

Björn Persson

# Komponenter med J2EE

## Del 1

Komponentbaserad applikationsutveckling  
april 2005

## Om denna sammanfattning

Detta är del 1 av sammanfattningen och behandlar teknologier som bör förstås för att bättre kunna skapa applikationer med EJB (*Enterprise JavaBeans*). En stor del av sammanfattningen behandlar servlets och JSP (*Java ServerPages*) då dessa är användbara vid skapande av webbgränssnitt mot EJB. Del 2 behandlar främst EJB och relationer mellan EJB.

Avsikten med denna sammanfattning är att ge en introduktion till Sun's Java 2, Enterprise Edition (J2EE) och hur man skapar komponenter (d.v.s. EJB) med J2EE. Sammanfattningen har skrivits av författaren för att lära sig, och just sammanfatta, hur dessa teknologier fungerar. Detta är **inte en ersättning till eventuell kurslitteratur!**

Kapitlen i denna sammanfattning har skrivits så att de är oberoende av varandra, d.v.s. det bör vara säkert att hoppa över kapitel som inte är av intresse. Senare kapitel använder sig dock av teknologier som beskrivs i tidigare kapitel samt det kan finnas referenser till tidigare kapitel.

Koden i denna sammanfattning har skapats med Java 2, Enterprise Edition (J2EE) SDK version 1.4.x på datorer med operativsystemen Windows XP/2003. För att köra EJB så har Resin v. 3.0.x använts ([www.caucho.com](http://www.caucho.com) – om en annan EJB-server används så kan exempel behöva justeras för att fungera). Som databashanterare har främst MySQL använts ([www.mysql.com](http://www.mysql.com)), men vissa exempel har även testats i Oracle ([www.oracle.com](http://www.oracle.com)). Övrig programvara som använts är Java-programmeringsmiljön JCreator ([www.jcreator.com](http://www.jcreator.com) – för att skapa EJB och servlets), text-editorn Crimson Editor ([www.crimsoneditor.com](http://www.crimsoneditor.com) – för att skapa *deployment descriptors*, webbsidor, JSP samt skärmdumpar av mappstrukturer) och Microsoft Visio 2003 för klass- och sekvensdiagram (samt "givetvis" Word 2003 för detta dokument samt CutePDF och Adobe Acrobat för att skapa PDF ☺).

## Konventioner i sammanfattningen

Vissa engelska begrepp saknar generellt accepterade översättningar och är därför skrivna på engelska för att kunna relatera till begreppen i engelsk litteratur. I de fall då översättning används så följs det översatta ordet första gången med det engelska inom parentes. Dessa ord skrivs i kursiv stil för att visa att de har "lånats", t.ex. *servlet* och Java-böner (*Java beans*).<sup>1</sup>

Kod har skrivits med typsnitt av fast bredd (Courier New) för att göras mer lättläst samt längre exempel har inneslutits i en ram (se exempel nedan).

```
enVariabel = 1 + 2 //Kommentarer i kod skriv med fet stil
```

Jag är givetvis tacksam för alla konstruktiva synpunkter på sammanfattningens utformning och innehåll.

Eskilstuna, april 2005

Björn Persson, e-post: [bjorn.persson@mdh.se](mailto:bjorn.persson@mdh.se)

Ekonomihögskolan, Mälardalens högskola

Personlig hemsida: <http://www.eki.mdh.se/personal/bpn01/>

---

<sup>1</sup> Eftersom många termer i J2EE saknar bra översättning, t.ex. *servlets* och *deployment descriptor*, så skrivs dessa endast med kursiv stil i inledande avsnitt.

# Innehållsförteckning

<b>1</b>	<b>KOMPONENTTEKNOLOGIER.....</b>	<b>5</b>
1.1	CORBA, J2EE och COM.....	5
<b>2</b>	<b>JAVA 2, ENTERPRISE EDITION (J2EE).....</b>	<b>6</b>
2.1	Vad är J2EE?.....	6
2.2	Varför använda J2EE? .....	7
2.2.1	Fler fördelar med J2EE .....	7
2.2.2	Nackdelar med J2EE (och andra applikationsserverar).....	7
2.3	J2EE-applikationer.....	8
2.3.1	Klienter .....	8
2.3.2	Datakällor .....	8
2.3.3	Affärslogiken .....	8
2.3.4	Kommunikationsprotokoll .....	8
2.4	Teknologier i J2EE .....	8
<b>3</b>	<b>JAVA NAMING AND DIRECTORY INTERFACE (JNDI).....</b>	<b>10</b>
3.1	Användningsområden för JNDI i J2EE.....	10
3.2	Exempel på några namntjänster inom J2EE.....	11
3.3	Namntjänster i JNDI .....	11
3.3.1	Paketet Naming.....	11
3.4	Katalogtjänster i JNDI .....	13
3.4.1	Paketet Directory .....	13
3.5	J2EE vs. COM/COM+ .....	14
<b>4</b>	<b>SERVLETS.....</b>	<b>15</b>
4.1	HyperText Transport Protocol (HTTP).....	15
4.2	Historiken bakom servlets.....	15
4.3	Klasser och gränssnitt för servlet .....	16
4.3.1	Generiska servlets .....	16
4.3.2	HTTP servlets .....	17
4.3.3	Exempel på en enkel HTTP-servlet .....	17
4.3.4	Kompilera och installera servlet.....	18
4.4	Funktioner i servlets.....	19
4.4.1	Inkludera webbsurser .....	19
4.4.2	Servlets och formulär.....	20
4.4.3	Sessionshantering.....	22
4.4.4	Använda JavaBeans .....	26
4.4.5	<i>redirect</i> och <i>forward</i> i servlets .....	28
4.5	Mer om servlets.....	30
4.5.1	SingleThreadModel.....	30
4.5.2	Servlets och alias.....	31
4.5.3	Nackdelen med servlets .....	31
<b>5</b>	<b>JAVA SERVER PAGES (JSP).....</b>	<b>32</b>
5.1	Strukturen av en JSP .....	32
5.1.1	Text.....	32
5.1.2	Kommentar .....	32
5.1.3	Direktiv .....	32
5.1.4	Skriptelement.....	33
5.1.5	Händelser ( <i>actions</i> ).....	34
5.1.6	”Inbyggda” objekt.....	35
5.2	Funktioner i JSP.....	35
5.2.1	Selektion och HTML-kod .....	35
5.2.2	Inkludera webbsurser .....	36
5.2.3	JSP och formulär.....	36
5.2.4	Sessionshantering.....	38
5.2.5	Använda JavaBeans .....	43
5.2.6	<i>redirect</i> och <i>forward</i> i JSP .....	46
<b>6</b>	<b>SERVLETS OCH JSP I SAMARBETE .....</b>	<b>49</b>
6.1	HTML-formulär i JSP och servlet.....	49
6.1.1	Filer i exempel .....	49
6.2	Servlet som styr exekvering.....	51
6.2.1	Filer i exempel .....	52

<b>7</b>	<b>REMOTE METHOD INVOCATION (RMI)</b> .....	<b>55</b>
7.1	Introduktion.....	55
7.2	Skapa och publicera distribuerade objekt.....	56
7.2.1	Definiera remote-gränssnitt (Hello) .....	56
7.2.2	Implementera remote-gränssnitt (HelloServer).....	57
7.2.3	Generera stub (och skeleton) (HelloServer_Stub).....	57
7.2.4	Starta RMI-register .....	57
7.2.5	Registrera en instans av objekt och binda till ett namn .....	58
7.2.6	Testköra distribuerat objekt med en klient (HelloClient).....	59
7.3	Dynamiskt skapade av objekt.....	59
7.4	RMI och Internet Inter-ORB Protocol (IIOP) .....	60
7.5	RMI och EJB.....	60
<b>8</b>	<b>EXTENSIBLE MARKUP LANGUAGE (XML)</b> .....	<b>61</b>
8.1	Ett XML-dokuments struktur .....	61
8.1.1	Taggar .....	61
8.1.2	Attribut.....	61
8.1.3	Document Type Definition .....	61
8.1.4	Stilmallar.....	61
8.2	Användningsområden för XML-dokument.....	62
8.3	Olika typer av XML-tolkar .....	62
8.4	Nackdelen med XML.....	62
<b>9</b>	<b>LITTERATURFÖRTECKNING</b> .....	<b>63</b>
9.1	Webbadresser till förlag .....	63
9.2	Webbadresser .....	63

# 1 Komponentteknologier

En **komponent** är en självständig enhet av mjukvara som på egen hand eller tillsammans med andra komponenter används för att bygga upp en applikation. Komponenter kan användas i både en- och fleranvändarsystem, i det senare fallet kallas de ibland för distribuerade komponenter då de ofta anropas över nätverk.

Komponentteknologier är en typ av mjukvaruarkitektur som ofta bygger på en treskiktad (eller n-skiktad) lösning: gränssnitt, affärslogik och datalager. Gränssnittet är mjukvara som kommunicerar med användaren av applikationen, affärslogiken sköter det mesta av exekveringen och datalagret lagrar data permanent. Tanken med denna arkitektur är bl.a. att vi kan använda s.k. tunna klienter (bl.a. webbläsare) som jobbar mot affärslogiken – mjukvara som centraliserats till en eller flera kraftfulla servrar. Affärslogiken har separerats från datalagret för att göra den oberoende av datakällor (t.ex. databashanterare).

Det finns idag tre huvudsakliga komponentteknologier: OMGs CORBA, Suns J2EE och Microsofts COM/COM+ (samt numera även .NET).

---

## 1.1 CORBA, J2EE och COM

**CORBA** (Common Object Request Broker Architecture) är en öppen standard från OMG (Object Management Group) samt grunden till både J2EE och COM. CORBA har skapats för att vara oberoende av bl.a. programmeringsspråk.

**J2EE** (Java 2, Enterprise Edition) är en ren Java-miljö, d.v.s. komponenterna skapas i programspråket Java, och har skapats av Sun<sup>2</sup>. Komponentteknologin heter Enterprise Java Beans (EJB) och är tillsammans med flera andra standarder (t.ex. JDBC) en del av J2EE. Delar av J2EE ligger till grund för delar av version 3 av CCM (CORBA Component Model).

**COM**<sup>3</sup>, Component Object Model, (samt COM+ och i viss mån även .NET) är Microsofts egna standard för komponenter, d.v.s. den bygger på Windows som operativsystem (även om .NET är tänkt att kunna användas på andra operativsystem). Utvecklaren kan välja ett programmeringsspråk av eget tycke så länge det är ”COM-kompatibelt”, d.v.s. följer specifikationerna för COM. Microsoft själva tillhandahåller språken Visual Basic, Visual C++ och Visual J++, men andra språk som Borlands Delphi kan även användas. COM+ är en utökning av COM samt integration av MTS (Microsoft Transaction Server).

Microsofts .NET är en ”uppföljare”<sup>4</sup> till (eller vidareutveckling av) COM/COM+. Tanken med .NET är att den inte längre ska vara beroende av operativsystemet Windows, d.v.s. .NET-applikationer ska kunna exekvera på även andra operativsystem. Microsoft har också gått ett steg längre och kräver att programmeringsspråk i .NET ska skapa moduler (bl.a. komponenter) som kan användas från alla andra programmeringsspråk. .NET är idag byggt ovanpå operativsystemet och använder sig av COM/COM+-tjänsterna som operativsystemen ger.

---

<sup>2</sup> Sun är skapare av Sun Sparc-datorer och operativsystemet Solaris, en UNIX variant.

<sup>3</sup> COM benämns ibland som DCOM, Distributed COM. Men DCOM är egentligen den del av COM som hanterar kommunikation mellan klienter och komponenter på olika datorer, d.v.s. en förlängning av COM som gör COM till en distribuerad komponentteknologi.

<sup>4</sup> Jag har satt ordet ”uppföljare” inom parentes då den är ett skal ovanpå operativsystemet, d.v.s. .NET är beroende av MTS/COM+ som applikationsserver. Resultatet är att .NET-applikationer har många av de begränsningar som MTS/COM+-applikationer har. .NET har alltså **inte** ersatt COM/COM+ fullständigt.

## 2 Java 2, Enterprise Edition (J2EE)

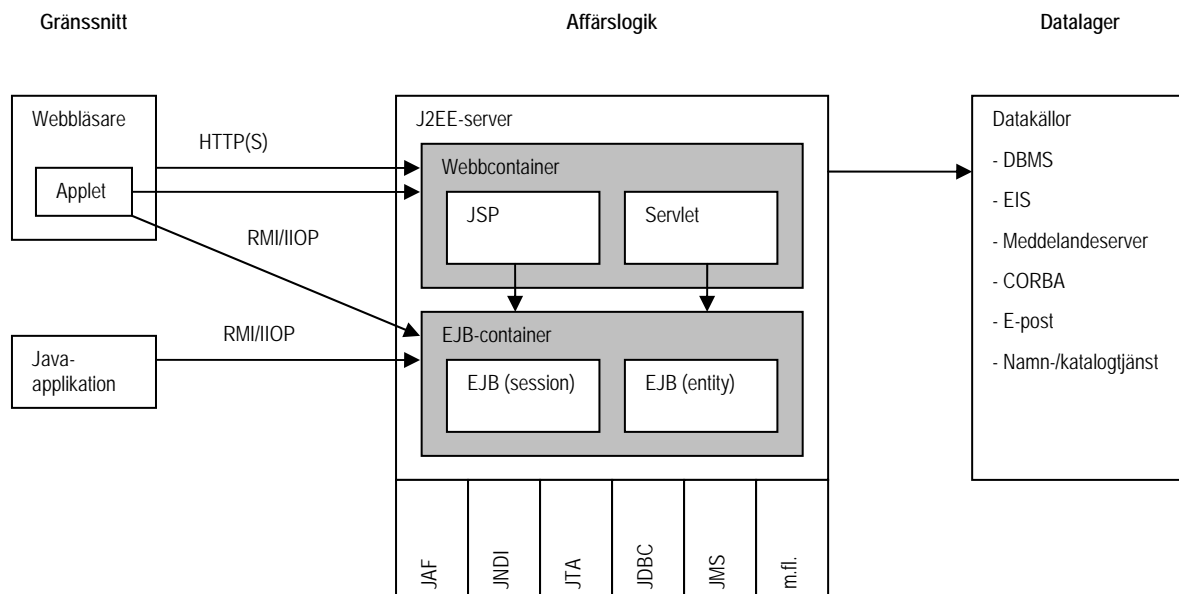
J2EE är en utökning av Java 2, Standard Edition (J2SE), d.v.s. J2EE behöver J2SE för att fungera. J2SE är "klientdelen", d.v.s. ska installeras på klientdator<sup>5</sup>, och innehåller bl.a. JDBC, RMI och JNDI som är fundamentala för J2EEs funktion. I och med version 1.4 av J2EE så levereras J2SE med i installationspaketet man laddar ner från Sun. Tidigare versioner av J2EE krävde att man laddade ner J2SE separat och installerade först.

### 2.1 Vad är J2EE?

J2EE är en samling av standarder som definierats av Sun, men som är tänkta att implementeras av andra. Med "andra" menas systemutvecklare, som gör egenutvecklade system (för internt bruk eller till kund), och mjukvaruföretag, som gör servermjukvaran (J2EE-servern) som de egenutvecklade systemen ska exekvera i. Sun har dock, ibland tillsammans med andra företag/organisationer, gjort referensimplementationer av servrar (t.ex. Tomcat, en servlet- och JSP-motor, med Apache).

Systemutvecklare skapar klasser som implementerar speciella gränssnitt som J2EE-servern kan kommunicera med för att meddela händelser, m.m.. Mjukvaruföretagen skapar servrar som implementerar andra gränssnitt som systemutvecklaren kan anropa för att erhålla tjänster från J2EE-servern. J2EEs standardgränssnitt är alltså ett kontrakt mellan systemutvecklarens klasser och J2EE-servern.

Komponentbaserade applikationer bygger på n-skiktslösningar, d.v.s. vi separerar gränssnitt från affärslogik och data.



I en J2EE-server exekverar servlet/JSP och EJB i containers. Containers används för att skapa en miljö för servlets/JSP och EJB, d.v.s. för att erbjuda tjänster till dessa, men de används även för att fånga upp alla metदानrop till objekten för att kunna erbjuda tjänsterna som applikationsserver erbjuder. Genom att använda containers, d.v.s. gränssnittet mellan J2EE-server och servlet/JSP/EJB, så kan mjukvaruföretag implementera J2EE-servrarna hur de vill.

<sup>5</sup> Behöver inte installeras om applikationer använder webbgränssnitt – då räcker det med en webbläsare.

---

## 2.2 Varför använda J2EE?

Allt fler applikationer idag kräver 24-timmars drift (ibland kallat 24 x 7 – 24 timmar om dygnet, 7 dagar i veckan). Detta kräver en utvecklingsplattform som är stabil och tillförlitlig men också säker.<sup>6</sup> Detta kräver ofta att vi använder transaktioner, ibland distribuerade transaktioner som kan spänna över fler olika typer av datakällor. Vi kan även behöva fördela exekveringen i applikationen över flera servrar för att minska svarstider eller rent av duplicera logiken på flera servrar för att kunna hantera många simultana begäran. Att duplicera affärslogiken gör också applikation mindre känslig för driftsstörningar, t.ex. att en server kraschar.

Och för att kunna hantera ett ökande behov så krävs även att utvecklingsplattformen är skalbar, d.v.s. kan utökas för att klara fler klienter. Ett sätt är att använda flera servrar för att distribuera exekveringen över flera processorer.

Tid och pengar har investerats i existerande system. För att inte låta dessa pengar gå till spillo så bör nyutvecklade system kunna kommunicera med existerande system. D.v.s. nya system bör kunna agera som ett skal mot existerande system.

J2EE är Suns utvecklingsplattform som är tänkt att klara av detta men ger även andra fördelar. (CORBA och COM ger liknande tjänster och fördelar.)

### 2.2.1 Fler fördelar med J2EE

Java är plattformoberoende, d.v.s. applikationer bör kunna flyttas mellan olika typer av datorer (operativsystem) utan att behöva kompileras om. Eftersom J2EE-applikationer bygger på standarder så bör även dessa kunna flyttas mellan J2EE-servrar från olika tillverkare.

Komponenter byggda som EJB är små självständiga programmoduler som kan skapas och testas oberoende av resten av J2EE-applikationen. EJB kan även återanvändas i framtida applikationer.

J2EE-servrar ger tjänster så som transaktionshantering, samtidighet (*concurrency*) och poolning av resurser. Detta är grundläggande funktioner som systemutvecklaren inte behöver skriva kod för om de använder en J2EE-server. Systemutvecklaren kan fokusera på att utveckla endast koden för affärslogiken.

### 2.2.2 Nackdelar med J2EE (och andra applikationsservrar)

Att använda sig av applikationsservrar som J2EE är inte bara en dans på rosor. Det finns nackdelar också. En av dessa nackdelar är prestanda – att utnyttja sig av J2EE-servrars tjänster är inte alltid det mest effektiva. Tjänsterna har gjorts generella för att kunna passa så många syften som möjligt. Men att utveckla tjänsterna själv tar också tid, d.v.s. ibland är det värt att göra avkall på prestanda för att vinna tid vid utveckling. Prestanda kan vi även vinna genom att använda en bättre processor eller flera processorer.

Ytterligare en nackdel är att objektmodellen, d.v.s. sättet vi utvecklar EJB, är mer komplext än om vi använt ”vanliga” objekt. Men när man väl tagit sig över inlärningströskeln så blir det rutin som med ”vanliga” objekt.

---

<sup>6</sup> Använd inte Resin installerad på Ekonomihögskolans utvecklingsserver Kompis som en referensram om den skulle strula. ☺

---

## 2.3 J2EE-applikationer

### 2.3.1 Klienter

En klient till en J2EE-applikation kan vara en webbläsare, en applet eller en Java-applikation<sup>7</sup>. Om klienten är en webbläsare så använder vi webbserver för att skapa gränssnittet till applikationen. På webbservern kan vi använda servlets och/eller JSP som då kommunicerar med affärslogiken (EJB). I fallet med applets så kan applet kommunicera antingen med servlets eller direkt med affärslogiken. Och i sista fallet så kommunicerar Java-applikationen oftast direkt med affärslogiken.

Denna del av sammanfattningen fokuserar på Suns teknologier för klienter och kommunikation med (och mellan) komponenter.

### 2.3.2 Datakällor

Datakällor är oftast relationsdatabaser men kan även vara existerande (*legacy*) applikationer. Genom att betrakta datakällor som ett separat lager så bör en datakälla från en tillverkar kunna ersättas med en datakälla från en annan tillverkare utan allt för mycket omskrivning av kod, om någon.

### 2.3.3 Affärslogiken

Det mesta av affärslogiken bör ske i EJB (*Enterprise Java Beans*). Genom att bygga upp affärslogiken med EJB så kan vi ersätta en EJB (som t.ex. innehåller en bugg) med en nyare version utan att klienter till EJB påverkas. Detta förutsätter givetvis att inte gränssnittet för EJB ändras. Som sagt tidigare så innebär utveckling av mindre moduler att dessa kan testas separat, d.v.s. mindre kod att buggtesta åt gången.

