Avdeling for informatikk og e-læring, Høgskolen i Sør-Trøndelag

1.Introduksjon til web-programmering med JSP

Tomas Holt, Else Lervik
Lærestoffet er utviklet av Tomas Holt
for faget LV193D Web-programmering med JSP
og bearbeidet og tilpasset LO348D/LN349D Web-applikasjoner med JSP og JSF av Else Lervik.

1 Introduksjon til web-programmering med JSP

Resymé: Leksjonen begynner med å klargjøre begrepene klient og tjener. Etter å ha installert en web-tjener skal du ved å kjøre enkle eksempler lære forskjellen mellom klient- og tjenerprogrammering. Hovedtema i dette kurset er tjenerprogrammering. Leksjonen gir en innføring i de viktigste språkelementene i JavaServer Pages (JSP), slik at du etter å ha gått gjennom denne leksjonen er i stand til å lage meget enkle JSP-scripts uten kommunikasjon med brukeren.

Innhold

1 INTRODUKSJON TIL WEB-PROGRAMMERING MED JSP 1
1.1 INNLEDNING TIL DENNE LEKSJONEN 2
1.2 INSTALLASJON AV WEB-TJENER 2
1.2.1 Praktisk installasjon 2
1.2.2 Om localhost og porter 3
Hvordan finner jeg maskinnavnet og domenenavnet til maskinen min? 4
1.3 PRESENTASJON AV INFORMASJON PÅ WEB 5
1.3.1 Statiske web-sider 5
1.3.2 Klientside-script 6
1.3.3 Tjenerside-script 7
1.3.4 HTML-skjema 8
1.3.5 Klientside-script kontra tjenerside-script 9
Klientside-script og HTML-skjema 9
Tjenerside-script og databaser 10
Tjenerside-script og tunge beregninger 11
1.3.6 Klientside- og tjenerside-script sammen 11
1.3.7 Scripting kontra programmering 12
Programmering på klientsiden 12
Programmering på tjenersiden 12
Støtte hos nettleserne 12
1.4 SPRÅKELEMENTENE I JSP 13
1.4.1 Elementer i JSP 13
Kommentarer 13
Direktiv 14
Uttrykk (expressions) 14
Scriptlets 15
Deklarasjoner 16
1.4.2 Innebygde objekter i JSP 18
Objektet out 19
1.4.3 Utskrift inne i metoder 20

Opphavsrett: Forfatterne og Stiftelsen TISIP

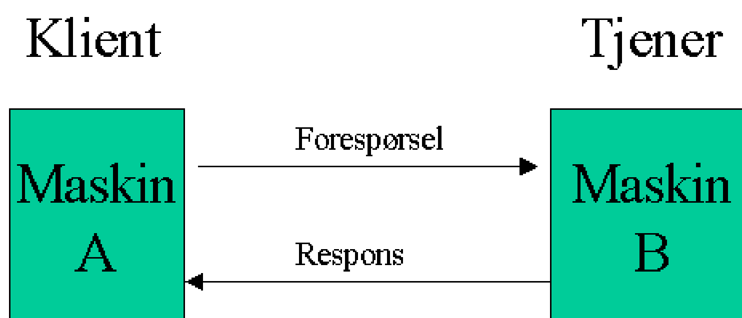
1.1 Innledning til denne leksjonen

Leksjonen består av tre deler:

- Installasjon og test av web-tjener.
- Presentasjon av informasjon på web. Dette kapitlet gir en introduksjon til ulike måter å programmere for web. HTML, JSP, Java-appleter, JSF og Ajax nevnes. Sentrale begreper er statisk, kontra dymamisk webside, og klient- kontra tjenerprogrammering.
- Elementene i JSP. Her får du presentert de viktigste elementene i scriptspråket JSP, og sammenhengen som eksisterer mellom enhver JSP og den tilsvarende Java-klassen beskrives.

1.2 Installasjon av web-tjener

Spredning av informasjon på web er basert på at man har klienter (nettlesere) som kobler seg opp mot tjener-applikasjoner. Klienten forespør den informasjonen man er ute etter, og tjeneren vil returnere denne informasjonen hvis mulig. Klienten vil i dette tilfelle kjøres på en klientmaskin, mens tjener-applikasjonen vil kjøres på en tjenermaskin. Ofte brukes navnet klient/tjener både om maskinen og programmet som kjører på maskinen, og denne uttrykksmåten brukes også i denne teksten når det er naturlig.



Figur 1: Klient- og tjenerroller

Merk at både maskin A og maskin B kan være både klientmaskin og tjenermaskin. Hvis maskin A gjør forespørsel mot maskin B vil førstnevnte være klientmaskin og sistnevnte tjenermaskin. Snur man på flisa slik at maskin B gjør forespørsel til maskin A vil de være hhv. klient- og tjenermaskin.

Det er rollen maskinen spiller som er utslagsgivende. Den maskinen som gjør forespørselen er *klientmaskin* og maskinen som mottar (og betjener) forespørselen er *tjenermaskin*. Derfor er det mulig å la en og samme maskin være både klient og tjener. Det er ofte en praktisk løsning i en undervisningssituasjon. Vi legger derfor opp til at du bruker din egen PC både som tjener og klient.

1.2.1 Praktisk installasjon

For at vi skal kunne øve oss på web-programmering (scripting) trenger vi en tjener der vi kan legge ut web-sider. Til nå har du sannsynligvis kun brukt din egen maskin som klientmaskin, dvs. andre har ikke kunnet gjøre forespørsler mot din maskin. Dette vil være nødvendig i det vi nå skal begynne med web-programmering. Det er jo ingen vits i å lage web-sider (med fine script ☺) uten at andre kan få tak i dem (og at vi selv kan se dem). Fordi mange kun har én

Opphavsrett: Forfatterne og Stiftelsen TISIP

maskin vil det være naturlig å bruke denne maskinen både som tjenermaskin og klientmaskin. Dette er mulig fordi klienten og tjeneren kun er applikasjoner. Klienten er nettleseren i vårt tilfelle. Tjeneren er et program som startes opp på maskinen og venter (i det uendelige) på forespørsler fra klienten. Når en forespørsel kommer så utføres denne, og tjeneren vil så vente på neste forespørsel. Tjeneren er altså et program som du må stanse ”manuelt” for at den skal stoppe.

Tidligere har vi i dette kurset lagt opp til at man bruker Tomcat web-tjener med en enkel teksteditor, for eksempel TextPad. Tomcat krever ikke store maskinressurser, og det kan av denne eller andre grunner være aktuelt for enkelte av dere å bruke Tomcat også i år. Enkel installasjons- og brukerveiledning finner du på http://javabok.no/installasjoner/Tomcat.html.

Spørreundersøkelser blant studentene viser imidlertid at det er ønske om å bruke integrerte verktøy. Fra og med i år gis det derfor veiledning i bruk av NetBeans i tilknytning til dette kurset. Om du ønsker å bruke andre verktøy, som for eksempel Eclipse, kan du gjerne det, men vi kan dessverre ikke gi brukerstøtte på andre verktøy enn NetBeans.

Der du fant denne leksjonen finner du også introduksjon til bruk av NetBeans for å lage JSP-applikasjoner.

