Kapittel 13:
Unntakshåndtering

Redigert av:

Khalid Azim Mughal (`khalid@ii.uib.no`)

Kilde:

Java som første programmeringsspråk (3. utgave)

Khalid Azim Mughal, Torill Hamre, Rolf W. Rasmussen
Cappelen Akademisk Forlag, 2006.
ISBN: 82-02-24554-0
`http://www.ii.uib.no/~khalid/jfps3u/`
(NB! Boken dekker opptil Java 6, men notatene er oppdatert til Java 7.)

Emneoversikt

- Programutføring
- "kast og fang"-prinsippet
- Unntakspropagering
- Typiske scenarier ved bruk av `try-catch`-setning
- Utdrag av unntaksklasser
- Eksplisitt kast av unntak: `throw`-klausul
- `try`-blokk med flere `catch`-blokker
- `try`-blokk med felles `catch`-blokk
- Typiske programmeringsfeil ved unntakshåndtering
- Håndtering av kontrollerte unntak og `throws`-klausul
- Definere egne kontrollerte unntak
- Utføring av `finally`-blokk

Hva er et unntak?

- Et program må være i stand til å håndtere unormale situasjoner som kan oppstå under utføring.
- Et *unntak* (eng. *exception*) i Java signaliserer en feilsituasjon som oppstår under programutføring.
- Eksempler på feilsituasjoner:

1. Programmeringsfeil -- oppstår på grunn av logiske feil.
   - Ulovlig indeksering i en tabell.
   - Forsøk på å dividere med 0.
   - Metodekall med ulovlige parametere.
   - Bruk av `null`-referanse for å aksessere medlemmer i et objekt.
2. Kjørefeil -- som programmet har lite herredømme over.
   - Åpning av fil som ikke eksisterer.
   - Lese- eller skrivefeil ved bruk av fil.
   - En nettforbindelse som går ned.

Litt om programutføring

- En *programstabel* brukes til å håndtere utføring av metoder.
- Et element (kalt *stabelramme*) på programstabelen tilsvarer et metodekall.
  - Hvert metodekall fører til at det opprettes en ny stabelramme.
  - Stabelrammen inneholder diverse opplysninger, bl.a. om lagring av lokale variabler.
  - Når metoden er ferdig, fjernes dens ramme.
  - Metoden som har stabelrammen på toppen av programstabelen blir utført.
- Ved retur fra en metode fortsetter programutføringen i metoden som tilsvarer stabelrammen som nå er blitt avdekket på toppen av programstabelen.
- Under utføringen vil alle stabelrammer på programstabelen til enhver tid angi hvilke metoder som er *aktive*, det vil si ikke ferdig utførte.
  - Ved et gitt tidspunkt under utføringen, angir *programstabelutlistingen* hvilke metoder som er aktive.

Metodeutføring og unntakspropagering

- Vi skal bruke følgende problem til å illustrere metodeutføring og unntakspropagering.

  Beregne hastighet når avstand og tid er gitt.

  Programmet bruker tre metoder:

1. `main()`
2. `skrivHastighet()`
3. `beregnHastighet()`

- Merk heltallsdivisjon i beregning av uttrykket `(avstand/tid)`.
  – heltallsdivisjon med `0` er en *ulovlig* operasjon i Java

Metodeutføring (Program 13.1)

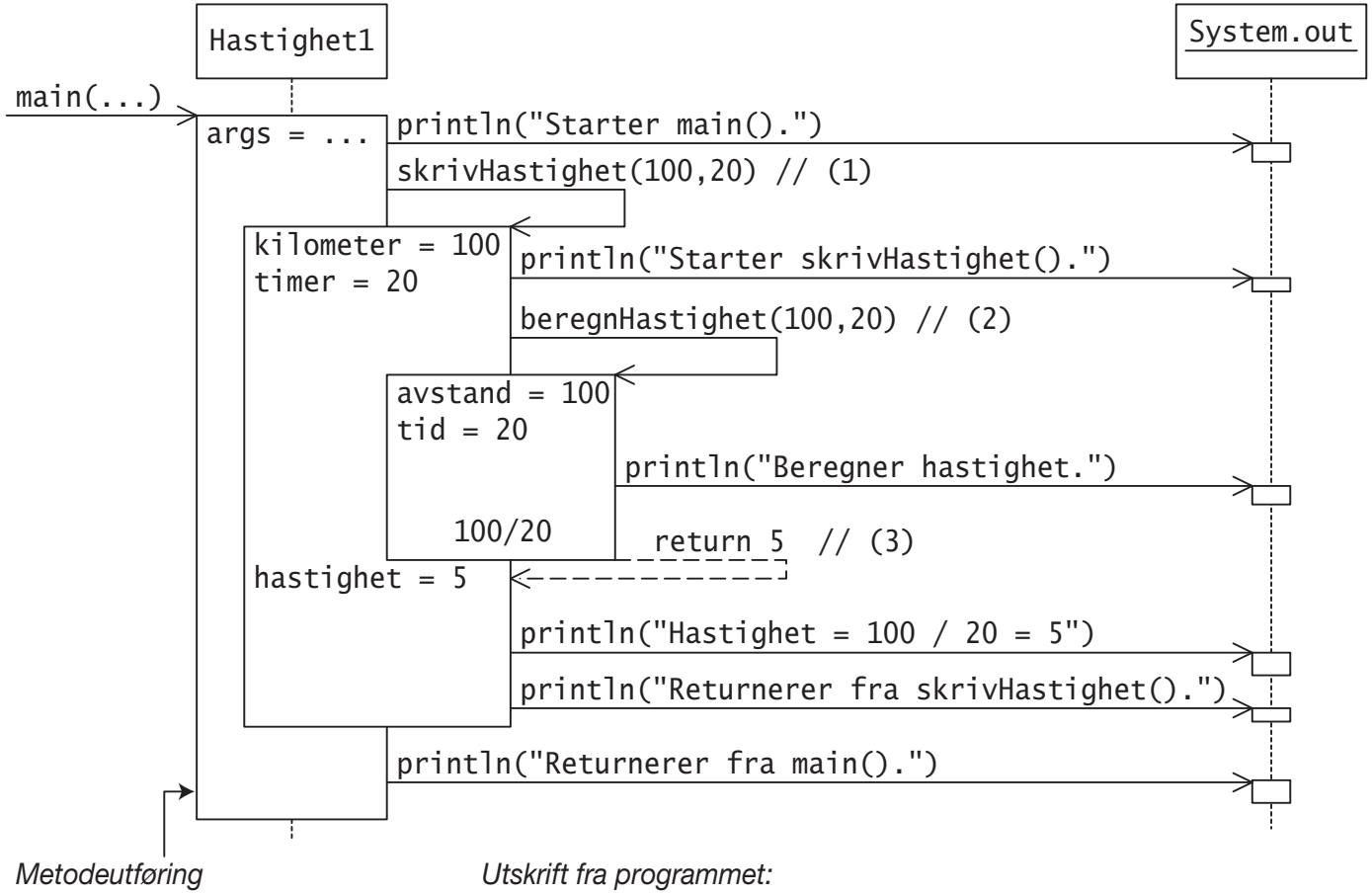
```
public class Hastighet1 {

  public static void main(String[] args) {
    System.out.println("Starter main().");
    skrivHastighet(100, 0);                                // (1)
    System.out.println("Returnerer fra main().");
  }

  private static void skrivHastighet(int kilometer, int timer) {
    System.out.println("Starter skrivHastighet().");
    int hastighet = beregnHastighet(kilometer, timer);    // (2)
    System.out.println("Hastighet = " +
                       kilometer + "/" + timer + " = " + hastighet);
    System.out.println("Returnerer fra skrivHastighet().");
  }

  private static int beregnHastighet(int avstand, int tid) {
    System.out.println("Beregner hastighet.");
    return avstand/tid;                                   // (3)
  }
}
```