Den andra delen av denna sammanfattning fokuserar på Suns teknologier för affärslogik, d.v.s. komponenter.

### 2.3.4 Kommunikationsprotokoll

Java-klienter kommunicerar med EJB med *Java Remote Method Invocation* (Java RMI<sup>8</sup>). Men om vi använder servlets/JSP så kan vi använda HTTP (eller HTTPS) mellan webbläsare och servlets/JSP, som i sin tur kommunicerar med EJB (med t.ex. Java RMI).

---

## 2.4 Teknologier i J2EE

J2EE består av ett antal olika teknologier. Varje teknologi har sitt eget versionsnummer – ett versionsnummer som inte följer J2EE:s versionsnummer. J2EE har nått version 1.4 (040602) medan EJB nått version 2.1 och servlets version 2.4.

Resterande kapitel i denna sammanfattning behandlar några av de ”viktigaste”<sup>9</sup> teknologierna i J2EE.

- *Java Naming and Directory Interface* (JNDI) – gränssnitt mot namntjänster.
- *Java Servlet* – webbkomponenter

---

<sup>7</sup> Klienter kan även vara skrivna i andra programmeringsspråk, t.ex. VB/VB.NET (vilket för övrigt är ganska vanligt). Detta kräver dock en form av brygga mellan teknologierna.

<sup>8</sup> RMI använder i sin tur Java Remote Method Protocol (JRMP – Suns egna protokoll) eller Internet Inter-ORB Protocol (RMI-IIOP). RMI (i alla fall JRMP) motsvara Microsofts Distributed COM (DCOM).

<sup>9</sup> Med viktigaste menar jag de mest använda teknologierna.



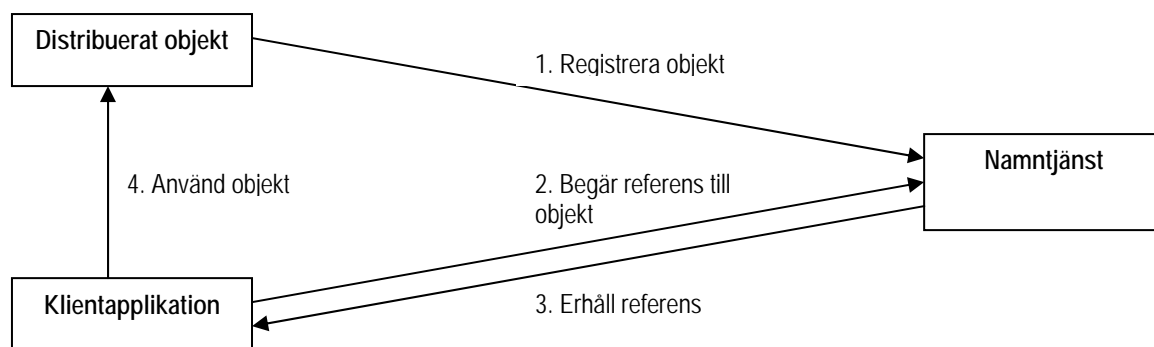
- *Java Server Pages* (JSP) – webbkomponenter
- *Remote Method Invocation* (RMI) – protokoll för kommunikation med distribuerade objekt.
- *Enterprise Java Beans* (EJB) – komponentteknologi (se del 2 av sammanfattning).

Sist (i denna del av sammanfattning) beskrivs även XML kort, trots att det inte är en Sun-teknologi. ☺

### 3 Java Naming and Directory Interface (JNDI)

JNDI ger ett enhetligt gränssnitt (API) mot namn- och katalogtjänster (*naming and directory services*). En **namntjänst** kopplar samma ett namn med ett eller flera värden. Ett typiskt exempel är DNS (Domain Name Service) som översätter webbadresserna som vi anger i våra webbläsare till IP-adresser som vår dator kan använda.<sup>10</sup>

I Java så registreras distribuerade objekt (t.ex. EJB) i en namntjänst så att klienter kan fråga namntjänster om en referens till objektet. Klienter använder sen referenserna för att kommunicera med objekten (d.v.s. anropa metoder via referensen). Namntjänsten är bara involverad i att den inledande fasen, d.v.s. att förse klienten med en referens till det distribuerade objektet. Därefter sker alla kommunikation direkt med distribuerade komponenter.



En **katalogtjänst** är en form av databas som innehåller data om t.ex. användare, datorer och andra objekt i ett nätverk. De flesta katalogtjänsterna bygger på den internationella standarden X.500 och brukar även tillåta åtkomst via Lightweight Directory Access Protocol (LDAP). Exempel på kommersiella katalogtjänster är Novell Directory Service (NDS, numera eDirectory) och Microsofts Active Directory (AD)

För att JNDI ska fungera krävs en *service provider* – en drivrutin, ofta från tillverkaren av katalogtjänsten – för aktuell namn- eller katalogtjänst. Sun ger ut drivrutiner för bl.a. LDAP, COS Naming (CORBA), RMI Registry (Java), NIS, DSML, DNS och filsystem.<sup>11</sup> (En *service provider* implementerar gränssnittet i paketet `javax.naming.spi`, något vi inte behöver bry oss om som utvecklare av EJB. ☺)

I exempel nedan ges bara exempel på hur JNDI kan användas – JNDI kan användas till mer än en vad exempel nedan visar.

#### 3.1 Användningsområden för JNDI i J2EE

JNDI kan användas för att ”slå upp” adresser till distribuerade objekt (RMI/CORBA/EJB). Genom att använda JNDI för att lagra adresserna så slipper man ändra klientapplikationer om ett distribuerat objekt skulle flyttas.

Detta ska jämföras Microsofts DCOM (Distributed COM) där klienter själva måste känna till var de distribuerade objekten finns. (Jag ska erkänna att adresser till objekten finns i datorns register, men om ett distribuerat objekt flyttas så måste alla klienter konfigureras om för den

<sup>10</sup> DNS:en är lite unik då den även kan översätta IP-adresser till webbadresser (kallat *reverse lookup*).

<sup>11</sup> Se <http://java.sun.com/products/jndi/serviceproviders.html> för mer information.

nya adressen. Med en namntjänst, som i Java, så behöver klienter endast konfigureras om ifall namntjänsten flyttar.)

## 3.2 Exempel på några namntjänster inom J2EE

RMI (Remote Method Invocation), CORBA (Common Object Request Broker Architecture) och EJB (Enterprise JavaBeans) använder "register" (*registry* – namntjänster) för att lagra referenser till distribuerade objekt. RMI och CORBA använder sitt eget format på namntjänsterna medan EJB "bygger på" JNDI (EJB utvecklades efter JNDI). D.v.s. tillverkare av J2EE-server skapar sin egen (eller använder en redan existerande) namntjänst och förser systemutvecklare med drivrutin till namntjänsten för JNDI (som Resins Burlap- och Hessianservrar<sup>12</sup>).

## 3.3 Namntjänster i JNDI

Ett objekt **binds** till ett namn (*name binding*) och placeras i ett **namnutrymme** (*namespace*), även kallat **kontext**. Alla namn i ett namnutrymme måste vara unika. Namn kan vara **sammansatta** av flera delar inom samma namnutrymme (*compound*) eller av flera delar från olika namnutrymmen (*composite*). Delarnas namn skapas efter konventioner för respektive namnutrymme. Ett namn kan även vara bundet till en **referens** (eller en adress) som pekar till det faktiska objektet.

Gränssnitten för namntjänster finns i paketet `javax.naming`, d.v.s. det paket som behöver importeras för att vi ska kunna använda namntjänster i JNDI.

### 3.3.1 Paketet Naming

För att använda JNDI måste vi först skapa ett initialt kontext, vilket är utgångspunkten för resterande uppgifter. Vi skapar en instans av klassen `InitialContext`, som implementerar gränssnittet `Context`, båda i paketet `javax.naming`. Som parameter till klassens konstruktor skickar vi en vektor (av typen `java.util.Hashtable` eller någon av dess subclasser) med egenskaper för aktuellt kontext (t.ex. *service provider* och adress/URL till namntjänst).

Egenskaperna för aktuellt kontext är beroende av vilken *service provider* som används, d.v.s. hur namntjänsten är uppbyggd. Nedan följer två exempel som visar hur man använder filsystem och LDAP.

Om vi vill använda filsystemet så skriver vi följande kod för att ange egenskaper för kontext:

```
Hashtable env = new Hashtable(); //Skapa vektor för egenskaper

//Ange vilken service provider som ska användas - filsystem här
env.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.fscontext.RefFSContextFactory");

//Ange sökväg som ska vara rot
env.put(Context.PROVIDER_URL, "file:C:/student");
```

Vill vi istället använda LDAP så skriver vi följande kod

```
Hashtable env = new Hashtable(); //Skapa vektor för egenskaper

//Ange vilken service provider som ska användas - LDAP här
env.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.ldap.LdapCtxFactory");
```

<sup>12</sup> Om vi använder servlets som klienter mot EJB i Resin så behöver vi inte använda någon av dessa namntjänster – vi använder lokala gränssnitt och går direkt på EJB. Mer om detta i kapitel om EJB.

```
//Ange sökväg som ska vara rot
env.put(Context.PROVIDER_URL, "ldap://server.doman.se:389");
```

Där efter skapar vi en instans av klassen `InitialContext` och tilldelar till en variabel av typen `Context` (d.v.s. av gränssnittets typ):

```
try {
    Context initCtx = new InitialContext(env);
    ...
}
catch(...) { ... }
```

## Metoder lista och hämta referenser från gränssnittet `Context`

Som parameter till nedanstående metoder kan en sträng eller ett objekt av typen `Name` skickas.

- `list()` – returnerar en vektor av typen `NamingEnumeration` innehållande objekt av typen `NameClassPair` (bundet namn för objekt och objektets klassnamn).
- `listBindings()` – returnerar en vektor av typen `NamingEnumeration` innehållande objekt av typen `Bindings` (bundet namn för objekt och objektet själv).
- `lookup()` – returnerar objektet bundet till skickat namn.

## Andra metoder av intresse i gränssnittet `Context`

- `bind()` – binder namnet i första parametern till objektet i andra parametern.
- `rebind()` – binder namnet i första parametern till objektet i andra parametern. Om namnet redan används så ersätts det gamla objektet med det som skickas i andra parametern.

## Exempel som visar hur åtkomst av filsystem kan ske med JNDI

För att detta exempel ska fungera krävs filerna `fscontext.jar` och `providertil.jar`. Dessa filer kan laddas ner från Suns hemsida, <http://java.sun.com/jndi/>. För att exempel ska fungera krävs även en mapp `student` i roten på hårddisk C (eller att sökvägen för egenskapen `PROVIDER_URL` ändras).

```
import java.util.Hashtable;
import javax.naming.*; //Importerera klasser och gränssnitt för JNDI

class ListFiles {

    public static void main(String[] args) {
        //Skapa vektor att placera egenskaper för namntjänst att använda
        Hashtable env = new Hashtable(); //Skulle kunna vara Properties

        //Ange vilken service provider som ska användas - filsystem här
        env.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.fscontext.RefFSContextFactory");

        //Ange sökväg som ska vara rot
        env.put(Context.PROVIDER_URL, "file:C:/student");

        try {
            //Temporär variabel för objekt i mapp
            Object obj = null;

            //Hämta kontext för vår rot
```

```

Context initCtx = new InitialContext(env);

//Hämta en lista med objekt i kontext
NamingEnumeration objList = initCtx.list("/");

System.out.println(
    "*****");
System.out.println("*** Filer och mappar:");
System.out.println(
    "*****");

//Loopa och visa filer och mappar
while(objList.hasMore()) {
    //Hämta nästa fil eller mapp
    obj = objList.next();

    //Skriv ut fil/mapps namn och klassnamn (File eller Context)
    System.out.println(obj);
}

System.out.println("");
}
catch(Exception e) {
    e.printStackTrace(System.out);
}
} //main()
} //class ListFiles

```

## 3.4 Katalogtjänster i JNDI

Gränssnitten för katalogtjänster fungerar på liknande sätt som de för namntjänsterna. Skillnaden ligger i att data i katalogtjänster kan vara flera värden för ett namn, d.v.s. att objekt lagras.

En katalogtjänst har en rot som är början på katalogtjänsten, d.v.s. första noden (katalogen) som innehåller andra noder<sup>13</sup> och löv<sup>14</sup>. Men varje nod, som kan innehålla löv och andra noder, kan även agera rot för underliggande noder och löv. Aktuell ”rot” kallas för **initialt kontext**.

Gränssnitten för namntjänster finns i paketet `javax.naming.directory`, men vi behöver importera även paketet för namntjänster för att katalogtjänster ska fungera.

### 3.4.1 Paketet Directory

Paketet `Directory` (`javax.naming.directory`) utökar paketet `Naming` (`javax.naming`) genom att ge tillgång till även katalogtjänster. Koden vi behöver för att använda katalogtjänster påminner om den för namntjänster med skilljer sig på ett par punkter. Bl.a. så använder vi klassen `InitialDirContext` som implementerar gränssnittet `DirContext` (istället för `InitialContext` respektive `Context`). Vi har även möjlighet att söka ett objekt – sökningen kan ske efter andra egenskaper än namnet som objektet/-en har bundits till.

En annan skillnad är att vi har fått fler metoder för att söka efter objekt. Dessa metoder tar olika parametrar som kan användas för att avgränsa antalet objekt m.h.a. sökkriterier samt för att avgöra vilka värden som ska returneras i objekt.

En viktig detalj att komma ihåg är att olika katalogtjänster har olika typer av värden som lagras. T.ex. så innehåller NDS ett värde med sökvägen till användarnas hemkataloger medan ett telefonregister saknar detta värde.

<sup>13</sup> En nod är en struktur som kan innehålla andra objekt.

<sup>14</sup> Ett löv är en struktur som endast kan innehålla ett värde, d.v.s. inte en annan nod.

## Exempel som visar hur man listar objekt i LDAP med JNDI

M.h.a. av ett objekt (innehållande en vektor) av typen BasicAttributes anges vilka värden vi vill ska returneras från katalogtjänsten.

Detta exempel kräver en LDAP-server och att URL till LDAP-server ändras i kod nedan.

```
import java.util.Hashtable;
import javax.naming.*;           //Importerera paket för namntjänster
import javax.naming.directory.*; //Importerera paket för katalogtjänster

public class ListLDAP {

    public static void main(String[] args) {
        Hashtable env = new Hashtable();

        env.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.ldap.LdapCtxFactory");

        env.put(Context.PROVIDER_URL,           //URL till LDAP-server samt kontext
            "ldap://saruman.eki.mdh.se/o=MDH"); //Ändra URL till egen LDAP-server

        try {
            DirContext dctx = new InitialDirContext(env);

            Attributes attrs = new BasicAttributes(true);
            attrs.put(new BasicAttribute("cn"));

            String[] returnAttrs = {"cn"};

            NamingEnumeration objList = dctx.search("ou=AVD", null, returnAttrs);

            while(objList.hasMore()) {
                System.out.print("Objekt: ");
                System.out.println(objList.next());
            }
        }
        catch(Exception e) {
            e.printStackTrace(System.out);
        }
    } //main()
} //class ListLDAP
```

## 3.5 J2EE vs. COM/COM+

Microsofts version av JNDI är egentligen två produkter: Active Directory Services Interface (ADSI) för åtkomst av katalogtjänster och "namntjänsten" Service Control Manager (SCM). SCM:en är en del av COM/COM+ och fungerar som en ORB. Trots namnet på ADSI så fungerar ADSI även med andra katalogtjänster än Microsofts egna Active Directory (AD) i Windows 2000/XP, bl.a. Windows NT, Novells bindery (versioner tidigare än NetWare 4) och Novell Directory Services (NDS, numera kallat eDirectory) samt LDAP.

## 4 Servlets

*Servlets* är en utveckling av *server-side*-exekvering som SSI (*Server Side Includes*), CGI (*Common Gateway Interface*), *server-side scripting* (så som Active Server Pages, ASP) samt insticksmoduler så som ISAPI (*Internet Server API* för IIS) och NSAPI (*Netscape Server API*).

Namnet *servlet* anspelar på *applets* (små programmoduler som exekverar i webbläsare) och exekverar på en server istället.

---

### 4.1 HyperText Transport Protocol (HTTP)

HTTP är ett protokoll för kommunikation mellan webbläsare och webbserver. Protokollet är tillståndslöst, d.v.s. bygger på begäran och svar – klienten begär en resurs (HTML-dokument, servlet, m.m.) från webbserver och webbserver svarar med att skicka begärd resurs. Med tillståndslöst menas att klient etablerar förbindelse med webbserver för att skicka begäran och webbserver etablerar en ny förbindelse med klient för att skicka svaret. När svaret har skickats så kvarstår ingen förbindelse mellan klient och server.

För varje extern resurs (så som bilder) i ett HTML-dokument så måste klienten göra en ny begäran från webbserver. D.v.s. en webbsida med många bilder innebär många olika begäran från klient till webbserver. (Detta problem har till viss del löst i version 1.1 av HTTP.)

---

### 4.2 Historiken bakom servlets

De flesta webbserverar stödjer idag SSI – ett sätt att infoga dynamisk information som inkludera filer, visa datum, m.m.. SSI:s användningsområde är dock mycket begränsat – det går inte att utveckla applikationer med SSI, endast infoga viss dynamisk information i webbsidor. För att erhålla mer exekveringsmöjligheter utvecklades därför CGI – ett gränssnitt för kommunikation mellan webbserver och ett externt program. Problemet med CGI att varje begäran av ett CGI-program startar en ny process som är skild från webbserverns process. Att skapa processer är kostsamt. Detta har delvis lösts med **FastCGI** som använder en pool av processer, d.v.s. processer som kan återanvändas.

Nästa steg var att utveckla ett gränssnitt för **insticksmoduler** (SAPI, Server API) som utökar webbserverns funktionalitet: ISAPI för Microsofts IIS, NSAPI för Netscapes webbserver och mod-filer för Apache. Denna funktionalitet exekverar i samma process som webbservern (som DLL:er i Windows-miljö), d.v.s. är mer effektiva än CGI. Nackdelen med insticksmodulerna är att om det uppstår ett fel vid exekvering så kan detta haverera webbserverns process, d.v.s. webbservern slutat att svara på begäran av webbsidor (webbserver kan krascha). En annan nackdel är att dessa insticksmoduler ofta (i Windows) måste skivas med C eller C++ – två språk som är svåra att felsöka (göra buggfria).

För att göra exekvering mer tillgänglig på webbserverar utvecklades *server-side scripting* – skriptkod som tolkas och exekveras på webbserver men som inte kräver förändringar av webbserverns mjukvara. Typiska exempel är ASP från Microsoft och den öppna standarden PHP. Nackdelen med ASP och PHP är att koden måste tolkas och kompileras varje gång som ASP eller PHP-sida begärs, d.v.s. inte vidare effektivt men gör tekniken tillgänglig för fler än insticksmoduler (vilka webbserveransvariga inte gärna installerar om det inte skrivit dem själv eller att de är välbeprövade).

*Servlets* är små ”programsnuttar” skrivna i ren Java, är kompilerad kod och laddas första gången som URL för servlet begärs. Från en servlet kan vi anropa objekt i andra J2EE-teknologier, så som JDBC och EJB. Exekveringen av servlets kan ske i samma process som

webbserver eller i en separat process, beroende på tillverkare av webbserver och servlet-motor (*servlet engine*).

## 4.3 Klasser och gränssnitt för servlet

Det finns främst två paket där servlets definieras: `javax.servlet` som innehåller definitioner för generella servlets och `javax.servlet.http` (som utökar förra paketet) för servlets som använder HTTP. Gränssnittet `Servlet` (i `javax.servlet`) definierar de tre intressanta metoderna som alla servlet bör implementera: `init()`, `service()` och `destroy()`. Metoden `init()` anropas då servlet laddas och bör innehålla kod som t.ex. öppnar en databaskoppling. När sen servlet laddas ur, t.ex. om webbserver stängs av, så anropas metoden `destroy()`. I `destroy()` bör finnas kod för att rensa upp och stänga eventuella öppna resurser (så som databaskopplingar). Metoden `service()` anropas av webbservern varje gång som URL för servlet begärs, d.v.s. här placeras logiken som ska exekveras när servlet begärs av en webbläsare.

### 4.3.1 Generiska servlets

Servlets är inte bara en webbföreteelse – servlets kan anropas från ”vanliga” Java-applikationer. Denna typ av servlets ärver lämpligen från klassen `GenericServlet` (`javax.servlet`), som implementerar gränssnittet `Servlet`. Den nya klassen bör omdefiniera minst metoden `service()` – metoden där all exekvering sker för en begäran. Som parametrar till `service()` skickar webbserver objekt av typen `ServletRequest` och `ServletResponse`. `ServletRequest` innehåller information om begäran av servlet, bl.a. eventuella parametrar som skickades till servlet. `ServletResponse` används för att skicka information tillbaka till klienten som skickade begäran.