Gjør følgende: Installer NetBeans (eventuelt annet verktøy) og test installasjonen som beskrevet i veiledningen. Merk hvordan vi bruker nettleseren til å teste installasjonen. Med andre ord:

GJØR OPPGAVE 1a) - 1c) i ØVING 1 FØR DU GÅR VIDERE

1.2.2 Om localhost og porter

Under testen foran hentet du fram web-applikasjonen *Leksjon1Eksempler* ved å skrive slik i nettleseren:

http://localhost:8080/Leksjon1Eksempler/

Adressen *localhost* sender alltid forespørselen til den maskinen du sitter på (uansett hvilken maskin det er). Skal andre kunne ta kontakt med web-tjeneren på din maskin, må de skifte ut *localhost* med adressen til din maskin (f.eks. noe sånt som http://www.torilhansen.no/, se nedenfor for å finne din egen maskins navn/adresse). I tillegg så må eventuelle brannmurer være slått av eller konfigurert for å slippe trafikk inn på port 8080.

Ja, hva mener vi når vi sier ”port 8080”? En port er en inngang til et program som kjører på maskinen. Hvert enkelt program som skal kunne nås fra andre maskiner må ha sitt eget (port-) nummer. Mye brukte program, som for eksempel web-tjenere og databasesystemer, har etter hvert fått mer eller mindre standardiserte nummer, såkalte ”Wellknown Port Numbers”(søk på nettet etter dette uttrykket!).

La oss se på en virkelig nettadresse, for eksempel http://hist.no/. Prøv med http://hist.no:8080/ og med http://hist.no:80/. Begge deler går bra. Men eksempelvis fungerer ikke http://hist.no:8083/. 80 er standardnummeret for http, og dersom du ikke oppgir portnummer er det den som gjelder. (Prøv http://localhost/Leksjon1Eksempler/ og http://localhost:80/Leksjon1Eksempler/. Det går ikke så bra, og grunnen er at du kjører web-tjeneren på port 8080. Det er mulig å endre denne i konfigurasjonsfilene til web-tjeneren, men det ser vi bort fra nå.) At begge portnumrene fungerer for hist.no skyldes at den er satt opp til å lytte på begge portene.

Opphavsrett: Forfatterne og Stiftelsen TISIP

Hvordan finner jeg maskinnavnet og domenenavnet til maskinen min?

Mitt maskinnavn er *th* og domenenavnet er *idb.no* (jfr. URL *http://th.idb.no:8080*). Måten du finner fram til maskinnavn og domenenavn er litt avhengig av operativsystem.

I Windows kan vi skrive *ipconfig*. Du får da opp noe slikt:

```
C:\>ipconfig

IP-konfigurasjon av Windows 2000

Ethernet-kort Lokal tilkobling:

        Tilkoblingsspesifikt DNS-suffiks  . . . . :
        IP-adresse. . . . . . . . . . . . . . . . : 158.38.51.141
        Nettverksmaske. . . . . . . . . . . . . . : 255.255.254.0
        Standard gateway. . . . . . . . . . . . . : 158.38.50.1

C:\>
```

Figur 1: ipconfig

Vi leser av IP-adresse i denne figuren og bruker den i vår neste kommando, *nslookup <IP-adresse>*. I mitt tilfelle blir det *nslookup 158.38.51.141* (du vil få en annen IP-adresse). Resultatet blir som følger:

```
C:\>nslookup 158.38.51.141
Server:  skaunix.hist.no
Address:  158.38.49.10

Navn:    th.idb.hist.no
Address:  158.38.51.141


C:\>
```

Figur 2: nslookup

Det er de to siste linjene som er av interesse for oss. Vi ser at adressen er den samme som jeg oppga i kommandoen *nslookup*. Dette er derfor min maskin. Navn inneholder her både maskinnavn og domenenavn, og er som vi ser for min maskin *th.idb.hist.no*.

Endelig nevner vi at 127.0.0.1 er ekvivalent for *localhost*. Vi har altså fire muligheter til å nå web-applikasjonen vår:

1. http://localhost:8080/Leksjon1Eksempler/
2. http://127.0.0.1/Leksjon1Eksempler/
3. http://158.38.51.141:8080/Leksjon1Eksempler/
4. http://th.idb.no:8080/Leksjon1Eksempler/

Kun de to siste kan brukes fra andre maskiner enn vår egen.

GJØR OPPGAVE 1d) og 1e) i ØVING 1 FØR DU GÅR VIDERE

Opphavsrett: Forfatterne og Stiftelsen TISIP

1.3 Presentasjon av informasjon på Web

Kode-eksemplene i denne leksjonen ligger i eksempelsamlingen som følger med leksjonen. Hvis du har fulgt installasjonsveiledningen for NetBeans har du allerede eksemplene installert på maskinen din. Følg med på din egen skjerm etter hvert som du leser om de ulike presentasjonsformene nedenfor. Gjør gjerne mindre endringer i HTML-kodene og studer effekten det har på utseendet i nettleseren. (Husk å trykke på Oppdater eller lignende i nettleseren for hver endring.)

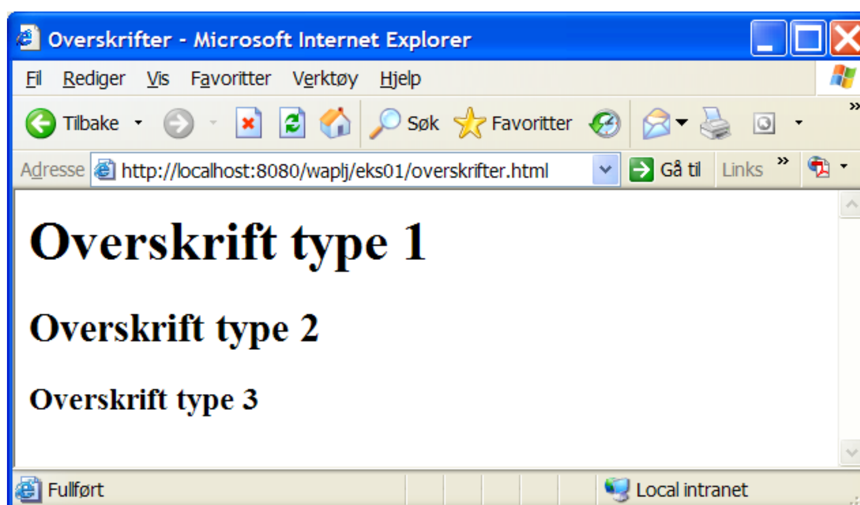
1.3.1 Statiske web-sider

På midten av 1990-tallet begynte web å ta av for alvor. Fram til den tid var det stort sett universiteter som hadde benyttet seg av spredning av informasjon på web. Det skjedde i form av statiske web-sider (f.eks. publisering av dokumenter o.l), noe som krevde forholdsvis lite av teknologien. HTML (Hypertext Meta Language) ble brukt for å beskrive hvordan informasjonen skulle presenteres for brukeren.

Eksempel på informasjon på HTML-form

```
<!-- Filnavn eksempelsamling 1: overskrifter.html -->
<!-- Eksempel på en statisk web-side -->
<html>
<head><title>Overskrifter</title>
</head>
<body>
<h1>Overskrift type 1</h1>
<h2>Overskrift type 2</h2>
<h3>Overskrift type 3</h3>
</body>
</html>
```

Skriften mellom tegnene < og > er HTML-koden. Presentasjonen av denne informasjonen gjøres av en nettleser (f.eks. Internet Explorer eller Firefox). Figuren under viser hvordan HTML-koden over kan se ut i nettleseren.



