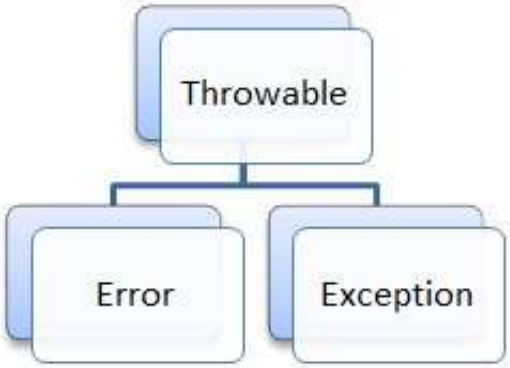
Algoritmer og datastrukturer
D – Feilhåndtering og unntaksklasser

D Feilhåndtering og unntaksklasser

D.1 Klassehierarki for Throwable



Figur D.1 a) : Hierarki for Throwable

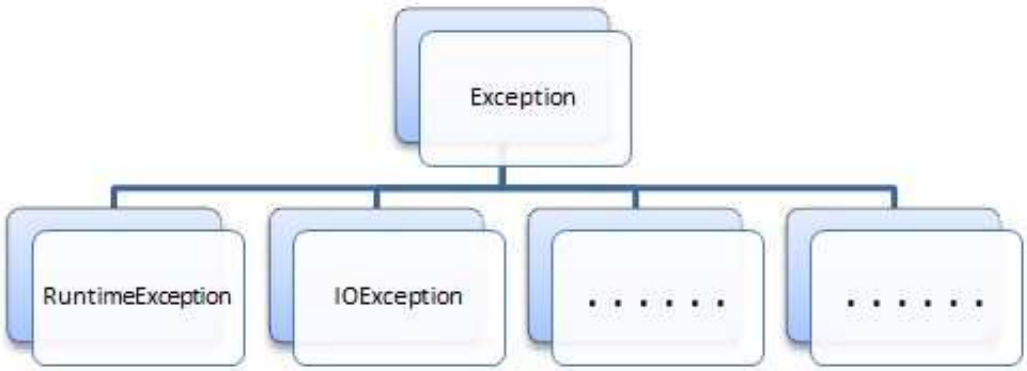
Klassen **`Throwable`** er basisklasse (eng: superclass) for alle feil og unntak i Java. Det er bare instanser av denne klassen eller av en av dens subklasser, som kan kastes (*throw*) eller bli fanget opp (*try – catch*).

Instanser av subklassene til **`Error`** og **`Exception`** brukes til å fortelle at en eksepsjonell situasjon har oppstått. På engelsk heter det: «Typically, these instances are freshly created in the context of the exceptional situation so as to include relevant information (such as stack trace data)». «Stack trace data» kommer i form av et såkalt «øyeblikksbilde» (eng: snapshot) av programstakken når unntaket skapes. Her er et eksempel:

```
Exception in thread "main" java.lang.NegativeArraySizeException
  at algdat.Program.randPerm(Program.java:78)
  at algdat.Program.main(Program.java:133)
```

Første linje i «øyeblikksbildet» inneholder unntakstypen og en eventuell melding. Neste linje sier hvor (linjenummer) og i hvilken metode unntaket ble skapt. Hvis metoden har blitt kalt av en annen metode, så kommer den. Osv. til vi kommer til *main*.

Klassen **`Exception`** har flere direkte subklasser. De to som vi oftest kommer i kontakt med er **`RuntimeException`** og **`IOException`**.



Figur D.1 b) : RuntimeException og IOException er subklasser til Exception

Et unntak (exception) er enten sjekket (eng: checked) eller usjekket (eng: unchecked). Et sjekket unntak er et som kompilatoren sjekker for å se om koden/programmet behandler det på en godkjent måte. Det betyr at det enten må være fanget opp eller være sendt videre. Alle unntak som ligger under **`Exception`** i arvehierarkiet (dvs. er en subklasse til **`Exception`**), bortsett fra **`RuntimeException`** og subklassene til **`RuntimeException`**, er sjekkede unntak.

Eksempel: Vi skal lage en metode som returnerer et Reader-objekt knyttet til en tekstfil der filnavnet er gitt ved en tegnstreng:

```
public static Reader åpneFil(String filnavn)
{
  return new FileReader(filnavn);
}
```

Programkode* D.1 *a)

Programkode D.1 *a)* slik som den er nå, vil ikke la seg kompilere. Feilmeldingen kan f.eks. være: «*Unsupported exception FileNotFoundException; must be caught or declared to be thrown*». Det skyldes at konstruktøren til **`FileReader`** kaster en **`FileNotFoundException`** og den er en subklasse til **`IOException`**. Med andre ord er dette et sjekket unntak.

