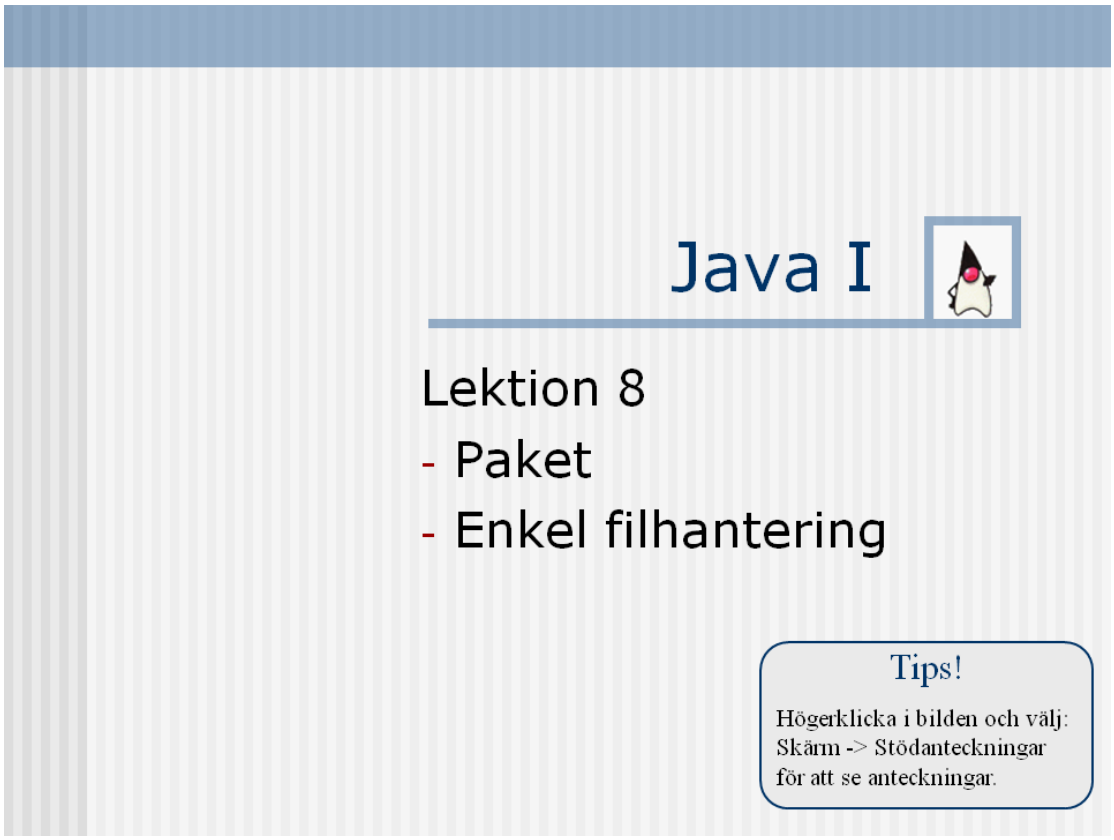
Mittuniversitetet
MID SWEDEN UNIVERSITY

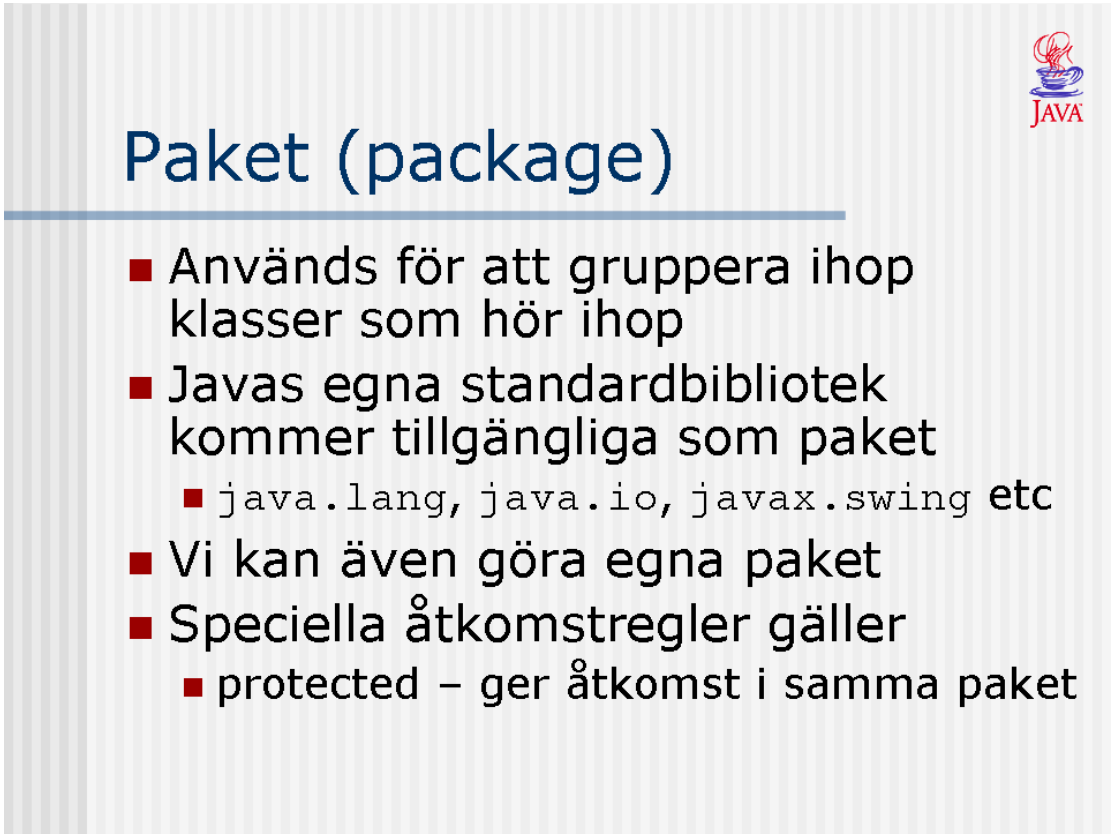
Lektion 8

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

Syfte:	Att känna till vad paket är och hur vi gör för att deklarera våra klasser att tillhöra ett paket. Att känna till vad en I/O-ström i Java är och hur olika klasser kan användas för att läsa från och skriva till filer.
Att läsa:	Kursboken, Kapitel 3.1 (Paket) Kursboken, Kapitel 5.1 – 5.5 (Läsning och skrivning) The Java™ Tutorial, Package https://docs.oracle.com/javase/tutorial/java/package/ The Java™ Tutorial, I/O Streams https://docs.oracle.com/javase/tutorial/essential/io/streams.html



I kursens sista lektion ska vi ta en närmare titt på hur vi skapar egna paket och använder oss av dessa, samt en väldigt kort introduktion till hur vi använder s.k. strömmar i Java för att bl.a. läsa och skriva till filer.



För att bl.a. göra klasser lättare att hitta och använda kan vi gruppera ihop klasser som hör samman i olika paket (package). Paket är ett sätt att organisera sina klasser.