Normal utføring (Figur 13.1)"kast og fang"-prinsippet



Utskrift fra programmet:

```
Starter main().
Starter skrivHastighet().
Beregner hastighet.
Hastighet = 100/20 = 5
Returnerer fra skrivHastighet().
Returnerer fra main().
```

- Et unntak *kastes* når en feilsituasjon oppstår under programutføring, og *fanges* av en *unntakshåndterer* som behandler det.
- Erstatt i Program 13.1:

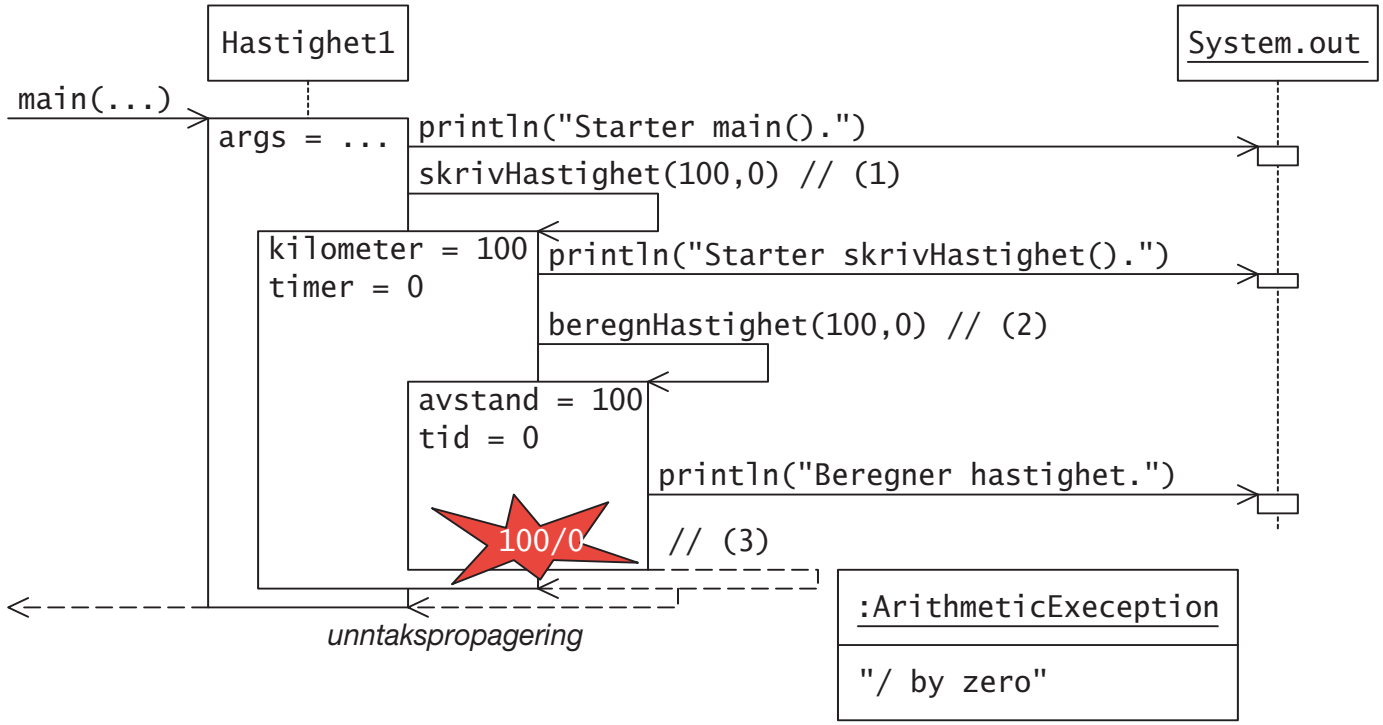
```
skrivHastighet(100, 20);                    // (1)
```

med

```
skrivHastighet(100, 0);                     // (1) Andre parameter er 0.
```

- Utføringen av programmet er illustrert i sekvensdiagrammet i Figur 13.2.
- I (7) *kastes* det et `ArithmeticException`-unntak, som formidles tilbake (*propageres*) gjennom stabelrammer på programstabelen.

Unntakspropagering (ved heltallsdivisjon med 0) (Figur 13.2)



Utskrift på terminal: *Utskrift fra programmet* *Utskrift fra standard unntakshåndterer*

```
Starter main().
Starter skrivHastighet().
Beregner hastighet.
Exception in thread "main" java.lang.ArithmeticException: / by zero
  at Hastighet1.beregnHastighet(Hastighet1.java:20)
  at Hastighet1.skrivHastighet(Hastighet1.java:12)
  at Hastighet1.main(Hastighet1.java:6)
```

klassenavn *metodenavn* *filnavn* *linjenummer i filen der kallet til neste metode forekommer*

Unntakspropagering

- Programutføringen fortsetter ikke på normal måte under unntakspropageringen, og unntaket sendes ikke videre med `return`-setningen.
- Unntaket tilbys til aktive metoder i tur og orden.
  - Dersom en aktiv metode ikke *fanger* unntaket, avbrytes utføringen av metoden, m.a.o. fjernes dens stabelramme fra programstabelen.
- Når unntaket har propagert til "toppnivå", blir det behandlet av en *standard unntakshåndterer* i Javas virtuelle maskin.
  - Standard unntakshåndtereren genererer en programstabelutlisting på terminalen.

Unntakshåndtering: try-catch

- En `try`-blokk kan inneholde vilkårlig kode, men hensikten er å legge inn setninger som man vet kan lede til at et unntak blir kastet under utføring.
- En `catch`-blokk utgjør en unntakshåndterer.
  - En `catch`-blokk hører sammen med en `try`-blokk.
  - Et unntak som kan bli kastet som resultat av utføring av koden i `try`-blokken, kan fanges og behandles i en tilhørende `catch`-blokk.

Typiske scenarier ved bruk av try-catch-setning

try-blokken inneholder kode som kan føre til at et unntak blir kastet.

try-blokk

catch-blokk

```
try {
  int hastighet = beregnHastighet(kilometer, timer);
  System.out.println("Hastighet = " +
                     kilometer + "/" + timer + " = " + hastighet);
}
catch (ArithmeticException unntak) {
  System.out.println(unntak + " (behandlet i skrivHastighet())");
}
```

