Lösningsförslag till instuderingsuppgifter läsvecka 2

1.

Vid implementationsarv ärver en klass implementationen (koden) från en annan klass. Vid specifikationsarv ärver en klass specifikationer från ett interface.

2.

Multipelt arv innebär att en klass ärver från mer än en klass eller ett interface. Java tillåter implementationsarv endast från en klass, varför multipelt arv i Java alltid involverar specifikationsarv från minst ett interface.

3.

Specialisering innebär att man från en superklass skapar subklasser – subklasserna utökar den uppsättning egenskaper och beteenden som superklassen har. Generalisering innebär att gemensamma egenskaper och beteenden hos flera klasser samlas i en gemensam superklass.

4.

Här är ett problem med multipelt arv:

```
public class Medic {
   private Date birthDate;
   private int examinationYear;
   public void operate() { /*...*/ }
}// Medic

public class Engineer {
   private Date birthDate;
   private int examinationYear;
   public void construct() { /*...*/ }
}// Engineer

public class MedicAndEngineer extends Medic, Engineer {
   . . .
}//MedicAndEngineer
```

En MedicAndEngineer är både Medic och Engineer, och klassen MedicAndEngineer ärver därför egenskaperna birthDate och examinationYear *två* gånger, från Medic respektive Engineer. Hur ska dessa kollisioner hanteras? Hur många födelsedatum och examensår ska en MedicAndEngineer ha?

För födelsedatum är det naturligt att slå ihop de två instansvariablerna till en enda variabel, för en person har bara ett födelsedatum, och detta är samma oavsett yrke. Men en person som är både läkare och ingenjör har (förmodligen) två olika examensår, så examensåren ska inte slås ihop till en enda variabel. Det är svårt för kompilatorn att förstå skillnaden.

Det kan även uppstå namnkonflikter på metoder. Två klasser som ärvs ifrån kan ha metoder med samma namn och profil, men metoderna gör helt olika saker.

5.

En abstrakt klass kan föredras framför ett interface om det finns beteenden som delas mellan alla subklasser och som man vet inte kommer att förändras.

Abstrakta klasser används för att faktorisera ut duplicerad kod i arvshierarkier eller då man vill tvinga subklasser att implementera en metod (t.ex. som i design mönstret Template method).

6.

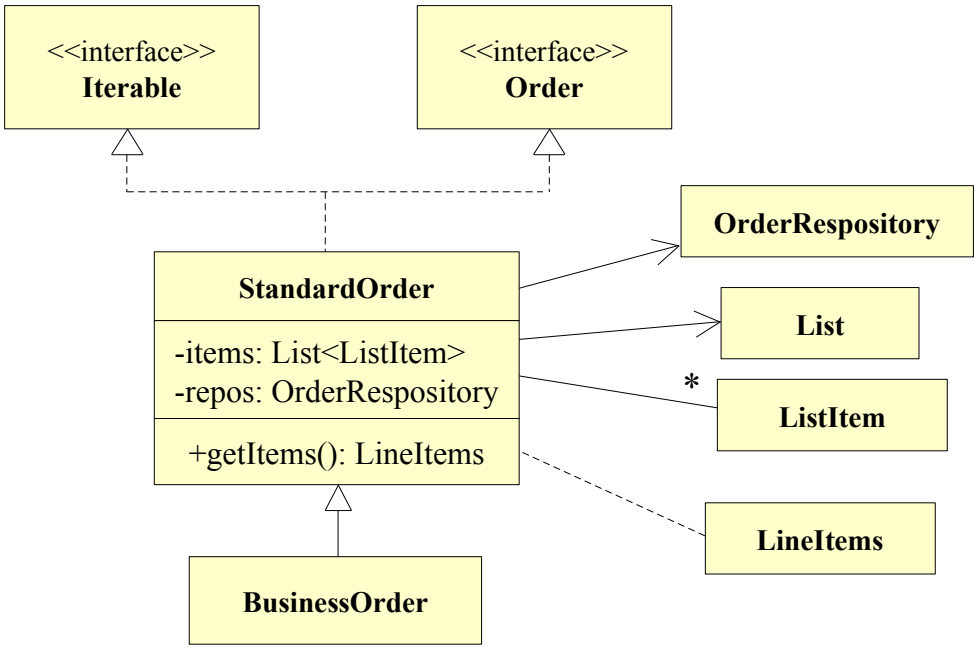
Fem relationer finns definierade: Generalisering, realisering, association, inkapsling samt beroende.

