Checklista vid handledande av laborationerna – enbart för handledare

I det här dokumentet finns tips om vad du speciellt bör titta på i samband med att du handleder laborationerna. Det är saker som är speciellt viktiga eller inte fångas upp av testprogrammet. Det finns också en lista med vanliga fel och missuppfattningar. Sist i varje avsnitt finns svar på förberedelse- och diskussionsfrågorna.

Innehåll

Laboration 1 – program med arv

Målet med den här laborationen är att alla ska komma igång med att skriva Java-program med arv. Studenterna får också träna på att läsa klassdiagram.

Test

- En datorspelare ska spela ta stickor" mot användaren. Man tar växelvis en eller två stickor. Den som tar sista stickan vinner. När programmet är klart ska det förstås gå bra att spela mot datorn. Vettiga utskrifter ska finnas så att användaren förstår vad som ska göras. I ett första steg sker kommunikationen via "vanlig" inläsning och utskrift. Det är då ok att förutsätta att användaren skriver in siffror när heltal förväntas.

  Datorspelaren väljer först slumpmässigt en eller två stickor. I en sista deluppgift ska en subklass till `Player` skrivas med en annan strategi. Här får ni ge stor frihet åt studenterna och vilken strategi de väljer är inte så kvistigt. Den vinnande strategin är egentligen ganska enkel. Man ser bara till (om det är möjligt) att antal stickor på bordet är jämnt delbart med tre när det är motståndarens tur. Men det får studenterna komma på själva. Någon kanske istället vill ha en datorspelare som spelar så dåligt som möjligt till exempel.

  Det finns alltså inget fiffigt sätt att se om programmet fungerar än att granska och provköra.

  För de som vill finns en frivillig extrauppgift där dialogrutor används för att kommunicera med användaren.

Kontrollera speciellt

- Att arv använts på rätt sätt. Kontrollera att supertypen `Player` används som typ inuti programmet.

Vanliga fel och missuppfattningar

- -

Förberedelseuppgifter

F3. superklass

F4. Alt. 2 och 3 är korrekta.

Diskussionsuppgifter

D5. Här kan man ha en ganska öppen diskussion och det viktiga är att studenterna förstår hur arv fungerar och vad en abstrakt klass och en abstrakt metod är.

- Fördelarna med arv är bl.a. att vi får en modell som stämmer bättre överens med verkligheten. Koden blir enklare om gemensamma attribut och metoder deklareras en gång i superklassen. Vi får en gemensam typ att använda för objekt av de olika subklasserna. Vi kan t. ex. använda typen `Player` oavsett vilken slags spelare det är frågan om och kan lätt byta ut en spelartyp mot en annan.
- En abstrakt metod består bara av metodrubriken och ska implementeras i en subtyp. Här är det själva strategin i metoden `takePins` som varierar. För att kunna göra följande anrop måste `takePins` vara abstrakt:

  ```
  Player current = ...;
  int n = current.takePins(board);
  ```

  Kompilatorn kommer att söka i klassen `Player` efter metoden `takePins` (och uppåt i arvhierarkin ifall den inte hittar metoden där.)

Laboration 2 – använda abstrakta datatyper

Laborationen är en introduktion till att använda de abstrakta datatyperna `Map`, `List` och `Set`. Idén är att studenterna ska förstå hur datatyperna *används* innan de ger sig på att implementera dem senare under kursen.

Test

- Det är lätt att se att programet fungerar genom att granska utskrifterna.
  - När en andra `TextProcessor` lagts till:

    ```
    nils: 75
    norge: 1
    ```

  - De fem vanligaste landskapen är

    ```
    skåne: 53
    småland: 38
    lappland: 36
    uppland: 25
    dalarna: 24
    ```

  - De ord som förekommer minst 200 gånger är

    ```
    akka: 287
    hörde: 205
    pojken: 1019
    skogen: 247
    snart: 201
    vildgässen: 298
    ```

- Vid jämförelse mellan `HashMap` och `TreeMap` ska man förhoppningsvis se att `HashMap` är lite snabbare i vårt exempel, medan `TreeMap` ger oss ett sorterat resultat.

  Det finns en risk för att exekveringstiden varierar beroende på system, så om ni inte lyckas se någon begriplig skillnad mellan de två implementationerna får vi låta det bero. Då kan du ju nämna att `HashMap` ofta är snabbare, och att de kommer att få implementera en sådan senare.