Figur 3: Presentasjon av HTML-kode

Nettleserens jobb er todelt. Som beskrevet over er en del av jobben å presentere informasjonen som er innpakket i HTML-kode for brukeren. For å kunne gjøre dette må

Opphavsrett: Forfatterne og Stiftelsen TISIP

informasjonen først hentes. Hvordan gjøres så dette? Informasjonen ligger rundt omkring på tjenere, og nettleseren kan hente ned denne informasjonen gjennom en URL (*Universal Resource Locator)* som beskriver det dokumentet man er ute etter. F.eks. kan en URL se slik ut: http://www.hist.no/index.html Denne tolkes som følger: http (*Hyptertext Transfer Protocol)* beskriver protokollen (regelverket) maskinene bruker for å samarbeide, mens *www.hist.no* beskriver hvilken maskin dokumentet ligger på. Dokumentet vi er ute etter heter index.html. Når vi enten skriver inn URL'en over i adressefeltet til nettleseren eller trykker på den (i det dokumentet du nå leser), vil nettleseren sørge for å koble seg opp mot tjenermaskinen *www.hist.no* og be om å få dokumentet *index.html* (prøv å trykke på URL'en selv og se på adressen i adressefeltet til nettleseren). Tjeneren vil så returnere innholdet i filen index.html, og nettleseren presenterer innholdet. For å se hva som faktisk returneres fra tjeneren kan en bruke *Vis/Kilde* i Internet Explorer eller *Vis/Kildekode* i Opera. Andre nettlesere har liknende funksjonalitet. Prøv dette!

Man skulle etter hvert oppdage at HTML hadde klare svakheter. Dette kom av at man bare kunne presentere statiske web-sider. Når siden var laget, så kunne den ikke forandres (uten å redigere selve HTML-koden). F.eks. vil det ikke være mulig å skrive ut klokkeslettet på en slik HTML-side fordi siden ikke kan forandre seg (m.a.o så vil klokken være det samme hele tiden!). Det ble etter hvert klart at man hadde behov for web-sider som var *dynamiske* (dvs. kunne tilpasses/forandres).

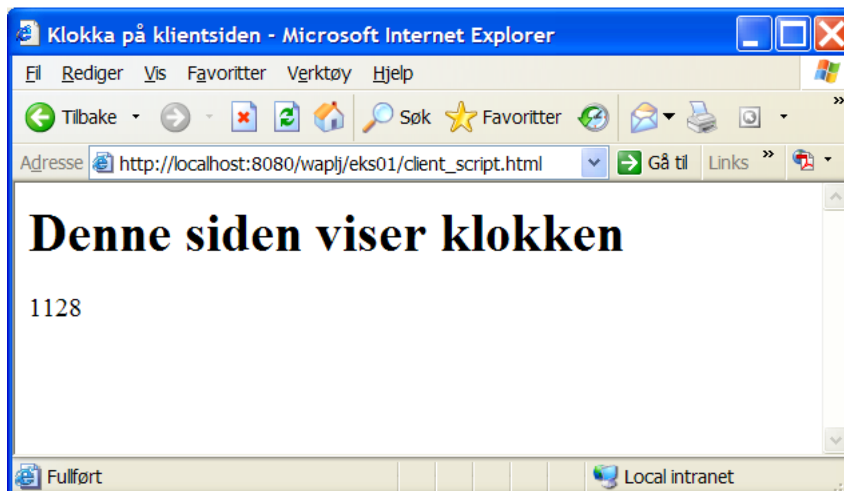
1.3.2 Klientside-script

En løsning på problemet var å lage klienter (her nettlesere) som kunne kjøre script. Dette kalles klientside-script da det er klienten som utfører scriptet. I dette tilfellet blander man HTML-kode sammen med script-kode. HTML-koden vil fortsatt være statisk, men nettleseren vil utføre eventuelle script som medfølger. Deretter presenteres resultatet for brukeren (dvs. resultatet av både HTML-koden og scriptet). Under vises et eksempel på hvordan HTML blandes med script-kode (sistnevnte i fet skrift, og i dette tilfelle JavaScript som er de facto standard for klientside-scripting).

```
<!-- Filnavn eksempelsamling 1: client_script.html -->
<!-- Eksempel på programmering på klientsiden-->
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Klientside-script</title></head>
<body>
<h1> Denne siden viser klokken </h1>
<script type="text/JavaScript">
    var now = new Date();
    document.write(now.getHours());
    document.write(now.getMinutes());
</script>
</body>
</html>
```

Presentasjonen i nettleseren kan være som følger:

Opphavsrett: Forfatterne og Stiftelsen TISIP



Figur 4: Klientside-script

Når klientside-script brukes, er det mulig for brukeren av nettleseren å se koden til scriptet. Dette er fordi HTML-kode blandes med script. HTML-koden til en web-side kan sees som tidligere forklart ved å bruke *Vis/Kilde* i nettleseren.

1.3.3 Tjenerside-script

Ofte er det flere løsninger på problemer, og problemet med statiske web-sider var det også mulig å løse på flere måter. Som allerede omtalt var den ene muligheten å kjøre script på klientsiden. Den andre muligheten var motsetningen, nemlig å kjøre script på tjenersiden. Hele problemet med statiske web-sider bunner jo i at tjeneren returnerer et dokument (HTML-kode) slik det er lagret på tjeneren. En løsning på problemet vil derfor være at web-tjeneren selv kan forandre på innholdet i dokumentet som returneres. Dvs. at tjeneren må kunne lage HTML-koden (eller deler av den) dynamisk når en forespørsel om et dokument kommer.

Hvordan løses så dette i praksis? Man kan tenke seg en løsning som er veldig lik løsningen for klientside-script, nemlig ved å putte inn script-elementer inn i HTML-koden. Det er denne løsningen som brukes ved JSP (JavaServer Pages). Filen som ligger på tjenersiden vil i disse tilfellene ikke lenger ha etternavnet .html (eller .htm). F.eks. vil en fil ikke lenger hete *fil.html* men *fil.jsp*. (Det kanskje mest kjente scriptspråket for tjenerprogrammering er ikke JSP, men PHP, da har filene endelsen .php). Grunnen til at en velger et annet etternavn i disse tilfellene er at web-tjeneren skal kunne avgjøre når det trengs å gjøres noe med innholdet i filen som etterspørres. Er den en fil som slutter med .html, er alt tjeneren trenger å gjøre å returnere filen. Slutter imidlertid filnavnet med f.eks. .jsp vil tjeneren måtte prosessere innholdet i filen. På samme måte som for klientside-script vil kun de delene av filen som inneholder script-elementer prosesseres. HTML-koden vil være som den er. Under vises et eksempel på hvordan en JSP-fil kan se ut. (filnavn i eksempelsamlingen er *server_script.jsp*)

```
<!-- Filnavn eksempelsamling 1: server_script.jsp -->
<!-- Eksempel på programmering av tjenersiden -->
<html>
<head><title>Klokka på tjeneren</title></head>
<body>
<h1> Denne siden viser klokken nå </h1>
        <% out.println(new java.util.Date()); %>
</body>
</html>
```

Opphavsrett: Forfatterne og Stiftelsen TISIP

Legg merke til at alle linjene bortsett fra én (fete typer) er HTML-kode. Web-siden som er resultatet av koden over vil se slik ut:

