# Bakgrund

Det var i stort sett mycket bra rapporter och systembeskrivningar men vissa saker verkar vi inte riktigt fått fram och behöver klara ut och förklara igen. Detta dokument sammanfattar de flesta av de som var lite oklara eller lite missvisande i rapporttexterna.

Observera att rapporterna inte handlade om rätt eller fel! De tjänade två syften:

1. Ni skulle tvingas tänka igenom vad ni gjort för att förbereda er inför labb 2 och 3
2. Vi lärare skulle få en bild av vad vi misslyckats förmedla och måste förklara på ett bättre sätt

Nedan kommer några områden där vi verkar varit otydliga eller otillräckliga i vår undervisning och kommunikation. Vi hoppas med detta dokument kunna sprida lite ljus över labben!

# Webb, html, http

En sak som är förvirrande är alla dessa relaterade men lite olika begrepp (som ofta nästan heter samma sak till råga på allt!) som förekommer inte minst i webbteknik. Vi försöker bena ut begreppen här.

## HTTP och Webbserver

En webbserver är ett program som kör kontinuerligt (det startas en gång med förhoppningen att aldrig krascha eller behöva startas om). Webbserverns uppgift är att erbjuda filer till en mängd klienter. Detta är typiskt för alla program som kallas servrar, att ett program liksom serverar något till många klienter. Lite som en restaurang - där serveras mat till en mängd klienter. Klienterna kommer in och begär (request) någon mat och restaurangen tar fram maten och serverar den.

En klient till en webbserver är typiskt sett en webbläsare (Chrome, Firefox, Opera, Dillo, Lynx, w3m, Safari, Edge eller Internet Exploder - förlåt, Explorer menade vi).

Hur sker då kommunikationen mellan en klient och en webbserver? Den sker via nätverket (t ex internet) och formaliseras i ett protokoll som heter HTTP (vilket också brukar stå först i URL:er till webbservrar och filer på en webbsever). HTTP specificerar hur en klient ska be att få någon resurs av webbservern, det kan röra sig om en webbsida (skriven i HTML), en fil eller data av något slag.

Data som skickas måste skickas i ett format som är standardiserat för att klienten ska kunna tolka och förstå data som kommer från servern. I labben så använder vi JSON som är en standard för att representera data om vad som helst på ett sätt som följer reglerna för formatet JSON. Man kan spara JSON-data som kommer från en webbserver som en fil på sin dator. Då kan man säga att det är som att hämta vilken fil som helst från webbservern. Men ofta så hämtar en klient (som då inte är just en webbläsare) JSON-data utan att spara den direkt på fil, utan i stället direkt tolkar den och gör något med datan. Så fungerar t ex ofta Android-appar, som t ex Västtrafiks reseplanerare. Den hämtar JSON med data motsvarande vad användaren matar in för resmål, och presenterar data för användaren i form av en föreslagen resväg.

HTTP specificerar också hur en klient ska tolka det svar den får från en webbserver. Om er webbläsare hämtar filen som beskrivs av URL:en https://www.gu.se/digitalAssets/1314/1314204_google.html så är det alltså filen 1314204_google.html som kommer skickas. I filen finns HTML-kod, vilket är ett språk främst

avsett för webbläsare. Koden i HTML beskriver hur text och bilder, länkar och annat ska visas för en vanlig människa. Webbläsaren är en klient som är expert på att parsea HTML (tolka och förstå HTML) och representera den som vanlig text och bild som vi människor är vana att se på webbsidor vi besöker. Men hur vet webbläsaren att det är just HTML som kommer från webbservern på www.gu.se? Det har inte med filändelsen .html att göra. Det har med HTTP att göra. HTTP dikterar att om en webbserver skickar ett svar med HTML-innehåll, så ska den skicka med en beskrivning i svarets "headers" som säger att det är just HTML som kommer.

Om jag ber att få filen ovan från www.gu.se så ser headers ut precis så här:

```
200 OK
Connection: close
Date: Fri, 16 Feb 2018 14:44:28 GMT
Via: 1.1 varnish (Varnish/5.2)
Accept-Ranges: bytes
Age: 245
ETag: W/"f4b1c-756-48ea0c9cfbac9"
Server: nginx/1.12.2
Vary: Accept-Encoding
Content-Length: 1878
Content-Type: text/html; charset=UTF-8
Last-Modified: Wed, 25 Aug 2010 07:25:38 GMT
Client-Date: Fri, 16 Feb 2018 14:44:28 GMT
Client-Peer: 130.241.151.114:443
Client-Response-Num: 1
Client-SSL-Cert-Issuer: /C=NL/ST=Noord-Holland/L=Amsterdam/O=TERENA/CN=TERENA
SSL CA 3
Client-SSL-Cert-Subject: /C=SE/L=G\xC3\xB6teborg/O=G\xC3\xB6teborgs
universitet/CN=www.gu.se
Client-SSL-Cipher: ECDHE-RSA-AES128-SHA256
Client-SSL-Socket-Class: IO::Socket::SSL
Grace: none
Title: Cookie-information, Google
X-Backend: tcsession: tcsession
X-Varnish: 5385451 4218782
```

Titta på raden i header-blocket som lyder **`Content-Type: text/html; charset=UTF-8`** . Det är den raden som talar om för min webbläsare att den ska förvänta sig HTML och visa innehållet, inte som kod utan som en fin webbsida. Nu är den inte så fin eller avancerad, men titta gärna på källkoden till samma sida och jämför. Källkoden är HTML exakt så som den skickas från webbservern. Det vi ser i vår webbläsare är HTML som är parsead och renderat som ett fint textdokument osv.

