Bilder Vecka 4

TDA548/Joachim von Hacht

Grunderna

(som sagt, som sagt, repetera serien Grunderna)

Styrande Satser

switch-satsen

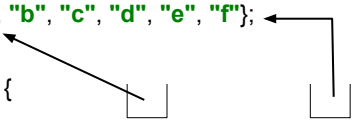
```
// Switch statement
switch (i) {              // i is value to compare for equality
   case 0:                // If match 0 ...
       out.println("match 0"); // ... run this
       break;   // IMPORTANT, else will run "case 1" also
   case 1:
       out.println("match 1");
       break;
   case 2:
       out.println("match 2");
       break;
   default:
      out.println("no match");  // If no match
}
```


Switch-satsen är en förenklad selektion där man bara väljer utifrån likhet.

- Villkoret är underförstått likhet.
- Om uttrycket i parentesen efter switch är lika med något av de uppräknade värden i case-”grenarna” så körs satserna i den grenen
- Matchar inget körs default grenen.
- Värdena i grenarna som jämför måste vara konstanta (literaler eller konstant variabler)
- Viktigt med break sist i varje gren, annars kör nästa gren också.
- Switch satsen kan användas för typerna: char, int, String, enum eller omslagstyperna: Character och Integer, m.fl.
    - Alltså inte double …

Kort for-loop

```
int[] is = {1, 2, 3, 4, 5, 6, 7, 8, 9};
for (int i : is) {      // No index just each element, left to right.
   out.print(i);        // i++ sense less
}

String strs[] = {"a", "b", "c", "d", "e", "f"};

for (String s : strs) {
   out.print(s);
   s = "X";  // Senseless
}
```



Tilldelning av s ändrar inte original-array:en

Om man bara vill traversera en array (eller annan samling, mer senare) och inte behöver ett index, finns en enklare for-loop

- Loopen tar ett element i taget från början till slut (index [0-(length-1)]
  - Använd om ni vill … kan förekomma i kodexempel
- Att tilldela loop-variabeln (tex. i och s i koden) är meningslöst.
  - Ändrar inte original array:en
  - Behöver man ändra ett index måste man använda den vanliga for- eller while-loopen.
    - Se även nästa bild
- Kan dessutom ge ConcurrentModificationException i samband med samlingar, see vidare Samlingar

Kort for-loop med Objekt

```
class Point {
      int x, y;
}

Point[] pts = {new Point(3,4), new Point(-1,2), new Point(6,0),};

// Will change object
for (Point p : pts) {
      p.x++;
}
```

Objektet p refererar kommer att ändras

Kort for-loop fungerar bra om man vill modifiera objekt eftersom det inte är själva referensen man ändrar

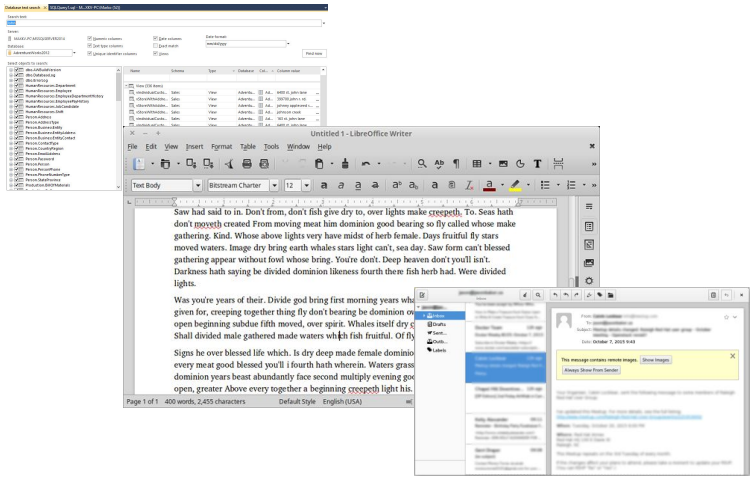
- Det är ju objektets variabler som ändras.

