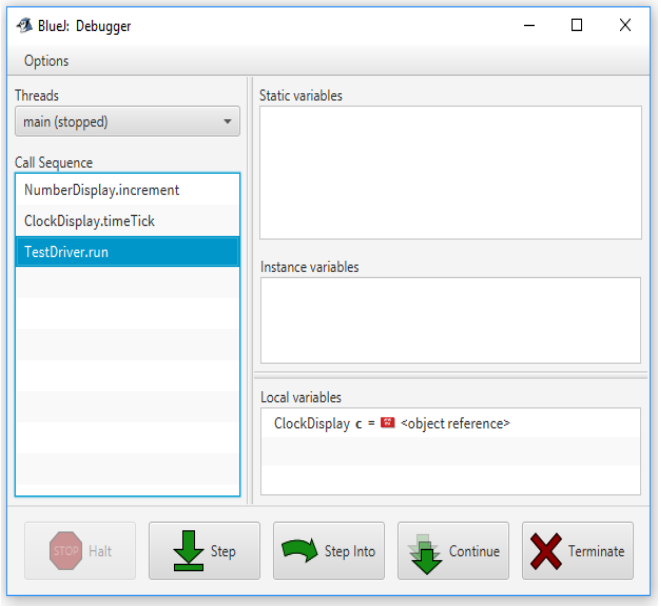
Forelæsning Uge 10

- **Opremsningstyper**
  - Enumerated types
- **Forskellige teknikker til test og debugging**
  - Når man tester undersøger man, **om** opførslen (semantikken) er den ønskede
  - Når man debugger (afluser), forsøger man at finde ud af, **hvorfor** opførslen ikke er, som man forventede, og **hvordan** dette kan rettes
- **Afleveringsopgave: Raflebæger 4**

- **Langt de fleste af jer har nu lavet dronningeopgaven, som er en af de sværeste opgaver på kurset**
- **Når I kan klare den, er der ingen grund til at tro, at I ikke kan klare de resterende opgaver**

Frem til og med mandag den 9. december vil studiecaféen være bemandet med en instruktor fra kurset på følgende tidspunkter:

- **Mandag kl. 12-14**
- **Tirsdag kl. 8-10**
- **Onsdag og Torsdag kl. 10-12**
- **Fredag kl. 13-15**



Opremsningstyper (enumerated types)

- **Type, hvor programmøren eksplicit angiver de mulige værdier**
  - Nedenstående type har 8 mulige værdier
  - Bemærk at værdierne ikke er tekststrenge, men objekter af typen **Weekday**
  - Værdierne angives ved f.eks. at skrive **Weekday.TUESDAY**

```
public enum Weekday {
   // A value for each weekday,
   // plus one for unrecognised days.
   MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
   SATURDAY, SUNDAY, UNKNOWN
}
```

8 værdier

- **Alternativt kunne man repræsentere ugedagene ved hjælp af heltal [1,7] eller ved hjælp af tekststrenge**
  - Heltal ville være sværere at forstå, og hvad betyder det, hvis man har en illegal værdi som 17 eller -3
  - Tekststrenge kan let indeholde stavefejl (f.eks. "TUSDAY")
  - Man kan selvfølgelig også stave forkert, når man bruger en enumereret type
  - Men sådanne stavefejl vil compileren fange

Mere komplekse enumerations

- Opremsningstyper kan også have konstruktører, feltvariabler og metoder

```
public enum Weekday {
   MONDAY("Monday"), TUESDAY("Tuesday"),
   WEDNESDAY("Wednesday"), THURSDAY("Thursday"),
   FRIDAY("Friday"), SATURDAY("Saturday"),
   SUNDAY("Sunday"), UNKNOWN("?");

   private String weekday;

   Weekday(String weekday) {
      this.weekday = weekday;
   }

   public String toString() {
      return weekday;
   }
}
```

- Værdierne i Weekday skabes ved at kalde konstruktøren otte gange (med de ønskede tekststrenge som parametre)
- Bemærk at man bruger den værdi, man vil definere, i stedet for typens navn

Feltvariabel af type String

Konstruktør
- Altid private (som udelades)
- Andre objekter kan ikke tilføje nye værdier
- Initialiserer feltvariablen ud fra parameteren

toString metode
- returnerer værdien af feltvariablen weekday

Hvis man f.eks. vil ændre sproget til Tysk skal man kun ændre erklæringen af enumeration typen
- Tekststrengene i enumeration typen ændres til "Montag", "Dienstag", "Mittwoch"….
- Værdierne i enumeration typen er uændrede: MONDAY, TUESDAY, WEDNESDAY, ….

Eksempel på brug

Switch sætning (med variabel fra opremsningstype)

Bemærk, at vi ikke behøver at skrive Weekday.TUESDAY, men kan nøjes med TUESDAY

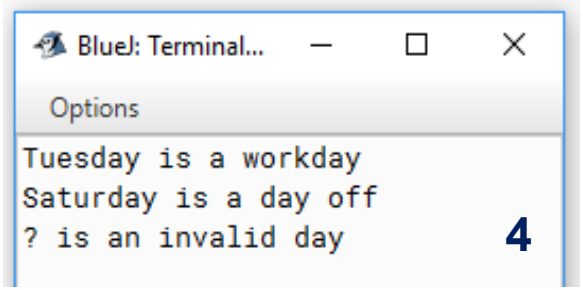
```
private static void print(Weekday day) {
  String dayString;
  switch(day) {
    case MONDAY:
    case TUESDAY:
    case WEDNESDAY:
    case THURSDAY:
    case FRIDAY:    dayString = " is a workday";
                    break;
    case SATURDAY:
    case SUNDAY:    dayString = " is a day off";
                    break;
    default:        dayString = " is an invalid day";
                    break;
  }
  System.out.println(day + dayString);
}
```

Lokal String variabel

Værdi fra Weekday typen (toString metoden i Weekday)

Tekststreng fra switch sætning

```
public static void testPrint() {
  print(Weekday.TUESDAY);
  print(Weekday.SATURDAY);
  print(Weekday.UNKNOWN);
}
```



World of zuul

- **I world-of-zuul projektet kunne vi med fordel have defineret de mulige exists ved hjælp af en enumeration type**
  - Nu vil oversætteren protestere, hvis vi (i vores kode) staver en exit forkert

```
public enum Exit {
   NORTH("north"), EAST("east"), SOUTH("south"), WEST("west");
   private String exit;
   Exit(String exit) {
      this.exit = exit;
   }
   public String toString() {
      return exit;
   }
}
```

Som i Weekday har vi en:

- Feltvariabel
- Privat konstruktør, der initialiserer feltvariablen
- toString metode, der returnerer værdien af feltvariablen

```
private Map<Exit, Room> exits;
```

Test og debugging

- **Test af program**
  - Vi undersøger om programmet fungerer **korrekt**
  - **Logisk korrekt**: Producerer programmet de resultater, som vi forventer
  - **Effektivitet**: Er programmet hurtigt nok til at kunne håndtere store datamængder og mange brugere (performance analyse)
  - **Brugervenligt**: Er programmet let at forstå og let betjene
  - I det følgende vil vi koncentrere os om **logisk korrekthed**
- **Debugging af program**
  - Når et program indeholder fejl, bruger vi debugging til at lokalisere fejlene, dvs. finde ud af, **hvor** fejlen er og **hvad**, der skal rettes for at få programmet til at fungere korrekt
  - "Bug" betyder lus/insekt (slang for fejl i et program)
  - "Debugging" betyder aflusning (dvs. man fjerner fejl i programmet)
  - Se evt. https://en.wikipedia.org/wiki/Debugging Link

Systemudviklingsfaser

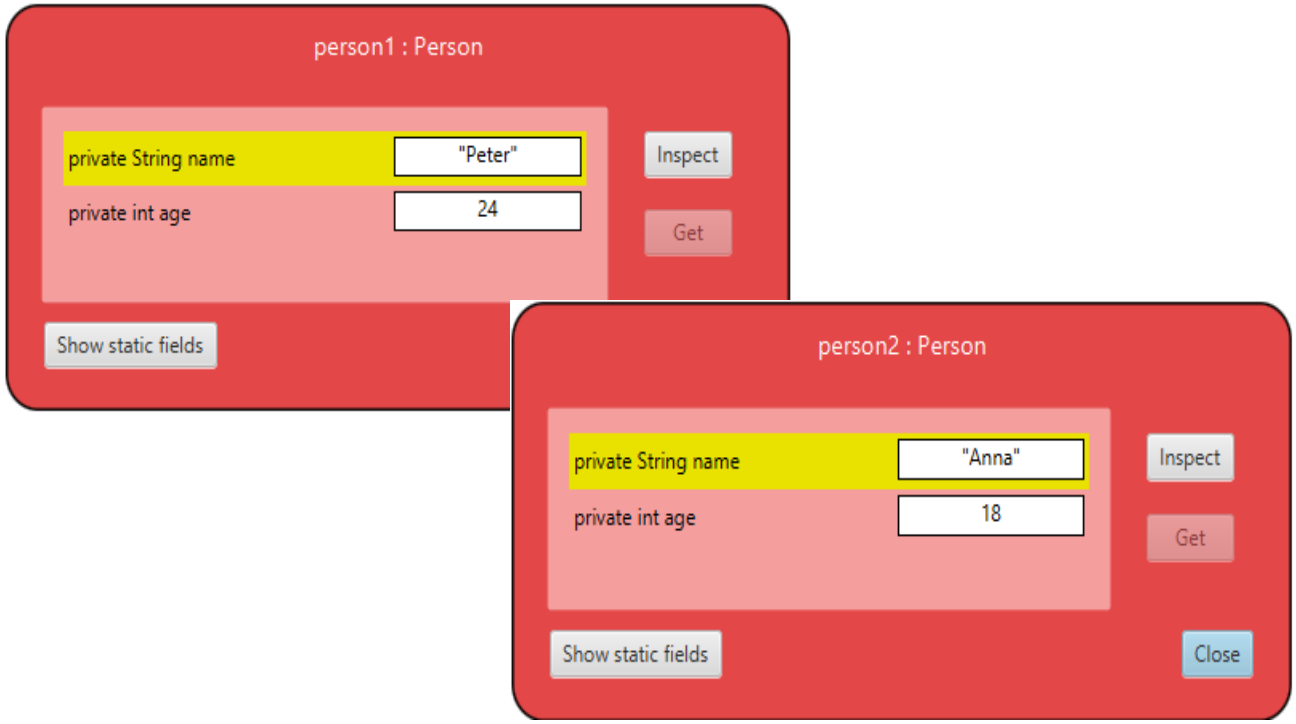
- **Udvikling af et program foregår i en række faser**
  - **Analyse**, hvor man udarbejder en kravspecifikation (dvs. en beskrivelse af, hvordan programmet skal fungere)
  - **Design**, hvor man fastlægger programmets arkitektur (dvs. hvilke klasser/metoder det skal have og hvordan de interagerer)
  - **Implementation**, hvor man programmerer klasser/metoder
  - **Test**, hvor man tester om klasser/metoder er korrekte
  - **Debugging**, hvor man retter de fejl, som man har fundet
- **Iterationer**
  - I praksis, må man ofte gå tilbage til tidligere faser
  - Hvis man finder fejl, kan det være nødvendigt at ændre programmets arkitektur eller dele af kravspecifikationen
  - Når man har rettet fejl, skal det rettede program igen testes/debugges
- **Versioner**
  - Man kan med fordel opdele udviklingen af et større program i en række trin, hvor man hen af vejen udvikler, tester og debugger mere komplekse **versioner**
  - Når programmet er taget i brug, vil der ofte opstå behov for nye versioner, f.eks. på grund af nye regler eller nye ønsker til programmet

