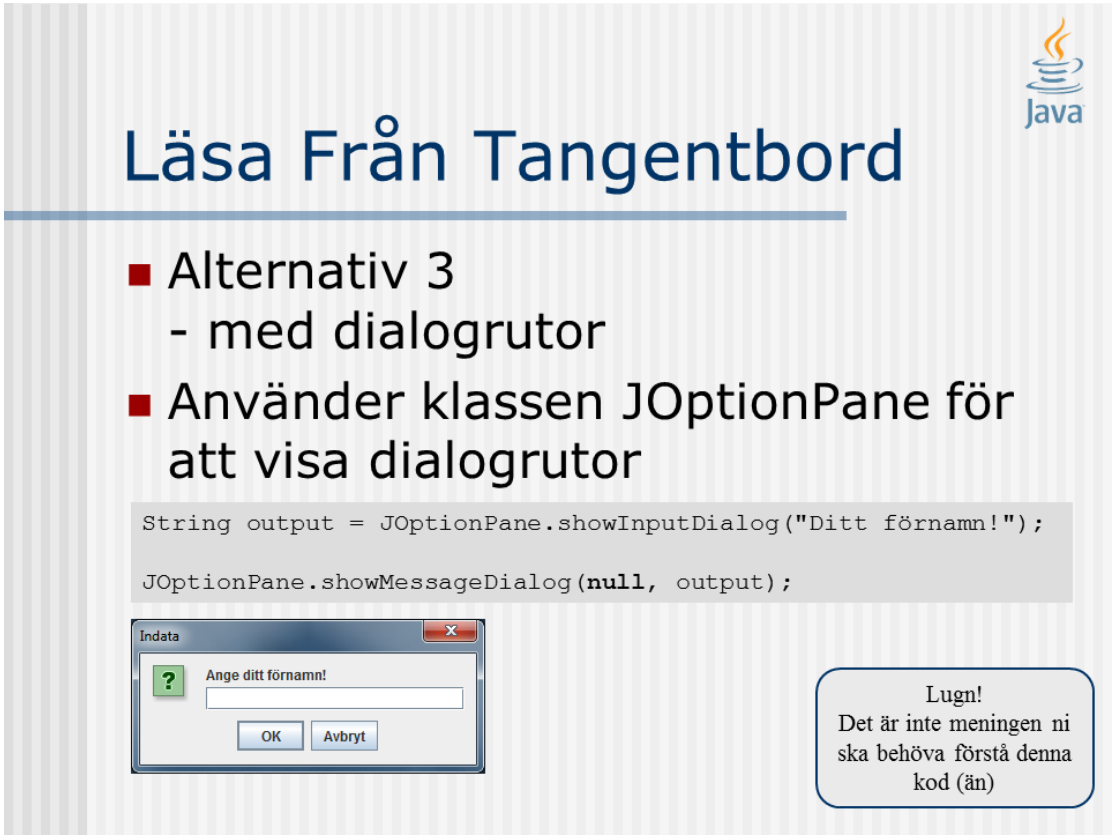


En annan klass som ingår i paketet `java.util` är klassen **Scanner**. Scanner presenteras först i kapitel 5 i läroboken men är så användbar att vi tar upp den redan här. Med en Scanner kan vi läsa in data på ett något enklare sätt än med BufferedReader. För att använda klassen Scanner i våra program 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 radbryts-tecken. Eftersom ett tal inte kan innehålla tecken annat än siffror läses inte radbryts-tecknet av Scanner. Därför måste du alltid göra ett extra anrop till `nextLine()` när du läst ett tal och direkt efter vill läsa en sträng. Titta på exemplet **ScannerTest.java** där Scanner används för att både läsa data från tangentbordet och från fil. Använder även metoden format för att formatera utskriften. Glöm inte att konsultera API-Dokumentation och Skansholms bok.

Ta en titt på exemplet **Input2.java** för att testa inmatning med Scanner från kommandofönstret samt ScannerTest.java.



Det andra alternativet är att använda klassen `JOptionPane` för att visa grafiska dialogrutor där användaren kan göra sina inmatningar. Med `JOptionPane` kan vi även skriva ut ett resultat i en dialogruta, istället för att skriva ut det på skärmen med `System.out.println`.