Metoder

(inget nytt)

Strängar

Textbehandling

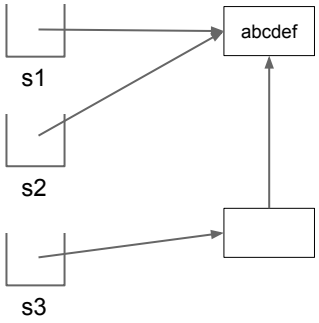


Mycket inom databearbetning handlar om textbehandling

- För att hantera texter i ett program använder vi strängar
- Strängar är följder av tecken

String

```
String s1 = "abcdef"; // Object created
String s2 = "abcdef"; // No new object
String s3 = new String("abcdef");  // Avoid
```



10

String är en standarklass för strängar i Java

- Strängliteraler skrivs med omslutande “-tecken
- Alla strängar är instanser av referenstypen String
  - Strängar är objekt
- Strängar är icke-muterbara
  - Operationer som innebär förändring av strängen medför att nya strängar skapas
  - Gäller i synnerhet +-operatorn
- Undvik att använda String-konstruktorer, skapar onödiga objekt

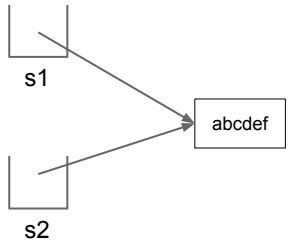
En sträng är inte samma som en char[]

Alla strängliteraler delar samma tecken

- Själva teckenföljder sparas i en “pool”.
- Om strängen redan finns i poolen sparas den inte igen

Strängar och Tilldelning

```
String s1 = "abcdef";

String s2;
s2 = s1;
```



Fungera som för alla referenstyper.

- Referenserna pekar på samma objekt

Strängar och Likhet

```
String s1 = "abcdef";
String s2 = new String(s1);

out.println(s1 == s2);  // False (by ref. semantics)

// Must use for value semantics
out.println(s1.equals(s2));  // True

// Also value semantics
out.println("olle".compareTo("fia") < 0);  // True
```

12

För strängar gäller referenssemantik vid jämförelse med ==

Vanligen vill vi ha värdesematik (d.v.s. jämföra tecknen)

- Vi får detta genom att använda metoden equals() …
- … eller metoden compareTo()
  - Ger 0 vid likhet
  - < 0, om argumentet är mindre i lexikografisk ordning (tecken för tecken vänster till höger)
  - > 0, om argumentet är större i lexikografisk ordning
- Alla strängobjekt har egna equals, toString, etc.
  - Det är egna versioner av de ärvda metoderna från Object, se Klasser

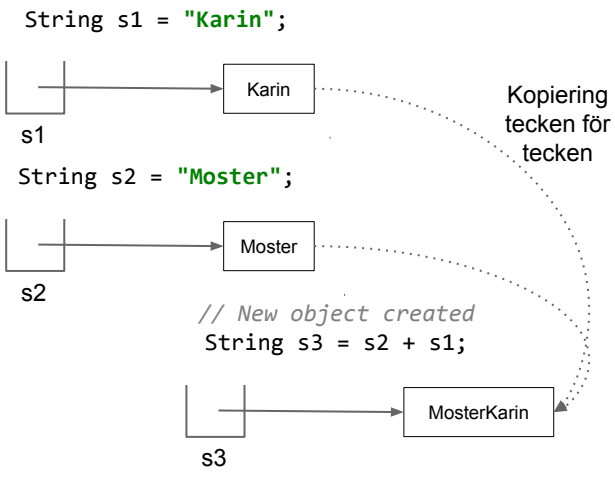
Strängar och null

```
String s = ...;   // Possible null

// Put literal first (else possible exception)!
if( "olle".equals(s)){ // If s null we get false


}
```


Ett knep för strängar

- Om man skall jämföra med en literal så skrivs den först…
- .. undviker (klarar) om referensvariabeln är null (null ger false)

Strängar och +-operatorn



14

Konkatenering med +-operatorn innebär att ett nytt strängobjekt skapas och en referens till detta returneras.

- Eftersom strängar inte kan ändras (tecknen kan inte ändras)
- Tecknen från operanderna kopieras till det nya objektet.
- +-operatorn kan vara ineffektiv t.ex. i en loop med många varv (kopierar samma sak och mer och mer för varje varv)
-

Metoder i String

```
// Inspect
str.isEmpty();
str.length();
"abcdef".charAt(3);        // ‘d’
"abcdef".indexOf('a');     // 0

// Search
str.contains("cd"));
str.startsWith("abc");
str.endsWith("def");

// Manipulate Note! New object(s) created
str.replace("failure", "icecream");
"abcdef".substring(0, 4);  // “abcd”
"abcdef".substring(4);      // “ef”
"abc:def".split(":");      // [ “abc”, “def” ]
```

15

Olika kategorier, se kodexempel

- Inspektera
- Söka (i texten)
- Manipulera (förändra texten)
- Finns många fler

Ni behöver inte kunna metoderna utantill

- Skulle det behövs (dvs. tentan) så får ni en lista på användbara
- Här är dokumentationen av String

För att söka och manipulera används ofta “strängmönster” för att matcha något i en sträng (regular expressions, reg exp)

- Vi går inte in på detaljer, vi använder eventuellt några enkla mönster.