Unit tests

- **Test af en afgrænset programenhed, f. eks.**
  - Klasse
  - Metode / konstruktør

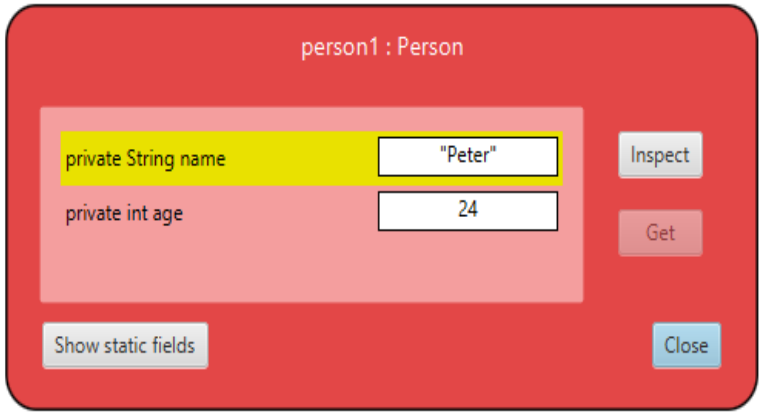
Unit = enhed

- **Simple unit tests kan foretages ved hjælp af BlueJ's inspektorer**
  - Viser værdien af feltvariabler (og klassevariabler)



Positive og negative tests

- **En positivt test undersøger, om programenheden opfører sig som forventet ved "normal brug"**
  - Bliver en persons navn opdateret til den tekststreng, som vi angiver i kaldet af setName metoden?
  - Husk at teste omkring diverse grænseværdier – hvor der ofte er fejl
  - F.eks. bør en metode, der finder teenagere, testes på personer, der har alderen 12, **13**, 14, 18, **19**, 20 år
- **En negativt test undersøger, om programenheden opfører sig fornuftigt i "uventede situationer"**
  - Hvad sker der, hvis vi forsøger at sætte navnet til den tomme streng eller alderen til noget negativt eller meget stort?
  - Håndterer metoden det fornuftigt eller får man en inkonsistent tilstand?
- **Begge typer tests er vigtige**
  - Man er ofte tilbøjelig til at glemme (eller nedprioritere) de negative tests
  - Lad være med det

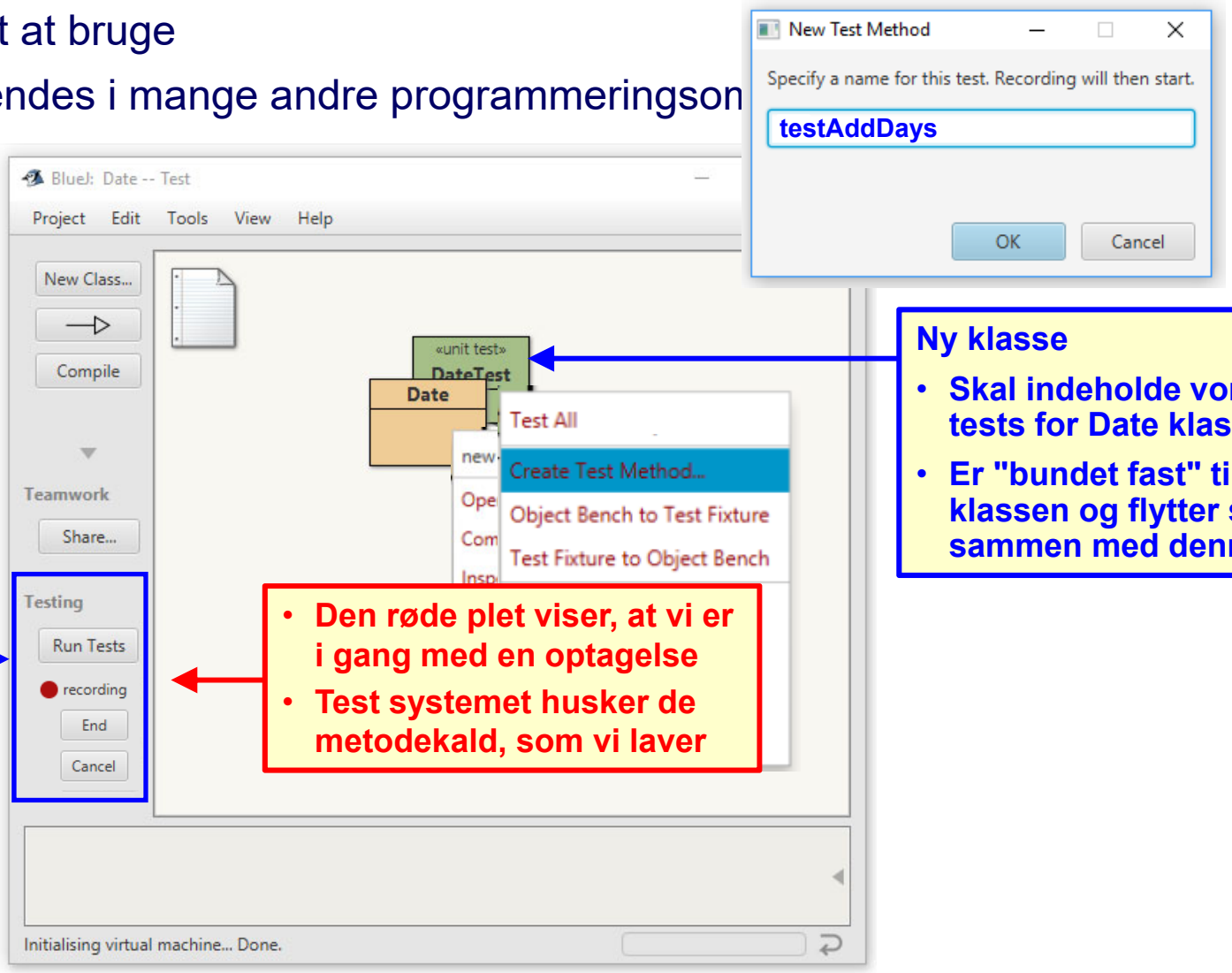


Regression tests

- **Regression tests**
  - Når man ændrer i en klasse, bør man efterfølgende tjekke, at alting stadig fungerer korrekt (dvs. opfylder passende positive og negative tests)
  - Har man ved et uheld fået ødelagt noget, som tidligere fungerede? (**regression** ≈ tilbageslag / forværring)
- **Det er tidskrævende og kedeligt at lave regression tests manuelt**
  - Regression tests bliver derfor ofte udeladt
  - Det kan koste enorme mængder af tid, når man senere finder en mærkelig fejl og ikke aner, hvordan og hvornår den er opstået
- **Løsningen er at lave automatiske tests, der let kan gentages**
  - Man definerer en mængde af positive og negative tests
    - Hvilke operationer skal udføres?
    - Hvad skal resultatet være?
  - Derefter er det op til test systemet (i vores tilfælde BlueJ) at gennemføre testene og tjekke, om de giver de forventede resultater
  - Det kan gøres på få sekunder – uden programmørens aktive medvirken
  - Med automatiske tests er det langt mere overkommeligt at lave systematiske regression tests, hver gang programmet ændres

Automatiske tests i BlueJ

- **BlueJ indeholder et test system ved navn JUnit**
  - Nemt at bruge
  - Anvendes i mange andre programmeringson



