Typer

Sidst ændret: 07/19/2018 14:11:10

"..."

- ...

Først definerer vi typebegrebet i termer af operatorer og operander. Dernæst gennemgår vi heltal, komma-tal og tegn, der alle er primitive typer. Vi berører kort tekststrenge. Vi studerer kompatibilitet mellem typer og ser hvorledes man kan bruge casting. Bemærk at visse af udskrifterne fra compileringsforsøgene, i afsnittet om kompatibilitet, har et ekstra linieskift for at gøre dem smallere. Ligeledes er linienumre i disse udskrifter korrigeret så de passer med de dele af kildeteksten der er vist.

Forudsætninger:

Grundlæggende mængdelære, samt forståelse af funktionsbegrebet med afbildninger. Det er en fordel, men ikke nødvendigt, at have læst kapitlet "Talsystemer" under "Assembler Programmering".

1. Typebegrebet

Hvad er en type? Lad os se definitionen:

Definition: Type

En type er givet ved:

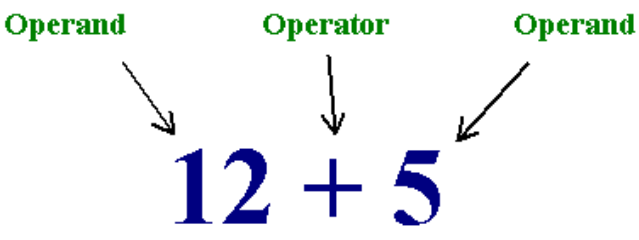
- En værdimængde
- En række operatorer

Værdi-mængde

Værdimængden er de mulige værdier en variabel af typen kan antage. Alle typer har en endelig værdimængde - træerne gror ikke ind i himmelen. Når man vælger en type, gør man det ud fra hvilken værdimængde man ønsker. Hvis man vælger en type med for begrænset værdimængde vil der opstå logiske fejl, mens en for stor værdimængde vil give et pladsspild. Pladsspildet kommer af den bagved liggende datarepræsentation - jo større værdimængde, jo mere plads skal der bruges til at repræsentere en værdi.

Operatorer

Operatorerne bringer liv til typen. De gør det muligt at arbejde med værdierne - at operere på dem. Lad os se et eksempel.



Figur 1: *Operator og operander*

Operander

Vi har her en operator, som opererer på de to værdier 12 og 5. Man kalder de to værdier **operander**,

fordi de optræder i forbindelse med en operator, som opererer på dem. Når en operator virker på operander skabes der en ny værdi, i dette tilfælde 17, der normalt tilhører den samme værdimængde som operanderne (der er undtagelser).

Unære, binære og tertiære operatorer

Plus er en binær operator, fordi den tager to operander. Der findes også unære operatorer der kun tager én operand, f.eks. fortegns-minus. Resultatet af at bruge operatoren fortegns-minus på en operand, er at man får vendt fortegnet. Der findes også noget så specielt som en tertiær operator, en operator der tager tre operander - den ser vi på i kapitlet "Selektion".

2. Heltal

2.1 Værdimængde

Intervaller omkring 0

Den første gruppe af typer vi vil se nærmere på er heltal. Heltal i Java er kendetegnet ved at deres værdimængde fordeler sig symmetrisk omkring 0. Der er en ligelig fordeling mellem positive og negative værdier. Operatorerne er fælles for de forskellige heltals-typer, det er værdimængderne der er forskellige

Tabel 1: *Heltals-typer*

Type	Minimum	Maximum	Lager
`byte`	-128	127	8 bits
`short`	-32.768	32.767	16 bits
`int`	-2.147.483.648	2.147.483.647	32 bits
`long`	-9.223.372.036.854.775.808	9.223.372.036.854.775.807	64 bits

Her er minimum- og maximum-værdierne for de fire heltals-typer i Java angivet (se evt. kapitlet "Talsystemer" under "Assembler Programmering" vedrørende den lille asymmetri i intervallet, samt betydning af antallet af bits).

2.2 Operatorer

Operatorerne er de almindelige aritmetiske operatorer +, -, *, / og %.