Hvis vi bruker et programmeringsmiljø, vil vi i tillegg til en feilmelding også få forslag til hvordan dette skal repareres. F.eks. vil *NetBeans* komme med to forslag:

1. Add throws clause for FileNotFoundException
2. Surround Statement with try - catch

Hvilket forslag bør vi velge? En regel sier at man skal fange opp et sjekket unntak på laveste nivå hvor det er mulig å behandle unntaket på en meningsfull måte. Her kunne vi si at det ikke er mulig å gjøre noe fornuftig med unntaket og at det derfor bør sendes «oppover»:

```
public static Reader åpneFil(String filnavn) throws FileNotFoundException
{
  return new FileReader(filnavn);
}
```

Programkode* D.1 *b)

Alternativt kunne vi signalisere ved å bruke *null* som returverdi, at filnavnet må være feil:

```
public static Reader åpneFil(String filnavn)
{
  try
  {
    return new FileReader(filnavn);
  }
  catch (FileNotFoundException ex)
  {
    return null;
  }
}
```

Programkode* D.1 *c)

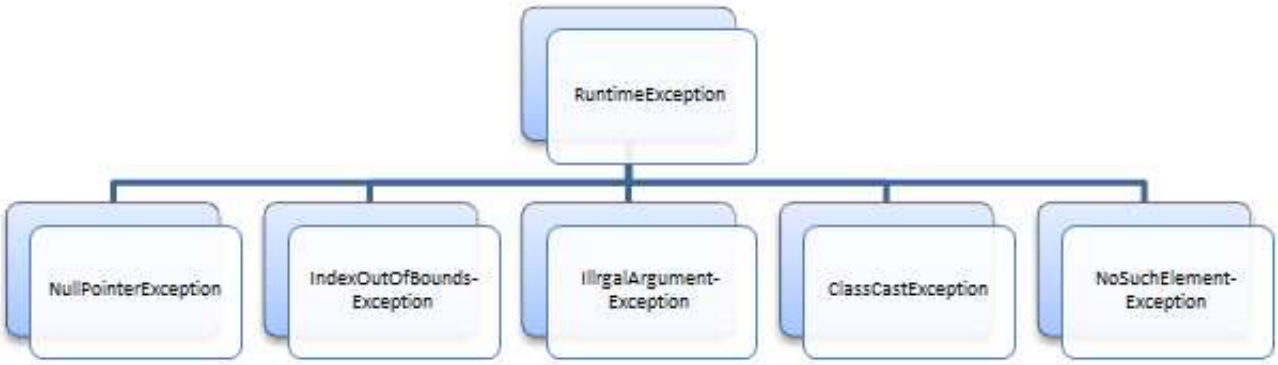
Denne siste versjonen er best hvis metoden skal inngå i et «brukerstyrt» program. Hvis metoden returnerer *null*, må «brukeren» informeres om at det ikke finnes noen fil med det filnavnet som ble oppgitt.

Konklusjon: Hvis en programsetning (konstruktør-, metodekall eller generelt en operasjon) kaster et **sjekket** unntak, skal unntaket enten fanges opp (og behandles) i en **`try - catch`** eller sendes «oppover» (**`throws`**). En grunnregel er at hvis det skal fanges opp, skal det skje på det laveste nivå hvor det er mulig å behandle det på en meningsfull måte.

D.2 RuntimeException

Sjekkede unntak handler i hovedsak om slike ting som programmereren ikke kan kontrollere. Derfor må de alltid behandles, dvs. at de må enten fanges opp eller de må bli sendt videre «oppover». Man fanger dem opp hvis de kan behandles på en meningsfull måte. Hvis ikke, sendes de «oppover».

Et unntak av typen **`RuntimeException`** eller en subklasse til **`RuntimeException`**, er et **usjekket** unntak. Slike unntak skyldes oftest feil eller svakheter i koden og skal derfor normalt **ikke** behandles. Dermed vil programmet stoppe og feilen kan bli identifisert og rettet opp.