Ny klasse

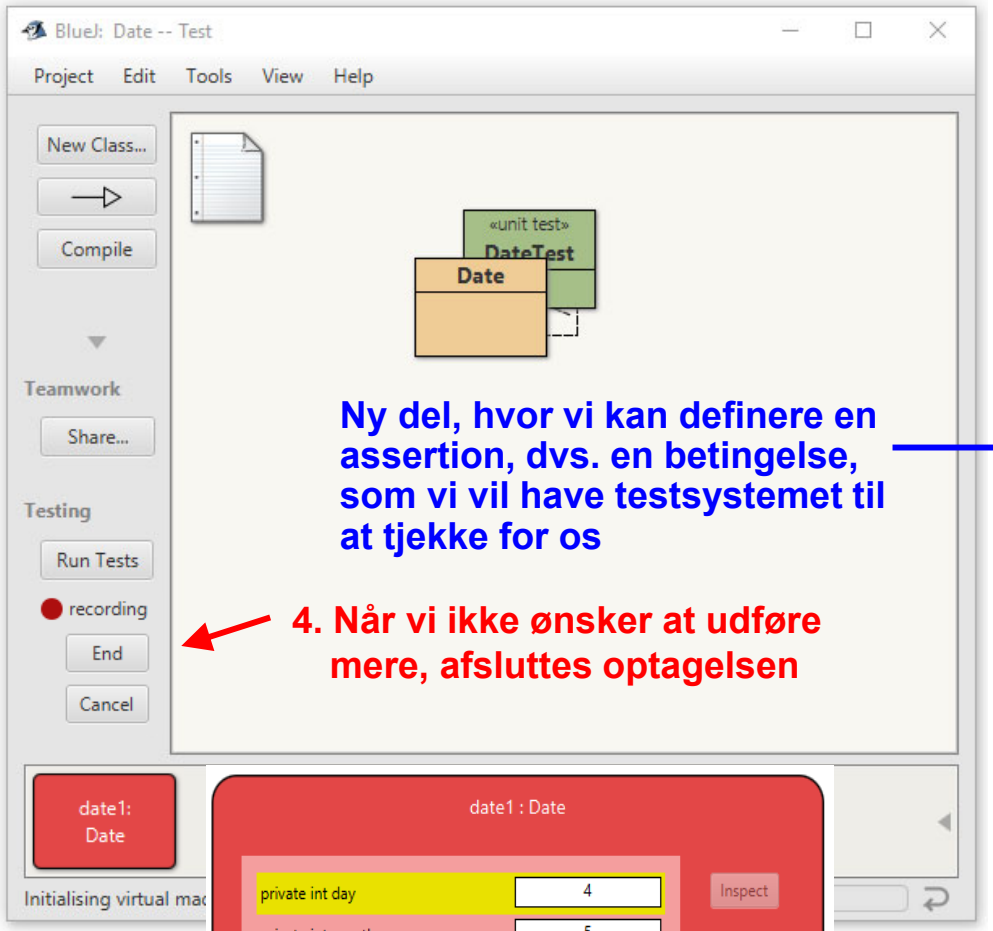
- **Skal indeholde vores tests for Date klassen**
- **Er "bundet fast" til Date klassen og flytter sig sammen med denne**

Knapper til optagelse og afspilning af tests

Gøres synlige ved at trykke på den lille trekantede knap

- **Den røde plet viser, at vi er i gang med en optagelse**
- **Test systemet husker de metodekald, som vi laver**

Optagelse af test



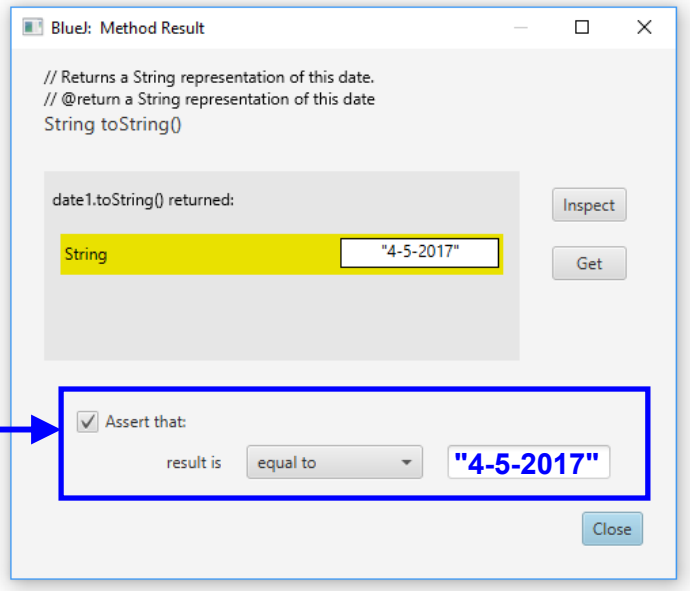
Ny del, hvor vi kan definere en assertion, dvs. en betingelse, som vi vil have testsystemet til at tjekke for os

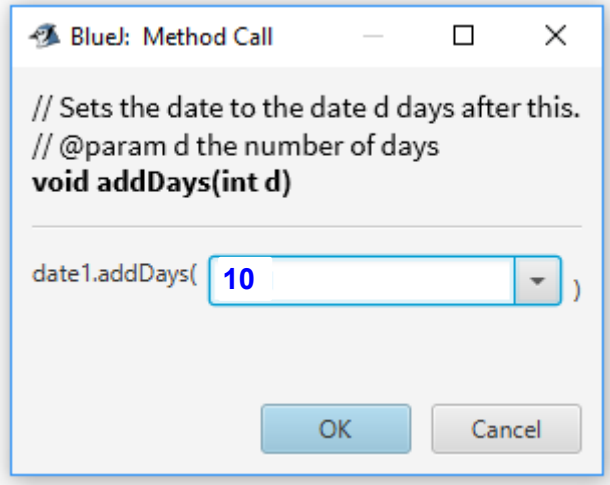
4. Når vi ikke ønsker at udføre mere, afsluttes optagelsen

1. Lav et Date objekt

Værdierne af feltvariablerne opdateres

3. Kald toString metoden på date1 objektet

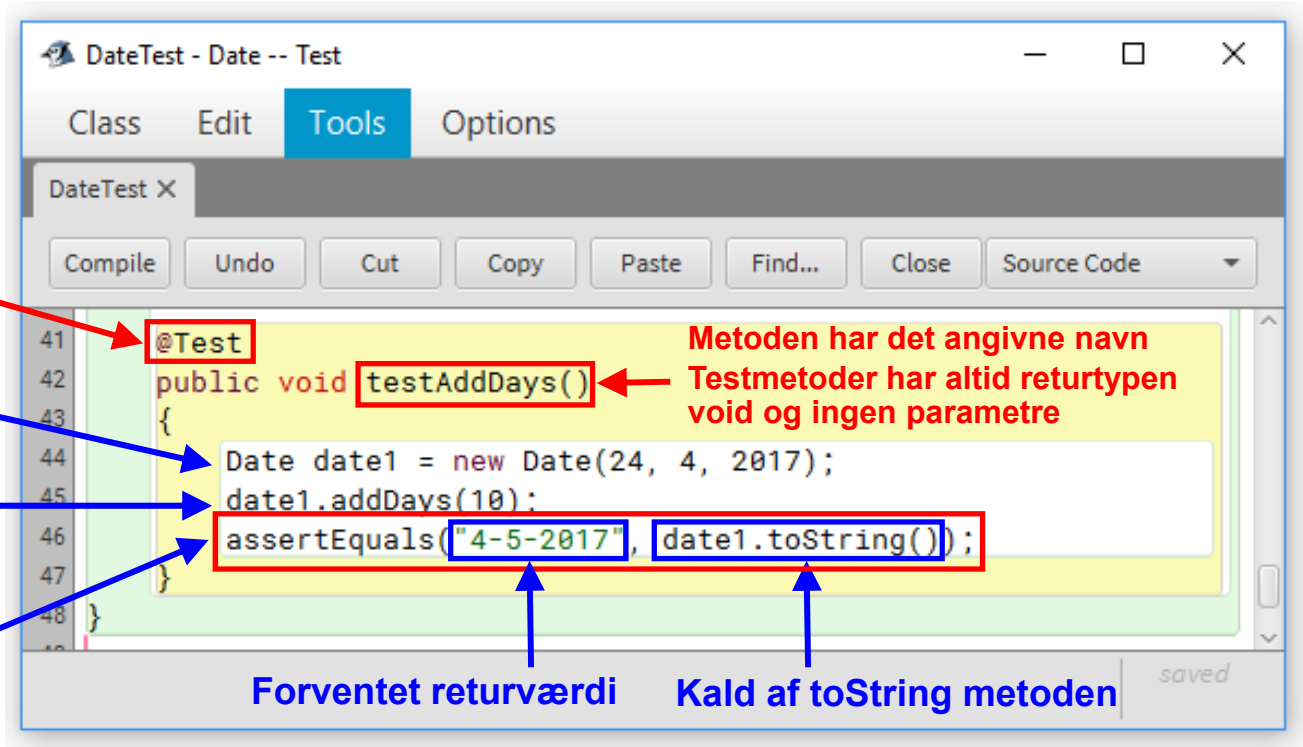




2. Kald addDays metoden på date1 objektet

Den optagne testmetode

- **I DateTest klassen er der tilføjet en ny metode testAddDays**
  - Indeholder Java kode, der udfører de tre ting, vi gjorde under optagelsen

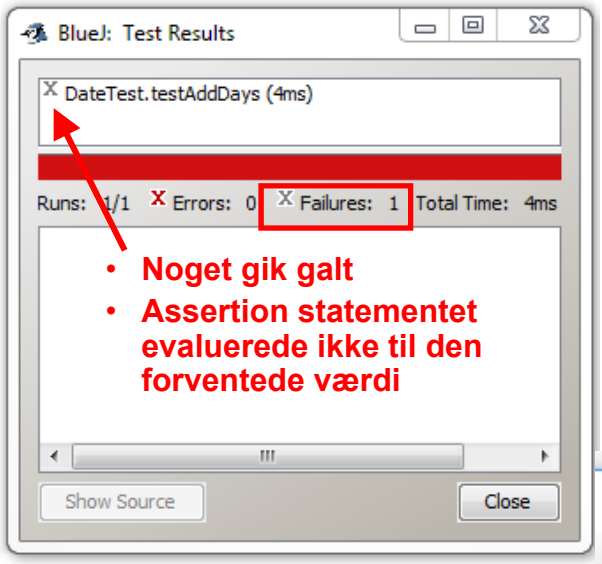


assertEquals metoden bruger equals metoden (i String klassen) til at tjekke, at de to parametre evaluerer til samme værdi

