2019-08-30.kl.08-12

# Tentaupplägg

**Allmänna Tips**

Läs igenom ALLA uppgifterna. Välj den du känner är lättast först. Det kan gärna ta 10-20 minuter. Försök skriva saker som kan vara problem i uppgifterna. Är det något du absolut kommer att fastna på så kanske det är fel uppgift att ge sig på. Tiden du lägger på att noga läsa uppgifterna tjänar du in på att välja rätt uppgift. Kolla ibland till kommunikationsfönstret. Det kan ha kommit information till alla utan att ni skickat in en fråga. Om ni har problem med eclipse eller dylikt som INTE har med uppgifterna att göra, räck upp handen så kommer en assistent. Detsamma gäller om hur man kopierar givna filer. Frågor om själva uppgifterna tar vi i första hand via tentasystemet. Testa ditt program noga innan du skickar in. Om din lösning inte är korrekt kommer du få ett kompletteringsmeddelande och har möjlighet att rätta den och skicka in igen. Kompletteringar måste dock åtgärdas. Vi tolererar inte att ni spammar oss med samma lösning om och om igen.

| ***Kommando*** | ***Effekt*** |
|---|---|
| `cp given_files/* .` | Kopierar alla givna filer till mappen du står |
| `xdg-open given_files/tenta.pdf` | Öppnar själva tentan. Blir läsbar då tentan startar. |
| `eclipse &` | Startar eclipse. |

**Generella krav (gäller för alla uppgifter om inget annat anges):**

- Fullständiga uppräkningar och kodduplicering skall undvikas med klasshierarkier och polymorfi i de fall det inte går att generalisera på andra sätt (underprogram, loopar, etc.).
- Korrekt inkapsling krävs, d.v.s. rätt synlighet på instans/klassvariabler och metoder.
- Lämpliga konstruktorer skall finnas för alla klasser.
- Instansvariabler får inte användas som globala/lokala variabler (d.v.s. använd parametrar och lokala variabler i första hand).
- Korrekt användning av static krävs.
- Överanvändning av ***instanceof*** och typkonverteringar kan leda till komplettering.

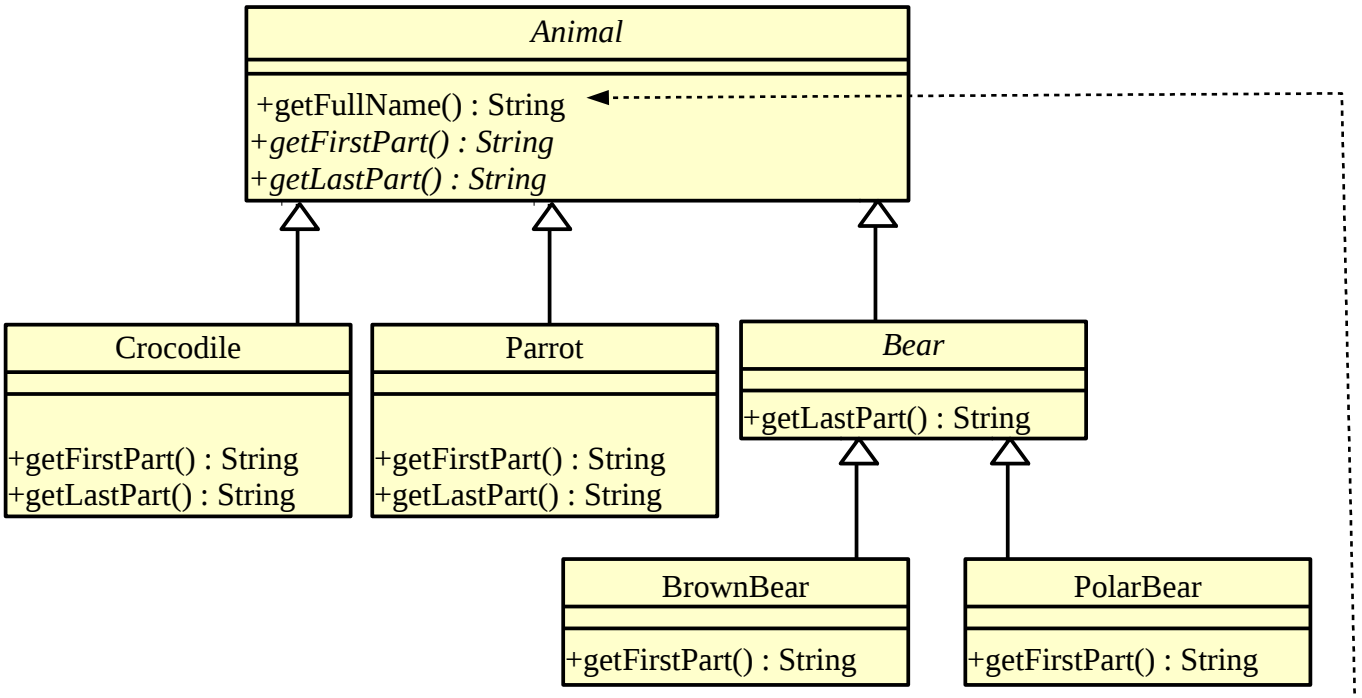
| **Betygsgränser:** | **Tid** | **TDDE10/11** | **725G90** |
|---|---|---|---|
| 1 poäng | 12:00 | Betyg 3 | Betyg G |
| 3 poäng | 11:00 | Betyg 4 | |
| 3 poäng | 10:40 | | Betyg VG |
| 3 poäng | 10:20 | Betyg 5 | |
| ≥4 poäng | 11:00 | Betyg 5 | |
| ≥4 poäng | 11:20 | | Betyg VG |

