

# **Introduktion till objektorientering, grundläggande Java**

# Vad är objekt-orientering?

- ▶ Allt är objekt
- ▶ Inkapsling (tillstånd och beteende)
- ▶ Polymorfism
- ▶ Dynamisk bindning
- ▶ I någon bemärkelse även arv (men vi väntar med det)

Java:

- ▶ Statiskt typat, klass-baserat OO-språk
- ▶ Automatisk minneshantering
- ▶ Inte rent objekt-orienterat: primitiva datatyper
- ▶ Enkelt implementationsarv
- ▶ Multipelt gränssnittsarv

# Objekt kontra struct

- ▶ En strukt är en “död” samling data
- ▶ Alla operationer på datat definieras externt i procedurer och funktioner – *man gör saker med datat*
- ▶ Betrakta följande C-struktur – varför kan man säga att den är ”passiv”?

```
struct person {  
    char* firstName;  
    char middleInitial;  
    char* lastName;  
    char* ssn;  
    int age;  
};
```

## Objekt kontra struct (forts.)

- ▶ Hur kan man se till att age alltid är ett vettigt tal?
- ▶ Hur kan man se till att ssn alltid följer 2-1-metoden?
- ▶ Hur tar man fram en persons hela namn?
- ▶ Hur hanteras aliasering?

## Objekt kontra struct (forts.)

- ▶ Objekt är "aktiva" – och tar ansvar för sitt eget data

```
class Person {  
    private String firstName, middleInitial, lastName, ssn;  
    private int age;  
  
    String getFullName() { return firstName + " " +  
                           middleInitial + " " + lastName; }  
  
    void setAge(int age) {  
        if (age >= 0 && age < 120) {  
            this.age = age;  
        } else {  
            // do nothing for now, in future signal an error  
        }  
    }  
  
    void setSSN(String ssn) { ... }  
}
```

# Ännu bättre

```
class Person {  
    private String firstName, middleInitial, lastName;  
    private Personnummer ssn;  
    private int age;  
  
    String getFullName() { return firstName + " " +  
                           middleInitial + " " + lastName; }  
  
    void setAge(int age) {  
        if (age >= 0 && age < 120) {  
            this.age = age;  
        } else {  
            // do nothing for now, in future signal an error  
        }  
    }  
  
    void setSSN(Personnummer ssn) { ... }  
}
```

## Gör personunnummret intelligent" (OBS fulkod)

```
class Personnummer {  
    private int[] numbers;  
    Personnummer(int[] numbers) {  
        int[] copy = new int[10];  
        if (numbers.length != 10) ... // error, too few numbers  
        int sum = 0;  
        for (int i=0; i<9; ++i) {  
            sum += (numbers[i]*(2-i%2) / 10 + numbers[i]*(2-i%2) % 10);  
            copy[i] = numbers[i];  
        }  
        if (numbers[9] != 10 - (sum % 10)) ... // error, bad checksum  
        copy[9] = numbers[9];  
        this.numbers = copy;  
    }  
  
    String toString() {  
        String result = "";  
        for (int i=0; i<10; ++i) result += numbers[i];  
        return result;  
    }  
}
```

# Metoder kontra funktioner

- ▶ En metod opererar alltid på ett objekt (**this**)
- ▶ Objektet måste finnas för att man skall kunna anropa en metod på det
- ▶ Publika metoder, privata data – inkapsling
- ▶ Specialfall: konstruktorer
  - ▶ Kan bara anropas en gång, när objektet skapas
  - ▶ Svåra att få till korrekt, mer om det senare
  - ▶ Om det finns en konstruktor måste den anropas vid instansiering
  - ▶ På så sätt kan man se till att objekt alltid har korrekta värden (jmf. Personunnummer)
- ▶ Polymorfism: olika objekt kan ha olika implementationer för en metod med samma namn

# Polymorphism

```
class Cowboy {  
    void draw() { ... }  
}  
  
class Circle {  
    void draw() { ... }  
}  
  
Cowboy c1 = new Cowboy();  
Circle c2 = new Circle();  
  
c1.draw();  
c2.draw();
```

## Polymorphism (forts.)

```
interface Drawable {  
    void draw();  
}  
  
class Cowboy implements Drawable {  
    void draw() { ... }  
}  
  
class Circle implements Drawable {  
    void draw() { ... }  
}  
  
void someMethod(Drawable d) {  
    d.draw();  
}
```

## Sammanfattning: OO

- ▶ Tankesättet kretsar kring objekt som inkapslar tillstånd och beteende
- ▶ Ett objekt slår vakt om sitt datas integritet
- ▶ Istället för göra något med datat (procedurellt) ber man datat att utföra någonting – vad som händer är upp till objektet (OO)
- ▶ Resultatet blir separation och abstraktion, vilket underlättar konstruktion och underhåll av system
- ▶ Objekt specificeras normalt genom klasser som beskriver alla objekt av en viss "typ"
- ▶ Ett objekt *instantieras* genom att man ber klassen om att skapa en instans av sig själv
- ▶ Ovanstående är fullt möjligt även i C, bara inte lika enkelt

# Vad är Java?

Man kan mena två olika saker:

- ▶ *Programmeringsspråket*
- ▶ *Plattformen*. Definierar en omgivning i vilken programmen exekverar.