Alla de klasser som följer med i installationen av Java, och som vi kan använda för att utveckla egna program, är organiserade i olika paket. Klasserna är så organiserad att klasser som hör samman på något vis ligger i samma paket.

Vi har hittills använd oss av dessa klasser utan att nämna något om de paket dessa ligger i. T.ex. har vi använt oss av klasserna `String`, `Math`, `System` och olika omslagsklasser. Alla dessa ligger i paketet `java.lang` och här finns sådant som är av extra stor vikt för Javaspråket. I paketet `java.io` har vi använt oss av klassen `BufferedReader` för att kunna läsa data från tangentbordet. Här finns klasser som hanterar in- och ut-data från olika källor. Det kan vara för att läsa/skriva filer till/från hårddisken (eller tangentbordet). I paketet `javax.swing` ligger klasser för att skapa grafiska användargränssnitt och där har vi t.ex. använt klassen `JOptionPane`.

Vi har även möjlighet att gruppera våra egna klasser i olika paket och anledningen till att vi vill göra detta kan vara flera, men framför allt för att få ordning på våra källkodsfiler. När vi gör paket regleras det nämligen i vilka mappar på hårddisken filerna ska sparas, vilket kan vara bra om vi skapar större program bestående av ett flertal olika klasser.

Det finns även en speciell åtkomsttyp som reglerar accessen andra har till klasser i ett paket. Klasser i ett paket som är `public` är fortfarande åtkomliga för andra utanför paketet, `private` är private både mot klasser inom samma paket och klasser utanför paket. Med andra ord fungerar `public` och `private` så som vi hittills är vana. Däremot så är klasser/medlemmar som är deklarerad som `protected` åtkomliga för alla inom samma paket, men inte åtkomlig för klasser utanför paketet. Något av ett mellanting mellan `public` och `protected`.

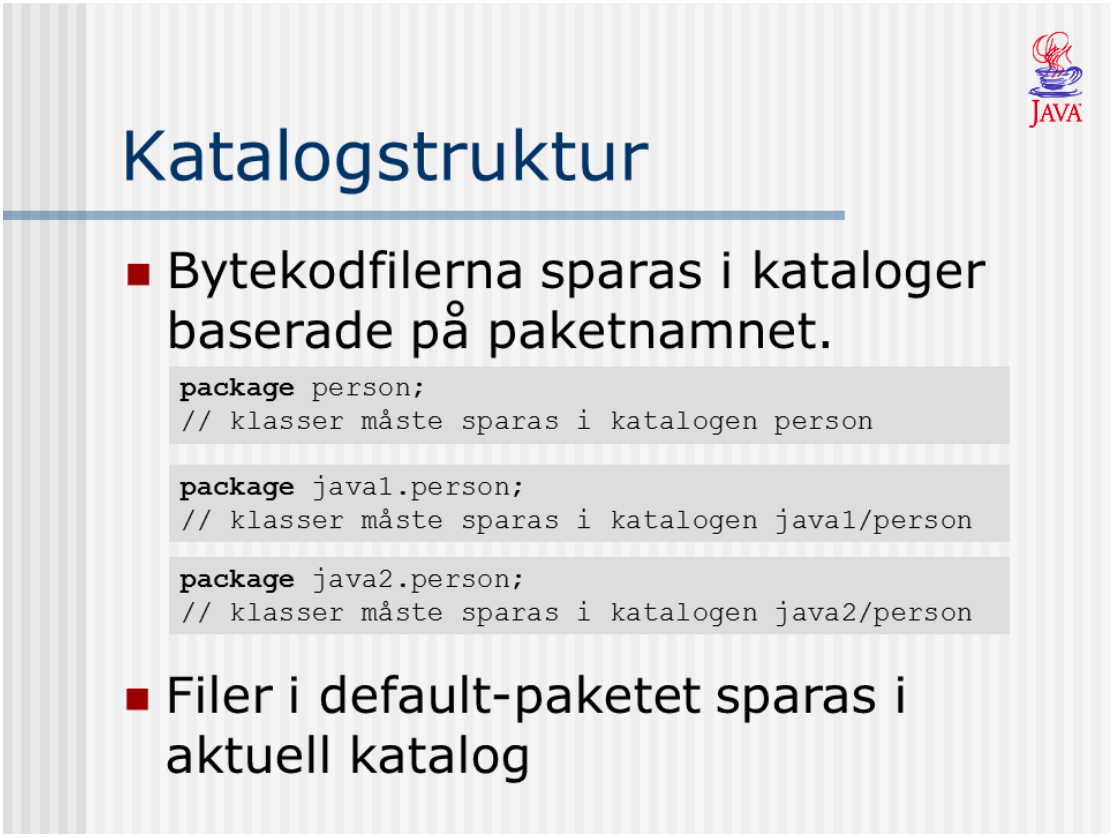
För att ange vilket paket en klass ska tillhöra måste vi först i vår källkods-fil ange paketets namn på följande sätt: `package paketets_namn;`

Med nyckelordet `package` följt av namnet på paketet, anger alltså vilket paket klassen ska tillhöra. En klass kan endast tillhöra ett paket, och paket deklarationen måste stå absolut först (överst) i filen (före eventuella import-satser, men den kan stå efter kommentarer).

Ett paketnamn kan bestå av flera delar (underpaket) och vi separerar dessa delar med en punkt. T.ex. vill vi kanske placera våra olika `Person`-klasser i ett paket som heter `person` och eftersom detta är första javakursen vill vi placera klasserna i paketet `java1.person`. På det här sättet kan vi sen i del 2 av kursen (Java II) skapa nya `Person`-klasser och placera dessa i paketet `java2.person`. Med paket är det alltså möjligt att ha flera klasser med samma namn. Anledningen är att de finns definierade i olika paket.

Hittills har vi i våra klasser inte angett någon paketdeklaration. Java kommer då att, som vid många andra tillfällen, att tilldela klassen ett standardvärde. Klassen kommer då automatiskt att hamna i ett paket utan namn. En sådan ”paketlös” klass är typiskt en klass som är helt applikationsspecifik och endast ingår i den applikation där den är definierad i. Den anses dock ändå tillhöra ett paket (ett namnlöst paket) och berörs av samma regler som andra paket.

Vi har hittills endast jobbat med namnlösa paket vilket är vanligt då applikationerna inte är så stora och komplexa.



Som nämnts tidigare reglerar även paketdeklarationen var bytekodsfilerna ska sparas på hårddisken (vill man kan man spara källkoden på ett annat ställe, men enklast för oss är att ha dem på samma ställe).

Har vi deklarerat att en klass ska tillhöra paketet `java1.person` ska även klassen sparas i en katalog java1/person någonstans på hårddisken (dvs i katalogen person som ligger i katalogen java1). Tillhör en klass paketet `java2.person` ska klassen sparas i katalogen java2/person.

Filer som inte tillhör något paket, eller rättare sagt som tillhör standardpaketet, sparas i aktuell katalog, dvs den katalog man för närvarande befinner sig i.