**OBS:** Delpoäng delas inte ut på uppgifterna. För att få poäng på en uppgift måste man alltså lösa uppgiften helt (och enligt specifikation).
M.v.h.
/Erik och Magnus

# Uppgift 1 - Korsa djur [1p]

Pappa frågar Ciri vad man får om man korsar en Krokodil och en Papegoja.
"En Krokogoja!" ropar Ciri och tjuter av skratt.
"Ja, men skulle det inte kunna vara en Papedil?" kontrar Pappa.
Ciri blir med ens tyst och fundersam. "Undrar vilka andra djur man kan korsa" tänker hon...

Implementera följande klasser som visas i UML:diagrammet:



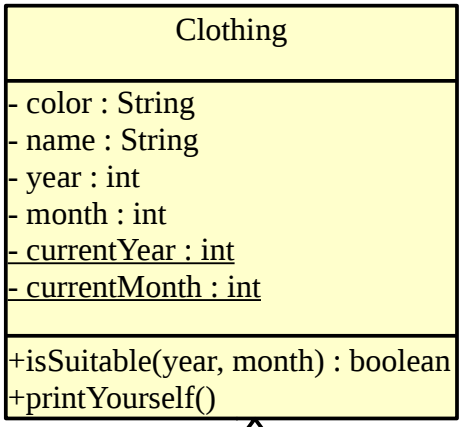

**KRAV:** Låt getFullName() i Animal anropa de andra två metoderna. Det hela namnet skall ju alltid vara sammanslagningen av djurets två delnamn. Problemet är dock att en ärvande klass fortfarande kan (felaktigt) överskugga getFullName(). För att förbjuda detta så **skall** denna metod deklareras som ***final***.

Lägg till den kod som behövs i TestAnimals.java för att djuren faktiskt skall skapas. Kör sedan testprogrammet för att få se många roliga namn på korsade djur.

# Uppgift 2 - Barnkläder [1p]

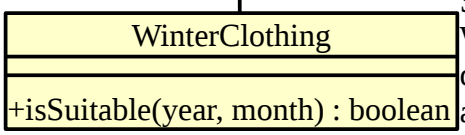
Ciri håller på att skapa ett nytt datorsystem för att hålla rätt på sina kläder. Hon vill kunna lagra många olika typer av plagg och dessutom om de är inne eller uteplagg. Hon bestämmer sig därför för att skapa en präktig klasshirarki för dessa data!

Klassen Clothing representerar plagg i allmänhet. Alla Ciri's plagg har en storlek. Det är dock inte en storlek i cm eller så utan snarare en undre tidsgräns då det är troligt att plagget kommer att passa Ciri. Ett exempel är hennes nya jeansjacka, som passar barn vid 18 månader. Denna tidsgräns representeras med två heltal, ett för år och ett för månader. (18 månader är alltså 1 och 6). Plagg har också en beskrivning och en färg (båda är strängar). Metoden printYourself() skriver ut information om plagget.




Tanken är också att plaggen skall kunna ange huruvida de är aktuella just för tillfället, så att man vet vilka plagg som skall plockas fram/bort. Detta gör man genom att titta på tidsgränsen. Man skall därför kunna anropa metoden *isSuitable(birthyear, birthmonth)*, där man anropar metoden med Ciris födelsetid*. Metoden returnerar sant om plagget är lämpligt (d.v.s. barnets nuvarande ålder är lika med eller överstiger plaggets "storlek").

För att detta skall fungera så måste klassen Clothing därför också lagra vilket som är nuvarande år och nuvarande månad. Rimligtvis är dessa två attribut samma för alla instanser av Clothing och bör sättas till 2019 och 08, men kan självklart ändras senare (getters och setters kan vara användbart). Detta innebär alltså att om man senare ändrar nuvarande år/månad så skall det slå igenom i alla Clothing-objekt.