Innan man skickar något från en servlet bör man ange vad som skickas (vilken MIME-typ).<sup>15</sup> Detta gör man med metoden `setContentType()` i `ServletResponse`-objektet och som parameter till metoden skickas en sträng som innehåller MIME-typen, t.ex. ”text/html”. Nästa steg är att erhålla ett objekt av typen `PrintWriter` (`java.io`) (en ”utström”, *output stream*) för att kunna skriva information som returneras som resultat från exekvering. `PrintWriter`-objektet erhålles genom att anropa metoden `getWriter()` i `ServletResponse`-objektet.

```
import java.io.PrintWriter;
import javax.servlet.*;

public void service(ServletRequest request, ServletResponse response)
{
    response.setContentType("text/html"); //Ange typ av data som returneras
    PrintWriter out = response.getWriter(); //Hämta utström

    out.println("Hello World!"); //Skriv till utström

    out.close(); //Stäng utström
}
```

Därefter använder vi metoderna `print()` eller `println()` i `PrintWriter`-objektet för att skicka information till klienten. Sist av allt bör vi stänga vårt `PrintWriter`-objekt genom att anropa dess metod `close()`.

<sup>15</sup> Om man utelämnar anropet av `setContentType()` så kommer vissa webbläsare (t.ex. Mozilla) visa HTML-koden (d.v.s. källan) istället för själva hemsidan!



Om vi vill hämta eventuella parametrar som skickats till servlet så använder vi metoden `getParameter()`, till vilken vi skickar namnet på parameter som en sträng. Metoden returnerar parametervärdet som en sträng eller `null` om parameter saknas. (Se exempel med HTTP servlets för hur metoden används.)

Eftersom de flesta Java-applikationer idag använder en Java-klient eller webbläsare så diskuteras inte generiska servlets mer i denna sammanfattning.

### 4.3.2 HTTP servlets

Servlets som ska användas av klienter som använder HTTP (d.v.s. webbläsare) ärver lämpligen från klassen `HttpServlet` (`javax.servlet.http`) istället för `GenericServlet`.<sup>16</sup> `HttpServlet` har implementerat metoden `service()` på ett sådant sätt att den anropar någon av metoderna `doGet()` och `doPost()` om servlet begärs med HTTPs GET respektive POST.<sup>17</sup> Därför bör servlets som ärver från klassen `HttpServlet` omdefiniera dessa metoder istället för metoden `service()`. Vill man inte skilja på begäran med GET och POST så kan t.ex. `doPost()` anropa `doGet()`.

Till metoderna `doGet()` och `doPost()` skickas objekt av typen `HttpServletRequest` och `HttpServletResponse` (som motsvarar `ServletRequest` och `ServletResponse` i `service()`).

Ytterligare en skillnaden mot `GenericServlet` är att vi har tillgång till alla värden som skickats med HTTP-begäran (t.ex. besökarens IP-adress).

### 4.3.3 Exempel på en enkel HTTP-servlet

I detta första exempel med HTTP-servlet så gör vi som man ”brukar göra” – skriver ut texten `Hello World!`. ☺

På första raden börjar vi med att ange i vilket paket som vi vill placera vår servlet. Detta (eller dessa) namn på paket måste vi sen använda när vi vill ladda servlet, t.ex.

```
http://localhost:8080/servlet/bpn.EnkelServlet. Sen importerar vi paket för servlets och HTTP-servlets (för bl.a. klassen HttpServlet). Och eftersom vi använder utmatning, d.v.s. använder I/O, så måste vi även importera paketet för I/O (bl.a. för klassen PrintWriter).
```

Sen påbörjar vi vår klass som ärver från `HttpServlet`. Namnet på klassen behöver **inte** ha suffixet `Servlet`, men när vi håller på att lära oss J2EE så underlättar det oftast.<sup>18</sup> I klassen lägger vi sen till metoden `doGet()` med parametrar för begäran och svaret som vi ska skicka tillbaka.<sup>19</sup> Parametern för begäran (*request*), `Request`-objektet, innehåller data om hur klient begärde servlet, om klienten (t.ex. IP-adress) och eventuell data från formulär. Och parametern för svar (*response*), `Response`-objektet, används för att returnera genom en instans typen `PrintWriter` – en ”utström” (*out stream*), d.v.s. en ström av tecken. I metoden `doGet()` kan vi fånga undantag av typen `IOException` eller låta metoden slänga vidare dem (d.v.s. lägga till `throws ...` efter metodsignatur). Eftersom det oftast inte är intressant att fortsätta om

---

<sup>16</sup> `HttpServlet` ärver från klassen `GenericServlet`

<sup>17</sup> Det finns andra sätt att begära webbresurser och därmed fler metoder utöver `doGet()` och `doPost()`.

<sup>18</sup> Samtidigt kan vi skapa alias för servlet så att besökare av webbserver kanske ser ett namn som `Enkel` istället för `EnkelServlet`. D.v.s. använd gärna suffixet `Servlet` för servlets.

<sup>19</sup> Parametrarna brukar ges namnen `request` och `response` (eller ibland förkortade till `req` resp. `res` eller `resp`). Samma namn på motsvarande objekt används i JSP, d.v.s. det gör det lättare att hoppa mellan servlets och JSP.

vi får problem med I/O så rekommenderas det senare alternativet (vilket också visas i koden nedan).

Det första vi bör göra i metoden är att tala om för webbläsare vad som kommer returneras (via "utströmen").<sup>20</sup> Det gör vi genom att anropa metoden `setContentType()` i `Response`-objektet och bifoga typ av data med en sträng – oftast "text/html". Därefter kan vi hämta en referens till "utströmen" genom att anropa metoden `getWriter()` (om text ska returneras, vilket det oftast ska). `getWriter()` returnerar en referens till ett objekt av typen `PrintWriter` och referensen brukar placeras i en variabel med namnet `out`.<sup>21</sup>

Sist av allt så stänger vi "utströmen" genom att anropa metoden `close()` i "utströmen". Mellan anropet av `getWriter()` och `close()` så exekverar vi kod för att exekvera logik för att skapa dynamiskt innehåll i webbsida samt anropar metoden `print()` eller `println()` i "utströmen" (variabeln `out`) för att skriva ut HTML-kod som strängar. I detta exempel så skriver vi ut taggarna `<HTML>` och `<BODY>` samt strängen `Hello World!`. Vill vi göra en mer avancerad webbsida (än detta exempel) så anropar vi alltså `print()/println()` fler gånger och skickar med HTML-kod vi vill att webbsida ska bestå av. ☺

```

package bpn;                //Paket att placera servlet i

import javax.servlet.*;     //Importerera paket för servlet
import javax.servlet.http.*; //Importerera paket för HTTP-servlet
import java.io.*;           //Importerera paket för I/O (PrintWriter/IOException)

public class EnkelServlet extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException
    {
        response.setContentType("text/html"); //Ange typ av data som returneras
        PrintWriter out = response.getWriter(); //Hämta utström (out stream)
        out.print("<html><body>Hello world!</body></html>"); //Skriv till utström
        out.close(); //Stäng utström
    } //doGet()
} //class EnkelServlet

```

Om vår servlet även ska ta emot data från formulär som skickas med HTTP-metode POST så måste vi även lägga till metoden `doPost()` (se *Exempel med formulär* för mer).

#### 4.3.4 Kompilera och installera servlet

För att kunna kompilera servlet så måste vi ha tillgång till filen `J2EE.JAR`, d.v.s. den måste finnas i *class path* (den sökväg som Java letar efter klasser). Filen finns, om J2EE version 1.4, i mappen `C:\Program\Sun\AppServer\lib\` (om J2EE installerades under mappen `Program` och inte i roten). (Om JCreator används så kan man lägga till ett "required library" och där ange sökväg till JAR-filen – gärna även dokumentationen i mappen `docs`. Varje projekt därefter kan då bocka för detta "required library" då det behövs. Alternativt används en miljövariabel `%CLASS_PATH%` för att ange sökväg till JAR-fil.)

När vi vill installera servlet så brukar det ske genom att vi kopierar CLASS-filen till en speciell mapp på webserver. Om vi t.ex. använder Resin så ska filen placeras i en

<sup>20</sup> Utelämnar vi detta steg kommer vissa webbläsare (t.ex. Mozilla) visa HTML-koden istället för hemsidan!

<sup>21</sup> Återigen så är det praktiskt att namnge variabeln enligt standard, d.v.s. `out`, då ett objekt med samma namn finns i JSP. Samtidigt så påminner det om `System.out` som vi använder i konsolapplikationer. ☺

webbapplikations mapp `WEB-INF\classes`. Observera att om servlet finns i ett paket så måste CLASS-filen placeras i en mapp med samma struktur som paketet. Med Resin och ovanstående exempel så ska filen alltså placeras i mappen `WEB-INF\classes\bpn`. (Hade paketet varit `bpn.servlets.gurka` så skulle filen placeras i `WEB-INF\classes\bpn\servlets\gurka`.)

När vi ska testa servlets så brukar de flesta ”servlet-motorer” (*servlet engines*) var konfigurerade så att man laddar den med `/servlet/` före i URL. Om servlet t.ex. installeras på samma dator som du sitter så öppnar vi servlet genom att skriva `http://localhost:8080/bpn/servlet/bpn.EnkelServlet`.

---

## 4.4 Funktioner i servlets

Nedan beskrivs några av de mest användbara ”funktionerna” i servlets, d.v.s. saker som används relativt ofta.

### 4.4.1 Inkludera webbresurser

Inkludering i webbgränssnitt innebär att man infogar externa resurser i en annan resurs. Detta görs för att bl.a. kunna dela på gemensam kod (d.v.s. slippa upprepning och endast behöva uppdatera på ett ställe), t.ex. HTML-filer med sidhuvud och sidfötter.

#### Inkludera (statiska) HTML-filer

I detta exempel inkluderas två statiska HTML-filer med sidhuvud och sidfot för en webbsida. Följande kod finns i filen `sidhuvud.html` (som ger vit text mot blå bakgrund)

```
<p style="background-color: blue; color=white">Text i sidhuvud</p>
```

och följande kod i filen `sidfot.html` (som ger ”standardfärg” på text mot gul bakgrund).

```
<p style="background-color: yellow">Text i sidfot</p>
```

Filerna ovan inkluderas i nedanstående servlet. Eftersom alla klienter kommer använda samma filer (som inkluderas) så deklarerar strängar med URL:er (till inkluderade filer) som konstanter i klassen.<sup>22</sup> När resurser inkluderas kan metoden slänga undantaget `ServletException` – något vi kan välja att hantera i metoden `doGet()` eller bara slänga vidare, d.v.s. metod måste då slänga det undantaget också (utöver `IOException`).

För att inkludera en resurs så måste vi hämta en instans av klassen `RequestDispatcher` från `Request`-objektet. Detta görs med metoden `getRequestDispatcher()` som vi bifogar URL till resurs att inkludera. Eftersom detta exempel ska inkludera två HTML-filer så behöver vi alltså två instanser av `RequestDispatcher`. Sen anropas metoden `include()` i respektive `RequestDispatcher` där vi vill att webbresurs ska inkluderas i servlet.

```
package bpn;
```

---

<sup>22</sup> På så sätt kommer det endast finnas en instans av respektive sträng (oavsett antal klienter) och vi sparar minne. ☺ Prefixet `c` i konstanter namn ska visa att det är konstanter (*constant*).

```

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class IncludeServlet extends HttpServlet {
    //Konstanter för URL:er till webbresurser (HTML-filer i detta exempel)
    private static String cstrUrlHuvud = "/sidhuvud.html"; //Sökväg till filer
    private static String cstrUrlFot = "/sidfot.html"; // relativt webbapps rot

    //Slänger ServletException då include() kan slänga detta...
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
        response.setContentType("text/html"); //Ange typ av data som returneras
        PrintWriter out = response.getWriter(); //Hämta utström (out stream)

        //Hämta en RequestDispatcher för respektive webbresurs att inkludera
        RequestDispatcher rdHuvud = request.getRequestDispatcher(cstrUrlHuvud);
        RequestDispatcher rdFot = request.getRequestDispatcher(cstrUrlFot);

        out.println("<html><body>"); //Skriv till utström
        rdHuvud.include(request, response); //Inkludera webbresurs (1)
        out.println("<p>Denna text finns i servlet som inkluderar webbresurser</p>");
        rdFot.include(request, response); //Inkludera webbresurs (2)
        out.println("</body></html>");
        out.close(); //Stäng utström
    } //doGet()
} //class IncludeServlet

```

## Mer om inkludering i servlets

Det går även att inkludera andra servlets (samt JSP och andra dynamiska resurser). Dessa inkluderade servlets bör då inte anropa metoder som `setContentType()` eller skriva ut taggar som `<HTML>` och `<BODY>`.

För att bifoga data vid inkludering av en resurs så kan vi använda metoden `setAttribute()` i Request-objektet. Nedan bifogas värdet "Björn" med attributnamnet "webmaster".

```

request.setAttribute("webmaster", "Björn"); //Bifoga data till inkluderad resurs
//...
rdFot.include(request, response);

```

## 4.4.2 Servlets och formulär

HTML-formulär (eller bara formulär) används som sättet att kommunicera med besökare av webbplats, d.v.s. ett sätt att efterfråga indata från besökaren. (Utan formulär så blir en webbplats en envägskommunikation från skaparen av webbplats till besökaren.)

Nedan visas (än så länge) ett exempel med HTML-fil med formulär och en servlet som "behandlar" data i formulärets fält.

### HTML-fil och servlet

I detta exempel visas ett formulär (i en HTML-fil) där besökare fyller i sitt namn och födelseår, vilket skickas till servlet. Servlet läser namn och födelseår samt visar namn och beräknar hur många år besökare fyller i år. För att kunna beräkna åldern på besökaren behövs klassen `Calendar` för att ta reda på vilket år det är (aktuellt år).

### HTML-fil med formulär

HTML-formuläret (eller bara formuläret) är ganska rakt på sak: två textrutor för namn (`namn`) och födelseår (`fodelsear`) samt en skickaknapp. I formulärets action-attribut anges URL till servlet och eftersom method-attribut utelämnats så kommer formulär skickas med HTTP GET.

```

<html>
<body>
<p>Fyll i formulär och klicka på Skicka-knappen.</p>
<form action="servlet/bpn.Formular1Servlet">
  <p>Namn: <input type="text" name="namn"></p>
  <p>Födelseår: <input type="text" name="fodelsear"></p>
  <p><input type="submit" value="Sicka"></p>
</form>
</body>
</html>

```

## Servlet som ”behandlar” data från formulär

Eftersom formuläret skickas med HTTP GET så måste servlet implementera metoden `doGet()`. Om formuläret istället skickats med HTTP POST så skulle vi bara behövt implementera metoden `doPost()`.

I `doGet()` deklarerar vi variabler för att hämta data från fälten i formulär som strängar – data i formulär returneras endast som strängar som måste konverteras till t.ex. tal om det är det som önskas (vilket vi vill med besökarens födelseår i detta exempel). Eftersom vi vill jobba med tal i exemplet så deklarerar vi även variabler för tal.

För att läsa data från fält i ett formulär så används metoden `getParameter()` i Request-objektet. (Observera att Java skiljer på gemener och versaler i namnen på fälten!) Denna metod returnerar, som sagt, innehållet i fältet som strängar. Eftersom vi har ett födelseår så måste vi konvertera strängen i fältet `fodelsear` till ett heltal, vilket görs med metoden `parseInt()` i klassen `Integer`. Sen används klassen `Calendar` för att hämta aktuellt år, som sen används genom att subtrahera år som besökare är född för att erhålla ålder på besökare.

```

package bpn;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.Calendar;

public class Formular1Servlet extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException {
        String strNamn = null, strAr = null;
        int intAr = 0, intAktuelltAr = 0;

        response.setContentType("text/html"); //Ange typ av data som returneras
        PrintWriter out = response.getWriter(); //Hämta utström (out stream)

        strNamn = request.getParameter("namn"); //Hämta data från formulär
        strAr = request.getParameter("fodelsear");

        //Konvertera sträng till heltal
        try {
            intAr = Integer.parseInt(strAr);
        }
        catch(NumberFormatException nfe) {
            out.println("<pre>");
            nfe.printStackTrace(out);
            out.println("</pre>");
        }

        //Hämta aktuellt år
        intAktuelltAr = Calendar.getInstance().get(Calendar.YEAR);

        out.print("<html><body><p>Ditt namn är "); //Skriv till utström
        out.print(strNamn);
        out.print(" och du är ");
        out.print(intAktuelltAr - intAr); //Beräkna ålder och skriv ut
        out.print(" år gammal.</p>");
        out.print("</body></html>");
    }
}

```

```

        out.close(); //Stäng utström
    } //doGet()
} //class Formular1Servlet

```

## Enbart servlet

Att enbart använda servlets (d.v.s. utan HTML-fil) är fullt möjligt (men kanske inte det mest gångbara alternativet ☺). Principen för en servlet av detta slag är att om ingen data skickats via formulär så visas HTML-formuläret. Men när data skickats så behandlas data i formulär. Exempel kanske kommer...

### 4.4.3 Sessionshantering

Varje klient som begär en resurs på webbserver motsvaras av en session. På detta sätt kan en webbserver hålla reda på vilka klienterna är (och vilka klienter som är ”anslutna”<sup>23</sup>) samt lagra data relaterad till klienterna. Sessionsdata görs tillgänglig via ett Session-objekt i servlets. För att erhålla en referens till klientens Session-objekt så används metoden `getSession()` i Request-objektet. En session brukar som standard vara 20 minuter efter att klient begärde sista resursen från webbserver.<sup>24</sup> En session hanteras oftast genom webbserver skickar en *cookie* till klient – en *cookie* som sen klient bifogar med alla efterföljande begäran.<sup>25</sup>

Sessionsdata, eller ”sessionsvariabler”<sup>26</sup>, skrivs och läses med metoderna `setAttribute()` respektive `getAttribute()` i klientens Session-objekt. Den första metoden har två parametrar – den första en sträng för namnet på sessionsvariabeln och den andra av typen `Object` för själva värdet som ska lagras. Medan den andra metoden endast har en parameter – en sträng för namnet på variabeln – men som returnerar ett värde av typen `Object`. Om variabel inte satts så returnerar `getAttribute()` värdet `null`.

Ett användningsområde för sessionsdata är t.ex. i en webbutik där klienter handlar artiklar eller för att hålla reda på data om inloggad besökare.

### Enkelt exempel med räknare i sessionsvariabel

Här används en sessionsvariabel för att hålla reda på hur många webbsidor som besökare besökt under aktuell session. För att hålla reda på antalet så måste vi använda servlets (eller JSP) för varje webbsida.

#### Servlet för webbsida 1

I `doGet()` deklaras en lokal variabel för att hålla aktuellt räknarvärde (som bl.a. ökas på i servlet). Här deklaras även en variabel för att hålla reda på Session-objektet, genom vilket vi kan lagra sessionsvariabler. Sessionsvariabler lagras som klassen `Object`, d.v.s. vi behöver en variabel av denna typ också för att kunna hämta värden i sessionsvariabler. För att hämta en sessionsvariabel så använder vi metoden `getAttribute()` i Session-objektet – en metod som returnerar `null` om ingen sessionsvariabel med det namnet existerar. Som parameter till

<sup>23</sup> Jag har satt ”anslutna” inom citattecken då HTTP är ett tillståndslöst protokoll. Det är denna nackdel med HTTP som sessioner försöker lösa.

<sup>24</sup> Detta ger möjlighet för besökare att tänka lite, t.ex. vid ifyllande av formulär, eller springa och hämta kaffe. ☺

<sup>25</sup> Observera att om klienten inte stödjer cookies så fungerar inte sessionshantering som det ska. Det kan visserligen lösas med *URL rewriting* – cookie skickas som del av URL.

<sup>26</sup> Jag kallar det för ”sessionsvariabler” eftersom det är en form av namngivna variabler som gäller för klientens session.

metoden skickas namnet på sessionsvariabeln som en sträng. Observera att Java skiljer på gemener och versaler i namn på sessionsvariabler!