I zip-filen med exempel som följer med lektionen finns en katalog med namnet tmp. I denna ligger klasserna **Person.java**, **Child.java**, **Employee.java** och **PakageTest.java**. Dessa ska du jobba med under denna lektion för att prova de olika stegen för att skapa egna paket. Börja nu med att öppna klasserna `Person`, `Child` och `Employee`. Gör så att dessa klasser tillhör paketet `java1.person` (dvs längst upp i källkodsfilerna skriver du: `package java1.person;`). Kompilera filerna i ett kommandofönster och börja med **Person.java**.

Som du märker går det inte att kompilera **Child.java** och **Employee.java** då kompilatorn ”klagar på” att `Person` inte hittas. Skapa nu en katalog med namnet java1 (i tmp-katalogen). I denna katalog skapar du sen katalogen person. Flytta nu klasserna som tillhör paketet `java1.person` till katalogen tmp/java1/person.

För att kompilera klasserna måste du i kommandofönstret stå i katalogen tmp. Skriv därefter: `javac java1\person\Person.java` för att kompilera **Person.java** `javac java1\person\Child.java` för att kompilera **Child.java** etc. (Skriv `javac java1\person *.java` för att kompilera alla Javafiler i katalogen). Prova därefter att kompilera klassen **PakageTest.java** (på vanligt sätt).

När vi ska använda en klass från ett annat paket (antingen ett av standardpaket som följer med Java eller egna paket) i vår kod, kan detta göras genom att direkt i källkoden ange paketnamnet före namnet på den klass som ska användas. Detta är klassens s.k. fullständigt kvalificerade namn. För att använda klassen `Person` om den ligger i paketet `java1.person` måste vi skriva följande:

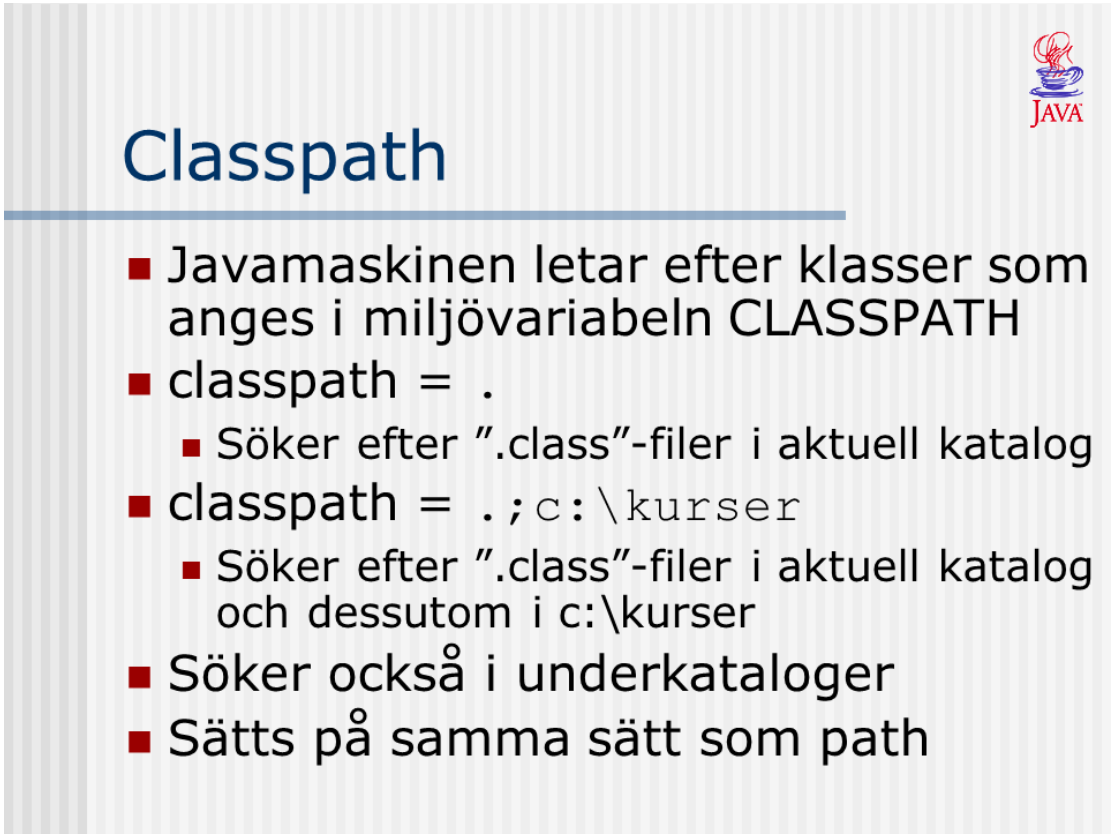
```
java1.person.Person = new java1.person.Person("Kalle", 33);
```

Som synes ger detta en ganska omständlig och upprepad syntax. Ska vi använda klassen endast en gång i koden kan vi göra på detta sätt. Är det däremot flera klasser i paket som ska användas eller om en klass används ofta, så är det att föredra att klassen eller hela paketet importeras vilket görs med en import-sats.

Ett paket importeras genom en import-sats som placeras efter eventuell paketdeklaration. Efter att ett paket har importerats så kan de publika klasserna i paketet användas direkt med sitt klassnamn. Det vi uppnår med en import-sats är att slippa använda det fullständigt kvalificerade namnet för klassen, vi importerar inte innehållet i själva klassen eller liknande till vår egen källkodsfil.

I en import-sats kan antingen en enda klass importeras, eller så kan vi genom att ange en asterisk (*) importera samtliga klasser i ett paket (observera att inga underpaket importeras!).

Ändra nu i **PakageTest.java** så att alla klasser i paketet `java1.person` importeras. Prova att kompilera och testkör.



För att den virtuella Javamaskinen ska hitta alla de klasser som används i en applikation letar den efter klasser i de kataloger som anges i miljövariabeln CLASSPATH. Utifrån de kataloger som anges i CLASSPATH söks ”.class”-filerna för ett visst paket utifrån paketnamnet, med varje paketnamnsdel som en underkatalog.