Heltals-division

En speciel ting ved heltal, er betydningen af /. Der er tale om **heltals-division**. Der afbildes over i heltal ved at en evt. rest smides væk. Det gælder både for positive og negative værdier, som det ses af følgende liste med division med 3 på intervalet [-4:4]:

Tabel 2: *Division med 3 på intervallet [-4:4]*

```
 4/3 = 1
 3/3 = 1
 2/3 = 0
 1/3 = 0
 0/3 = 0
-1/3 = 0
-2/3 = 0
-3/3 = -1
-4/3 = -1
```

Som det ses, svarer det til, at man for de negative tal først tager en heltals-division uden fortegn og efterfølgende sætter minus foran resultatet.

Modulus

% er **modulus**, som vi introducerede i kapitlet "Simpel Programmering". Vi vil her også se den for negative værdier, idet vi ligeledes vil liste modulus 3 på intervallet [-4:4]:

Tabel 3: *Modulus 3 på*

```
4%3 = 1
3%3 = 0
```

intervallet [-4:4]

```
 2%3 = 2
 1%3 = 1
 0%3 = 0
-1%3 = -1
-2%3 = -2
-3%3 =  0
-4%3 = -1
```

Igen kan man tolke det som, at man for de negative tal tager modulus 3 uden fortegn og dernæst sætter et minus foran resultatet.

2.3 Implementation

2. komplement

Heltals-typerne i Java er implementeret som binære tal med 2. komplement (se evt. kapitlet "Talsystemer")

2.4 Overflow

Udtryk med heltal giver typemæssigt et resultat af samme type som operanderne, men hvad sker der hvis det ikke er muligt? Hvis resultatet f.eks. bliver for stort for værdimængden?

Lad os se et simpelt eksempel hvor en **`byte`**-variabel bliver talt op over grænsen på 127.

Bemærk at vi i eksemplet skriver **`b++`**. Det er en forkortet skrivemåde for **`b=b+1`**. Vi vil ikke her komme nærmere ind på mulighederne for at forkorte syntaksen (se evt. kapitlet "Syntaktisk Sukker"), men forkortelsen er desværre nødvendig for at holde eksemplet simpelt[1].

```
class TestOverflow {
  public static void main( String[] argv ) {
    byte b = 127;
    b++;
    System.out.println( b );
  }
}
```

```
-128
```

Ud af den ene ende og ind af den anden

Der sker ganske enkelt en fejl. En fejl der bevirker at man forsvinder ud af den ene ende af værdimængden (den positive) og kommer ind af den anden (den negative). En nærmere forklaring af hvad der rent teknisk sker, kræver at man kender binær repræsentation med 2. komplement (se evt. kapitlet "Tal-systemer" under "Assembler Programmering").

Den samme fejl forekommer hvis man forsvinde ud af den negative ende; hvor man tilsvarende vender tilbage gennem den positive. Dette sker ved at trække for meget fra, så man ryger ud af værdimængden.

3. Komma-tal

Reelle tal

Komma-tal kaldes også flydende tal eller decimal-tal. Målet er at kunne repræsentere de reelle tal, et mål der dog må begrænses på forskellig måde for at det kan lade sig tilnærme.

Punktum i stedet for komma

Hvis man ikke allerede er bekendt med det, skal man være opmærksom på, at den engelske notation anvendes i alle programmeringssprog; hvor der bruges punktum i stedet for komma. Vi vil dog betegne dette punktum som "komma".

3.1 Værdimængde

Der findes to typer til repræsentation af komma-tal. Man kan vælge mellem **`float`** og **`double`** alt efter hvor store, små eller nøjagtige tal man vil have.

Nøjagtighed

Til daglig er vi vant til at ikke alle tal er lige nøjagtige. Når vi hører, at noget har kostet 50.000,- kr., så ved vi godt, at nullerne ikke nødvendigvis skal tages så bogstaveligt. Det kunne f.eks. godt være prisen helt nøjagtigt havde været 50.120,50 kr.

Sådan er det også med repræsentation af komma-tal. Nogle af cifrene kan vi regne med, mens andre kun angiver størrelsen.

Følgende tabel angiver disse to egenskaber ved repræsentationen af komma-tal:

Tabel 4: *Komma-tals-typer*

Type	Nøjagtighed	Nuller	Lager
`float`	7 cifre	38	32 bits
`double`	15 cifre	307	64 bits

Hvis vi f.eks. vil repræsentere tallet:

```
125552501513
```

med en **`float`**, vil begrænsningerne i dens kapacitet betyde at tallet kun bliver repræsenteret som:

```
125552500000
```

idet vi ikke vil kunne huske de fem sidste cifre, men kun de syv forreste.

