Högskolan Kristianstad
291 88 Kristianstad
044 250 30 00
www.hkr.se

Fakulteten för naturvetenskap
Andreas Nilsson

DA556, Programmering i Java

Lektion 4 - Exekveringskontroll

Introduktion

Detta dokument syftar till att på ett, förhoppningsvis, enkelt sätt täcka stora delar av det som nämns under modulens läsanvisningar. Dokumentet riktar sig främst till de studenter som vill få en andra genomgång av materialet, på svenska, men alla studenter uppmanas att läsa igenom detta dokument och göra instuderingsuppgifterna sist i dokumentet.

I den här modulen kommer innehållet att fokusera på olika sätt att kontrollera vilken kod som körs när.

Begreppet exekveringskontroll (control flow på engelska) innefattar två typer av mekanismer för att styra hur kod körs. Den första mekanismen är upprepning som innebär att ett kodblock upprepas en eller flera gånger beroende på vad programmeraren vill. Den andra mekanismen är val som innebär att programmeraren kan välja om ett visst kodblock skall köras eller ej.

Båda dessa mekanismer använder sig ofta av variabler för att kontrollera ifall ett kodblock skall köras eller om det skall hoppas över.

Målet är att efter lektionen skall studenten ha fått en inblick i hur båda dessa mekanismer fungerar samt kunna börja använda dem i sin kod.

Varför behöver vi kunna detta?

Tidigare har det nämnts att programmering bygger på tre grundpelare, variabler, repetition och val. Den första grundpelaren, variabler, har redan behandlats och nu är det dax att titta på de två kvarvarande.

Att exekveringskontroll (repetition och val) omnämns som grundpelare inom programmering gör att dessa är extra viktiga att förstå. I stort sett alla icke triviala program som skrivs innehållet både repetition och val.

När det kommer till att skriva kod har en programmerare mycket att vinna på att så fort som möjligt lära sig hur upprepning och val fungerar. Anledningen är att om man som programmerare behärskar dessa två koncept kan man undvika att skriva svårtolkad kod, duplicera kod och göra avkall på funktionalitet.

Med hjälp av upprepning kan programmeraren skriva några instruktioner som sedan upprepas ett valfritt antal gånger, exempelvis kan programmeraren skriva tio rader kod och säga att dessa skall exekveras 1000 gånger istället för att kopiera och klistra in dessa tio rader kod 1000 gånger och få 10 000 rader kod. Val låter programmeraren konstruerar flera olika alternativ (snuttar av kod) som kan köras beroende på vad användaren gör. Exempelvis kan olika saker ske beroende på vilket menyval en användare gör.

Ytterligare ett argument för varför en programmerare bör lära sig det som modulen handlar om är att det underlättar konstruktionen av ett program, detta genom att programmeraren på ett enklare sätt kan översätta sina tankar till ett program.

Exempel: ”Om X sker så gör si, annars gör så. Oavsett vilket så gör Y tio gånger”.

Aktivitetsdiagram

I kurslitteraturen[1] går man igenom, och använder sig av, aktivitetsdiagram för att beskriva och tydliggöra hur kontrollstrukturerna fungerar. Att rita upp aktivitetsdiagram innan man börjar programmera är något som rekommenderas då det hjälper programmeraren att kontrollera att logiken som skall implementeras stämmer.

Även om det kan verka som både extra arbete, och ibland onödigt arbete, rekommenderas alla studenter att titta närmare på hur aktivitetsdiagram är tänkta att ritas och studenten uppmanas även att rita aktivitetsdiagram för de flesta övningarna för att på så sätt öva på att rita och förstå aktivitetsdiagram.

Instruktionen *if*

Som nämns i kurslitteraturen används instruktionen if för att köra en eller flera instruktioner ifall ett villkor är sant. Skulle villkoret vara falskt hoppas instruktionerna över.

I kurslitteraturen förklarar man både hur if och if-else instruktionerna fungerar på ett bra sätt. Det finns dock några punkter som är värda att notera och som studenten bör beakta.

1) Instruktioner i ett nytt kodblock, eller som följer på kontrollstrukturer (upprepningar eller val), skall använda sig av indrag. Det vill säga att raden/raderna som tillhör exempelvis en if instruktion skall ”skjutas in” med ett tabbsteg alternativt fyra mellanslag. Notera att ifall man nästlar flera kontrollstrukturer kommer det att bli flera nivåer av indrag.