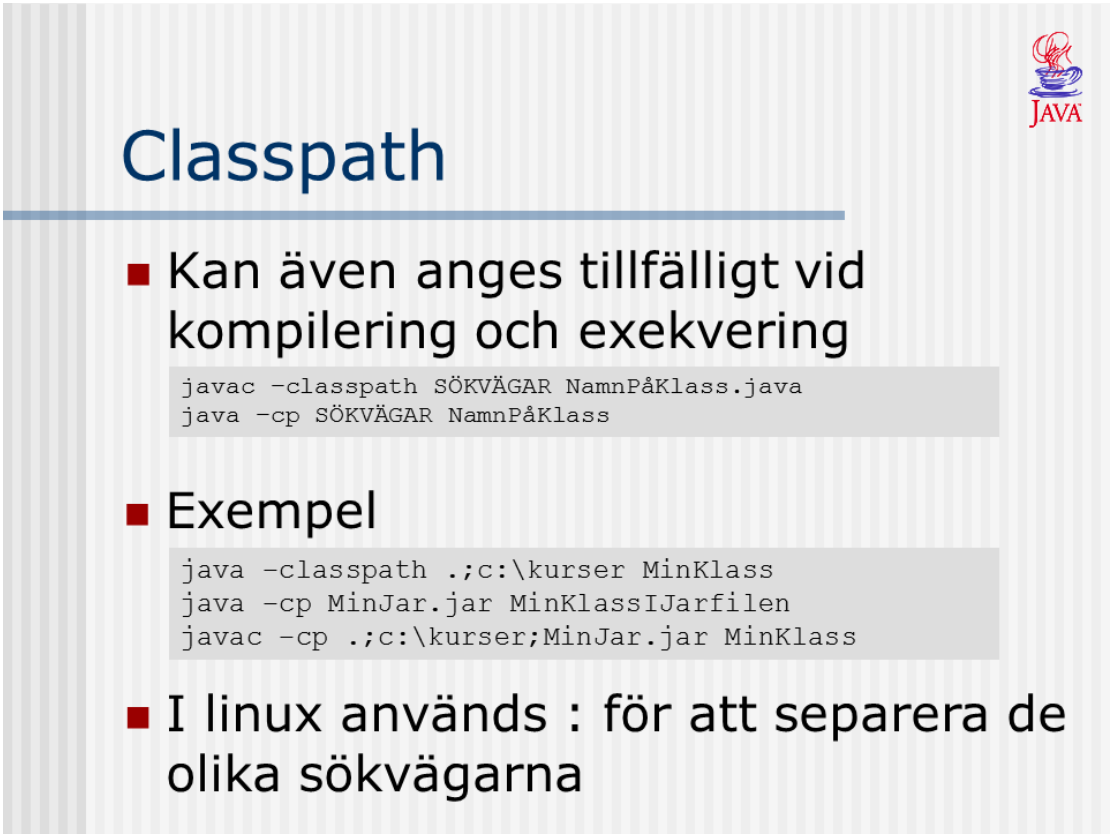
Normalt behöver vi inte sätta någon classpath, men ibland behöver javamaskinen hjälp med att hitta klasserna vi vill använda. När vi jobbar med egna paket kan det många gånger underlätta om vi sätter classpath. Om ingen classpath angetts söker den virtuella Javamaskinen alltid efter klasser i aktuell katalog (.) . (punkt) som även är namnet på default-paket om vi inte angett något paketnamn. Javamaskinen söker även alltid i den jar-fil som innehåller alla klasser som följer med i JDK.

Sätter vi classpath till endast en punkt (.) betyder detta att javamaskinen söker efter klasser i aktuell katalog, dvs i den katalog i vilken filen vi startade programmet med ligger.

Sätter vi classpath till något annat, t.ex. c:\kurser, kommer javamaskinen att söka efter klasser i katalogen kurser på c: samt i underkataloger till kurser. Sätter vi classpath till något bör vi alltid sätta den även till aktuell katalog (dvs .). Vi separerar alla kataloger som class-filer ska sökas efter med ett semikolon (;).

Exempelvis om CLASSPATH sätts till c:\java;. kommer en import av paketet java1.person att leda till att filerna för detta paket söks i katalogen c:\java\javaI\person samt i aktuell katalog. Vi ska alltså i classpath ange katalogen som innehåller första katalogen i första delen av paketets namn.

Att sätta classpath gör vi på samma sätt som vi satte miljövariabeln path (som angav var java och javac kan hittas).



I stället för att ange sökvägen till klasser i miljövariabeln kan vi tillfälligt ange sökvägen när vi kompilerar och exekverar. Detta görs genom att använda växeln –classpath eller –cp när vi kompilerar med javac eller exekverar med java.

Skriver vi

```
java –classpath .;C:\kurser MinKlass
```

kommer javamaskinen att söka efter klassen `MinKlass` (och klasser den använder) i aktuell katalog (varifrån kommandot java utfördes) samt i mappen kurser som ligger på hårdisken c:.

Skriver vi

```
java –cp MinJar.jar MinKlassIJarfilen
```

kommer javamaskinen att söka efter klassen `MinKlassIJarfilen` (och klasser den använder) i jar-filen MinJar.jar (som ligger i samma katalog som kommandot java utfördes ifrån).

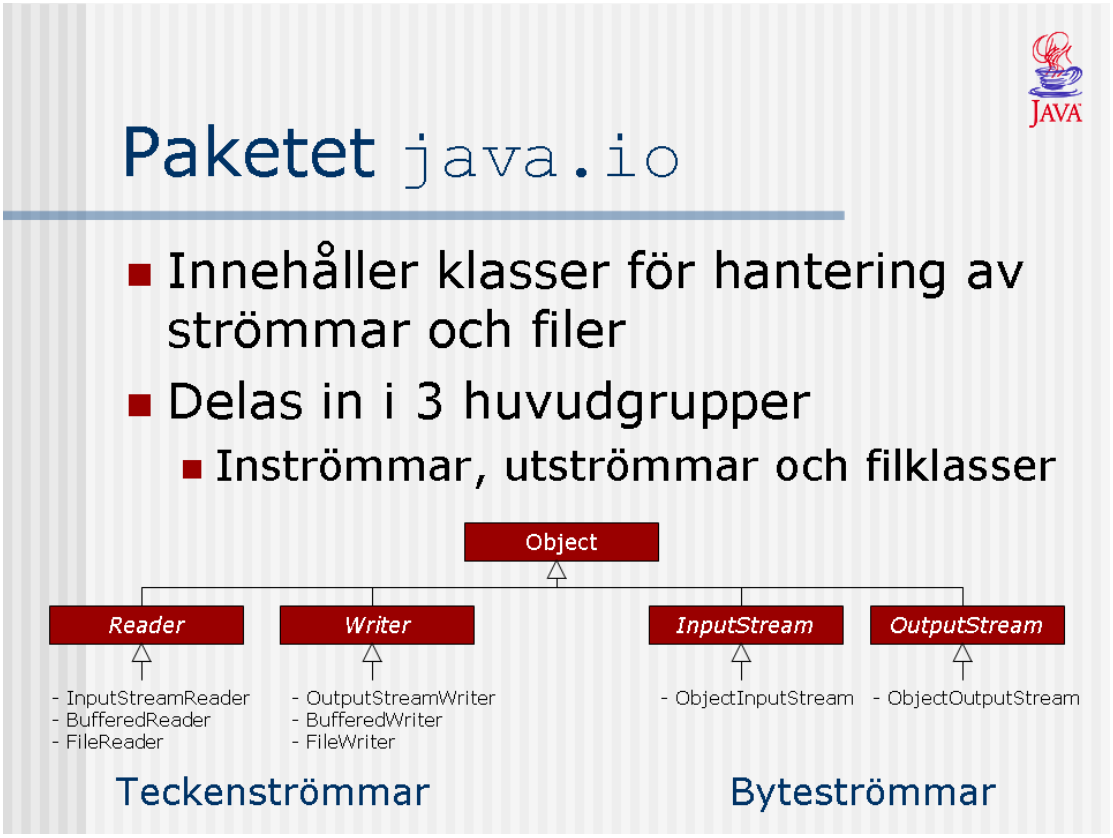
I det sista exempelt kommer klassen `MinKlass` i både aktuell katalog, mappen kurser på c: och i jar-filen MinJar-jar.