Om värde på sessionsvariabel (räknare i exempel) har satts (d.v.s. inte är null) så kan det konverteras från Object till *wrapper*-klassen Integer, som vi sen kan fråga efter heltalsvärdet. Heltalsvärdet, d.v.s. räknaren, ökas sen på för sen skrivs tillbaka till sessionsvariabel med metoden setAttribute(). Parametrar i metoden setAttribute() är en sträng för namnet på sessionsvariabel och själva värdet, av typen Object, på sessionsvariabeln. Eftersom räknaren är ett heltal, d.v.s. av typen int, så måste vi använda *wrapper*-klassen Integer för att lagra värdet.

```

package bpn;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class Raknare1Servlet extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException {
        int intRaknare = 0;

        response.setContentType("text/html");    //Ange typ av data som returneras
        PrintWriter out = response.getWriter(); //Hämta utström

        //Hämta session för besökare - skapa om inte existerar
        HttpSession session = request.getSession(true);

        //Hämta ev räknarvärde från sessionsvariabel
        Object objRaknare = session.getAttribute("raknare");

        //Om värde finns - konvertera till int ...
        if(objRaknare != null)
            intRaknare = ((Integer)objRaknare).intValue();

        intRaknare++;                                //Öka på räknarvärde

        //Skriv räknarvärde till sessionsvariabel
        session.setAttribute("raknare", new Integer(intRaknare));

        out.print("<html><body><h1>Sida 1</h1><p>Du har besökt ");
        out.print(intRaknare);
        out.print(" webbsidor hos oss.</p>");
        out.print("<p><a href=\"bpn.Raknare2Servlet\">Sida 2</a></p>");
        out.print("</body></html>");

        out.close();                                //Stäng utström
    } //doGet()
} //class Raknare1Servlet

```

## Servlet för webbsida 2

Koden i denna servlet är i stort sett identisk med den ovan – skillnaden ligger i att alla referenser till ”Sida 1” har ändrats till ”Sida 2” och tvärt om. Se förklaring till kod ovan.

```

package bpn;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class Raknare2Servlet extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException {
        int intRaknare = 0;

```

```

response.setContentType("text/html"); //Ange typ av data som returneras
PrintWriter out = response.getWriter(); //Hämta utström

//Hämta session för besökare - skapa om inte existerar
HttpSession session = request.getSession(true);

//Hämta ev räknarvärde från sessionsvariabel
Object objRaknare = session.getAttribute("raknare");

//Om värde finns - konvertera till int...
if(objRaknare != null)
    intRaknare = ((Integer)objRaknare).intValue();

intRaknare++; //Öka på räknarvärde

//Skriv räknarvärde till sessionsvariabel
session.setAttribute("raknare", new Integer(intRaknare));

out.print("<html><body><h1>Sida 2</h1><p>Du har besökt ");
out.print(intRaknare);
out.print(" webbsidor hos oss.</p>");
out.print("<p><a href=\"bpn.RaknareServlet\">Sida 1</a></p>");
out.print("</body></html>");

out.close(); //Stäng utström
} //doGet()
} //class Raknare2Servlet

```

## Använda sessionsvariabler för att hantera inloggning

I detta exempel används en HTML-fil med formulär för användaridentitet och lösenord samt servlets för validering av data, privat webbsida och för utloggning (d.v.s. en HTML-fil och 3 servlets).

### HTML-fil med formulär

HTML-formuläret innehåller två fält – ett för användaridentitet och ett för lösenord. Innehållet skickas till servlet nedan (VerifieraServlet – fulla namnet på klass används eftersom servlet är i ett paket). Attributet method har utelämnats i form-taggen, vilket gör att formulär kommer skickast med metoden GET.

```

<html>
<body>
<p>Fyll i användaridentitet och lösenord samt klicka på knappen Logga in för
att verifiera din identitet.</p>
<form action="servlet/bpn.VerifieraServlet">
<p>Användaridentitet: <input type="text" name="anvId" value="abc00001"></p>
<p>Lösenord: <input type="password" name="losenord" value="gurka"></p>
<p><input type="submit" value="Logga in"></p>
</form>
<p><a href="servlet/bpn.PrivatServlet">Privat sida</a> (som kräver
inloggning).</p>
</body>
</html>

```

### Servlet för att verifiera användaridentitet och lösenord

Denna servlet läser data från formulär (d.v.s. användaridentitet och lösenord) samt verifierar att användaridentitet och lösenord är rätt. Här görs kontroller att fälten fanns i formulär (d.v.s. att `getParameter()` returnerar annat än `null`) och att fälten innehåller något. Sen kontrolleras att användaridentitet och lösenord är riktiga – om så är fallet så visas en länk till privat webbsida, annars en länk tillbaka (m.h.a. JavaScript).

I ett verkligt exempel så skulle användaridentitet och lösenord t.ex. kunna läsas från en databas.

```
package bpn;
```



```

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class VerifieraServlet extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException {
        String strAnvId = null, strLosenord = null;

        //Hämta data från formulär
        strAnvId = request.getParameter("anvId");
        strLosenord = request.getParameter("loosenord");

        response.setContentType("text/html"); //Ange typ av data som returneras
        PrintWriter out = response.getWriter(); //Hämta utström

        //Hämta session för besökare - skapa om inte existerar
        HttpSession session = request.getSession(true);

        out.print("<html><body>"); //Påbörja HTML-dokument

        //Om fält fanns i HTML-formulär
        if((strAnvId != null) && (strLosenord != null))
        { //Om fält i formulär innehöll något
            if((strAnvId.length() > 0) && (strLosenord.length() > 0))
            { //Om anvId och lösenord är "riktiga" - skriv anvId till sessionsvariabel
                if(strAnvId.equals("abc00001") && strLosenord.equals("gurka")) {
                    session.setAttribute("anvId", strAnvId); //Skriv sessionsvariabel
                    out.print("<h1>Gratulerar!</h1>");
                    out.print("<p>Din identitet har verifierats " + strAnvId + "!</p>");
                    out.print("<p><a href=\"bpn.PrivatServlet\">Fortsätt</a> till privat "
                        + "hemsida.</p>");
                }
            }
            else
                out.print("<p>Användaridentitet och/eller lösenord är felaktigt. "
                    + "Försök igen!</p>");
        }
        else {
            out.print("<h1>Fel!</h1>");
            out.print("<p>Användaridentitet och/eller lösenord har <b>inte</b> "
                + "skickats!</p>");
            out.print("<p><a href=\"javascript:history.back()\">Backa</a> och försök "
                + "igen.</p>");
        }
    }
    else
        out.print("<p>Formulär är felaktigt. Kontakta administratör och meddela "
            + "detta!</p>");

    out.print("</body></html>"); //Avsluta HTML-dokument

    out.close(); //Stäng utström
} //doGet()
} //class VerifieraServlet

```

## Servlet med privat information

En ”privat” webbsida är en webbsida som kräver att besökare är inloggad. I detta exempel visas olika saker beroende på om besökare är inloggad eller inte. En annan lösning är att använda *redirect* eller *forward* (se avsnitt nedan) för att förflytta icke inloggad besökare till ett inloggningsformulär.

```

package bpn;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class PrivatServlet extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException {
        Object objAnvId = null;

```

```

String strAnvId = null;

response.setContentType("text/html");    //Ange typ av data som returneras
PrintWriter out = response.getWriter(); //Hämta utström

    //Hämta session för besökare - skapa om inte existerar
    HttpSession session = request.getSession(true);

out.print("<html><body>"); //Påbörja HTML-dokument

out.print("<h1>Privat sida</h1>");
out.print("<p>För att kunna se innehållet i denna webbsida måste man vara "
    + "inloggad.</p>");

objAnvId = session.getAttribute("anvId"); //Hämta värde på sessionsvariabel

    //Om sessionsvariabel finns - konvertera till sträng ...
    if(objAnvId != null) {
        strAnvId = (String)objAnvId;
        out.print("<p>Du är inloggad som " + strAnvId + ".</p>");
        out.print("<p><a href=\"bpn.UtloggningServlet\">Logga ut</a></p>");
    }
    else { //... annars meddela besökare att ej inloggad
        out.print("<p>Du är <b>inte</b> inloggad! ");
        out.print("<a href=\"../session_inloggning.html\">Logga ");
        out.print("in</a> för att se innehållet.</p>");
    }

out.print("</body></html>"); //Avsluta HTML-dokument

out.close(); //Stäng utström
} //doGet()
} //class PrivatServlet

```

## Servlet för utloggning

För att logga ut så avslutas session för besökare. Detta görs genom att hämta en referens till Session-objekt och anropa dess metod invalidate().

```

package bpn;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class UtloggningServlet extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException {
        response.setContentType("text/html");    //Ange typ av data som returneras
        PrintWriter out = response.getWriter(); //Hämta utström

        //Hämta session för besökare - skapa INTE
        HttpSession session = request.getSession(false);
        session.invalidate(); //Avsluta session

        out.print("<html><body>"); //Påbörja HTML-dokument
        out.print("<p>Du är nu utloggad.</p>");
        out.print("<p><a href=\"bpn.PrivatServlet\">Privat sida</a></p>");
        out.print("</body></html>"); //Avsluta HTML-dokument

        out.close(); //Stäng utström
    } //doGet()
} //class UtloggningServlet

```

## 4.4.4 Använda JavaBeans

Att använda JavaBeans i servlets innebär användande av sessionsvariabler. (Exempel finns med här för jämförelse med nästa kapitel om JSP.) I detta exempel används två servlets – en för att skapa instans av JavaBeans och skriva till sessionsvariabel samt en andra som läser sessionsvariabel och läser värden på JavaBeans attribut.

## JavaBean-klass

Denna enkla klass innehåller två attribut `anvId` och `namn` (och motsvarande accessmetoder).

```
package bpn;

public class EnkelJavaBean {
    /** Instansvariabler *****/
    private String anvId;
    private String namn;

    /** Accessmetoder för klassens attribut *****/
    public void setAnvId(String id) {
        anvId = id;
    }
    public String getAnvId() {
        return anvId;
    }

    public void setNamn(String n) {
        namn = n;
    }
    public String getNamn() {
        return namn;
    }
} //EnkelJavaBean
```

## Servlet 1

Denna servlet skapar en instans av JavaBean (`EnkelJavaBean`), sätter värden på dess attribut samt skriver den till en sessionsvariabel (`servletBona`). För att förenkla koden importeras även klassen för JavaBean.

```
package bpn;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import bpn.EnkelJavaBean; //Importera JavaBean

public class JavaBeans1Servlet extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException {
        EnkelJavaBean bona = new EnkelJavaBean(); //Skapa böna
        bona.setAnvId("abc00001"); //Sätt värden på attribut
        bona.setNamn("Anders Boviac");

        response.setContentType("text/html"); //Ange typ av data som returneras
        PrintWriter out = response.getWriter(); //Hämta utström

        //Hämta session för besökare - skapa om inte existerar
        HttpSession session = request.getSession(true);
        session.setAttribute("servletBona", bona); //Skriv bona till sessionsvariabel

        out.print("<html><body>"); //Påbörja HTML-dokument
        out.print("<h1>Servlet 1</h1>");
        out.print("<p>Denna servlet fyller JavaBean med data.</p>");
        out.print("<p><a href=\"bpn.JavaBeans2Servlet\">Servlet 2</a></p>");
        out.print("</body></html>"); //Avsluta HTML-dokument

        out.close(); //Stäng utström
    } //doGet()
} //class JavaBeans1Servlet
```

## Servlet 2

Denna servlet läser värde på sessionsvariabel (`servletBona`) samt läser bönas värdena för utskrift.

```
package bpn;

import javax.servlet.*;
```

```

import javax.servlet.http.*;
import java.io.*;
import bpn.EnkelJavaBean;           //Importer JavaBean

public class JavaBeans2Servlet extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException {
        EnkelJavaBean bona = null;

        response.setContentType("text/html");    //Ange typ av data som returneras
        PrintWriter out = response.getWriter();  //Hämta utström

        //Hämta session för besökare - skapa om inte existerar
        HttpSession session = request.getSession(true);
        bona = (EnkelJavaBean)session.getAttribute("servletBona"); //Läs sessionsvar.

        out.print("<html><body>");                //Påbörja HTML-dokument
        out.print("<h1>Servlet 2</h1>");
        out.print("<p>Denna servlet läser data från JavaBean.</p>");
        out.print("<p>Användaridentitet är: " + bona.getAnvId() + "</p>");
        out.print("<p>Namn är: " + bona.getNamn() + "</p>");
        out.print("<p><a href=\"bpn.JavaBeans1Servlet\">Servlet 1</a></p>");
        out.print("</body></html>");            //Avsluta HTML-dokument

        out.close();                            //Stäng utström
    } //doGet()
} //class JavaBeans2Servlet

```

#### 4.4.5 *redirect* och *forward* i servlets

De flesta webbgränssnitt stödjer *redirect*, d.v.s. vidarebefodran. Egentligen är en *redirect* en del av HTTP-protokollet för att tala om för klienter att resurs t.ex. flyttats och vänligen ber klienten att ladda en ny adress.

En *forward* fungerar på ett liknande sätt som *redirect* – skillnaden ligger i att webbserver inte ber klienten ladda en ny adress. Webbserver exekverar istället en annan servlet (eller JSP) som om det vore aktuell adress. Fördelen med *forward* är att bl.a. synlig då vi använder formulär vi inloggning: besökaren fyller i ett formulär som skickas till servlet för validering, om validering gick bra så returneras en webbsida med positivt besked. Annars returneras formuläret igen med ett negativt besked och besökaren kan försöka fylla i formuläret igen. Allt detta sker utan att klienten är medveten om att olika servlets/JSP exekveras.

För att *redirect* och *forward* ska fungera ordentligt så bör dessa göras **innan** någon data skickats till klienten!

I detta exempel använder vi en kryssruta i ett formulär för att avgöra om *redirect* eller *forward* ska utföras.

#### HTML-fil med formulär

Formuläret (i `redirect_forward1.html`) innehåller en kryssruta (`doForward`) och en skickaknapp. Om besökaren bockar för kryssruta så kommer *forward* utföras – annars *redirect*.

```

<html>
<body>
<h1>Utföra redirect eller forward - Sida 1</h1>
<form action="servlet/bpn.RedirectForward2Servlet">
  <p>Utföra forward? <input type="checkbox" name="doForward"></p>
  <p><input type="submit" value="Skicka"></p>
</form>
</body>
</html>

```

## Servlet som utför *redirect* eller *forward*

I den andra servlet (`RedirectForward2Servlet`) kontrolleras om kryssruta är förbockad eller inte. Om den är förbockad så utförs en *forward*, annars en *redirect*.

I exemplet visas även hur vi kan bifoga attribut till sida som *forward* utförs till (men inte *redirect*). För att göra detta så använder vi metoden `setAttribute()` i `Request`-objekt. I exempel har ett attribut `doneForward` med värdet "true" (som sträng!) skickats.

```
package bpn;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class RedirectForward2Servlet extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException {
        static String cstrUrl = "bpn.RedirectForward3Servlet"; //URL att skickas till
        String strDoForward = null;

        //Hämta ev. värde på fält - kryssrutor skickas endas om förbockad
        strDoForward = request.getParameter("doForward");

        //Om kryssruta var förbockad, dvs värde kunde hämtas
        if(strDoForward != null)
        {
            //Kontrollera att fältet innehåller något
            if(strDoForward.length() > 0)
            {
                //Hämta dispatcher
                RequestDispatcher rd = request.getRequestDispatcher(cstrUrl);
                request.setAttribute("doneForward", "true"); //Lägg t. attribut i request
                try {
                    rd.forward(request, response); //Utför forward
                }
                catch(ServletException se) {
                    se.printStackTrace();
                }
            }
        }
        else //... annars utför en redirect, dvs be webbläsare ladda ny URL
            response.sendRedirect(cstrUrl);

        //Nedanstående kod bör aldrig exekveras (om inte fel)
        response.setContentType("text/html"); //Ange typ av data som returneras
        PrintWriter out = response.getWriter(); //Hämta utström

        out.print("<html><body>"); //Påbörja HTML-dokument
        out.print("<h1>Utföra redirect eller forward - Sida 2</h1>");
        out.print("<p>Denna webbsida bör aldrig visas...</p>");
        out.print("</body></html>"); //Avsluta HTML-dokument
        out.close(); //Stäng utström
    } //doGet()
} //class RedirectForward2Servlet
```

## Servlet som besökare skickas till

Den sista servlet (`RedirectForward3Servlet`) är webbsida som besökare skickas till när exekvering i andra servlet utförs. Om det är *redirect* eller *forward* kan bl.a. avgöras med hjälp av URL – om URL innehåller filnamn för aktuell servlet (d.v.s. filnamn innehåller en 3:a) så är det *redirect*. Men om filnamnet innehåller en 2:a så är det *forward*.

Här visas hur vi kan läsa eventuella parametrar som skickats vid *forward* i andra JSP. Att läsa parametrar fungerar som att läsa fält i formulär. Parametrar skickas dock alltid med HTTP POST.

```
package bpn;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
```

```

public class RedirectForward3Servlet extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException {
        String strDoneForward = null, strMeddelande = null;

        //Hämta ev. attribut i request satt vid forward
        strDoneForward = (String)request.getAttribute("doneForward");

        //Om attributvärde finns - har satts...
        if(strDoneForward != null)
        {
            //Om värde är true - ange hur besökare skickats till servlet
            if(strDoneForward.equals("true"))
                strMeddelande = "forward utförd";
        }
        else //... annars har redirect utförts
            strMeddelande = "redirect utförd";

        response.setContentType("text/html"); //Ange typ av data som returneras
        PrintWriter out = response.getWriter(); //Hämta utström

        out.print("<html><body>"); //Påbörja HTML-dokument
        out.print("<h1>Utföra redirect eller forward - Sida 3</h1>");
        out.print("<p>Du har blivit skickad till denna webbsida med redirect eller "
            + "forward. ");
        out.print("Vilket avgörs om filnamnet innehåller en 3:a resp. inte.</p>");
        out.print("<p>" + strMeddelande + "</p>");
        out.print("</body></html>"); //Avsluta HTML-dokument

        out.close(); //Stäng utström
    } //doGet()
} //class RedirectForward3Servlet

```

## 4.5 Mer om servlets

När vi skapar en ny servlet så brukar det vara en bra idé att ha utvecklat webbsida i ett hemsidesprogram (WYSIWYG-editor som FrontPage) eller en ”vanlig” editor samt testat webbsida i webbläsare. Därefter kan vi kopiera HTML-kod och lägga till i anrop av print-metoderna. Att skapa (designa) webbsidor direkt i servlet är en komplexitet som oftast bara tar tid...

Ett problem vi kan råka utför är att vi måste använda delade resurser som kräver att vi synkroniserar användningen av resurserna. För detta kan vår servlet implementera gränssnittet `SingleThreadModel`.

### 4.5.1 SingleThreadModel

Gränssnittet `SingleThreadModel` innehåller inga metoder utan används för att ”typa” en servlet som entrådig, d.v.s. att endast en tråd kan exekvera en metod i servlet åt gången. Servlets som inte implementerar detta gränssnitt är flertrådiga som standard. (Webbcontainer kan hantera enkeltrådiga servlets genom att använda en instanspool eller genom att arrangera alla anrop av metoden `service()` i sekvens [*serialized*].)

Flertrådiga servlets kan hantera samtidigheten (*concurrency*) genom att inte använda några instansvariabler (som är globala för servlet) eller synkronisera användning av delade resurser (så som instansvariabler och databaskopplingar). Detta är oftast inte ett problem då HTTP är tillståndslöst, d.v.s. vi har oftast inte behov av instansvariabler. ☺

## 4.5.2 Servlets och alias

I utvecklings-/testmiljö brukar man konfigurera så att servlets kan laddas med en URL i stil med `/servlet/KlassNamn`.<sup>27</sup> I Resin används t.ex. följande konfiguration i `web.xml` för att det ska fungera.

```
<servlet-mapping url-pattern="/servlet/*" servlet-name="invoker" />
```

I produktionsmiljö är det standard att skapa alias för servlets istället för att använda ovanstående nämnda sökväg. Alias skapas genom att lägga till två taggar i konfigurationsfil för webbapplikation (`web.xml` i Resin). De första taggen, `<servlet>`, används för att skapa ett "alias" för en klass, d.v.s. ett namn som kan användas istället för klassens fulla namn (*fully qualified name* – paket och klassnamn). I exempel nedan skapas ett alias `EnkeltNamn` för klassen `bpn.EnkelServlet`. För att exekvera servlet så kan en URL (för webbapplikation `bpn`) i stil med `http://localhost:8080/bpn/servlet/EnkeltNamn` användas.

```
<servlet>
  <servlet-name>EnkeltNamn</servlet-name>
  <servlet-class>bpn.EnkelServlet</servlet-class>
</servlet>
```

Den andra taggen, `<servlet-mapping>`, används för att tala om vilken URL som ska användas för att exekvera servlet. Denna tagg måste användas i samband med taggen ovan. I exempel nedan (som bygger vidare på exempel med taggen ovan) så anges att URL `/EnkelUrl` (i aktuell webbapplikation) ska kunna användas för servlet med alias `EnkeltNamn`. För att exekvera servlet så kan vi nu använda en URL (för webbapplikation `bpn`) i stil med `http://localhost:8080/bpn/EnkelUrl` användas.

```
<servlet-mapping>
  <url-pattern>/EnkelUrl</url-pattern>
  <servlet-name>EnkeltNamn</servlet-name>
</servlet-mapping>
```

## 4.5.3 Nackdelen med servlets

Största nackdelen med servlet är att kod för presentation och logik har blandats, d.v.s. Java-koden kan bli ganska svårt läst. En annan nackdel är att HTML-koden (presentationen) brukar skapas av webbdesigners, vilka ofta inte har en förståelse för Java-programmering. Samtidigt måste servlet kompileras om vid varje ändring av HTML-kod.