När vi ber att få JSON från vårt webb-api, så skickar faktiskt webbservern motsvarande rad men då är Content-Type satt till application/JSON i stället, så att ingen webbserver får för sig att tolka innehållet som HTML.

## Winstone

Här har vi varit otydliga. Men vi gör ett försök att reparera skadan. Winstone är ett Java-program som är en webbserver. Den huvudsakliga uppgiften för Winstone är alltså att köra kontinuerligt och lyssna efter klientanrop. Om en klient skickar en begäran om en resurs (t ex en fil) så ska Winstone hitta filen någon stans i webroot-katalogen eller någon av dess underkataloger. Anropet (request) ska innehålla en relativ sökväg till filen eller resursen, utgående från webbrot.

Skickar en klient (t ex er webbläsare) således följande request:
GET /search.html
så ska Winstone leta i sin webroot efter filen search.html som ska ligga direkt i webroot-katalogen. Sökvägen till resursen är en del av den URL ni har matat in i webbläsaren, t ex i detta fall http://localhost:8080/search.html

Det som kommer efter webbadressen (eller i förekommande fall efter port-numret 8080) är en slash, vilket är root-katalogen i webbserverns filsystem. Winstone får bara läsa filer i / (vilket alltså är kopplat till den webroot ni startat winstone med) och kataloger som ligger i webroot-katalogen. Enda undantaget är katalogen WEB-INF som den inte får läsa filer ur och skicka till klienter. Den är liksom hemlig för klienterna.

Om webroot hade innehållit en katalog som heter images och däri det legat en fil Henrik.jpg, så skulle ni kunnat hämtat och tittat på den bilden i en webbläsare genom att ange URL:en http://localhost:8080/images/Henrik.jpg . Den sökväg som gör att Winstone kan läsa filen och skicka dess innehåll till er webbläsare är då alltså `/images/Henrik.jpg` det vill säga en relativ sökväg utifrån webroot-katalogen ända till bildfilen. Er webbläsare skulle visa bilden (kanske föreställer den Henrik i en pinsam pose). Men hur tusan kan er webbläsare visa en bild från en annan dator? Jo, Winstone läser filen bit för bit och skickar den data den läser till er webbläsare. Återigen så är det headers som är lösningen på gåtan. I headers kommer Winstone ha skrivit att Content-Type är image/jpeg (eller liknande) och hur många byte stor filen är. Er webbläsare vet då att det minsann inte är fråga om HTML utan en bild. Webbläsaren har kod som inte bara kan rendera webbsidor av HTML, utan också kod som kan läsa data från en JPEG-bildfil och visa bilden.

Det som gör Winstone till en speciell sorts webbserver är att vi kan ställa in den så att den vet att vissa relativa sökvägar till resurser INTE är sökvägar till filer i dess webroot-katalog, utan i stället sådant som ska lämnas över till ett Java-program, kallat en Servlet. Winstone är i er labb inställd att veta att alla sökvägar som börjar med /search/ (följt av vad som helst) ska hanteras av den Servlet som ingår i labben. Vad Winstone då gör är att paketera hela det Request som t ex er webbläsare skickat via HTTP till ett Java-objekt med all information (kallat

HTTPServletRequest). Servleten kommer då att anropas med detta objekt, samt ett objekt för att svara med, ett HTTPServletResponse. När Servleten skriver sitt svar till response-objektet, ser Winstone det och låtsas helt enkelt att det är Winstone själv som svara klienten. Er webbläsare fattar inte skillnaden.

När vill man då ha en Servlet? När det gäller filer (HTML och bilder mm), så behövs ingen Servlet - det klarar vilken webbserver som helst av att servera. Det är när det är data som inte redan finns på filsystemet (under webroot) som ska serveras, som det behövs lite programmering t ex med hjälp av en Servlet.

Ett vanligt exempel är en sökning på en webbsida. Om ni matar in ordet “Sport” i sökrutan på google eller för den delen www.gp.se eller någon annan tidningssida, så får ni som svar en HTML-sida med länkar till alla sidor som innehåller ordet sport. Så fungerar även vår Wiki! Det finns en sökruta där. Använd den gärna.

Men den HTML-sida som visas med länkarna till alla sidor med sökordet, kan ju inte finnas redan i webroot. För vi har ju ingen aning om vad för konstiga ord som folk kommer söka efter! Fråga google vad folk söker efter och ni kommer bli deprimerade.