Havde vi haft et lille tal:

```
0.0000125552501513
```

der ligeledes blev repræsenteret i en **`float`**, ville vi kun huske det som:

```
0.0000125552500000
```

idet nøjagtigheden igen kun vil være de syv **mest betydende cifre**.

Mest betydende cifre

Betegnelsen "mest betydende cifre" refererer til de cifre der har den største **positionelle værdi**, og som har betydning for tallets nøjagtighed.

Positionel værdi

Den positionelle værdi er den positionsværdi som et ciffer har i kraft af sin placering i tallet. F.eks. har cifret 5 i tallet 152, den positionelle værdi 10, fordi den er antallet af ti-ere.

Mht. betydning for tallets nøjagtighed, så er f.eks. ingen af de fire nuller mellem kommaet og 1 betydende, på trods af at de står forrest og deres positionsværdier derfor er større. Det skyldes, at de kun angiver størrelsen af tallet, at vi er nede i titusindedele, og ikke har betydning for nøjagtigheden.

`float` er meget begrænset

En **`float`** har ikke den store nøjagtighed. Tal på mere end 99.999,99 kr. kan ikke repræsenteres med øres nøjagtighed. Derfor bør man altid bruge **`double`** med mindre man har konkrete pladsproblemer. Det ville være rigtig træls at lave et system til en bank og så have brugt **`float`** et eller andet sted!

Nedenfor, under repræsentation, vender vi tilbage til værdimængden for komma-tal, da den har

nogle yderligere matematiske begrænsninger i forhold til de reelle tal.

3.2 Operatorer

Ingen modulus

Operatorerne er de sædvanlige aritmetiske operatorer +, -, * og /. Bemærk, at der ikke findes noget modulus for komma-tal.

3.3 Implementation

Hvordan er komma-tal implementeret? Man kan tage udgangspunkt i heltal. Forskellen på et heltal og et komma-tal er at der ved heltallet ikke kommer nogen cifre efter kommaet, men kommaet er der på sin vis alligevel.

Cifre og størrelse hver for sig

Når vi skal repræsentere et komma-tal med et vist antal betydende cifre kan vi derfor gøre det ved at tage cifrene for sig og størrelsen for sig.

Hvis vi f.eks. skal repræsentere 24312 med tre cifres nøjagtighed, kan vi tage cifrene 243 og huske at der skal sættes to nuller efter - at de er hundreder.

Hvis vi derimod skulle repræsentere f.eks. 0.0024312, ligeledes med tre cifres nøjagtighed, ville vi igen tage cifrene 243 men denne gang huske, at det er hundredetusindedele eller at der skal sættes et komma med to nuller foran.

Normalisere

Det er det samme princip der anvendes til repræsentere komma-tal i Java[2] (og stort set alle andre sprog). Forskellen er at man normaliserer tallet så kommaet står efter det første ciffer, der ikke er 0. Hvis vi igen tager 24312, ville man gemme 2.43 og huske at det skal ganges med 10000. At det skal ganges med 10000 huskes mere præcist som 4, idet $10000 = 10^4$.

De cifre man husker, 243, kaldes **mantissa**'en og antallet af nuller der sættes efter, 4, kaldes **exponenten**. Formlen er som følger:

$$\text{fortegn} * \text{mantissa} * 10^{\text{exponent}}$$

Som man ser ganges der også et fortegn på.

Mantissa'en er repræsenteret binært, men hvis vi holder os til, at den er den nøjagtighed der er angivet for typerne ovenfor, vil det lette forståelse af den følgende diskussion af **`float`** og **`doubles`** matematiske begrænsninger ifht. de reelle tal.

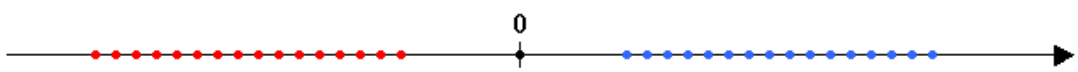
Ikke kontinuert

Den begrænsning vi skal se på, er at de reelle tal er kontinuerte - det er repræsentationen ikke. For de reelle tal gælder, at tager man to vilkårlige tal findes der altid et tal mellem dem (faktisk findes der *uendelig* mange reelle tal mellem dem). Det er den egenskab som repræsentationen mangler.