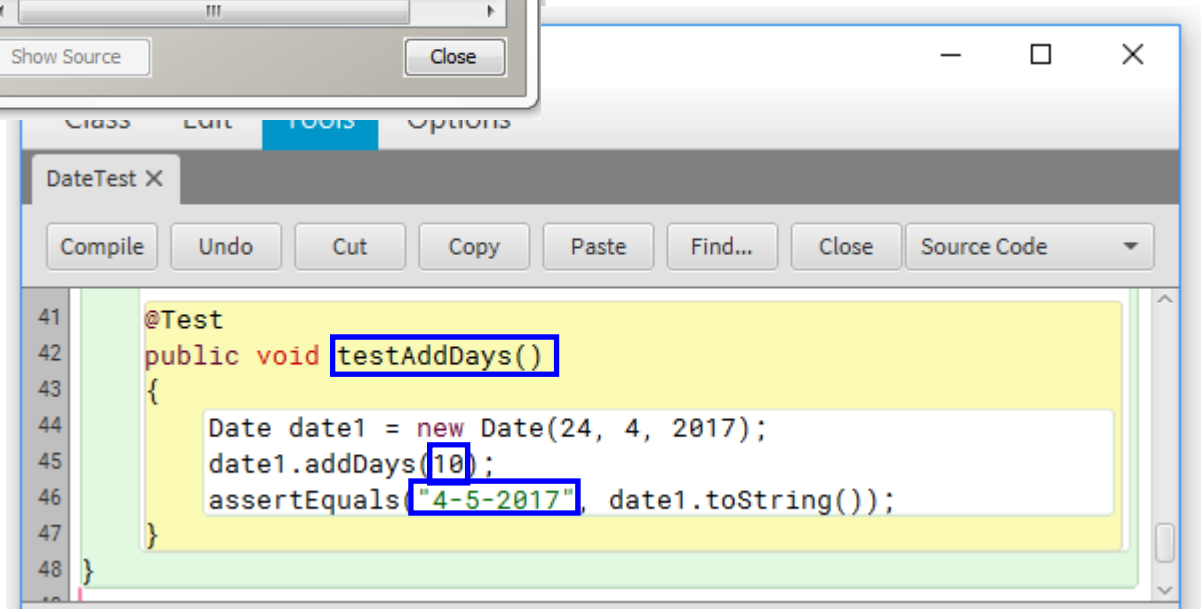
- **Anden parameteren er det metodekald, som vi vil teste, mens første parameteren er den returværdi, som vi forventer**
- **Hvis de to parametre ikke evaluerer til samme værdi, rejses en assertion error, og testmetoden stopper med angivelse af, at testen fejlede**

Kørsel af testmetode

- **Vi kan nu køre testmetoden, ved at trykke på Run tests knappen eller ved at kalde TestAll operationen for klassen DateTest**



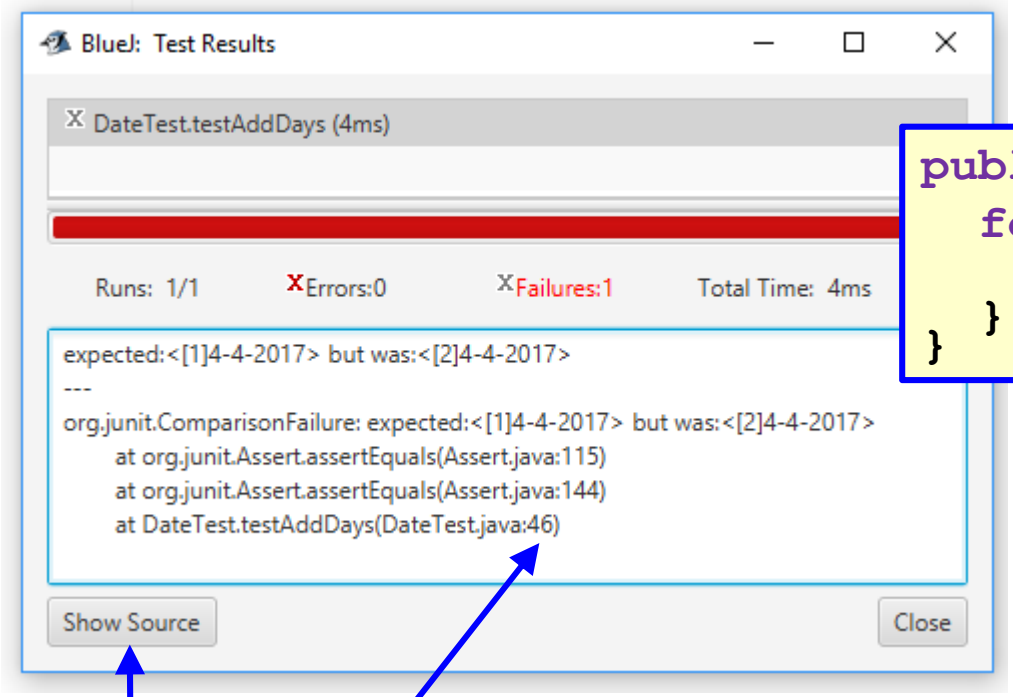
- **Hvis vi vil ændre i den optagne metode kan vi**
  - Lave en helt ny optagelse
  - Modificere Java koden i klassen DateTest



- **10 → 20**
- **"14-5-2017"**
- **Stadig OK**

- **10 → –10**
- **"14-4-2017"**
- **ERROR**

Beskrivelse af hvad der gik galt




```
public void addDays(int d) {
   for(int i=0; i<d; i++) {
      setToNextDate();
   }
}
```

- Hvis parameteren er negativ udføres kroppen af for løkken slet ikke
- Kan evt. ændres, så man for negative værdier kalder setToPreviousDate et antal gange

Viser hvilken assertion, der fejlede
(i vores tilfælde har vi kun én assertion,
så der ved vi godt, hvilken det var)

Når en testmetode fejler, kan det enten være fordi den testede metode er forkert, eller fordi testmetoden er forkert

Når man har lavet nogle få optagelser af testmetoder og set, hvordan den genererede Java kode ser ud, er det langt hurtigere at skrive testmetoderne selv – i stedet for at optage dem

Assertions

- **Brug altid den forventede værdi som første parameter og metodekaldet (den beregnede værdi) som anden parameter**
  - Nedenstående fire sætninger er ækvivalente
  - Men den første giver den bedste fejlmeddelelse

```
assertEquals(4, list.size());
assertEquals(list.size(), 4);
assertTrue(4 == list.size());
assertFalse(4 != list.size());
```

```
expected:<4> but was:<2>
expected:<2> but was:<4>
no exception message
no exception message
```

  - **assertEquals** bruger (som navnet antyder) **equals** metoden til at sammenligne de to værdier af en Objekt type
  - For primitive typer bruges **==** operatoren
  - Undlad at bruge **assert sætninger** (med det reserverede ord assert) i forbindelse med tests, idet disse kan give lidt tekniske komplikationer

- **Bemærk at knappen Run Tests kun tester metoderne i de test klasser, der allerede er succesfuldt oversat (compileret)**
  - Når man skriver eller ændrer en testmetode, skal man derfor huske at oversætte / genoversætte den pågældende test klasse

org.junit + import static

- **assert metoderne er klassemetoder i Assertions klassen, som tilhører pakken org.junit.jupiter.api**
  - Pakken er ikke en del af Javas standard klassebibliotek
  - Men I kan let google den og på den måde få adgang til at læse dens API

- **Når man importerer klassen, kan man tilføje nøgleordet static samt .* efter klassens navn**

```
import static org.junit.jupiter.api.Assertions.*;
```

  - Det bevirker, at man kan kalde klassemetoderne og bruge klassevariablerne **uden** at skrive "**Assertions.**" foran
  - Vi kan skrive

```
assertEquals(4, list.size());
assertTrue(1 <= sides && sides <= 6);
```

  i stedet for

```
Assertions.assertEquals(4, list.size());
Assertions.assertTrue(1 <= sides && sides <= 6);
```

  - Ovenstående import sætning indsættes automatisk i BlueJ's testklasser (sammen med import sætninger for tre andre klasser, som bruges i forbindelse med BlueJ's testmetoder)

Dokumentation af testklasser

- **Dokumentationen for jeres testklasser kan holdes på et minimum**
  - Testklassens navn fortæller, hvilken klasse, den tester
  - Testmetodens navn fortæller, hvilken metode, den tester
  - Testmetoder har ingen parametre og returnerer intet, så @param og @return tags giver ikke mening
- **I kan derfor nøjes med at indsætte**
  - @author og @version tags
  - Forklarende // kommentarer i kompleks testkode

Regression tests

- **Forberedelser**
  - For hver af klassens metoder laves en testmetode
  - Disse kan enten optages, eller man kan kode dem direkte i Java, hvilket er meget lettere
  - Den enkelte testmetode kan indeholde mange assertions og dermed teste flere forskellige ting
- **Man bruger ofte de samme objekter i mange testmetoder**
  - Man kan så (en gang for alle) definere en såkaldt **test fixture**, der indeholder de pågældende objekter
  - Fixture betyder "fast inventar" – detaljer er forklaret i afsnit 9.4.4
  - Test fixturen genetableres **inden** hver testmetode
- **Hver gang man ændrer en eller flere af klassens metoder**
  - Kører man alle testmetoderne – ved ét enkelt tryk på Run Tests / TestAll
  - Man kan så på få sekunder se, om alt stadig fungerer som forventet
  - Hvis der er fejl, skal man nøje overveje, om det er den testede metode eller testmetoden, der er forkert og skal rettes

Kan regression tests betale sig?

