Objektorienterad Programmering DAT043

Föreläsning 12: Repetion och Sammanfattning
26/2 -18
Moa Johansson

I denna föreläsning kommer vi repetera mycket av det viktiga i kursen. Därmed inte sagt att denna föreläsning är “komplett”, vissa saker i kursen kommer vi naturligtvis inte ha tid att ta upp idag. Men ställ gärna frågor!

Objektorienterad programmering i Java

- Java är **imperativt** och **objektorienterat**.
- Kod organiseras i **klasser**.
- Objekt-typer beskrivs av **instansvariabler** i klassen.
  - **Instans** av en klass: värde/variabel av objekttyp.
- Finns även **klassvariabler** (static) i.e. globala variabler.

Studenter ska veta vad dessa begrepp betyder (se föreläsning 1).
Objektorientering: Hur programmet struktureras efter vilka “objekt” som vi vill modellera. Objekt i grammatik = saker man “gör något med”. E.g. Flytta tornet två steg till vänster.
Imperativt: “kommandobaserat”, programmet består av en sekvens av kommandon som kan ha sidoeffekter.

Klasser: Beskriver en viss objekt-typ (instansvariabler) samt vad man kan göra med dessa (metoder). Kan även innehålla klassvariabler (statiska variabler) vilka är globala variabler som delas av alla instanser av en klass.

Objektorienterad programmering i Java

- **Metoder** specificerar vad man kan göra med objekt.
- **Konstruerare:** Speciella metoder för att skapa nya objekt.
  - Initierar instansvariabler.
  - Samma namn som klassen.
- **Instansmetoder**
  - Kallas på ett objekt (*implicit argument*).
  - Kommer åt objektets instansvariabler.
- **Klassmetoder** (static)
  - Kalls med klassens namn.
  - Kommer åt klassvariabler.

Konstruerare skapar nya objekt av den typ som definieras i klassen. Kan ha en eller flera konstruerare. Om man inte skriver någon egen konstruerara finns det implicit en parameterlös konstruerare som initierar instansvariabler med deras förvalda värden (null för objekttyper, förvalt värde för primitiva typer).

Instansmetoder kallas på objekt av klassens typ. Detta objekt blir som ett extra, implicit argument till metoden, utöver de “vanliga” argument som metoden tar som input. En instansmetod kan alltså komma åt det objektets instansvariabler. Den kan även komma åt klassens globala klassvariabler naturligtvis.

Klassmetoder (statiska metoder) kallas med klassens namn som Prefix (e.g. Math.sin). De kommer naturligtvis enbart åt klassvariabler.

Mini Quiz

```
public class IdentifyMyParts {
    public static int x = 7;
    public int y = 3;
}
```

1. Vilka är klassvariablerna i klassen ovan?
2. Vilka är instansvariablerna i klassen ovan?

Mini Quiz

1. Ett objekts tillstånd beskrivs av värden hos objektets ____.
2. Det man kan göra med ett objekt representeras av ___.
3. Att gömma den interna representationen av data, och bara tillåta att få tillgång till datan genom publika metoder kallas__.
4. Beteenden som definieras i en superklass kan ärvas av en ___ med hjälp av det reserverade ordet ____.
5. En samling metoder utan implementationer kallas ___.
6. Begreppet *klassvariabel* är ett annat namn för ____.
7. En *lokal variabel* beskriver ett tillfälligt tillstånd/värde och deklareras inuti en ___.

Mini Quiz

1. Ett objekts tillstånd beskrivs av värden hos objektets instansvariabler.
2. Det man kan göra med ett objekt representeras av metoder.
3. Att gömma den interna representationen av data, och bara tillåta att få tillgång till datan genom publika metoder kallas inkapsling.
4. Beteenden som definieras i en superklass kan ärvas av en subklass med hjälp av det reserverade ordet extends.
5. En samling metoder utan implementationer kallas interface.
6. Begreppet *klassvariabel* är ett annat namn för en statisk variabel.
7. En *lokal variabel* beskriver ett tillfälligt tillstånd/värde och deklareras inuti en metod.