Därför utvecklades Java Server Pages (JSP), vilket nästa kapitel tittar på. Men JSP är **inte** en ersättning för servlets utan ett komplement!

---

<sup>27</sup> Sökvägen används främst i utvecklings- och testmiljö – detta är nämligen en säkerhetsrisk.

## 5 Java Server Pages (JSP)

JSP har utvecklats för att överbrygga nackdelarna med servlets, d.v.s. blandningen av presentation och logik. En JSP är HTML-kod med små snuttar av kod (skriptkod). För att göra JSP mer effektiv än andra *server-side scripting* teknologier (t.ex. PHP och Microsofts ASP) så kompileras JSP till servlets första gången de begärs. Efterföljande begäran använder sig av de redan kompilerade servlets som JSP resulterar i (om inte JSP ändrats ☺).

JSP är tänkta att komplettera servlets, **inte** ersätta! Men JSP kan "missbrukas" genom att en massa logik placeras i skriptkod. Tanken är att man skapar ett gränssnitt i HTML och använder skriptkoden för att anropa logik i servlets, JavaBeans eller EJB. Det finns möjlighet att skapa anpassade JSP-taggar (*custom tags*) – taggar som exekverar kompilerad kod – och som skapats av Java-programmerare. På detta sätt kan webbdesigners skapa HTML-koden och använda de anpassade JSP-taggar för att erhålla funktionalitet.

---

### 5.1 Strukturen av en JSP

En JSP består av text, kommentarer, direktiv, skriptelement och händelser (*actions*) – vilka beskrivs nedan.

#### 5.1.1 Text

Text är vad det låter som – "ren" text, eller snarare (X)HTML-/XML-kod. Text tolkas inte av webbserver utan skickas direkt till klienten (webbläsaren). Detta är (eller bör vara) den övervägande delen av en JSP.

```
<HTML>
<BODY>
  <!-- Här placeras t.ex. HTML-kod för design och dynamiskt innehåll -->
</BODY>
</HTML>
```

#### 5.1.2 Kommentar

Även i JSP kan vi använda kommentarer. Utöver kommentartecken i Java-kod kan denna speciella form av kommentar användas. Fördelen med denna typ av kommentarer är att vi bl.a. kan "kommentera bort" direktiv och händelsetaggar (se nedan för förklaring av dessa).

Kommentar skrivs inom öppnande tagg `<%--` och avslutande tagg `--%>`. Observera att bindestrecket måste skrivas ihop med procenttecknet (d.v.s. utan mellanslag) och att även avslutande tagg har ett procenttecken.

```
<HTML>
<BODY>
<%--
  Detta är en JSP-kommentar. I motsats till HTML-kommentarer så syns detta inte i
  den resulterande HTML-koden.
--%>
</BODY>
</HTML>
```

#### 5.1.3 Direktiv

Direktiv påverkar exekveringen i en JSP. Exempel på direktiv är t.ex. språk<sup>28</sup> som används i skriptelement och inkludering av filer.

---

<sup>28</sup> JSP är **inte** begränsat till att endast använda Java som språk. Det finns även JSP-motorer som klarar JavaScript (läs Resin – fr.o.m. version 3 av Resin så stöds dock inte JavaScript längre).



Direktiv anges inom öppnande tagg `<%@` och avslutande tagg `%>`. Observera att snabel-a:et (`@`) måste skrivas ihop med procenttecknet (d.v.s. utan mellanslag).

I exempel nedan används attributet `language` för att ange språk (standard är oftast Java) och attributet `contentType` för att innehållet som returneras till klient är HTML (vilket oftast är standard). Dessa två direktiv kan alltså oftast utelämnas men bör ändå tas med för att vara säker på att JSP fungerar om de flyttas till en annan server.

```
<%@ page language="java" contentType="text/html" %>
<HTML>
<BODY>
  <!-- Här placeras t.ex. HTML-kod för design och dynamiskt innehåll -->
</BODY>
</HTML>
```

### 5.1.4 Skriptelement

Skriptelement kan i sig bestå av deklaratiorer, *scriptlets* (kodsnuttar) och uttryck. Det är i skriptelementen som exekveringen sker, d.v.s. där det dynamiska innehållet skapas i en JSP.

#### Deklarationer

Deklarationer motsvarar instansvariabler och -metoder i en Java-klass (d.v.s. i den servlet som JSP resulterar i). Variablerna och metoderna är tillgängliga för alla klienter, d.v.s. delas mellan alla klienter av servlet. Av denna anledning bör man undvika att deklarerar variabler i denna typ av skriptelement eftersom vi får problem med samtidig åtkomst (synkronisering).

Variabler och metoder deklarerar mellan öppnande tagg `<%!` och avslutande tagg `%>`.

I exempel nedan deklarerar en variabel `dblPi` av typen `double`.

```
<%!
  double dblPi = 3.14;
%>
<HTML>
<BODY>
  <!-- Här placeras t.ex. HTML-kod för design och dynamiskt innehåll -->
</BODY>
</HTML>
```

#### Scriptlets

Scriptlets är snuttar av Java-kod som skrivs mellan öppnande tagg `<%` och avslutande tagg `%>`. Java-koden i scriptlets konverteras till metoden `doGet()`, d.v.s. eventuella variabler som deklarerar i detta skriptelement kommer vara lokala variabler i metoden. Och varje begäran av en servlet (eller snarare exekvering av metoden `doGet()`) kommer exekveras av en egen tråd, d.v.s. vi får inga problem med samtidighet (synkronisering).

För att skriva ut ett resultat från exekvering i JSP så används variabeln `out` och någon av `print`-metoderna, d.v.s. `print()` och `println()`. Variabeln `out` innehåller en instans av typen `PrintWriter` – referensen som vi hämtar från `Response`-objektet i en servlet.

I nedanstående exempel deklarerar (lokala) variabler som adderas och vars summa skrivs ut.

```
<HTML>
<BODY>
<P>1 + 1 är
<%
  int intA = 1, intB = 1;    //Deklarera variabler och tilldela värde

  out.println(intA + intB); //Skriv ut variabler adderade
%>
</P>
</BODY>
</HTML>
```

## Uttryck

Uttryck används för att skriva ut en variabel, eller resultatet av en programsats, och skrivs mellan öppnande tagg `<%=` och avslutande tagg `%>`. Detta är ett annat sätt att skriva ut från JSP än att använda metoderna `print()/println()` i objektet `out`. Vi kan **endast** skriva ut en sak med dessa taggar.

I nedanstående exempel deklarerar en (instans-)variabel `dblPi` som sen skrivs ut som ett uttryck.

```
<%! //Deklaration som skrivs ut som ett uttryck nedan
    double dblPi = 3.14;
%>
<HTML>
<BODY>
<P>Pi är <%= dblPi %></P>
</BODY>
</HTML>
```

### 5.1.5 Händelser (*actions*)

Även händelser används för att exekvera logik i JSP. Händelser är taggar som ersätts med motsvarande Java-kod för vad som ska utföras. Några av de mest använda förklaras kort nedan (och beskrivs i kommande avsnitt).

- `<jsp:useBean>` – används för att instansiera en `JavaBean` (**inte EJB!**).
- `<jsp:setProperty>` – används för att sätta ett värde på attribut i `JavaBean`.
- `<jsp:getProperty>` – används för att hämta ett värde på attribut i `JavaBean`.
- `<jsp:include>` – inkluderar en webbresurs (som kan vara ”dynamisk”), vilket beskrivs i avsnittet *Inkludera webbresurser* nedan.
- `<jsp:forward>` – skickar vidare exekveringen till annan JSP eller servlet **utan** att blanda in klienten (d.v.s. detta är inte en *redirect*), vilket förklaras i avsnittet *redirect och forward i JSP*.
- `<jsp:param>` – används för parametrar att skicka till t.ex. inkluderade webbresurser.<sup>29</sup>

Taggarna `useBean`, `setProperty` och `getProperty` är för att använda `JavaBeans` (inte EJB!) och beskrivs i avsnittet *Använda JavaBeans* nedan. Och den sista taggen, `<jsp:param>`, förklaras i samband med taggarna `<jsp:include>` och `<jsp:forward>`.

### Anpassade taggar

Java-programmerar kan skapa bibliotek med egna händelsetaggar (*custom tag libraries*). Dessa taggar kommer att ersättas med Java-kod när JSP kompileras till servlet. Genom att utveckla egna taggar kan utvecklare skriva logik som webbdesigners efterfrågar – d.v.s. webbdesigners använder taggarna och Java-programmerare skriver logiken (rätt man/kvinna på rätt plats ☺).

Ämnet behandlas inte i denna sammanfattning.

---

<sup>29</sup> Denna händelse ska inte förväxlas med `<jsp:params>` som används med taggen `<jsp:plugin>` (som inte beskrivs i denna sammanfattning).

### 5.1.6 ”Inbyggda” objekt

I JSP finns ett antal ”inbyggda” (*implicit*) objekt – objekt vars referenser vi bl.a. måste fråga efter i en servlet. Referenser till dessa inbyggda objekt finns i följande variabler.

- `request` – Request-objekt som motsvaras av parameter till metoder som `doGet()` i servlet.
- `response` – Response-objekt som motsvaras av parameter till metoder som `doGet()` i servlet.
- `out` – `PrintWriter`-objekt som motsvaras av referens som erhålles när vi anropar `getWriter()` i servlet.
- `session` – referens till objekt som håller reda på klients sessionsdata.
- `application` – referens till objekt som håller reda på applikationens data.

Utöver ovanstående variabler så finns några till – se JSP-dokumentation för beskrivningar av dessa.

---

## 5.2 Funktioner i JSP

Nedan beskrivs några av de mest användbara ”funktionerna” i JSP, d.v.s. saker som används relativt ofta.

### 5.2.1 Selektion och HTML-kod

Selektion, d.v.s. if-satser, kan användas för att avgöra vilken HTML-kod som ska användas. Detta gör att vi slipper använda komplexa strängar med print-metoderna (det som var en av servlets nackdelar).

I detta enkla exempel så visas olika HTML-kod beroende på tid, d.v.s. olika hälsningsfraser. Vi börjar med att importera klassen `Calendar` som vi ska använda för att ta reda på timme i aktuellt klockslag. Sen görs en if-sats med fyra ”grenar” – en för morgon (före 9), en för förmiddag (mellan 9 och 12), en för eftermiddag (mellan 12 och 18) och en för kvällen (efter 18).

**Observera** att vi **alltid måste** använda hakparenteser (d.v.s. `{` och `}`) för att innesluta HTML-kod i selektioner! Detta gäller även om vi endast har en rad HTML-kod i if-satsernas ”grenar”, som i detta exempel.

```
<%@ page import="java.util.Calendar" %>
<%
    int intTimme = Calendar.getInstance().get(Calendar.HOUR);
%>
<HTML>
<BODY>
<% if(intTimme < 9) { %>
<P>God morgon!</P>
<% } else if(intTimme <12) { %>
<P>God förmiddag!</P>
<% } else if(intTimme <18) { %>
<P>God eftermiddag!</P>
<% } else { %>
<P>God kväll!</P>
<% } %>
</BODY>
</HTML>
```

Observera att det är ganska lätt att glömma ”hakparenteser” då koden blir mer svårläst. Nu brukar dock servlet-motorer visa felmeddelande om detta. ☺

## 5.2.2 Inkludera webbresurser

Inkludering i webbgöransnitt innebär att man infogar externa resurser i en annan resurs. Detta görs för att bl.a. kunna dela på gemensam kod (d.v.s. slippa upprepning och endast behöva uppdatera på ett ställe). Det finns två sätt att inkludera webbresurser: med direktiv eller händelsetagg. I det första fallet så kan endast statiska webbresurser inkluderas, t.ex. filer med sidhuvud och sidfötter.

### Inkludera med direktiv

Som ett exempel så skulle t.ex. följande kod finnas i filen `sidhuvud.html` (som ger vit text mot blå bakgrund)

```
<p style="background-color: blue; color:white">Text i sidhuvud</p>
```

och följande kod i filen `sidfot.html` (som ger ”standardfärg” på text mot gul bakgrund).

```
<p style="background-color: yellow">Text i sidfot</p>
```

Filerna ovan inkluderas i nedanstående fil.

```
<HTML>
<BODY>
<% include file="sidhuvud.html" %>
<p>Denna text finns i huvudfilen.</p>
<% include file="sidfot.html" %>
</BODY>
</HTML>
```

### Inkludera med händelsetagg

I det andra fallet, det med händelsetagg, så kan även dynamiska resurser, så som servlets och andra JSP, inkluderas. Nedanstående visar motsvarande kod i filen som inkluderar dom andra (d.v.s. filerna `sidhuvud.html` och `sidfot.html` i exempel ovan).

```
<HTML>
<BODY>
<jsp:include page="sidhuvud.html" flush="true" />
<p>Denna text finns i huvudfilen.</p>
<jsp:include page="sidfot.html" flush="true" />
</BODY>
</HTML>
```

## 5.2.3 JSP och formulär

Formulärhantering i webbsidor kan ske på ett antal olika sätt. I början är det oftast lättast att börja med en HTML-fil med formuläret och en JSP-fil som ”behandlar” formuläret. När man börjar bli lite ”varm i kläderna” så kan man placera både formulär och logik för att ”behandla” formuläret i samma JSP-fil. Dessa två sätt kommer visas nedan.

Ytterligare ett sätt är att använda en JSP-fil med formuläret och servlet för att ”behandla” formuläret samt en HTML- eller JSP-fil för att presentera resultatet. Detta behandlas i nästa kapitel.

## Exempel med HTML- och JSP-fil

För att läsa data från formulär så används metoden `getParameter()` i variabeln `request`. Metoden returnerar en sträng som representerar värdet i formulärets fält, vars namn skickats som parameter till metod. Om fältet inte finns i formulär så returneras `null`. Observera att namnet på fältet är känsligt för skillnad i skiftläge (d.v.s. skiljer på gemener och versaler)!

I detta enkla exempel så skapas en webbsida (`jsp_formular1.html`) med två textfält (för namn och födelseår) och en skicka-knapp. I formuläret används attributet `action` för att tala om till vilken webbsida som formuläret ska skickas (`jsp_formular2.jsp` i detta exempel).

```
<html>
<body>
<p>Fyll i formulär och klicka på Skicka-knappen.</p>
<form action="jsp_formular2.jsp">
  <p>Namn: <input type="text" name="namn"></p>
  <p>Födelseår: <input type="text" name="fodelsear"></p>
  <p><input type="submit" value="Skicka"></p>
</form>
</body>
</html>
```

I JSP-sidan (`jsp_formular2.jsp`) så importeras klassen `Calendar` som behövs för att ta reda på aktuellt år så att besökares ålder kan räkna ut. Detta motsvarar import-satser i Java-klasser. Sen deklaras variabler för att lagra värden från formulär – kom ihåg att `getParameter()` returnerar en sträng även om fältet innehåller ett tal. Vi måste alltså konvertera strängen till ett tal. I webbsidans ”kropp” används sen uttryck för att skriva ut namnet på personen samt differensen mellan aktuellt år och personens födelseår.

```
<%@ page import="java.util.Calendar" %>
<%
  String strNamn = null, strAr = null;
  int intAr = 0, intAktuelltAr;

  //Hämta värden från fält i formulär
  strNamn = request.getParameter("namn");
  strAr = request.getParameter("fodelsear");

  intAr = Integer.parseInt(strAr); //Konvertera sträng till tal

  //Hämta aktuellt år
  intAktuelltAr = Calendar.getInstance().get(Calendar.YEAR);
%>
<html>
<body>
<p>Ditt namn är <%= strNamn %> och du är <%= intAktuelltAr - intAr %> år
gammal.</p>
</body>
</html>
```

## Exempel med endast JSP-fil

Att endast använda JSP-fil gör genast koden mer komplex. Principen är att vi kontrollerar om alla obligatoriska fält i formulär fyllts i – om inte så visas formulär och eventuellt

felmeddelande.<sup>30</sup> Annars visas ”resultatet” av formulärets ”behandling”. För att göra det lite lättare att följa kod så används lämpligen ”flaggor” – flaggor som vi ”sätter” för att ange att något är sant (t.ex. `blnVisaForm` som används för att avgöra om formulär ska visas eller inte).

Om formuläret fyllts i felaktigt så är det praktiskt om felaktig data visas i textutor så att besökare kan korrigera felaktighet. För detta använda attributet `value` i input-taggen nedan tillsammans med JSP-uttryck.

```
<%@ page import="java.util.Calendar" %>
<%
    boolean blnVisaForm = true; //Flagg om formulär ska visas
    String strNamn = null, strAr = null, strFel = "";
    int intAr = 0, intAktuelltAr;

    //Hämta värden från fält i formulär
    strNamn = request.getParameter("namn");
    strAr = request.getParameter("fodelsear");

    //Om fältet namn fanns... och om det innehåller något...
    if(strNamn != null)
        if(strNamn.length() > 0) {
            try {
                intAr = Integer.parseInt(strAr); //Konvertera sträng till tal
                blnVisaForm = false;
            }
            catch(NumberFormatException nfe) {
                strFel = "<b>FEL:</b> Födelseår måste vara ett årtal!";
            }
        }
        else //... annars meddela besökare om att namn krävs.
            strFel = "<b>FEL:</b> Du måste fylla i ett namn!";

    //Hämta aktuellt år
    intAktuelltAr = Calendar.getInstance().get(Calendar.YEAR);
%>
<html>
<body>
<% if(blndVisaForm) { %>
<p>Fyll i formulär och klicka på Skicka-knappen.</p>
<form action="jsp_formular3.jsp">
    <p>Namn: <input type="text" name="namn" value="<%= strNamn %>"></p>
    <p>Födelseår: <input type="text" name="fodelsear" value="<%= strAr %>"></p>
    <p><input type="submit" value="Skicka"></p>
</form>
<p><%= strFel %></p>
<% } else { %>
<p>Ditt namn är <%= strNamn %> och du är <%= intAktuelltAr - intAr %> år
gammal.</p>
<% } %>
</body>
</html>
```

I exempel ovan bör eventuellt innehållet i fältet `fodelsear` kontrolleras att det innehåller siffror innan konvertering till ett tal sker, d.v.s. här kan fel uppstå. ☺

## 5.2.4 Sessionshantering

Varje klient som begär en resurs på webbserver motsvaras av en session. En session hanteras oftast genom att webbserver skickar en cookie till klient – en cookie som sen klient bifogar med alla efterföljande begäran.<sup>31</sup> På detta sätt kan en webbserver hålla reda på vilken klient

<sup>30</sup> Detta innebär givetvis att första gång JSP visas så kommer ”fel” uppstå, d.v.s. `getParameter()` kommer returnera `null` eftersom formulär inte finns och därmed inte fält heller.

<sup>31</sup> Observera att om klienten inte stödjer cookies så fungerar inte sessionshantering som det ska. Det kan visserligen lösas med *URL rewriting* – cookie skickas som del av URL.

som är vilken (och vilka klienter som är ”anslutna”<sup>32</sup>) samt lagra data relaterad till klienterna. Sessionsdata görs tillgänglig via variabeln `session` i JSP. En session brukar som standard vara 20 minuter efter att klient begärde sista resursen från webbserver.<sup>33</sup>

Sessionsdata, eller ”sessionsvariabler”<sup>34</sup>, skrivs och läses med metoderna `setAttribute()` respektive `getAttribute()` i variabeln `session`. Den första metoden har två parametrar – en sträng för namnet på sessionsvariabeln och en för själva värdet, av typen `Object`, som ska lagras. Metod nummer två har endast en parameter – en sträng för namnet på variabeln – och returnerar ett värde av typen `Object`. Om variabel inte satts så returnerar `getAttribute()` värdet `null`.

Ett användningsområde för sessionsdata är t.ex. i en webbutik där klienter handlar artiklar eller för att hålla reda på data om inloggad besökare.

## Enkelt exempel med räknare i sessionsvariabel

I detta exempel tittar vi på hur vi kan använda sessionsdata för att lagra antalet webbsidor som klient besökt. Exemplet visar även hur vi kan kontrollera om en viss sessionsvariabel har getts ett värde.