Figur 5: Tjenerside-script

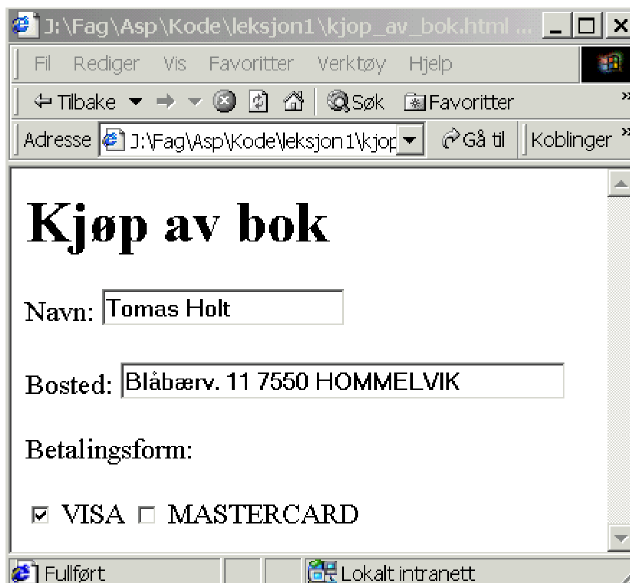
Tjeneren kjenner igjen script-elementene ved at disse starter med <% og slutter med %>. Når tjeneren støter på disse kjennetegnene, vil den kjøre scriptet, og kommandoene som er beskrevet i script-koden utføres. Legg merke til at script-koden ikke returneres til nettleseren, kun resultatet av utførelsen av scriptet. Det er derfor kun HTML-kode som returneres til nettleseren. Ved å kjøre kommandoen *Vis/Kilde* i nettleseren får vi ut følgende:

```
<!-- Filnavn eksempelsamling 1: server_script.jsp -->
<!-- Eksempel på programmering av tjenersiden -->
<html>
<head><title>Klokka på tjeneren</title>
</head>

<body>
<h1> Denne siden viser klokken nå </h1>
        Fri Jan 05 11:30:34 CET 2007

</body>
</html>
```

1.3.4 HTML-skjema

Vi har nå sett på hvordan en fikk statiske HTML-sider til å bli dynamiske gjennom script. Alle var selvsagt henrykte over at de nå hadde muligheten til å følge med klokka på web, men de hadde fremdeles ikke muligheten til å meddele sin begeistring til web-redaktøren (e-post, telefon osv. regnes ikke med her). Det man trengte var muligheten for å ha felter i web-sidene der en kunne skrive inn tekst, gjøre enkle valg osv. Dette problemet ble løst ved at man utvidet HTML med noe som kalles *HTML-skjema* (engelsk: HTML forms). Ved hjelp av HTML-skjema er man i stand til å lage HTML-kode der også brukeren kan taste inn opplysninger, gjøre valg osv. Under vises et eksempel på hvordan kjøp av bok med HTML-skjema kan se ut:

Opphavsrett: Forfatterne og Stiftelsen TISIP



Figur 6: HTML-skjema

For at eksempelet over skal fungere er man selvsagt avhengig av at informasjonen som er tastet inn blir sendt til tjeneren (slik at bokhandelen får disse dataene). Hvordan dette gjøres, vil vi se på i neste leksjon. Nå konstaterer vi bare at dette er mulig.

1.3.5 Klientside-script kontra tjenerside-script

Er det så noen grunn til å ha både klientside- og tjenerside-script? Ville det ikke ha vært enklere med kun en av variantene? Selvfølgelig hadde det vært enklere kun med en variant, men det er fordeler og ulemper med de to variantene.

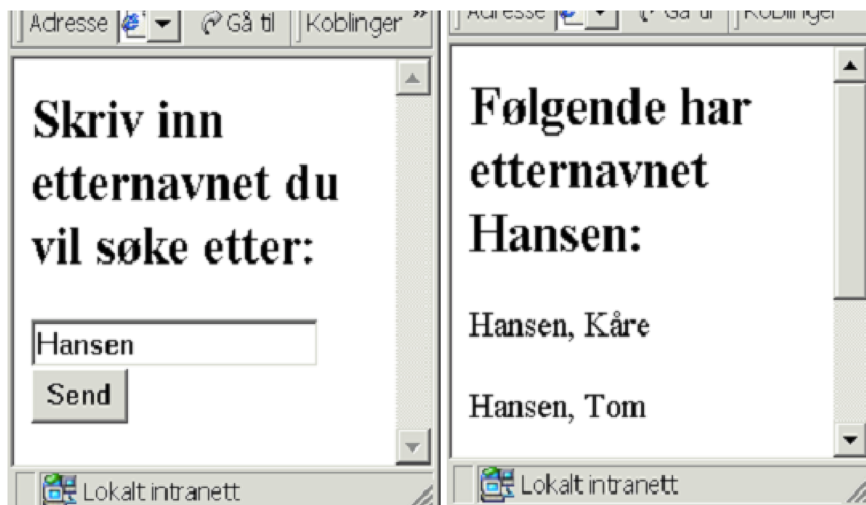
Klientside-script og HTML-skjema

Klientside-script har den fordelen at en kan gjøre ting på klientsiden. F.eks. vil det være mulig å oppdatere klokken i en web-side uten å forespørre tjeneren om en ny (eller oppdatert) web-side. Nettleseren vil selv tolke medfølgende script og gjøre nødvendige oppdateringer. Det kan i mange tilfeller være fornuftig med en slik løsning. Tenk bare på de tilfeller der en fyller inn opplysninger på en web-side. Ved bestilling av en bok må en eksempelvis oppgi diverse personalia (navn, bosted osv.). For de som tar imot bestillingen er det viktig at nødvendige opplysninger er fylt inn (mangler f.eks. bosted vet man ikke hvor boka skal sendes). At de oppgitte opplysningene kan sjekkes er derfor en nødvendighet. Hvis disse opplysningene sjekkes av et klientside-script kan brukeren få beskjed med en gang om han f.eks. glemte å fylle ut hvordan betaling skal skje (visa eller mastercard). Dette gjøres uten at opplysningene sendes til tjeneren. Opplysningene vil først sendes når alle nødvendige data er utfylt. Skal man i stedet bruke et tjenerside-script for å sjekke at alle de nødvendige feltene er fylt ut, må opplysningene i HTML-skjemaet sendes til tjeneren hver gang de skal sjekkes. Hvis ikke informasjonen er fylt inn på en tilfredsstillende måte må tjeneren sende tilbake en ny web-side der brukeren bes om å oppgi opplysningene på nytt. Dette vil føre til ekstra nettverkstrafikk, og kan gi mye tregere respons. Det tar tid å utføre forespørsler over nettverk (responstiden vil variere avhengig av nettverkstrafikk og belastning på tjeneren) noe som gjør at det kan ta tid før eventuelle feilmeldinger vises for brukeren.

Opphavsrett: Forfatterne og Stiftelsen TISIP

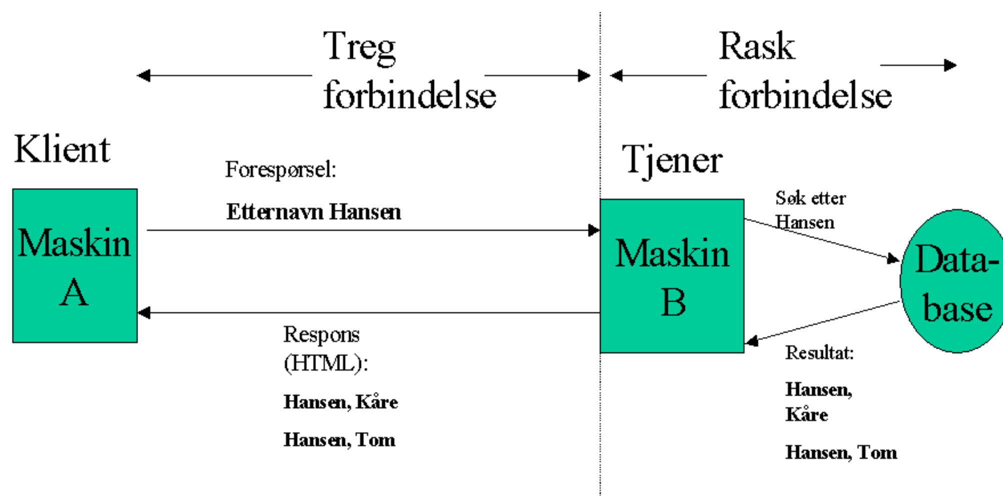
Tjenerside-script og databaser

Noe av den samme argumentasjonen kan en bruke for tjenerside-script. Ofte vil det være aktuelt at en web-side skal vise utdrag fra en database. Det kan f.eks. være at man ønsker å tilby en tjeneste som gjør det mulig å finne folk på etternavn. Tjenesten skal være slik at når en søker etter Hansen så vil alle som heter Hansen vises på web-siden.