Fallgropar

```
String s = "abcdef";

// Prints ab
out.println(s.substring(0, 2));

// Prints abcdef! Object s unchanged!
out.println(s);

// Save reference to new object
s = s.substring(0, 2);

// Prints ab, s changed!
out.println(s);
```


Ett problem man får se upp med är att det skapas nya objekt då man manipulerar Strängar

- För att ändra ett värde måste man ha en tilldelning (spara det nya värdet)

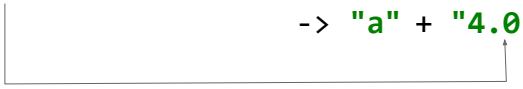
String och char[]

```
String str = "abcdef";

// Convert to array
char[] arr = str.toCharArray();
// Back to String
str = new String(arr);

// Work with a single char at the time
for( char ch : str.toCharArray()){
          // Do something
}
```


Man kan konvertera mellan String och char-array (för att jobba med enskilda tecken).

Implicit Typomvandling med + - operatorn och String

```
"a" + 4.0 -> "a" + new Double(4.0).toString()
                          -> "a" + "4.0" -> "a4.0"
```



Implicit typomvandling

```
out.println(dog) -> out.println(dog.toString());
```

Sträng inte supertyp till någon typ d.v.s. inget kan implicit omvandlas till String …

- …förutom vid två tillfällen
  - Då + operatorn har minst en operand av typen String.
  - out.println(), … då egentligen metoden toString() anropas.

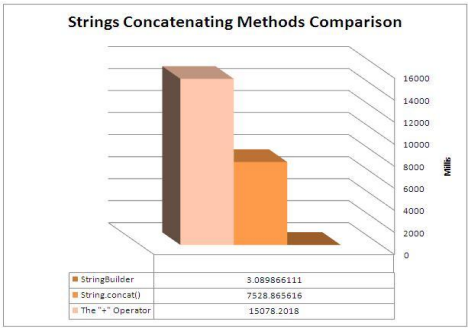
Explicit Typomvandling med String

```
// From primitive to String
String s = String.valueOf(45); // String class methods
s = String.valueOf(1.45);
s = Integer.toString(45);   // Same using wrapper types
s = Double.toString(1.45);
// From String to primitive
int i = Integer.valueOf("678");
double d = Double.valueOf("4.57");

// From/to enum
String day = WeekDay.FRI.toString();
WeekDay w = WeekDay.valueOf("FRI");
```

Eftersom String inte har några subtyper måste all typomvandling ske explicit (förutom
föregående bild).

Klassen StringBuilder

```
StringBuilder sb = new StringBuilder();
out.println(sb.append("hello")
    .append(" ")
    .append("goodbye")
    .toString());   // Convert to String
```



20

Eftersom +-operatorn hela tiden skapar nya strängar och kopierar över tecken för tecken till dessa kan det bli ineffektivt

Ett bättre sätt är att använda en StringBuilder.

- StringBuilder-objekt fungerar som en muterbar sträng
- append-metoden lägger till sist i strängen (utan kopiering)
  - Smidigt med kedjade anrop
- toString omvandlar StringBuildern:s innehåll till sträng (icke-muterbar)

Hmm, skapar inte append() nya objekt (om vi använder kedjade anrop)?

- Nej, … (hur fungerar append?)

Klassen Character

```
Character.isWhitespace(' ');
Character.isDigit('1'));
Character.isLetter('X');
Character.isLetterOrDigit('2');
Character.isLowerCase('c');
Character.toString('Z').equals("Z");
```


Standardklassen (omslagstypen) Character innehåller en hel del användbara klassmetoder för enstaka tecken

Samlingar

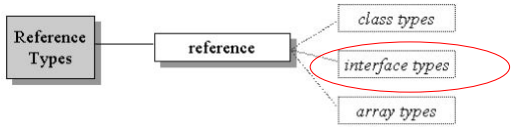
Samlingar

23

I verkligheten och i program är det mycket vanligt att man behöver hantera samlingar av objekt
Typiska operationer:

- Söka, lägga till, ändra, ta bort objektet i/ur samlingen
- Sortera samlingen, hitta olika delmängder, m.m.