I klassen `JOptionPane` finns det ett antal fördefinierade dialogrutor vi kan använda i vårt program. En av dessa kommer vi åt genom att anropa metoden `showMessageDialog()` på klassen `JOptionPane`. Denna metod kräver två argument (indata). Det första är var på skärmen dialogrutan ska visas. Här anger vi null vilket innebär att rutan kommer att centreras på skärmen. Det andra argumentet anger vilken text som ska visas i dialogrutan. Ex:

```
JOptionPane.showMessageDialog(null, "Snart är det julafton");
```

Med en annan fördefinierad dialogruta kan vi låta användaren mata in text från tangentbordet som vi sen kan ta reda på. Denna dialogruta kommer vi åt genom att anropa metoden `showInputDialog()`, också i klassen `JOptionPane`. Denna metod tar ett argument vilket är en sträng med den text som ska visas i dialogrutan (frågan/uppmaningen till användaren).

När användaren har skrivit in sin text och tryckt på knappen Ok (eller trycker enter) så returneras det inskrivna som vi kan ta emot och lagra i en sträng.

Ex)
```
String svar = JOptionPane.showInputDialog("Vilken veckodag är det?");
```

När vi har ett program med ett grafiskt användargränssnitt är det viktigt att vi avslutar programmet genom att skriva `System.exit(0);` Argumentet 0 till metoden exit anger att programmet avslutades på ett korrekt sätt. Detta värde returneras till kommandopromten varifrån vi startade vår applikation. Om vi vill kan vi ange olika felkoder istället för 0 om något fel har inträffat i programmet.

Ta en titt på programmet **Input3.java** i exemplen för att se hur dialogrutor kan användas för in- och utmatning av data.

Java har **fyra** stycken möjliga åtkomstformer till instansvariabler och metoder:

- Publik åtkomst (`public`)
- Skyddad åtkomst (`protected`)
- Privat åtkomst (`private`)
- Paketåtkomst.

Java skiljer sig alltså från C++ endast genom paketåtkomst. Om inget av nyckelorden `public`, `protected` eller `private` anges framför ett instansvariabel eller en metod så får den som default paketåtkomst. Det finns inget nyckelord för att uttryckligen ange paketåtkomst. Med paketåtkomst gäller följande regler för dessa klassmedlemmar:

- De är åtkomliga inom det paket i vilken klassen ingår
- De är inte åtkomliga utanför det paket i vilket klassen ingår

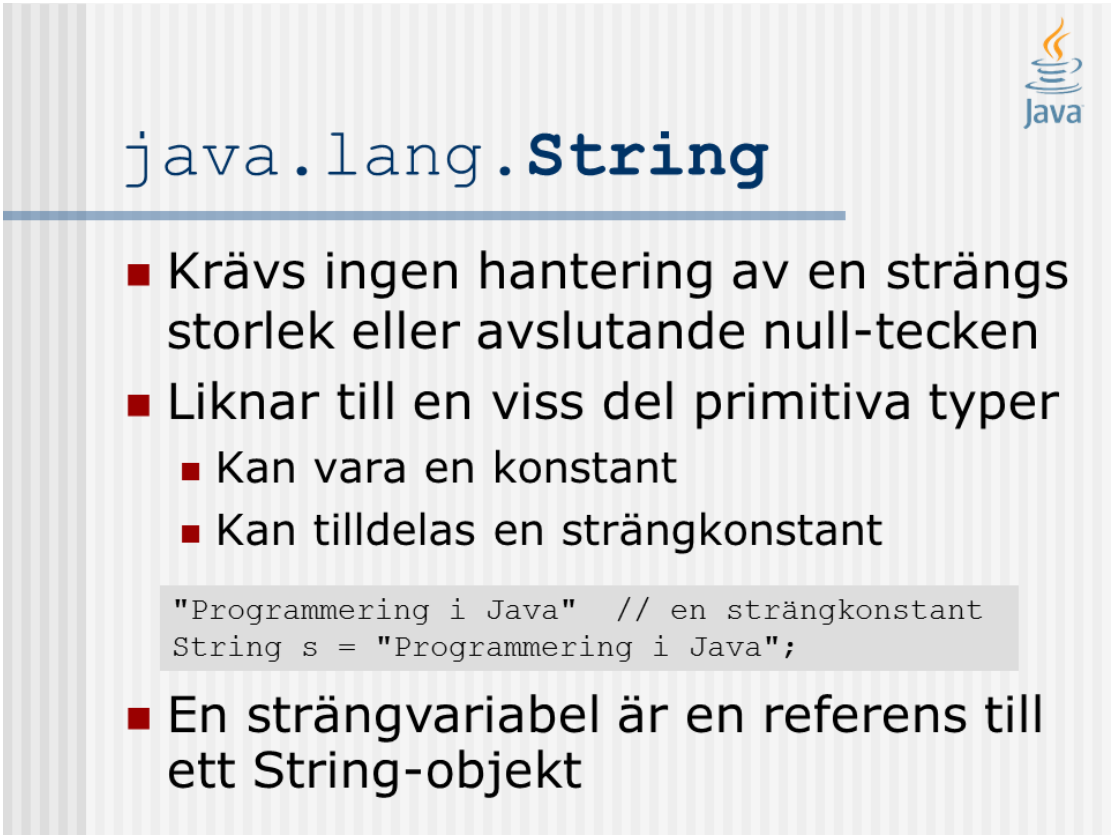
Generellt gäller att man inte bör använda sig av paketåtkomst utan alltid skriva om en medlem ska vara `public, protected eller private` för att klart ange vad som gäller.