Kontrollera speciellt

- I klasserna `MultiWordProcessor` och `GeneralWordCounter` används mappar och mängder. Kontrollera att sökmetoderna i respektive klass används och att det inte finns några egna onödiga sökloopar. (Vid sådana fel kan det vara intressant att be studenten mäta tiden före resp efter ändring.)

Vanliga fel och missuppfattningar

- Många missar finessen med listan i `main`-metoden och byter ut en ordprocessor mot en annan istället för att helt enkelt lägga in dem efterhand i listan.
- Listan med entry-objekt ska till slut sorteras i första hand efter antal förekomster, i andra hand i bokstavsordning. Lösningen är *inte* att anropa sort två gånger efter varandra. Se till att studenterna formulerar ett lambda-uttryck där bägge delarna tas om hand. Det kan se ut så här:

```
wordList.sort((w1, w2) -> { // D11
    int res = Integer.compare(w2.getValue(), w1.getValue());
    if (res != 0) {
```

```
            return res;
        } else {
            return w1.getKey().compareTo(w2.getKey());
        }
    });
```

Ovanstående lösning duger bra, men det går också att göra så här (OBS! ingår inte i kursen att kunna):

```
wordList.sort(Comparator.comparingInt(Map.Entry<String, Integer>::getValue).reversed()
        .thenComparing((Map.Entry<String, Integer>::getKey)));
```

- Ett lurigt fel kan dyka upp i samband med jämförelser när de mest frekventa orden ska skrivas ut:
  Jämförelsen sker med avseende på antal förekomster, vilket är värdedelen av `Entry`-objekten, och som har typen `Integer`. I lösningsförslaget används metoden `Integer.compare` för att jämföra de två `Integer`-objekten. Ett alternativ är att anropa `compareTo` på det ena `Integer`-objektet med det andra `Integer`-objektet som argument. Det går också att konvertera `Integer`-objekten till den primitiva typen `int`, antingen genom att införa två nya `int`-variablar eller med unboxing direkt i `if`-satsen. Notera dock att det senare fungerar för jämförelseoperationerna <, <=, >, >= men *inte* för ==, `!=`. Således fungerar följande kod:

  ```
  if (o1.getValue() < o2.getValue()) {
      ...
  } else if (o1.getValue() > o2.getValue()) {
      ...
  }
  ```

  Här kommer operanderna till > och < att implicit konverteras till `int` med unboxing, och jämförelsen utförs då korrekt. Följande kod fungerar dock *inte*:

  ```
  if (o1.getValue() == o2.getValue()) {
      ...
  }
  ```

  Här jämförs de två operanderna med referenslikhet. Således anses de två operanderna som lika endast om de refererar till samma objekt.

Förberedelseuppgifter

F1. De fel som uppträder är:

- `NullPointerException` i metoden `machineWithLeastToDo` på rad 19. Orsaken är att en ny lokal variabel `machines` har deklarerats i konstruktorn och attributet `machines` fortfarande har värdet `null`. Tag bort `Machine[]` i konstruktorn.
- `ArrayIndexOutOfBoundException` i metoden `printSchedule` på rad 49. Orsaken är att man går ett steg för långt i for-satsen. Det ska vara < och inte <=.

Det tal som visas sist i felutskriftens första rad är antingen antal element i vektorn eller det felaktiga indexet (beror på vilken version man kör).

F3. Försäkra dig om att studenterna verkligen har kört debbugern och fråga om det är något särskilt de känner sig osäkra på.

Punkt 4: Felet finns i metoden `makeSchedule` klassen `Scheduler`. De tider som ska jämförs i lambdauttrycket ska kastas om.

Punkt 5: Felet finns i metoden `assignJob` i klassen `Machine`. Det ska vara `+=` istället för `=` (ingen summering av tiderna görs).

F4. Den abstrakta datatyp som skulle kunna användas är prioritetskö.

F6. a) Här ska de se att det är en vanlig lista, och svaren är alltså ”20” och ”Ja”.

b) Nu är det istället en mängd, och då är svaren ”10” och ”Nej”.

c) `<String, Integer>`

d) Svaret är `7`. Här ska de förstå att vi bara kan hålla reda på ett värde för varje nyckel.

e) Metoden `containsKey` har de nytta av i implementationen av `MultiWordCounter`.

Diskussionsuppgifter

D14. Tre första punkterna: Här kan man ha en ganska öppen diskussion. Det är två saker vi vill att de funderar på här:

- Skillnaden mellan ett interface (som `Map`) och en klass som implementerar interfacet (som `HashMap`). Det handlar alltså om att hålla isär en abstrakt datatyp och dess implementation(er).