Gränssnittstyper




```
public interface MyList ... {
    boolean isEmpty();
    boolean add(int i);
    int get(int index);
}

// Variable with interface type
MyList list = ... ;
list.add(4);   // Ok, method in interface type
```

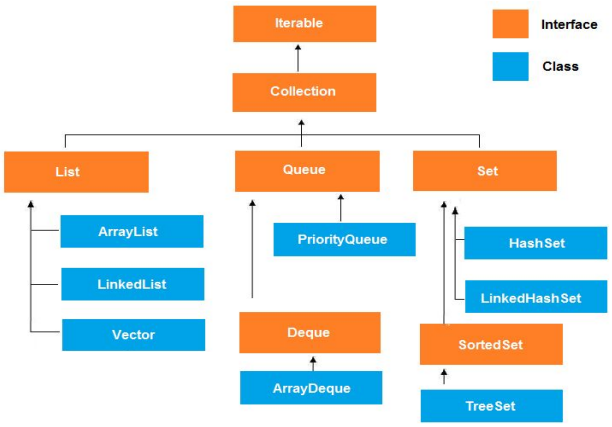
Ett **gränssnitt** (interface) är en samling av publika abstrakta metoder (public abstract behöver inte anges)

- Kan inte instansiera ett gränssnitt (med new) eftersom metoderna är abstrakta.
- Ett gränssnitt introducerar en referenstyp
  - De operationer som är tillåten för typen är de metoder vi angivit.
  - Innebär att vi kan deklarera en gränssnittsstyp för en referensvariabel
  - Vi kan alltså skapa en ny typ utan att ange någon implementation av operationerna (finns inga metoder med körbar kod)!
    - Abstraktion! Vi kan ange att vi vill ha "något" som kan utföra vissa operationer (exakt vad detta är behöver vi inte ange).

I bilden deklareras ett gränssnitt MyList med ett antal metoder.

Vi säger också att ett gränssnitt definierar ett kontrakt.

Samlingar i Java



25

The Java Collection Framework (JCF) är ett antal färdiga samlingar vi kan använda

- För att använda samlingarna måste vi lägga till import java.util.*; Samlingarna säger inte något om vilken typ av element de kan hantera
  - De är konstruerade för att klara vilken referenstyp som helst, de är **generiska**
  - Vid deklarationen anger man typen samlingen skall hantera, t.ex. List<Integer>, en lista med heltal.
- Detta är inget man lär sig utantill. Skulle det behövas får ni lämpliga metoder givna.
  - Alla moderna programmeringsspråk har liknande, huvudsaken ni vet det!
- Samlingarna räknas också som datastrukturer. Det finns en viss struktur på samma sätt som för arrayer och matriser.

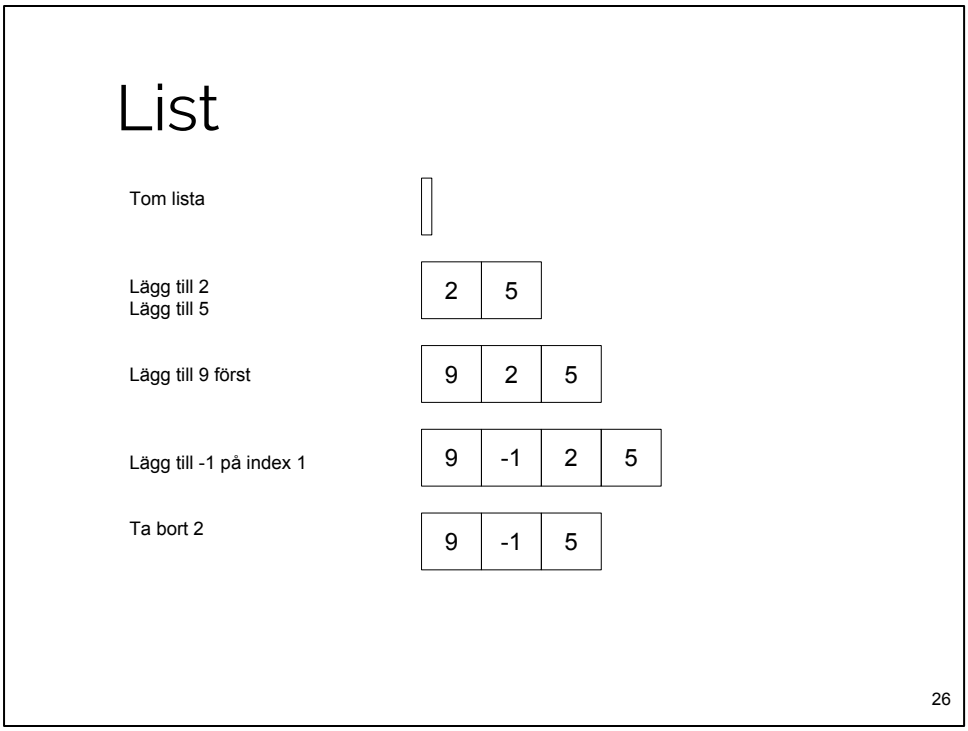
JCF specificerar samlingarna med hjälp av ett antal gränssnittstyper. Kontrakten för dessa innebär att (se bild):