7.

Vid aggregation kan ett objekt vara associerat med flera ägarobjekt. Försvinner ägarobjektet/ägarobjekten kommer det associerade objektet att leva vidare. Aggregation beskriver ”känner till”-relationen.

Vid komposition kan en objekt endast vara associerat med ett ägarobjekt. Försvinner ägarobjektet, så försvinner också det associerade objekt. Komposition beskriver ”har”-relationen (eller ”del av”-relationen).

8.



9.

Se föreläsningsanteckningarna.

10.

Se föreläsningsanteckningarna.

11.

Se föreläsningsanteckningarna.

12.

Om S är en subtyp till T, så är mängden av objekt i typen S en delmängd till mängden av objekt i typen T, och mängden av operationer i typen T är en delmängd av mängden operationer i typen S.

13.

Ett programmeringsspråk är starkt typat om typkompabilitet för alla uttryck som representerar värden kan avgöras från programkoden vid kompilieringstillfället. Starkt typade programspråk har således statisk typkontroll.

14.

Se föreläsningsanteckningarna.

15.

Se föreläsningsanteckningarna.

16.

Se föreläsningsanteckningarna.

17.

Se föreläsningsanteckningarna.

18.
Se föreläsningsanteckningarna.

19.
Se föreläsningsanteckningarna.

20.
Se föreläsningsanteckningarna.

21.
Se föreläsningsanteckningarna.

22.
Om en metod är annoterad med @Override, men metoden inte överskuggar någon metod ger kompilatorn ett kompileringsfel.

23.
Kovarians innebär att en subklass kan ersätta returtypen i en metod som överskuggas med en subtyp till returtypen som superklassen anges för den överskuggade metoden.

24.
Begreppet inkapsling (*encapsulation*) betyder att data och de operationer som kan utföras på denna data sammanförs till en enhet, dvs i det objektorienterade paradigmet till en klass. Informationsdöljning (*information hiding*) innebär extern åtkomst till datan i den inkapslade enheten (=klassen) endast kan ske via ett väldefinierat publikt gränssnitt.

25.
En mutator-metod är en metod som förändrar tillståndet hos ett objekt.

26.
Instanser av en icke-muterbar klass förändrar inte sina tillstånd under sin livstid.

27.
Nej, inte nödvändigtvis! Exempelvis kan en instansvariabel vara publikt tillgänglig, varför tillståndet i ett objekt kan förändras.

28.

a) A.A()

b) A.A()
B.B()

c) A.A()
B.B()
C.C()

d) A.A()
B.B()
C.C()

e) A.f(A)

f) B.h(a) statisk metod innebär statisk bindning!

g) C.f(A)

h) B.g(A)

i) C.g(B)

j) compile time error klassen A har definierar ingen metod h

k) B.g(A)

29.

a) Ger utskriften ”c( ) in D”

b) Ger utskriften ”true”

c) Tilldelningen C x = **new** D() ger kompileringsfel, eftersom klassen D är abstrakt.

d) Ger utskriften ”b( ) in E”

e) Anropet x.b() ger kompileringsfel, eftersom typen C inte har operationen b().

f) Ger exekveringsfel. Typen F kan inte typomvandlas till typen D.

g) Tilldelningen D x = **new** C() ger kompileringsfel, eftersom typen C inte är subtyp till D.

h) Tilldelningen C y = x ger kompileringsfel, eftersom x är av typen A och A är ingen subtyp till C.

30.

Fel i Main.java

rad 6: obj.f4() är fel, eftersom C2 inte har någon metod f4().

rad 12: func(**new** C1()) är fel, eftersom C1 är en abstrakt klass.

rad 15: func2(**new** C2()) är fel, eftersom C2 inte är en subtyp till C3.

Om dessa fel avlägsnas blir utskriften:

```
C2.f1
C2.f2
C1.f3
++C2.f1++
++C3.f2++
++C1.f3++
++C3.f4++
```

31.

```
public interface A {. . .}
public interface B extends A {. . .}
public interface D {. . .}
public abstract class C implements B {. . .}
public class E extends C implements D {. . .}
```

32.

```
public class A {
   private B objB = new B();
   private C objC;
   public A(C objC) {
      this.objC = objC;
   }
   . . .
}

public class B{. . .}

public class C {. . .}

public class D {
   private C objC;
   public D(C objC) {
      this.objC = objC;
   }
   . . .
}
```