Som i de flesta applikationer så börjar vi med att deklarera variabler – en av typen `Object` då `getAttribute()` returnerar en instans av typen och en av typen `int` för själva värdet på räknaren. Sen frågar vi efter eventuellt värde på sessionsvariabeln `raknare` – ett värde som vi kontrollerar att det existerar (d.v.s. inte är `null`). Om värdet är skilt från `null` så kan vi konvertera från typen `Object` till `Integer` – ett objekt vi kan fråga om heltalsvärdet. Heltalet, räknaren, kan sen ökas med 1. Om värdet inte finns (sessionsvariabeln inte har ett värde) så är det besökarens första besök i aktuell session och vi sätter räknaren till 1. Sist skriver vi räknaren till (eller tillbaka till) sessionsvariabeln – vilket vi kan göra genom att använda `wrapper`-klassen `Integer` till. Vi måste använda `wrapper`-klass eftersom metoden `setAttribute()` kräver en parameter av typen `Object` för värdet.

```
<%
  Object objRaknare = null; //Deklarera variabler
  int intRaknare = 0;

  //Hämta ev. värde i sessionsvariabel raknare
  objRaknare = session.getAttribute("raknare");

  //Om värde finns - konvertera till int och plussa på värde...
  if(objRaknare != null)
  {
    intRaknare = ((Integer)objRaknare).intValue();
    intRaknare++;
  }
  else //... annars sätt värde till 1
    intRaknare = 1;

  //Skriv värde till sessionsvariabel - kräver konvertering till Integer
  session.setAttribute("raknare", new Integer(intRaknare));
%>
<html>
<body>
<h1>Sida 1</h1>
<p>Du har besökt <%= intRaknare %> webbsidor hos oss.</p>
```

<sup>32</sup> Jag har satt ”anslutna” inom citattecken då HTTP är ett tillståndslöst protokoll. Det är nackdelen med HTTP som sessioner försöker lösa.

<sup>33</sup> Detta ger möjlighet för besökare att tänka lite, t.ex. vid ifyllande av formulär, eller springa och hämta kaffe. ☺

<sup>34</sup> Jag kallar det för ”sessionsvariabler” eftersom det är en form av namngivna variabler som gäller för klients session.

```
<p><a href="jsp_session2.jsp">Sida 2</a></p>
</body>
</html>
```

## Använda sessionsvariabler för att hantera inloggning

Ett, av många sätt, att hantera inloggade besökare är att använda sessionsvariabler. Principen bygger på att besökare fyller i ett formulär, d.v.s. användaridentitet och lösenord. Formuläret skickas sen till webbserver som validerar besökares identitet. Om validering gick bra så lagras t.ex. användaridentiteten i en sessionsvariabel. Varje webbsida som kräver inloggning kan då kontrollera att användaridentitet finns lagrad i en sessionsvariabel. Om så inte är fallet så kan besökare skickas till formulär för inloggning.

### Inloggningsformulär

Inloggningsformuläret (`session_inloggning.jsp`) innehåller ett textfält för användaridentitet (`anvId`), ett lösenordsfält (`type="password"`) för lösenord (`losenord`) och en skickaknapp. Längst ner finns även en länk till den privata sidan, d.v.s. den som kräver inloggning, för test.

```
<html>
<body>
<p>Fyll i användaridentitet och lösenord samt klicka på knappen Logga in för
att verifiera din identitet.</p>
<form action="session_verifiera.jsp">
  <p>Användaridentitet: <input type="text" name="anvId"> (abc00001)</p>
  <p>Lösenord: <input type="password" name="losenord"> (gurka)</p>
  <p><input type="submit" value="Logga in"></p>
</form>
<p><a href="session_privat.jsp">Privat sida</a> (som kräver inloggning).</p>
</body>
</html>
```

Denna webbsida innehåller endast HTML-kod, d.v.s. det skulle kunna vara en vanlig HTML-fil (med filändelsen `.html`). Men om vi i framtiden vill göra om inloggningsfunktionen och t.ex. skicka felmeddelande till inloggningsformulär så måste denna webbsida vara en JSP (se beskrivning i nästa JSP nedan).

### JSP för att verifiera användaridentitet och lösenord

JSP (`session_verifiera.jsp`) som "behandlar" data i formulär börjar med att deklarera variabler. Här används variabler för rubrik för webbsida och meddelande som ska visas samt en flagga för att avgöra vilken länk som ska visas (tillbaka eller fortsatt). Sen hämtas data från formulär, och eftersom dessa kan vara `null` (om fält inte finns i formulär) så testas vi att de inte är det.

Ibland händer det att besökare klickar på skickaknapp (eller trycker på Returtangenten) av misstag – därför testas om textfält innehåller tomma strängar. Om fälten inte är tomma så testas om dess innehåll innehåller en existerande användaridentitet och dess lösenord. I detta exempel används en "hårdkodad" användaridentitet (`abc00001`) och lösenord (`gurka`), men här kan t.ex. en uppslagning ske mot databas. Om användaridentitet verifierats så skrivs till sessionsvariabeln `anvId`, rubrik för webbsida sätts liksom meddelande som ska visas för besökare och vi sätter flaggan `blnInloggad` så att rätt länk visas.

I HTML-koden skrivs rubrik och meddelande ut och flaggan `blnInloggad` används för att avgöra om vi ska visa länk till privat sida eller tillbaka till inloggningsformulär (eller snarare back i webbläsarens "historia" – d.v.s. vi använder JavaScript-koden `history.back()`).



```

<%
String strAnvId = null, strLosenord = null;
String strRubrik = "Fel!", strMeddelande = null;
boolean blnInloggad = false; //Flagga om besökare verifierad, dvs inloggad

//Hämta data från formulär (om något...)
strAnvId = request.getParameter("anvId");
strLosenord = request.getParameter("loosenord");

//Om formulär innehåller fälten anvId och losenord - formulär korrekt
if((strAnvId != null) && (strLosenord != null))
{
    //Om fälten innehåller tecken - fortsätt
    if((strAnvId.length() > 0) && (strLosenord.length() > 0))
    {
        //Om anvId och lösenord är "riktiga" - skriv anvId till sessionsvar.
        if(strAnvId.equals("abc00001") && strLosenord.equals("gurka"))
        {
            session.setAttribute("anvId", strAnvId);
            strRubrik = "Gratulerar!";
            strMeddelande = "<p>Din identitet har verifierats abc00001!</p>";
            blnInloggad = true;
        }
        else
            strMeddelande = "<p>Användaridentitet och/eller lösenord är felaktigt."
                + " Försök igen!</p>";
    }
    else
        strMeddelande = "<p>Användaridentitet och/eller lösenord har <b>inte</b>"
            + " skickats.</p>";
}
else
    strMeddelande = "<p>Formulär är felaktigt. Kontakta administratör och meddela"
        + " detta!</p>";
%>
<html>
<body>
<h1><%= strRubrik %></h1>
<%= strMeddelande %>
<% if(blnInloggad) { %>
<p><a href="session_privat.jsp">Fortsätt</a> till privat hemsida.</p>
<% } else { %>
<p><a href="javascript:history.back()">Backa</a> och försök igen.</p>
<% } %>
</body>
</html>

```

En mer praktisk lösning är att skicka tillbaka besökaren till inloggningsformuläret om användaridentitet och/eller lösenord är felaktiga samt där meddela att de är felaktiga. För detta kan vi använda *redirect* eller ännu bättre *forward* (se avsnitt *redirect och forward i JSP* nedan för hur dessa fungerar). För att detta ska fungera som måste webbsida med inloggningsformulär vara en JSP (vilket hänvisas till i beskrivning av JSP ovan).

### JSP med privat information

Denna JSP (`session_privat.jsp`) börjar också den med att deklarera en del variabler. Eftersom sessionsvariabler lagras som typen `Object` så måste vi först hämta till en variabel av denna typ. Om sessionsvariabel är satt, d.v.s. `getAttribute()` inte returnerar `null`, så kan vi konvertera det till en sträng. Även här används en flagga, `blnInloggad`, för att avgöra om besökare är inloggad och vilket meddelande (och länkar) som ska visas.

Om besökare är inloggad så visas vem besökare är inloggad som samt en länk till utloggningssida. Men om besökare inte är inloggad så visas besked om detta och en länk till inloggningsformulär.

```

<%

```

```

Object objAnvId = null;
String strAnvId = null;
boolean blnInloggad = false; //Flagga f att avgöra om innehåll i sida ska visas

objAnvId = session.getAttribute("anvId"); //Hämta sessionsvariabel anvId

//Om sessionsvariabel satt - konvertera till sträng o sätt flagga blnInloggad
if(objAnvId != null)
{
    strAnvId = (String)objAnvId;
    blnInloggad = true;
}
%>
<html>
<body>
<h1>Privat sida</h1>
<p>För att kunna se innehållet i denna webbsida måste man vara inloggad.</p>
<% if(blnInloggad) { %>
<p>Du är inloggad som <%= strAnvId %>.</p>
<p><a href="session_utloggning.jsp">Logga ut</a></p>
<% } else { %>
<p>Du är <b>inte</b> inloggad! <a href="session_inloggning.jsp">Logga in</a> för
att se innehållet.</p>
<% } %>
</body>
</html>

```

Även här kan vi använda *redirect* eller *forward* för att skicka besökare till inloggningsformulär och där meddela att man måste vara inloggad för att se den privata webbsidan.

### JSP för utloggning

JSP för utloggning (`session_utloggning.jsp`) innehåller endast ett anrop till metoden `invalidate()` i variabeln `session`. I HTML-koden finns även en länk till den privata sidan för test.

```

<%
    session.invalidate();
%>
<html>
<body>
<p>Du är nu utloggad.</p>
<p><a href="session_privat.jsp">Privat sida</a></p>
</body>
</html>

```

### Mer om sessionsvariabler

Sessionshantering sköts alltså med objektet som variabeln `session` refererar till. Utöver metoderna `setAttribute()` och `getAttribute()` finns ett antal metoder till, några som beskrivs här efter. För att ta bort en sessionsvariabel används lämpligen metoden `removeAttribute()` där namnet på variabeln skickas som argument till metod. För att avsluta en session så används metoden `invalidate()`. Metoden `getAttributeNames()` kan användas för att fråga efter en vektor med namnen på alla sessionsvariabler.

Eftersom `setAttribute()` har typen `Object` som andra parameter (d.v.s. för själva värdet) så kan vi skapa enkla klasser som vi lagrar sessionsdata i, d.v.s. JavaBeans.

En viktig detalj är att sessionsdata lagras på webbserver – d.v.s. ju fler samtida besökare, ju mer data lagras. Sessionsdata ska m.a.o. användas med förstånd då webbservrars minne är begränsat.

Vi kan också använda händelsetaggen `<jsp:useBean ...>` (med attributet `scope` satt till `session`) för att lagra JavaBeans som sessionsdata, vilket beskrivs i nästa avsnitt.

## 5.2.5 Använda JavaBeans

JavaBeans är enkla klasser med främst attribut (och motsvarande accessmetoder, d.v.s. set- och get-metoder), d.v.s. används för att lagra data. JavaBeans kan bl.a. användas för att dela på data mellan servlets och/eller JSP (vid t.ex. *forwards* – se nedan). Observera att JavaBeans inte har så mycket mer än namnet gemensamt med EJB (*Enterprise JavaBeans*) – användningsområdena är mycket olika.

Nedan visas först ett enkelt exempel där en instans av JavaBean skapas, och attribut fylls med värden, i en första JSP och ytterligare en JSP där värdena läses i JSP. Det andra exemplet visar hur man kan fylla attribut i en JavaBean direkt med fält i ett HTML-formulär.

### Exempel med enkel JavaBean

I detta exempel skapas en instans av en enkel JavaBean, vars attribut fylls med data, i en JSP och ytterligare en JSP där attribut i JavaBeans läses.

#### JavaBean-klass

Detta är samma klass som användes i exempel med servlets. Klassen innehåller två attribut `anvId` och `namn` (och motsvarande accessmetoder).

```
package bpn;

public class EnkelJavaBean {
    /** Instansvariabler *****/
    private String anvId;
    private String namn;

    /** Accessmetoder för klassens attribut *****/
    public void setAnvId(String id) {
        anvId = id;
    }
    public String getAnvId() {
        return anvId;
    }

    public void setNamn(String n) {
        namn = n;
    }
    public String getNamn() {
        return namn;
    }
} //EnkelJavaBean
```

#### JSP 1

I första JSP (`jsp_javabeans1.jsp`) så används först taggen `<jsp:useBean>` för att skapa en instans av JavaBean eller hitta en redan existerande instans. Attributet `id` används för att namnge referens till instans (motsvaras av en variabel i vanlig Java-kod) – om det redan finns en instans med detta namn så används den existerande instansen. Och attributet `class` används för att ange klass för instans och attributet `scope` för instansens livslängd.

Livslängden för instans i detta exempel är `session`, d.v.s. motsvarar en sessionsvariabel, så att den kan läsas i JSP 2.

Sen används taggen `<jsp:setProperty>` för att sätta (fylla) värden på attribut i JavaBean. Här används attributet `name` för att namnge instans på JavaBean (motsvarar attributet `id` i taggen `<jsp:useBean>`), attributet `property` för att ange vilket attribut i JavaBean som ska sättas och attributet `value` för att ange värde att sätta. I exempel nedan så sätts värden på

attributen `anvId` och `namn` i `JavaBean`. Vi använder namnen på attributen, d.v.s. inga prefix set eller get.

```
<html>
<body>
<h1>JSP 1</h1>
<p>Denna JSP fyller JavaBean med data.</p>
<jsp:useBean id="bona" class="bpn.EnkelJavaBean" scope="session" />
<jsp:setProperty name="bona" property="anvId" value="abc00001" />
<jsp:setProperty name="bona" property="namn" value="Anders Boviac" />
</body>
</html>
```

Om vi inte vill använda JSP-taggar (vilket vi vanligen vill ☺) så skulle koden ovan kunna skrivas om enligt följande kod.

```
<%@ page import="bpn.EnkelJavaBean" %>
<%
    EnkelJavaBean bona = new EnkelJavaBean(); //Skapa instans av JavaBean

    bona.setAnvId("abc00001"); //Sätt värden på attribut i JavaBean
    bona.setNamn("Anders Boviac");

    session.setAttribute("bona", bona); //Skriv JavaBean till sessionsvar.
%>
<html>
<body>
<h1>JSP 1b</h1>
<p>Denna JSP fyller JavaBean med data.</p>
</body>
</html>
```

## JSP 2

I andra JSP (`jsp_javabeans2.jsp`) så används först taggen `<jsp:useBean>` för att hitta en redan existerande instans av `JavaBean` (eller skapa en instans om den inte finns!). Attributen i taggen är de samma som i första JSP. Sen används taggen `<jsp:getProperty>` för att hämta värde för attribut i `JavaBean` – attributet `name` används för att namnge instans av `JavaBean` och attributet `property` för vilket attribut i `JavaBean` som ska hämtas. `<jsp:getProperty>` returnerar värdet på attributet, d.v.s. vi behöver inte använda uttryck eller någon av print-metoderna för utskrift.

```
<html>
<body>
<h1>JSP 2</h1>
<p>Denna JSP läser data från JavaBean.</p>
<jsp:useBean id="bona" class="bpn.EnkelJavaBean" scope="session" />
<p>Användaridentitet är: <jsp:getProperty name="bona" property="anvId" /></p>
<p>Namn är: <jsp:getProperty name="bona" property="namn" /></p>
</body>
</html>
```

Även koden i andra JSP kan skrivas om med scriptlets.

```
<%@ page import="bpn.EnkelJavaBean" %>
<%
    Object obj = null; //Deklarera variabler
    EnkelJavaBean bona = null;
    String strAnvId = null, strNamn = null;
```

```

obj = session.getAttribute("bona"); //Hämta ev. sessionsvariabel

if(obj != null) //Om instans fanns - konvertera och läs värden
{
    bona = (EnkelJavaBean)obj;
    strAnvId = bona.getAnvId();
    strNamn = bona.getNamn();
}
%>
<html>
<body>
<h1>JSP 2b</h1>
<p>Denna JSP läser data från JavaBean.</p>
<p>Användaridentitet är: <%= strAnvId %></p>
<p>Namn är: <%= strNamn %></p>
</body>
</html>

```

(Vi kan dock skippa stegen där vi läser attributen från JavaBean till variabler och använda get-metoderna i uttrycken direkt, om vi vill minska på koden. Men jag är lite obstinat och tycker om att visa alla steg i koden, d.v.s. koden är några rader längre än vad den behöver vara. 😊)

## HTML-formulär och JavaBeans

HTML-formulär (eller bara formulär) kan användas för att sätta värden direkt i en JavaBean. Det viktiga är att fälten i formulären heter (exakt) samma som attributen i JavaBean. Observera att Java, som vanligt, skiljer på gemener och versaler!

### JavaBean-klass

I detta exempel används samma JavaBean som i exempel ovan.

### HTML-fil med formulär

Första JSP innehåller endast ett formulär (d.v.s. skulle kunna vara en vanlig HTML-fil). I formuläret så finns två textfält som har givits (exakt) samma namn som attributen i exemplet JavaBean, d.v.s. anvId och namn, och, givetvis, en skickaknapp.

```

<html>
<body>
<h1>JSP 3</h1>
<p>Denna JSP visar ett formulär som används för att fylla data i JavaBean i nästa JSP.</p>
<form action="jsp_javabeans4.jsp">
  <p>Användaridentitet: <input type="text" name="anvId"></p>
  <p>Namn: <input type="text" name="namn"></p>
  <p><input type="submit" value="Skicka"></p>
</form>
</body>
</html>

```

### JSP som fyller JavaBean med data från formulär

I denna andra JSP (jsp\_javabeans4.jsp) så inleds med att skapa en instans av JavaBean med taggen <jsp:useBean> (enligt exempel ovan). Här ges instans av JavaBean namnet bona2 för att inte komma i konflikt med exempel ovan (eftersom livslängd för båda JavaBeans är session). Sen används taggen <jsp:setProperty> för att fylla instans av JavaBean direkt med data från formuläret. Eftersom namnen på fälten i formulär är de samma som namnen på attribut i JavaBeans så kan vi använda attributet property i taggen med värdet "\*" (asterisk) för att tala om att JSP ska fylla instansen med data från formulär. D.v.s.

vi slipper läsa värden från formulärets fält och sen använda en `<jsp:setProperty>` för varje attribut i `JavaBean`.

För att hämta värdena från attributen i `JavaBean` så måste vi dock använda en `<jsp:getProperty>` för varje attribut vi vill läsa.

```
<jsp:useBean id="bona2" class="bpn.EnkelJavaBean" scope="session" />
<jsp:setProperty name="bona2" property="*" />
<html>
<body>
<h1>JSP 4</h1>
<p>Denna JSP fyller JavaBean med data och läser data direkt.</p>
<p>Användaridentitet är: <jsp:getProperty name="bona2" property="anvId" /></p>
<p>Namn är: <jsp:getProperty name="bona2" property="namn" /></p>
</body>
</html>
```

## 5.2.6 *redirect* och *forward* i JSP

De flesta webbgränssnitt stödjer *redirect*, d.v.s. vidarebefodran. Egentligen är en *redirect* en del av HTTP-protokollet för att tala om för klienter att resurs t.ex. flyttats och vänligen ber klienten att ladda en ny adress.

En *forward* fungerar på ett liknande sätt som *redirect* – skillnaden ligger i att webbserver inte ber klienten ladda en ny adress. Webbserver exekverar istället en annan servlet eller JSP som om det vore aktuell adress. Fördelen med *forward* är att bl.a. synlig då vi använder formulär vi inloggning: besökaren fyller i ett formulär som skickas till servlet för validering, om validering gick bra så returneras en webbsida med positivt besked. Annars returneras formuläret igen med ett negativt besked och besökaren kan försöka fylla i formuläret igen. Allt detta sker utan att klienten är medveten om att olika servlets/JSP exekveras.

För att *redirect* och *forward* ska fungera ordentligt så bör dessa göras **innan** någon data skickats till klienten!

I detta exempel använder vi en kryssruta i ett formulär för att avgöra om *redirect* eller *forward* ska utföras.

### HTML-fil med formulär

Formuläret (i `jsp_redirect_forward1.jsp`) innehåller en kryssruta (`doForward`) och en skickaknapp. Om besökaren bockar för kryssruta så kommer *forward* utföras – annars *redirect*.

```
<html>
<body>
<p></p>
<form action="jsp_redirect_forward2.jsp">
  <p>Utföra forward? <input type="checkbox" name="doForward"></p>
  <p><input type="submit" value="Skicka"></p>
</form>
</body>
</html>
```

### JSP som utför *redirect* eller *forward*

I den andra JSP (`jsp_redirect_forward2.jsp`) kontrolleras om kryssruta är förbockad eller inte. Om den är förbockad så sätts flaggan `blnForward` – en flagga som sen används för att avgöra om taggen `<jsp:forward>` ska exekveras eller inte.

