NTNU
Institutt for datateknikk
og informasjonsvitenskap

Eksamensoppgave i

TDT4100 – Objektorientert programmering

Torsdag 12. august 2010, kl. 09:00 - 13:00

Oppgaven er utarbeidet av faglærer Hallvard Trætteberg og kvalitetssikret av Svein Erik Bratsberg. Kontaktperson under eksamen er Hallvard Trætteberg (mobil 918 97263)

Språkform: Bokmål
Tillatte hjelpemidler: C
Én valgfri lærebok i Java er tillatt.
Bestemt, enkel kalkulator tillatt.

Sensurfrist: Torsdag 2. september.

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 det nødvendig å gjøre.

Del 1 – Typer (15%)

Anta at du har tre klasser `A`, `B` og `C` og ett grensesnitt `G`. `B` og `C` arver fra `A` (altså med `extends`) og `B` implementerer `G`. Anta også at du har følgende variabeldeklarasjoner (og initialiseringer):

```
A a = new A();
B b = new B();
C c = new C();
G g = null;
```

a) Fyll ut følgende tabell med "tillatt" eller "ulovlig", for å angi om tilordningene `a = a`, `a = b`, `a = c`, `a = g`, `b = a`, … `g = g` er tillatt eller ulovlig, iht. Java-kompilatoren (Eclipse-editoren). Som du ser er deler av tabellen fylt ut, fordi en variabel alltid kan tilordnes sin egen verdi, dvs. `a = a`, `b = b`, `c = c` og `g = g` alle er tillatt.

	`a`	`b`	`c`	`g`
`a =`	tillatt	tillatt	tillatt	ulovlig
`b =`	ulovlig	tillatt	ulovlig	ulovlig
`c =`	ulovlig	ulovlig	tillatt	ulovlig
`g =`	ulovlig	tillatt	ulovlig	tillatt

b) Fyll ut følgende tabell med "true" eller "false", for å angi om `instanceof`-uttrykkene gir `true` eller `false` som resultat. Som du ser er deler av tabellen fylt ut, for å angi at `a instanceof A`, `b instanceof B` og `c instanceof C` alle gir `true`, dvs. at `System.out.println(a instanceof A)`, `System.out.println(b instanceof B)` og `System.out.println(c instanceof C)` alle skriver ut "true".

	`A`	`B`	`C`	`G`
`a instanceof`	true	false	false	false
`b instanceof`	true	true	false	true
`c instanceof`	true	false	true	false
`g instanceof`	false	false	false	false

c) Noen av `instanceof`-uttrykkene vil alltid gi false uavhengig av hvilken verdi variablene faktisk har, og disse vil derfor bli markert som ulovlige av Java-kompilatoren. Hvilke `instanceof`-uttrykk er av denne grunn ulovlige?

En B kan aldri være en C eller omvendt (søsken i arvetreet), så `b instanceof C` og `c instanceof B` kan beviselig aldri bli true.

Del 2 – Memory (25%)

Memory er et spill hvor en må huske lengre og lengre sekvenser av f.eks. tall, bokstaver eller farger. Skriv klassen **Memory** som implementerer et tekstbasert memory-spill hvor en må huske sekvenser av tall/sifre. Legg vekt på å gjøre koden ryddig.

Et eksempel på interaksjon er vist nedenfor. Systemets utskrift er vist i *kursiv*, mens brukerens er i **fet** skrift. `\c` er en fiktiv spesialkode som istedenfor å synes (slik den gjør under), blanker ut det som er skrevet ut og inn tidligere, slik at en unngår juks med copy & paste.

```
Tall 1: 2
2
```

```
\cRiktig!
Tall 2: 1
21
\cRiktig!
Tall 3: 2
212
\cRiktig!
Tall 4: 4
2123
\cFeil!
Du klarte 3 tall!
```