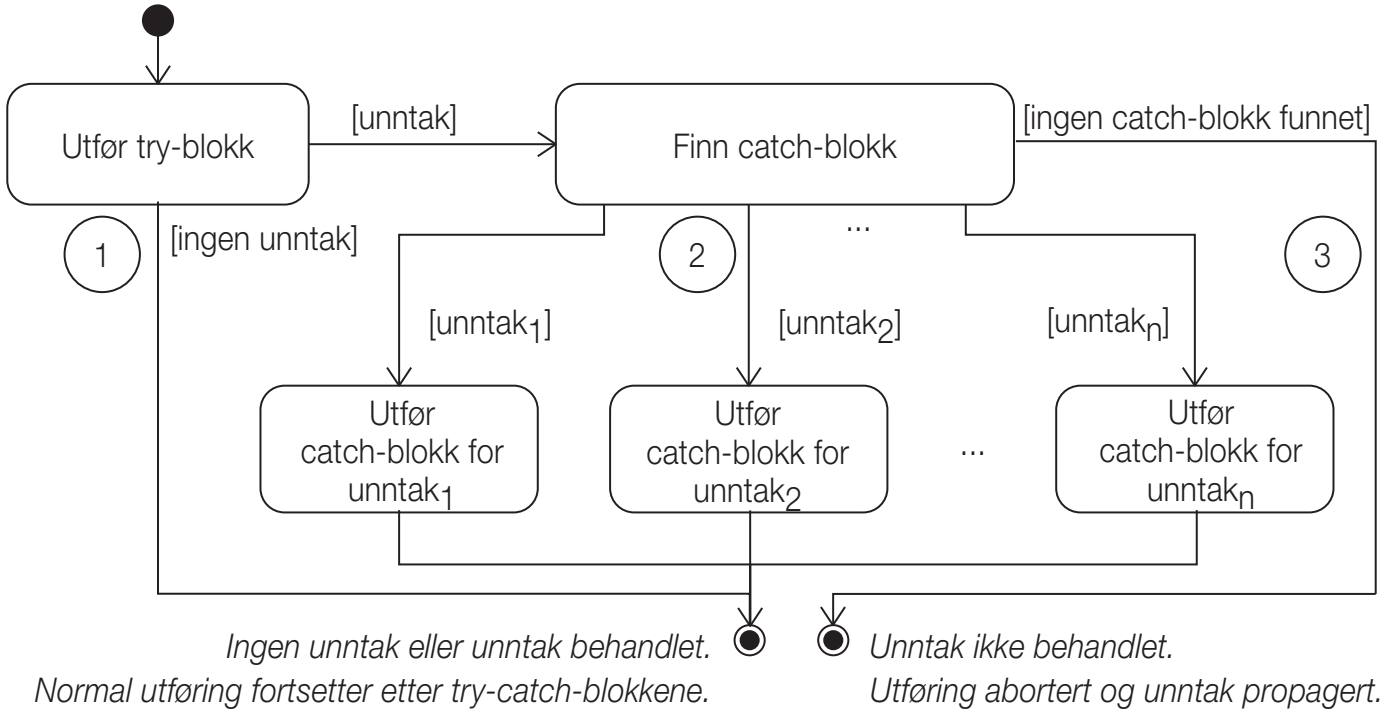
én catch-blokk-parameter

En catch-blokk kan fange et unntak og vil behandle det dersom det er av riktig type.

1. Koden i `try`-blokken utføres, og ingen unntak blir kastet.
   – Normalutføring fortsetter etter `try-catch`-blokkene.
2. Koden i `try`-blokken utføres, og et unntak blir kastet. Dette unntaket blir fanget og behandlet i en tilhørende `catch`-blokk.
   – Avbryter utføringen av `try`-blokken, slik at handlingene i resten av `try`-blokken ikke blir utført.
   – Normalutføring fortsetter etter `try-catch`-blokkene.

3. Koden i `try`-blokken utføres, og et unntak blir kastet, men ingen `catch`-blokk blir funnet for å behandle unntaket.
   - Avbryter utføringen av `try`-blokken, slik at handlingene i resten av `try`-blokken ikke blir utført.
   - Unntaket blir propagert.

try-catch-scenarier (Figur 13.4)



try-catch-scenario 1

- I Program 13.2 bruker metoden `skrivHastighet()` en `try-catch`-setning, (3).
- Den tilhørende `catch`-blokken, (6), er deklarert til å fange unntak av typen `ArithmeticException`.
- Utføringsmønsteret for Program 13.2 er vist i Figur 13.5.

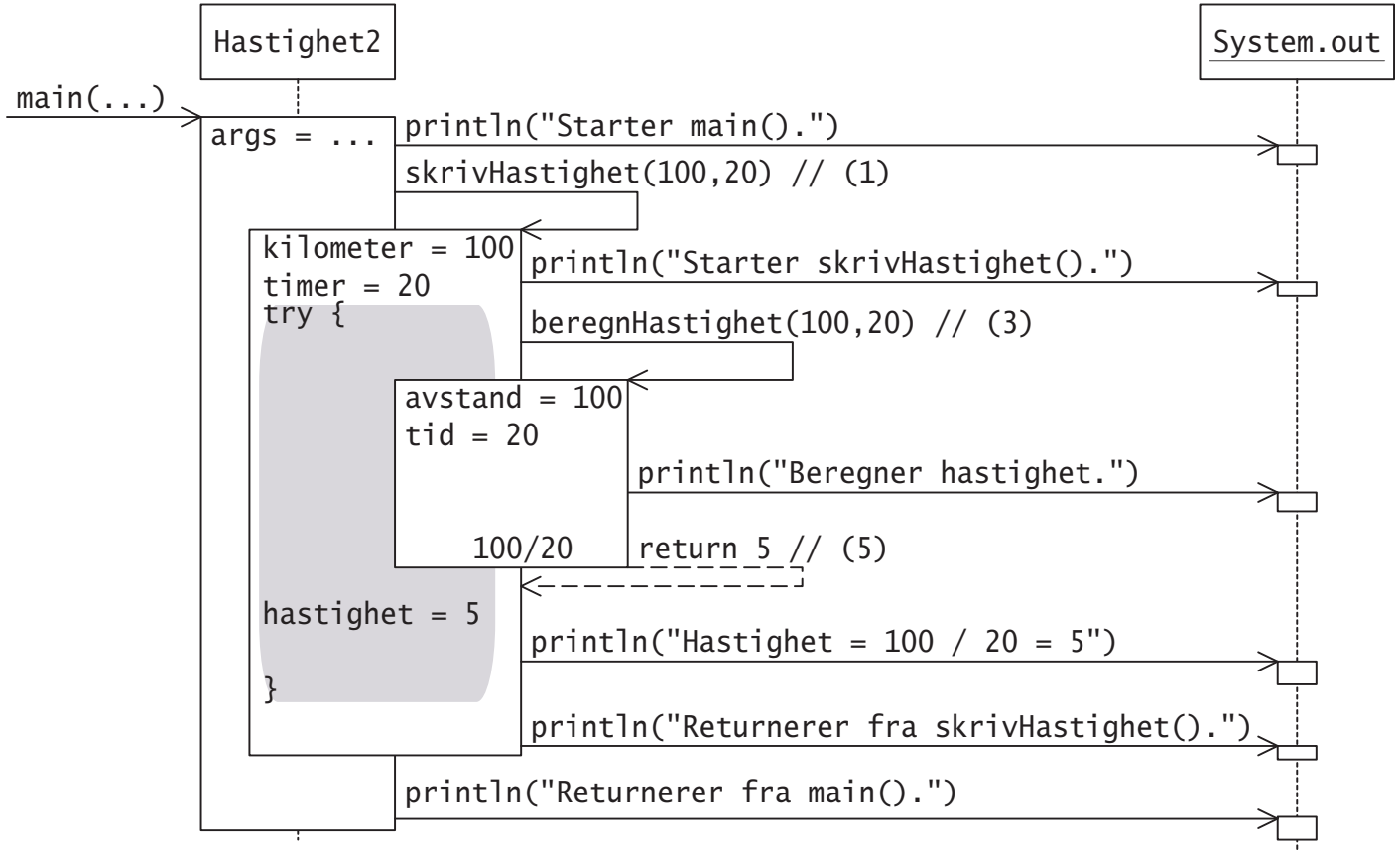
Unntakshåndtering (Program 13.2)

```
public class Hastighet2 {
  public static void main(String[] args) {
    System.out.println("Starter main().");
    skrivHastighet(100, 0);                                          // (1)
    System.out.println("Returnerer fra main().");
  }

  private static void skrivHastighet(int kilometer, int timer) {
    System.out.println("Starter skrivHastighet().");
    try {                                                            // (2)
      int hastighet = beregnHastighet(kilometer, timer);             // (3)
      System.out.println("Hastighet = " +
                         kilometer + "/" + timer + " = " + hastighet);
    }
    catch (ArithmeticException unntak) {                             // (4)
      System.out.println(unntak + " (behandlet i skrivHastighet())");
    }
    System.out.println("Returnerer fra skrivHastighet().");
  }

  private static int beregnHastighet(int avstand, int tid) {
    System.out.println("Beregner hastighet.");
    return avstand/tid;                                              // (5)
  }
}
```