I exemplet visas även hur vi kan bifoga parametrar till sida som *forward* utförs till (men inte *redirect*). För att göra detta så använder vi både öppnande och avslutande tagg för taggen `<jsp:forward>`. I denna tagg placeras en tagg `<jsp:param>` för varje parameter som ska

skickas vidare med *forward*. I exempel har en parameter `doneForward` med värdet "true" (som sträng!) skickats med hjälp av attributen `name` respektive `value`.

```
<%
String strDoForward = null;
boolean blnForward = false;

//Hämta ev. värde på fält - kryssrutor skickas endas om förbockad
strDoForward = request.getParameter("doForward");

//Om kryssruta var förbockad, dvs värde kunde hämtas
if(strDoForward != null)
{ //Kontrollera att fältet innehåller något
  if(strDoForward.length() > 0)
    blnForward = true; //Sätt flagga för forward
}
else //... annars utför en redirect, dvs be webbläsare ladda ny URL
  response.sendRedirect("jsp_redirect_forward3.jsp");
%>
<% if(blnForward) { %>
<jsp:forward page="jsp_redirect_forward3.jsp">
<jsp:param name="doneForward" value="true" />
</jsp:forward>
<% } %>
<html>
<body>
<h1>Utföra redirect eller forward - Sida 2</h1>
<p>Denna webbsida bör aldrig visas...</p>
</body>
</html>
```

Om inga parametrar ska bifogas med *forward* så kan vi använda taggformen utan avslutande tagg, d.v.s. vi avslutar taggen `<jsp:forward>` med ett snedstreck (/) som i exempel nedan.

```
<jsp:forward page="jsp_redirect_forward3.jsp" />
```

## JSP som besökare skickas till

Den sista JSP (`jsp_redirect_forward3.jsp`) är webbsida som besökare skickas till när exekvering i andra JSP utförs. Om det är *redirect* eller *forward* kan bl.a. avgöras med hjälp av URL – om URL innehåller filnamn för aktuell JSP, d.v.s. filnamn innehåller en 3:a, så är det *redirect*. Men om filnamnet innehåller en 2:a så är det *forward*.

Här visas hur vi kan läsa eventuella parametrar som skickats vid *forward* i andra JSP. Att läsa parametrar fungerar som att läsa fält i formulär. Parametrar skickas dock alltid med HTTP POST.

```
<%
String strDoneForward = null, strMeddelande = null;

//Hämta ev. parameter satt vid forward
strDoneForward = request.getParameter("doneForward");

//Om parametervärde finns - kontrollera att det är värdet som sattes...
if(strDoneForward != null)
{
  if(strDoneForward.equals("true"))
    strMeddelande = "forward utförd";
}
else //... annars har redirect utförts
  strMeddelande = "redirect utförd";
%>
<html>
```

```
<body>
<h1>Utföra redirect eller forward - Sida 3</h1>
<p>Du har blivit skickad till denna webbsida med redirect eller forward.
  Vilket avgörs om filnamnet innehåller en 3:a resp. inte.</p>
<p><%= strMeddelande %></p>
</body>
</html>
```



## 6 Servlets och JSP i samarbete

Servlets och JSP var för sig har stora svagheter (eller nackdelar). Men genom att kombinera dessa teknologier så får man ett mycket flexibelt (och ”nifftigt”<sup>35</sup>) sätt att utveckla webbapplikationer.

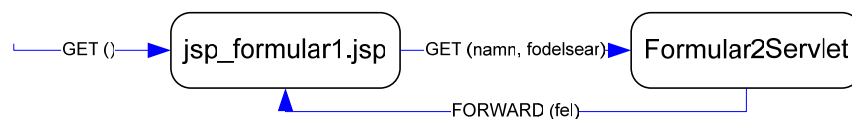
### 6.1 HTML-formulär i JSP och servlet

Detta exempel bygger vidare på tidigare exempel med HTML-fil och servlet. Exemplet visar hur man använder ett ”vanligt” HTML-formulär i en JSP-sida som skickas till en servlet för verifiering. Om verifiering misslyckas så skickas besökare tillbaka till JSP-sida med besked om vad som är fel, annars visas besked om att allt är OK.

#### 6.1.1 Filer i exempel

I detta exempel används en JSP-fil och en servlet. Först laddas JSP-filen med ett formulär, vars innehåll skickas till servlet för ”verifiering”. Verifieringen består i att kontrollera att fälten innehöll något och att fält med födelseår innehåller ett tal (d.v.s. kan konverteras). Exemplet visar även hur vi kan hantera fel i verifiering:

- om fält är tomma så skrivs ett felmeddelande ut av servlet
- eller som när konvertering går fel där felmeddelande läggs till i HTTP-begäran (*request* – attributet *fel*) och besökare skickas tillbaka till JSP med en *forward*.



För enkelhetens skull (d.v.s. för att slippa problem med sökvägar) så lägger vi även till ett servlet-alias så att vi slipper blanda in *servlet* i URL (se slutet på detta avsnitt).

### JSP med formulär

I grund och botten innehåller JSP-filen (*jsp\_formular1.jsp*) ett HTML-formulär med två fält för namn (*namn*) och födelseår (*fodelsear*).

Här börjas med att eventuellt fel från servlet hämtas (i form av ett attribut som lagts till i HTTP-begäran – *fel*). Om det finns ett felmeddelande så har en forward från servlet gjorts. Om så är fallet så kan vi konvertera felmeddelandet från typen *Object* till typen *String* samt hämta eventuella värden som besökare fyllt i formuläret förra gången det laddades.

I HTML-formuläret använder vi attributen *value* för att fylla i eventuella värden som besökare fyllt i förra gången formuläret laddades. Variablerna *strNamn* och *strFodelsear* tilldelas en tom sträng istället för *null* om formuläret inte tidigare fyllts i. Om vi inte gör det så kommer text ”*null*” skrivas ut i textfälten.

<sup>35</sup> Jag har utvecklat applikationer i PHP, ASP, ASP.NET och bara servlets. Dessa teknologier fungerar och är användbara, men kombinationen av servlets och JSP är det bästa sätt jag sett och är värt att lägga ner tid på att lära sig. Sun har **verkligt** tänkt till när dom utvecklade dessa teknologier (mer än Microsoft och ASP.NET).

```

<%
    Object objFel = request.getAttribute("fel");
    String strMeddelande = "", strNamn = "", strFodelsear = "";
    if(objFel != null) {
        strMeddelande = (String)objFel;
        strNamn = request.getParameter("namn");
        strFodelsear = request.getParameter("fodelsear");
    }
%>
<html>
<body>
<p>Fyll i formulär och klicka på Skicka-knappen.</p>
<p><%= strMeddelande %></p>
<form action="Formular2">
    <p>Namn: <input type="text" name="namn" value="<%= strNamn %>"></p>
    <p>Födelseår: <input type="text" name="fodelsear"
        value="<%= strFodelsear %>"></p>
    <p><input type="submit" value="Skicka"></p>
</form>
</body>
</html>

```

## Servlet som validerar formulär

Till att börja med så placeras servlet-klass i ett paket (bpn i kod nedan – ersätt gärna med egen användaridentitet i nätverk) för att slippa en del problem. Därefter importerar de ”vanliga” klasserna och paketen samt klassen Calendar för att kunna fråga efter aktuellt år. I klasser brukar jag sen deklarerar konstanter på värden som kan tänkas ändras (bl.a. för att lättare hitta dessa värden när de ska ändras).

I metoden doGet() börjar vi med att hämta innehållet i (eventuellt) HTML-formulärs fält samt testa omfälten existerar. Omfälten finns så skapas en instans av klassen Calendar för att hämta aktuellt år – ett år som vi ska använda för att räkna ut hur gammal besökare fyller i år. Men omfälten inte finns så sätts en flagga (blnFel) och en variabel med meddelande om vad som är fel.

Om felflaggan (blnFel) inte är satt så görs även ett försök att konvertera besökarens födelseår till ett heltal. Om denna konvertering misslyckas så skickas besökare tillbaka till formuläret (JSP-sidan) med ett meddelande om att det inte gick. Om besökare inte skickats till formulär (och därmed exekvering av servlet avbrutits) så skrivs antingen besökarens namn och ålder ut – om inget fel – eller eventuellt felmeddelande.

```

package bpn;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.Calendar;

public class Formular2Servlet extends HttpServlet {
    //Konstant med URL till formulär som skickas till denna servlet
    private static String cstrUrl = "jsp_formular1.jsp";

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
        String strNamn = null, strFodelsear = null, strMeddelande = null;
        int intAktuelltAr = 0, intFodelsear = 0, intAlder = 0;
        boolean blnFel = false; //Flagga för om fel uppstått

        strNamn = request.getParameter("namn"); //Hämta innehåll i formulärs fält
        strFodelsear = request.getParameter("fodelsear");

        //Om formulär innehåller fält - hämta aktuellt år ...
        if((strNamn != null) && (strFodelsear != null)) {
            intAktuelltAr = Calendar.getInstance().get(Calendar.YEAR);
        }
        else { //... annars sätt flagga med fel och felmeddelande
            blnFel = true; //Sätt flagga att något är fel
            strMeddelande = "FEL: Formulär är felaktigt! Meddela administratör.";
        }
    }
}

```

```

    }

    response.setContentType("text/html"); //Ange typ av data som returneras
    PrintWriter out = response.getWriter(); //Hämta utström (out stream)

    //Om inga fel - beräkna ålder
    if(!blnFel) {
        try {
            intFodlsear = Integer.parseInt(strFodlsear); //Konvertera till tal
        }
        catch(NumberFormatException nfe) {
            strMeddelande = "FEL: Födelseår måste vara ett tal!";
            RequestDispatcher rd = request.getRequestDispatcher(cstrUrl);
            request.setAttribute("fel", strMeddelande);
            rd.forward(request, response);
        }

        intAlder = intAktuelltAr - intFodlsear; //Beräkna ålder
    }

    out.print("<html><body>"); //Påbörja HTML-dokument

    if(!blnFel)
        out.print("<p>Ditt namn är " + strNamn + " och du är " + intAlder
            + " år gammal.</p>");
    else
        out.print("<p>" + strMeddelande + "</p>");

    out.print("</body></html>"); //Avsluta HTML-dokument

    out.close(); //Stäng utström
} //doGet()
} //class Formular2Servlet

```

## Konfiguration av servlet-alias

För att slippa problem med sökvägar så lägger vi till följande i webbapplikations konfigurationsfil (web.xml i Resin). Exemplet nedan visar även hur vi kan använda attribut i taggen `<servlet-mapping>` istället för inneslutna taggar (`<url-pattern>` och `<servlet-name>`). Här ges namnet `Formular2` till klassen `bpn.Formular2Servlet` samt URL `/Formular2` för klassen.

```

<servlet>
  <servlet-name>Formular2</servlet-name>
  <servlet-class>bpn.Formular2Servlet</servlet-class>
</servlet>

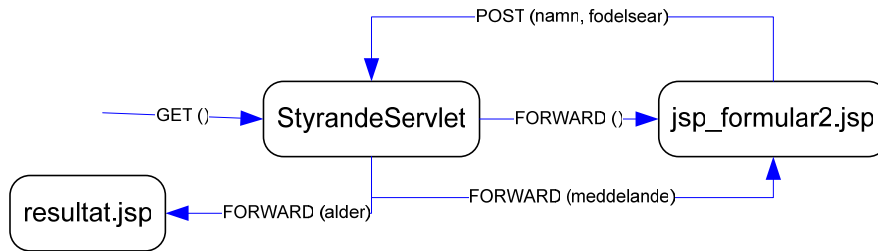
<servlet-mapping url-pattern="/Formular2" servlet-name="Formular2" />

```

## 6.2 Servlet som styr exekvering

Detta exempel visar på det idealiska sättet att använda servlets och JSP i kombination. Men innan man sätter igång att koda bör man dock tänka till först. Lämpligen skapas en skiss (som nedan) för att visa tänkta navigeringsvägar.

I detta exempel laddas servlet (`StyrandeServlet`) för att visa formulär (med HTTP-GET). Eftersom servlet endast ska styra exekveringen så skickas exekvering vidare med en *forward* till JSP med formulär (`jsp_formular2.jsp`). När besökare fyllt i formuläret så skickas det tillbaka till servlet med HTTP-POST för verifiering av data i formulär. Om verifiering lyckades så skickas exekvering vidare till JSP för att visa resultatet (`resultat.jsp`).



Skisser enligt ovan kan även kompletteras med möjliga (d.v.s. inte bara tänkta) navigeringsvägar så att de även täcker in eventuella *redirect* eller *forward* vid fel. I skiss ovan så visas en *forward* där attribut (meddelande) lagts till för att kunna skicka meddelande till JSP. Detta visar hur vi hanterar fallet då formulär är felaktigt ifyllt, d.v.s. att exekvering skickas vidare till JSP igen med ett felmeddelande.

## 6.2.1 Filer i exempel

I detta exempel används som sagt en servlet och två JSP-filer.

### Servlet

Som i tidigare exempel så placeras servlet i ett paket (som ni gärna får ändra namn på) och paket/klasser importeras. Sen deklarerar konstanter för värden som kan ändras – namn på JSP-filer här.

I `doGet()` skickas besökare direkt till formulär, ett formulär som använder HTTP-POST för att skicka data i formulär till servlet. Därför sker det mesta av exekveringen i `doPost()`. Här hämtas eventuellt namn och födelseår samt kontrolleras att fälten fanns (d.v.s. inte är `null`). Om fälten inte fanns så skickas besökare tillbaka till formulär med en *forward* genom att anropa metoden `visaFormular()` med meddelande till besökare. Annars kontrolleras att fälten innehöll något. Om fälten innehöll något så anropas metoden `visaResultat()`, annars skickas besökare tillbaka till formulär med metoden `visaFormular()`.

I `visaResultat()` så försöker vi konvertera födelseåret till ett tal så att vi kan räkna ut besökarens ålder. Om detta misslyckas så skickas besökare tillbaka till formuläret (återigen med metoden `visaFormular()`). Annars så hämtas aktuellt år och besökarens ålder räknas ut. Ålder läggs till som attribut i HTTP-begäran så att det kan läsas i JSP som ska visa resultatet. Sist skickas exekvering vidare till JSP med en *forward*.

```

package bpn;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.Calendar;

public class StyrandeServlet extends HttpServlet {
    //Konstant med URL till formulär som skickas till denna servlet
    private static String cstrUrlFormular = "jsp_formular2.jsp";
    private static String cstrUrlResultat = "resultat.jsp";

    //Metod som anropas om servlet begärs med HTTP GET - bör bara vara första ggn
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
        visaFormular(request, response, null); //Skicka vidare exekvering
    } //doGet()

    //Metod som anropas om servlet begärs med HTTP POST - bör vara resten av ggr
    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
        String strNamn = null, strFodelsear = null;
  
```

```

//Hämta data från formulär
strNamn = request.getParameter("namn");
strFodelsear = request.getParameter("fodelsear");

//Om formulär inte innehåller fält - meddela besökare
if((strNamn == null) || (strFodelsear == null))
    visaFormular(request, response,
        "FEL: Formulär är felaktigt! Meddela administratör.");
else //... annars testa om fält innehöll något
    if((strNamn.length() > 0) && (strFodelsear.length() > 0))
        visaResultat(request, response, strNamn, strFodelsear);
    else
        visaFormular(request, response,
            "FEL: Både namn och födelseår måste fyllas i!");
} //doPost()

/** Eegna metoder *****/
//Anropas för att skicka besökare till formulär med ev. meddelande
public void visaFormular(HttpServletRequest request,
    HttpServletResponse response, String meddelande)
    throws IOException, ServletException {
    RequestDispatcher rd = request.getRequestDispatcher(cstrUrlFormular);
    if(meddelande != null) //Om inte är null - skicka vidare till JSP
        request.setAttribute("meddelande", meddelande);
    rd.forward(request, response);
} //visaFormular()

//Anropas om formulär ifyllt
public void visaResultat(HttpServletRequest request,
    HttpServletResponse response, String namn, String fodelsear)
    throws IOException, ServletException {
    int intFodelsear = 0, intAktuelltAr = 0, intAlder = 0;

    try {
        intFodelsear = Integer.parseInt(fodelsear); //Konvertera till tal
    }
    catch(NumberFormatException nfe) {
        visaFormular(request, response, "FEL: Födelseår måste vara ett tal!");
    }

    intAktuelltAr = Calendar.getInstance().get(Calendar.YEAR);
    intAlder = intAktuelltAr - intFodelsear;

    RequestDispatcher rd = request.getRequestDispatcher(cstrUrlResultat);
    request.setAttribute("alder", new Integer(intAlder));
    rd.forward(request, response);
} //visaResultat()
} //class StyrandeServlet

```

## JSP-fil med formulär

Denna JSP fungerar som i förra exemplet. Läs eventuellt felmeddelande – om närvarande, konvertera till sträng samt hämta även tidigare ifyllt namn och födelseår.

```

<% //Hämta eventuellt meddelande från servlet
Object objMeddelande = request.getAttribute("meddelande");
String strMeddelande = "", strNamn = "", strFodelsear = "";

//Om meddelande inte skickats från servlet
if(objMeddelande != null) {
    //Konvertera attribut från servlet till en sträng
    strMeddelande = (String)objMeddelande;

    //Hämta eventuellt namn och födelseår från felaktigt formulär
    if((strNamn = request.getParameter("namn")) == null)
        strNamn = "";
    if((strFodelsear = request.getParameter("fodelsear")) == null)
        strFodelsear = "";
}
%>
<html>
<body>
<p>Fyll i formulär och klicka på Skicka-knappen.</p>
<form action="Styrande.html" method="post">

```

```
<p>Namn: <input type="text" name="namn" value="<%= strNamn %>"></p>
<p>Födelseår: <input type="text" name="fodelsear"
  value="<%= strFodelsear %>"></p>
<p><input type="submit" value="Skicka"> <%= strMeddelande %></p>
</form>
</body>
</html>
```

## JSP-fil för att visa resultat

Eftersom vi redan lyckats konverterat besökarens födelseår i servlet så har besökarens ålder bifogats som del av HTTP-begäran så att denna JSP-sida blir så enkel som möjlig. D.v.s. här behöver vi endast hämta besökarens namn och ålder samt skriva ut informationen i JSP.

```
<%
  //Hämta namn från formulär
  String strNamn = request.getParameter("namn");

  //Hämta ålder som lagts till som attribut i begäran i servlet
  Object objAlder = request.getAttribute("alder");
%>
<html>
<body>
<p>Ditt namn är <%= strNamn %> och du är <%= objAlder %> år gammal.</p>
</body>
</html>
```

## Konfiguration av servlet-alias

Detta exempel visar även hur man kan lura besökare till att tro att man använder vanliga HTML-filer (Styrande.html i detta exempel) fast det är en servlet (och JSP) som hanterar begäran.

```
<servlet>
  <servlet-name>Styrande</servlet-name>
  <servlet-class>bpn.StyrandeServlet</servlet-class>
</servlet>

<servlet-mapping>
  <url-pattern>/Styrande.html</url-pattern>
  <servlet-name>Styrande</servlet-name>
</servlet-mapping>
```

## 7 Remote Method Invocation (RMI)

RMI är ett protokoll för kommunikation mellan applikationer (klienter) och objekt (servrar) i olika processer. Objekts processer kan finnas på samma dator som klient eller på en annan dator. Suns implementation av RMI kallas **Java RMI**.

För att skapa en instans (d.v.s. det distribuerade objektet) med ”bara” RMI och göra det tillgängligt för klienter så måste vi registrera objektet. Registreringen sker i Javas RMI-register (*RMI registry*).

(Detta kapitel är bara till för att visa hur man kan göra distribuerade objekt utan att använda en J2EE-server. Men för att utveckla komponenter så använder vi oftast EJB (*Enterprise JavaBeans*) – RMI används då, ”osynligt”, bara för kommunikation mellan klient och EJB. D.v.s. vi bör veta vad RMI är, men vi använder det utan att märka det.)

I exempel nedan skapas ett distribuerat objekt som returnerar en sträng.

---

### 7.1 Introduktion

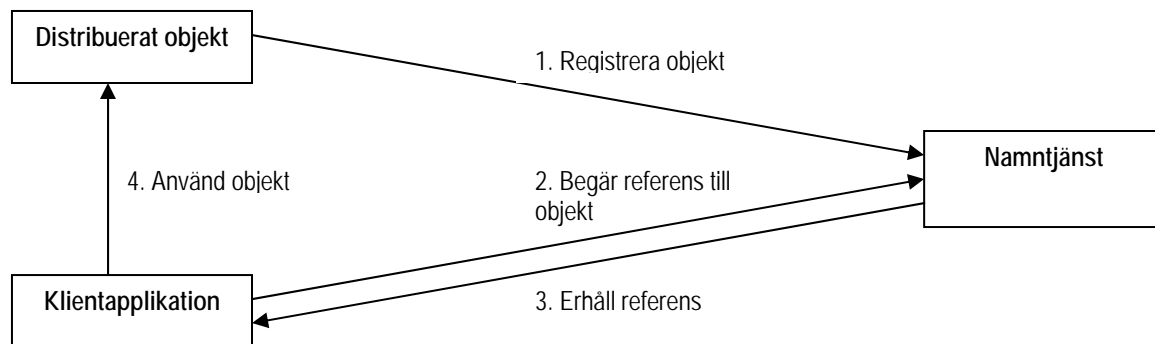
Java RMI är ett protokoll för hur kommunikation mellan en klient och ett distribuerat objekt sker. I protokollet ingår även (m.h.a. JNDI) hur man ”finner” ett objekt. Med finner menas hur man erhåller en referens till ett existerande objekt eller hur en ny instans av objekt skapas och referens till det erhålls.

