F5 – Selektion och iteration

ID1004 – Objektorienterad programmering

Fredrik Kilander fki@kth.se

Boolska uttryck

- Boolska uttryck använder sig av jämförelseoperatorer
- < > <= >= == !=
- Resultatets datatyp är boolean (true, false)
- 3 < 99 ger värdet true
- 99 > 99 ger värdet false
- 1 != 0 ger värdet true

Boolska uttryck - if

- Boolska värden gör det möjligt att välja vilken kod som ska exekveras
- **`if (`**boolean**`)`** t-stmt**`;`**
- t-stmt exekveras endast om det boolska uttrycket är sant (true)

```
if (x < 0) {
  x = -x;
}
```

Boolska uttryck – if-else

- Boolska värden gör det möjligt att välja vilken kod som ska exekveras
- **`if (`**boolean**`)`** t-stmt**`; else`** f-stmt**`;`**
- t-stmt exekveras om det boolska uttrycket är sant (true), annars exekveras f-stmt

```
if (n == 1) {
  s = ”thing”;
} else {
  s = ”things”;
}
```

Likhetsoperatorn

- `==`
- Prövar om två uttryck har samma värde

```
if (count == 6) { ...

if (b == false) { ...
```

Likhetsoperatorn

- ==
- Om två referenser jämförs så måste det vara *samma* referens för att det ska bli true

```
Die die1 = new Die(); // En instans
Die die2 = new Die(); // En annan instans

boolean b = (die1 == die2); // false

die2 = die1; // die2 blir alias till die1

b = (die1 == die2); // true
```

Likhetsoperatorn

- ==
- Tänk på att den liknar tilldelning!

```
Die die1 = new Die();
Die die2 = new Die();

boolean b = (die1 == die2);

if (b = true){
  System.out.println(b);
}
```

Alltid sant...

Logiska operatorer

- Logiska operatorer tar boolska värden som argument
- `!` negation
- `!true` ger värdet `false`
- && konjunktion, logiskt OCH: båda sidor måste vara true
- || disjunktion, logiskt ELLER: en eller båda sidorna måste vara true

Logiska operatorer

- Används for kombinerade villkor
- `if (done || (maxLimit <= count)) { ...` // stop
- `if (!done && (count < maxLimit)) { ...` // go on
- Vilken datatyp har variabeln **done**?

Satsblock

- Med hjälp av satsblock {...} kan flera programsatser behandlas som en enhet

```
if (count < maxLimit) {
   count += 1;
   sum /= 2;
   ...
}
```

Satsblock

- Satsblock är ibland obligatoriska

```
public class Foo {
   int[] nums = new int[]{2,3,5,7,11};
   public void getNum() {
      ...
   }
}
```

Satsblock

- Satsblock *rekommenderas* i if-satser, for-, while- och do-loopar.

```
if (count < maxLimit) {
   count += 1;
   sum /= 2;
   ...
}
else {
   return sum;
}
```

Villkorsoperatorn ?

- Operatorn ? tar tre argument
- villkor ? t-värde : f-värde ;

```
public int min(int a, int b) {
  return (a < b) ? a : b;
}
```

Villkorsoperatorn ?

- Operatorn ? tar tre argument
- villkor ? t-värde : f-värde ;

```
System.out.print(”You have ”);
System.out.print(score);
System.out.print(” point”);
System.out.println ((score == 1) ? ”.” : ”s.”);
```

```
You have 0 points.
You have 1 point.
You have 2 points.
```

Nästlade if-satser

- Använd satsblock och indentering för att undvika förvirring

```
if (num1 < num2)
if (num1 < num3)
min = num1;
else
min = num3;
else
if (num2 < num3)
min = num2;
else
min = num3;
```

rekommenderas ej

```
if (num1 < num2)
  if (num1 < num3)
    min = num1;
  else
    min = num3;
else
  if (num2 < num3)
    min = num2;
  else
    min = num3;
```

indenterad

```
if (num1 < num2){
  if (num1 < num3) {
    min = num1;
  } else {
    min = num3;
  }
} else {
  if (num2 < num3) {
    min = num2;
  } else {
    min = num3;
  }
}
```

indenterad och med satsblock

Jämföra data – floating point

- == operatorn kräver exakt likhet (alla bitar)
- Flyttal avrundas ofta vid beräkningar
- Definiera likhet som *tillräckligt nära*

```
float f1, f2; // Två flyttal
...
if (f1 == f2) { ... // exakt lika

if (Math.abs(f1 – f2) < TOLERANCE) {
  // tillräckligt lika...
}
```