Figur D.2 a) : Noen få av subklassene til RuntimeException

Figur D.2 *a)* inneholder noen vanlige unntak, f.eks. **`NullPointerException`**. Det er nok den som ferske programmerere får flest ganger. Flg. tabell inneholder flere av subklassene til **`RuntimeException`**. Også erfarne programmere kan få noen av dem:

Unntaksklasse	**Package**	**Årsak**
NullPointerException	java.lang	Bruker et objekt som ikke er opprettet.
IndexOutOfBoundsException	java.lang	Negativ eller for stor indeks i en liste.
ArrayIndexOutOfBoundsException	java.lang	Negativ eller for stor indeks i en tabell.
StringIndexOutOfBoundsException	java.lang	Negativ eller for stor indeks i en tegnstreng.
IllegalArgumentException	java.lang	Parameterverdi (eller argument) som ikke gir mening.
ClassCastException	java.lang	Konvertering til en ulovlig type.
UnsupportedOperationException	java.lang	Bruker en metode som ikke har fått kode.
ConcurrentModificationException	java.util	F.eks. en endring etter at en iterator har startet.
NoSuchElementException	java.util	Ønsker å få tak i noe som ikke finnes.

Figur D.2 b)

Det finnes massevis av unntaksklasser i Java. Hver mappe (package) har sine unntaksklasser. Bare noen få av subklassene til **`RuntimeException`** er satt opp i tabellen i *Figur* D.2 *b)*.

Oppgaver til Avsnitt D.2

1. Hvor mange feil- og unntaksklasser er det i pakken *java.lang*?
2. Java har lange navn på feil- og unntaksklasser. Poenget er at navnet på klassen skal gi mest mulig informasjon om årsaken til feilen eller unntaket. Hva er det lengste navnet du

kan finne i java-pakkene?

D.3 Når skal vi kaste unntak?

De eksepsjonelle situasjonene i koden vår som fører til at det kastes et unntak av typen **`RuntimeException`**, er det Java som tar seg av. Da er det javamaskinen (JVM) som implisitt sørger for at det kastes rett unntak. Er det da behov for at vi eksplisitt kaster unntak, dvs, bruker *throw*? Ja, f.eks. av flg. årsaker:

- Det er ønskelig med en bedre beskrivelse av årsaken til feilen.
- Det er ønskelig å bruke en annen unntaksklasse enn den Java implisitt genererer.
- Det kan være situasjoner med en logisk feil uten at det er en Java-teknisk feil.
- Det er ønskelig med en unntaksklasse med et navn etter eget ønske.

Eksempel 1 To tegnstrenger *a* og *b* kan sammenlignes ved å bruke *a.compareTo(b)*. I en anvendelse viser det seg at det er fordelaktig å ha en sammenligningsmetode der begge strengene er parametre. Det kan gjøres slik:

```
public static int compare(String a, String b)
{
  return a.compareTo(b);
}
```

Hvis denne metoden blir kalt med verdier *a* og *b* der en av dem eller begge er *null*, vil det bli kastet en **`NullPointerException`**. Men det kommer ingen melding om hvem av dem som var *null*. For å få til det må vi selv kaste unntak:

```
public static int compare(String a, String b)
{
  if (a == null || b == null)
  {
    String melding;

    if (a == null && b == null) melding = "både a og b er null";
    else if (a == null) melding = "a er null";
    else melding = "b er null";

    throw new NullPointerException(melding);
  }

  return a.compareTo(b);
}
```

Eksempel 2 En metode skal til et gitt heltall fra 1 - 7, gi oss navnet på tilsvarende ukedag. Dvs. 1 skal gi oss MANDAG, 2 skal gi oss TIRSDAG, osv. Dette kan vi løse på flg. måte:

```
public static String ukedag(int n)
{
  String[] dag = {"MAN","TIRS","ONS","TORS","FRE","LØR","SØN"};

  return dag[n-1] + "DAG";
}
```

En parameterverdi *n* som ikke er fra 1 til 7, vil gi en **`ArrayIndexOutOfBoundsException`** siden $n - 1$ da blir ulovlig indeks i tabellen *dag*. Men her vil det være mer naturlig å fortelle at verdien *n* er ulovlig og samtidig fortelle hvilken verdi det var. Det kan gjøres slik:

```
public static String ukedag(int n)
{
  if (n < 1 || n > 7) throw new IllegalArgumentException
    ("Ulovlig dag(" + n + ") - må være 1 - 7");

  String[] dag = {"MAN","TIRS","ONS","TORS","FRE","LØR","SØN"};

  return dag[n-1] + "DAG";
}
```