Själva kommunikationen mellan klient och objekt kan (idag) ske med två (ytterligare) protokoll: *Java Remote Method Protocol* (JRMP) eller RMI-IIOP (se avsnittet *RMI och Internet Inter-ORB Protocol* (IIOP) nedan).

Java RMI bygger på tre saker

- Ett distribuerat objekt.
- En namnserver att registrera objektet i och som sen klient kan fråga efter referens till objekt.
- En klient.

Principen bygger på att ett distribuerat objekt skapas och registreras i en namntjänst (1 i bild nedan). Klienter använder sen namntjänsten för att fråga efter en referens till det distribuerade objektet (2). När väl klienten erhållit en referens till det distribuerade objektet (3) kan klienten anropa metoder i objektet som om det vore ett lokalt objekt (4). D.v.s. det speciella med ett distribuerat objekt är alltså hur man erhåller en referens till objektet. (Med ”vanliga” lokala objekt så hade vi t.ex. använt det reserverade ordet `new` för att skapa objektet.) När vi väl har en referens till ett distribuerat objekt så döljer RMI hur själva kommunikationen med objektet sker. Den enda skillnaden vi kan märka är att svarstiden (tiden det tar från att metod anropas till ett svar erhålls) är längre om det distribuerade objektet finns på en annan dator i ett nätverk.



I exempel nedan visas hur vi kan skapa ett distribuerat objekt samt hur vi kan använda det från en klient. För detta exempel så behöver **inte** använda flera datorer, d.v.s. objektet och klienten kan finnas på samma dator.

**Observera** att exempel nedan kräver att `%JAVA_HOME%\bin`<sup>36</sup> finns i sökvägen (PATH) eller att vi lägger till sökvägen före varje kommando (applikation)!

## 7.2 Skapa och publicera distribuerade objekt

Att skapa ett distribuerat objekt och publicera det för klienter består av 5 steg. I nedanstående lista anges namn på gränssnitt/klass i exempel inom parentes.

1. Definiera remote-gränssnitt (Hello).
2. Implementera remote-gränssnitt (HelloServer).
3. Generera stub (och skeleton) (HelloServer\_Stub).
4. Starta RMI-registret.
5. Registrera instans av distribuerat objekt och bind till ett namn i RMI-registret.

När en instans av vårt distribuerade objekt är registrerat i RMI-registret så kan vi skapa en klient som använder sig av objektet.

### 7.2.1 Definiera remote-gränssnitt (Hello)

Ett distribuerat objekts remote-gränssnitt innehåller de publika metoderna som objektet svarar på. Därför brukar namnet på gränssnittet reflektera objektets funktion (t.ex. Person eller Hus). Gränssnittet måste ärvas från (utöka, eng. *extend*) gränssnittet `java.rmi.Remote`<sup>37</sup> för att tala om att det är ett distribuerat objekt. Och alla metoder i vårt remote-gränssnitt måste deklarerat att dom (minst) kan generera (*throw*) undantaget (*exception*) `java.rmi.RemoteException`.

Nedan visas definitionen av vårt remote-gränssnitt – vi börjar med att importera allt i paketet `java.rmi` (för tillgång till gränssnittet `Remote` och undantaget `RemoteException`).

```

package hello;

import java.rmi.*;

public interface Hello extends Remote {
    public String sayHello() throws RemoteException;
}
  
```

Kompilera gränssnittet för att skapa CLASS-filen.

<sup>36</sup> Om J2EE v.1.4 installerats i standardmapp så är `%JAVA_HOME%` antagligen `C:\Sun\AppServer\jdk`.

<sup>37</sup> Gränssnittet `java.rmi.Remote` innehåller inga metoder, d.v.s. vi behöver inte implementera några metoder (som definierats i gränssnittet) i vår klass som sen ska implementera vårt remote-gränssnitt.



## 7.2.2 Implementera remote-gränssnitt (HelloServer)

Nästa steg är att skapa en klass som implementerar vårt remote-gränssnitt. Klassen måste även ärva från klassen `java.rmi.server.UnicastRemoteObject`. Klassen `UnicastRemoteObject` är en klass som kapslar in mycket av koden som behövs för att göra metदानrop via t.ex. nätverk.

I vår server (`HelloServer`) – klassen som implementerar vårt remote-gränssnitt – behöver vi tillgång till undantaget `RemoteException` samt klassen `UnicastRemoteObject` och importerar därför dessa (från paketen `java.rmi` resp. `java.rmi.server`). Vi importerar även `java.util.Date` för att få tillgång till datumklassen.

Därefter skapar vi en konstruktor för klassen, som endast anropar superklassens konstruktor, och implementerar vår metod, `sayHello()`, från vårt remote-gränssnitt.

```
package hello;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.Date;

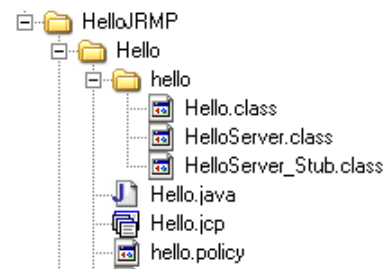
public class HelloServer extends UnicastRemoteObject implements Hello {
    public HelloServer() throws RemoteException {
        super();
    }

    public String sayHello() throws RemoteException {
        return "Hello World, the current system time is " + new Date();
    }
} //class HelloServer
```

Kompilera klassen för att skapa CLASS-filen.

## 7.2.3 Generera stub (och skeleton) (HelloServer\_Stub)

För att generera stub-klass så använder vi verktyget `rmic` (i mappen `%JAVA_HOME%\bin`). För att kunna generera stub så måste vi stå i (aktuell mapp vara) mappen där klassen eller paketet finns. Eftersom exempel finns i paketet `hello` så måste vi alltså stå i en mapp där vi har en undermapp som heter `hello` och som innehåller filen `Hello.class`. I bilden till höger måste vi alltså stå i mappen som heter `Hello`.



Byt därför aktuell mapp till ”rätt” mapp och skriv följande i kommandotolken:

```
rmic -v1.2 hello>HelloServer
```

Detta genererar en fil som heter `HelloServer_Stub.class` i samma mapp som `Hello.class`. Stub-filen behövs för att starta RMI-registret samt kompilera och exekvera klienten.

## 7.2.4 Starta RMI-register

Innan vi kan starta RMI-registret så bör vi skapa en policyfil som ger rättigheter till vårt distribuerade objekt. För enkelhetens skull så skapar vi en policyfil som ger fullständiga rättigheter. **Observera** att det **inte** bör ske i en produktionsmiljö!

Filen, `hello.policy`, ska innehålla följande:

```
grant {
    permission java.security.AllPermission;
};
```

Filen behöver inte kompileras (och om vi vill så kan vi använda verktyget `policytool`, i `%JAVA_HOME%\bin`<sup>38</sup>, för att skapa filen).

Sen är det dags att starta RMI-registret, vilket görs med filen `rmiregistry.exe` som finns i `%JAVA_HOME%\bin`. För att kunna göra detta måste vi ha tillgång till policyfilen samt `stub`. Återigen så bör vi stå i (aktuell mapp vara) mappen som innehåller klassen eller paketet. Och eftersom vår klass finns i paketet `hello` så står vi i samma mapp som vi gjorde när vi genererade `stub` (se bild i förra avsnittet). Byt aktuell mapp till ”rätt” mapp och starta RMI-registret genom att skriva följande i kommandotolken:

```
rmiregistry -J-Djava.security.policy=hello.policy
```



Observera att vi **inte** får någon bekräftelse på att start av RMI-registret lyckades. Vi får endast meddelande om något gick fel (t.ex. att klasser inte kan hittas).

## 7.2.5 Registrera en instans av objekt och binda till ett namn

För att registrera och binda vårt distribuerade objekt i RMI-registret så skapar vi en körbar klass (en klass med en `main`-metod) och som heter `HelloRegister` med nedanstående kod. I klassen skapas en instans av objektet och metoden `rebind()` används för att binda objektet till ett namn i RMI-registret. Första parametern till metoden är kontext som namn ska bindas till.

```
import java.rmi.*;
import hello.*;

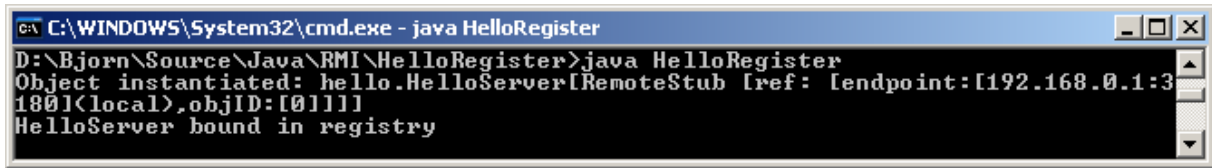
public class HelloRegister {
    public static void main(String args[]) {
        try {
            HelloServer obj = new HelloServer();
            System.out.println("Object instantiated: " + obj);
            Naming.rebind("/HelloServer", obj);
            System.out.println("HelloServer bound in registry");
        } //try
        catch(Exception e) {
            System.out.println(e);
        } //catch
    } //main()
} //class HelloRegister
```

Kompilera klassen för att skapa CLASS-filen. För att kunna kompilera klassen så behöver vi bl.a. tillgång till CLASS-filer för remote-gränssnittet. Detta kan lösas genom att placera alla filer i ett och samma ”projekt” (om t.ex. JCreator används) eller att filer kopieras. (Glöm **inte**

<sup>38</sup> `%JAVA_HOME%` är den mapp som J2SE SDK installerats, t.ex. `C:\Program\Java\jdk1.5.0_01` (eller `C:\Program\Sun\AppServer\jdk` om J2EE SDK installerats).

att om klasser finns i paket så måste vi ha mappar som motsvarar paketen!) Öppna ytterligare ett kommandotolksfönster och kör klassen genom att skriva följande i kommandotolken:

```
java HelloRegister
```



```
C:\WINDOWS\System32\cmd.exe - java HelloRegister
D:\Bjorn\Source\Java\RMI\HelloRegister>java HelloRegister
Object instantiated: hello.HelloServer[RemoteStub [ref: [endpoint:[192.168.0.1:3180]](local).objID:[0]]]
HelloServer bound in registry
```

Om allt gick bra så bör vi ha ett resultat liknande det i bilden ovan.

## 7.2.6 Testköra distribuerat objekt med en klient (HelloClient)

Denna enkla applikation frågar namnserver om en referens till det distribuerade objektet och anropar dess metod samt skriver ut returvärdet.

I koden skapas först en instans av `RMISecurityManager` för att hantera säkerhetsaspekter. Därefter används JNDI för att leta upp objektet som bundits till RMI-registret. Därefter anropas metod i objektet som om det vore ett "vanligt" objekt.

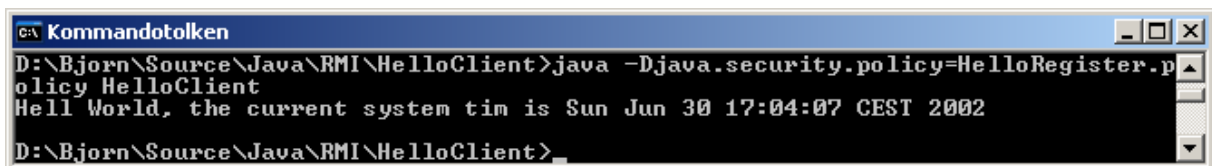
```
import java.rmi.*;
import hello.*;

public class HelloClient {
    public static void main(String args[]) {
        if(System.getSecurityManager() == null)
            System.setSecurityManager(new RMISecurityManager());

        try {
            Hello obj = (Hello)Naming.lookup("/HelloServer");
            String message = obj.sayHello();
            System.out.println(message);
        } //try
        catch(Exception e) {
            System.out.println("HelloClient exception: " + e);
        } //catch
    } //main()
} //class HelloClient
```

Kompilera klassen för att skapa CLASS-filen. Återigen behöver vi bl.a. tillgång till CLASS-fil för det distribuerade objektet. Vi behöver även tillgång till policyfilen och kör programmet genom att skriva följande i kommandotolken:

```
java -Djava.security.policy=hello.policy HelloClient
```



```
Kommandotolken
D:\Bjorn\Source\Java\RMI\HelloClient>java -Djava.security.policy=HelloRegister.p
olicy HelloClient
Hell World, the current system time is Sun Jun 30 17:04:07 CEST 2002
D:\Bjorn\Source\Java\RMI\HelloClient>
```

Om allt gick bra så ska resultatet se ut något som i bilden ovan.

## 7.3 Dynamiskt skapade av objekt

Ovanstående exempel kräver att ett objekt (instans av klass) skapas och att den registreras i RMI-registret. Det är även möjligt att skapa instanser dynamiskt, d.v.s. först när klienten behöver objektet. Detta behandlas inte här...

---

## 7.4 RMI och Internet Inter-ORB Protocol (IIOP)

RMI-IIOP tillåter RMI över CORBAs IIOP istället för Java RMI standardprotokoll (JRMP). Detta introducerar ett antal restriktioner på RMI, bl.a.:

- Undvik konstanter eller använd endast konstanter av enkla typer som kan verifieras vid kompilering.
- Använd inte namn som kan hamna i konflikt med de genererade från IDL-kompilator.
- Ärv inte så att ett gränssnitt ärver samma metod från två superklasser.
- Namn som endast skiljer i gemener/versaler kanske inte fungerar.
- Det går inte att använda DGC (*Distributed Garbage Collector*). Referensen till objektet måste förstöras för att objektet ska förstöras.

RMI-IIOP är användbart för att göra applikationer mer tillgängliga (inte bara i Java-applikationer) men behandlas inte ytterligare i denna sammanfattning.

---

## 7.5 RMI och EJB

När vi använder EJB så registreras en EJB (själva komponenten och inte instanser av komponenten!) i en EJB-server. Denna server innehåller ofta en namnserver (motsvarande RMI-registret) samt hanterar livscykel för (skapande och förstörande av) instanser av EJB. D.v.s. vi designar och registrerar en EJB i EJB-server. Klienter kan sen fråga EJB-server (via namnserver) om en instans av EJB, d.v.s. instanser skapas dynamiskt efter behov.

Utöver remote-gränssnittet måste vi dock skapa ytterligare ett gränssnitt (kallat *home*-gränssnitt) så att EJB-server kan hantera EJB-instansers livscykel. Vi behöver dock inte skapa *stub*-objekt (eller *skeleton*) – detta görs av EJB-server. Vi behöver inte heller använda policyfiler för SecurityManager som i exempel ovan.

Det finns alltså en del skillnader mellan distribuerade objekt med bara Java RMI och EJB. Del 2 av denna sammanfattning är uteslutande om hur man skapar EJB.

## 8 Extensible Markup Language (XML)

Ett uppmärkningspråk (*markup language*) används för att märka upp olika element i ett dokument. Med ”märka upp” menas var element börjar och slutar samt eventuella egenskaper/attribut som elementet har. Ett typiskt exempel på uppmärkningspråk är HTML och XML. XML är **inte** en uppföljare till HTML.<sup>39</sup> XML heter *extensible* för att tanken är att antalet typer av element ska kunna utökas (eller snarare skapas) efter behov.

---

### 8.1 Ett XML-dokuments struktur

En viktig detalj med XML (jämfört med HTML) är att ett XML-dokument måste vara riktigt strukturerat. D.v.s. tolkar för XML är inte lika förlåtande som webbläsare (som tolkar HTML-kod).

Ett XML-dokument består av ett antal olika delar, bl.a. taggar, attribut och data. Utöver detta kan en DTD (*Document Type Definition*) kopplas till XML-dokument (för att kunna verifiera dokumentet) och stilmallar appliceras för att ändra presentation av data i dokument.

#### 8.1.1 Taggar

Elementen i ett XML-dokument märks upp med s.k. taggar (*tag*) – ett namn inneslutet i hakparentser (< och >), t.ex. <person>. Taggar har en öppnande tagg och en avslutande tagg. Den avslutande taggen ser likadan ut som den öppnande med skillnaden att namnet på taggen föregås av ett snedstreck (/), t.ex. </person>.

Namnet i den öppnande och avslutande taggen måste ha samma skiftläge då gemener och versaler tolkas som olika tecken i XML.

Om ett element inte innehåller någon data, d.v.s. inte har någon text mellan öppnande och avslutande tagg, så kan avslutande tagg utelämnas. Istället så skriver man ett snedstreck sist i den öppnande (och enda) taggen, t.ex. <person namn="Björn" />.

#### 8.1.2 Attribut

Ett attribut är ett namn-värde par som anges i den öppnande taggen, t.ex. <person namn="Björn" />. Attributets namn får inte innehålla några mellanslag och värdet för attributet måste inneslutas i dubbla citattecken (").

#### 8.1.3 Document Type Definition

XML-dokument kan beskrivas i en DTD (Document Type Definition) och då oftast i en separat fil (för att kunna användas med fler XML-dokument). DTD:er kan användas för att verifiera XML-datas riktighet.

#### 8.1.4 Stilmallar

För att kunna presentera data i ett XML-dokument kan man applicera en XSL-stilmall (eXtensible Stylesheet Language). Med hjälp av stilmallen kan man t.ex. transformera XML-data till ett HTML-dokument för visning i webbläsare.

---

<sup>39</sup> Uppföljaren till HTML heter XHTML.

## 8.2 Användningsområden för XML-dokument

Fördelen med XML är att det är en öppen standard och inte beroende av någon speciell<sup>40</sup> form av mjukvara (d.v.s. som måste köpas) för att tolkas. Detta gör att det är ett ypperligt format för att distribuera data mellan olika mjukvaror, eller rent av mellan olika organisationer.

Eftersom definitionen av ett XML-dokument finns i DTD så kan man bifoga DTD:n (eller göra den tillgänglig på Internet) för mottagaren så att denna ska kunna validera data i dokumentet eller skapa ett eget dokument.

XML används även i konfigurationsfiler till mjukvaror. I EJB använder vi även XML för att beskriva våra komponenter och eventuella inställningar som komponenten bör använda i J2EE-servern.

---

## 8.3 Olika typer av XML-tolkar

En XML-tolk (*parser*) kan användas för att läsa ett XML-dokument och extrahera data i dokumentet. Det finns idag främst två typer av tolkar: SAX (*Simple API for XML*) och DOM (*Document Object Model*). En tolk av typen SAX är oftast minnessnål då den läser in endast delar av dokumentet åt gången och använder händelser för att meddela att ett element påträffats. DOM-tolkar läser in hela dokumentet på en gång och skapar objekt för de olika taggarna.

---

## 8.4 Nackdelen med XML

Eftersom taggar måste omgärda all data så kan ett XML-dokument ganska snabbt växa i storlek. Ibland kan data i ett XML-dokument ta upp mer plats än motsvarande data i en relationsdatabas. Men ibland kan detta pris vara värt att betala, speciellt för att erhålla ett dataformat som gör att olika mjukvaror kan kommunicera (t.ex. ett Java-program i UNIX och ett COM-program i Windows).

---

<sup>40</sup> Ofta används s.k. parsers, funktioner eller klasser som skapats för att tolka XML. Dessa parsers är dock implementerade i många språk som används idag, t.ex. Java och VB.NET.

## 9 Litteraturförteckning

- Allamaraju, S., et al, **Professional Java Server Programming – J2EE Edition**, Wrox Press, 2000. ISBN: 1-861004-65-6.
- Bergsten, Hans, **JavaServer Pages**, O'Reilly, 2001. ISBN: 1-56592-746-X.
- Horton, Ivor, **Beginning Java 2**, Wrox Press, 1999. ISBN: 1-861002-23-8.
- Hunter, Jason & Willian Crawford, **Java Servlet Programming, 2<sup>nd</sup> Ed.**, O'Reilly, 2001. ISBN: 0-596-00040-5.
- Orfali, Robert, et al, **Instant CORBA**, Wiley Computer Publishing, 1997. ISBN: 0-471-18333-4.
- Sasbury, Stephen & Scott R. Weiner, **Developing Enterprise Java Applications, 2<sup>nd</sup> Edition**, Wiley Computer Publishing, 2001. ISBN: 0-471-40593-0.
- Siegel, Jon, **CORBA 3 – Fundamentals and Programming, 2<sup>nd</sup> Edition**, Wiley Computer Publishing, 2000. ISBN: 0-471-29518-3.

*Några av dessa böcker kan finnas i nyare upplagor!*

---

### 9.1 Webbadresser till förlag

- O'Reilly & Associates  
<http://www.ora.com/>
- Wiley (och Wiley Computer Publishing)  
<http://www.wiley.com/> (<http://www.wiley.com/compbooks/>)
- Wrox Press  
<http://www.wrox.com/>

---

### 9.2 Webbadresser

- Object Management Group (OMG) och CORBA  
<http://www.omg.org/>
- Sun och Java  
<http://java.sun.com/>