Større gab omkring 0

Lad os for nemheds skyld sige, at vi ikke kan have en exponent mindre end -5 og mantissa'en kan huske 3 cifre. Det mindste positive tal vi kan repræsentere bliver 0.0000**100**; hvor de tre sidste cifre er mantissa'en. Mellem 0 og dette tal findes der ikke andre. Det næste tal væk fra 0 bliver 0.0000**101** og det næste 0.0000**102**. Man observerer, at der mellem 0 og det nærmeste tal er et større mellemrum end mellem de andre tal.

Man kan illustrere dette gitter af punkter med en tallinie:



Figur 2: **`double`** *og* **`float`***s dækning af tallinien*

Denne figur tydeliggør repræsentationens manglende kontinuitet.

4. Tegn

4.1 Værdimængde

Tegn er tal

Alt i en computer er tal. Derfor er tegn også repræsenteret som tal - de har hver et nummer. A har nummer 65, + nummer 43, q nummer 113 osv. Java anvender UniCode standarden for hvilke tegn der har hvilke numre. Det giver mulighed for op til 65536 forskellige tegn.

Når man programmerer i Java, får man aldrig brug for at slå et tegns nummer op i en tabel. Det skyldes at man i stedet kan, og bør, anvende tegn-literaler. Et tegn-literale består af tegnet selv med appostrofer omkring. F.eks.:

```
System.out.println( 'T' );
```

```
T
```

I Java findes typen **`char`**, som bruges til tegn-variable. Hver **`char`** kan kun indeholde ét tegn. F.eks.:

```
char tegn;
tegn = 'A';
System.out.println( tegn );
```

```
A
```

4.2 Operatorer

Eftersom tegn i Java er numre, betyder det at de kan behandles fuldstændig som tal. F.eks.:

```
char tegn = 'A';
tegn++;
System.out.println( tegn );
```

```
B
```

Det er dog kun i specielle situationer man direkte behandler tegn som tal. Indirekte anvendes de ofte. Det er det, det efterfølgende afsnit om tekststrenge giver os et lille indblik i.

4.3 Implementation

ASCII er en del af UniCode

Den interne repræsentation for et tegn er på 16 bits; hvilket giver de 65536 muligheder. Tidligere brugte man **ASCII** standarden, med 7 bits og 128 tegn, men med internationaliseringen er det naturligvis helt utilstrækkeligt - tænk blot på de kinesiske skrifttegn! ASCII's 128 tegn er de samme som de første 128 i UniCode.

5. Tekststrenge

De typer vi hidtil har set, har været **primitive type**. Primitive typer "udemærker" sig ved, at deres værdimængde *ikke* består af **objekter**. Vi skal senere høre meget mere om objekter.

Ikke primitiv type

Tekst repræsenteres ikke med en primitiv type, men med et objekt. Det betyder at flere ting er meget anderledes. Egentlig burde vi vente med at se på tekst til senere, men da det er en nødvendig ting i mange programmer vil vi alligevel vove det ene øje, men kun lidt. Vi vil ikke gå i dybden, men kun se på nogle enkle anvendelser.

Navnet på typen er **`String`**, og variable af denne type kaldes i almindelighed **tekststrenge**. F.eks.:

```
String tekst;

tekst = "Dette er en tekst";
System.out.println( tekst );
```

```
Dette er en tekst
```

Variablen **`tekst`** får den værdi, der er angivet med tekst-literalet.

5.1 Klistre-plus

En af operanderne skal være en tekststreng

Når talen falder på typer er klistre-plus interessant. Det skyldes at dens operander kan være af forskellig type, men resultatet altid er en tekststreng. Det kræves at mindst en af operanderne er en tekststreng, mens den anden operand kan være af en hvilken som helst primitiv type.

Lad os se nogle eksempler:

```
class TestKlistrePlus {
  public static void main( String[] argv ) {
    int tal=10;

    System.out.println( "Tal: " + 5 );
    System.out.println( 5 + " er et tal" );
    System.out.println( "Integer-variabel: " + tal );
    System.out.println( "Tegn: " + 'A' );
    System.out.println( 'T' + "ekst" );
  }
}
```

```
Tal: 5
5 er et tal
Integer-variabel: 10
Tegn: A
Tekst
```

Som det ses er det uden betydning om tekststrengen står til højre eller venstre for +, det bliver regnet for et klistre-plus i begge tilfælde.