I dette eksemplet kan vi gå et skritt videre. Den javatekniske årsaken (eng: cause) til at det blir et problem, er som nevnt over, at en verdi på *n* utenfor 1 - 7 vil gi ulovlge tabellindekser for tabellen *dag*. Det er mulig også å fortelle om det når unntaket kastes:

```
public static String ukedag(int n)
{
  if (n < 1 || n > 7)
  {
    Exception ex = new ArrayIndexOutOfBoundsException(n - 1);

    throw new IllegalArgumentException
      ("Ulovlig dag(" + n + ") - må være 1 - 7", ex);
  }

  String[] dag = {"MAN","TIRS","ONS","TORS","FRE","LØR","SØN"};

  return dag[n-1] + "DAG";
}
```

Hvis vi kjører flg. program, kommer det en lang melding. Hvis du kjører programmet, vil du få samme typen melding, men med dine egne navn på prosjekt, pakke og linjenummer:

```
public class Program
{
  public static String ukedag(int n)
  {
    // kode som den over
  }

  public static void main(String[] args)
  {
    String ukedag = ukedag(0);
  }
}
```

Det kastes en **`IllegalArgumentException`** med en **`ArrayIndexOutOfBoundsException`** som årsak (eng: cause):

```
Exception in thread "main"
  java.lang.IllegalArgumentException: Ulovlig dag(0) - må være 1 - 7
    at algdat.Program.ukedag(Program.java:13)
    at algdat.Program.main(Program.java:24)
  Caused by: java.lang.ArrayIndexOutOfBoundsException:
    Array index out of range: -1
    at algdat.Program.ukedag(Program.java:9)
    ... 1 more
```

Det er ikke alle unntaksklasser som har muligheten til å ha med en årsak. For eksempel har **`IllegalArgumentException`** det, men ikke **`ArrayIndexOutOfBoundsException`**. Dette finner en ut ved å se hvilke konstruktører en unntaksklasse har.

Eksempel 3 Klassen **`Rektangel`** skal ha *lengde* og *bredde* som instansvariabler:

```
public class Rektangel
{
  private int lengde, bredde;

  public Rektangel(int lengde, int bredde)
  {
    this.lengde = lengde;
    this.bredde = bredde;
  }

  public int omkrets()
  {
    return 2 * (lengde + bredde);
  }

  // Øvrige metoder
}
```

Et problem med koden over er at det er ingenting som hindrer oss fra å opprette et rektangel der lengde eller bredde er negative. I vårt tilfelle er det en logisk feil, men ingen feil som fører til at Java kaster et unntak. Vi ser også at omkretsen kan bli positiv. Dermed er det ikke sikkert vi vil oppdage feilen:

```
Rektangel r1 = new Rektangel(10,5);   // et ok rektangel
Rektangel r2 = new Rektangel(10,-5);  // ulovlig, men det kastes ikke et unntak

System.out.println(r1.omkrets());     // Utskrift: 30
System.out.println(r2.omkrets());     // Utskrift: 10
```

Det vi må gjøre er å kaste et unntak i konstruktøren. F.eks. slik:

```
public Rektangel(int lengde, int bredde)
{
  if (lengde < 0) throw new
    IllegalArgumentException("lengde(" + lengde + ") er negativ");

  if (bredde < 0) throw new
    IllegalArgumentException("bredde(" + bredde + ") er negativ");

  this.lengde = lengde;
  this.bredde = bredde;
}
```

Oppgaver til Avsnitt D.3

1. Alle unntaksklassene har minst to konstruktører - en standardkonstruktør og en som har en tegnstreng (en melding) som parameter. Noen klasser har også mulighet for å få med en årsak. De har ytterligere to konstruktører - en med en **`Throwable`** som parameter og en med både en tegnstreng og en **`Throwable`**. Se f.eks. **`IllegalArgumentException`**. Finn flere unntaksklasser av denne typen!
2. Hva slags konstruktører har **`ArrayIndexOutOfBoundsException`**?

D.4 Hvordan lage egne unntaksklasser?