Figur 7: Forespørsel og svar

Figuren under viser en hvordan denne forespørselen vil foregå.



Figur 8: Forespørsel mot database

Ofte vil det i slike tilfeller være slik at web-tjeneren og database-tjeneren ligger på samme maskin eller på samme lokale nettverk. Dette gjør at web-tjeneren raskt kan skaffe informasjonen som ligger i databasen (fordi det er rask forbindelse mellom dem).

Generelt sett er tjenerside-script langt sikrere enn klientside-script. Som vi har sett vil klient-script (samt HTML-koden) sees ved å bruke *Vis/Kilde* e.l. i nettleseren. Hvis brukeren lagrer denne filen og senere tar den inn i en teksteditor kan han forandre på scriptet (og også HTML-koden selvfølgelig). Dette gjør at brukeren faktisk kan få scriptet til å gjøre ting det ikke var

Opphavsrett: Forfatterne og Stiftelsen TISIP

tenkt å gjøre. Dette vil være umulig med tjenerside-script fordi brukeren aldri vil se selve scriptet på klientsiden, kun HTML-koden som er *generert av* tjenerside-scriptet.

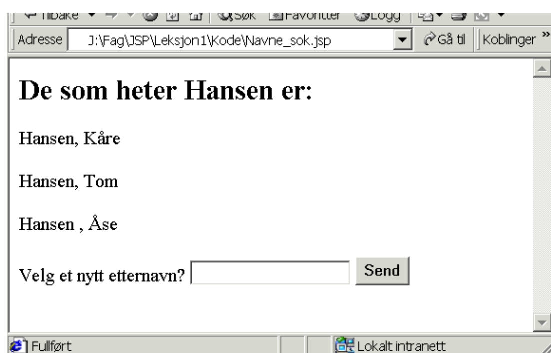
Tjenerside-script og tunge beregninger

Når det kreves tunge beregninger vil det vanligvis være lurt å legge disse til tjenersiden. Dette kommer av at tjenermaskinene ofte er mye raskere enn klientmaskinene. Klientmaskinene kan variere stort i hastighet (ikke alle kjøper seg nye datamaskiner hvert år), mens man ofte er avhengig av å ha raske tjenermaskiner. Tunge beregninger kan derfor gå *mange ganger* så fort hvis de gjøres på tjenermaskinen i forhold til på klientmaskinen. Dette må imidlertid ses i forhold til trafikken mot det aktuelle nettstedet. I takt med utviklingen av maskinvaren, også for klienter, flyttes stadig grensen for hva som er "tunge beregninger". Vi er nok nå på vei mot tykkere klienter (jmf. Ajax). Java-appleter er et eksempel på tykk klient som kanskje også kommer til heder og verdighet igjen.

1.3.6 Klientside- og tjenerside-script sammen

Du har nå skjønt at om scriptet befinner seg på klientsiden eller på tjenersiden vil avhenge av omstendighetene. Vil det ikke være omstendigheter der en blanding av de to vil være fornuftig? Jo. Tenk bare på eksemplet hvor man kan søke på personers etternavn. Som forklart vil det være en fordel at selve spørringen mot databasen skjer av et tjenerside-script, men hva med å sjekke at lovlig etternavn tastes inn i nettleseren? F.eks. vil ikke et lovlig etternavn bestå av tall. En slik sjekk foretas best med klientside-script.

Hva skal man så velge? Det er ingenting i veien med å benytte begge typene script. Klientside-scriptet kan gjøre sin jobb helt uavhengig av tjenerside-scriptet, og motsatt. Imidlertid vil det være mulig for tjenerside-scriptet å lage et klientside-script som sendes sammen med HTML-koden til nettleseren. Tenk bare på vår berømte etternavnsøker. Kanskje har vi lyst til at brukeren etter å ha mottatt en liste over alle som har et bestemt etternavn, skal ha muligheten til å søke på et nytt etternavn. Dette kan gjøres ved at vi har et tekstfelt (HTML-skjema) nederst på den samme siden.



Figur 9: Søk på navn

Vi har også lyst til at et klientside-script skal sjekke innholdet i dette tekstfeltet før det sendes til tjeneren igjen. Husk at denne web-siden i utgangspunktet ble generert av et tjenerside-script. Tjenerside-scriptet vil i dette tilfelle inkludere et klientside-script (f.eks. JavaScript) sammen med HTML-koden som sendes tilbake til klienten. Dette er fullt ut mulig, og scriptene vil kjøre helt uavhengige av hverandre.

Tema i dette kurset er scripting på tjenersiden.

Opphavsrett: Forfatterne og Stiftelsen TISIP

1.3.7 Scripting kontra programmering

Til nå har vi kun konsentrert oss om scripting på klient og tjener. Vi har så langt ikke definert hva vi mener med scripting. For å være helt nøyaktig er det en forskjell mellom et script og et program. Både når vi lager script, og når vi lager et program, bruker vi kommandoer som utfører et eller annet (f.eks. viser klokka på skjermen). Forskjellen ligger i måten et script og et program kjøres på.

Når et script kjøres, så vil scriptet tolkes en og en linje. Denne tolkingen sørger for at kommandoene blir gjort om til maskinkode. Denne maskinkoden består kun av 0'ere og 1'ere fordi det er kun dette datamaskinen forstår. Hvis scriptet skal kjøres på nytt, så vil det tolkes på nytt. Et script vil derfor være lesbart for mennesker. Omformingen mellom script-kode og maskinkode skjer automatisk når vi starter scriptet, og er derfor ikke noe vi mennesker trenger å bry oss med.

Når vi lager et program *må* det kompileres før det kjøres. Akkurat som for script vil programmet bestå av kommandoer (programkode), men under kompilering vil programmet gjøres om til maskinkode. Det vil derfor ikke være mulig å lese et program etter at det er kompilert. Kompilert programkode består kun av 0'ere og1'ere og er derfor uforståelig for oss mennesker. Denne omformingen mellom lesbar kode og maskinkode skjer altså kun én gang (i motsetning til script hvor omformingen gjøres hver gang).

Er det så mulig å bruke programmering i stedet for scripting på web? Ja det er det. Dette er ikke noe vi har diskutert så langt, men det er mulig med programmering både på klient- og tjenersiden.

Programmering på klientsiden

Du har kanskje hørt om Java-appleter og ActiveX. Dette er teknologier som gjør det mulig å kjøre klientprogrammer i stedet for klient-script i nettleseren. I dette tilfellet vil det da lastes ned ferdig kompilert programkode som kjøres inne i nettleseren. Det som er fordelen med programmering i forhold til scripting er at en ofte har bedre kontroll og større muligheter til å skreddersy løsningen. Sikkerheten blir også ofte større i programmer. Det er flere grunner til dette. Blant annet vil ikke brukeren ha mulighet til å lese og forandre koden.

Programmering på tjenersiden

Programmering på tjenersiden har ikke den samme sikkerhetskonsekvensen som klientside-programmering har. Grunnen til dette er at man i ingen av tilfellene vil se noe til koden (jeg snakker her om web). Klienten vil kun få returnert resultatet av kjøringen, mens kjøringen av koden vil foregå på tjeneren. Både JSP og JSF kan kommunisere med vanlige Java-klasser, og det er også slik at både JSP og JSF i realiteten oversettes til Java-klasser, som så kompileres.

