NTNU
Institutt for datateknikk
og informasjonsvitenskap

Eksamensoppgave i

TDT4100 – Objektorientert programmering

Mandag 6. august 2012, kl. 15:00 - 19:00

Oppgaven er utarbeidet av faglærer Hallvard Trætteberg og kvalitetssikrer Rune Sætre.
Kontaktperson under eksamen er Rune Sætre (mobil 452 18103)

Språkform: Bokmål
Tillatte hjelpemidler: C
Kun Java Pocket Guide, utgitt av O'Reilly forlag, er tillatt.
Legg merke til vedlegget på siste side i oppgavesettet, etter den engelske språkvarianten.

Sensurfrist: Mandag 27. august.

Les oppgaveteksten nøye. Finn ut hva det spørres etter i hver oppgave.

Dersom du mener at opplysninger mangler i en oppgaveformulering, gjør kort rede for de antagelser og forutsetninger som du finner nødvendig.

Del 1 – Innkapsling og konstruktører (15%)

a) Hva er formålet med / begrunnelsen for å implementere en eller flere konstruktører for en klasse?

b) I forhold til innkapsling, hva er hovedgrunnen til å la en konstruktør ta en eller flere parametre?

c) I hvilke situasjoner bør en konstruktør være henholdsvis **public**, **protected** og **private** og hvordan vil kallet til konstruktøren se ut?

d) Gitt følgende klasse:

```
public class V {

	public int v0, v1 = v0++, v2;

	public V() {
		v2 = v0++;
	}
}
```

Hva vil verdien til feltene v0, v1 og v2 være etter at konstruktøren er utført?

e) Java vil ved gitte forutsetninger automatisk opprette en konstruktør for en klasse, slik at instanser av klassen kan opprettes, selv om en ikke eksplisitt har definert en konstruktør for klassen. Hva er betingelsen(e) for at Java skal gjøre dette og hvordan ser denne konstruktøren ut?

Del 2 – Klasser (45%)

I denne oppgaven skal du implementere klasser og metoder for å håndtere en ordliste, f.eks. brukt til å gjøre tekstinput på en mobiltelefon mer effektivt. For alle oppgavene gjelder det at du kan *bruke* andre metoder deklarert i samme eller tidligere deloppgaver, selv om du ikke har implementert dem (riktig).

a) Du skal implementere en klasse **WordList** for å representere en ordliste. **WordList** skal inneholde metoder for å si om et ord finnes, finne alle ord som begynner med et bestemt prefiks og legge til og fjerne ord.

Implementer følgende metoder inkl. nødvendige deklarasjoner av felt og hjelpemetoder:
- **boolean containsWord(String word)**: returnerer om **word** finnes i denne ordlista.
- **Collection<String> getWordsStartingWith(String prefix)**: returnerer en samling av alle ordene i denne ordlista som begynner med **prefix**. Dersom **prefix** selv finnes i ordlista, så skal den også være med. Denne metoden er nyttig for å kunne foreslå ord som passer til det en har begynt å skrive. F.eks. kan en kalle metoden med ordet ”program” og få tilbake en liste som bl.a. inneholder ordene ”program”, ”programmer”, ”programmerer” og ”programmering” (forutsatt at disse faktisk er lagt inn i denne ordlista på forhånd).
- **boolean addWord(String word)**: Legger **word** til denne ordlista, dersom **word** ikke finnes i ordlista fra før. Mellomrom foran og bak skal fjernes, og tomme ord skal ikke legges inn. Returverdien skal angi om ordlista faktisk ble endret.

- **boolean removeWord(String word)**: Fjerner **word** fra denne ordlista. Returverdien skal angi om ordlista faktisk ble endret.
- **boolean removeWordsStartingWith(String prefix)**: Fjerner alle ord som begynner på **prefix** fra denne ordlista. Dersom den kalles med med ordet ”program” så skal bl.a. ordene ”program”, ”programmer”, ”programmerer” og ”programmering” fjernes fra denne ordlista. Som for **removeWord** skal returverdien angi om noen ord i ordlista faktisk ble fjernet.

b) Det er noen ganger nyttig å kunne finne alle ord som har et gitt sett med endelser. Implementer følgende metoder, som er til hjelp for dette.

