

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

# Sammanfattning

```
public class TheClassName {  
    variable declaration (instances and classes)  
    method declarations (instances and classes)  
}
```

- ▶ Ingen speciell ordning krävs men ...
- ▶ Instansvariabler skall i regel inte vara åtkomliga av utomstående (**private**)
- ▶ Instansvariabler har *defaultvärdet* beroende på typ (typiskt 0 eller null för referensvariabler)
- ▶ Instansvariabler kan initieras i deklarationen till godtyckliga uttryck (variabler i sådana uttryck måste vara definierade)
- ▶ Forward-referenser i initiering är inte tillåtna

```
public class InstvarOrder {  
    int i = j;  
    int j = 7;  
}
```

```
dhcp-11-194:Temp tobias> javac InstvarOrder.java  
InstvarOrder.java:2: illegal forward reference  
    int i = j;  
           ^
```

```
public class InstvarOrder {  
    int j = 7;  
    int i = j;  
}
```

```
dhcp-11-194:Temp tobias> javac InstvarOrder.java  
dhcp-11-194:Temp tobias>
```

# Metodedefinition

```
åtkomstmodifierare ReturTyp metodNamn(Typ1 param1, ..., TypN paramN) {  
    lokala variabler, satser och eventuell värderetur  
}
```

- ▶ Modifierarna anger t.ex. synlighet, klass/instansmetod
- ▶ Lokala variabler har inga defaultvärden utan måste tilldelas före användning
- ▶ I metoder som har en returtyp som inte är void måste alla möjliga vägar genom metoden sluta med return *exp* där *exp* är ett uttryck av den aktuella returtypen
- ▶ Metoder kan ha samma namn om de har olika *signatur* dvs skiljer sig i parameterantal och/eller typ (*överlagring*) – undvik detta i möjligaste mån, förutom för konstruktorer

# Metodanrop

```
exp.metodNamn(arg1, ..., argN);
```

- ▶ där *exp* är ett uttryck som returnerar en referens (ett värde av referenstyp)

- ▶ Exempel:

```
this.setName(first + last)  
(first + last).length()  
first.trim().substring(0,7).length()  
someMethod()
```

- ▶ I det sista fallet kommer someMethod() att *bindas* till antingen en instansmetod (=this.someMethod()) eller en klassmetod (=MyClass.someMethod()), beroende på vilken som existerar (båda är inte tillåtet i Java)
- ▶ Uttryck i parameterlistan evalueras från vänster till höger

# Konstruktorer

- ▶ En konstruktor är en metod som körs automatiskt så fort minne allokerats för ett objekt, och instansvariabler tilldelats sina "startvärden"
- ▶ En konstruktor har alltid samma namn som klassen den tillhör och saknar returtyp
- ▶ Flera konstruktorer i en klass är vanligt för att kunna skapa objekt på flera sätt (t.ex. ström från fil, ström från sökväg)
- ▶ En klass utan konstruktor får en *defaultkonstruktor* automatiskt vid kompilering av Java-kompilatoren

# Parameter och resultatöverföring

Java har *referenssemantik* för objekt och *värdesemantik* för primitiva datatyper.

- ▶ Referenssemantik = referensen till objektet kopieras, kopian pekar ut samma objekt som originalet; ger på så sätt upphov till aliasering
- ▶ Värdesemantik = värdet i variabeln kopieras i sin helhet
- ▶ Aliasering = två variabler som pekar ut samma objekt
- ▶ Minns att pekare inte finns! En variabel i en metod kan inte peka ut ett värde i en variabel i en annan metod. (Motsvarande beteende uppnås enklast med objekt.)

# Arrayer

Arrayer är objekt med attribut och metoder. Eftersom en array är ett objekt hanteras den med referenssemantik vid parameteröverföring och returnering.

Deklaration:

*typ* [] namn

*typ* [] namn = *exp*

*typ* [] namn = {*v*<sub>0</sub>, *v*<sub>1</sub>, ..., *v*<sub>*n*</sub>}

där *v<sub>i</sub>* godtyckligt beräkningsbart uttryck av rätt typ.