2) För att undvika problem med att veta till vilken if en viss else sats hör, bör man alltid inte bara använda sig av indrag utan också av ett kodblock. Detta även om if-satsen bara har en rad kopplad till sig bör man använda ett kodblock. Ett exempel på vad som menas kan ses i exemplet här nedan:

```
if(a==10) {
    System.out.println(”Ten, we have a winner!”);
} else {
    System.out.println(”Sorry, try again.”);
}
```

   En annan vinst med att alltid använda sig av kodblock (en { följd av en eller flera instruktioner och en avslutande }) är att ifall man senare lägger till

exempelvis ytterligare en utskrift i else-satsen riskerar man inte att den hamnar på ”fel ställe” så som exempelvis utanför else:n.

3) Ytterligare en sak att notera är att inledande ”måsvingar” skall stå på samma rad som den inledande instruktionen (i exemplet ovan är if(a==10) den inledande instruktionen). Detta gäller generellt och inte bara för if-satser.

Vidare gäller att varje if-sats bara kan ha en else-sats kopplad till sig. Däremot kan en if-sats ha en eller flera else-if satser kopplade till sig. Else-if satsen låter oss välja mellan flera olika alternativ och inte som i fallet med if-else bara mellan två alternativ. Nedan givna exempel är tänkt att illustrera att beroende på en students betyg kan olika betyg tilldelas. Betygen har fler steg än två så vi måste använda oss av else-if.

```
//Koden är ej komplett, exempelvis måste Scannern input skapas…

int grade = input.nextInt();             // Läs in ett betyg
if(grade == 3) {                         // Om betyget är tre…
    System.out.println(”You got grade 3.”);
} else if(grade == 4) {                  // Om betyget är fyra…
    System.out.println(”You got grade 4.”);
} else if(grade == 5) {                  // Om betyget är fem…
    System.out.println(”You got grade 5.”);
} else {                                 // För alla andra fall
    System.out.println(”Sorry you failed.”);
}
```

Instruktionen *switch*

Instruktionen switch används i likhet med instruktionen if när man i koden vill göra val beroende på värdet på någon specific data. Exemplet man använder i kurslitteraturen är att användaren upprepade gånger matar in en poängsumma för att på så vis räkna hur många betyg man har per betygssteg.

Exemplet är väl valt då switch är tänkt att användas när man har ett begränsat och bestämt antal utfall som man vill använda sig av. Dock skulle samma exempel kunna lösas med hjälp av if och else-if istället och som ni kommer att märka kan mycket inom programmering lösas på mer än ett sätt. Det är viktigt att känna till de olika sätten och kunna välja det som är mest lämpligt för uppgiften.

Ett annat exempel på när switch är lämpligt att använda är när man skall använda sig av en meny. Det är ett fåtal, bestämda val användaren kan göra så det passar ypperligt att använda switch. Just i fallet med en meny passar det extra bra med switch då man gärna vill tala om för användaren att ifall hen väljer något ogiltigt val. Nedan finns ett exempel på hur en switch skulle kunna se ut för en meny där användaren kan göra tre val, 1, 2 och 3.

```
int choice = input.nextInt();
switch(choice) {
    case 1 : {
        System.out.println(”Val 1 – Insättning”);
        break;
    }
    case 2 : {
        System.out.println(”Val 2 – Uttag”);
        break;
    }
    case 3 : {
        System.out.println(”Val 3 – Avbryt”);
        break;
    }
    deafult : {
        System.out.println(”Ogiltigt val”);
        break;
    }
}
```

På slutet i exemplet ovan kan man se default konstruktionen som används för att markera att alla värden på choice som inte definierats skall hanteras på samma sätt, de skall hanteras under default. Det är även värt att notera att i varje case används break för att markera att case:t är slut. Utan break skulle Java ”trilla över” i nästa case. Exempelvis, ponera att vi tar bort break under case 1 ovan, det skulle resultera i att ifall choice håller värdet 1 kommer först case 1 att köras och eftersom ett break saknas kommer Java att ”trilla över” och även köra case 2 tills den därefter stöter på ett break och hoppar ur switchen.

En begränsning för switch är att man bara kan använda sig av heltal eller text (String) att ”switcha” på. Skulle man behöva göra val utifrån exempelvis flyttal eller en boolean får man använda sig av instruktionen if istället.