Så vi behöver alltså något slags logik för att skapa den sida som ska innehålla länkarna till de sidor som innehåller sökordet. Detta kan man göra med en Servlet (eller annan liknande teknik såsom PHP, JSP, ASP, ASP.NET).

Nu till hur Servleten kan veta vad ni sökt efter. När ni skriver något i en sökruta, så ligger sökrutan i en HTML-sida. Titta på källkoden till filen search.html som vi inkluderat i er webroot, för ett exempel.

HTML-koden med sökformuläret innehåller instruktioner till er webbläsare vart sökordet ska skickas, och vad sökordet ska kallas i det request som kommer skapas.

Om ni har ett enkelt formulär för sökning så kan det se ut (förenklat) så här i HTML:

```
<html>
<body>
<form action="/search" method=”GET”>
Search: <input type="text" name="search_word" />
<input type="submit" value="Search!">
</form>
</body>
</html>
```

Det som är centralt i ovanstående kod är:

- `action` (värdet `/search` )
- `method` (värdet `GET`)
- `name` (värdet `search_word`)

Skriver ni in “sport” i sökrutan och klickar på knappen [Search!] så kommer er webbläsare göra om det till följande URL som skickas till samma server som sidan med sökformuläret låg på med följande innehåll:

```
/search?search_word=sport
```

Det är en parameter med nyckel-värde där nyckeln kommer från HTML-koden och värdet kommer från det ni skrev i textfältet! Er webbläsare är verkligen om sig och kring sig.

Det som kommer efter ? i en URL kallas för GET-parametrar. Om detta skickas vidare till en Servlet (av t ex Winstone), så kommer HttpServletRequest innehålla GET-parametern “search_word” och värdet “sport”.

Servleten kan nu i Java (den är ju skriven i Java och exekverar i en JVM) hämta ut den String som är värdet i parametern search_word - "sport". Nu kan Servleten på något fiffigt vis leta i alla HTML sidor som finns i webrott-katalogen och dess underkataloger (utom WEB-INF, för den var ju superhemlig). När den hittar en träff, så skapar den en länk i HTML, t ex

```
<a href="/sport/sportnytt.html">Senaste nytt om sportens fina värld</a>
```

och så vidare för varje träff. Detta kan Servleten sedan skicka via sitt response-objekt som en fullständig HTML-sida. Den måste dock komma ihåg att själva sätta Content-Type till text/html så att er webbläsare har en chans att fatta hur den ska visa innehållet i svaret.

I er webbläsare ser alltså URL:en ut som om den gick till en sida på samma server och en fil som hette `/search?search_word=sport` - men vi som läst kursen TIG058 vet ju att detta inte är en fil, egentligen, utan HTML som skapats dynamiskt (vid behov, on the fly) av Servleten.

## Vad är en Servlet?

En servlet är ett objekt av en klass som implementerar interfacet javax.servlet.Servlet, oftast indirekt genom att ärva javax.servlet.http.HttpServlet. Man kan se en servlet som ett program som anropas av en webbserver som kan köra servletar (se Winstone ovan för ett exempel på en sådan webbserver).

En HttpServet (som vi hävdar är den absolut vanligaste formen av Servletar) är då ett program som kan hantera protokollet HTTP som en server. HTTP används i en klient-server-arkitektur där klienterna ofta är webbläsare och servern ofta är en HTTP-server (kallat en webbserver).

Det finns flera metoder som har med HTTP att göra i en HttpServlet. De vi fokuserat på i kursen är void doGet(HttpServletRequest req, HttpServletResponse resp) och void init() .

doGet får sina argument av t ex Winstone om Winstone beslutar sig för att ett request från en klient ska hanteras av en servlet och inte representerar en fil som ska hämtas från filsystemet. request-parametern innehåller all information om den ursprungliga requesten från klienten. response-parametern, innehåller metoder för att sätta headers (content-type t ex) och skriva svar till klienten (via en OutputStream). doGet är den metod som ansvarar för ett request av metodtypen GET i HTTP (dvs den vanligaste typen av request, t ex när ni skriver en adress i webbläsaren och trycker enter).

init() är en metod som körs en gång när en servlet anropas första gången. Lite som en engångs-konstruktor (i brist på bättre beskrivning). Det var i init() som i labbens servlet vi fixade problemet med svenska operativsystem som fick Rikards kassa JSON-skapande att skapa felaktig JSON för flyttal, då svenska operativsystem tycker att decimalavskiljaren är ett kommatecken medan JSON vill ha en punkt (eftersom komma har en annan betydelse i JSON). Det var också i init() som vi läste in vilken sorts ProductLine som ProductLineFactory ska välja. Detta kommer ni se i labb3 men det går ut på att vi kan ändra i web.xml från FAKE till DB för att få ProductLineFactory att inte välja FakeProductLine utan SQLBasedProductLine i stället (som hämtar produkter från databasen). Detta gör att vi inte behöver ändra en enda rad kod i Servleten i labb3 för att skifta från FakeProductLine till SQLBasedProductLine (som ni kommer skriva i labb3). Vi kommer bara behöva ändra i web.xml (och starta om Winstone).