Instantiering:

```
new RefTyp[47]; // skapar en array av 47 referenser  
new int[4]; // skapar en array av 4 intar  
new boolean[2] {true, false};
```

# Exempel

```
public class Circles {  
    private Circle[] circles;  
    private int numberofCircles = 0;  
  
    public Circles() { circles = new Circle[5]; }  
  
    public void add(Circle c) { circles[numberofCircles++] = c; }  
  
    public static void main(String[] args) {  
        Circles circles = new Circles();  
        Point p = new Point(0,0);  
        for (int i = 1; i<=10; i++)  
            circles.add(new Circle(p, 10*i));  
    }  
}  
kursa$ java Circles  
java.lang.ArrayIndexOutOfBoundsException: 5  
    at Circles.add(Compiled Code)  
    at Circles.main(Compiled Code)  
kursa$
```

# En bättre add-metod, och några observationer

```
public class Circles {  
    private Circle [] circles;  
    private int numberofCircles;  
  
    public void add(Circle c) {  
        if (circles.length == numberofCircles) {  
            Circle[] _ = new Circle[2*circles.length];  
            System.arraycopy(circles, 0, _, 0, circles.length);  
            circles = _;  
        }  
        circles[numberofCircles++] = c;  
    }  
    ...  
}
```

- ▶ *Attributet length*
- ▶ Metoden `System.arraycopy`
- ▶ Grund kopiering (shallow copy)

## final-specifikation

Klasser, metoder och variabler kan deklareras som **final**. Här talar vi bara om final-variabler.

En variabel som är **final** måste tilldelas sitt värde vid initiering eller konstruktorn; detta värde kan sedan inte förändras

```
public class Point {  
    public final double x;  
    public final double y;  
  
    public Point(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public Point() {  
        this(0.0, 0.0); // Anropar andra konstruktorn  
    }  
}
```

# Typkonverteringar

- ▶ Java konverterar automatiskt när det kan ske ”riskfritt”
- ▶ Explicit med s.k. *typecasts*. Ex: (`short`)
- ▶ Vid aritmetik omvandlas `byte`, `short` till `int`
- ▶ Om en operand är `long` så omvandlas den andra till `long` (om den är av heltalstyp)
- ▶ Funktioner i Math returnerar `double` för att undvika precisionsförlust

# Omslagsklasser

- ▶ Primitiva datatyper kan hanteras effektivt
- ▶ Ibland vill man dock behandla dem som objekt – då finns en motsvarande klass för varje primitiv datatyp (t.ex. Integer)
- ▶ Klasserna har oftast samma namn som de primitiva typerna men följer namnprinciperna för klasser. Wrapper-klasserna för `int` och `char` heter dock Integer och Character.
- ▶ Dessa klasser innehåller
  - ▶ Ett antal *klass*-attribut med konstanter (t.ex. max och minvärde)
  - ▶ Ett antal *klass*-metoder för t.ex. konverteringar
  - ▶ Ett antal instansmetoder motsvarande för ett objekt

# Exempel: Räkna olika typer av tecken

```
import java.io.*;  
  
public class Tecken {  
    public static void main(String[] args ) throws IOException {  
        int nUpper=0, nLower=0, nOthers=0;  
        char c;  
        BufferedReader inf =  
            new BufferedReader(new InputStreamReader(System.in));  
        System.out.print("> ");  
  
        while((c=(char)inf.read()) != '\n') {  
            if (Character.isUpperCase(c))  
                nUpper++;  
            else if (Character.isLowerCase(c))  
                nLower++;  
            else  
                nOthers++;  
        }  
        System.out.println("U: " + nUpper + ", L: " + nLower + ", Ø: " + nOthers );  
    }  
}
```

## Observerationer på klassen Tecken

- ▶ Scanner-klassen passar inte för att läsa tecken (ingen "nextChar")
- ▶ Använder en BufferedReader som via en InputStreamReader kopplas till System.in. Ett standardsätt.
- ▶ Import av java.io.\*
- ▶ code returnerar en **int** i intervallet 0—65535 eller -1 vid filslut (strömslut)
- ▶ Raden **throws IOException** i början av main

Gå till javadokumentationen och titta vad som finns i klasserna omslagsklasserna! Lägg speciellt märke till metoderna parseInt och valueOf i Integer och motsvarande i Double.