Observera att tecknet för att separera de olika sökvägarna i Windows är ; medan tecknet : används i Linux.

All kommunikation (till/från tangentbord, filer, nätverk) sker i Java via s.k. strömmar. En ström i Java hanterar en sekvens av antingen byte eller tecken. Teckenströmmar används för att läsa/skriva tecken (char) medan byteströmmar är tänkt att användas för att läsa/skriva binärdata. En ström kopplas mellan en källa och en destination.

För att läsa in information till en applikation, öppnas en ström från en informationskälla (en fil, minnet, socket etc) och läser denna information sekventiellt (tecken för tecken).

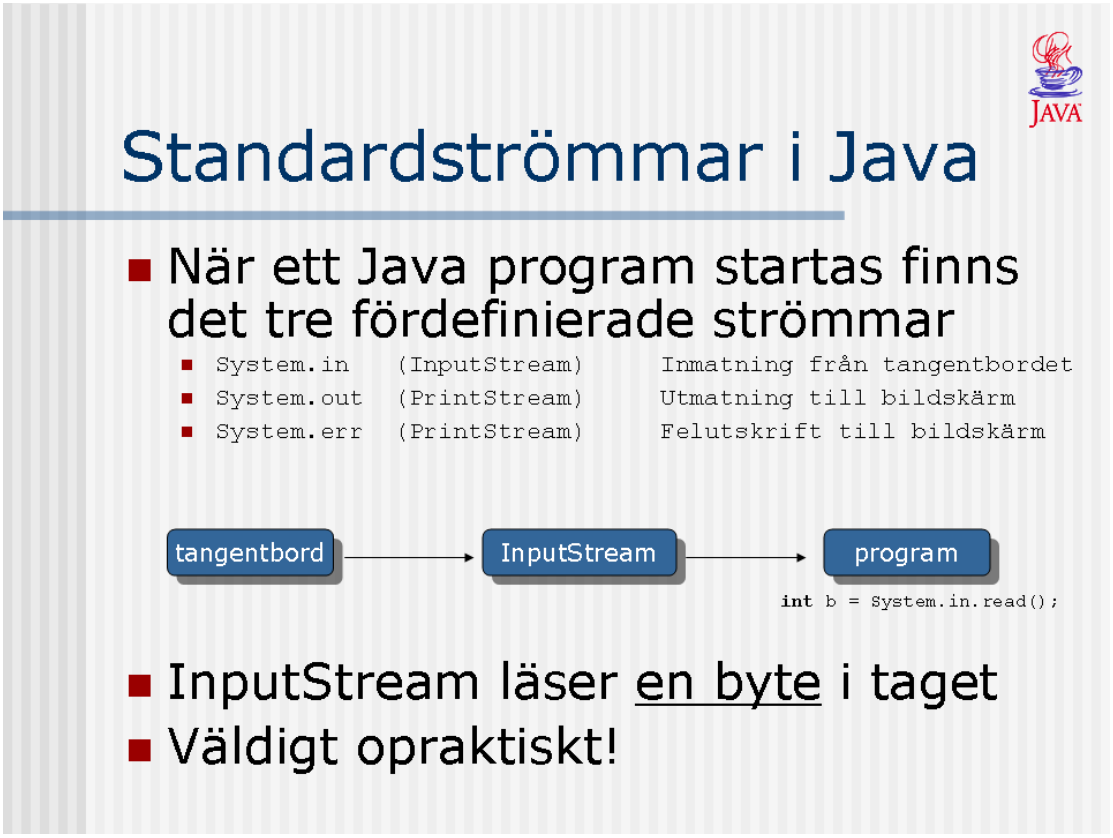
På liknande sätt kan ett program skicka information till en extern destination genom att öppna en ström till denna destination och skriva informationen sekventiellt.



I paketet `java.io` finns de allra flesta klasserna för att läsa från och skriva till strömmar. Paketet `java.io` är det mest omfattande paketet som följer med JDK. Klasserna i paketet delas in i tre huvudgrupper. Inströmmar som hanterar strömmar för att läsa från en källa, utströmmar som hanterar strömmar för att skriva till en destination, och diverse filklasser (bl.a. Klassen `File` för att representera en fil i aktuell plattform). In- och utdataströmmarna delas även in i två olika typer som nämnts tidigare. Teckenströmmar och byteströmmar.

En teckenström känner man igen genom att ordet Reader eller Writer återfinns i klassnamnet. En byteström känner man igen genom att ordet Stream återfinns i klassnamnet.

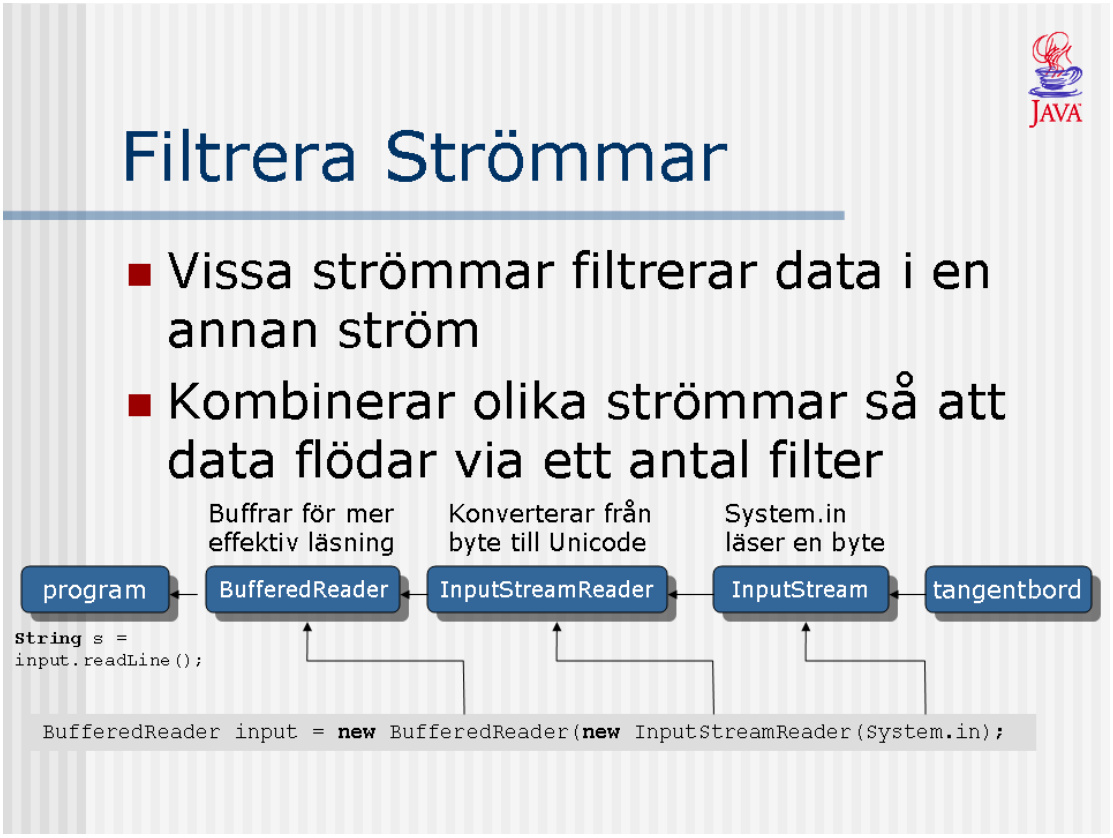
Vi kommer i denna lektion endast ta en kort titt på ett par vanliga klasser i `java.io` och någon närmare förklaring på hur exakt dessa ska användas kommer inte att ges. Mer om strömmar kommer i kursen Java II och Java III.



