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			
`b =`		tillatt		
`c =`			tillatt	
`g =`				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			
`b instanceof`		true		
`c instanceof`			true	
`g instanceof`				

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?

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!
```

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);
```

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.

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.

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 …
}
```

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.

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**.

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

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.

NTNU
Institutt for datateknikk
og informasjonsvitenskap

Eksamensoppgåve i

TDT4100 – Objektorientert programmering

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

Oppgåva er utarbeidd av faglærar Hallvard Trætteberg og kvalitetssikra av Svein Erik Bratsberg. Kontaktperson under eksamen er Hallvard Trætteberg (mobil 918 97263)

Språkform: Nynorsk
Tillatte hjelpemidler: C
Éi valfri lærebok i Java er tillaten.
Bestemt, enkel kalkulator tillaten.

Sensurfrist: Torsdag 2. september.

Les oppgåveteksten nøye. Finn ut kva det blir spurt etter i kvar oppgåve.

Dersom du meiner at opplysningar manglar i ei oppgåveformulering, gjer kort greie for dei føresetnader som du finn det naudsynt å gjere.

Del 1 – Typer (15%)

Anta at du har tre klasser `A`, `B` og `C` og eitt grensesnitt `G`. `B` og `C` arvar frå `A` (altså med `extends`) og `B` implementerer `G`. Anta også at du har følgjande variabeldeklarasjonar (og initialiseringar):

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

a) Fyll ut følgjande tabell med "tillaten" eller "ulovleg", for å angi om tilordningane `a = a`, `a = b`, `a = c`, `a = g`, `b = a`, … `g = g` er tillatne eller ulovlege, iht. Java-kompilatoren (Eclipse-editoren). Som du ser er delar av tabellen fyld ut, fordi ein variabel alltid kan tilordnast sin eigen verdi, dvs. `a = a`, `b = b`, `c = c` og `g = g` alle er tillatne.

	`a`	`b`	`c`	`g`
`a =`	tillaten			
`b =`		tillaten		
`c =`			tillaten	
`g =`				tillaten

b) Fyll ut følgjande tabell med "true" eller "false", for å angi om `instanceof`-uttrykkane gir `true` eller `false` som resultat. Som du ser er delar av tabellen fyld 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 skriv ut "true".

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

c) Nokre av `instanceof`-uttrykkane vil alltid gi false uavhengig av kva for verdi variablane faktisk har, og desse vil difor bli markert som ulovlege av Java-kompilatoren. Kva for `instanceof`-uttrykk er av denne grunn ulovlege?

Del 2 – Memory (25%)

Memory er eit spel kor ein må hugse lengre og lengre sekvensar av t.d. tal, bokstavar eller fargar. Skriv klassen **Memory** som implementerer eit tekstbasert memory-spel kor ein må hugse sekvensar av tal/sifre. Legg vekt på å gjere koden ryddig.

Eit døme på interaksjon er vist nedanfor. Systemets utskrift er vist i *kursiv*, medan brukarens er i **fet** skrift. `\c` er ein fiktiv spesialkode som i staden for å synast (slik den gjer under), blankar ut det som er skriven ut og inn tidlegare, slik at ein 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!
```

Del 3 – GPS (30 %)

Du skal implementere (delar av) eit system for å handtere GPS-data. Eit GPS-punkt (**GPSPoint**) består av to koordinatar (desimaltal). Ei GPS-måling (**GPSSample**) består av eit GPS-punkt og eit tidspunkt (antall sekund relativt til eit eller anna starttidspunkt).

a) Lag klasser for GPS-punkt (**GPSPoint**) og GPS-måling (**GPSSample**) slik at dei inneheld naudsynte felt, konstruktører og metodar. Informasjonen skal *ikkje* kunne endrast etter at objekta er laga.

Med desse klassene implementert, skal det være lov å skrive følgjande kode:

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

b) Anta det finst ein **GPSUtil**-klasse med metoden **static double distance(GPSPoint pkt1, GPSPoint pkt2)**, som returnerer avstanden mellom to GPS-punkt i meter. Implementer metodane **double distance(GPSPoint)** og **double distance(GPSSample)** i **GPSSample**-klassen, slik at desse returnerer avstanden (i meter) frå eit **GPSSample**-objekt til parameteret (henholdsvis **GPSPoint**-objektet og **GPSSample**-objekt).