Støtte hos nettleserne

Det er mulig å slå av støtte for f.eks. JavaScript (og Java-appleter) hos de fleste nettleserne. Ideelt sett bør en derfor lage applikasjonene uavhengig av JavaScript. I praksis skjer imidlertid dette sjelden, da det er meget ressurskrevende å lage og vedlikeholde slike parallelle løsninger. Når dette er sagt, kontroll av inndata bør alltid ligge på tjenersiden, uavhengig av hva som ligger på klientsiden. Grunnen til dette er at det kan komme til å bli laget klienter som ikke utfører den nødvendige kontrollen.

Opphavsrett: Forfatterne og Stiftelsen TISIP

1.4 Språkelementene i JSP

1.4.1 Elementer i JSP

Her følger en tabell over de ulike elementene i JSP:

JSP-element	Kode	Kommentar
HTML-kommentar	`<!-- kommentar -->`	Kommentar som sendes til nettleseren sammen med resten av HTML-koden.
JSP-kommentarer	`<%-- kommentar --%>`	Kommentar som ikke sendes til nettleseren.
Direktiv	`<%@ page xxx %>` f.eks. `<%@ page import = "java.util.Date" %>`	Et direktiv kan få JSP'n til å oppføre seg på en spesiell måte. *import* er en av flere muligheter.
Uttrykk (expressions)	`<%= uttrykk %>`	Skriver ut verdier til nettleseren.
Scriptlet	`<% java-kode %>`	Utfører java-kode.
Deklarasjon	`<%! deklarasjon %>`, f.eks. `<%! int tall; %>`	Brukes for å deklarere metoder og ikke-lokale variabler.

Alle elementene over plasseres direkte i HTML-koden. Se f.eks. JSP-koden tidligere i leksjonen der *uttrykk* brukes for å skrive ut dato og klokkeslett. Merk at det ikke er mulig å nøste JSP-elementer. Det medfører at du f.eks. ikke kan ha uttrykk inne i en scriptlet.

Nedenfor følger noen eksempler på de ulike elementene.

Kommentarer

HTML-kommentar

```
<!-- dette er en HTML-kommentar -->
```

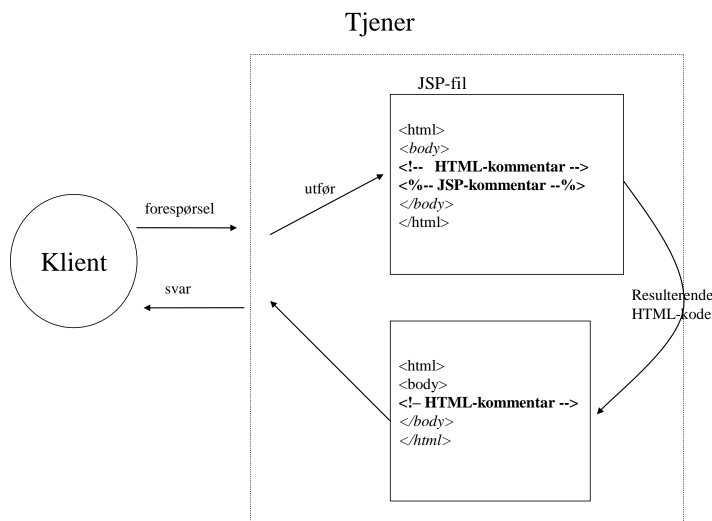
sendes til nettleseren, men vil ikke presenteres av nettleseren. Dette er kommentarer du vil se hvis du bruker *Vis/Kilde* i nettleseren.

JSP-kommentar

```
<%-- dette er en JSP-kommentar --%>
```

sendes ikke med til nettleseren. JSP-kommentarer fungerer derfor kun som kommentarer til de som skal lese selve JSP-koden. Legg merke til at det er enkelt å kommentere ut andre JSP-elementer (direktiv, uttrykk og scriptlet'er). Det er bare å legge til -- (to bindestrek) etter <% og før %> slik at vi får <%-- ett eller annet --%>.

Opphavsrett: Forfatterne og Stiftelsen TISIP



Figur 10: Bruk av kommentarer

Direktiv

Det finnes flere direktiv. Vi bryr oss om kun et foreløpig, nemlig

```
<%@ page import= ”XXX” %>
```

Denne gir oss mulighet til å importere java-klasser på samme måte som vi bruker *import* i vanlige java-programmer. XXX skiftes ut med den klassen vi vil importere. Vil vi importere flere klasser kan dette gjøres ved å skille dem fra hverandre med et komma.

Eksempel:

```
<%@ page import= ”java.util.Date” %>
```

i JSP tilsvarer

```
import java.util.Date;
```

i en java-fil. Merk at JSP-varianten ikke slutter med semikolon!

Etter vi har importert klassen kan den brukes på vanlig måte i en *scriptlet* eller et uttrykk.

Uttrykk (expressions)

Vi har allerede brukt denne type JSP-element i tidligere eksempel. Uttrykk starter med <%= og slutter med %>. Linjen

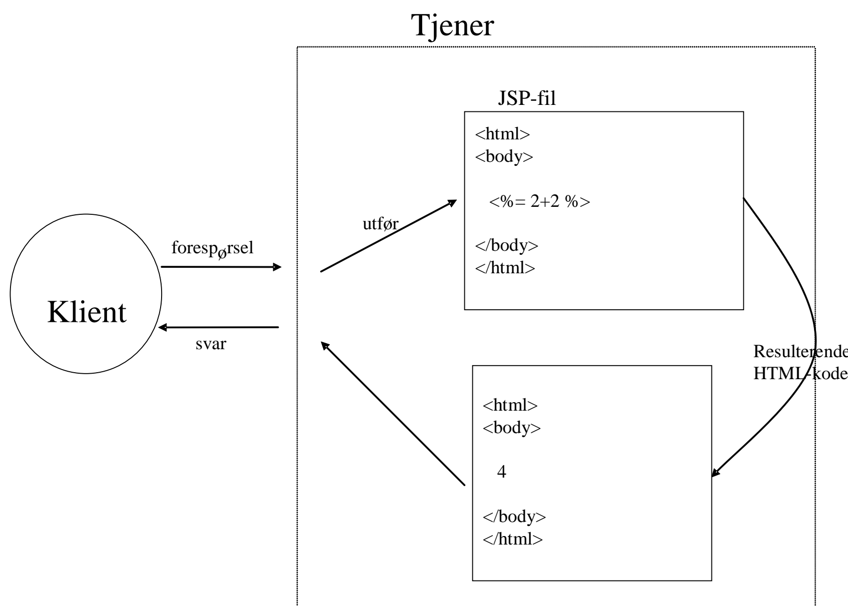
```
<h1> Klokka er:  <%= new java.util.Date() %> </h1>
```

inneholder et slikt uttrykk. Resultatet av linjen over er at klokken og datoen blir skrevet ut. Uttrykk fungerer også for primitive datatyper. F.eks. vil

```
<%= 2+2 %>
```

sørge for at 4 blir sendt til nettleseren. Figuren under viser hva som skjer. **Merk deg at det ikke skal være semikolon til slutt i et uttrykk**. Du vil da få feilmelding (og ikke en direkte selvforklarende en)!

Opphavsrett: Forfatterne og Stiftelsen TISIP



Figur 11: Uttrykk

Scriptlets

En scriptlet er et element som starter med <% og slutter med %>. Det som settes inni en slik scriptlet er java-kode. Vi vil derfor kontrollere hva som skjer i vår JSP med scriptlet'er. Vi kan bruke alle typer kontrollstrukturer, objekter osv. som vi er vant med fra java (vi kan også bruke vanlige java-kommentarer med //).