  Här kan man också komma ihåg att vi (förhoppningsvis) bara behövde ändra på ett ställe för att byta från `HashMap` till `TreeMap`. Det är ett exempel på hur koden blir lättare att vidareutveckla när vi tydligt håller isär interface och implementation.
- Skillnader mellan olika implementationer av `Map`. Det handlar om att de förstår att samma abstrakta datatyp kan implementeras på olika sätt. Man kan exempelvis fråga dem vilken av implementationerna som passar bäst i föregående fråga. (Svaret är förstås ”det beror på”, beroende på om man tycker att prestanda eller sorterat resultat är viktigast.)
- Interfacet `Comparators` roll vid sortering: Här är det meningen att studenten ska förstå att det skickas med ett objekt (av en subtyp till `Comparator`) som argument till `sort`. Och att `compare` anropas på detta objekt vid jämförelser.

Laboration 3 – grafiska användargränssnitt

Meningen med laborationen är att man ska få prova att skriva ett program med användargränssnitt (här med paketet Swing). Dessutom blir det träning på lamdauttryck.

Test

- Om programmet fungerar som det ska brukar koden vara ok. Det räcker oftast att kasta en snabb blick på koden för att kontrollera att det ser normalt"ut. Fråga studenterna vilka valbara uppgifter (ska vara minst 3) som lösts och låt gärna studenten demonstrera dessa.

Kontrollera speciellt

- Om klass som implementerar `Comparator` använts istället för lambdauttryck så tipsa studenten om det senare. Samma sak om anonyma klasser använts istället för lambdauttryck. Låt studenten ändra koden så det blir träning på lambdauttryck.

Vanliga fel och missuppfattningar

- Om `ConcurrentModificationException` kastas så beror det troligen på en trådbugg (race-condition för att vara exakt). Studenten har skapat sin `BookReaderController` innan anropen av `process(word)`. Inuti BookReaderController anropas getWordList som orsakar en iterering av nyckel-värde-paren i den hashmap som håller reda på orden. Om man samtidigt genom att anropa `process` ändrar i hashmappen får man detta exception. Se alltså till att `BookReaderController` skapas sist i main-metoden.

Förberedelseuppgifter

F3. `entrySet`

F4. `e2.getValue() - e1.getValue()`

F6.
a) `addActionerListener`
b) `getText`
c) `add`
d) `setSelectionIndex`

Diskussionsuppgifter

D11.
- I de program studenterna skrivit tidigare är det satserna i main-metoden som styr vad som händer. Men här är det istället användaren som kör programmet som styr genom att klicka på knappar.
- Det passar det bra med lambdauttryck där man till knappens händelsehanterare kan skicka med en ”kodsnutt” som beskriver vad som ska göras när knappen klickas på.
- Parametern till `addAcrtonListnener` har typen `ActionListener` vilket är ett funktionellt interface. Se på den abstrakta metoden i interfacet (`actionPerformed`). Den har en parameter och returtypen `void`.
- Om studenterna följt mallen för fönsterklassen finns ett lamdauttryck i konstruktorn i klassen `BookReaderController`. Det kan se ut så här:

  ```
  SwingUtilities.invokeLater(() -> createWindow(counter, "BookReader", 200, 300));
  ```

  Metoden `invokeLater` har en parameter av typen `Runnable`. I det interfacet finns en abstrakt metod utan parametrar och med returtypen `void`.

- Inuti `SortedListModel` anropas `fireContentsChanged` när listans innehåll förändras. Detta signalerar till `JList` att innehållet måste uppdateras. (`JList` har lagt till sig som lyssnare till `SortedListModel`).

Laboration 4 – länkad lista

Laborationens viktigaste mål är att ge träning på länkad lista. Papper och penna rekommenderas! Uppmuntra studenterna att rita för att bena ut problem.

Just den här laborationen tar en del tid att granska eftersom studenterna ska skriva egna test.

Test

- Tester finns (utom för metoden `append`). Koden i metoden `append` och testerna av den måste granskas speciellt.

