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.

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?

En konstruktør har som formål å *initialisere* et objekt, slik at det fra starten av har en *gyldig* tilstand.

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

Parametrene til en konstruktør er informasjon som trengs for å gi objektet en gyldig starttilstand, som oftest verdien til felt som må være satt og det ikke finnes noen fornuftig default-verdi.

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

public brukes når andre klasser skal kunne instansiere klassen direkte med **new <klasse>(…)**. **protected**-konstruktører brukes for å initialisere felt i en superklasse og kalles av subklassens konstruktør med **super(…)**. **private** brukes på hjelpekonstruktører internt i en klasse og kalles med **this(…)**.

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?

v0 = 2, v1 = 0, v2 = 1. v0 blir til slutt 2 fordi den blir økt to ganger. v1 blir initialisert *før* konstruktøren blir utført og settes til v0 sin verdi *før* v0 økes (første gang). v2 blir initialisert *etter* v1 og settes til v0 sin verdi *før* v0 økes (for andre gang).

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?

Forutsetningen for å automatisk opprette en konstruktør er at den er instansierbare dvs. ikke abstrakt, og det ikke er definert noen andre konstruktører. En slik konstruktør vil være **public** og ha tom parameterliste.

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 bokstavene i **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 den 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.

Vi velger å bruke en Collection til å holde ordlista. add- og remove-metodene kan brukes direkte, siden de returnerer om lista ble endret. Skal en gjøre det selv, må en initialisere en lokal variabel til et uttrykk/løkke med med contains og returnere denne etter add/remove/removeAll-kallet.

```
private Collection<String> wordList = new ArrayList<String>();

public boolean containsWord(String s) {
	return wordList.contains(s);
}

public Collection<String> getWordsStartingWith(String s) {
	Collection<String> matchingWords = new ArrayList<String>();
	for (String word : wordList) {
		if (word.startsWith(s)) {
			matchingWords.add(word);
		}
	}
	return matchingWords;
}

public boolean addWord(String s) {
	s = s.trim();
	return (s.length() > 0 && (! wordList.contains(s)) ? wordList.add(s) :
false);
}
```

```
public boolean removeWord(String s) {
	return wordList.remove(s);
}

public boolean removeWordStartingWith(String s) {
	return wordList.removeAll(getWordsStartingWith(s));
}
```

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-”**).

```
	public String getPrefix(String word, String suffix) {
		if (word.endsWith(suffix)) {
			return word.substring(0, word.length() - suffix.length());
		}
		return null;
	}

	public boolean hasSuffixes(String prefix, List<String> suffixes) {
		for (String suffix : suffixes) {
			if (! containsWord(prefix + suffix)) {
				return false;
			}
		}
		return true;
	}

	public List<String> findPrefixes(List<String> suffixes) {
		List<String> prefixes = new ArrayList<String>();
		for (String word : wordList) {
			String prefix = getPrefix(word, suffixes.get(0));
			if (prefix != null && hasSuffixes(prefix, suffixes)) {
				prefixes.add(prefix);
			}
		}
		return prefixes;
	}
```

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

getPrefix kan være **static** siden den ikke bruker felt eller metoder som er ikke-**static**.

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** kan brukes lenger ned i koden?

Den *deklarerte* typen bestemmer hvilke metoder en kan kalle, uavhengig av om objektet som variabelen refererer til har andre/flere metoder. Altså vil en for **wordList1** kunne kalle alle metodene i **WordList**, mens en for **wordList2** kun kan kalle metodene i **Words**.

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

Alle gir **true**, fordi **instanceof** tar utgangspunkt i den faktiske (dynamiske) typen til objektet det refereres til.

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.

Poenget er å forstå hva som er riktig logikk og hvordan det må delegeres til hver av de to listene.

```
public boolean containsWord(String s) {
      return words1.containsWord(s) || words2.containsWord(s);
}

public Collection<String> getWordsStartingWith(String s) {
      List<String> matchingWords = new ArrayList<String>();
      matchingWords.addAll(words1.getWordsStartingWith(s));
      matchingWords.addAll(words2.getWordsStartingWith(s));
      return matchingWords;
}

public boolean addWord(String s) {
      if (! containsWord(s)) {
            words1.addWord(s);
            return true;
      }
      return false;
}

public boolean removeWord(String s) {
      return words1.removeWord(s) | words2.removeWord(s);
}
```

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

a) Lag en metode **void read(InputStream input)** i WordList som fyller ordlista med ord lest fra den angitte **input**-strømmen. Du kan 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 skal overlates til kalleren av metoden.

Eksempler:

```
java # enkeltordformat: legger “java” inn i lista
# kommentarlinje, ingen ord
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
```

Unntaket må deklareres med throws. Input må omsluttes med en Reader for å håndtere tekstkoding riktig. De ulike formatene må håndteres (kommentar som hele eller deler av linja, enkeltord, prefiks med endelse).

```
public void read(InputStream input) throws IOException {
      BufferedReader reader = new BufferedReader(new InputStreamReader(input));
      String line = null;
      while ((line = reader.readLine()) != null) {
            int pos = line.indexOf('#');
            if (pos >= 0) {
                  line = line.substring(0, pos);
```

```
            }
            pos = line.indexOf('-');
      if (pos < 0) {
                  addWord(line);
            } else {
                  String prefix = line.substring(0, pos).trim();
                  String[] suffixes = line.substring(pos + 1).split(",");
                  for (int i = 0; i < suffixes.length; i++) {
                        addWord(prefix + suffixes[i].trim());
                  }
            }
      }
}
```

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

En Exception som ikke er en RuntimeException er en checked exception. En slik unntakstype krever enten try/catch eller en throws-deklarasjon for å unngå kompileringsfeil.

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

En må fange unntaket med en try/catch og så kaste en ny unchecked exception, altså RuntimeException eller en egnet subklasse:

```
try { ... kode som kaster checked exception ...}
catch (Exception e) { throw new RuntimeException(e);}
```

Appendix

Useful methods in the String class:

`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`.