Eksempel:

```
<body>
      <% for (int i=0; i < 2; i++) {
            int j = i;
         }
      %>
</body>
```

Hva vises i klienten i dette tilfellet? Ingenting. Det som skjer i scriptlet'en i eksempelet er at vi går igjennom for-løkka og setter verdien til en variabel j, men dette sendes ikke til nettleseren. Vi kan imidlertid bruke scriptlets til å sørge for output til nettleseren. Dette kan vi f.eks. gjøre ved å "kontrollere HTML-koden".

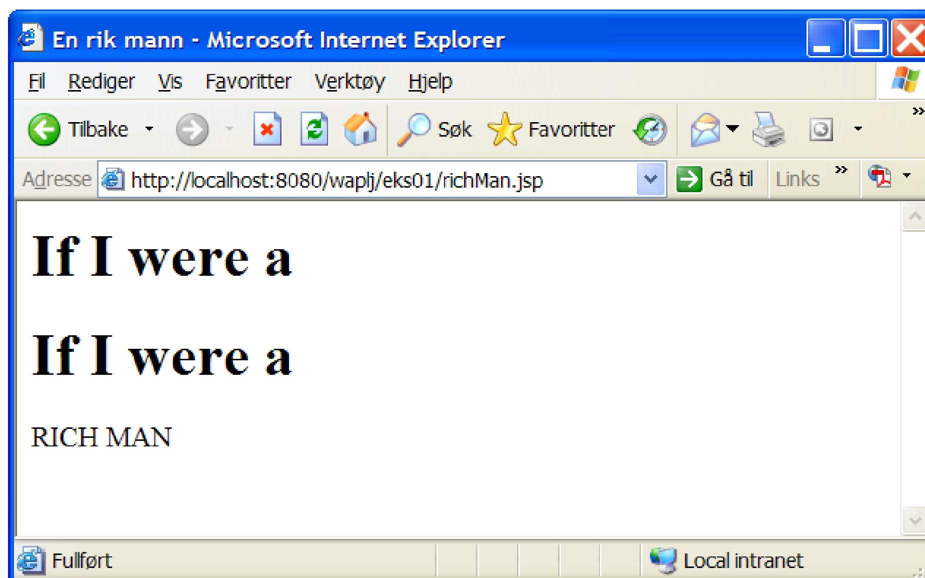
Eksempel:

```
<!-- Filnavn eksempelsamling 1: richMan.jsp -->
<!-- Løkke som kontrollerer HTML-koden -->
<html>
<head><title>En rik mann</title>
</head>

<body>
<%
  for (int i = 0; i < 2; i++) {
    %>
```

Opphavsrett: Forfatterne og Stiftelsen TISIP

```
      <h1> If I were a </h1>
    <%
    }
%>
RICH MAN
</body>
</html>
```

Figuren under viser resultatet. Vi ser at linjen <h1> If I were a </h1> blir utført to ganger. Dette skjer pga. av for-løkka. Vi ser at alt mellom de to klammeparentesene { og } blir utført som en del av løkka. Scriptlet'er brukes derfor til å styre det som skjer i en JSP.



Figur 12: Kjøring av eksempel

Deklarasjoner

Når det gjelder deklarasjoner er det viktig å huske følgende:

Hver JSP-fil blir oversatt til en Java-klasse (mer nøyaktig: en servlet-klasse, dvs en klasse som arver fra *HttpServlet*). Første gangen JSP-siden blir etterspurt opprettes et objekt av klassen[1], og en metode ved navn *init()* kjøres. Deretter kjører en metode *service()*. Hver nedlasting av siden genererer et kall på enten *doPost()* eller *doGet()* på vegne av dette ene objektet. Disse metodene kaller igjen *service()*. Objektet lever vanligvis inntil web-tjeneren stopper.

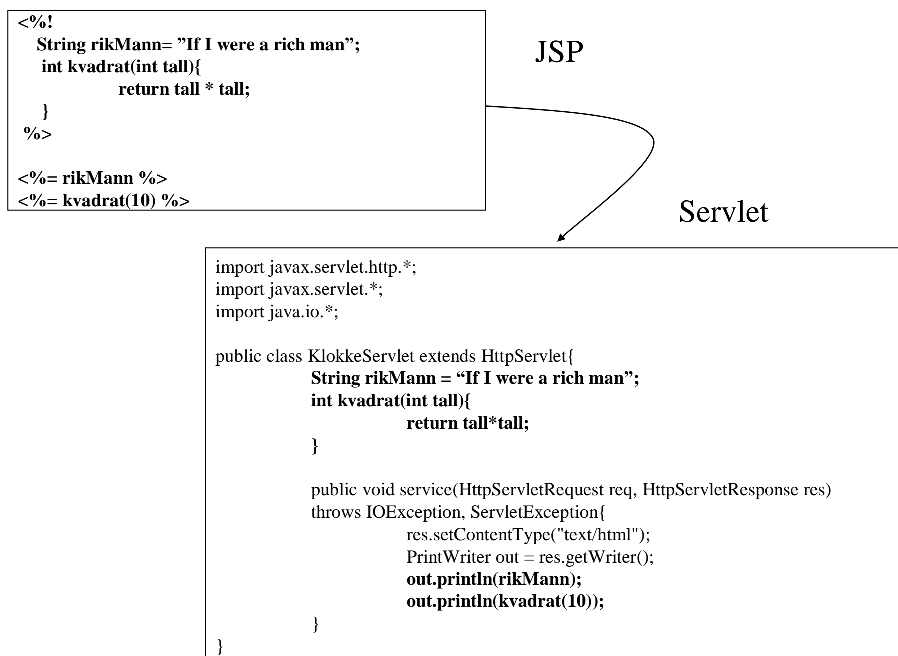
(I NetBeans får du fram java-koden ved å ta fram jsp-filen under fliken *Projects*, høyreklikke på den og velge "View servlet".)

Figuren nedenfor viser en JSP med en tilsvarende servlet-klasse. Servlet-klassen du finner i NetBeans (eller hos andre web-tjenere) er atskillig mer komplisert, men det kommer av at den er laget automatisk og skal dekke mange flere tilfeller enn akkurat vår lille servlet.

[1] Dette er det vanlige, men hvis du krever at kun en tråd med *service()*-metoden skal kjøre av gangen, *kan* web-tjeneren løse dette ved å opprette mer enn ett objekt av klassen. Denne type konfigurasjoner forholder vi oss ikke til nå.

Opphavsrett: Forfatterne og Stiftelsen TISIP

I JSP-en finner vi deklarasjon av en variabel (strengen *rikMann*) og av en metode (*kvadrat()*). Legg merke til hvordan du finner disse igjen i klassen *KlokkeServlet*.



Figur 13: JSP med tilsvarende servlet-klasse

Alle som etterspør den aktuelle siden, henvender seg til det *samme* objektet. Det betyr at eventuelle objektvariabler (instance variables) kan oppdateres av flere klienter samtidig – med de fordeler, og spesielt de ulempene dette medfører. Generelt bør vi være veldig forsiktig med å bruke objektvaraibler på grunn av at de kan oppdateres av flere klienter samtidig (jmf. transaksjonsbegrepet som en del av dere kjenner fra databasehåndtering.)

Hver henvendelse til siden starter opp *service()*-metoden i en ny tråd. Det vil si at hver klient får sin egen utgave av denne metoden – og dermed sitt eget sett av *lokale* variabler. Deklarasjoner inni scriptlets blir lokale variabler i *service()*-metoden.