- **String getPrefix(String word, String suffix)**: Dersom **word** ender på **suffix** så skal prefikset frem til suffix-endelsen returneres. Ellers skal **null** returneres. F.eks. skal **getPrefix(”java-program”, ”program”)** returnere **”java-”**, mens **getPrefix(”java-program”, ”programmering”)** skal returnere **null**.
- **boolean hasSuffixes(String prefix, List<String> suffixes)**: Returnerer **true** dersom **prefix** finnes i denne ordlista med alle endelsene i **suffixes**-lista. Dersom du legger **”tjue-en”** og **”tjue-to”** inn i ordlista og kaller metoden med argumentene **”tjue-”** og en liste med ordene **”en”** og **”to”**, så skal metoden altså returnere **true**.
- **List<String> findPrefixes(List<String> suffixes)**: Returnerer lista av *alle* prefiks som forekommer (i denne ordlista) med *alle* endelsene i **suffixes**. Merk at prefikset selv trenger ikke å være et ord i ordlista. Dersom du legger **”tjue-en”**, **”tjue-to”**, **”tretti-en”**, **”tretti-to”** og **”førti-en”** (og ingen andre ord) inn i ordlista og kaller metoden med en liste med ordene **”en”** og **”to”** som argument, så skal det returneres en liste med ordene **”tjue-”** og **”tretti-”** (men ikke med **”førti-”**).

c) Hvilke(n) av de tre metodene i deloppgave b) kunne vært deklarert som **static** og hvorfor?

Del 3 – Grensesnitt og delegering (20%)

Det innføres et interface **Words**, som deklarerer de *fire* første metodene spesifisert i 2 a), altså **containsWord**, **getWordsStartingWith**, **addWord** og **removeWord**. **WordList** endres til å deklarere at dette grensesnittet implementeres, altså **class WordList implements Words { … }**.

a) Anta så at en deklarerer to variabler som følger:
WordList wordList1 = new WordList();
Words wordList2 = new WordList();
… **wordList1.**??? <= hva kan stå her? …
… **wordList2.**??? <= hva kan stå her? …

Hvordan bestemmer deklarasjonene hvordan **wordList1** og **wordList2** deretter kan brukes, f.eks. hvilke metoder som kan erstatte ”???” over?

b) Hva blir verdiene av de fire uttrykkene **wordList1 instanceof Word**, **wordList1 instanceof WordList** , **wordList2 instanceof Word** og **wordList2 instanceof WordList**?

c) Du skal lage en ny **Words**-implementasjon kalt **DelegatingWordList**, som kombinerer to andre (interne) ordlister med *delegeringsteknikken*. **DelegatingWordList** skal altså oppføre seg om den inneholder alle ordene i de to interne ordlistene. Ta utgangspunkt i følgende felt-deklarasjoner og konstruktør:

```
public class DelegatingWordList implements Words {
```

```
	private Words words1, words2;

	public DelegatingWordList(Words words1, Words words2) {
		this.words1 = words1;
		this.words2 = words2;
	}

	... implementer Words-metodene her ...
}
```

Forklar med tekst og (pseudo)kode hvordan du vil implementere **Words**-metodene med logikken som angitt i deloppgave 2 a) og basert på de interne ordlistene. Prøv å unngå å endre de interne ordlistene mer enn nødvendig.

Del 4 – Input/output (IO) og unntak (20%)

a) Skriv en metode **void read(InputStream input)** i **WordList** som fyller ordlista med ord lest fra den angitte **input**-strømmen. Anta at **input**-strømmen er fra en tekstfil eller tekstlig nettressurs. Hver tekstlinje består enten av et enkeltord eller et prefiks etterfulgt av bindestrek (‘-‘) og så en liste med endelser med komma (‘,’) mellom. Merk at ekstra mellomrom rundt skilletegnene ‘-‘ og ‘,’ må utelates fra prefiks og endelser. Du trenger ikke sjekke om ordene inneholder rare tegn. I tillegg kan en linje inneholde en ‘#’, som betyr at alt fra og med ‘#’-tegnet regnes som en kommentar som skal ignoreres. Alle unntak som kan oppstå, skal overlates til kalleren av **read**-metoden.