En HttpServlet kan också implementera metoden POST som finns i HTTP för att skicka större mängder data (och detta utan att använda URL:ens GET-parametrar). Det används dock inte i denna laboration in någon del.

Man använder alltså Servletar när man vill kunna generera ett HTTP-response on-the-fly, dynamiskt, i stället för att bara låta en webbserver hämta filer från sin webroot. Tjänster på internet använder servletar eller liknande teknik, vilket låter användarna av tjänsterna använda t ex HTML-sidor för att interagera med tjänsten. Sidorna skickar request till webbservern som skickar dem vidare till Servleten (eller motsvarande teknik skriven i annat språk) och får ett vanligt HTTP-svar precis som vid kommunikation med vanliga webbservrar. En sida med en tjänst för att t ex översätta text från svenska till engelska skulle alltså kunna ha ett formulär i HTML som skickas till en webbserver som skickar vidare hela requestet till en Servlet. Servleten kan göra översättningen via fiffig Java-kod som anropas från doGet() och svara med HTML som innehåller översättningen av texten. Det är naturligtvis inte möjligt att använda färdiga, statiska HTML-filer för översättning, eftersom vi inte har en aning om vilken text som ska översättas varje gång någon anropar översättartjänsten.

## localhost:8080 - vad är det egentligen?

Ni anropar er servlet på adressen localhost:8080. Man använder adressen “localhost” när man vill ansluta till en server på sin egen dator, utan att gå via nätverket. Ni kan tänka på localhost som att er webbläsare går ut på ert nätverkskort som vanligt, men vänder i dörren och går tillbaka till er egen dator. Ett alternativ till adressen localhost är att använda IP-numret 127.0.0.1. Vad gäller 8080 så är det en så kallad port (se t ex vår wikisida Introduction to network). Ett nätverkskort på en dator brukar ha en IP-adress (också kallat IP-nummer) så att andra datorer kan hitta och kontakta datorn. Om datorn kör en server (ett program som lyssnar på nätverkskortet) så har den servern en port som den lyssnar på, på IP-numret. Det gör ju att man kan köra flera serverprogram på samma dator, så länge de har olika portnummer. Winstone använder port 8080 vilket är en inofficiell standard för HTTP där webbservern körs av en vanlig användare och inte administratörskontot (eller super user som det heter i vissa OS). En vanlig webbserver kör på standardporten 80. Detta är så standardiserat att ni inte ens behöver skriva in :80 när ni surfar till en webbsida på internet (men ni kan göra det). 8080 brukar man välja för webbservrar som körs av vanliga användare på en dator, eftersom vanliga användare inte får starta en server med en port lägre än 1024. Se vår wiki (länkad tidigare) för en lista på vanliga standardportar för olika sorters serverprogram (olika nätverkstjänster).

Att ni kör med localhost:8080 betyder inte att ert webb-api inte är åtkomligt via ert nätverkskort från andra datorer. Just på GU och nätet eduroam, så går det inte på grund av att GU satt upp det trådlösa nätet på så vis att det förhindrar datorer på samma nät att kontakta varandra. Det är av säkerhetsskäl helt enkelt. På nätverksworkshopen satt lärarna upp ett eget trådlöst nät (tre stycken faktiskt) som inte hade denna säkerhetsbegränsning. De såg deltagarna i workshopen att det gick att komma åt varandras webb-apier, om man visste varandras IP-nummer.

När man utvecklar en servertjänst, är det vanligt att man arbetar med localhost innan man är klar. Det gör att man kan använda samma dator för att anropa servern som för att köra servern. Man kan t ex starta Winstone i en terminal och använda wget, curl, nc, eller annat nätverksprogram i en annan terminal för att kommunicera med Winstone.

# JSON

JSON är alltså en standard för textfiler som representerar data. Den är väldigt fiffigt formulerad, så att det går att läsa ut hierarkiska strukturer av data. T ex kan man med JSON representera listor av objekt, där varje objekt består av nyckel-värde par och även nya listor och nya objekt.

Här är ett exempel från vår Wiki (ett exempel vi snott från Oracle iofs men ändå):

```
{
   "firstName": "Duke",
   "lastName": "Java",
```

```
    "age": 18,
    "streetAddress": "100 Internet Dr",
    "city": "JavaTown",
    "state": "JA",
    "postalCode": "12345",
    "phoneNumbers": [
       { "Mobile": "111-111-1111" },
       { "Home": "222-222-2222" }
    ]
}
```

Denna JSON beskriver ett enda objekt med data, en person verkar det vara. Personen har förnamn, efternamn, ålder, adress, stad, postnummer och en lista med telefonnummer. Varje telefonnummer i listan är ett objekt med nyckel-värde-par.