Primitiva typer vs Objekttyper

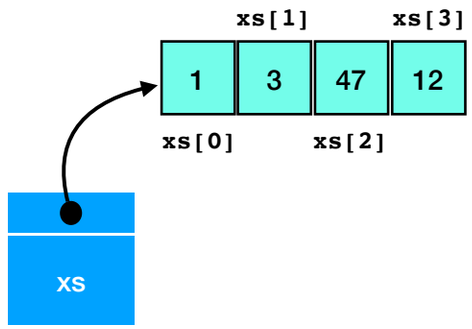
- Utöver objekt finns i Java ett antal s.k. primitiva typer.
  - `byte, short, int, long`
  - `float, double`
  - `boolean, char`
- Operatorer på primitiva typer, e.g.
  - +, -, *, %, >, <, >=, <=, ...
- Skillnad i hur Java hanterar primitiva typer och objekttyper.

Studenter ska känna till vilka de primitiva typerna är, samt skillnaden mellan ett objekt och en primitiv typ och hur dessa hanteras när de t.ex. ges som parametrar till en metod. Man ska även känna till vilka som är de förvalda värden för primitiva numeriska typer (0) och booleans (false). Man ska även känna till de vanligaste operatorerna på primitiva typer, samt deras precedens.

Se föreläsning 1.

Vissa typomvandlingar kan ske automatiskt (e.g. int -> double) om man inte riskerar att förlorar någon information. Även t.ex. int -> String går bra. Vissa omvandlingar måste man dock explicit göra en type-cast för (e.g. double -> int) eftersom man riskerar att förlora information (här decimaldelen).

Referenser vs. primitiva värden



En variabel av objekttyp (här en array) “innehåller” en referens till en position i minnet.

En variabel av primitiv typ (t.ex. int eller boolean) “innehåller” sitt värde direkt.

```
int[] xs = {1,2,3,4,5,6};
print(xs); // Skriver ut 1,2,3...
reverse(xs); // reverse har typ void
print(xs); // Skriver ut 6,5,4...
```

```
int x = 5;
System.out.println(x); // Skriver 5
negera(x);
System.out.println(x); // Skriver 5
```


Man kan skicka objekttyper som argument till metoder. Detta skiljer sig från primitiva typer (såsom int) genom att det inte är värdet (d.v.s. alla elementens värde) som skickas till metoden utan en referens till en position i minnet.
Det är en fördel för t.ex. stora arrayer på det sättet att det går snabbare. Det har också den effekten att ändringar som metoden gör i arrayen finns kvar när exekveringen av den anropande metoden fortsätter. Här gäller det att ha tungan rätt i mun. Imperativ programmering kryllar av sidoeffekter av det här slaget, men det kan leda till svårhittade fel.

Se demo från Föreläsning 2!

I exemplet till vänster, notera att vi för arrayen får nya värden *trots att vi inte skrivit xs = reverse(xs)* och reverse har returtyp void! Referensen är den samma, men det är det den pekar på som har ändrats. En metod med returtyp void kan alltså ha sidoeffekter som uppdaterar något i minnet, trots att den inte explicit returnerar något.

I det högra exemplet måste vi explicit skriva: x = negera x. negera måste ha returtypen int. Gör vi den till void kan vi varken returnera ett värde, eller uppdatera argumentet då det är av primitiv typ.