I *Avsnitt* 1.1.7 bestemte vi at metoden *maks* som finner posisjonen til den største verdien i en heltallstabell, skal kaste en **`NoSuchElementException`** hvis tabellen er tom. Vi kan imidlertid lage vår egen unntaksklasse for dette tilfellet, f.eks. med navn **`TomTabellUnntak`**. Klassen kan vi legge under mappen (pakken) *hjelpeklasser*:

```
public class TomTabellUnntak extends RuntimeException
{
  public TomTabellUnntak()
  {
    super();   // kaller basisklassens konstruktør
  }

  public TomTabellUnntak(String melding)
  {
    super(melding);  // kaller basisklassens konstruktør
  }
}
```

Hvis klassen **`TomTabellUnntak`** er lagt under mappen (pakken) **`hjelpeklasser`**, kan metoden *maks* i *Programkode* 1.1.4 kodes slik:

```
public static int maks(int[] a)
{
  if (a.length < 1)
    throw new hjelpeklasser.TomTabellUnntak("a er tom");

  int m = 0;                // indeks til største verdi
  int maksverdi = a[0];     // største verdi

  for (int i = 1; i < a.length; i++) if (a[i] > maksverdi)
  {
    maksverdi = a[i];     // største verdi oppdateres
    m = i;                // indeks til største verdi oppdateres
  }
  return m;   // returnerer indeks/posisjonen til største verdi

} // maks
```

Hvis flg. programbit kjøres, vil vi få en feilmelding av flg. type:

```
int[] a = {};  // en tom tabell
int m = maks(a);
```

```
Exception in thread "main" hjelpeklasser.TomTabellUnntak: a er tom
  at algdat.Program.maks(Program.java:10)
  at algdat.Program.main(Program.java:29)
```

Oppgaver til Avsnitt D.4

1. Lag unntaksklassen **`UlovligTilstandUnntak`** som subklasse til **`RuntimeException`**. La den få fire konstruktører: 1) standard, 2) en tegnstreng (en melding) som parameter, 3) en årsak (**`Throwable`**) som parameter og 4) en tegnstreng (en melding) og en årsak (**`Throwable`**) som parametre.

D.5 Misbruk av unntak

Unntak av typen **`RuntimeException`** har som formål å rapportere om eksepsjonelle forhold. Annen bruk vil i de fleste tilfeller bli sett på som misbruk. Se flg. eksempel:

I *maks*-metoden i *Programkode* 1.1.5 ble det brukt en teknikk med en «vaktpost» for å redusere arbeidet i en for-løkke. Det gjorde at sammenligningen **`i < a.length`** kunne fjernes. Det samme kan vi få til ved å fange opp det unntaket vi vil komme til å få når *i* blir lik tabellens lengde. Dette signaliserer da at vi har vært gjennom hele tabellen. Koden blir slik:

```
public static int maks(int[] a)
{
  if (a.length < 1)
  {
    throw new java.util.NoSuchElementException("a er tom");
  }

  int m = 0;             // indeks til største verdi
  int maksverdi = a[0];  // største verdi

  try
  {
    for (int i = 1; ; i++) if (a[i] > maksverdi)
    {
      maksverdi = a[i];  // største verdi oppdateres
      m = i;             // indeks til største verdi oppdaters
    }
  }
  catch(ArrayIndexOutOfBoundsException e)
  {
    // ingenting
  }

  return m;  // returnerer indeks/posisjonen til største verdi

} // maks
```

Vi ser at for-løkken: **`for (int i = 1; ; i++)`** mangler **`i < a.length`**. Her er et unntak brukt for at vi ikke skal havne utenfor en tabell. Det er i strid med formålet til unntak og anses derfor som misbruk.

Denne spesielle versjonen av *maks*-metoden vil virke. Sjekk at f.eks. flg. kodebit virker:

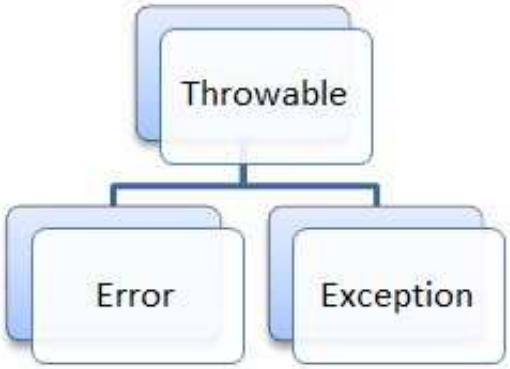
```
int[] a = {7,3,5,1,9,2,10,8,4,6};
System.out.println(a[maks(a)]);  // Utskrift: 10
```

Oppgaver til Avsnitt D.5

1. Sjekk at *maks*-metoden over virker.
2. Sammenlign effektiviteten til denne *maks*-metoden med de andre som ble laget i *Delkapittel* 1.1. Se *Avsnitt* 1.1.10.