Eksempler:

```
java # enkeltordformat: legger “java” inn i lista
# kommentarlinje, ingen ord legges til
2-1,2,3 # prefiks og liste med endelser, legger “21”, “22” og “23” inn i lista
tretti- # prefiks med tom liste av endelser: legger “tretti” inn i lista
```

b) Hva er en *checked exception*? Anta at metoden **m2** bruker metoden **m1** og at **m1** (muligens) kaster en *checked exception*. Da er det to måter å kode **m2** på som gjør at den kompilerer, hvilke?

c) Anta (igjen) at metoden **m2** bruker metoden **m1** og at **m1** (muligens) kaster en *checked exception*. Hvordan kan en kode **m2** slik at den kaster en *unchecked exception* når **m1** kaster en *checked exception*?

NTNU
Institutt for datateknikk
og informasjonsvitenskap

Exam for

TDT4100 – Object-oriented programming

Monday 6. August 2012, 15:00 - 19:00

The exam is made by responsible teacher Hallvard Trætteberg and quality assurer Rune Sætre.
Contact person during the exam is Rune Sætre (mobile 452 18103)

Language: English
Supporting material: C
Only Java Pocket Guide, published by O'Reilly, is allowed.
Note the appendix at the end of this document.

Deadline for results: Monday 27. August.

Read the text carefully. Make sure you understand what you are supposed to do.

If information is missing you must clarify what assumptions you find necessary.

Note the percentages for each part, so you use your time wisely.

Part 1 – Encapsulation and constructors (15%)

a) What is the purpose / reason for implementing one or more constructors for a class?

b) With respect to encapsulation, what is the main reason for making a constructor take one or more parameters?

c) In what cases should a constructor be **public**, **protected** and **private**, respectively, and what will the call to the constructor look like?

d) Given the following class:

```
public class V {

      public int v0, v1 = v0++, v2;

      public V() {
            v2 = v0++;
      }
}
```

What the values do the fields v0, v1 and v2 have after the constructor has been executed?

e) Java will in certain cases automatically create a constructor for a class, so that instances of the class can be created, even if there is no explicitly defined constructor for the class. What are the conditions for this case and what does such a constructor look like?

Part 2 – Classes (45%)

In this part, the task is to implement classes and methods for managing a dictionary (list of words), e.g. for support quicker text input on a mobile phone. Remember that for all programming tasks, you can *use* other methods declared in th same or previous sub-parts, even if you haven't implemented them (correctly).

a) You must implement a class **WordList** for representing a dictionary (collection of words). **WordList** must include methods for telling if a word exists, find all words that begin with a specific prefix and add and remove words.

Implement the following methods including necessary field declarations and utility/helper methods:
- **boolean containsWord(String word)**: returns if this dictionary includes **word**.
- **Collection<String> getWordsStartingWith(String prefix)**: returns the collection of all the words in this dictionary that begins with **prefix**. If the dictionary includes **prefix** itself, it should also be included in the result. This method is useful for suggesting words you might be writing. E.g. you can call it with the string "program" and get a list including "program", "programs", "programmer" and "programming" (provided these have been added to the dictionary).
- **boolean addWord(String word)**: Adds **word** to this dictionary, if the dictionary does not already include **word**. Leading and trailing spaces must be removed, and empty strings should not be added. The return value should indicate if the dictionary actually was modified.

- **boolean removeWord(String word)**: Removes **word** from this dictionary. The return value should indicate if the dictionary actually was modified.
- **boolean removeWordsStartingWith(String prefix)**: Removes all words beginning with **prefix** from this dictionary. If it is called with the string"program" then the words "program", "programs", "programmer" and "programming" must be removed from this dictionary. As for **removeWord**, the return value should indicate if the dictionary actually was modified..

b) It is sometimes useful to find all words with a specific set of endings. Implement the following supporting methods.

- **String getPrefix(String word, String suffix)**: If **word** ends with **suffix** then the prefix up to the suffix must be returned. Otherwise **null** must be returned. E.g. **getPrefix("java-program", "program")** must return **"java-"**, while **getPrefix("java-program", "programming")** must return **null**.
- **boolean hasSuffixes(String prefix, List<String> suffixes)**: Returns **true** if **prefix** exist in this dictionary with all the suffixes in the **suffixes** list. If you add **"twenty-one"** and **"twenty-two"** to the dictionary and call the method with the arguments **"twenty-"** and a list containing the strings **"one"** and **"two"**, then the method should return **true**.
- **List<String> findPrefixes(List<String> suffixes)**: Returns a list containing *all* the prefixes that occur in this dictionary with *all* the suffixes in **suffixes**. Note that prefix itself need not occur in the dictionary. If you add **"twenty-one"**, **"twenty-two"**, **"thirty-one"**, **"thirty-two"** and **"fourty-one"** (and no other words) to this dictionary and call the method with a list containing the strings **"one"** and **"two"** as arguments, then a list containing the strings **"twenty-"** and **"thirty-"** (but not **"fourty-"**) must be returned.

c) Which (one or more) of the three methods in b) can be declared with **static** and why?