Unntakshåndtering (Figur 13.5) (scenario 1)



Utskrift fra programmet:

```
Starter main().
Starter skrivHastighet().
Beregner hastighet.
Hastighet = 100/20 = 5
Returnerer fra skrivHastighet().
Returnerer fra main().
```

try-catch-scenario 2

- Erstatt i Program 13.2:

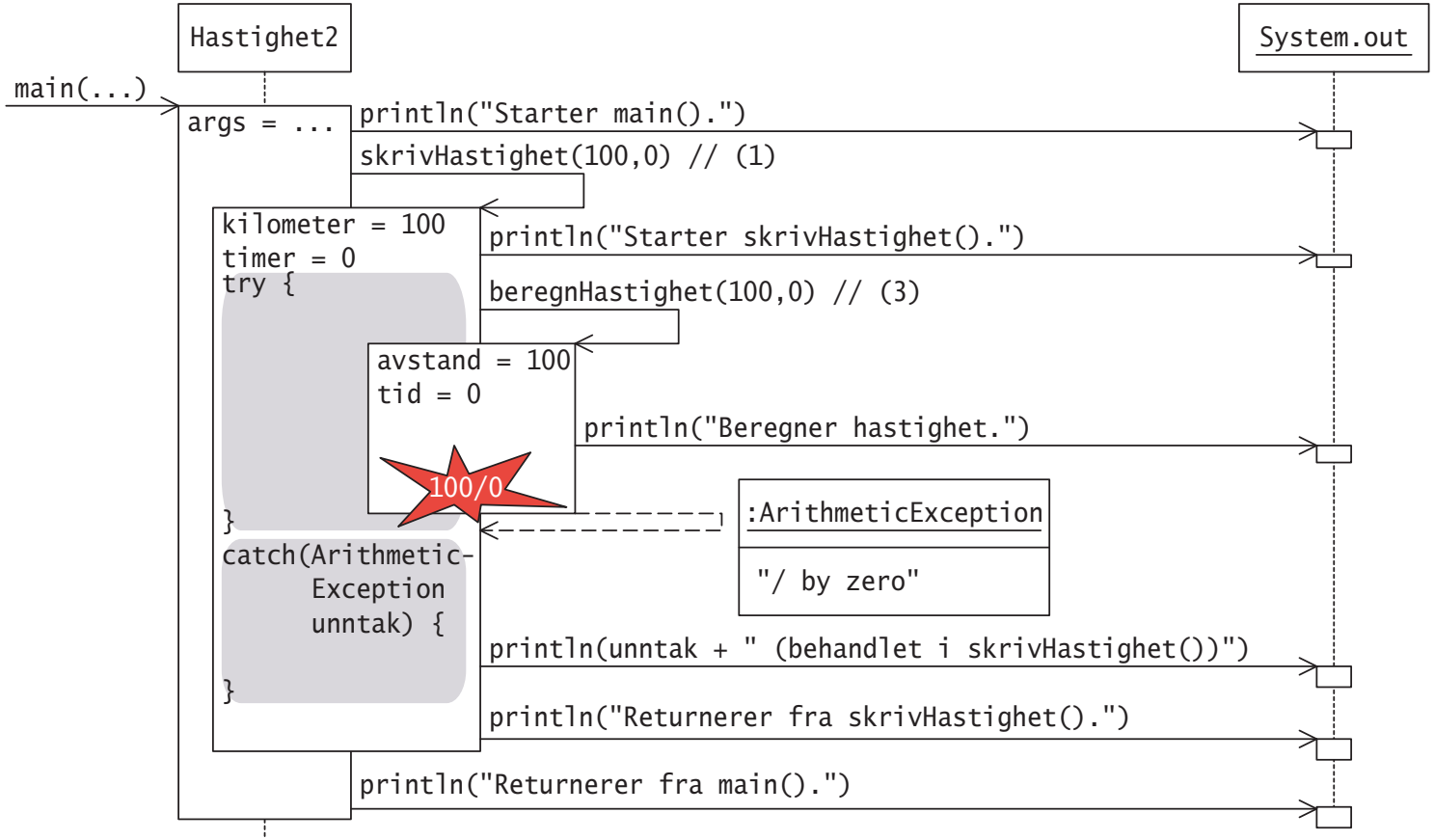
```
skrivHastighet(100, 20);    // (1)
```

- med

```
skrivHastighet(100, 0);     // (1) Andre parameter er 0.
```

- Heltallsdivisjon med `0` fører til at et `ArithmeticException`-unntak blir kastet ved (11) i metoden `beregnHastighet()`.
  - Utføringen av denne metoden avbrytes, og unntaket propageres.
  - Det blir fanget av `catch`-blokken i metoden `skrivHastighet()`.
  - Etter behandlingen av unntaket gjenopprettes normalutføring av programmet.
  - Utskriften viser normalutføring av metoden `skrivHastighet()` fra dette tidspunktet.

Unntakshåndtering (Figur 13.6) (scenario 2)



Utskrift fra programmet:

```
Starter main().
Starter skrivHastighet().
Beregner hastighet.
java.lang.ArithmeticException: / by zero (behandlet i skrivHastighet())
Returnerer fra skrivHastighet().
Returnerer fra main().
```

try-catch-scenario 3

- Scenario 3 belyser hva som skjer når et unntak blir kastet under utføring av en `try`-blokk og det ikke finnes en tilhørende `catch`-blokk for å behandle unntaket.
- Programmets utføringsmønster i Figur 13.7 viser at heltallsdivisjon med `0` igjen fører til at et `ArithmeticException`-unntak blir kastet ved (15) i metoden `beregnHastighet()`.
  - Utføringen av denne metoden avbrytes, og unntaket propageres.
- `ArithmeticException`-unntaket blir *ikke* fanget av `catch`-blokken i metoden `skrivHastighet()`, siden den fanger unntak av type `IllegalArgumentException`.
- Utføringen av metoden `skrivHastighet()` avbrytes (handling i (13) blir ikke utført), og unntaket propageres videre.
- Unntaket `ArithmeticException` blir nå fanget av `catch`-blokken i metoden `main()` ved (3).
  - Etter behandlingen av unntaket i `catch`-blokken i metoden `main()` gjenopprettes normalutføring av programmet.
  - Utskriften viser normalutføring av metoden `main()` fra dette tidpunktet.

