Introduktion till objektorientering, grundläggande Java

F22

Övningar

1. Skriv ett tidigare program du skrivit i C i Java. Välj någonting enkelt, t.ex. en tidig övning från kursens C-del.
2. I samband med övningen ovan, fundera över skillnaderna mellan C och Java. Syntaxen är ofta snarlik, men är semantiken det också?
3. Lämpligen i samband med den första övningen, jämför Java-kompilatorns felmeddelanden med C-kompilatorns. Vilka är skillnaderna? Vilken föredrar du?

Åtkomstmodifikatorer, instantiering, referenser, identitet och ekvivalens, samt klassvariabler

F23

Övningar

1. Skriv en klass `Person` med en konstruktor som tar namn och personnummer som indata, och som håller reda på hur många personer som har instantierats sedan programmet startades. Det sistnämnda görs lämpligen med en *privat klassvariabel*. När skall den räknas upp? Hur kan man garantera att den alltid räknas upp? Skriv även en instansmetod (vanlig metod) **`int`** `getCount()` som returnerar klassvariabelns värde.
2. Skriv personklassen så att utomstående inte har direkt åtkomst till ett personobjekts namn och personnummer. Namn skall gå att byta, men inte personnummer.
3. Utöka personklassen ovan så att *klassen* person har en lista över samtliga personer som skapats i systemet. Modifiera `getCount()` till att returnera denna listas längd istället för att ha en räknare. Finns det några problem med denna typ av design? Vad får det för effekt på minneshantering?

Övningar (forts)

4. Utöka personklassen med en **`boolean`** `equals(Object)`-metod som returnerar **`true`** vid jämförelse av två personobjekt med samma personnummer, annars **`false`**.
5. Utöka personklassen så att det inte går att skapa två personer med samma personnummer. Med denna garanti i systemet blir implementationen av `equals`-metoden nu trivial. Vilken är den minsta möjliga implementationen av `equals` man behöver i personklassen och varför?

Sammanfattning, arrayer, inre och nästlade klasser, undantagshantering, wrapperklasser mm

F25

Övningar

1. I övningarna till föreläsning 24 fanns en övning ”Skriv personklassen så att utomstående inte har direkt åtkomst till ett personobjekts namn och personnummer. Namn skall gå att byta, men inte personnummer.” Implementera det sistnämnda med **`final`**.
2. Skriv en metod som tar emot två argument av typerna **`int[]`** och **`double[]`** av samma längd och returnerar en array **`Object[]`** med vartannat elemenent **`Integer`** och vartannat **`Double`**.
3. Modifiera personklassen från tidigare övningar så att ett *egendefinierat* undantag kastas om det angivna personnumret inte är korrekt enligt Luhn-algoritmen[1]. Undantaget skall ärva från `IllegalArgumentException`, dvs. **`class`** `SomeName` **`extends`** `IllegalArgumentException ...`.
4. Ändra undantaget ovan så att det istället ärver `Exception`, dvs. **`class`** `SomeName` **`extends`** `Exception ...`. Vad händer vid omkompilering? Varför? Ändra programmet så att de kompilerar!
5. Skriv ett enkelt ”driverprogram” som skapar ett par personobjekt. Gör sedan undantaget ovan till en nästlad klass i personklassen, alternativt en inre klass. Hur påverkas driverprogrammet?

[1] Se `http://sv.wikipedia.org/wiki/Personnummer_i_Sverige`.

Koddokumentation med JavaDoc

F26

Övningar

1. Dokumentera personklassen och personnummerklassen du skrivit om du gjort tidigare övningar med JavaDoc, generera HTML-dokumentation, och titta på den i en webbläsare.
2. Experimentera med att slå på och av synlighet för privata medlemmar i klasser.
3. Se till att typer, både standardtyper som `String` och egendefinierade typer är korrekt länkade till i den genererade dokumentationen så att det går att klicka på dem och komma till rätt sida. Använd Google för ledning!

Parsing med Recursive Descent, Avbildningsklasser

F28

Övningar

1. Skriv ett program som läser radorienterat indata från standard in på formen `key:value` och stoppar in i en `Map<String,String>`. Du kan omdirigera standard in, precis som i C, till att läsa från en fil.
2. Ändra programmet så att alla `value`:s är heltal.
3. Ändra programmet så att insertering av en dublett medför borttagning, d.v.s., om nyckeln A pekar ut värdet 27 och ytterligare en uppdatering görs med nyckeln A och värdet 27, så skall avbildningen tas bort helt ur "mappen".
4. Javas standardbibliotek har flera olika mappar, bl.a. `HashMap` och `TreeMap`. Kan du se någon förändring i programmets exekveringstid beroende på vilken slags map som används? Testa med indatafiler med 10, 1000, 100 000 och 1 000 000 rader med 25% dubbletter för olika versioner av programmet som anväder olika typer av avbildningsklasser.
5. Lägg programmet i ett paket. Hur påverkar det kompilering och körning av programmet?

Interface

F30

Övningar

1. Utöka den personklass som använts i tidigare övningar så att den implementerar `Comparable`-interface:t och sorterar med avseende på personnummer som också bör utökas till att implementera `Comparable`.
2. Skriv ett program som skapar tio slumpvisa personer och lägger dem i en lista som sedan skall sorteras med `Collections.sort(listan)`.
3. Skriv två klasser `A` och `B` som båda ärver från `Object` samt har en variabel `value` av typen **`int`**. `A` och `B` skall implementera `Comparable` så att när en lista med blandade `A`- och `B`-objekt sorterar sig själv kommer alla `A`-objekt först, inbördes sorterande i stigande ordning med avseende på `value`, följt av alla `B`-objekt i fallande ordning med avseende på `value`. Testa programmet genom att generera slumpmässigt data som stoppas in i en lista och skriv ut listan före och efter sortering med `Collections.sort(listan)`.
4. Skriv enhetstest med `JUnit` för att testa programmet ovan.

Generiska klasser, kombination med arv, interface etc.

F31 & F33

Övningar

1. Uppdatera övningarna från föreläsning 30 så att `Comparable`-interface:t är parameteriserat av en lämplig typ.
2. Skriv en abstrakt klass `Pair` som tar två typparametrar `T` och `V` och innehåller två privata variabler `first` och `second` av typerna `T` respektiver `V`. `Pair` skall ha metoderna `getFirst()`, `setFirst()` etc. med lämpliga typer.
3. Skapa en klass `IntStringPair` som är en subklass till `Pair` som representerar par av **`int`**:ar och strängar.
4. Utöka `Pair` till att implementera `Comparable`. Hur skall typparametern till Comparable se ut?
5. Skriv ett program som skapar olika slags par (t.ex. heltal & sträng, sträng & sträng, personnummer & person, etc.) och stoppar in dem i en lista, som sorteras med `Collections.sort(listan)`. Vilka typparametrar skall listan ha? Varför?