JSON gör det möjligt att överföra data från ett system till ett annant, eller en applikation till en annan. En applikation kanske läser själva datan från en databas, skapar JSON och skickar det till en annan applikation. Den andra applikationen kan då läsa JSON (i stället för att läsa databasen) och tolka datan i JSON-texten på något vis. Kanske presenterar den andra applikationen datan fint och översiktligt för en användare, kanske gör den något annat.

Med JSON så kan vi alltså överföra data på ett neutralt sätt. Den som hämtar JSON har ingen aning om vilket programmeringsspråk som den som skickar JSON är skriven i. Och den som skickar JSON behöver inte bekymra sig över vad för programmeringsspråk den som tar emot JSON är skriven i. En annan fördel med JSON är att även vi människor faktiskt kan tolka och förstå den data som är kodad i JSON-formatet. Vi väljer dock att helst inte gå in i en diskussion om skillnaden mellan data och information, det överlåter vi med varm hand till andra.

# Parsing

Att parsea brukar betyda att läsa in något i ett känt format och kolla att formatet dels stämmer med förväntningarna, dels försöka representera det inlästa med hjälp av ett programmeringsspråk och dess datatyper.

Tittar vi på JSON-exemplet ovan, så skulle vi t ex med ett Java-program kunna läsa in JSON-texten och kolla att det följer reglerna för JSON-standarden. Sedan skulle vi kunna tolka det genom att läsa ut de data som finns i texten och skapa Java-objekt för denna data. Vi skulle kunna representera texten i exemplet som ett objekt av klassen Person och låta denna klass deklarera instansvariabler för förnamn, efternamn, address, stad, postnummer och en lista av PhoneNumber-objekt för telefonnumren. Ett PhoneNumber-objekt skulle kunna ha två variabler, type (t ex mobile) och number (själva numret).

Detta gör att vi kan få data om personer från världen utanför vårt Java-program men tolka in den i Java-världen och representera den som objekt av olika klasser. Detta kan man kalla parsing. Eller möjligen parsning på svenska.

Ett annat fall av parsning har ni använt när ni vill göra om en String till en double (t ex i Product.Builder där en Product.Builder har en metod price() som vill ha en double men ni kanske har en String). I klassen Double finns en statisk metod som heter parseDouble som tar en String som argument och returnerar en double som är resultatet av att parsea (tolka och verifiera) den String som kom som argument. Om det är omöjligt att tolka texten som en double kommer metoden att kasta ett exception av typen NumberFormatExeption. Motsvarande metoder finns i klasserna Integer, Byte, Long, Short, Boolean, Float osv.

Det Servleten parsear är dock själva GET-parametrarna. GET-parametrarna utgör ju en kodning av vad för produkter vi vill ha i form av JSON. Eller rättare sagt, en kodning av kriterier som de produkter vi vill ha måste uppfylla (t ex hur mycket och litet de får kosta, vilken alkoholstyrka de minst eller mest får ha osv).

GET-parametrarna kommer ju dock som en text med nyckel-värde-par separerade med &-tecknet från URL:en. Det Servleten ber sin parser att göra, är att tolka parametrarna och bygga upp ett Java-objekt som representerar det parametrarna till sammans uttrycker, i form av ett så kallat Predicate (ett predikat - tänk på det som ett påstående som är sant eller falskt för varje produkt). Varför vill Servleten att parsern skapar ett Predicate av parametrarna då? Jo, för att Servleten vill ju ha de produkter som uppfyller det kriterium som parametrarna beskriver tillsammans. Och Servleten har ju tillgång till en ProductLine, vilken råkar ha just en metod som givet ett Predicate kan ge en lista på alla produkter för vilket detta predicate är sant - det vill säga en lista på produkter som uppfyller kriteriet som klienten har skickat.

Servleten använder sedan denna lista med rätt (filtrerade) produkter för att skapa en JSON-text att skicka tillbaka som svar till klienten.

Servleten är dock lite lat, och ber ett objekt som är specialist på att skapa JSON av produktlistor att göra jobbet, och får en String med hela JSON-grejen tillbaka och skickar den som svar till klienten (efteratt ha sagt att Content-Type ska vara application/json så klart!).

Att gå från Java (t ex en List<Product>) till en JSON-text kallas inte för parsning. Java vet hur listor och Product fungerar redan och behöver inte tolka om dem. Det kallast för att formatera data, eller omvandla data eller transformera data eller serialisera data. Att exportera data från Java till ett format som inte har med Java att göra helt enkelt. Det finns många namn på detta, men vi valde att kalla objektet för Formatter - vilket kanske var förvirrande för vissa. Poängen här är dock att processen att exportera ut data från Java-objekt till ett format avsett att skickas till annan applikation inte heter parsning. Det är något annat - ja, till och med tvärt emot vad parsning är faktiskt.