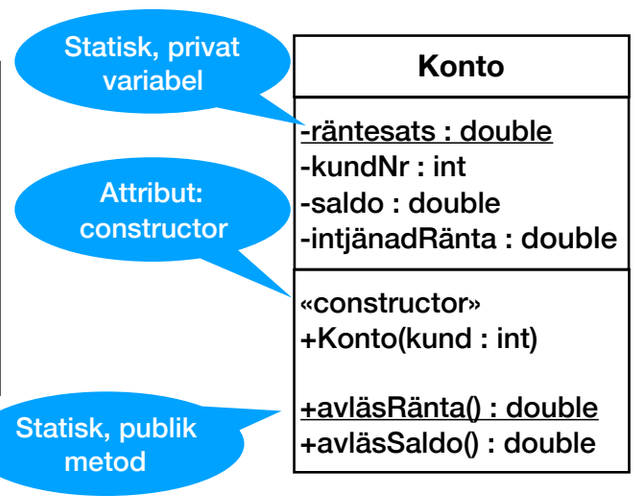
UML, tillgänglighet

- Tillgänglighet, för metoder och variabler:
  - `private, public, protected, package`.
- Inkapsling. Vad behöver synas utanför klassen?

```
public class Konto{
  private static double räntesats;
  private int kundNr;
  private double saldo;
  private double intjänadRänta;

  public Konto (int kund){...}

  public static double avläsRänta{...}
  public double avläsSaldo{...}
```



Studenter ska kunna förklara när och varför man ska använda de olika tillgänglighetsmodifierarna. Fördelarna med “inkapsling”. Se Föreläsning 2.

UML: se föreläsning 10. Studenter ska kunna läsa, förstå och skissa enkla UML diagram.

Mini Quiz

1. Vilka är de 8 primitiva typerna i Java?
2. För att invertera värdet på en boolean, vilken operator använder du?
3. Vilken operator används för att jämföra två värden, = el. == ?
4. Hur skriver du en oändlig while-loop i Java? Och vilket kommando skulle du använda för att få exekveringen att “hoppa ur” en sån loop?
5. I ett UML-diagram, vad betyder ett understruket variabel- eller metodnamn? Vad betyder symbolerna + och -?
6. Vad är värdet på `arr` resp. `x` efter följande (statiska) metoder har exekverats:

```
String [] arr = {“apa”, “boll”, “citron”};
int x = arr.length;
reverse(arr);
negera(x);
```

Mini Quiz

1. Vilka är de 8 primitiva typerna i Java?
   `byte, short, int, long, float, double, boolean, char`
2. För att invertera värdet på en boolean, vilken operator använder du? !
3. Vilken operator används för att jämföra två värden, = el. == ?
4. Hur skriver du en oändlig while-loop i Java? Och vilket kommando skulle du använda för att få exekveringen att “hoppa ur” en sån loop? while (true){...}, break.
5. I ett UML-diagram, vad betyder ett understruket variabel- eller metodnamn? Vad betyder symbolerna + och -? Understruket betyder att den är en klass variabel/metod (static), + betyder public, - private.
6. Vad är värdet på `arr` resp. `x` efter följande (statiska) metoder har exekverats: `arr -> {“citron”, “boll”, “apa”}`, `x -> 3`.

Arv

- Javas klasser bildar en hierarki.
  - `Object`: implicit superklass till alla klasser.
  - Ärver variabler och metoder från superklass (aka föräldraklass).
- *Överskuggning* av metoder: subklass "definierar om" en metod med samma namn, returtyp och argument.
  - En klass kan bara ha ***en*** (direkt) superklass.
    - `class B extends A`.

```
A a = new B();   //OK
B b = new A();    //FEL!
B b2 = (B) a;     //OK, kontrolleras vid exekvering.
C c = (C) a;      //Kontrolleras vid exekvering. Beror på vad C är!
Object o = a;     //OK.
Object o2 = b;    //OK.
```

Se föreläsning 4.

Studenter ska känna till Javas grundläggande klasshierarki med Object högst upp.
Veta att alla metoder och variabler ärvs, subklassen innehåller allt som superklassen innehåller + något mer.

Studenter ska även känna till hur man kan göra typomvandlingar mellan super- och subklasstyper. Ska även känna till vad som orsakar ett ClassCastException när programmet körs.

Nyckelordet "super" - syftar på metod/variabel från superklass. Dessa kan överskuggas i subklassen. Studenten ska känna till vad detta betyder och innebär.

Interfaces, Abstrakta klasser

- **Interface:**
  - Signaturer/abstrakta metoder: metodnamn, returtyp, argumenttyper.
  - (Även s.k. default-metoder med implementationer).
  - **Funktionsinterface**: har exakt en abstrakt metod.
    - Kan skapa anonyma klasser m.h.a. lambda-uttryck.
- **Abstrakta Klasser:**
  - Blandar vanliga metoder med abstrakta metoder (metodsignaturer).