Unntakshåndtering (Program 13.3) (scenario 3)

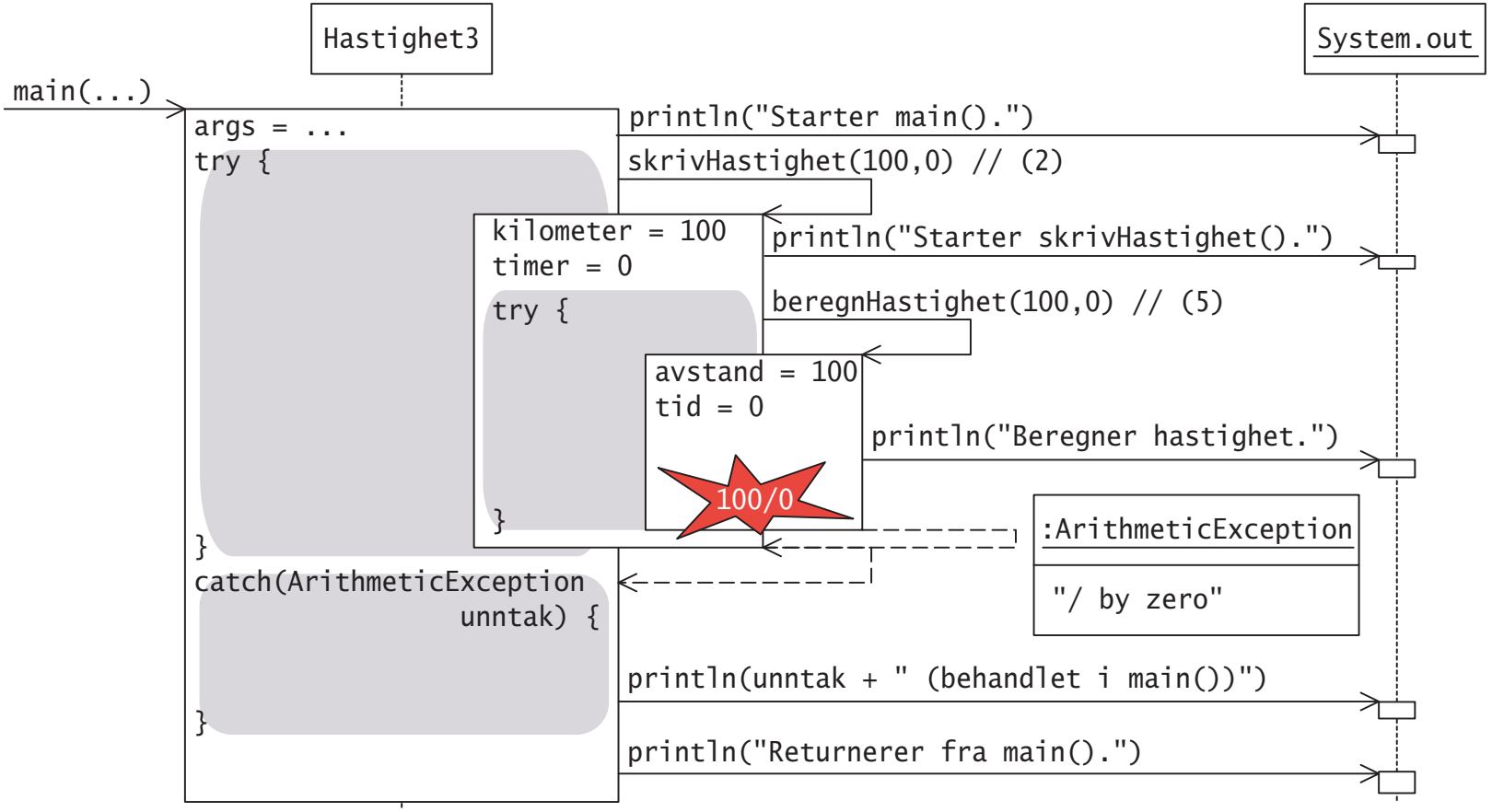
```
public class Hastighet3 {

  public static void main(String[] args) {
    System.out.println("Starter main().");
    try {                                                          // (1)
      skrivHastighet(100, 0);                                      // (2)
    }
    catch (ArithmeticException unntak) {                           // (3)
      System.out.println(unntak + " (behandlet i main())");
    }
    System.out.println("Returnerer fra main().");
  }

  private static void skrivHastighet(int kilometer, int timer) {
    System.out.println("Starter skrivHastighet().");
    try {                                                          // (4)
      int hastighet = beregnHastighet(kilometer, timer);           // (5)
      System.out.println("Hastighet = " +
                         kilometer + "/" + timer + " = " + hastighet);
    }
    catch (IllegalArgumentException unntak) {                      // (6)
      System.out.println(unntak + " (behandlet i skrivHastighet())");
    }
    System.out.println("Returnerer fra skrivHastighet().");
  }
```

```
  private static int beregnHastighet(int avstand, int tid) {
    System.out.println("Beregner hastighet.");
    return avstand/tid;                                        // (7)
  }
}
```

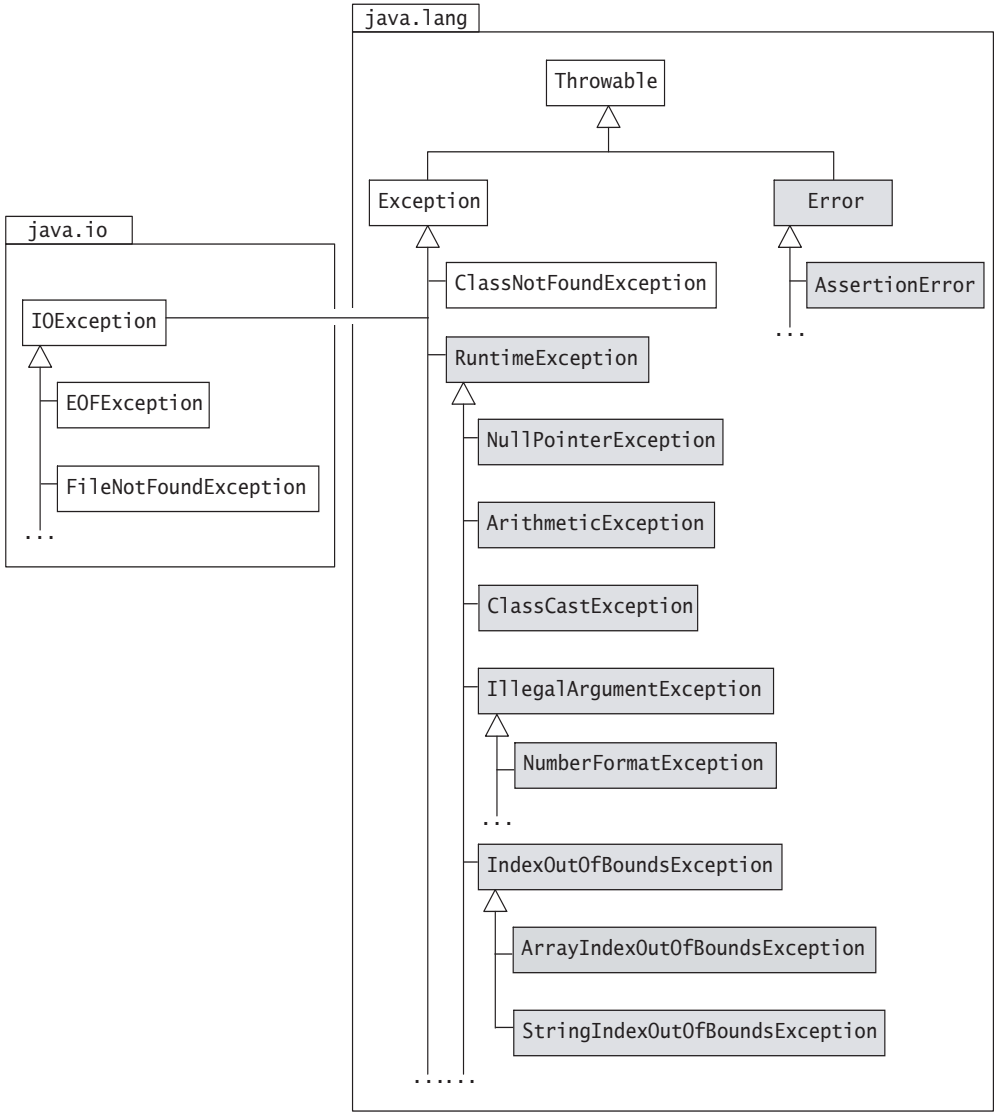
Unntakshåndtering (Figur 13.7) (scenario 3)