Kontrollera speciellt

- Se till att inga extra attribut lagts till (t.ex. något attribut `first` som refererar till första elementet i kön). Inga andra attribut än `last` och `size` ska finnas. (Däremot är det förstås fritt fram att lägga lokala variabler i metoderna. Studenterna brukar vara onödigt försiktiga med det.)
- Kasta en snabb blick på koden för att konstatera att listan är cirkulär och inga konstiga loopar finns.
- Borttagning av element (`poll`): Se till att alla specialfall är med. Fallet lista med ett element brukar glömmas bort (`last` ska få värdet `null`). Detta upptäcks inte av testprogrammet om man i övriga metoder använder `size` för att avgöra om listan är tom eller ej. (Även om klassen fungerar som avsett trots detta kan det bli fel om man senare ändrar i den.)
- Ingen loop (som flyttar elementen ett och ett) ska finnas i `append`. Eftersom implementeringen av listorna är tillgänglig ska detta utnyttjas för att göra implementeringen av `append` effektiv.
- Kontrollera JUnit-testet av `append`. Många slarvar trots följande anvisningarna i labbhäftet: ”Testen ska åtminstone täcka in följande fall för konkatenering:
  - två tomma köer
  - tom kö som konkateneras till icke-tom kö
  - icke-tom kö som konkateneras till tom kö
  - två icke-tomma köer
  - försök att slå ihop en kö med sig själv

  I testen ska du både kontrollera storlek *och* att elementen hamnat i rätt ordning. Glöm inte att kontrollera att den andra kön är tom efter sammanslagningen.”

Övrigt

Ifall frågor dyker upp om inre klasser och statiskt nästlade klasser (`QueueNode` är en sådan) så finns här en tuturial: https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html. Nästlade klasser behandlas också kortfattat i föreläsnnigsbilderna.

Vanliga fel och missuppfattningar

- Man tror att man kan använda de metoder som finns implementerade i `AbstractQueue` och t.ex. anropa `add` inuti `offer`. Detta går inte eftersom det fungerar tvärtom. I `AbstractQueue` implementeras så många som möjligt av metoderna med hjälp av andra metoder. Övriga metoder är abstrakta. Till exempel implementeras `add` (genom att anropa `offer`) och `isEmpty` (genom att utföra `return size() == 0;`). Den som skriver en klass som ärver från `AbstractQueue` måste implementera `offer` och `size` från grunden.
- Man tror att iteratorn ska fortsätta från början då man hämtat sista elementet med `next`.

Förberedelseuppgifter

F3. a) `null`

b) `QueueNode<E> n = last.next;`

c) `E e = last.next.element;`

F4. a) `assertEquals`

b) `assertFalse`

Diskussionsuppgifter

D7.

- 1. Fördelar: Snabbt, enkelt. Nackdelar: Risk att glömma bort att behandla listan som en kö och anropa fel metoder.
  2. Fördelar: Tillgängliga metoder begränsa till kö-operationer, snabbt, enkelt. Nackdelar: ”dubbla metod-anrop”, onödigt komplicerad datastruktur med referenser åt bägge hållen. Men fördelarna uppväger om man inte har några speciella krav.
  3. Fördelar: Kontroll över datastrukturen gör att metoder, som t.ex. `append`, kan skrivas effektivare. Nackdelar: Mer kod att skriva, testa och underhålla.

  Det är mycket ofta lämpligt att använda en befintlig standardklass, som `LinkedList` eller `ArrayDeque`. Att implementera en egen kö, som vi gjort i den här laborationen, är bara lämpligt om vi har speciella krav som gör att standardklasserna inte kan användas.
  Ett exempel på ett sådant speciellt krav är om vi är beroende av en effektiv `append`-metod, så som implementeras i laborationen. Som vi sett kan en sådan metod inte implementeras lika effektivt med hjälp av standardklasserna. Om vår tillämpning inbegriper listor med många element, och `append` visar sig vara viktig för systemets prestanda, så kan det alltså löna sig att implementera en egen kö-klass.
- Nej, man har bara visat att din klass fungerar i just dessa testfall. Det finns oändligt många fall och man kan inte testa alla. Testfallen kan dessutom vara felaktiga eller för ”snälla”.

Laboration 5 – rekursion

Laborationen ska ge träning på rekursion.

Test

- Om den figur som ritas av programmet ser ut som den ska (oregelbunden, men inte för taggig) är programmet troligen korrekt.

Kontrollera speciellt

- Kontrollera att den rekursiva metoden som ritar fraktalen ser ok ut. Beröm dem som beräknat ny mittpunkt i egen privat metod. Tipsa de övriga om detta.
- Kasta en snabb blick i klassen `Side` för kontrollera att `hashCode` (given i uppgiften) och `equals` ser ok ut. Hashtabeller har inte behandlats i kursen när denna laboration görs. Studenterna kan med rätta vara lite konfunderade över var dessa metoder anropas. Diskutera gärna det med dem.