När man laddar ner Java *software development kit* (SDK) så får man

- ▶ en kompilator (`javac`)
- ▶ en virtuell maskin (`java`)
- ▶ ett *klassbibliotek* eller *API* (application programming interface)

# Karakteristik av språket Java

- ▶ Objektorienterat
- ▶ Statiskt typat (som C)
- ▶ Syntaktiskt likt C (och C++)
- ▶ Väldefinierat
- ▶ Automatisk skräpsamling
- ▶ Säkert (kontroll av arraygränser, typer, odefinierade värden . . . )

# Hello world i Java

```
public class HelloWorld {  
  
    public static void main(String [] args) {  
        System.out.println("Hello world!");  
    }  
}
```

Observationer:

- ▶ Funktionen ("metoden") main är förpackad i en *klass*
- ▶ Ordet public
- ▶ main returnerar ingenting (typ void)
- ▶ static annan betydelse än i C
- ▶ main har *ett* argument som är en array av strängar (lite annan syntax)
- ▶ Utskrift på standard output med System.out.println

## Vad kan vi ta med oss från C?

- ▶ De grundläggande datatyper: int, float, double, ... men med exakta definitioner av talområden och precision
- ▶ variabeldeklarationer
- ▶ operatorerna + - \* / ++ - = += == < <= ...
- ▶ Satsen: **if**, **for**, **while**, **do**, **switch**
- ▶ Syntaxen för "funktioner" som här för det mesta kallas för *metoder*

## Vad kan vi *inte* ta med?

- ▶ preprocessorn
- ▶ programstruktur med funktioner på filer
- ▶ deklarationsfiler
- ▶ pekare – motsvaras av *referenser* som är mycket mer begränsade
- ▶ **struct** – ersätts av *klass* som är ett *mycket* kraftfullare begrepp

# Primitiva datatyper

Typnamn	datatype	minnesutrymme	exempel
byte	heltal	1 byte	-127, 47
short	heltal	2 byte	4711
int	heltal	4 byte	-748471
long	heltal	8 byte	434112345L
float	flyttal	4 byte	-4.57e10f
double	flyttal	8 byte	3.123e-128
boolean	logisk	1 byte	<b>true, false</b>
char	tecken (Unicode)	2 byte	'x', '4', '+', '\n'

# Klassen String

Förutom de primitiva datatyperna kan programmen hantera *objekt*. Ett objekt tillhör alltid en viss klass.

Exempel: Den inbyggda klassen String

```
String s;
String t = "sträng";
System.out.println("Konkatenering av " + t + "ar");
s = "Denna strängs längd: ";
System.out.println(s + s.length());
s = "sträng";
if (s==t)
    System.out.println("Detta kommer INTE att skrivas");
if (s.equals(t))
    System.out.println("Detta kommer att skrivas");
```

## Klassen String forts

Alltså:

- ▶ Typen String avser ett objekt av klassen String
- ▶ String v deklarerar en variabel v som antingen refererar till en sträng eller null
- ▶ Vid tilldelning till en String-variabel behöver vi inte tänka på att frigöra minnet för den gamla strängen
- ▶ String-värden kan *konkateneras* med additionsoperatorn (+), skapar ett nytt objekt
- ▶ Automatisk typkonvertering vid konkatenering
- ▶ Operationer ("metoder") (length, equals, ...) definierade för strängobjekt. *Punktnotation* (jfr -> i C).
- ▶ Relationsoperatorerna == och != testar *referenslikhet* – inte om objekten innehåller samma data (jfr string-pekar i C)

## Utmatning i terminalfönstret

System.out.print(String s)

System.out.println(String s)

System.out.println()

Enbart radbyte

Från och med Java 5 finns en metod med namnet printf. Exempel:

```
System.out.printf("Längden av strängen '%s' är %d\n", s, s.length());
```

Den inbyggda klassen NumberFormat kan användas för mer avancerade formateringar i enlighet med olika nationella konventioner.

## Scanner-klassen

Kan användas för att läsa ord, tal mm (s.k. "tokens") från tangentbordet.