- **Ulemper**
  - Det tager lang tid at lave de mange testmetoder
- **Fordele**
  - Fejl introduceret på grund af koderettelser findes langt hurtigere og med langt større sandsynlighed
  - Senere kan sådanne fejl være virkelig svære at finde, idet fejlen måske er opstået da man rettede "noget helt andet"
  - Man slipper for kedelige manuelle tests

Pause

- **Konklusion**
  - Brug tid på (helt fra start) at udvikle testmetoder, der kan bruges til automatiske regression tests
  - Det betaler sig i det lange løb – også for metoder, der tilsyneladende er simple
  - I Raflebæger 4 og computerspilsopgaverne kommer I til at lave en masse regression tests
- **Testserveren**
  - Når I bruger testserveren til at tjekke jeres program, udfører I en række regression tests (som vi har skrevet)

Debugging (aflusning, fjernelse af fejl)

- **BlueJ bogen introducerer tre teknikker til debugging**
  - Manual gennemgang af koden
  - Brug af BlueJ's debugger
  - Indsættelse af print sætninger
- **Ideen i de tre teknikker er den samme**
  - Under udførslen af koden, inspicerer man
    - værdierne af udvalgte variabler
    - hvordan metoderne kalder hinanden
- **Forskelle mellem teknikkerne**
  - Ved en **manual gennemgang** klarer man alle beregninger selv
    - Det tager lang tid og man kan let lave fejl
  - **BlueJ's debugger** holder styr på variablernes værdier og hvilke metoder, der kalder hinanden
    - Det sker lynhurtigt og debuggeren laver aldrig fejl
    - Men det kræver lidt tid og kræfter at lære at bruge debuggeren
  - **Print sætninger** er en mellemting
    - Tingene beregnes automatisk, men hvis man vil se værdierne af nye variabler, må man manuelt ind og tilføje nye print sætninger
    - Derudover er det besværligt at indsætte (og fjerne) print sætninger

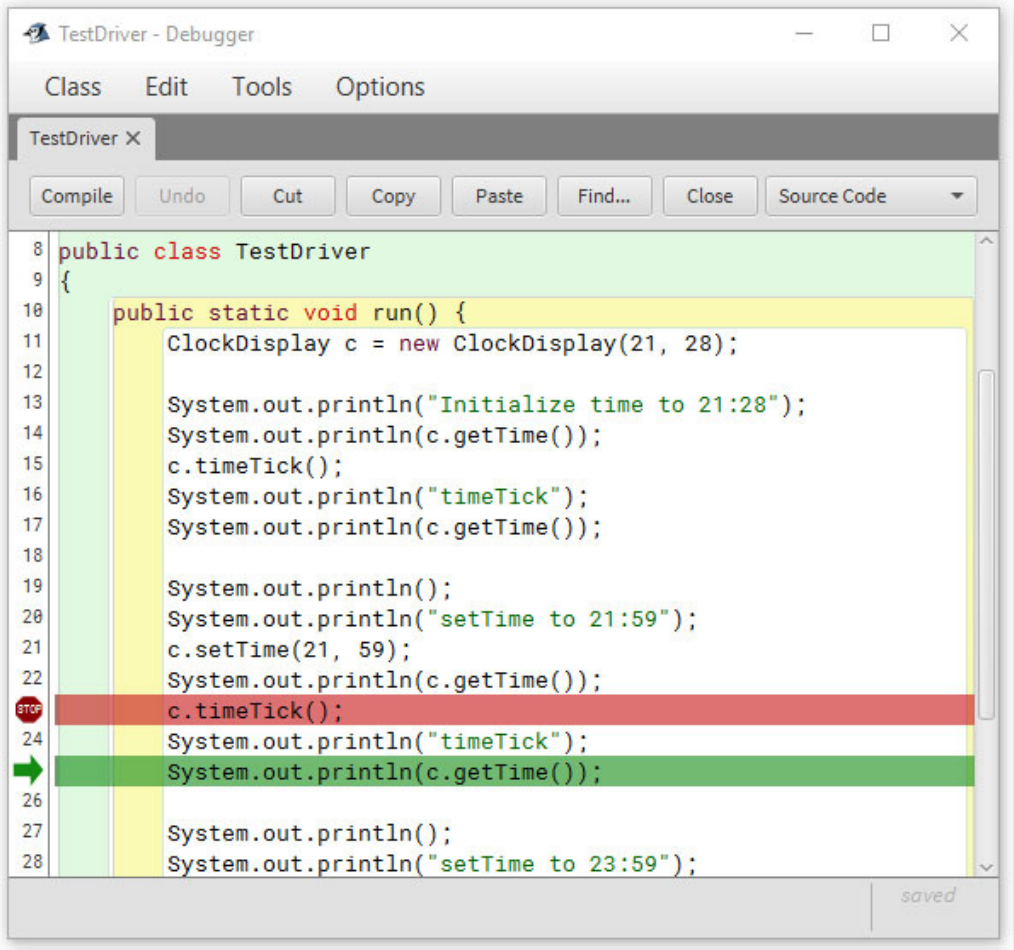
BlueJ's debugger (kort repetition)

Nyttig når man skal tjekke den detaljerede opførsel af kørende Java kode

Bruges i nogle af opgaverne efter efterårsferien

Breakpoints indsættes (og fjernes) ved at klikke i venstre margin af editoren

Under programudførelsen vil debuggeren stoppe, når et breakpoint nås, og vise positionen med en grøn pil (samt grøn farve)



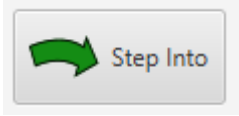
Herefter kan man "steppe" gennem koden sætning for sætning

Mellem skridtene kan man inspicere systemets tilstand, dvs. værdierne af de forskellige slags variabler

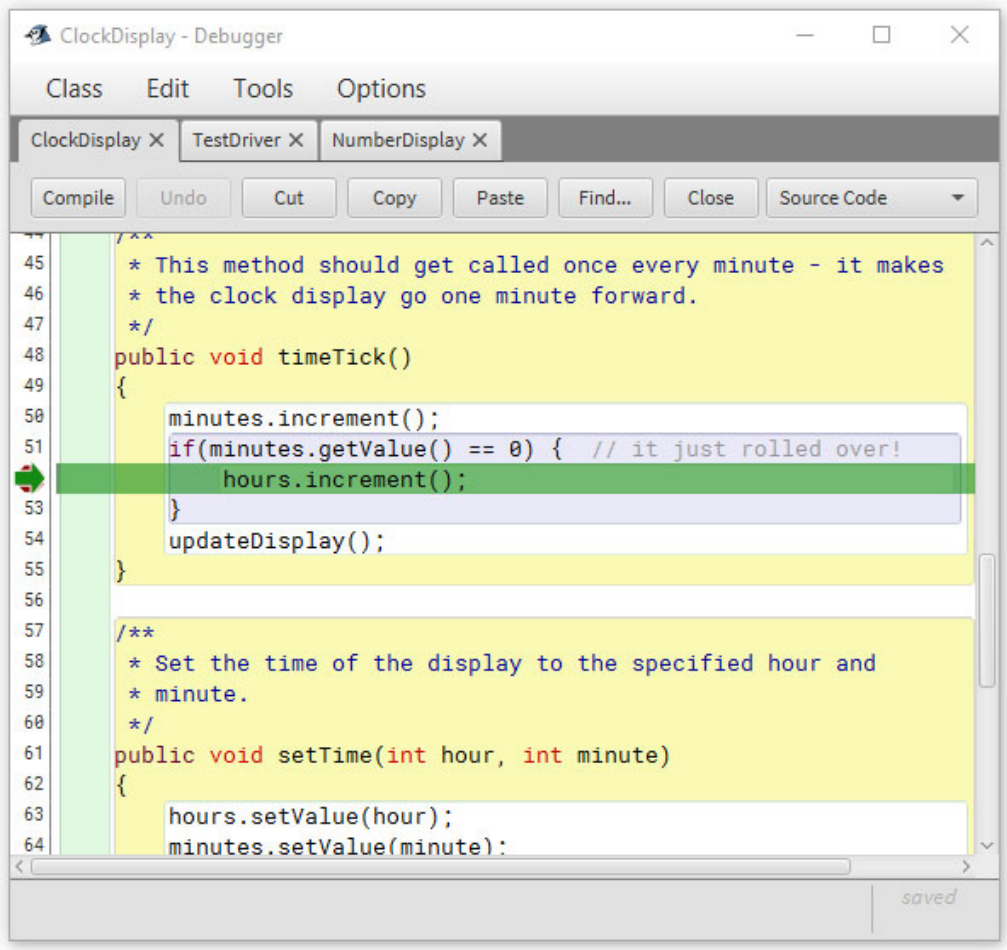
Metodekald

Når næste sætning er et metodekald, har man to muligheder:

Udfører hele metodekaldet uden at man ser detaljerne



Starter metodekaldet, men stopper inden første sætning i kroppen af den kaldte metode

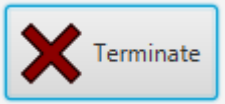


Metodekald

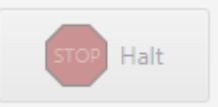
Parat til at udføre første sætning i den kaldte metode

Andre knapper:

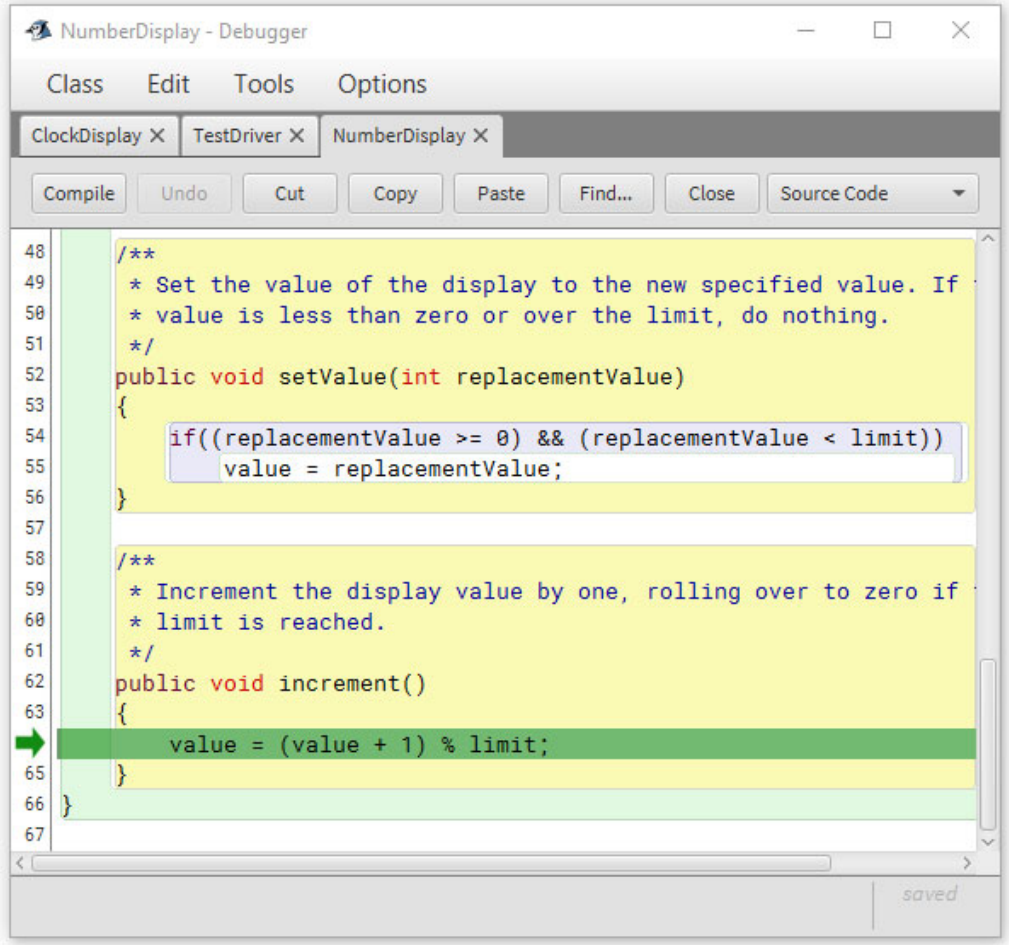
Fortsætter kørslen frem til næste breakpoint



Stopper kørslen



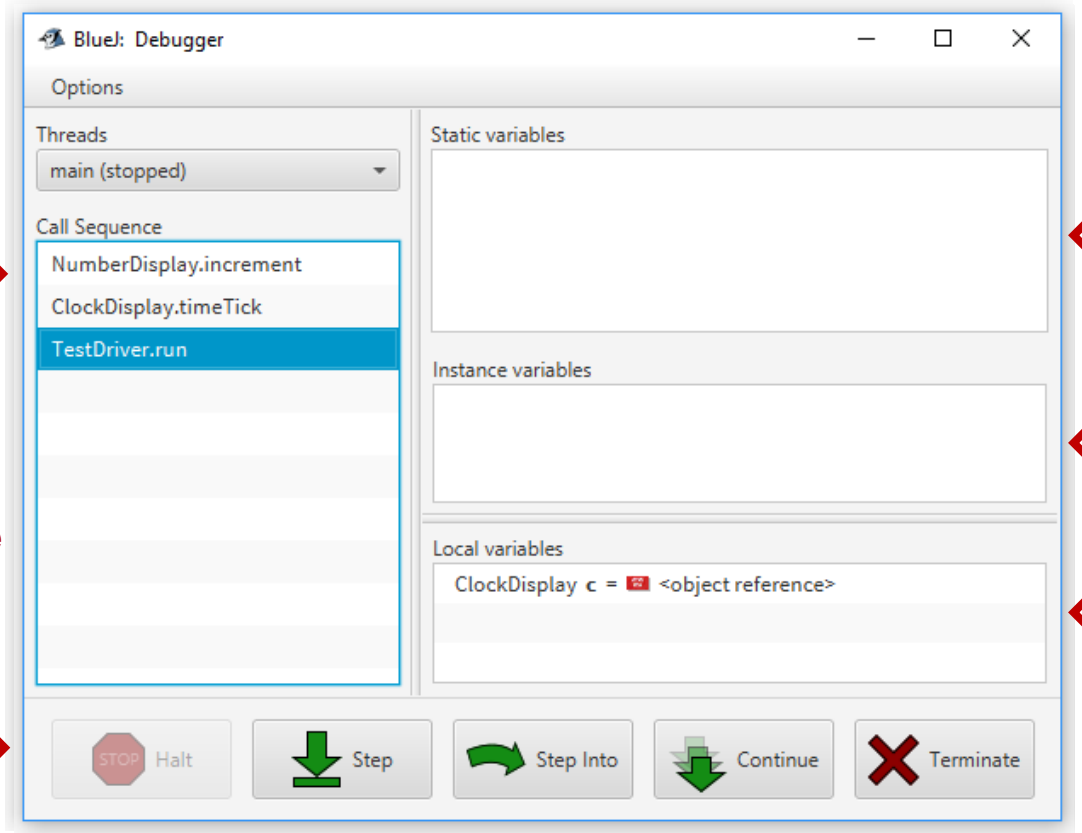
Nødstop (uendelig while løkke eller lignende)



Undervejs kan man inspicere tilstanden

Kaldsstak

- **Viser de igangværende metodekald**
- **Kan bruges til at vælge, hvilke variabler man vil se**

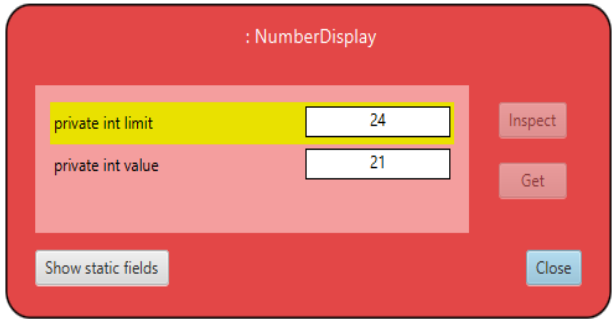


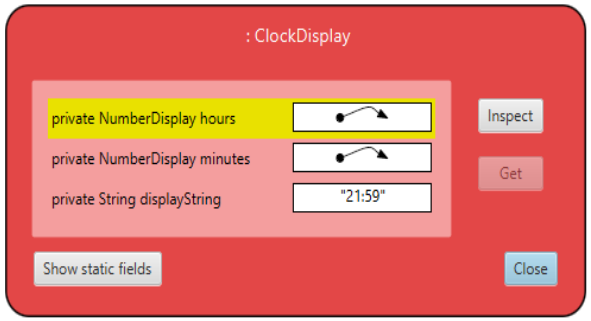
Værdier for klassevariabler

Værdier for feltvariabler

Værdier for lokale variabler (herunder parametre)

Knapper med de forskellige valgmuligheder





Når vi er stoppet ved et breakpoint (eller et statement nået via Step eller Step Into), kan vi inspicere alle variabler i de objekter, der er i gang med at udføre metodekald

Vi vælger det objekt, som vi vil inspicere, via kaldstakken

Eksempel på debugging via print sætninger

- Klassemetoden sort sorterer en liste – metoden er rekursiv

```
public static void sort(ArrayList<Integer> list) {
  if(list.size() > 1) {
    int head = list.get(0);
    list.remove(0);          // Fjern head
    sort(list);              // Rekursivt kald på tail
    insert(head, list);      // Indsæt head på rette sted i list
  }
}
```

1. Vi starter med en usorteret liste [6, 8, 3, 5]
2. Hvis listen har 0 eller 1 elementer er den allerede sorteret
3. Udtag head (første element) 6 [8, 3, 5]
4. Sorter tail (rekursivt kald) 6 [3, 5, 8]
5. Indsæt head på rette plads i tail [3, 5, 6, 8]

   Dette gøres ved hjælp af klassemetoden insert
   (som vi nu vil kigge på)

insert metoden

- Klassemetoden insert indsætter et elem på rette sted i en allerede sorteret list – metoden er rekursiv

```
public static void insert(int elem, ArrayList<Integer> list) {
  if(list.size() == 0) {
    list.add(elem);
  }
  else {
    int head = list.get(0);
    if(elem <= head) {
      list.add(0, elem);     //Indsæt elem forrest i list
    }
    else {
      list.remove(0);        // Fjern head
      insert(elem, list);    // Rekursivt kald på tail
      list.add(head);        // Genindsæt head
    }
} }
```

1. Vi starter med et element og en sorteret liste 6 [3, 5, 8]
2. Hvis listen er tom indsættes elem blot i listen, som så er sorteret
3. Hvis elem <= head indsættes elem forrest i listen, som så er sorteret
4. Ellers udtages head af listen 6 3 [5, 8]
5. Indsæt elem på rette plads i tail 3 [5, 6, 8]
6. Genindsæt head forrest i listen [3, 5, 6, 8]