Mer switch

Även om switch instruktionen är väldigt användbar i vissa lägen dras den som vi redan varit inne på med vissa problem, exempelvis att man måste skriva break annars trillar man över och kör nästa case. Ett annat problem med switch instruktionen är att Java inte kan klura ut om man täckt alla möjliga fall så man måste ha med default för att vara på ”den säkra sidan”.

Det finns i nyare versioner av Java en möjlighet att använda en annan typ a switch instruktion som kallas *switch expression* (jämför med den vanliga switches som kallas *switch instruction*). När man skapade denna nya konstruktion var målet att den skulle användas vid något som kallas Pattern Matching. Det är dock möjligt att använda detta typ av switch i andra sammanhang.

I denna kursen kommer vi inte att använda oss av *switch expressions* men den nyfikne läsaren kan hitta mer information om den här nya typen av switch i JEP 361: Switch Expressions[2].

Jämföra strängar

Skall man använda strängar (String) i antingen en if-sats är det bra att känna till att man inte skall jämföra strängar med operatorn ==. Istället skall man använda sig av en metod som heter equals och som finns i String klassen. Vi har inte ännu gått igenom vare sig metoder eller klasser så för tillfället nöjer vi oss med att lära oss hur vi skall göra för att jämföra strängar. Förklaringen till både metoder och klasser kommer senare i kursen.

Felaktig jämförelse av strängar:

```
String name1 = ”Kalle”;
String name2 = ”Pelle”;
if(name1 == name2) { }
```

Korrekt jämförelse av strängar:

```
String name1 = ”Kalle”;
String name2 = ”Pelle”;
if(name1.equals(name2)) { }
```

Jämförelse av strängar i Java är lite klurigt då det ibland fungerar att använda sig av == men inte alltid. Anledningen att det ibland fungerar är att Java optimerar den kod programmeraren skriver så tillvida att ifall Java upptäcker att det på två ställen finns en text som är ”Kalle” kommer den endast spara ”Kalle” en gång i minnet och låta båda ställena i Java koden referera till denna enda lagring i minnet. När det kommer till användning av == för att jämföra strängar är det inte innehållet i variabeln som jämförs utan själva referensen till minnet där innehållet finns lagrat som jämförs. Med anledning av detta skall man alltid använda equals metoden så som visas ovan.

Upprepning

Inom programmering är det ofta som en viss instruktion eller ett kodblock bestående av flera instruktioner behöver upprepas ett antal gånger. Ibland vet man på förhand hur många gånger och ibland är detta inte känt förrän programmet körs.

Om inte Java skulle haft funktioner för upprepningar i stil med vad som nämns ovan inbyggt skulle Java program bli mycket komplicerade eller utrymmeskrävande (tänk klipp och klistra för koden som skulle köras flera gånger). Lyckligtvis finns det flera olika möjligheter för upprepning i Java varav *for* och *while* är de i särklass vanligaste.

När det kommer till användning av *for* och *while* kan man alltid byta ut den ena mot den andra även om de båda har sina starka sidor och passar bäst i vissa lägen.

Instruktionen *for*

Om det på förhand (innan upprepningen påbörjas) är känt hur många upprepningar som skall göras lämpar det sig att använda en *for*-loop.

Kurslitteraturen går noggrant igenom hur en for-loop skall konstrueras men kort kan man säga att den består av fyra delar:

- Initiering
- Villkor
- Ökning/minskning
- Kodkropp (body)

I Java hänger delarna ihop enligt följande skiss:

```
for (initiering; villkor; ökning/minskning) {
    kodkropp (en eller flera instruktioner)
}
```

Ett exempel skulle kunna vara att ett meddelande skall skrivas ut tio gånger på skärmen.

```
for (int i=0; i<10; i++) {
    System.out.println(”Hej!”);
}
```

Variabler som deklareras i initieringen av for-loopen gäller bara i loppen och initieras en gång, innan första varvet i loopen. Det vill säga, de gäller bara i det kodblock som är associerat till loopen. Oftast är det bara en räknevariabel som används i en for-loop men det är inget som hindrar att man deklarerar flera variabler, man skulle exempelvis kunna deklarera både i och j enligt följande exempel.

```
for (int i=0, j=10; i<10; i++, j--) {
    System.out.println(”Hej!”);
}
```

Notera att datatypen int bara anges en gång därefter separeras variablerna i och j med ett komma.