Koppla ett Scanner-objekt till inströmmen:

```
Scanner sc = new Scanner(System.in);
```

Några metoder:

sc.hasNext()	boolean
sc.next()	String
sc.nextLine()	String
sc.hasNextInt()	boolean
sc.nextInt()	int
sc.hasNextDouble()	boolean
sc.nextDouble()	double

## Exempel: Tabell med funktionsvärden

```
// TableIO.java - Demonstrarer användning av Scanner
import java.util.Scanner; // <<< import

class TableIO {
    public static void main(String[] args) {
        double x, xlow, xhigh;
        int number;
        Scanner sc = new Scanner(System.in);
        System.out.print("Undre gräns: ");
        xlow = sc.nextDouble();
        System.out.print("Övre gräns: ");
        xhigh = sc.nextDouble();
        System.out.print("Antal värden: ");
        number = sc.nextInt();
        double step = (xhigh - xlow) / (number-1);
        for (int i = 1; i<=number; i++) {
            x = xlow + (i-1)*step;
            System.out.print(x);
            System.out.println("\t" + Math.log(x)); // <<< Math.log
        }
    }
}
```

## Exempel: Tabell med funktionsvärden forts

Körresultat:

```
vega$ javac TableIO.java
vega$ java TableIO
Undre gräns: 0
Övre gräns: 10
Antal värden: 11
0.0 -Infinity
1.0 0.0
2.0 0.6931471805599453
3.0 1.0986122886681096
4.0 1.3862943611198906
5.0 1.6094379124341003
6.0 1.791759469228055
7.0 1.9459101490553132
8.0 2.0794415416798357
9.0 2.1972245773362196
10.0 2.302585092994046
```

## Exempel: Tabell med funktionsvärden forts

Vad händer om man matar in felaktiga indata?

```
vega$ java TableIO
Undre gräns: 0
Övre gräns: 10
Antal värden: 12.3
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:819)
        at java.util.Scanner.next(Scanner.java:1431)
        at java.util.Scanner.nextInt(Scanner.java:2040)
        at java.util.Scanner.nextInt(Scanner.java:2000)
        at TableIO.main(TableIO.java:16)
vega$
```

Vi skall titta mer på felhantering senare.

## Klasser och objekt

Klasser används ofta för att beskriva någon typisk enhet i programmet. t ex något konkret fysiskt objekt (*bil, kund, kassa, kö*) eller något mer abstrakt begrepp (*teckenström, skärmfönster*) eller kanske något ännu mer abstrakt (*fyrdimensionellt klot, funktion ...*)

En klass är alltså en abstrakt beskrivning av en viss typ av objekt.

Objekten karakteriseras av

- ▶ **egenskaper** eller **attribut** t ex *färg, form, bränslemängd, kölängd, expedieringstid, ...* och
- ▶ **operationer** eller **metoder** dvs vad man kan göra med dem (*fylla på bensin i ett bilobjekt, ta ut första värdet ur ett köobjekt, ändra storleken på ett fönsterobjekt ...*)

## Exempel: En Kund-klass

```
public class Kund {  
    private int ankomstTid;  
    private int expTid;  
  
    public Kund(int aTid, int eTid) { // En konstruktur  
        ankomstTid = aTid;  
        expTid = eTid;  
    }  
  
    public int getAnkomstTid() { // selektor  
        return ankomstTid;  
    }  
  
    public void setExpedieringsTid(int expTid) { // mutator  
        this.expTid = expTid;  
    }  
}
```

## Observationer på Kund-klassen

- ▶ Två privata attribut
- ▶ En konstruktor som ger värden till attributen
- ▶ Två publika metoder som returnerar värdet på respektive attribut

Operatorn **new** används för att skapa objekt. Exempel:

```
Kund k1 = new Kund(10, 5);
Kund k2 = new Kund(11, 20);
System.out.println("Sammanlagd etid: " + (k1.getEtid() + k2.getEtid()));
```

Observera parenteserna i anropet till `println`!

## Exempel: En tärningsklass

Antag att man vill representera en eller flera tärningar.

Vilka egenskaper (attribut) och vilka operationer (metoder) skall vi ge tärningar?

Om vi skall använda klassen för att simulera "slå tärning(ar) och titta på resultatet" så kan vi ignorera flera av de egenskaper som verkliga tärningar har: färg, storlek, material ...

Egenskaper vi behöver: *antal sidor* och *aktuellt värde*.

Operationer vi behöver: *skapa tärning*, *slå tärning* och *avläs värde*.

## Exempel: En tärningsklass (forts.)

```
public class Die {  
    private int number0fSides;  
    private int value;  
  
    public Die() { // Konstruktor  
        number0fSides = 6;  
    }  
  
    public Die(int nS) { // Konstruktor  
        number0fSides = nS;  
    }  
  
    public int roll() { // Mutator  
        return value = (int) (Math.random()*number0fSides) + 1;  
    }  
  
    public int get() { return value; } // Selektor  
}
```

(Klassen är inte perfekt men vi kan inte få allt på en gång ...)