Utskrift fra programmet:

```
Starter main().
Starter skrivHastighet().
Beregner hastighet.
java.lang.ArithmeticException: / by zero (behandlet i main())
Returnerer fra main().
```

Utdrag av unntaksklasser (Figur 13.8)



Skyggelagte klasser (og deres subklasser) representerer ikke-kontrollerte unntak.

Utdrag av metoder fra `Throwable`-klassen (Tabell 13.1)

Metode	Beskrivelse
`String getMessage()`	Returnerer strengen i unntaket. Strengen angir ytterligere forklaring på unntaket.
`void printStackTrace()`	Skriver på terminalen programstabelutlistingen som førte til at unntaket ble kastet.

Unntaksklassen Exception og kontrollerte unntak

- Unntak som tilhører `Exception`-klassen og dens subklasser, med unntak av subklasser til `RuntimeException`-klassen, kalles *kontrollerte unntak* (eng. *checked exceptions*).
  - Kompilatoren kontrollerer at en metode som kaster et kontrollert unntak, eksplisitt håndterer unntaket (*mer om det senere*).

Unntaksklassen RuntimeException og ikke-kontrollerte unntak

- Klassen `RuntimeException` og dens subklasser representerer unntak som angår typiske uforutsette feil, som for eksempel *programmeringsfeil*.
- Unntak som er definert ved klassen `RuntimeException` og dens subklasser kalles *ikke-kontrollerte unntak* (eng. *unchecked exceptions*).
  - Det er ikke påkrevet at de fanges, men feilårsaken må rettes opp i programmet.

Unntaksklassen Error

- Subklassen `AssertionError` brukes til å signalisere at en påstand ikke holder under utføringen.
  - Slike unntak burde ikke fanges.

Eksplisitt kast av et unntak

- Et program kan eksplisitt kaste et unntak med `throw`-setningen:

```
throw new ArithmeticException("Avstand og tid kan ikke være < 0");
```

- Utføring av en `throw`-setning avbryter normalutføring av programmet, og unntaket propageres på vanlig måte.
- Det er vanlig å bruke en passende unntaksklasse til å definere en feilsituasjon og gi supplerende opplysninger om unntaket i konstruktørkallet.
- I Program 13.4 vil programutføringen føre til at et unntak av klassen `ArithmeticException` blir kastet i metoden `beregnHastighet()`, (2).
  - Dette unntaket blir propagert og blir fanget av `catch`-blokken i `main()`-metoden.
  - Fra dette tidspunktet fortsetter normalutføring av programmet.

Eksplisitt kast av unntak (Program 13.4)

```
public class Hastighet4 {
  public static void main(String[] args) {
    System.out.println("Starter main().");
    try {
      skrivHastighet(-100, 10);                    // (1) Verdi mindre enn 0.
    }
    catch (ArithmeticException unntak) {
      System.out.println(unntak + " (behandlet i main())");
    }
    System.out.println("Returnerer fra main().");
  }

  private static void skrivHastighet(int kilometer, int timer) {
    System.out.println("Starter skrivHastighet().");
    int hastighet = beregnHastighet(kilometer, timer);
    System.out.println("Hastighet = " +
                       kilometer + "/" + timer + " = " + hastighet);
    System.out.println("Returnerer fra skrivHastighet().");
  }

  private static int beregnHastighet(int avstand, int tid) {
    System.out.println("Beregner hastighet.");
    if (avstand < 0 || tid < 0)                                    // (2)
      throw new ArithmeticException("Avstand og tid kan ikke være < 0"); // (3)
    return avstand/tid; }}
```

- Utdata fra programmet:

```
Starter main().
Starter skrivHastighet().
Beregner hastighet.
java.lang.ArithmeticException: Avstand og tid kan ikke være < 0 (behandlet i main())
Returnerer fra main().
```

Håndtering av unntak med flere `catch`-blokker

- Dersom koden i en `try`-blokk kan kaste forskjellige typer unntak, kan man spesifisere én `catch`-blokk for hver type unntak etter `try`-blokken (Program 13.5).
- Indeksering i strengtabellen `args`, (3) og (4), kan kaste et ikke-kontrollert `ArrayIndexOutOfBoundsException`-unntak med mindre minst to strenger er spesifisert på kommandolinjen.
- Konvertering til heltall med `parseInt()`-metoden, (3) og (4), kan kaste et ikke-kontrollert `NumberFormatException`-unntak dersom noen av strengene inneholder tegn som ikke kan inngå i et heltall.

try-blokk med flere catch-blokker (Progam 13.5)

```
public class Hastighet5 {

  public static void main(String[] args) {
    System.out.println("Starter main().");
    int arg1, arg2;                                          // (1)
    try {                                                    // (2)
      arg1 = Integer.parseInt(args[0]);                      // (3)
      arg2 = Integer.parseInt(args[1]);                      // (4)
    }
    catch (ArrayIndexOutOfBoundsException unntak) {          // (5)
      System.out.println("Angi både kilometer og timer.");
      System.out.println("Bruk: java Hastighet5 <kilometer> <timer>");
      System.out.println(unntak  + " (behandlet i main())");
      return;
    }
    catch (NumberFormatException unntak) {                   // (6)
      System.out.println("Kilometer og timer må være heltall.");
      System.out.println("Bruk: java Hastighet5 <kilometer> <timer>");
      System.out.println(unntak  + " (behandlet i main())");
      return;
    }
    skrivHastighet(arg1, arg2);                              // (7)
    System.out.println("Returnerer fra main().");
  }
```

```
  private static void skrivHastighet(int kilometer, int timer) {
    System.out.println("Starter skrivHastighet().");
    try {
      int hastighet = beregnHastighet(kilometer, timer);
      System.out.println("Hastighet = " +
                         kilometer + "/" + timer + " = " + hastighet);
    }
    catch (ArithmeticException unntak) {
      System.out.println(unntak  + " (behandlet i skrivHastighet())");
    }
    System.out.println("Returnerer fra skrivHastighet().");
  }

  private static int beregnHastighet(int avstand, int tid) {
    System.out.println("Beregner hastighet.");
    if (avstand < 0 || tid < 0)
      throw new ArithmeticException("Avstand og tid kan ikke være < 0");
    return avstand/tid;
  }
}
```