6. Kompatibilitet

Lad os tage udgangspunkt i et eksempel:

```
byte  b = 10;
```

```
short s = 10;
int   i = 10;
long  l = 10;
```

Den første reaktion på eksemplet kunne måske være: "Hvad interessant er der ved det?". Det interessante er faktisk at det compilerer uden fejl! Lad os se et andet eksempel der ikke gør:

```
byte  b = 1000;
short s = 1000;
int   i = 1000;
long  l = 1000;
```

```
--------------------------- Compiler Output ---------------------------
test.java:1: Incompatible type for declaration.
Explicit cast needed to convert int to byte.
    byte  b = 1000;
          ^
1 error
```

Problemet ligger i spørgsmålet om hvilken type 1000 har! 10 kan uden problemer passe ind i en byte, men det kan 1000 ikke, den er betydelig større end den maksimale værdi på 127.

Svaret er, at ethvert heltals-literale er en **`int`**.

Lad os se endnu et eksempel:

```
int tal=10;

byte  b = tal;
short s = tal;
int   i = tal;
long  l = tal;
```

```
--------------------------- Compiler Output ---------------------------
test.java:3: Incompatible type for declaration.
Explicit cast needed to convert int to byte.
    byte  b = tal;
          ^
test.java:4: Incompatible type for declaration.
Explicit cast needed to convert int to short.
    short s = tal;
          ^
2 errors
```

Før var der ingen problemer, da vi brugte 10, men nu går det helt galt. Hvad er meningen?

Compileren giver hurtigt op

Nok er 10 begge steder en **`int`**, men i det første eksempel kan compileren *direkte* se, at det vil gå godt, at 10 kan være i alle typerne. Derfor laver den selv 10 om til en **`byte`** henholdsvis en **`short`**. I det andet eksempel kræver det lidt mere analyse at nå frem til det samme resultat, en analyse som compileren *ikke* udfører. Dvs. at compileren nok vil hjælpe os, men kun lidt.

6.1 Casting

Vi kan godt se det

Hvad gør vi så? For *vi* kan godt se, at det vil aldrig vil gå galt! Hvordan overbeviser vi compileren om, at den skal lave det om til en **`byte`** henholdsvis en **`short`**.

Man anvender noget der hedder **casting**. Lad os se eksemplet igen, med casting:

```
int tal=10;

byte  b = (byte) tal;
short s = (short) tal;
```

```
int   i = tal;
long  l = tal;
```

Nu er der ingen fejl, men hvad gør forskellen?

Konvertering

Typeangivelserne i parantes før variabelen **`tal`** angiver at det efterfølgende skal ændres til den anførte type. Denne ændring sker først når programmet kører og højresiden evalueres. Man kalder sådanne ændringer af en værdi fra en type til en anden for **konvertering**.

Det er grimt

Casting ser grimt ud. Denne parantes der flagrer foran en variabel ser "forkert" ud. Paranteserne får det til at se ud som noget der holder effekten indenfor, mens det modsatte er tilfældet.

Binder som fortegn

Casting er en unær operator, og dens præcedens er på samme niveau som fortegn. Det betyder at den binder meget stært! Stærkere en man umiddelbart får indtryk af ud fra syntaksen. Lad os se et eksempel:

```
int tal=10;

byte b = (byte) tal + 5;
```

```
---------------------------- Compiler Output ----------------------------
test.java:3: Incompatible type for declaration.
Explicit cast needed to convert int to byte.
    byte b = (byte) tal + 5;
          ^
1 error
```

Casting stærkere end +

Hvorfor er compileren ikke tilfreds? Vi caster da højresiden af assignment til **`byte`**? Nej, vi caster kun **`tal`**. Det skyldes at castings præcedens gør den stærkere end +. Derved bliver højresiden summen af en **`byte`** og en **`int`**. Når man udfører en aritmetisk operation på to numeriske operander bliver resultatet den af de to operanders typer med størst værdimængde, dog mindst en **`int`**. I dette eksempel bliver højresiden derfor en **`int`**. Hvis vi vil caste hele højresiden til **`byte`**, må vi i stedet bruge paranteser:

```
int tal=10;

byte b = (byte)(tal + 5);
```

Nu vil hele højresiden blive castet til **`byte`** og compileren er tilfreds.