För att kunna använda en ström måste denna skapas genom att skapa ett objekt av någon av klasserna i `java.io`. Dock finns det redan tre skapade strömmar när ett Javaprogram körs. Dessa är `System.in`, `System.out` och `System.err` (in, out, err är publika statiska objekt i klassen `System`). `System.in` är av typen InputStream och är kopplad till ”källan” tangentbordet. `System.out` och `System.err` är av typen PrintStream är kopplade till ”destinationen” kommandofönstret.

Klassen `PrintStream` innehåller bl.a. metoderna `print` och `println` för att skriva ut olika typer av data (primitiva typer och objekt). Du känner väl igen dessa metoder från `System.out.println`? Med metoden `read` i InputStream läser man en byte i taget från källan. Det är väldigt opraktiskt och inte speciellt effektivt om man i en applikation vill läsa strängar från tangentbordet.

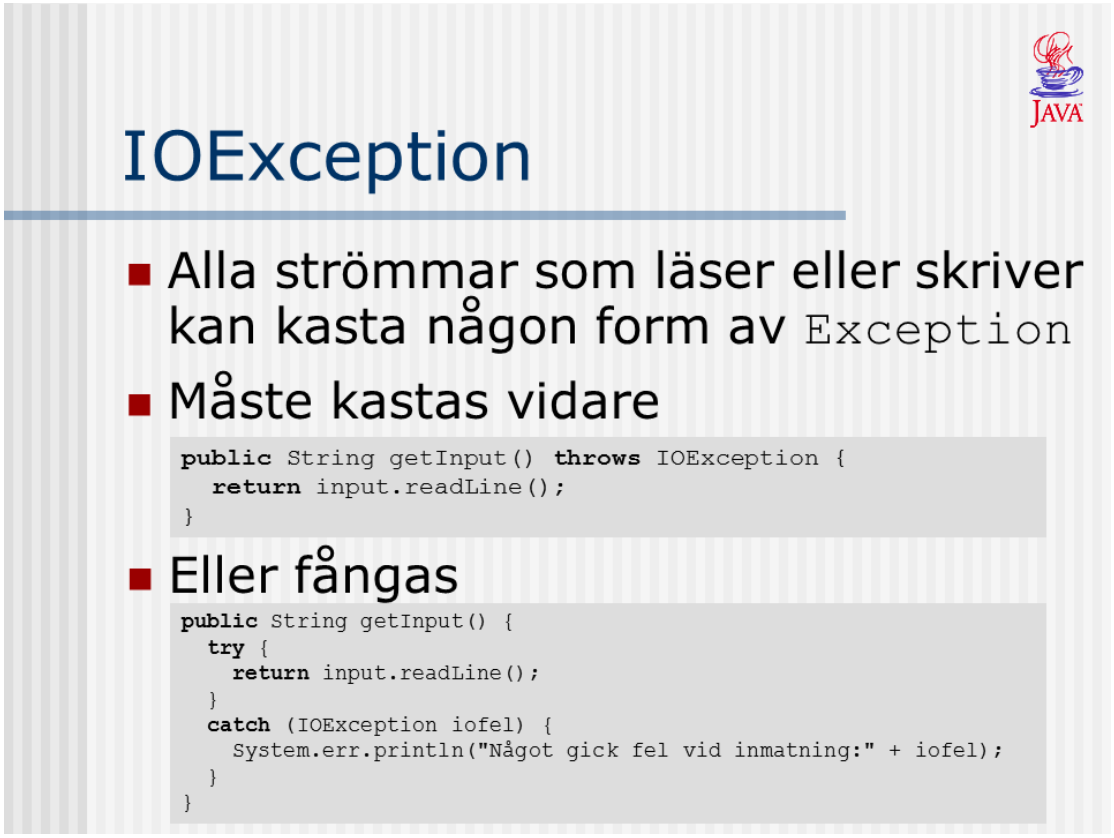
Ta en titt på exemplet **SystemIn.java** där jag enbart använder `System.in` för att läsa från tangentbordet.



Det är sällan att man enbart använder en klass i `java.io` när man läser/skriver via strömmar. Normalt använder man flera strömmar som kopplas till varandra. Detta för att låta en ström filtrera data från en annan ström. Detta har vi t.ex. gjort när vi använt klassen `BufferedReader` för att läsa data från tangentbordet. Vi har då kopplat strömmen `System.in` till en ström av klassen `InputStreamReader` (som konverterar mellan de olika typerna av strömmar; byte och tecken). Denna ström kopplar vi sen till en ström av klassen `BufferedReader` som innehåller metoden `readLine` för att läsa en hel rad från tangentbordet.

Ett annat exempel kan vara om vi vill läsa ett tal från en fil i en zip-fil. Om vi vill läsa tal från en zip-fil kan vi använda klassen `FileInputStream` för att läsa binärdata från zip-filen. Vi kopplar denna ström till en ström av klassen `ZipInputStream` som kan läsa filer i zip-formatet (packade och opackade). Vi kopplar denna ström i sin tur till en ström av klassen `DataInputStream` som innehåller metoder för att läsa primitiva typer.

I exemplet **SystemIn2.java** använder jag `BufferedReader` så som vi är vana med (är annars uppbyggd på samma sätt som SystemIn.java).