I labb 3 kommer ni att få denna JSON som webb-apiet (servleten) har skapat, rakti in i Swing-klienten. Nu måste ni kunna parsea JSON till Java-objekt (närmare bestämt till en List<Product>) eftersom Swing inte direkt kan presentera JSON för en användare (utom som ren JSON-text men det vill användare sällan se!). Swing är däremot bra på att visa tabeller och listor av Java-sort. T ex en List<Product> är lätt att visa med Swing. Java gillar Java-objekt liksom. Så er parsning i labb 3 kommer att handla om att göra om den JSON-String ni får av servleten till en List<Product> - en klassisk parsning!

# Double.parseDouble vs. Double.valueOf

I klassen Double finns (bland annat) två statiska metoder, parseDouble och valueOf som bägge tar en String som argument. Båda "parsear" en String med text som (förhoppningsvis) representerar ett flyttal och omvandlar texten till något i Java-världen.

parseDouble returnerar ett värde av typen double (en primitiv typ, alltså en värdetyp). valueOf, däremot, returnerar ett nytt Double-objekt som representerar värdet i texten. Bägge kastar ett runtime-exception (NumberFormatException) om texten i String:en inte går att tolka om till ett flyttal. Och bägge kastar ett NullPointerException (även det ett runtime-exception) om argumentet är null.

Varför fungerar det att använda båda då, i t ex argumentet till Product.Builder:s price-metod? Metoden price(double) vill ju ha ett primitivt double-värde. Anledningen till att båda fungerar är att Java är flexibelt när det gäller att använda Double-referens eller ett double-värde. Om en metod vill ha ett double-värde och man ger som argument en referens till ett Double-objekt, så är Java snällt nog att göra om Double-objektet referensen refererar till, till ett double-värde åt oss.

# Systembolagets XML-fil

Vi visade i början av introduktionen till labbarna hur vi hämtat data från systembolaget och varför vi vill skapa ett eget webb-api. Bakgrunden är att Systembolaget har en sida som de kallar för API, där man kan hämta en XML-fil med alla produkter bland annat. Filen uppdateras varje dag och innehåller alla produkter (cirka 20 000 produkter). Formatet de har valt för att exportera alla data om produkterna är XML. Det är ett format som används till samma saker som JSON - att i textformat koda data så att människor (som kan läsa XML) och datorer kan använda datan.

Det som vi lärare såg som ett problem med att Systembolaget erbjuder alla sina produkter som XML var inte valet av format, utan det faktum att man som utvecklare får alla produkter eller inget alls. Det är ganska otympligt att tänka sig att t ex en Android-app ska behöva hämta alla 20 000 produkter i form av XML-filen varje gång den vill söka efter en viss produkt enligt något

sökkriterium. Vore det inte bättre om Systembolaget hade erbjudit ett riktigt API, ett webb-api där vi kan be att få vissa proukter enligt ett sökkriterium?

Detta ledde till idén bakom labbarna. Detta kan ju våra smarta studenter skriva! Ett webb-api där man kan be att få vissa produkter filtrerade på något kriterium. Vi valde att låta ert system erbjuda just detta men i formatet JSON. JSON är lite lättare (mindre text och mindre byråkratiskt) än XML. I övrigt är det inget fel på XML. Men det fanns ett problem kursen handlar inte om Java och XML (eller ens om JSON). Kursen handlar om Java och Databaser. Detta löste vi genom att sjävla hämta hem alla produkter (för ett tag sedan, så det är inte dagsfärska data!) och lägga in dem i en databas. Detta gjorde vi innan kursen började, så databasen finns (den ingår i de filer ni hämtat hem för labbarna). Vi skrev helt enkelt ett litet Java-program som läste XML-filen och skapade SQL för att lägga in samma data (inte allt men vissa urvalda delar om alla produkter) i en databas vi skapat.

Eftersom vi redan gjort det jobbet, så behöver ni inte bry er om XML-filen med alla produkter. Den nämndes av oss mer som kuriosa och bakgrundsinformation. Systembolagets så kallade API är inte ett API enligt oss, utan en export av deras produktdatabas i XML-form. Det går inte att programmera mot Systembolagets API (eftersom vi hävdar att det inte är ett API - själva definitionen av ett API är att det är ett gränssnitt för just programmering). För att skriva en applikation som använder Systembolagets data, måste man hämta hela XML-filen och tolka innehållet själv. Det är ju inte optimalt att behöva hämta 20 000 produkter om man vill ha bara produkter i ett visst prisintervall, t ex. Dels så hämtar man ju för många produkter där de flesta är ointressanta. Dels så måste man göra filtreringen själv, genom att löpa igenom alla 20 000 produkter i XML-filen och spara de som uppfyller sökkriteriet.

För att labbarna ska passa bättre in i kursen så gjorde alltså vi detta jobbet åt er innan kursen ens börjat. Så ur ert perspektiv, så finns inte XML-filen som en del av ert system. Det finns en databas i stället (haha! så att vi kan tvinga er att använda JDBC och SQL i stället för att parsea XML!).