Med desse metodene implementert, skal det altså vere lov å skrive følgjande 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));
```

Dei to siste linjene skal båe skrive ut `0` (tallet 0), siden avstanden frå eit punkt til seg sjølve er 0.

c) GPS-dataene skal lagrast i og handterast av klassen **GPSData**, som ein liste med GPS-målingar. Skriv **GPSData**-klassen. Det skal (i fyrste omgang) kun vere mogleg å leggje til nye GPS-målingar og lese dei ein og ein vha. ein **Iterator**. Lag metodar for dette.

d) Kva må til (korleis må **GPSData**-klassen kodast) for at ein skal kunne iterere over GPS-målingane i eit **GPSData**-objekt vha. for-each-løkker, slik:

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

Del 4 – Ekstra GPSData-metodar (15 %)

a) For lengre turar kan det bli mange GPS-målingar å lagre. Skriv ein metode **int reduce()** i **GPSData**-klassen som går gjennom lista og fjernar **GPSSample**-objekt som 1) ligger nærare forrige måling i tid enn 30 sekundar, eller 2) er mindre enn 15 meter unna forrige måling. Metoden skal returnere talet på GPS-målingar som blei fjerna.

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

c) Lag ein metode **GPSSample closest(GPSPoint)** som returnerer den målinga i lista som er nærast det oppgitte punktet.

Del 5 – Testing (15%)

a) Forklar korleis du vil teste **GPSPoint** og **GPSData**. Eksemplifiser med testkode og kommenter eventuelle skilnader i testeteknikken. Vi er ikkje så nøye på detaljane, bare den generelle testeteknikken kjem tydeleg fram.

NTNU
Institutt for datateknikk
og informasjonsvitenskap

Exam for

TDT4100 – Object-oriented programming

Thursday 12. August 2010, 09:00 - 13:00

The exam is made by responsible teacher Hallvard Trætteberg and quality assurer Svein Erik Bratsberg. Contact person during the exam is Hallvard Trætteberg (mobile 918 97263)

Language: English
Supporting material: C
One printed Java textbook is allowed. Other printed books are not allowed.

Deadline for results: Thursday 2. September.

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 – Types (15%)

Assume three classes `A`, `B` and `C` and an interface `G`. `B` and `C` inherits from `A` (using `extends`) and `B` implements `G`. In addition, assume the following declarations (and initializations):

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

a) Fill inn the following table with ”allowed” or ”illegal”, to indicate which of the assignments `a = a`, `a = b`, `a = c`, `a = g`, `b = a`, … `g = g` are legal (or not) according to the Java compiler (Eclipse editor). Some table cells have already been filled in, because a variable can always be assigned to its own value, i.e. `a = a`, `b = b`, `c = c` and `g = g` are all allowed.

	a	**b**	**c**	**g**
a =	allowed			
b =		allowed		
c =			allowed	
g =				allowed

b) Fill inn the following table with ”true” or ”false”, to indicate if the `instanceof` expressions give `true` or `false` as result. Some table cells have already been filled in, to indicate that `a instanceof A`, `b instanceof B` and `c instanceof C` are all `true`, i.e. that `System.out.println(a instanceof A)`, `System.out.println(b instanceof B)` and `System.out.println(c instanceof C)` will all print ”true”.

	A	**B**	**C**	**G**
a instanceof	true			
b instanceof		true		
c instanceof			true	
g instanceof				

c) Some of the `instanceof` expressions will always be false independent of the actual value of the variables, and these expressions will therefore be marked as illegal by the Java compiler. Which `instanceof` expressions are illegal for this reason?

Part 2 – Memory (25%)

Memory is a game where you must remember longer and longer sequences of e.g. digits, characters or colors. Write a **Memory** class that implements a text-based memory game where you must remember sequences of digits. Take care to write tidy code.

An example of interaction is shown below. The system’s output is shown in *italics*, while the user’s input is **bold**. `\c` is a fictional code which is not output (as it is below), but instead blanks previous input and output, to avoid cheating with copy & paste.

```
Digit 1: 2
2
\cCorrect!
```

```
Digit 2: 1
21
\cCorrect!
Digit 3: 2
212
\cCorrect!
Digit 4: 4
2123
\cWrong!
You managed 3 digits!
```

Part 3 – GPS (30 %)

You must implement (part of) a system for managing GPS data. A GPS point (**GPSPoint**) consists of two coordinates (floating point numbers). A GPS sample (**GPSSample**) consists of a GPS point and a time point (number of seconds since some starting point).

a) Write classes for GPS point (**GPSPoint**) and GPS sample (**GPSSample**) that includes necessary fields, constructors and methods. It should *not* be possible to change the information after the objects have been created.

Given these classes, the following code should be legal:

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

b) Assume there is a **GPSUtil** class with the method **static double distance(GPSPoint pkt1, GPSPoint pkt2)**, which returns the distance between two GPS points in meters. Implement the methods **double distance(GPSPoint)** and **double distance(GPSSample)** in the **GPSSample** class, so they return the distance (in meters) from a **GPSSample** object to the parameter (the **GPSPoint** object and the **GPSSample** object, respectively).

Given these methods, the following code should be legal:

```
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));
```

The last two lines must both print `0` (the number 0), since the distance from a point to itself is 0.

c) The GPS data must be stored in and managed by the **GPSData** class, as a list of GPS samples. Write the **GPSData** class. It should (at this point) only be possible to add new GPS samples and read them one by one using an **Iterator**. Implement corresponding methods.

d) How must the **GPSData** class be coded, to make it possible to iterate over all the GPS samples in a **GPSData** object with a for-each loop, as follows:

```
GPSData gpsData = …
for (GPSSample sample: gpsData) {
   … code using sample here …
}
```

Part 4 – Extra GPSData methods (15 %)

a) For longer trips there will be mange GPS samples to store and manage. Write the method **int reduce()** in the **GPSData** class, that goes through the list and removes **GPSSample** objects that 1) is closer in time to the previous sample than 30 seconds, or 2) is closer in distance to the previous sample than 15 meter. The method must return the number of GPS samples that were removed.

b) Write the method **boolean contains(GPSPoint point, double distance)** in the **GPSData** class, that returns true if it contains a GPS sample that is closer to **point** than **distance**.

c) Write a method **GPSSample closest(GPSPoint)** that returns the GPS sample in the list that is closest to the provided point.

Part 5 – Testing (15%)

a) Explain how you would test **GPSPoint** and **GPSData** . Exemplify with test code and comment on possible differences in the testing technique. We are not concerned about the details, as long as the general testing method you employ is clear.