Jämföra data - char

- Tecken kodas som 16-bitars positiva heltal
- Dessa tal kan jämföras (t ex vid sortering)
- De första 127 koderna är samma som ASCII
- De första 256 koderna är samma som ISO-8859-1

```
if (ch1 > ch2) {
  // do stuff
}
```

I praktiken är det sällan man tittar på teckens numeriska värde – Java har goda verktyg för att hantera text.

Jämföra data - String

- Strängar är objekt; en referensdatatyp
- == är bara sann om det är samma objekt
- Strängar jämförs med metoder i java.lang.String

```
if (name1.equals(name2)) {...
```
Samma tecken

```
if (name1.equalsIgnoreCase(name2)) {...
```
Samma ord

```
if (name1.compareTo(name2) == 0) {...
if (name1.compareTo(name2) < 0) {...
if (name1.compareTo(name2) > 0) {...
```
Likvärdigt equals
name1 före name2
name1 efter name2

```
if (firstName.equals(”Allan”)) { ...
if (”Allan”.equals(firstName)) { ...
```
Strängkonstanter är också objekt

Mer om String

- Textsträngar är instanser av klassen String
- Varje instans är slutgiltig (immutable)
- En ändrad String är en ny String:

```
String s = ”foo”;
s = s + ”bar”;
```

```
String s = new String(”foo”);
String t = new String(”bar”);
s = new String(”foobar”);
```

```
System.out.println(s);
```

```
foobar
>
```

Iteration - while

- while (villkor) t-stmt;
- while (villkor) { ... }
- Villkoret prövas först, därefter utförs programsatsen eller blocket så länge villkoret är sant

```
int count = 1;
while (count < 5) {
  System.out.println(count);
  count++;
}
```

Oändliga loopar

- Det är lätt att skriva loopar som pågår längre än man tänkt sig

```
int count = 1;
while (count <= 25) {
  System.out.println(count);
  count--;
}
```

Denna loop pågår tills dess att count når Integer.MIN_VALUE, varvid den slår om och får värdet Integer.MAX_VALUE. Exekvering fortsätter efter loopen.

Oändliga loopar

- Det här är en oändlig loop

```
int count = 1;
while (count == count) {
  System.out.println(count);
  count--;
}
```

Denna loop pågår tills dess att programmet avbryts.

Oändliga loopar

- Det här är också en oändlig loop

```
int count = 1;
boolean done = false;
while (!done) {
  System.out.println(count);
  count--;
}
```

Denna loop pågår tills dess att programmet avbryts.

Oändliga loopar

- Ibland vill man ha en ”oändlig” loop
- Tänk web-server, mail-server, operativssystem

```
protected void doWork() {

  while (true) {
    ... // Serve next request
  }

}
```

Denna loop pågår tills dess att programmet avbryts.

Nästlade loopar

- En loop inuti en annan loop

```
int outerCount = 10;
while (0 < outerCount) {
  int innerCount = 5;
  while (0 < innerCount) {
    System.out.println (outerCount*10+innerCount);
    innerCount--;
  }
  outerCount--;
}
```

Hur många gånger anropas System.out.println()?

```
105
104
103
102
101
...
```

Mer kontroll över loopar

- break – hoppa ut ur det närmaste satsblocket
- continue – hoppa direkt till början på loopen

```
while (...) {
  ... // gör lite arbete
  if (...) break; // redan färdig med loopen?
  ... // gör lite mer arbete
}
```

```
while (...) {
  ... // gör lite arbete
  if (...) continue; // redan färdig med detta varv?
  ... // gör lite mer arbete
}
```

Iteratorer – java.util.Iterator

- En Iterator är ett hjälpobjekt för att gå igenom en datasamling
- Vissa metoder i vissa klasser erbjuder en Iterator
- Varje Iterator erbjuder tre metoder:

- **`boolean hasNext()`** – finns ett nästa element?
- **`E next()`** – hämta nästa element
- **`void remove()`** – ta bort senast hämtade från samlingen

Iteratorer – java.util.Iterator

- **`boolean hasNext()`** – finns ett nästa element?
- **`E next()`** – hämta nästa element
- **`void remove()`** – ta bort senast hämtade från samlingen

```
Iterator<Card> ic = deck.iterator();
while (ic.hasNext()) {
  Card cd = ic.next();
  ... // Gör nåt med kortet
}
```

Iteratorer – java.util.Iterator

- **`boolean hasNext()`** – finns ett nästa element?
- **`E next()`** – hämta nästa element
- **`void remove()`** – ta bort senast hämtade från samlingen

```
for (Iterator<Card> ic = deck.iterator();
     ic.hasNext();) {
  Card cd = ic.next();
  ... // Gör nåt med kortet
}
```

Interface java.lang.Iterable

- Fr o m Java 5
- Objekt som implementerar java.lang.Iterable kan användas direkt i for-loopen
- java.lang.Iterable != java.util.Iterator

```
for (char c : String s) {if (c == 'a') ...;}
```

```
int numbers[100];
int sum = 0;
for (int n : numbers) {
  sum += n;
}
```

Loopar - sammanfattning

- do – minst en gång genom loopen
- while – villkoret testas först
- for – antalet varv är känt
- for each – alla element i en samling

- break – hoppa ut i förtid
- continue – nästa varv genast

wha...

SLUT PÅ BILDER

Selektion – switch

- Satsen **`switch`** kan ersätta nästlade if-satser

```
if (idChar == 'A') {
  aCount = aCount + 1;
}
else
  if (idChar == 'B') {
    bCount = bCount + 1;
  }
  else
    if (idChar == 'C') {
      cCount = cCount + 1;
    }
    else {
      System.out.println ("Error in ...");
    }
```