Sammendrag:

- Ett objekt pr JSP-side som lever fra første gang noen etterspør siden og inntil web-tjeneren stopper.
- Objektvariablene (variabler deklarert i <%! .. %>) blir derfor felles for alle klienter.
- Et sett lokale variabler (variabler deklarert inni scriptlets) opprettes pr henvendelse.

Eksempler på objektvariabler:

```
<html>
...
<%! double tall ;
    int antallGriser = 1000;
    String s = “Min dag i dag”;
    Date iDag = new Date();
%>
AntallGriser: <%= antallGriser %>
...
</html>
```

Opphavsrett: Forfatterne og Stiftelsen TISIP

Disse variablene er altså felles for alle klienter som henvender seg til denne siden, og de lever så lenge som objektet lever, vanligvis til web-tjeneren tas ned.

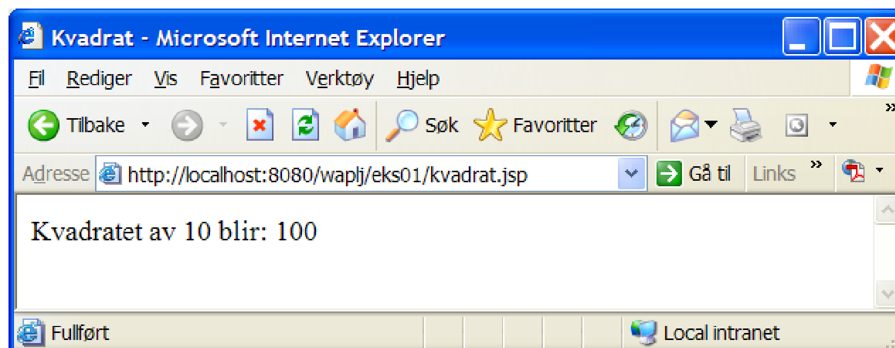
Vi vender så tilbake til metoden fra Figur 13:

```
<%! int kvadrat(int innTall){
       return innTall*innTall;
    }
%>
```

Metoden kan brukes i både scriptlet'er og uttrykk. F.eks. kan vi bruke den på følgende måte (resultatet vises i figuren under):

```
<!-- Filnavn eksempelsamling 1: kvadrat.jsp -->
<!-- Deklarasjon og bruk av metode -->

<html>
<head><title>Kvadrat</title>
</head>
<body>
<%! int kvadrat(int innTall){
       return innTall*innTall;
    }
%>
Kvadratet av 10 blir: <%= kvadrat(10) %>
</body>
</html>
```

Kjøring av scriptet:



Figur 14: Kjøring av jsp med metodekall

1.4.2 Innebygde objekter i JSP

Det fins flere ferdigdefinerte objekter i JSP (det betyr at de blir opprettet i den Java-klassen som JSP'en oversettes til – du skal ikke opprette dem i den koden du skriver).

I dette kurset trenger du å vite om følgende:

- *request*: Dette objektet beskriver klientens forespørsel, og det tilhører typen *HttpServletRequest*, som er et sub-interface til *ServletRequest*. Objektet er sentralt i den videre gjennomgangen av JSP og gjennomgås i leksjon 2.
- *response*: Dette objektet beskriver svaret web-tjeneren skal sende til klienten, og det tilhører typen *HttpServletResponse*.

Opphavsrett: Forfatterne og Stiftelsen TISIP

- *session*: Dette objektet beskriver klientens sesjon, og det tilhører typen *HttpSession*. Sesjoner er tema i leksjon 4.
- *out*: Dette objektet tilhører klassen *JspWriter* og gjør det mulig å skrive direkte til klientens HTML-side. Vi skal se nærmere på dette objektet nedenfor.

Med unntak av *JspWriter* hører alle de nevnte interfacene til pakken *javax.servlet.http*. API-dokumentasjonen finner du på

http://java.sun.com/products/servlet/2.5/docs/servlet-2_5-mr2/index.html

JspWriter har for våre formål samme funksjonalitet som *PrintWriter* som du kjenner fra Java SE.

Objektet out

Vi skal se litt nærmere på *out*-objektet. Vi har så langt brukt <%= … %> for å sende dynamisk innhold til nettleseren. Vi har sett hvordan en JSP svarer til en servlet-klasse. Figur 13 viser dette, samt at vi oppretter et *out*-objekt med følgende java-kode:

```
PrintWriter out = res.getWriter();
```

Etter at dette objektet er opprettet, kan det brukes til å sende HTML-kode til nettleseren.

I JSP er det et ferdigdefinert *out*-objekt av klassen *javax.servlet.jsp.JSPWriter*. Denne klassen har mange metoder, men den vi vil bruke mest er nok *println()*. Denne metoden gjør omtrent det samme som et *uttrykk*. Under vises et eksempel der begge disse måtene er brukt. Hvilket av eksemplene er mest oversiktlig?

<table>
<tr><th>Eksempel med out</th><th>Eksempel med uttrykk</th></tr>
<tr><td><pre>
<!-- Filnavn eksempelsamling 1:
      eks_med_out.jsp -->
<!-- Eksempel der out-objektet brukes
     for å sende dynamisk innhold
     til klienten -->
<html>
<head><title>Bruk av out</title>
</head>
<body>
<h1>Eksempel med out</h1>
<%
    boolean sann = true;
    String streng = "sann ...";
    if (sann) out.println(streng);
    else out.println("usann");
%>
</body>
</html>
</pre></td><td><pre>
<!-- Filnavn eksempelsamling 1:
     eks_med_uttrykk.jsp -->
<!-- Eksempel der uttrykk brukes
     for å sende dynamisk innhold
     til klienten -->
<html>
<head><title>Uttrykk</title>
</head>
<body>
<h1>Eksempel med uttrykk</h1>
<%
    boolean sann = true;
    String streng = "sann ...";
    if (sann) { %>
        <%= streng %>
    <% } else { %>
        <%= "usann ..."%>
    <% }
%>
</body>
</html>
</pre></td></tr>
</table>

Dette bør kunne gi en indikasjon på at det er tilfeller der det er greiere å bruke *out*-objektet i stedet for *uttrykk*. Merk at begge disse metodene vil sørge for å sende ”noe” til klienten. Vi kan derfor godt ha HTML-kode i en *out.println()*-kommando eller inne i *uttrykk*.

Opphavsrett: Forfatterne og Stiftelsen TISIP

```
<html>
. . . .
<% out.println(”<body>”); %>
    <h2> her blander vi litt </h2>
<%= ”</body>” %>
</html>
```

Resultatet vil for klienten se slik ut:

```
<html>
. . . .
<body>
    <h2> her blander vi litt </h2>
</body>
</html>
```

Du vil etter hvert gjøre deg opp en mening om når det er best å bruke HTML-kode, *out.println()* eller *uttrykk.*

1.4.3 Utskrift inne i metoder

out-objektet (som de øvrige ferdigdefinerte objektene) er en lokal variabel i service()-metoden. Hvis du vil skrive ut inni en metode, må du sende med *out*-objektet som argument. Dette kan f.eks. gjøres slik:

```
<!-- Filnavn eksempelsamling 1: skrivUt.jsp -->
<!-- Eksempel med utskrift inni metode -->
<html>
<head>
<title>Utskrift fra metode</title></head>
<body>
    <% skrivUt(out); %>
</body>
</html>

<%! void skrivUt(JspWriter outMetode){
      try{
        outMetode.println("Skriver ut fra min egen metode");
      } catch(Exception e){}
   }
%>
```

Vi ser at scriptlet'en kaller metoden, med *out*-objektet som argument. Legg merke til at vi må bruke try/catch i denne sammenhengen ellers vil vi få kompileringsfeil.

Opphavsrett: Forfatterne og Stiftelsen TISIP