Själva villkoret i loopen utvärderas innan varje varv i loopen. Det vill säga innan första varvet, innan andra varvet osv. Detta ger att om inte villkoret uppfylls när Java når loopen för första gången är det inte säkert att koden i loopen körs alls. Exemplet nedan kommer aldrig att skriva ut något på skärmen då villkoret aldrig kan bli sant.

```
for (int i=0; i > 100; i++) {
    System.out.println(”Hej!”);
}
```

Ökningen eller minskningen som ingår som en del i loopen sker i slutet av varje varv och innan villkoret utvärderas igen. Detta innebär att första varvet i loopen så håller kontrollvariabeln det värde den tilldelades under initieringen.

Instruktionen *while*

Till skillnad från for-loopen används while-loopen företrädelsevis när man inte på förhand vet hur många varv man kommer att behöva snurra i loopen. Man använder sig av ett villkor för att bestämma när det är dax att sluta snurra runt i loopen. Skillnaden mot en for-loop är att det inte sker någon automatisk upp eller nedräkning.

Det är viktigt att programmeraren ser till att villkoret någon gång blir falskt och loopen på så vis avslutas annars kommer programmet fastna i en evig loop och aldrig komma vidare i koden. Att se till att villkoret blir falskt skall ske inuti loopen.

Även om while-loopen företrädelsevis används när man inte vet antalet varv som behövs kan man också använda den när antalet varv är känt, se exemplet nedan. Notera att kontrollvariabeln måste räknas upp manuellt inuti loopen.

```
int i = 0;
while (i < 10) {
    System.out.println(i);
    i++;
}
```

Oftast används dock while-loopar med ett stoppvärde antingen i form av en boolean som blir falsk när man skall hoppa ur loopen eller med ett heltalsvärde som sätts till exempelvis -1 när man skall hoppa ur loopen. För ett exempel med en boolean som stoppvärde se nedan.

```
boolean cont = true;
while (cont) {
    // Här sker något upprepade gånger och
    // när det är dax att hoppa ur loopen sätts cont = false. För
    // att ha något exempel väljer vi att sätta cont = false om
    // de påhittade int:arna a och b tillsammans blir större än 10.
    if (a + b > 10) {
        cont = false;
    }
}
```

Alternativet *do-while*

Egentligen är do-while loopen väldigt lik while-loopen med den enda skillnaden att i do-while loopen utvärderas villkoret efter loopen. Detta får till följd att en do-while loop alltid kommer att köra ett varv i loopen oavsett om villkoret evakueras till sant eller falskt.

Syntaxen för en do-while loop skiljer sig något från en while-loop och de främsta skillnaderna är att villkoret kommer först på slutet och att man i en do-while loop avslutar med ett semikolon, se nedan.

```
do {
    // Här återfinns en eller flera instruktioner
    // som kommer att köras minst en gång…
} while (villkor);
```

I praktiken används do-while loopen sällan men som programmerare skall man känna till den och man skall behärska den. Ett exempel på när man skulle kunna tänka sig att använda en do-while loop är ifall man skall läsa in ett värde från en användare och om det är ett ogiltigt värde läser man in ett nytt. Här kommer inläsningen alltid att ske minst en gång och en do-while loop skulle kunna vara motiverad.

Exempel

Låt oss studera några korta exempel för att se hur exekveringskontroll kan användas rent praktiskt. Exemplen blir mycket renare och kanske lättare att förstå utan alla kommentarer men de finns med här för att tydliggöra vad som sker.

Exempel 1 - Instruktionen if

```
import java.util.Scanner;
public class Example {
    public static void main(String []args) {
        // Först skapar vi en Scanner för att kunna läsa in från
        // tangentbordet.
        Scanner input = new Scanner(System.in);
        // name ska vi använda för att lagra ett inläst namn
        String name;

        System.out.print(”What is your name? ”);
        // läs in ett namn
        name = input.nextLine();

       if(name.equals(”Kalle”)) {
            // Om det inlästa namnet är Kalle körs denna raden.
            System.out.println(”Hej Kalle”);
        } else if(name.equals(”Anna”)) {
            // Är det inlästa namnet Anna körs denna raden.
            System.out.println(”Hejsan Anna”);
        } else {
            // I alla andra fall körs denna raden
            System.out.println(”Välkommen ” + name);
        }
   }
}
```