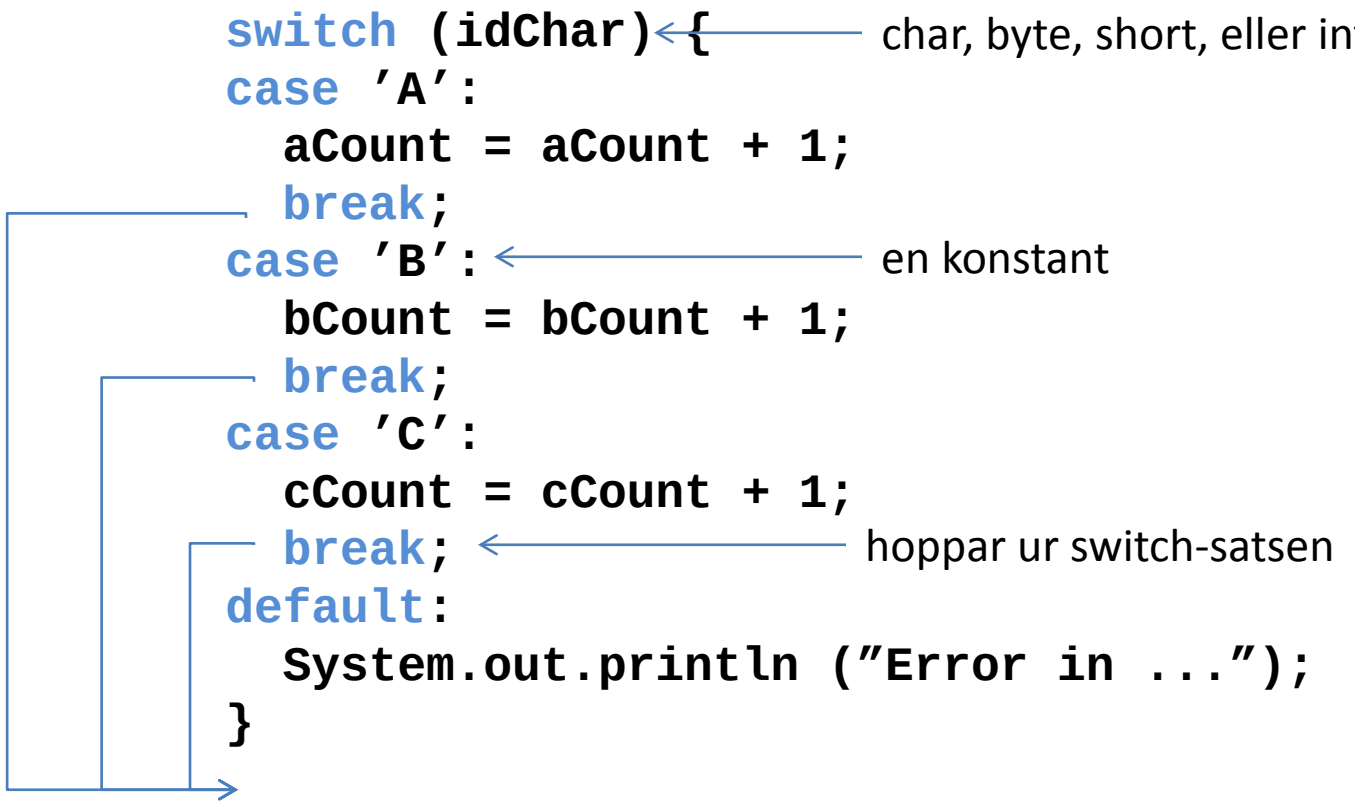
Selektion – switch

- Satsen **`switch`** kan ersätta nästlade if-satser

```
switch (idChar) {
case 'A':
  aCount = aCount + 1;
  break;
case 'B':
  bCount = bCount + 1;
  break;
case 'C':
  cCount = cCount + 1;
  break;
default:
  System.out.println ("Error in ...");
}
```

Selektion – switch

- Satsen **`switch`** kan ersätta nästlade if-satser



*) I Java 7 tillåts även String

Iteration – do - while

- do {...} while (villkor);
- Programsatsen utförs alltid första gången
- Därefter igen så länge villkoret är sant

```
int count = 0;
do {
  count++;
  System.out.println(count);
} while (count < 5);
```

Iteration - for

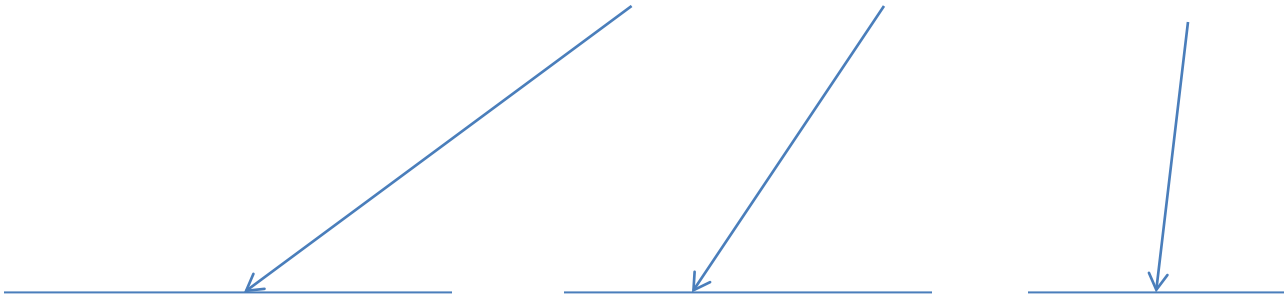
- for – loopen sammanfattar initiering, villkor och uppdatering
- Detta gör koden enklare

```
for ( initering ; villkor ; uppdatering ) {
  // kod i loopen
}
```

```
initering
while (villkor) {
   // kod i loopen
 uppdatering
}
```

Iteration - for

- for – loopen sammanfattar initiering, villkor och uppdatering



```
for (int count = 0; count < 25; count++) {
  System.out.println(count);
}
```

Variabeln count finns bara i for-loopen. Mindre risk för förväxling.

Iteration - for

- for – loopen sammanfattar initiering, villkor och uppdatering
- Tomma delar kan utelämnas

```
for (Iterator<Card> ic = deck.iterator();
     ic.hasNext(); )
{
  Card cd = ic.next();
}
```