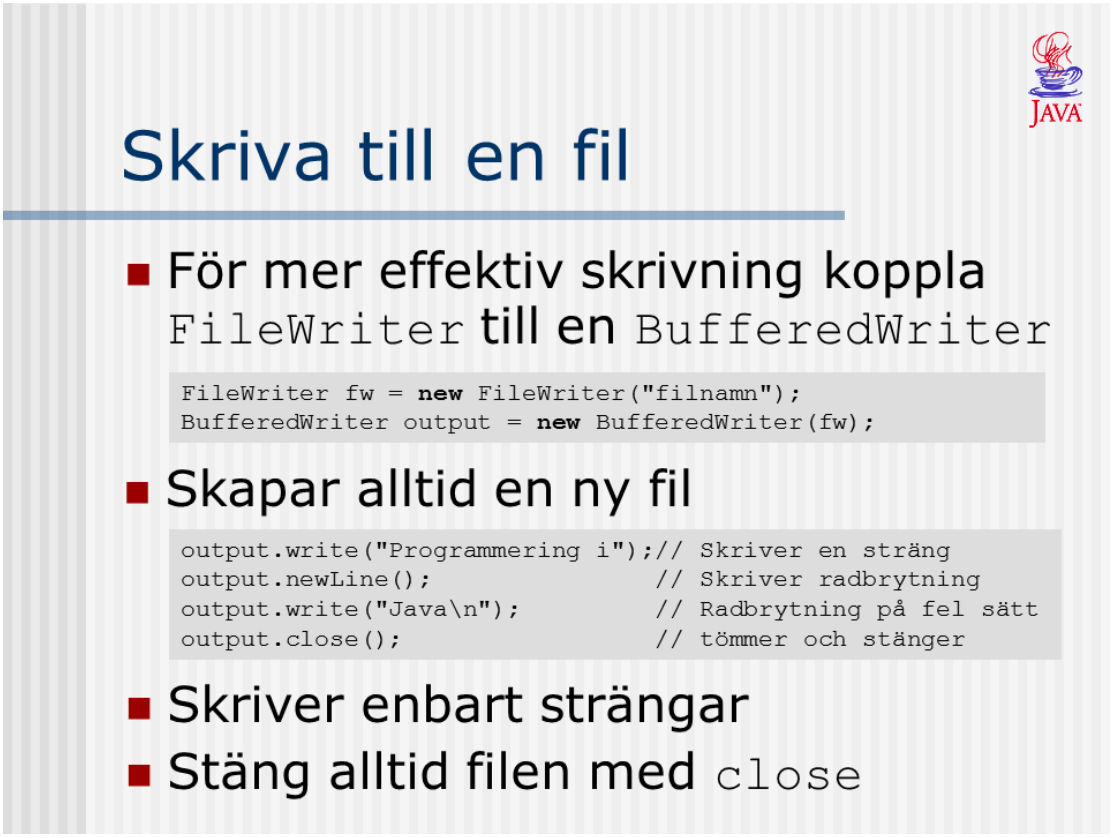
Alla strömmar som läser eller skriver data genererar fel (exception) om något oförutsett händer. Det kan t.ex. vara att filen vi vill läsa från inte finns eller att nätverkskopplingen till den fjärrdator vi kommunicerar med bryts. Som nämnts i tidigare lektioner och övningar måste vi ta hand om dessa fel på något sätt. Enklast är att enbart kasta dem vidare från den metod felet uppstod i. Detta gör vi genom att skriva `throws typ_av_exception` i metoddeklarationen.

Normalt vill vi dock kontroller vad det är för fel som har uppstått för att t.ex. se om vi kan åtgärda det på något sätt (t.ex. ange ett annat filnamn om en fil som ska läsas inte finns). Detta gör vi genom att fånga eventuella fel som kan uppstå genom en s.k. `try-catch` i källkoden. Den kod som kan generera ett exception omsluter vi i ett try-block. Efter blocket anger vi de eventuella fel som koden i try-blocket kan generera (catch), tillsammans med den kos som ska utföras om felet uppstår.

Att i en applikation läsa från och skriva till textfiler är så pass vanligt att det i Java finns speciella strömmar för detta. Dessa är `FileReader` och `FileWriter`. För att skapa en ström till en fil anger man namnet på filen i konstruktorn. Finns inte den fil vi anger som argument till konstruktorn, skapas den automatiskt. Om filen finns sedan tidigare skrivs innehållet över.

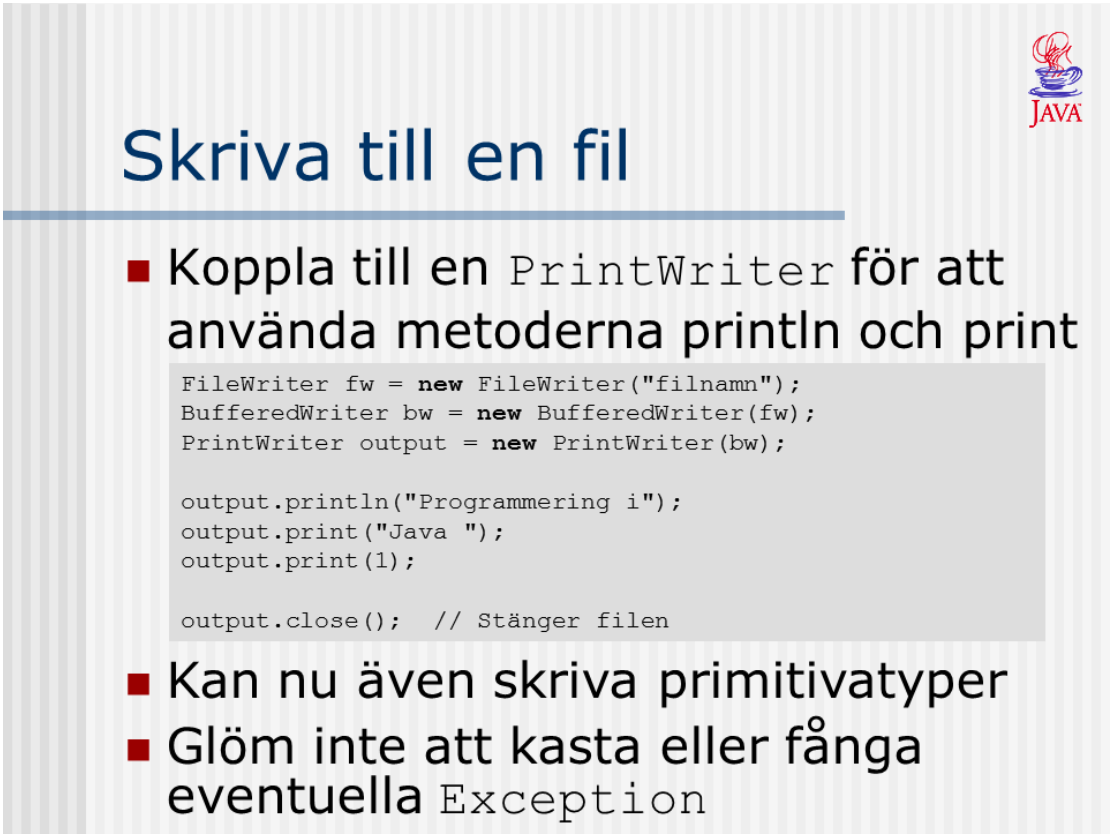
En `FileReader` är en subklass till `InputStreamReader` och används för att på ett enkelt sätt kunna läsa tecken från en fil, medan en `FileWriter` är en subklass till `OutputStreamWriter` och då givetvis används för att på ett enkelt sätt kunna spara tecken till en fil.