I nyare versioner av Java är det egentligen inte så stor skillnad mellan interfaces och abstrakta klasser, eftersom interfaces kan ha s.k. default-implementationer av metoder, och inte bara lista metodsignaturer.

En viktig skillnad är dock att en klass kan implementera fler än ett interface, medan en subklass till en abstrakt klass fortfarande bara kan ärva en superklass.

Att en klass implementerar ett interface innebär att den “lovar” att tillhandahålla de (abstrakta) metoder som finns i interfacet. Man kan därför ange även interface som typer i Java.

(se Föreläsningar 4, 10).

Interfaces exempel

```
public interface MyInterface{
 void myMethod();
 int myOtherMethod(boolean b);
}
```

```
class A implements MyInterface{
   ...
 void myMethod(){...} // Här måste vi ge implementation för metoden
 int myOtherMethod(boolean b){...}
}
```

```
MyInterface a = new A();    //OK
A a2 = (A) a;               //OK, kontrolleras vid exekvering.
```

Nyckelordet “implements” för interfaces.

Studenter ska även veta hur man kan använda interface som typer.

Inre klasser, anonyma klasser

- Klasser kan definieras inuti andra klasser.
- Exempel:
  - Lyssnare i Swing GUIs.
  - Klassen för länkar i LinkedList.
- En inre klass kan definieras som anonym (utan att namnges) där den ska användas. Vanligt för lyssnare.

```
button.addActionListener(new MyActionListener()); // inre klass

//Med anonym lyssnarklass.
button.addActionListener(new ActionListener{
            public void actionPerformed(ActionEvent e) {
                revTextLabel.setText(textField.getText().toUpperCase());
        }
}
```

Se Föreläsning 6-7 bl.a.

Mini Quiz

```
interface A {
   void aMethod();
}
```

```
class B {
  void bMethod(){...}
}
```

```
class C extends B implements A {
   ....
}
```

1. Vilka metoder *ärver* klassen C?
2. Vilka (om några) metoder måste klassen B implementera?
3. Vilka (om några) metoder måste klassen C implementera?
4. Vilka av följande rader innehåller fel?

```
A a1 = new A();
A a2 = new B();
A a3 = new C();
A a4 = new A()->
System.out.println(”Hello”));
```

```
B b1 = new A();
B b2 = new B();
B b3 = new C();
```

```
C c1 = new C();
B b4 = c1;
A a5 = c1;
```

Mini Quiz

```
interface A {
   void aMethod();
}
```

```
class B {
 void bMethod(){...}
}
```

```
class C extends B implements A {
   ....
}
```

1. Vilka metoder *ärver* klassen C? bMethod + metoder som B ärvt.
2. Vilka (om några) metoder måste klassen B implementera? inga
3. Vilka (om några) metoder måste klassen C implementera? aMethod
4. Vilka av följande rader innehåller fel?

```
A a1 = new A();
A a2 = new B();
A a3 = new C();
A a4 = new A()->
System.out.println("Hello"));
```

```
B b1 = new A();
B b2 = new B();
B b3 = new C();
```

```
C c1 = new C();
B b4 = c1;
A a4 = c1;
```

Notera att vi bara kan skriva new A (där A är ett interface) om vi samtidigt definierar en anonym klass. Det går inte att skapa objekt av interfacetyp "direkt" som för variabeln a1 ovan (här har vi ju ingen definition av metoden aMethod!). Vi kan däremot skriva new A(() -> ...) som i exemplet för a4 ovan (lambda-uttrycket definierar vad aMethod ska göra eftersom A är ett funktionsinterface). Vad vi egentligen gör där är att skapa en anonym klass som implementerar interfacet A, och så skapar vi ett nytt objekt av den anonyma klassen, inte av interfacet självt.