# Klassen String

- ▶ Ett objekt ur klassen `String` motsvarar en följd av tecken
- ▶ Strängkonstanter omges med citationstecken

```
String foo = "Omge strängar med \"-tecken";
```
- ▶ Objekt av typen `String` är *oföränderliga* (immutable). En strängoperation som har `String` som returtyp returnerar ett helt nytt strängobjekt.
- ▶ Operatorn `+` konkatenerar strängar
- ▶ Tecknen i strängen numreras från 0, liksom i en `char`-array

## Några metoder i klassen String

```
int length()
int compareTo(String otherString)
boolean equals(Object otherObject)
char charAt(int index)
String substring(int start)
String substring(int start, int end)
int indexOf(char char)
int indexOf(String string)
String replace(char oldChar, char newChar)
String toUpperCase()
String toLowerCase()
```

Läs själva i Java-dokumentationen!

## Exempel: klassen Palindrom

```
// Läser en följd av "ord" och avgör vilka av dessa som är palindrom
import java.util.Scanner;

public class Palindrom {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (sc.hasNext()) {
            String w = sc.next().toUpperCase();
            if (w.equals("STOP") || w.equals("QUIT"))
                break;
            final int n = w.length();
            boolean pal = true;
            for (int i=0; i < n/2 && pal; i++) {
                pal &= w.charAt(i)!=w.charAt(n-1-i);
            }
            System.out.println(w + "är" + (pal ? "" : " inte") + " ett palindrom");
        }
    }
}
```

## Arrayer av strängar

Parametern till main är ett exempel på en array av strängar. När programmet startas kommer orden på kommandoraden att lagras som element i denna array. Exempel:

```
public class Echo {  
    public static void main(String[] args) {  
        for (int i = 0; i<args.length ; ++i)  
            System.out.print(args[i] + " ");  
        System.out.println();  
    }  
}
```

Med körresultat:

```
kursa$ java Echo    hej du    glade!  
hej du glade!  
kursa$
```

# Konvertering av objekt till String

Man kan definiera omvandling av objekt till typen String för t.ex. utskriftsändamål genom att definiera en metod `toString()`:

```
public class Circle {  
    private Point center;  
    private double radius;  
  
    public String toString() {  
        return "Circle(" + center + ", " + radius + ") ";  
    }  
  
    public static void main(String [] args) {  
        Circle c = new Circle(new Point(0.0 , 0.0), 1.0);  
        System.out.println(c);  
    }  
}
```

```
kursa$ java Circle  
Circle(Point@1fa4d404, 1.0)  
kursa$
```

# Dynamiska strukturer (listor, träd ...) i Java

```
// En Stack-mekanism som lagrar heltal
public class Stack {

    private StackNode top = null;

    public boolean isEmpty() { return top == null; }

    public void push(int element) {
        top = new StackNode(element, top);
    }

    public int pop(){
        int _ = top.data;
        top = top.next;
        return _;
    }
}
```

## Stackexempel forts

(Anm: Java har inbyggda klasser som kan hantera stackar, köer, etc.)

```
public class StackNode {  
    private int data;  
    private StackNode next;  
  
    public StackNode(int data, StackNode node) {  
        this.data = data;  
        this.next = node;  
    }  
}
```

**Men:** hur skall StackNode se ut?

- ▶ Accessmetoder?
- ▶ Skall StackNode vara **public**?