Vi ska dock inte direkt läsa och skriva med strömmar från dessa klasser utan bör koppla dem via andra strömmar.



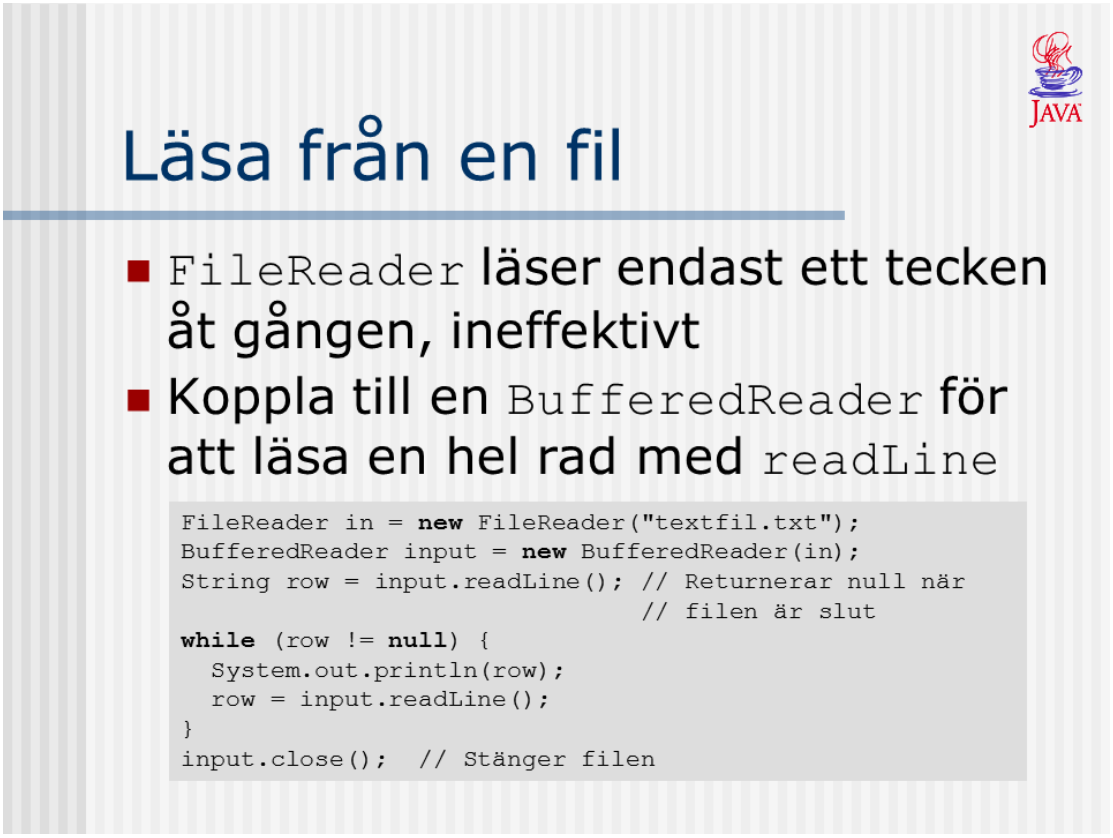
För att mer effektivt skriva data till en fil kopplar vi en ström av typen `FileWriter` (som i sin tur är kopplad till en fil) till en ström av klassen `BufferedWriter`. Denna klass innehåller metoden `write` som skriver en sträng till filen. För att åstadkomma en radbrytning måste metoden `newLine` användas. Anledningen till att vi inte kan använda `\n` är att olika plattformar hanterar radbrytning på lite olika sätt.

Det vi alltid ska komma ihåg när vi skriver data till en källa är att stänga strömmen genom att anropa metoden `close`. Gör vi inte detta är det inte säkert att de data vi skrivit till strömmen sparas. Anropet till `close` frigör dessutom den resurs som strömmen är kopplad till så att andra kan använda den.



För att göra skrivning av data till filer ännu mer bekvämt kan vi använda en ström av klassen `PrintWriter`. Denna klass innehåller metoderna `print` och `println` (som vi är vana att använda från `System.out`). Med dessa metoder kan vi nu enkelt skriva både strängar och primitiva typer till en fil. Radbryt hanteras även korrekt oavsett vilken plattform som används.

I exemplet **WriteFile.java** använder jag strömmar av klasserna `FileWriter`, `BufferedWriter` och `PrintWriter` för att spara rader som användaren skriver in till en fil på hårddisken.



Vi har redan tidigare använt `BufferedReader` för att läsa hela rader från tangentbordet. Då kopplade vi till denna en ström av `InputStreamReader` som i sin tur var kopplad till `System.in`. För att använda metoden `readLine` för att läsa en rad från en fil kan vi koppla en ström av klassen `BufferedReader` till en `FileReader`.

Som med `FileWriter` anger vi i konstruktorn till `FileReader` den fil som vi vill koppla strömmen till. Finns inte filen genereras ett `FileNotFoundException` som vi antingen kastar vidare med `throws` eller fångar upp med `try-catch`. Metoden `readLine` returnerar en sträng för varje rad i filen. När det inte finns fler rader att läsa returneras null. Detta kan vi utnyttja i en while-loop som fortsätter att snurra så länge som det returnerade värdet inte är null.

I exemplet **ReadFile.java** skrivs innehållet i en fil ut på skärmen.

Det absolut sista vi ska titta på i kursen är klassen `URL` som ligger i paketet `java.net`. Med denna klass kan vi relativt enkelt skapa en ström till en fil på nätet. Ett objekt klassen `URL` pekar ut en viss resurs på nätet. För att skapa ett `URL`-objekt anger vi adressen till filen som ett argument till konstruktorn. Adressen består av flera delar där man först måste ange vilket protokoll som ska användas (http, ftp, file etc). Därefter följer en sökväg till filen. Om man anger en adress som inte är korrekt uppbyggd kastas ett MalformedURLException som vi antingen måste kasta vidare eller fånga upp med try-catch.

Via `URL`-objektet kan man sen väldigt enkelt öppna en ström för att läsa filens innehåll. Detta kan göras på två sätt där ena sättet där ett sätt är att anropa metoden `openStream`. Denna metod returnerar en ström av klassen `InputStream`. Detta är samma typ av ström som `System.in` men att den nu är kopplad till en resurs på nätet istället för till tangentbordet.

Som jag visat tidigare i denna lektion kan vi läsa tecken för tecken direkt via en ström av `InputStream` (exempel **SystemIn.java**), men att det är betydligt enklare att läsa rad för rad om vi använder oss av `InputStreamReader` och `BufferedReader`. Vi kopplar dessa strömmar på samma sätt som när vi använder `BufferedReader` för att läsa från tangentbordet.

I exemplet **ReadURL.java** använder jag klassen `URL` för att peka ut en fil på Javas hemsida. Jag öppnar en ström till filen och visar innehållet på skärmen. I exemplet **ReadImageFromURL.java** visar jag på hur man kan spara en bild från Internet till hårddisken.