Lambda-uttryck

- Om vi bara vill åt en metod, måste vi då skapa en hel klass?
- **Funktionsinterface:** innehåller bara en abstrakt metod.
  - Kan dock även innehålla default-metoder.
- `ActionListener` är ett exempel på ett funktionsinterface.
  - Enda metod: `public void actionPerformed(ActionEvent e)`

```
buttonCount.addActionListener(
  e -> {    // Kompilatorn kan räkna ut att e har typ ActionEvent
    if (decreaseCheckBox.isSelected()) {
      count--;
    } else {
      count++;
    }
    labelCount.setText(Integer.toString(count));
  }
);
```

- I Java 8 kan man definiera en anonym klass och skapa en instans av den med lambda-uttryck.
- Det gäller klasser som enbart implementerar ett interface och att detta är av typen funktionsinterface, d.v.s. innehåller enbart en metod.
- Liksom anonyma klasser kan lambda-uttryck referera till lokala variabler.

Fler funktions-interface

- `Predicate<T>` - metoden `boolean test(T t)`
- `Runnable` - metoden `void run()`
- `ActionListener` - metoden `void actionPerformed`

```
Predicate<Integer> isEven = x -> x%2==0;
Predicate<Integer> isOdd = x -> x%2!=0;
```

```
Runnable r = () -> System.out.println("Hello from a thread!”);
Thread t = new Thread(r);
Thread t2 = new Thread(() -> System.out.println
                              ("Hello from another thread!”));
```

isEven/isOdd är objekt av typ Predicate<Integer>. Lambda-uttrycket anger vad metoden test ska göra i den båda fallen. Notera att vi slipper definiera en hel klass här!

Samma sak gäller för variabeln r, vi kan definiera den som något av typ Runnable (vilket är ett interface) men slipper definiera en hel klass när vi bara behöver ett enda objekt av klassen.

Enumereringar

- Uppräkningsbara typer:
  - Bestämt antal värden av typen.
  - T.ex. färger i kortlek, veckodagar.

```
public enum Färg = {HJÄRTER, RUTER, KLÖVER, SPADER}

public static void print(Färg f){
 switch(f){
   case HJÄRTER: System.out.println “Hjärter”;
   case RUTER:   System.out.println “Ruter”;
   case KLÖVER:  System.out.println “Klöver”;
   case SPADER:  System.out.println “Spader”;
 }
}
```

Se föreläsning 4. Studenter ska veta när det är lämpligt att använda en enumerering samt känna till switch-konstruktionen.

Generiska klasser och metoder

- Återanvända kod - typvariabler. Spelar ingen roll vilken typ arrayen innehåller:

```
public static <T> void reverse(T[] arr) {
  for (int i = 0; i < arr.length / 2; i++) {
    T temp = arr[i];
    arr[i] = arr[arr.length - 1 – i];
    arr[arr.length - 1 - i] = temp;
  }
}
```

- Klass för par av godtyckliga typer:

```
public class Pair<T, U> {
  T fst;
  U snd;
  public Pair(T fst, U snd) {
    this.fst = fst; this.snd = snd;
  }
}
```

Se föreläsning 8.

Mycket användbart för att kunna återanvända kod. Exempelvis spelar det ingen roll vilken typ arrayen innehåller när vi vill vända på ordningen. I tidiga versioner av Java var man tvungen att använda Object som typ (då denna är supertyp till alla andra typer), men då tvingas man även göra type-casts vilket leder till extra kod och är mindre elegant.

Mini Quiz

1. Vad är ett funktions-interface (functional interface)?
2. Klassen Thread har en konstruktor: `Thread(Runnable r){...}`. Konstruera ett `Thread` objekt som skriver ut “Hallå” i terminalfönstret. Tips: använd ett lambda-uttryck.
3. Hur kan du enkelt uttrycka ett predikat i Java, som är sant för alla heltal som är delbara med 3?

Mini Quiz

1. Vad är ett funktions-interface (functional interface)? Ett interface med exakt en abstrakt metod (men det kan även innehålla default-metoder).
2. Klassen Thread har en konstruktor: `Thread(Runnable r){...}`. Konstruera ett `Thread` objekt som skriver ut “Hallå” i terminalfönstret. Tips: använd ett lambda-uttryck.

```
new Thread(() -> System.out.println(“Hallå”))
```

3. Hur kan du enkelt uttrycka ett predikat i Java, som är sant för alla heltal som är delbara med 3?

```
Predicate<Integer> p = x -> x%3==0;
```

Java Collections Framework

- Interfaces och klasser för många standarddatastrukturer.
- Använder generics.
- Interfaces: List<E>, Set<E>, Map<K, V>...
- Klasser: ArrayList<E>, TreeList<E>, HashMap<K,V>...