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 strängen innehåller. Arrayer behandlas i en senare lektion.

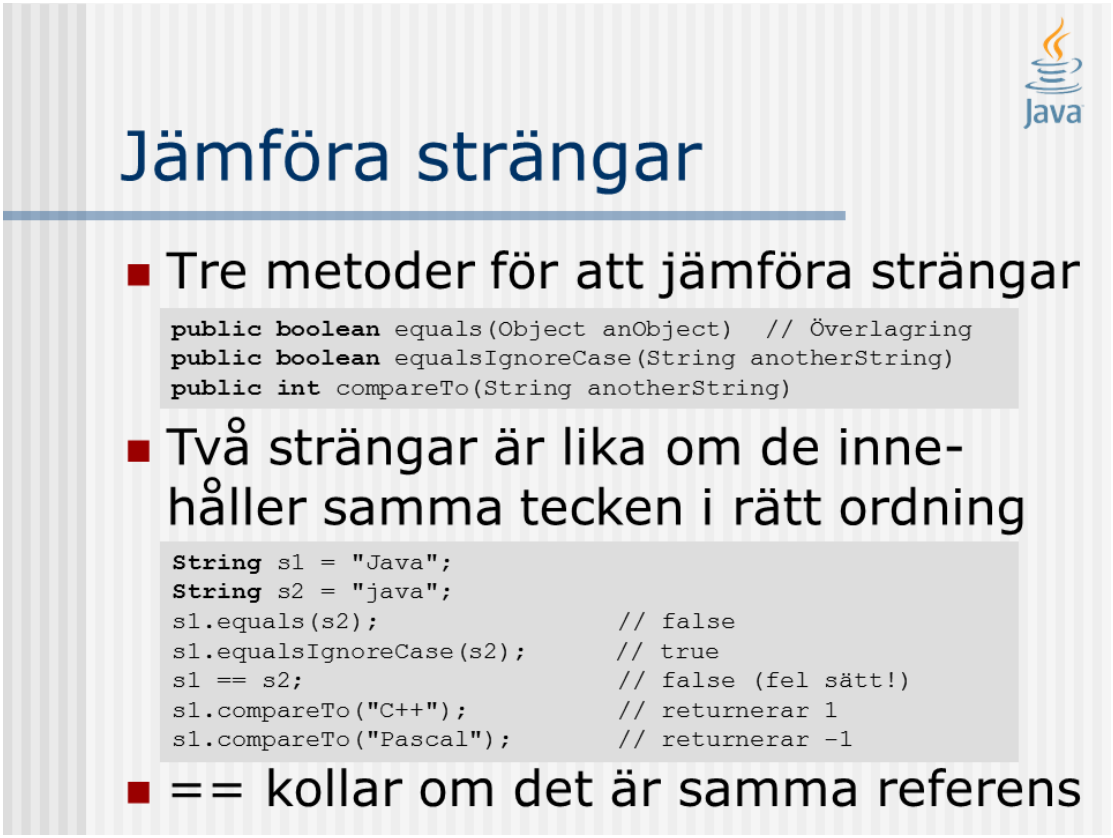


I och med att en sträng i Java endast hanteras av en speciell klass behöver vi inte ta hänsyn till att hålla reda på strängens storlek eller att avsluta den med `null`-tecken (som man t.ex. måste göra i C++ när vi håller på med tecken-vektorer).

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.



Java erbjuder tre olika metoder för att jämföra strängar med varandra.

`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 **Equals.java** används metoden `equals` och `qequalsIgnoreCase` för att jämföra två strängar med varandra.