## Observationer på Die-klassen

- ▶ Två konstruktorer – s.k. *överlagring*
- ▶ Klassen Math med metoden random
- ▶ Privata attribut, publika metoder
- ▶ Tilldelning har värde
- ▶ Inget static-deklarerat
- ▶ "Typecast" som t ex (int)

## Die-klassen forts

Hur skapar man en tärning?

```
Die t1 = new Die(); // Tärning med 6 sidor  
Die t2 = new Die(42); // Tärning med 42 sidor  
Die t0 = null; // Variabel som (ännu) inte refererar en tärning  
t1.roll(); // Slår den ena tärningen. (Behöver inte ta emot värdet)  
t0.roll(); // Nonsense, kraschar under körning
```

**Fråga:** Var kan man göra detta?

**Svar:** I (och endast i) andra metoder.

**Insikt:** Jag måste alltså ha skapat ett objekt innan jag kan anropa dess metoder ...

**Förundrad fråga:** Hur skapas då det första objektet?

**Svar:** Klasser är objekt under körning och skapas av Javas VM.

## Die-klassen (forts.)

```
public class Die {  
    private int numberOfSides;  
    private int value;  
  
    public Die() { numberOfSides = 6; }  
  
    public Die(int nS) { numberOfSides = nS; }  
  
    public int roll() {  
        return value = (int) (Math.random()*numberOfSides) + 1;  
    }  
  
    public int get() { return value; }  
}  
  
public class Program {  
    public static void main(String [] args) {  
        Die t = new Die();  
        for ( int i = 1; i<20; i++ )  
            System.out.println(t.roll());  
    }  
}
```

# Klasser, programstruktur och konventioner

- ▶ Ett program består av en eller flera *klasser* samlade i ett eller flera paket
- ▶ Varje klass lagras på en *fil* med *samma namn*
- ▶ Klassnamn *skall* börja på stor bokstav men attribut och metoder på liten.
- ▶ Attributen görs vanligen *private* medan metoderna oftast är *public* (default åtkomstmodifikator är *package*)
- ▶ En klass *main*-metod kan anropas vid programmets start och blir på så vis ett sätt en väg in i ett program (metoden måste vara deklarerad exakt som i exemplen ovan)

## Exempel: Slå tärningar till par

Användning av tärningsklassen från en annan klass.

```
public class RollUntilEqual {  
  
    public static void main(String [] args) {  
        Die t1 = new Die();  
        Die t2 = new Die();  
        int n = 1;  
        while (t1.roll() != t2.roll())  
            n++;  
        System.out.println("Antal kast till par: " + n);  
    }  
}
```

```
bellatrix$ ls -l
-rw-r--r--+ 1 tom      it          1009 Sep 23 20:47 Die.class
-rw-r--r--+ 1 tom      it          510 Aug 19 14:00 Die.java
-rw-r--r--+ 1 tom      it          251 Sep 23 21:18 RollUntilEqual.java
bellatrix$ javac RollUntilEqual.java
bellatrix$ ls -l
-rw-r--r--+ 1 tom      it          1009 Sep 23 20:47 Die.class
-rw-r--r--+ 1 tom      it          510 Aug 19 14:00 Die.java
-rw-r--r--+ 1 tom      it          776 Sep 23 21:24 RollUntilEqual.class
-rw-r--r--+ 1 tom      it          251 Sep 23 21:18 RollUntilEqual.java
bellatrix$ java RollUntilEqual
Antal kast till par: 13
bellatrix$ java RollUntilEqual
Antal kast till par: 1
bellatrix$ java RollUntilEqual
Antal kast till par: 4
bellatrix$ java RollUntilEqual
Antal kast till par: 4
bellatrix$
```

## Exempel: array

```
import java.util.Scanner;

public class CheckDie {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Number of sides: ");
        int nSides = sc.nextInt();
        int[] freq = new int[nSides];
        Die d = new Die(nSides);

        for (int i= 1; i<=1000; i++ ) {
            freq[d.roll()-1]++;
        }

        for (int i= 0; i<nSides; i++)
            System.out.println( (i+1) + "\t" + freq[i] );
    }
}
```

## Observationer på CheckDie-klassen

- ▶ Flera huvudprogram (`main()`) - bara ett används
- ▶ Användning av en *array*
- ▶ Syntaxen i array-deklarationen (placeringen av `[]`)
- ▶ Arrayer skapas dynamiskt med `new`
- ▶ Arrayer hanteras med referenser
- ▶ Minsta index 0 i array

# 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?