6.2 Kompatibilitet mellem primitive typer

Betragt følgende:

```
int tal=10;

long l = tal;
```

Her er compileren tilfreds, men hvorfor? Typerne er ikke de samme!

Sikker konvertering

Vi kan anlysere os frem til at det vil gå godt, men så langt analyserer compileren netop ikke, så hvordan kan den alligevel lade dette passere? Det kan den fordi den kan lave en sikker typekonvertering.

Delmængde

Værdimængden for typen **`int`** er en delmængde af værdimængden for typen **`long`**, og i sådanne situationer caster compileren for os.

Hvad så med heltal og komma-tal?

Lad os se et eksempel:

```
int tal=10;

double d = tal;
     tal = d;
```

```
--------------------------- Compiler Output ---------------------------
test.java:3: Incompatible type for =.
Explicit cast needed to convert double to int.
    tal = d;
          ^
1 error
```

Her er det først i tredie linie det går galt.

Fra heltal til komma-tal

Compileren caster fra ethvert heltal til komma-tal, idet kommatals-typerne på sin vis kan indeholde tal, der er større end den største heltals-type. Den anden vej vil den ikke, da kommatals-typerne kan repræsentere større tal end heltals-typerne.

I denne situation kan vi selv anføre casting, da vi i eksemplet kan se det vil gå godt:

```
int tal=10;

double d = tal;
     tal = (int) d;
```

Mellem **`double`** og **`float`** regnes **`float`**'s værdimængde som værende indeholdt i **`double`**'s, og konverteringen går derfor automatisk den ene vej.

Hvilken type har et kommatals-literale? Det er en **`double`**.

Indtil man bliver fortrolig med reglerne om konvertering, kan det være bekvemt at støtte sig til følgende skema, der angiver konvertering mellem de to mest anvendte primitive typer: **`int`** og **`double`**. Vi har valgt at tage typen **`String`** med i skemaet, da dens anvendelse er udbredt, selv i simple programmer. Man skal dog være opmærksom på, at en klar forståelse af **`String`** kræver kendskab til objektorienteret programmering.

Tabel 5: *Konverterings-tabel*

Fra: / Til:	`int i`	`double d`	`String s`
`int`	`i`	`(int) d`	`Integer.parseInt( s );`
`double`	`i`	`d`	`Double.parseDouble( s );`
`String`	`""+i`	`""+d`	`new String( s )`

6.3 Typeangivelser på literaler

Ikke kun `int` og `double`

Som nævnt vil et heltals-literale være en **`int`**, og et kommatals-literale en **`double`**. Når man angiver et numerisk literale har man dog mulighed for at anføre en bestemt type, som man ønsker det fortolkes som. På den måde behøver man ikke lave casting, da literalet i sig selv har den anførte type. Angivelsen er et suffix bestående af et enkelt tegn. F.eks.:

```
long   l = 106320142532221234L;
double d = 3.14D;
float  f = 3.14F;
```

Disse angivelser findes ikke for **`byte`** og **`short`**.

6.4 Casting-fejl

Hvad sker der når casting går galt? Når vi har sagt til compileren, at den skal caste, men værdien falder uden for værdimængden?

Lad os se et eksempel:

```
int tal=1000;

byte b = (byte) tal;

System.out.println( b );
```

```
-24
```

Logisk fejl

Dette er en logisk fejl, der opstår ved konvertering af den interne repræsentation. **`int`**'en bliver ganske enkelt klippet i stykker så den passer i **`byte`**'n, uden nogen form for kontrol.

7. Egne typer

Egne typer laves med klasser

I Java kan vi lave vores egne typer. Vi vil dog vente med det til senere, da det kræver, at man har lært objektorienteret programmering med klasser.

fodnoter

1 Hvis man skulle undgå at bruge syntaktisk sukker kunne man i stedet bruge **`b = (byte)(b+(byte)1);`**, da man ellers får en type-fejl. Da casting først introduceres senere i kapitlet ville det være uhensigtsmæssigt at løfte sløret for det, på det pågældende sted i kapitlet.

2 I Java anvendes IEEE 754 standarden. Ud fra denne standard for komma-tal er **`float`** og **`double`** konstrueret. De to typer har flg. opdeling af de 32 henholdsvis 64 bits.

Tabel 6: *Repræsentation af kommatals-typer*

Type	Exponent	Mantissa	Fortegn
`float`	8 bits	23 bits	1 bit
`double`	11 bits	52 bits	1 bit

Exponenten er i 2. komplement.