Vanliga fel och missuppfattningar

- Om vita fält inte försvinner i sista deluppgiften så tyder det på att redan behandlade sidor inte hittas i mappen. Börja med att kontrollera att `hashCode` (given i uppgiften) och `equals` är skuggade på vettigt sätt i klassen `Side`.
- Om figuren är för taggig eller för ”jämn” så är det troligen något fel med `dev`. T.ex. ha man ett attribut som heter `dev` och som används inuti metoder istället för att lägga till en parameter.

Förberedelseuppgifter

F3. a) 256

b) 10

Diskussionsuppgifter

D6. Här är det viktigaste att studenterna kan föra en vettig diskussion och förstå svaren, även om de inte nödvändigtvis har alla svaren på plats från början.

- -

Laboration 6 – binära sökträd

Laborationens mål är att ge träning på binära sökträd och därmed på länkade strukturer och rekursion.

Test

- Studenterna ska skriva en main-metod där trädet ska ritas ut före och efter ombyggnad. En JUnit-test av metoderna `add`, `height`, `size` och `clear` finns färdigt.

Kontrollera speciellt

- `add`: Kontrollera att metoden returnerar rätt värde (och inte bara `return true` på slutet). Kontrollera också att jämförelserna görs på rätt sätt och inte med `==`, `equals` eller `compareTo`. Detta fångas troligen inte av testfallen.
- `buildTree`: Se till att trädet byggs upp från början och inte sätts in med `add`. Kontrollera också att tomt träd klaras vid ombyggnaden.

Vanliga fel och missuppfattningar

- Om inget händer vid ombyggnad av trädet kan man ha glömt att tilldela `root` nytt värde.
- Ett vanligt fel i `add` är att man går ända ner till tomma underträd med sina rekursiva anrop. Man tilldelar parametern `n` referensen till den nyskapade noden (`n = new BinaryNode<E>(x);`) och ingenting händer.
  - Det finns en alternativ implementering av `add` som är utformad så att man kan gå ända ner till tomma träd. Den privata hjälpmetoden returnerar då den nya nod som skapas och i anropande upplaga kopplas denna nya nod in som barn. Konsekvensen blir att hjälpmetoden måste returnera en nod och inte (som i lösningsförslaget) en boolean. Men samtidigt finns ju kravet kvar att huvudmetoden `add` ska informera anroparen om insättning kunde genomföras eller ej. Det görs genom att införa en global variabel som deklareras som ett attribut i klassen. Om insättning görs får detta attribut värdet true, annars false. Men attribut ska användas för att representera tillstånd hos objekt av klassen och inte som globala hjälpvariabler. Vi godkänner den lösningen. Men tipsa dem om det bättre alternativ. T.ex. kan man kontrollera storleken före och efter `add` i den publika metoden.

Förberedelseuppgifter

F2. a) 10 20 30 47

b) 3

c) 4

F3. a) `Om trädet är tomt är höjden 0`
`annars är höjden 1 + max(höjden för vänster subträd, höjden för höger subträd)`

b) `max`

F4.
```
int mid = first + ((last - first) / 2);
```
eller (men med risk för overflow) `int mid = (first + last) / 2);`

Diskussionsuppgifter

D9.

- Användaren ska kunna välja hur elementen jämförs. Om konstruktorn med en parameter av typen `Compartor<E>` används kan ett lambdauttryck skickas med. Om konstruktorn utan parameter används ska elementklassen implementera interfacet Comparable.
- Den rekursiva metoden behöver en parameter av typen `BinaryNode<E>` för att hålla reda på vilken nod som är rot i det subträd anropet av den rekursiva metoden gäller. Klassen `BinaryNode` är en privat nästlad klass i trädklassen och inte något som ska synas utåt. Detsamma gäller attributet root. Den publika metoden kan därför inte ha någon parameter av typen `BinaryNode`.
- Eftersom vi snabbt ska hitta mitten passar `LinkedList` mindre bra. Vi måste i så fall gång på gång stega oss fram till mitten i listan.

  En vanlig vektor skulle däremot kunnas användas, men det blir lite pyssel med att hålla ordning på indexen:

```
private int toArray(BinaryNode<E> n, E[] a, int index) {
    if (n != null) {
        index = toArray(n.left, a, index);
        a[index] = n.element;
        index++;
        index = toArray(n.right, a, index);
    }
    return index;
}
```