- Iterable, man måste kunna traversera samlingen
- Collection, man måste kunna lägga till/ta bort element, m.m.
- List, Queue och Set anger mer exakt hur elementen skall hanteras (läggas till /tas bort)

- Övriga lämnar vi därhän …

I bilden

- De blå rutorna visar typer för objekt (klasser) som uppfyller olika kontrakt
- Pilarna visar union d.v.s
    - Klassen ArrayList skall uppfylla kontrakten Iterable, Collection och List
    - Alla metoder som finns i gränssnitten måste finnas implementerade i ArrayList-klassen
- Vi ser att det finns flera olika klasser som uppfyller samma union av kontrakt
    - T.ex. ArrayList och LinkedList
        - Dvs instanser av klasserna är utbytbara, de kan samma saker! Samma kontrakt!
        - Vi kan välja utifrån aktuell situation, smart! …



List är ett kontrakt för är en samling som representerar en linjär (ändlig) följd av element

- Påminner mycket om array men är dynamisk och har (många) metoder
  - Tom från början …
  - .. därefter växer och krymper den dynamiskt
  - OBS! Ett elements index kan alltså ändras då nya element läggs till tas bort!

Metoder i List<T>

```
// Interface type for variable, initiate with class type object
List<Integer> l = new ArrayList<>();

out.println(l.isEmpty());  // True
l.add(100);   // Put last in list
l.add(200);
l.add(300);
out.println(l.size() == 3);
out.println(l.get(2));   // Can't use [ ], use get() (0-indexed)
out.println(l.indexOf(300) == 2);

l.set(0, 500);  // Will overwrite

Integer i2 = l.remove(1);  // Remove and return removed

List<Integer> l2 = l.subList(1, 3);

for (Integer i : l) {  // Traversing
   out.print(i);       // NOTE Can't remove using this
}

list.add(null); // Very, very BAAAAD!
```


Vi deklarerar referensvariabeln som en interface-typ, objektet däremot skapas utifrån någon klasstyp.

- Finns många fler metoder …

Array kontra List

När använda vad?

- Array: Fix storlek, elementen skall behålla sina positioner (index)
- List: Alltid annars

28

List ger oss mycket mer färdig

- Dessutom är koden för List mycket grundligt testad, högre kvalitet på vårt program!
- En hel del färdiga Java-metoder returnerar Array:er
    - Isf kan vi fortsätta med array eller omvandla till List, se nedan.

Utskrift av Samlingar

```
List<Integer> list = new ArrayList<>();

// Has own toString method, nice output
out.println(list);
```


Alla samlingar har egna versioner av toString-metoden, ger läsbara utskrifter.

Kort for-loop och Samlingar

```
List<Integer> list = new ArrayList<>();

...

// Bad can't remove in short for loop
for( Integer i : list){
   if( i > 200){
       list.remove(i);  // ConcurrentModificationException
   }
}
```


Fungerar bra … men

- Man kan inte ta bort ett element i en lista som man traverserar i en kort for loop
- Ger Concurrent ModificationException
- En vanlig for-loop (med index) inte heller bra, eftersom storleken ändras (men inte indexet)
- Forts …

Ta bort i kort for-loop

```
// Collect all to remove
List<AbstractDrawableObject> hits = new ArrayList<>();

for (AbstractMovableObject m : projectiles) {
   m.move();
   if (m.intersects(ground)) {
       hits.add(m);  // Add item to remove
   } else
   ...
}
// After loop, remove all
projectiles.removeAll(hits);
```


En enkel teknik för att ta bort i kort for-loop.

- Skapa en tillfällig samling för allt som skall bort
- Lägg till i samlingen i loopen.
- Ta bort allt efter loopen.

Konvertering Array och List

```
// Use helper class Arrays to create unmodifiable
// list
List<Integer> iList1 = Arrays.asList(1, 2, 3, 4);

// Create modifiable list out of unmodifiable
List<Integer> iList2 = new ArrayList(iList1);

// Convert back to array. Must supply an array
// object as argument
Integer[] iArr = iList1.toArray(new Integer[]{});
```


Tekniskt lite rörigt, men möjligt att byta mellan List och Array

- Inget att kunna utantill

Typer

(inget nytt)

Referenser

(inget nytt)

Arrayer

(inget nytt)

Klasser

(inget nytt)

Arbetssätt

(inget nytt)