Det hade visserligen lika gärna kunnat vara så att Servleten fick data från en ProductLine som läser XML-filen. Men så är det alltså inte. I Labb 3, så kommer ni skriva en ProductLine som faktiskt läser in hela databasen i minnet och erbjuder den som en List<Product> samt en filtrerad List<Product> givet ett Predicate (precis som Servleten gör nu med er FakeProductLine). Ur Servletens perspektiv så kan det kvitta - den ser ju bara interfacet ProductLine och får en sådan från ProductLineFactory. Men vi ville ju ha med databaser i labben på grund av att kursen faktiskt handlar om databaser.

Det verkar ha förvirrat många av er att vi nämnde XML-filen så tidigt. Men, igen, XML-filen var alltså bara lite bakgrundsinformation om varifrån alla data kom, och varför vi ville bygga ett webb-api i stället för att använda den stora XML-filen.

# Interface och syftet med dem

I labbarna används många interface i den kod ni får från oss. Det första ni stötte på var i Servleten för webb-apiet, som ju behöver en ProductLine att be om produkter som uppfyller filtret som servleten kommit fram till genom att titta på sina GET-parametrar.

ProductLine är just ett interface och det deklarerar två metoder:
`List<Product> getAllProducts()` respektive
`List<Product> getProductsFilteredBy(Predicate<Product> predicate)`.
Ni kan repetera interface i boken som användes i introduktionskursen:

1. Chapter:Interfaces_-_Introduction
2. Chapter:Interfaces_-_Introduction_-_Exercises
3. Chapter:Interfaces_-_Implementing_an_interface
4. Chapter:Interfaces_-_Implementing_an_interface_-_Exercises
5. Chapter:Interfaces_-_Writing_an_interface
6. Chapter:Interfaces_-_Writing_an_interface_-_Exercises
7. Chapter:Interfaces_-_Program_against_an_interface
8. Chapter:Interfaces_-_Program_against_an_interface_-_Exercises
9. Chapter:Interfaces_-_Creating_an_anonymous_class
10. Chapter:Interfaces_-_Creating_an_anonymous_class_-_Exercises
11. Chapter:Interfaces_-_Rules_and_syntax
12. Chapter:Interfaces_-_Rules_and_syntax_-_Exercises

Anledningen till att vi använder ett interface för ProductLine är för att kunna dölja den riktiga klassen som används för de två metoderna. Genom att Servleten har en variabel `products` av interface-typen ProductLine, så kan vi byta ut den verkliga klassen senare utan att Servlet-koden behöver skrivas om. Anledningen är att ett interface blott deklarerar abstrakta metoder och inte kan instantieras (man kan inte skapa instanser med new av interface-typer, det kan man bara göra med konkreta klasser).

Interfacet tjänar alltså till att deklarera metoder som vissa klasser ska ha för att få kalla sig en ProductLine. Ni skrev ju klart klassen FakeProductLine. Den klassen säger att den är ett slags ProductLine redan i klassdeklarationen:

```
public class FakeProductLine implements ProductLine
```

När en klass deklareras med `implements NågotInterface` så vet vi sedan förra kursen två saker:

1. Vi kan använda `NågotInterface` som typ på en referens som refererar till en sådan klass
2. Vi vet att alla klasser som implementerar `NågotInterface` har samtliga metoder som interfacet listar, färdigimplementerade och redo att användas

Översatt till `ProductLine` och servleten betyder det att servleten kan ha en referensvariabel av typen referens till ProductLine, oavsett vad objektet den refererar till har för typ, eftersom objektet implementerar interfacet och alltså har båda metoderna `getAllProducts()` och `getProductsFilteredBy(Predicate)` färdiga att användas. Det betyder också att Servleten, när vi kommer till sista labben, kan behålla sin kod och logik intakt och helt oförändrad, även om variabeln av typen ProductLine då kommer att referera till en `SQLBasedProductLine` i stället för som nu en `FakeProductLine`.

Det vi köpte oss genom att skapa ett interface (och en factory) var alltså flexibilitet i framtiden om vi vill att servleten utan att skrivas om ska kunna använda *olika sorters* *`ProductLine`-implementationer* (en fake först och en som pratar databas senare).

Vi ville alltså lära er att genom att programmera mot interface så kan vi skydda vår kod mot förändringar i framtiden. Hur går då förändringen till? Hur byter vi från FakeProductLine till SQLBasedProductLine i framtiden? Det sköter klassen ProductLineFactory om. Servleten tilldelar ju sin ProductLine-variabel en ProductLine som den får av fabriken. Hur fabriken vet vilken sorts verklig (konkret) ProductLine den ska välja kommer vi visa inför Labb3 men den nyfikne kan ju titta i källkoden.

Ni kan läsa om Factory-mönster (lite olika varianter) i vårt kapitel om detta på vår wiki (pdf, övningar och videoföreläsning finns).

# Predicate

`Predicate` är ett interface med bara en abstrakt metod (sådana interface kallas även funktionella interface - de går att använda som en funktion med betoning på *en*). Interface (även funktionella interface) kan även deklarera så kallade default-metoder. Sådana metoder är inte abstrakta utan har faktiskt redan kod skriven direkt i interfacet. Ni kan repetera interface (se länkar ovan) i vår wiki.