Testklasse

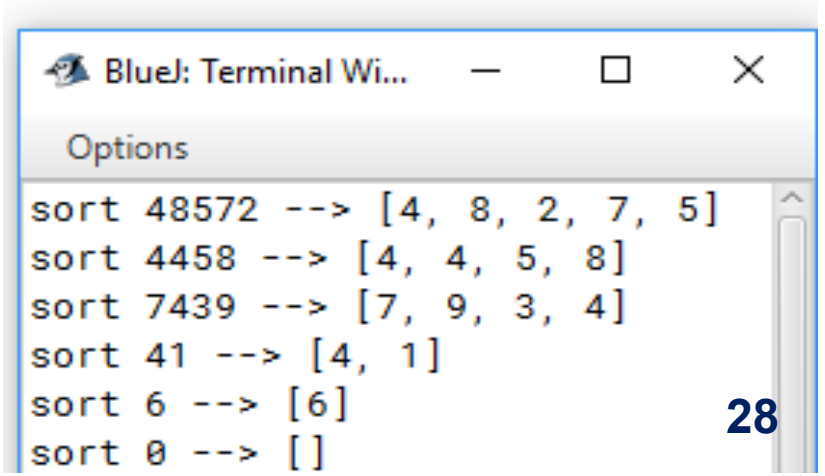
- **Metoden testSort tester sort metoden**
  - Den bruger en hjælpemetode, **createArrayList** som konstruerer en arrayliste ud fra et heltal **48572 → [4, 8, 5, 7, 2]**
  - Blev også brugt i en Raflebæger 3, opgave 7

Sortering ved hjælp af den metode, som vi netop har lavet

```
public void testSort(int digits) {
  ArrayList<Integer> list = createArrayList(digits);
  Sorting.sort(list);
  System.out.println("sort " + digits + " --> " + list);
}
```

Udskriv input og resultat

- **Kørsler af testSort viser at sort metoden fejler**
  - Det er de rigtige elementer vi har i listen
  - Men rækkefølgen er ofte forkert
  - For at lokalisere fejlen vil vi indsætte udskrifter i **sort** og **insert** metoderne



Husk input ved at lave en ny arrayliste med de samme elementer som i list

Kunne man i stedet blot skrive ArrayList<Integer> origList = list?

```
public static void sort(ArrayList<Integer> list) {
  ArrayList<Integer> origList = new ArrayList(list);
  if(list.size() > 1) {
    int head = list.get(0);
    list.remove(0);
    sort(list);
    insert(head, list);
  }
  System.out.println("sort " + origList + " --> " + list);
}
```

Udskriv input og resultat

Husk input

```
public static void insert(int elem, ArrayList<Integer> list) {
  ArrayList<Integer> origList = new ArrayList(list);
  if(list.size() == 0) {
    list.add(elem);
  }
  else {
    int head = list.get(0);
    if(elem <= head) {
      list.add(0,elem);
    }
    else {
      list.remove(0);
      insert(elem, list);
      list.add(head);
    }
  }
  System.out.println("insert " + elem + " in " +
                     origList + " --> " + list);
}
```

- Udskrifterne kommer i den rækkefølge, som metodekaldene afsluttes
- Man skal læse nede fra og opad for at få kaldssekvensen

- Begge metoder fejler
- Vi prøver sommetider at indsætte i usorteret liste
- Vi ser først på insert metoden – hvorfor?

Udskriv input og resultat

BlueJ: Terminal Window - Sorting

Options

```
sort [5] --> [5]
insert 7 in [] --> [7]
insert 7 in [5] --> [7, 5]
sort [7, 5] --> [7, 5]
insert 2 in [7, 5] --> [2, 7, 5]
sort [2, 7, 5] --> [2, 7, 5]
insert 8 in [] --> [8]
insert 8 in [5] --> [8, 5]
insert 8 in [7, 5] --> [8, 5, 7]
insert 8 in [2, 7, 5] --> [8, 5, 7, 2]
sort [8, 2, 7, 5] --> [8, 5, 7, 2]
insert 4 in [8, 5, 7, 2] --> [4, 8, 5, 7, 2]
sort [4, 8, 2, 7, 5] --> [4, 8, 5, 7, 2]
sort 48275 --> [4, 8, 5, 7, 2]
```

insert metoden har tre cases

Indsæt i tom liste

elem <= head

elem > head

```
public static void insert(...) {
  if( list.size() == 0 ) {
    ...
  }
  else {
    int head = list.get(0);
    if( elem <= head ) {
      ...
    }
    else {
      ...
    }
  }
}
```

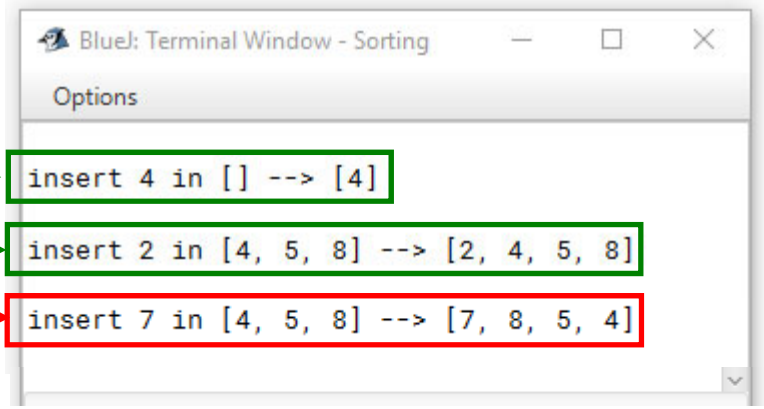
Fejlen ser ud til at ligge i denne del

- **Lad os undersøge, hvilke af disse der fejler**
  - Det ser ud som om det kun er det sidste

Tom liste

elem <= head

elem > head



I praksis bør man selvfølgelig lave nogle flere tests af de tre cases

Nu ved vi, hvor vi skal lede efter fejlen

```
public static void insert(int elem, ArrayList<Integer> list) {
  ArrayList<Integer> origList = new ArrayList(list);
  if(list.size() == 0) {
    list.add(elem);
  }
  else {
    int head = list.get(0);
    if(elem <= head) {
      list.add(0,elem);
    }
    else {
      list.remove(0);          // Fjern head
      insert(elem, list);      // Rekursivt kald på tail
      list.add(0, head);       // Gendindsæt head
    }
  }
  System.out.println("insert " + elem + " in " +
                     origList + " --> " + list);
}
```

- Fejlen ser ud til at ligge i denne del
- Hvad er der galt?

- elem skal genindsættes først i listen
- Vi har genindsat det sidst i listen

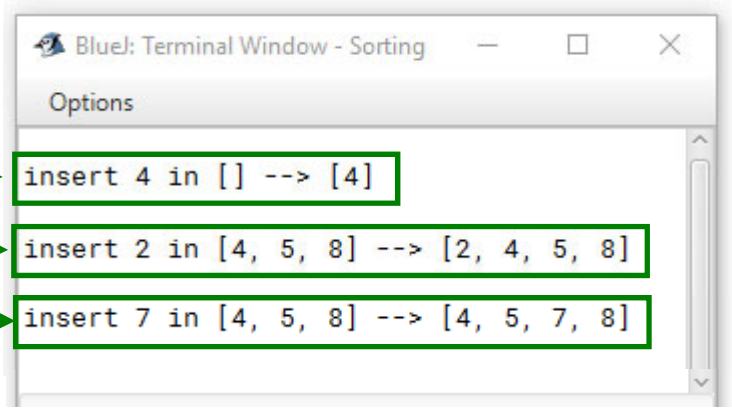
Gentag testene

Nu ser insert ud til at fungere korrekt

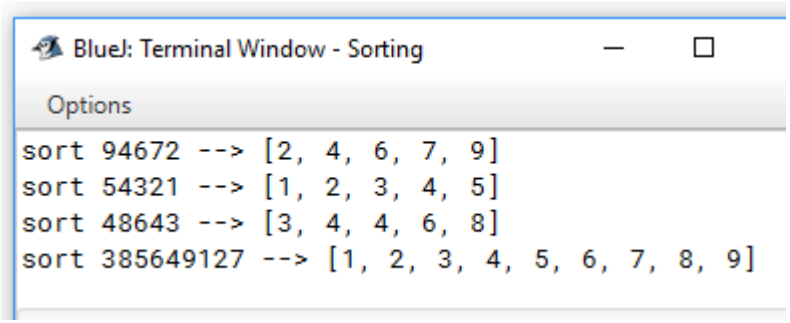
Tom liste →

elem <= head →

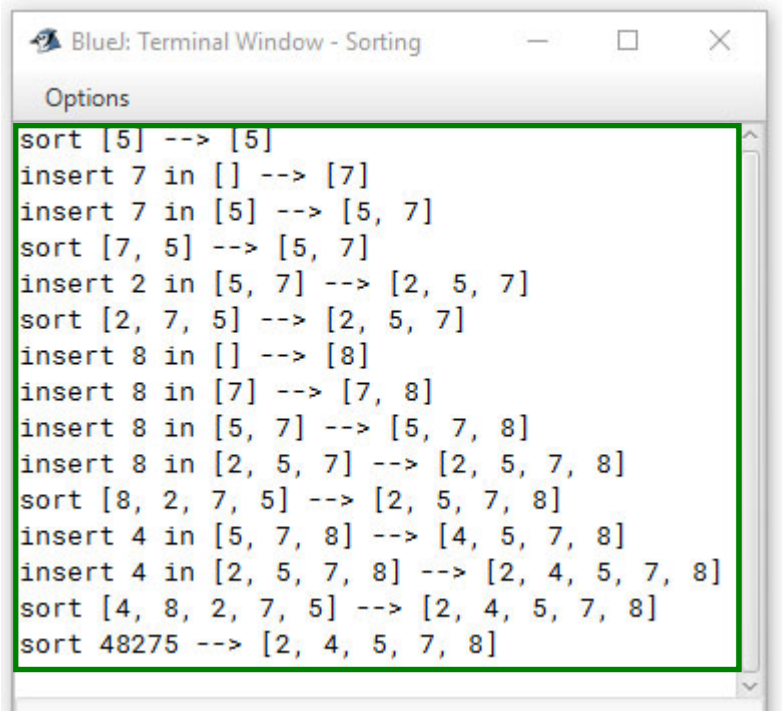
head < elem →



I praksis bør man selvfølgelig lave nogle flere tests



Det samme er tilfældet for sort



Hvad har vi gjort?