En må ha en liste som fylles med nye tilfeldige tall og kode (helst i en metode) som sammenligner lista med tall (på en eller annen form) som brukeren skriver inn. Hovedprogrammet blir en løkke som lager et nytt tilfeldig tall, legger det inn i lista, leser input fra brukeren og sammenligner inputet med tall-lista.

```
public class Memory {

	private List<Integer> numbers = new ArrayList<Integer>();

	int nextNumber() {
		int next = (int)(Math.random() * 10);
		numbers.add(next);
		return next;
	}

	int digitValue(char digit) {
		return digit - '0';
	}

	boolean compareNumbers(String digits) {
		if (digits.length() != this.numbers.size()) {
			return false;
		}
		for (int i = 0; i < digits.length(); i++) {
			if (digitValue(digits.charAt(i)) != this.numbers.get(i)) {
				return false;
			}
		}
		return true;
	}

	public static void main(String[] args) {
		Memory memory = new Memory();
		Scanner scanner = new Scanner(System.in);
		while (true) {
			int next = memory.nextNumber();
			System.out.println("Tall " + memory.numbers.size() + ": " + next);
			String line = scanner.nextLine();
			if (line == null) {
				break;
			} else if (memory.compareNumbers(line)) {
				System.out.println("Riktig!");
			} else {
				System.out.println("Feil!");
				break;
			}
		}
```

```
            System.out.println("Du klarte " + (memory.numbers.size() - 1) + " tall!");
      }
}
```

Del 3 – GPS (30 %)

Du skal implementere (deler av) et system for å håndtere GPS-data. Et GPS-punkt (**GPSPoint**) består av to koordinater (desimaltall). En GPS-måling (**GPSSample**) består av et GPS-punkt og et tidspunkt (antall sekunder relativt til et eller annet starttidspunkt).

a) Lag klasser for GPS-punkt (**GPSPoint**) og GPS-måling (**GPSSample**) slik at de inneholder nødvendige felt, konstruktører og metoder. Informasjonen skal *ikke* kunne endres etter at objektene er laget.

Med disse klassene implementert, skal det være lov å skrive følgende kode:

```
GPSPoint point = new GPSPoint(72.4, 32.5);
GPSSample sample = new GPSSample(point, 724567);
```

En trenger felt med riktig type. Siden informasjonen ikke skal kunne endres, så trengs en konstruktør som tar inn to verdier og en kan ikke ha set-metoder. En bør ha get-metoder, og kan deklarere feltene final:

```
public class GPSPoint {

      private final double v1, v2;

      public GPSPoint(double v1, double v2) {
            this.v1 = v1;
            this.v2 = v2;
      }

      public double getV1() {
            return v1;
      }

      public double getV2() {
            return v2;
      }
}
```

En trenger felt med riktig type. Samme logikk som over når det gjelder konstruktører og metoder. Vi har tatt med en konstruktør som tar inn koordinater, lager GPSPoint-objektet selv og kaller den andre konstruktøren.

```
public class GPSSample {

      private final GPSPoint point;
      private final long timestamp;

      public GPSSample(GPSPoint point, long timestamp) {
            this.point = point;
            this.timestamp = timestamp;
      }
      public GPSSample(double v1, double v2, long timestamp) {
            this(new GPSPoint(v1, v2), timestamp);
      }
```

```
	public GPSPoint getGPSPoint() {
		return point;
	}

	public long getTimestamp() {
		return timestamp;
	}
}
```

b) Anta det finnes en **GPSUtil**-klasse med metoden **static double distance(GPSPoint pkt1, GPSPoint pkt2)**, som returnerer avstanden mellom to GPS-punkt i meter. Implementer metodene **double distance(GPSPoint)** og **double distance(GPSSample)** i **GPSSample**-klassen, slik at disse returnerer avstanden (i meter) fra et **GPSSample**-objekt til parameteret (henholdsvis **GPSPoint**-objektet og **GPSSample**-objekt).

Med disse metodene implementert, skal det altså være lov å skrive følgende kode:

```
GPSPoint point = new GPSPoint(72.4, 32.5);
GPSSample sample = new GPSSample(point, 724567);
System.out.println(sample.distance(point));
System.out.println(sample.distance(sample));
```

De to siste linjene skal begge skrive ut **`0`** (tallet 0), siden avstanden fra et punkt til seg selv er 0.