D.6 Error – feil

En **Error** indikerer en alvorlig feil. I beskrivelsen til klassen står det: «An **Error** is a subclass of **Throwable** that indicates serious problems that a reasonable application should not try to catch.» **Error** og dens subklasser ses på som usjekkede unntak. De skal derfor, som sitatet også sier, normalt ikke fanges opp.



Figur D.6 a) : Hierarki for Throwable

Det sies at en **Error** skyldes en unormal (eng: abnormal) situasjon som egentlig ikke burde forekomme. Spørsmålet er da om vi vil oppleve at det kastes en **Error**.

Et tilfelle som mange uerfarne programmere vil kunne oppleve, er **StackOverflowError**. Et program starter med at *main*-metoden kjøres. Den inneholder normalt kall på andre metoder og disse har igjen kall på metoder, osv. Data knyttet til slike «nøstede» metodekall blir lagt på *programstakken* (eng: runtime stack). Til den er det avsatt en bestemt mengde plass som normalt er mer enn nok.

Men hvis en skal bruke en rekursiv metode, kan det gå galt. *Delkapittel* 1.5 handler om rekursjon og i *Avsnitt* 1.5.1 står det flere eksempler på rekursive metoder. De er ment som eksempler på selve ideen i rekursjon. Men for flere av dem er problemstillingen egentlig ikke egnet for en rekursiv løsning.

Et eksempel handlet om å finne summen av tallene fra 1 til *n*. Rekursiv idé: Kjenner vi summen av de *n* – 1 første tallene, får vi hele summen ved å legge til *n*. Det ble kodet slik:

```
public static int sum(int n)        // summen av tallene fra 1 til n
{
  if (n == 1) return 1;             // summen av 1 er lik 1
  return sum(n - 1) + n;            // summen av de  n - 1 første + n
}
```

Programkode **D.6** *a)*

Når denne metoden kjøres, vil det bli lagt *n* lag på programstakken siden metoden må kalles *n* ganger for at parameterverdien skal bli 1. Med andre ord vil dette gå greit hvis *n* ikke er stor. Men for en stor *n* får vi en «oversvømmelse». Hvis metoden kalles fra *main*, vil det komme: **Exception in thread "main" java.lang.StackOverflowError**. Det kommer også «stack trace data», dvs. en linje for hvert kall som ligger på stakken. Men heldigvis (i både Netbeans og Eclipse) stopper det etter 1024 linjer. Se *Oppgave* 1.

Det å finne summen av tallene fra 1 til *n* løses enklest ved hjelp av en formel. Summen er lik $n(n-1)/2$. Men generelt finner vi en sum av tall ved å legge sammen ett og ett, dvs. ved hjelp av en løkke (iterasjon). Det er imidlertid mulig å forbedre den rekursive teknikken slik at vi unngår at programstakken oversvømmes. Da kan vi bruke flg. idé: Hvis vi kjenner summen av både første og andre halvpart av tallene, får vi hele summen ved å addere de to. Hvis

«halvparten» består av kun ett tall, er summen lik tallet:

```
public static int sum(int m, int n)   // summen av tallene fra m til n
{
  if (m == n) return m;             // summen av ett tall er lik tallet
  int k = (m + n)/2;                // k er midt mellom m og n
  return sum(m,k) + sum(k+1,n);     // adderer de to delene
}

public static int sum(int n)        // summen av tallene fra 1 til n
{
  return sum(1,n);
}
```

Programkode D.6 *b)*

Vi kan bruke denne versjonen til å finne f.eks. summen av tallene fra 1 til 10000:

```
System.out.println(sum(10000));     // Utskrift: 50005000
```

Her får vi ingen problemer med **StackOverflowError**. Det kommer av at ved hvert rekursivt kall halveres tallmengden og antall lag på programstakken blir maksimalt lik $\log_2(n)$.

Vi får imdlertid et annet problem. Hvis vi skal finne summen av tallene fra 1 til n vil summen kunne bli for stor for datatypen **int**. Det går bra med n = 10000 som i eksemplet over. Men hva med f.eks. n = 100000? Se *Oppgave* 2.

Oppgaver til Avsnitt D.6

1. Hvor stor må n minst være for at et kall på metoden *sum* i *Programkode* D.6 *a)* skal gi **StackOverflowError**? Prøv deg frem! Start med n = 10000. Er den for stor eller for liten?
2. Hva er største verdi på n slik at summen av tallene fra 1 til n ikke blir for stor for datatypen **int**? Kast et unntak i den siste *sum*-metoden i *Programkode* D.6 *b)* hvis n er mindre enn 1 eller større enn denne største verdien.