- Utdata fra programmet:

```
>java Hastighet5 100
Starter main().
Angi både kilometer og timer.
Bruk: java Hastighet5 <kilometer> <timer>
java.lang.ArrayIndexOutOfBoundsException: 1 (behandlet i main())

>java Hastighet5 200 4u
Starter main().
Kilometer og timer må være heltall.
Bruk: java Hastighet5 <kilometer> <timer>
java.lang.NumberFormatException: For input string: "4u" (behandlet i main())

>java Hastighet5 200 -10
Starter main().
Starter skrivHastighet().
Beregner hastighet.
java.lang.ArithmeticException: Avstand og tid kan ikke være < 0 (behandlet i skrivHastig-
het())
Returnerer fra skrivHastighet().
Returnerer fra main().
```

```
>java Hastighet5 200 0
Starter main().
Starter skrivHastighet().
Beregner hastighet.
java.lang.ArithmeticException: / by zero (behandlet i skrivHastighet())
Returnerer fra skrivHastighet().
Returnerer fra main().

>java Hastighet5 100 20
Starter main().
Starter skrivHastighet().
Beregner hastighet.
Hastighet = 200/20 = 10
Returnerer fra skrivHastighet().
Returnerer fra main().
```

try-blokk med felles catch-blokk

- Dersom flere unntak skal håndteres på samme måte i en `try-catch` setning, kan vi bruke en *felles* `catch`-blokk.

```
try { ... }
catch(UnntakType1 | UnntakType2 |...| UnntakTypeN u) { // Felles catch
  /* Felles håndetering for unntak u. */
}
catch(UnntakType unntak) { ... } /* Evt. andre catch-blokker.
...
```

  – Dersom det forekommer et unntak i `try`-blokken, som har parametertype angitt i en felles `catch`-blokk, vil koden i denne `catch`-blokken blir utført.
  – Unntak-parameter `u` er implisitt endelig (`final`), derfor kan ikke dens referanseverdi endres i den felles `catch`-blokken.

try-blokk med felles catch-blokk (fort.)

- I `main()`-metoden i klassen `Hastighet5`, kan vi velge å håndtere unntakene av type `ArrayIndexOutOfBoundsException` og `NumberFormatException` på samme måte v.h.a. en felles `catch`-blokk.
- Se klassen `Hastighet5Alt`.
  - Programmets øvrige oppførsel er den samme som i klassen `Hastighet5`.

```
/** Klassen Hastighet5 er omskrevet til å bruke en felles catch-blokk. */
public class Hastighet5Alt {

  public static void main(String[] args) {
    System.out.println("Starter main().");
    int arg1, arg2;
    try {
      arg1 = Integer.parseInt(args[0]);
      arg2 = Integer.parseInt(args[1]);
    }
    catch (ArrayIndexOutOfBoundsException | NumberFormatException unntak) {
      System.out.println("Bruk: java Hastighet5 <kilometer> <timer>");
      System.out.println("Angi både kilometer og timer som heltall.");
      System.out.println(unntak  + " (behandlet i main())");
      return;
    }
```

```
      skrivHastighet(arg1, arg2);
      System.out.println("Returnerer fra main().");
    }
    ...
  }
```

Utdata fra programmet:

```
>java Hastighet5Alt 100 xx
Starter main().
Bruk: java Hastighet5 <kilometer> <timer>
Angi både kilometer og timer som heltall.
java.lang.NumberFormatException: For input string: "xx" (behandlet i main())

>java Hastighet5Alt 100
Starter main().
Bruk: java Hastighet5 <kilometer> <timer>
Angi både kilometer og timer som heltall.
java.lang.ArrayIndexOutOfBoundsException: 1 (behandlet i main())
```

```
>java Hastighet5Alt 550 25
Starter main().
Starter skrivHastighet().
Beregner hastighet.
Hastighet = 550/25 = 22
Returnerer fra skrivHastighet().
Returnerer fra main().

>java Hastighet5Alt 100 0
Starter main().
Starter skrivHastighet().
Beregner hastighet.
java.lang.ArithmeticException: / by zero (behandlet i skrivHastighet())
Returnerer fra skrivHastighet().
Returnerer fra main().
```

Typiske programmeringsfeil ved unntakshåndtering

- En parametertype i én `catch`-blokk kan skygge for andre unntakstyper i etterfølgende `catch`-blokker.
  - For eksempel skygger superklassen `RuntimeException` i `catch`-blokk (1) for subklassen `ArithmeticException` i `catch`-blokk (2):

```
try { ... }
catch (RuntimeException unntak1) { ... }        // (1)
catch (ArithmeticException unntak2) { ... }     // (2)
```

  - Unntak av typen `ArithmeticException` blir fanget av `catch`-blokk (1) og aldri av `catch`-blokk (2), siden objekter av subklasser kan tilordnes en superklassereferanse.
  - Kompilatoren vil gi feilmelding dersom slike tilfeller forekommer.
- Ikke fang alle unntak i én `catch`-blokk ved å bruke mer generelle unntaksklasser, for eksempel `Exception` og `RuntimeException`.
  - Bruk av spesifikke unntaksklasser, gjerne med flere `catch`-blokker, anbefales i `try`-`catch`-konstruksjonen for å øke programforståelsen.
- I en felles `catch`-blokk, unntakstypene til parametrene må være *disjunkte*, d.v.s. at det er ingen arv-relasjon mellom dem -- ellers vil kompilatoren gi feilmelding.

Håndtering av kontrollerte unntak

- Hensikten med unntakshåndteringen er å behandle *unormale* situasjoner under programutføring.
  - Det er mulig for en metode som kaster et unntak, å la unntaket propagere videre uten å gjøre noe med det.
  - Dette oppfordrer til utvikling av lite robuste programmer.
- Bruk av kontrollerte unntak gir en strategi der en metode som kan kaste et slikt unntak, tvinges til å ta stilling til hvordan unntaket skal håndteres:

1. Fange og behandle unntaket i en `try-catch`-setning.
2. Tillate videre propagering av unntaket med en `throws`-klausul spesifisert i metodedeklarasjonen.

- En `throws`-klausul spesifiseres i metodehodet, mellom parameterlisten og metodekroppen:

  ... metodeNavn (...) throws unntaksklasse_1, ..., unntaksklasse_n { ... }

- Siden kompilatoren unnlater å sjekke ikke-kontrollerte unntak og metodene kan la være å håndtere dem, spesifiseres vanligvis ikke slike unntak i en `throws`-klausul.

Håndtering av kontrollerte unntak

- Gitt at metode `foo()` kaller metode `bar()`:

```
void bar() throws K { /*...*/ }       // K er kontrollert unntak.

void foo() { bar(); }                 // Ikke godtatt av kompilatoren.
```

- Løsning:

  Enten

```
void foo() throws K { bar(); }        // Kaster unntaket videre.
```

  eller:

```
void foo() {
  try { bar();}
  catch(K k) {k.printStackTrace()}    // Fanger og behandler unntaket.
}
```

- Kompilatoren vil sjekke at en metode som kan kaste et kontrollert unntak oppfyller et av kravene ovenfor.
- Dette betyr at enhver klient av en metode som kan propagere et kontrollert unntak i en `throws`-klausul, må ta stilling til hvordan dette unntaket skal håndteres.
- Dersom et kontrollert unntak spesifisert i en `throws`-klausul blir propagert helt til toppnivået, vil det bli behandlet av en standard unntakshåndterer på vanlig måte.