Her er poenget å forstå hvordan vanlige metoder er knyttet til en eksisterende instans og kan kalle den statiske metoden med data knyttet til denne instansen (`this`) og data knyttet til parameteret (hhv. `pt` og `sample`).

```
	public double distance(GPSPoint pt) {
		return GPSUtil.distance(this.point, pt);
	}

	public double distance(GPSSample sample) {
		return GPSUtil.distance(this.point, sample.point);
	}
```

c) GPS-dataene skal lagres i og håndteres av klassen **GPSData**, som en liste med GPS-målinger. Skriv **GPSData**-klassen. Det skal (i første omgang) kun være mulig å legge til nye GPS-målinger og lese dem én og én vha. en **Iterator**. Lag metoder for dette.

En må han en intern liste og en metode for å legge til en GPS-måling i denne. Iteratoren hentes ut fra den interne lista.

```
public class GPSData implements Iterable<GPSSample> {

	private List<GPSSample> samples;

	public void addGPSSample(GPSSample sample) {
		samples.add(sample);
	}

	public Iterator<GPSSample> iterator() {
		return samples.iterator();
	}
}
```

d) Hva må til (hvordan må **GPSData**-klassen kodes) for at en skal kunne iterere over GPS-målingene i et **GPSData**-objekt vha. for-each-løkker, slik:

```
GPSData gpsData = …
for (GPSSample sample: gpsData) {
   … kode med sample her …
}
```

`GPSData`-klassen må implementere `Iterable<GPSSample>` for at for-each-syntaksen skal kunne brukes. Da må metoden som returnerer iteratoren deklareres og navngis som over.

Del 4 – Ekstra GPSData-metoder (15 %)

a) For lengre turer kan det bli mange GPS-målinger å lagre. Skriv en metode **int reduce()** i **GPSData**-klassen som går gjennom lista og fjerner **GPSSample**-objekter som 1) ligger nærmere forrige måling i tid enn 30 sekunder, eller 2) er mindre enn 15 meter unna forrige måling. Metoden skal returnere antall GPS-målinger som ble fjernet.

Denne kan være litt vrien. Trikset er å huske forrige liste-element og sammeligne med denne. Det er unaturlig å bruke en for-løkke siden lista skal endres underveis.

```
	public void reduce() {
		GPSSample lastSample = null;
		int i = 0;
		while (i < samples.size()) {
			GPSSample sample = samples.get(i);
			if (lastSample != null && (sample.getTimestamp() -
lastSample.getTimestamp() < 30 || sample.distance(lastSample) < 15)) {
				samples.remove(i);
			} else {
				lastSample = sample;
				i++;
			}
		}
	}
```

b) Lag en metode **boolean contains(GPSPoint point, double distance)** i **GPSData**, som returnerer true dersom det inneholder en GPS-måling som er nærmere **point** enn **distance**.

```
	public boolean contains(GPSPoint pt, double distance) {
		for (GPSSample sample: samples) {
			if (sample.distance(pt) < distance) {
				return true;
			}
		}
		return false;
	}
```

c) Lag en metode **GPSSample closest(GPSPoint)** som returnerer den målingen i lista som er nærmest det oppgitte punktet.

```
	public GPSSample closest(GPSPoint pt) {
		GPSSample closest = null;
		for (int i = 0; i < samples.size(); i++) {
			GPSSample sample = samples.get(i);
			if (closest == null || sample.distance(pt) < closest.distance(pt)) {
```

```
                        closest = sample;
                    }
                }
                return closest;
        }
```

Del 5 – Testing (15%)

a) Forklar hvordan du vil teste **GPSPoint** og **GPSData** . Eksemplifiser med testkode og kommenter eventuelle forskjeller i testeteknikken. Vi er ikke så nøye på detaljene, bare den generelle testeteknikken kommer tydelig frem.

GPSPoint er enkel å teste, siden den ikke er modifiserbar. En oppretter objekter med konstruktøren og sjekker at get-metodene gir riktig verdi med en assert-metode. GPSData kan endres og må en må derfor sjekke i flere trinn. En må legge til én og én GPS-måling og sjekke at iteratoren gir de samme elementene i samme rekkefølge. Ekstra-metodene må sjekkes med ulike sett med GPS-målinger, inkludert sett med 0 og 1 GPS-måling, for å se om logikken er riktig implementert.