Föreläsning 9

Studenterna ska känna till de vanligaste datastrukturerna från Java Collections Framwork och veta när och hur man ska använda dem. Man ska även förstå hur generics används i Collections.

Undantag/Exceptions

- **Checked:** varje metod som inte hanterar checkade undantag *måste explicit tala om att undantaget kan kastas eller hantera undantaget*. E.g. `IOException` och underklasser.
- **Unchecked:** metoder behöver inte hantera dem, eller deklarera att de kan kastas. E.g. `RuntimeException` och underklasser.

```
try {
    File infil = new File("in.txt");
    Scanner scanner = new Scanner(infil);
    System.out.println("Filen är öppen");
} catch (FileNotFoundException e) {
    System.out.println(e.getMessage());
    System.exit(0);
}
```

Föreläsning 4.

Studenter ska veta vad man behöver göra för att hantera checkade exceptions med try-catch.

I/O, trådar, Stream

- Standardklasser för att skriva/läsa filer.
  - E.g. `Scanner, BufferedReader, PrintWriter` mfl.
- Trådar och parallell exekvering - klassen `Thread`.
  - Trådsäker kod.
- Interfacet `Stream`
  - Operationer som returnerar strömmar -pipelines.
  - Kan parallelliseras

```
List<Integer> list = new LinkedList<Integer>();
...
list
  .parallelStream()                 // skapa en stream från list.
  .filter(e -> e%2==0)             // gör en eller flera stream-operationer
  .forEach(e -> System.out.println e); //Avslutande operation
```

Föreläsning 11.

Studenterna ska känna till vanliga klasser för att skriva till/från filer som exempelvis Scanner, BufferedReader etc.

Studenterna ska känna till vad en tråd är och vad parallell exekvering innebär, samt vad det betyder att kod är trådsäker, och något om vad man kan och inte kan göra på ett säkert sätt i parallella program.

Interfacet Stream är nytt i Java 8 och framåt och kan vara ett lämplig verktyg om man enkelt vill parallellisera vissa operationer.

Mini Quiz

1. Varför används generics så mycket i Java Collections Framework?
2. Vilken datastruktur används för att representera datasamlingar som lagras i en bestämd ordning, men som tillåter duplicerade element?
3. Vilken datastruktur används för att representera datasamlingar som *inte* tillåter duplicerade element, och *inte* bryr sig om ordning på element?
4. Vilka undantag måste programmet explicit hantera i Java? Ge även ett konkret exempel.
5. Vad kan hända om flera trådar försöker manipulera samma objekt?

Mini Quiz

1. Varför används generics så mycket i Java Collections Framework?

   Koden ska vara återanvändningsbar för många olika typer.

   Man ska även slippa uttrycka alla typer som Object, eftersom man då slipper göra type-casts upprepade gånger.

2. Vilken datastruktur används för att representera datasamlingar som lagras i en bestämd ordning, men som tillåter duplicerade element? Interfacet List<E> som implementeras av t.ex. ArrayList<E> och LinkedList<E> i Java.

3. Vilken datastruktur används för att representera datasamlingar som *inte* tillåter duplicerade element, och *inte* bryr sig om ordning på element? Interfacet Set<E> som implementerars av t.ex. TreeSet<E> och HashSet<E>.

4. Vilka undantag måste programmet explicit hantera i Java? Ge även ett konkret exempel. Undantag i kategorin CheckedExceptions måste hanteras (eller deklareras). T.ex. IOExceptions.

5. Vad kan hända om flera trådar försöker manipulera samma objekt? Det kan uppstå s.k. thread interference och objektet kan hamna i ett icke-tillåtet tillstånd. Kan åtegärdas genom att använda syncronsierade metoder.