## Stackexempel forts, två (av flera) möjliga lösningar:

- ▶ Låt Stack och StackNode ingå i ett *paket* och ta bort **public** från definitionen av StackNode eller
- ▶ Gör StackNode till en *inre klass* till Stack

```
// fil Stack.java
package stack;

public class Stack { ... }
```

```
// fil StackNode.java
package stack;

class StackNode { ... }
```

## Stackexempel forts, två (av flera) möjliga lösningar:

- ▶ Låt Stack och StackNode ingå i ett *paket* och ta bort **public** från definitionen av StackNode eller
- ▶ Gör StackNode till en *inre klass* till Stack

```
public class Stack {  
    private class StackNode {  
        int data;  
        StackNode next;  
  
        StackNode(int data, StackNode node) {  
            this.data = data;  
            this.next = node;  
        }  
    }  
  
    private StackNode top = null;  
  
    ...
```

# Något om felhantering

Olika typer av fel: statisk/dynamisk och syntax/semantik

- ▶ Syntaxfel upptäcks av javac (motsv. stafvel)
- ▶ Semantiska statiska fel upptäcks av kompilatorn (typsystemet, dead code analysis, etc.)
- ▶ Semantiska dynamiska fel upptäcks
  - ▶ av den virtuella maskinen,
  - ▶ av den egna programkoden (felkontroll, defensiv programmering),
  - ▶ av testkod,
  - ▶ eller inte alls! – felaktiga resultat

Liksom många andra språk använder Java en generell mekanism för att hantera dynamiska, semantiska fel: *undantag* (eng. *exceptions*)

# Undantag (*exceptions*)

- ▶ När programmet/javamaskinen upptäcker att något är fel så avbryts exekveringen och ett undantag genereras
- ▶ Ett undantag är ett objekt som beskriver felet och systemets tillstånd
- ▶ Därefter signalerar systemet att ett fel har inträffat – undantaget ”kastas”
- ▶ Exekveringen fortsätter sedan i den del av programmet som förklarat sig beredd att hantera denna typ av undantag. Man säger att en *hanterare fångar* undantaget.
- ▶ Om ingen hanterare finns används en standardhanterare som avbryter programmet med en felutskrift.

# Exempel på undantagshantering

```
try {  
    operation som kan leda till undantag  
}  
    catch (ExceptionTypA e1) {  
        kod som körs om ExceptionTypA kastades  
}  
    catch (ExceptionTypB e2) {  
        kod som körs om ExceptionTypB kastades  
}
```

# Exempel på undantagshantering

```
try {  
    x = 1/0;  
}  
catch (ArithmeticException e) {  
    System.err.println("Det blev ett fel: " + e.getMessage());  
}
```

# Exempel på att kasta ett undandag

```
1 public class Die {  
2     int sides = -1;  
3     public Die(int sides) {  
4         if (sides < 2) {  
5             throw new IllegalArgumentException("A die must have 2+ sides");  
6         }  
7         this.sides = sides;  
8     }  
9 }  
10 }  
11  
12 try { new Die(0); } catch (IllegalArgumentException e) { ... }
```

Under vilka omständigheter exekveras rad 7–8?

## Undantagshantering forts

Som programmerare måste man i princip alltid tala om vad skall hända om ett fel inträffar. Tre möjligheter:

- ▶ Ta hand om felet ("fånga det"): `catch`
- ▶ Skicka det vidare: `throws`
- ▶ Kombination: fånga och skicka vidare
- ▶ Det finns undantag (!) från den obligatoriska felhanteringen – s.k. *okontrollerade* undantag.
- ▶ **Rekommendation:** använd alltid okontrollerade undantag (unchecked exceptions)
- ▶ Designproblem i Java: checked exceptions leder till dålig kod

# Exempel på att kasta ett undandag vidare

```
1 public Die makeMeANewDie(int sides) {  
2     try {  
3         return new Die(sides);  
4     } catch (IllegalArgumentException e) {  
5         // Cannot recover from this error, so pass it on  
6         throw e;  
7     }  
8 }
```

```
1 public Die makeMeANewDie(int sides) {  
2     return new Die(sides);  
3 }
```

Vad är skillnaden mellan kodexemplen?