Eksempel: kontrollerte unntak (Program 13.6)

- Metoden `beregnHastighet()` kaster et unntak av typen `Exception` i en `if`-setning, (6).
  - Den velger å kaste dette unntaket videre i en `throws`-klausul, (5).
- Metoden `skrivHastighet()` som kaller metoden `beregnHastighet()`, velger også å kaste det videre i en `throws`-klausul, (4).
- Metoden `main()` som kaller metoden `skrivHastighet()`, velger å fange og behandle dette unntaket i en `try-catch`-blokk, (1) og (3).
- Forsøk på å utelate `throws`-klausulene vil resultere i kompileringsfeil.

Håndtering av kontrollerte unntak (Program 13.6)

```
public class Hastighet6 {

  public static void main(String[] args) {
    System.out.println("Starter main().");
    try {                                                    //(1)
      skrivHastighet(100, 20);                               //(2a)
//    skrivHastighet(-100,20);                               //(2b)
    }
    catch (Exception unntak) {                               //(3)
      System.out.println(unntak + " (behandlet i main())");
    }
    System.out.println("Returnerer fra main().");
  }

  private static void skrivHastighet(int kilometer, int timer)
                      throws Exception {                     //(4)
    System.out.println("Starter skrivHastighet().");
    double hastighet = beregnHastighet(kilometer, timer);
    System.out.println("Hastighet = " +
                        kilometer + "/" + timer + " = " + hastighet);
    System.out.println("Returnerer fra skrivHastighet().");
  }
```

```
  private static int beregnHastighet(int avstand, int tid)
                    throws Exception {                        //(5)
    System.out.println("Beregner hastighet.");
    if (avstand < 0 || tid <= 0)                              //(6)
      throw new Exception("Avstand og tid må være > 0");
    return avstand/tid;
  }
}
```

- Utføring med kodelinjen: `skrivHastighet(100, 20);` // (2a)

```
Starter main().
Starter skrivHastighet().
Beregner hastighet.
Hastighet = 100/20 = 5.0
Returnerer fra skrivHastighet().
Returnerer fra main().
```

- Utføring med kodelinjen: `skrivHastighet(100, 0);` // (2b)

```
Starter main().
Starter skrivHastighet().
Beregner hastighet.
java.lang.Exception: Avstand og tid må være > 0 (behandlet i main())
Returnerer fra main().
```

Definere egne kontrollerte unntakstyper

- Det anbefales å definere nye subklasser til klassen `Exception`.
  - Dette gjør at nye unntak automatisk blir kontrollert av kompilatoren.

```
class HastighetsBeregningsUnntak extends Exception { // kontrollert unntak

  HastighetsBeregningsUnntak(String str) { super(str); }

}
```

  - Det er vanligvis tilstrekkelig å definere en konstruktør som tar en strengparameter.

Eksempel med bruk av egne unntak (Program 13.7)

```
public class Hastighet7 {
  public static void main(String[] args) {
    System.out.println("Starter main().");
    try {                                                          //(1)
      skrivHastighet(100, 20);                                     //(2a)
//    skrivHastighet(-100,20);                                     //(2b)
    }
    catch (HastighetsBeregningsUnntak unntak) {                    //(3)
      System.out.println(unntak + " (behandlet i main())");
    }
    finally {                                                      //(4)
      System.out.println("Minner om kommandoen: java Hastighet7");
    }
    System.out.println("Returnerer fra main().");
  }

  private static void skrivHastighet(int kilometer, int timer)
                      throws HastighetsBeregningsUnntak {          //(5)
    System.out.println("Starter skrivHastighet().");
    double hastighet = beregnHastighet(kilometer, timer);
    System.out.println("Hastighet = " +
        kilometer + "/" + timer + " = " + hastighet);
    System.out.println("Returnerer fra skrivHastighet().");
  }
```

```
  private static int beregnHastighet(int avstand, int tid)
                     throws HastighetsBeregningsUnntak {                  //(6)
    System.out.println("Beregner hastighet.");
    if (avstand < 0 || tid <= 0)
      throw new HastighetsBeregningsUnntak("Avstand og tid må være > 0"); //(7)
    return avstand/tid;
  }
}
```

- Utføring med kodelinjen: `skrivHastighet(100, 20);` `// (2a)`

```
Starter main().
Starter skrivHastighet().
Beregner hastighet.
Hastighet = 100/20 = 5.0
Returnerer fra skrivHastighet().
Minner om kommandoen: java Hastighet7
Returnerer fra main().
```

- Utføring med kodelinjen: `skrivHastighet(-100, 20);` `// (2b)`

```
Starter main().
Starter skrivHastighet().
Beregner hastighet.
HastighetsBeregningsUnntak: Avstand og tid må være > 0 (behandlet i main())
Minner om kommandoen: java Hastighet7
Returnerer fra main().
```

  – Legg merke til at utføringen av metoden `skrivHastighet()` ble avbrutt *etter* at `finally`-blokken var utført.

Utføring av finally-blokk

- En `finally`-blokk kan forekomme sammen med en `try`-blokk.
  - En `catch`-block trenger ikke nødvendigvis å være spesifisert.
- Koden i `finally`-blokken blir *alltid* utført dersom `try`-blokken blir utført.
  - Det spiller ingen rolle om et unntak ble kastet eller ikke.
- "Oppryddingskode" kan legges i en `finally`-blokk slik at den alltid blir utført.
- Klassen `Hastighet7` illustrerer utføring av `finally`-blokk.
  - En `finally`-blokk er spesifisert ved (13), med tilhørende `try`-blokk ved (7).
  - Utskriften viser at koden i `finally`-blokken blir utført uansett.

Unntakskasting og arv

- En overkjørt metode i subklassen har bare lov å spesifisere i sin `throws`-klausul *en undermengde av unntak* fra `throws`-klausul i metodedefinisjonen fra superklassen.

```
// Klasse A
   public void enMetodeMedMangeUnntak() throws Unntak1, Unntak2, Unntak3 {...}
...
// I subklassen B som er utledet fra superklassen A
   public void enMetodeMedMangeUnntak() throws Unntak1, Unntak3 {...}
```

Oppsummering av unntakshåndtering

- "*Kast og fang* "-regel for *kontrollerte* unntak:
  - En metode kan fange unntak den kaster og behandle dem i `catch`-blokker.
- De kontrollerte unntakene som en metode kan kaste og ikke fanger, må deklareres i en `throws`-klausul til metoden.
- En metode kan eksplisitt kaste et unntak v.h.a. `throw`-setningen.

```
... enMetode(...) throws Unntak1, Unntak2, ... , Unntakn {
    ...
    try { // handlinger som kan føre til at et av følgende typer unntak kan kastes:
          // Unntak1, Unntak2, ... , Unntakn og Unntaki, ..., Unntakk
        throw new Unntaki(); // kaster et unntak av typen Unntaki
    }
    catch (Unntakj1 | Unntakj2 |...| Unntakjm uj) {// behandler flere typer unntak.
        // kode for å behandle unntak uj
    }
    ...
    catch (Unntakk uk) { // behandler unntak av typen Unntakk
        // kode for å behandle unntak uk
    }
    finally { ... } // Koden som alltid blir utført dersom try blir utført.
    ...
}
```

`try-catch-finally` blokk