Som ett specialfall av plagg finns vinterkläder (representerad av WinterClothing). Denna klass fungerar precis som sin förälder med det tillägg att isSuitable även kontrollerar huruvida det är vinter. Vi anser att det i sverige är vinter mellan december (månad 12) och februari (månad 2). Om det är vinter så är vinterkläder inte lämpliga.

Du skall implementera klasserna som beskrivs/visas ovan i texten/UML:en.

Se testprogrammet TestClothing.java för hur klasserna skall instansieras och hur utskrifterna från printYourself()-metoderna skall se ut.

---

* Detta är praktiskt om man vill använda systemet för ett annat barn som är fött vid annat tillfälle.

# Uppgift 3 - Cirkulär lista [2p]

Ibland, t.ex. när man i en grupp personer turas om att göra något vill man ha ett roterande schema. Man behöver då en abstrakt datastruktur som lagrar alla personer men som också automatiskt hanterar att "det går runt" så att det alltid finns en nästa person som är ansvarig. Det smidigaste då är att strukturen hela tiden håller reda på var man "står", d.v.s. att man har en pekare till det element som är aktuellt.

Du skall skapa klassen MyCircularList som tar en generisk parameter T och lagrar ett godtyckligt antal T i en cirkulär struktur. ADT:n MyCircularList skall ha följande operationer.

- add(T) som lägger till en T i strukturen (där vi för nuvarande "står").
- next() som returnerar elementet där vi för närvarande "står" och sedan stegar vidare.

Observera att *add(T)* alltså lägger till elementet "först" och om man upprepat använder denna så kommer strukturen bli "baklänges". Vill man lägga till "sist" istället så får man anropa add() och sedan direkt anropa next(). Om du vill kan du ha en metod som heter *addLast(T)* som gör precis detta, men det är inte ett krav.

Om vi t.ex. tänker oss att vi stoppar in bokstäverna A, B och C och sedan anropar next() tre gånger så borde vi få ut C,  B och sedan A. Om vi sedan anropar next() igen så bör vi få ut C igen, o.s.v.

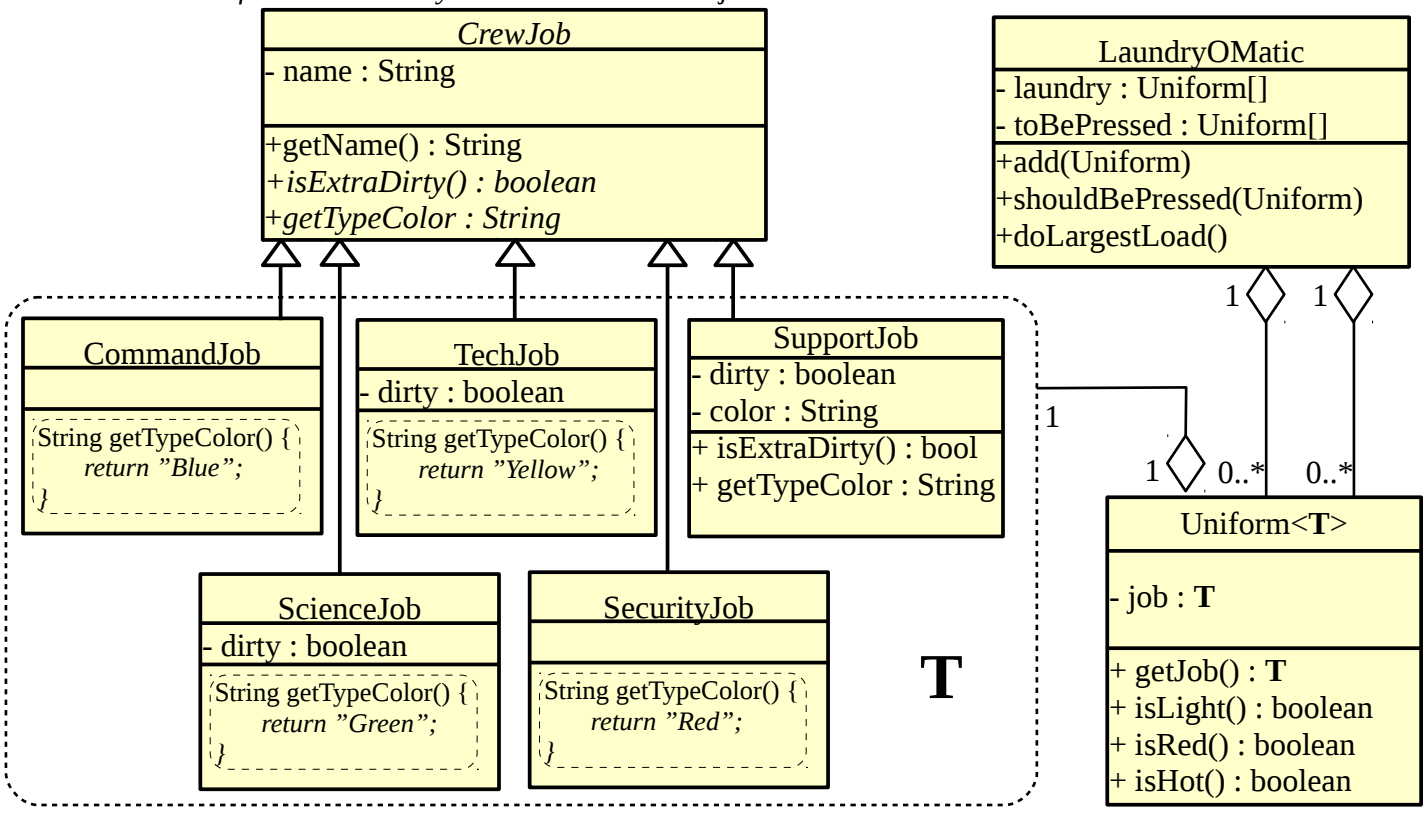
**Användningsexempel (finns även på *TestMyCircular.java*):**

```
MyCircularList<Character> list = new MyCircularList<Character>();
list.add('A');
list.add('B');
list.add('C');

for (int i = 4; i > 0; --i) {
    System.out.print(list.next() + ", "); // Vi bör få ut C, B, A, C
}
System.out.println(" ");
list.add('D');
for (int i = 5; i > 0; --i) {
    System.out.print(list.next() + ", "); // Vi bör få ut D, B, A, C, D
}
System.out.println(" ");

MyCircularList<String> whoPays = new MyCircularList<String>();
whoPays.add("Jenny");
whoPays.add("Steph");      // Här hade det varit trevligt med en addLast
whoPays.next();
whoPays.add("Claire");     // och här...
whoPays.next();
whoPays.add("Jack");       // och här.
whoPays.next();
while (true) {
    System.out.println(whoPays.next());  // Borde bli Jenny, Steph,
                                         // Claire, Jack, Jenny, Steph,
                                         // Clair, Jack, ...
}
```