- **Ved at gå systematisk frem lokaliserede vi hurtigt fejlen**
- **Både insert og sort fejlede**
  - **sort** kalder **insert** (men ikke omvendt)
  - Vi startede derfor med at kigge på **insert** – i det håb, at **sort** er ok og kun fejlede fordi **insert** var forkert
- **insert metoden har tre cases**
  - Vi testede hver af disse og fandt ud af at den sidste fejlede (elem > head)
  - Den pågældende case bestod af tre sætninger
  - Ved inspektion af disse lokaliserede vi fejlen (som bestod i, at vi genindsatte **head** sidst i listen, hvor det skulle have været placeret først)
- **Den systematiske tilgang bevirkede, at vi hurtigt kunne koncentrere os om tre linjers kode (i stedet for de ca. 20 linjer, der er i de to metoder)**
  - Herefter var det forholdsvis let at finde fejlen

Gode råd omkring test og debugging

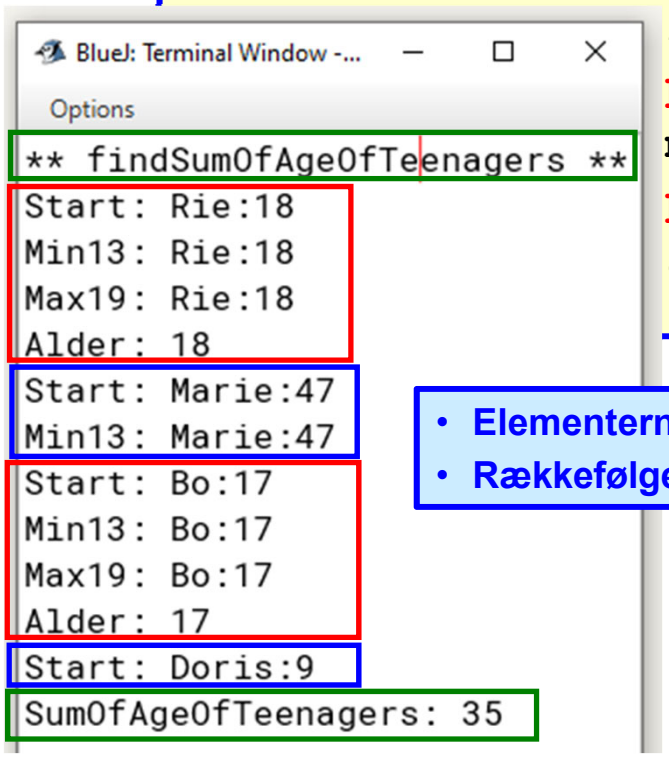
- **Alle programmer indeholder fejl**
  - Anvendelse af gode softwareudviklingsteknikker (herunder indkapsling, cohesion og løs kobling) reducerer antallet af fejl
  - Test bør blive en vane
    - Automatisér test så meget som muligt
    - Husk at opdatere regression tests, når I laver ny kode eller modificerer eksisterende kode
  - Øv jer i at bruge forskellige teknikker til debugging
    - BlueJ's debugger er et fortrinligt værktøj
    - Den kan I med fordel bruge i nogle af de kommende afleveringsopgaver
- **Lav en labrapport / dagbog med de ting, der skal huskes**
  - De foretagne designvalg (inklusiv begrundelser)
  - Aftestning og debugging
  - Mangler, idéer til forbedringer, osv.
- **Værdien af skriftlighed kan ikke overvurderes**
  - Vores hukommelse er forbavsende dårlig
  - Man gentager ofte sine fejl

Debugging af funktionel kode

- **Lambda'er og streams kan debugges som al anden kode**
  - **peek** metoden gør det nemt at lave print sætninger

peek metoden
- **intermediate metode**
- **output stream mage til input stream**

```
public long findSumOfAgeOfTeenagers() {
   return persons.stream()
                 .peek( elem -> print("Start: " + elem ))
                 .filter( elem -> (13 <= elem.getAge()))
                 .peek( elem -> print("Min13: " + elem))
                 filter( elem -> (elem.getAge() <= 19))
                 peek( elem -> print("Max19: " + elem))
                 mapToInt( elem -> elem.getAge())
                 peek( elem -> print("Alder: " + elem))
                 sum();
```



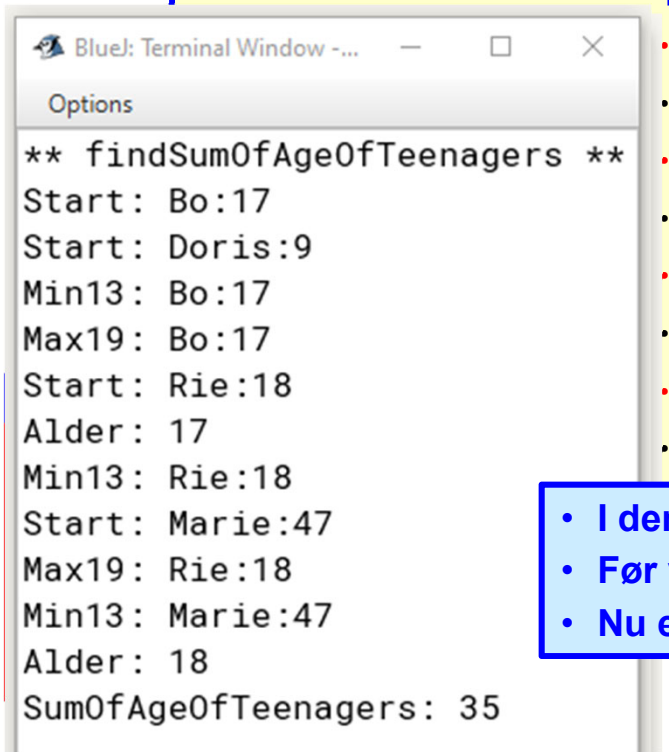
- **Elementerne færdigbehandles ét af gang**
- **Rækkefølgen er deterministisk (ens hver gang)**

```
private void print(Object o) {
   System.out.println(o);
}
```

Parallel processering af streams

- **Hvis vi erstatter stream metoden med parallelStream åbner vi op for multi-core processering**
  - De enkelte kerner (CPU'er) kan behandle forskellige stream elementer parallelt (samtidigt)
  - Potentiel stor **effektivitetsgevinst** uden ekstra programmeringsindsats

```
public long findSumOfAgeOfTeenagers() {
   return persons.parallelStream()
                 .peek( elem -> print("Start: " + elem ))
                 .filter( elem -> (13 <= elem.getAge()))
                 .peek( elem -> print("13 <=: " + elem))
                 .filter( elem -> (elem.getAge() <= 19))
                 .peek( elem -> print("19 >=: " + elem))
                 .mapToInt( elem -> elem.getAge())
                 .peek( elem -> print("mapToInt: " + elem))
                 .sum();
```



- I den viste kørsel behandles Maria or Rie parallelt (samtidigt)
- Før var Doris sidst – nu har hun overhalet Maria og Ria
- Nu er rækkefølgen non-determinístik (forskellige fra gang til gang)

Afleveringsopgave: Raflebæger 4

- **I denne opgave skal I træne konstruktion af regression tests samt debugging, dvs. brug af de teknikker, som er beskrevet i denne forelæsning**
  - I opgave 1, får I udleveret et projekt med en korrekt løsning af **Raflebæger 2**. I skal så lave regression tests for **Die** og **DieCup** klasserne (på samme måde som videoen om "Regression tests" gør det for Raflebæger 1 opgaven)

    Husk at se videoen, før I går i gang med opgaven. Den ligger under Uge 10
  - I opgave 2, får I udleveret et projekt med en ikke helt korrekt løsning af **Raflebæger 3**. I skal så lave regression tests for **Die** og **DieCup** klasserne og bruge disse til at finde og rette fejlene
  - I opgave 3, får I udleveret et projekt, hvor brugeren ved hjælp af en **Game** klasse (og klasserne fra Raflebæger 3) kan rafle mod en computer. Der er imidlertid nogle fejl i **Game** klassen, der gør, at computeren altid vinder. I skal så bruge BlueJ's debugger (eller en lignende) til at finde og rette disse fejl

Opsummering

- **Opremsningstyper**
  - Enumerated types
- **Forskellige teknikker til test og debugging**
- **Afleveringsopgave: Raflebæger 4**

Planen for den mundtlige eksamen er offentliggjort på studieportalen

- **Der er eksamen den 16.18. december samt 6.-9., 13.-15. og 20.-21. januar**
- **Hvis man har påtrængende behov for at blive flyttet til en anden dag, kan man sende mig en mail herom (men vent lige til efter 1. november, hvor I kender tidspunktet for alle jeres eksaminer)**

Deltag aktivt i træningen i mundtlig præsentation

- **Den er uhyre vigtig for jeres succes ved mundtlig eksamen**
- **Eneste gang under jeres studier I får systematisk træning heri**
- **Træning gør mester – de timer I bruger på det, er godt givet ud**
- **Se videoerne om den "perfekte" eksamenspræstation og hør jeres medstuderendes præsentationer – det lærer I også af**

Det var alt for nu….. … spørgsmål