# Javas undantag är indelade i olika kategorier:

- ▶ Error allvarliga fel som kan inte hanteras av programmeraren
- ▶ Exception
  - ▶ RuntimeException (*okontrollerade*)
    - ▶ IndexOutOfBoundsException
    - ▶ NegativeArraySizeException
    - ▶ NullPointerException
    - ▶ ArithmeticException
    - ▶ ...
  - ▶ IOException
    - ▶ EOFException
    - ▶ FileNotFoundException
    - ▶ ...
  - ▶ ...

# Exempel: För många pop

```
public class StackTest {  
    public static void main(String[] args) {  
        Stack s = new Stack();  
        s.push(1);  
        s.push(2);  
        s.push(3);  
        while (!s.isEmpty()) {  
            System.out.println(s.pop());  
        }  
        s.pop();  
    }  
}
```

```
bellatrix$ java StackTest  
3  
2  
1  
Exception in thread "main" java.lang.NullPointerException  
    at Stack.pop(Stack.java:36)  
    at Stack.main(Stack.java:48)  
bellatrix$
```

## Exempel: Fånga felet från pop

```
...
s.push(3);
while (!s.isEmpty()) {
    System.out.println(s.pop());
}
try {
    s.pop();
} catch (NullPointerException e) {
    System.err.println("Hoppla!");
}
```

```
bellatrix> java StackTest
```

```
3
```

```
2
```

```
1
```

```
Hoppla!
```

```
...
```

# Skapa en egen felklass

Gör det tydligare vilket problem som avses:

```
public class StackUnderFlowException extends RuntimeException {  
    public String getMessage() {  
        return "The stack was pop'd when it was empty";  
    }  
}
```

Modifering av pop:

```
public int pop() {  
    if (top==null) {  
        throw new StackUnderFlowException();  
    }  
    int r = top.data;  
    top = top.next;  
    return r;  
}
```

# Egen felklass forts

```
...
s.push(3);
while (!s.isEmpty()) {
    System.out.println(s.pop());
}
try {
    s.pop();
} catch (StackUnderFlowException e) {
    System.err.println("Hoppla!" + e.getMessage());
}
```

Med körresultat:

```
bellatrix$ java StackTest
3
2
1
Hoppla! The stack was pop'd when it was empty
bellatrix$
```

## Egen felklass forts

Två möjliga placering av StackException:

- ▶ I en egen fil i samma katalog som Stack (och i samma *paket* som Stack)
- ▶ Som en publik *nästlad* klass i *klassen* Stack:

```
public class Stack {  
    public static class StackUnderFlowException extends RuntimeException {  
        ...  
    }  
  
    private static class StackNode {  
        ...  
    }  
  
    ...  
}
```

# Skillnad mellan nästlad klass och inre klass

En inre klass är kopplad till det omslutande *objektet*:

```
Stack stack = new Stack();
StackNode stackNode = new stack.StackNode();

// Inside the Stack class, we can simply write new StackNode(), why?
```

En nästlad klass är kopplad till den omslutande *klassen*:

```
try {
    while (true)
        System.out.println(s.pop());
} catch (Stack.StackUnderFlowException e) {
    System.out.println("Hoppla! " + e.getMessage());
}
```

# Skillnad mellan nästlad klass och inre klass

Alla kontrollerade undantag som kan kastas i en metod måste antingen fångas eller så måste metoden ange att den kan skicka kontrollerade undantag (**throws**):

```
public String getPath(File someFile) throws IOException {
    return someFile.getCanonicalPath();
}
```

```
public String getPath(File someFile) {
    try {
        return someFile.getCanonicalPath();
    } catch (IOException e) {
        return null;
    }
}
```

# 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 element `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<sup>1</sup>. 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 kompilar!
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?

---

<sup>1</sup>Se [http://sv.wikipedia.org/wiki/Personnummer\\_i\\_Sverige](http://sv.wikipedia.org/wiki/Personnummer_i_Sverige).