**TIPS:** Om du använder en struktur av länkade noder, håll även reda på "föregående" element.

# Uppgift 4 - Uniformer [3p]

På rymdstationen *SS13* har personalchefen Connor fått nog av kaoset! Han bestämer sig för att ett datorsystem behövs för att hålla reda på all tvätt. Vilka plagg skall tvättas först, och vilka uniformer kan tvättas tillsammans (ljus/mörk/röd tvätt, och temperatur). Det finns många olika jobb på stationen t.ex. forskare, läkare, bartendrar, vaktmästare, tekniker, säkerhetspersonal, kaptenen m.m. För att systemet skall bli så bra och generellt som möjligt (Connor har även funderat på andra system som skulle kunna vara praktiska, t.ex. lönesystem, ledighetsansökningar, m.m.) vill han ha en ordentlig (och separat) klasshirarki för personalen och sedan klasserna *Uniform* och *LaundryOMatic* som hanterar själva tvätten.




Implementera klasserna i UML-diagrammet ovan. Klassen TestLaundry.java är given och hjälper dig att se hur klasserna skall instansieras och användas. Det som är speciellt här är att klassen Uniform skall lagra en instans av typen T, där T är en generisk parameter och någon av de fem subklasserna till CrewJob. Varje jobb på stationen har en dedikerad färg som även anger färgen på uniformen. Metoden isLight() returnerar sant om färgen är vit eller gul. Om färgen är röd eller rosa returnerar isRed() sant behöver tvättas separat. Om uniformen varken är ljus eller röd så är det mörk tvätt. Vissa uniformer behöver tvättas i högre temperatur. Detta gäller för de job där uniformen kan bli extra smutsig (representeras av metoden isExtraDirty()). För CommandJob och ScurityJob är detta alltid falskt, men för övriga jobbtyper så beror det lite på vad det är för jobb.

Klassen LaundryOMatic lagrar två uppsättningar med uniformer. All tvätt lagras i variabeln *laundry* och man lägger till tvätt med metoden add(). CommandJob-uniformer skall även pressas men det är mycket viktigt att ingan andra typer av uniformer pressas (av ekonomiska och säkerhetsskäl). Därför skall metoden shouldBePressed() **endast** gå att anropa med Uniform<CommandJob>, allt annat skall ge kompileringsfel. Metoden doLargestLoad() skall först räkna vilken typ av plagg (ljust, mörkt, rött) som skall köras först och sedan tvätta dessa plagg. Om något av plaggen var extra smutsigt behöver hela tvätten köras i 60 grader (istället för 40). Slutligen, om det var en mörk tvätt så skall alla Command-uniformer pressas.