Den enda abstrakta metoden i det funktionella interfacet Predicate heter
`public abstract boolean test(T t)`

T är en generic-typ, då interfacet är deklarerat `public interface Predicate<T>` Detta innebär att vi kan skapa predikat typade för en viss klass, på samma sätt som vi kan skapa en `List<Product>`. Vi kan alltså ha t ex ett `Predicate<Product>` . Vad är interfacet Predicate till för då? Jo, vi kan ha ett objekt som implementerar t ex `Predicate<Product>` och representerar ett påstående om en `Product`. T ex att price är mindre än 20.00. För detta predikat gäller alltså att antingen är predikatet (påståendet) sant, eller så är det falskt. Vi använder metoden test och ger en produkt som argument. Om produkten vi kastar in till metoden `test` i vårt predikat har ett pris som verkligen är mindre än 20.00, så returnerar test `true`. Annars `false`.

Det är ganska fiffigt med `Predicate<T>`-interfacet. Det gör att vi kan skapa villkor för vilken typ av objekt som helst och "kapsla in" detta villkor i ett objekt som vi senare kan fråga om villkoret är uppfyllt. I laborationerna så har ju `ProductLine` en metod för att hämta alla produkter som uppfyller ett visst villkor (ett predikat!).

Metoden `getProductsFilteredBy(Predicate<Product>)` tar ju ett produkt-predikat som villkor. Metoden kan då lätt filtrera ut ur den stora listan med alla produkter, de produkter som uppfyller predikatet!

Det bökiga är förstås att skapa ett predikat för produkter utifrån de GET-parametrar som servleten anropas med. Vi har föreläsningsfilm om hur detta görs och vi gick igenom det på en av workshoparna i början av kursen. Det finns även en föreläsnings-PDF till filmen.

I korthet så använder servleten ett objekt av klassen ParameterParser (som lärarna skrivit). Relevant kod för att omvandla GET-parametrarna till ett produkt-predikat i servletens `doGet`-metod är:

```
ParameterParser paramParser =
    new ParameterParser(request.getQueryString().split("&"));
Predicate<Product> filter = paramParser.filter();
```

Servleten skapar en `ParameterParser` och ger som argument till dess konstruktor alla GET-parametrar som en `String[]` (array av String) från "URLen" servleten anropades med. Sedan ber servleten parsern om ett produkt-predikat (som parsern är snäll nog att skapa). Detta gör servletens `doGet`-metod till ganska kort och förhållandevis lättläst, jämfört med om vi skapat hela predikatet direkt i `doGet`-metoden.

# ParameterParser

Väldigt mycket av Java-kod består av metoder som skapar objekt och anropar metoder i dessa objekt. Till en början kan detta förstås se konstigt ut och vara svårt att följa. Att det ser ut på detta vis har att göra med att i objektorienterade språk, som Java, så låter man objekt vara centrala och ha ansvarsområden eller förmågor. Om man inte använder sig av metoder som anropar andra metoder, så skulle man ju hamna i en situation där man har bara en enda metod som är fem kilometer lång (eller längre).

Visst hade vi kunnat låta Servleten vara den enda klassen och doGet den enda metoden i hela webb-apiet. Men det hade blivit en mardröm att läsa och underhålla den koden!

Strategin (eller designen om ni så vill) som vi använt oss av är att inte låta doGet göra allt jobb. I stället bryter vi ned jobbet som webb-apiet ska göra i abstraktioner för varje beståndsdel. Så vad är det servleten då måste göra i doGet?

1. Tolka requestet (förfrågan eller anropet, särskilt då GET-parametrarna)
2. Utifrån tolkningen av t ex min_price=100&max_price=150 hämta just de produkter som uppfyller det kravet (har ett pris mellan 100 och 150 kronor)
3. Svara den som anropade med en JSON-String
    a. Utifrån tolkningen skapa ett predikat-objekt som kan testa en produkt (i taget) mot kravet att priset ligger mellan 100 och 150 kronor)
    b. Hämta alla sådana produkter någonstans ifrån (från en ProductLine och dess productsFilteredBy(Predicate<Product>)-metod
    c. Göra om listan på sådana produkter till JSON

Klassen ParameterParser har en konstruktor som tar en String-array med GET-parametrar och skapar ett ParameterParser-objekt som kan utföra vissa uppgifter (via dess instansmetoder). En central metod här är metoden filter() som returnera ett produkt-predikat som motsvarar det GET-parametrarna uttryckte.

Så servleten använder alltså ett objekt av klassen ParameterParser för att få fram ett predikat för produkter - ett predikat som kan svara på frågan om en viss produkt uppfyller det kriterium som uttrycktes i GET-parametrarna (t ex min_price=100&max_price=150). Detta ersätter cirka 45 rader kod för att göra om GET-parametrar till ett Predicate<Product>. Låt ett objekt göra detta och göra detta bra och inte så mycket annat!