Sammenhængen mellem antal decimalers nøjagtighed og antallet af bits, kan beskrives med følgende ligning:

$$2^b = 10^d$$

hvor b er antallet af binære cifre og d er det tilsvarende antal decimale cifre. Man får:

$$d = b \log_{10} 2 = 0.3b$$

For **`float`** bliver det med 23 bits til 6.9 decimale cifre i mantissa'en (rundet op til 7, da det er meget tæt på). For **`double`** bliver det tilsvarende med 52 bits til 15.6 decimale cifre (rundet ned til 15).

For exponenterne kan **`float`** klare binært +/-127 svarende til +/-38.4 decimalt (afrundet til 38), og **`double`** binært +/-1023 svarende til 307.2 (afrundet til 307).

Repetitionsspørgsmål

1. Hvad er en type givet ved?
2. Hvad er værdimængden?
3. Hvad er en operatorer?
4. Hvad er en operand?
5. Hvad er en unær henholdsvis en binær operator?
6. Hvordan er værdimængden for heltal fordelt?
7. Hvad er specielt ved heltals-division?
8. Hvordan fungerer modulus på negative tal?
9. Hvad er overflow?
10. Hvad forsøger man at repræsentere med komma-tal?
11. Hvad er det mest betydende ciffer?
12. Hvad er problemet med **`float`**s nøjagtighed?
13. Hvordan repræsenterer man komma-tal?
14. Hvor i ligger repræsentationens problemer mht. kontinuitet?
15. Hvordan repræsenteres tegn?
16. Hvad hedder den type der repræsenterer et tegn?
17. Er der stor forskel på ASCII og UniCode?
18. Hvad er en primitiv type?
19. Hvornår er et + et klistre-plus?
20. Hvad er casting?
21. Hvad er castings præcedens?
22. Hvis en aritmetisk operator opererer på to numeriske operander af forskellig type; hvad bliver så den resulterende type?
23. Hvornår er typekonvertering sikker mellem heltals-typer?

24 Hvilken vej er konvertering mellem heltal og komma-tal sikker?

25 Hvornår er typekonvertering sikker mellem komma-tal?

26 Hvordan kan man angive numeriske literaler, der ikke er **`int`** eller **`double`**?

27 Hvornår går casting galt?

28 Hvilken slags fejl er en casting-fejl?

Svar på repetitionsspørgsmål

1 En værdimængde og en række operatorer.

2 Den mængde af værdier som variable og udtryk af typen kan antage.

3 En operator tager en række operander og afbilder dem over i et resultat.

4 En værdi som en operator opererer på.

5 En unær operator tager én operand, en binær tager to.

6 Symmetrisk omkring nul.

7 Heltals-division er antallet af gange det ene tal går op i det andet. Der er derfor ikke noget efter kommaet.

8 Som hvis man tog modulus på tallet uden fortegn og efterfølgende satte det foran resultatet.

9 Når en variabel af en given type løber ud af denne types værdimængde.

10 De reelle tal.

11 Det ciffer der har den største positionelle værdi. Populært sagt: Den der står længst til venstre.

12 At den er for lille til penge-beløb.

13 Man opdeler i en mantissa, der indeholder de betydende cifre og en exponent der beskriver størrelsen.

14 At den ikke er kontinuert. Mao. at der findes tal-par der ikke har noget tal mellem sig.

15 Som tal. Værdimængden går fra 0 til 65535.

16 **char**.

17 Både og, ASCII er de første 128 tegn af UniCode, men UniCode indeholder *betydelig* flere tegn end ASCII.

18 En type; hvis værdimængde ikke er objekter.

19 Når mindst en af operanderne er en tekststreng.

20 Når man konverterer en type til en anden.

21 Den samme som for fortegn.

22 Den af de to typer, der har den største værdimængde.

23 Når man konverterer til en type der har en større værdimængde end den man konverterer fra.

24 Fra heltal til komma-tal.

25 Fra float til double.

26 Der findes tre suffix' L, D og F som man kan sætte efter det numeriske literale. Literalet vil af compileren blive opfattet som værende af den type man har angivet. Tegnene står for **`long`**, **`double`** og **`float`** (D er naturligvis overflødig).

27 Når værdien falder uden for den lovede types værdimængde.

28 En logisk fejl.