Exempel 2 - Instruktionen switch

```
import java.util.Scanner;
public class Example {
    public static void main(String []args) {
        // Första delen av detta exempel är samma som i förra exemplet
        Scanner input = new Scanner(System.in);
        System.out.print("What is your name? ");
        String name = input.nextLine();

        // Nu är if-satsen utbytt till motsvarande switch-sats.
        // Viktigt att notera är användningen av break. Prova själv
        // att ta bort break på ett eller flera ställen för att se vad
        // som händer. Prova även att lägga till fler instruktioner i
        // varje ”case”.
        switch (name) {
            case "Kalle":
                System.out.println("Hej Kalle");
                break;
            case "Anna":
                System.out.println("Hejsan Anna");
                break;
            default:
                System.out.println("Välkommen " + name);
                break;
        }
   }
}
```

Exempel 3 - For-loopen

```
import java.util.Scanner;
public class Example {
    public static void main(String []args) {
        // Först deklarerar vi de variabler vi kommer att behöva,
        // en scanner för inläsning från tangentbordet och två
        // variabler som ska användas för att lagra en text och ett
        // nummer som vi läser in från tangentbordet
        Scanner input = new Scanner(System.in);
        String text;
        int rep;

        System.out.print(”Write a short sentence: ”);
        // Här läser vi en in text
        text = input.nextLine();
        System.out.print(”Write the number of repetitions: ”);
        // Läser in hur många gånger vi ska loopa
        rep = input.nextInt();

        // I loopens första varv kommer i ha värdet 0, därefter ökas
        // det med ett för varje varv i loppen. Varje varv skrivs
        // värdet av i och den inmatade texten ut.
        for(int i=0; i< rep; i++) {
            System.out.println(i + ” : ” + text);
        }
    }
}
```

Exempel 4 - While-loopen

```
import java.util.Scanner;
public class Example {
    public static void main(String []args) {
        // Samma funktionalitet som i exemplet med en for-loop men nu
        // löst med hjälp av en while-loop istället. Notera att while-
        // loopar oftast används när man inte vet antalet iterationer
        // men kan även användas för att ersätta for-loopar. Ofta
        // känner sig programmerare mer bekväma med den ena typen av
        // loopar, detta är helt naturligt.
        Scanner input = new Scanner(System.in);
        String text;
        int rep;
        // loopens räknevariabel är utflyttad ur loopen
        int i=0;

        System.out.print(”Write a short sentence: ”);
        // Här läser vi en in text
        text = input.nextLine();
        System.out.print(”Write the number of repetitions: ”);
        // Läser in hur många gånger vi ska loopa
        rep = input.nextInt();

       while(i < rep) {
            System.out.println(i + ” : ” + text);
            i++;  // Kan även skrivas som i += 1; eller i = i+1;.
                  // Oavsett hur man väljer att skriva ökningen av i
                  // är det mycket viktigt att den finns med annars
                  // skapas en evig loop.
        }
    }
}
```

Exempel 5 - do-while loopen

```
import java.util.Scanner;
public class Example {
    public static void main(String []args) {
        // Precis som i de förra exemplen använder vi en Scanner för
        // att läsa in från tangentbordet
        Scanner input = new Scanner(System.in);
        // Detta är den variabel som skall hålla koll på om vi ska
        // fortsätta snurra i loopen eller ej
        boolean cont = true;
        // För att spara inläst text
        String text;

        // Kodblocket associerat med do-while loopen kommer att köras
        // minst en gång.
        do {
            System.out.println("To exit the loop enter EXIT.");
            System.out.print("Enter text: ");
            text = input.nextLine();

            // Om vi skriver in EXIT med stora bokstäver kommer vår
            // "kontrollvariabel" sättas till false och loopen kommer
            // att avslutas vid nästa kontroll av cont variabeln.
            if(text.equals("EXIT")) {
                cont = false;
            }
        // kontroll av cont variabeln sker först efter loopen. Notera
        // även att semikolonet krävs.
        } while(cont);
   }
}
```

Referenser

[1] Paul Deitel, Harvey Deitel (2017), Java How to program, Early objects (Global Edition). Pearson.

[2] JEP 361: Switch Expressions. (2022) Oracle Corporation. [Online] Tillgänglig från: https://openjdk.java.net/jeps/361. [Tillgänglig: 2022-02-18]