Part 3 – Interfaces and delegation (20%)

An interface **Words** is introduced, that declares the *four* first methods specified in 2 a), i.e. **containsWord**, **getWordsStartingWith**, **addWord** and **removeWord**. **WordList** is modified to declare that it implements this interface, i.e. **class WordList implements Words { … }**.

a) Assume two variables are declared as follows:
WordList wordList1 = new WordList();
Words wordList2 = new WordList();
… **wordList1.**??? <= what can be written here? …
… **wordList2.**??? <= what can be written here? …

How do the declarations decide how **wordList1** and **wordList2** henceforth can be used, e.g. which methods that can replace "???" above?

b) What is the value of the four expressions **wordList1 instanceof Word**, **wordList1 instanceof WordList** , **wordList2 instanceof Word** and **wordList2 instanceof WordList**?

c) You must write a new **Words** implementation named **DelegatingWordList**, that combines two other (internal) dictionaries with the *delegation* technique. Thus, **DelegatingWordList** should behave as if it includes all the words in the two internal dictionaries. Assume the following field declarations and constructor:

```
public class DelegatingWordList implements Words {

      private Words words1, words2;

      public DelegatingWordList(Words words1, Words words2) {
            this.words1 = words1;
            this.words2 = words2;
      }

      ... implementer Words-metodene her ...
}
```

Explain with text and (pseudo)code how you will implement the **Words** methods with the logic specified in 2 a) and based on the two internal dictionaries. Minimize the extent to which the internal dictionaries are modified.

Part 4 – Input/output (IO) and exceptions (20%)

a) Write a method **void read(InputStream input)** in **WordList** that fills the dictionary with words read from the provided **input** stream. Assume that the source of the **input** stream is a text file or textual network resource. Each line of text either contains a single word or a prefix followed by a hyphen/dash (‘-‘) and then a list of suffixes separated by comma (‘,’). Note that extra whitespace around the separators ‘-‘ and ‘,’ must be removed from prefixes and suffixes. You do not need to check the words for strange characters. In addition, each line may contain a hash (‘#’), which means that all characters from and including the ‘#’ is a comment that should be ignored. The caller of the **read** method must handle all the exceptions that may occur.

Examples:

```
java # single word format: adds “java” to the dictionary
# comment, no words are added
2-1,2,3 # prefix and list of suffixes, adds “21”, “22” and “23” to the dictionary
thirty- # prefix with an empty list of suffixes: adds “thirty” to the dictionary
```

b) What is a *checked exception*? Assume that the method **m2** uses the method **m1** and that **m1** (possibly) throws a *checked exception*. Which two ways of coding **m2** can be used to ensure it compiles?

c) Assume (again) that the method **m2** uses the method **m1** and that **m1** (possibly) throws a *checked exception*. How can you code **m2** so it throws an *unchecked exception* whenever **m1** throws a *checked exception*?

Appendix

Useful string methods:

`int` **`length`**`():` Returns the length of this string.
`int` **`indexOf`**`(int ch):` Returns the index within this string of the first occurrence of the specified character. If no such character occurs in this string, then `-1` is returned.
`boolean` **`startsWith`**`(String suffix):` Tests if this string starts with the specified prefix.
`boolean` **`endsWith`**`(String suffix):` Tests if this string ends with the specified suffix.
`String` **`substring`**`(int beginIndex, int endIndex):` Returns a new string that is a substring of this string. The substring begins at the specified `beginIndex` and extends to the character at index `endIndex - 1`. Thus the length of the substring is `endIndex-beginIndex`.
`String` **`substring`**`(int beginIndex):` Returns a new string that is a substring of this string. The substring begins with the character at the specified index and extends to the end of this string.
`String` **`trim`**`():` Returns a copy of the string, with leading and trailing whitespace omitted.
`String[]` **`split`**`(String separator):` Splits this string around matches of the given `separator`.