

# Variabler, värden och typer

Viktigt att förstå på djupet:

- För programmering i många språk, t.ex. Java
- För kommande objektorientering!

# Fråga – kommentera – avbryt!



# Vad är en variabel?

Dags att vara (över)tydlig...

# Intro till variabler (1)



- Vad är en variabel?
  - I begynnelsen fanns *minnet*...
    - ...som var fullt av *heltal*...
  - ...och *minnesadressen* (ett "index" för varje byte)
    - **STA 49152** // **Lagra en byte** på adress 49152  
// Håll själv reda på att inget annat ska lagras där!
    - **LDA 49152** // **Läs in en byte** från adress 49152  
// Håll själv reda på hur denna byte ska tolkas  
// (Heltal? Bokstav? Index i lista av färger? ...)
    - **JMP 8282** // **Hoppa** till nästa instruktion på adress 8282



00000

64 kbyte minne

49152

(På denna sida: 8-bitars 6502-assemblerkod)

$$2^{16} - 1 = 65535$$

00000

- Sedan uppfanns *etiketten* (label)

- `colornum:` // **Översätts** till en adress, **kanske 37000**,  
  `.byte 03` // när man vet **var det finns plats**
- `STA colornum` // **Lagra** på namngiven minnesadress
- `LDA colornum`
- `JMP colornum` // Hoppa till adress `colornum`...  
  // **Oops**, undrar vad färgnumret betyder  
  // när det tolkas som en *instruktion!*

64 kbyte minne

37000

En första nivå av abstraktion!

Konkret adress,  
37000

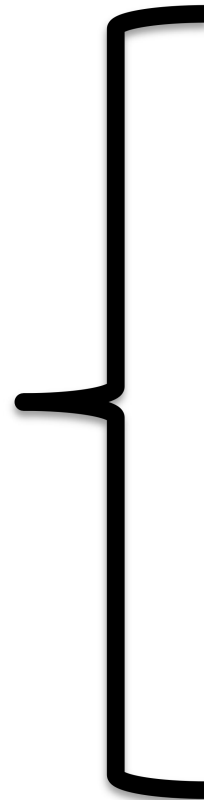


Abstrakt namn,  
`colornum`

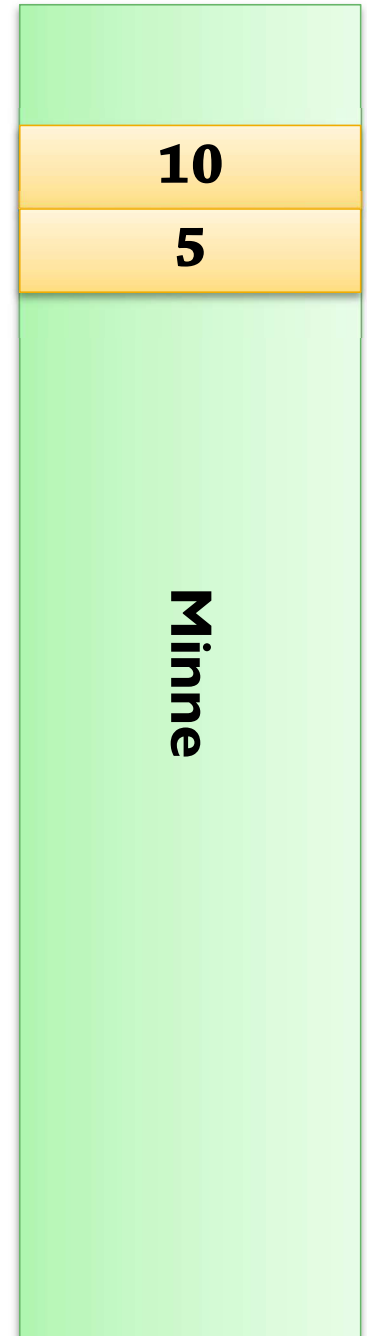
65535

- **Abstrahera språket mer → en variabel:**

- En **lagringsplats**  
(en eller flera bytes)
- Ett **symboliskt namn**  
på lagringsplatsen
  - längd = 10  
höjd = 5
- Skillnad på *variabler*  
och *funktioner*
  - Bara "vettiga" operationer  
är möjliga...
  - `längd(); // Går inte!`



|       |             |
|-------|-------------|
| längd | 10000–10003 |
| höjd  | 10004–10007 |



**Minnesadressen kan göras  
osynlig i språket!**

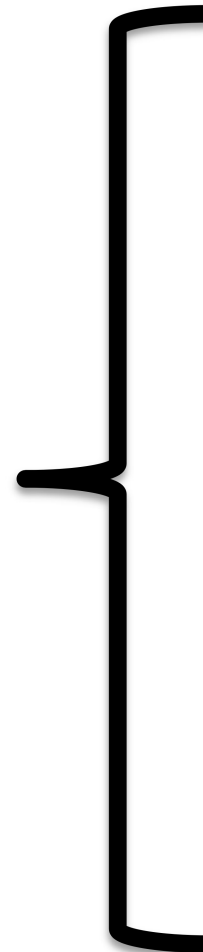
**Men den *finns* ändå där...**

# Intro till variabler (4)

- Kan stödja fler datatyper:

- Strängar
- Listor
- ...

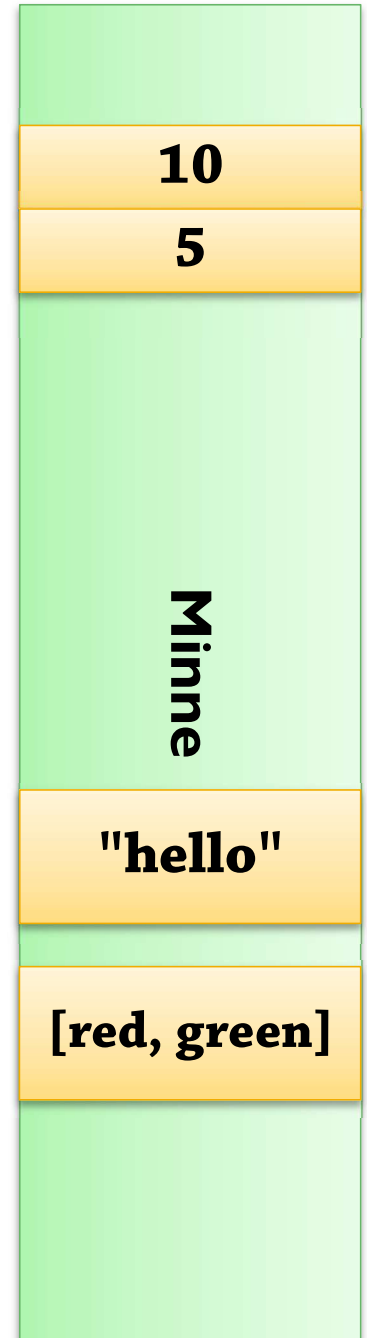
- längd = 10  
höjd = 5  
hälsning = "hello"  
färger = [red, green]



|       |             |
|-------|-------------|
| längd | 10000-10003 |
| höjd  | 10004-10007 |

|          |             |
|----------|-------------|
| hälsning | 40000-40024 |
|----------|-------------|

|        |             |
|--------|-------------|
| färger | 50000-50020 |
|--------|-------------|



# Intro till variabler (5)



## ■ Vi kan skriva över gamla värden...

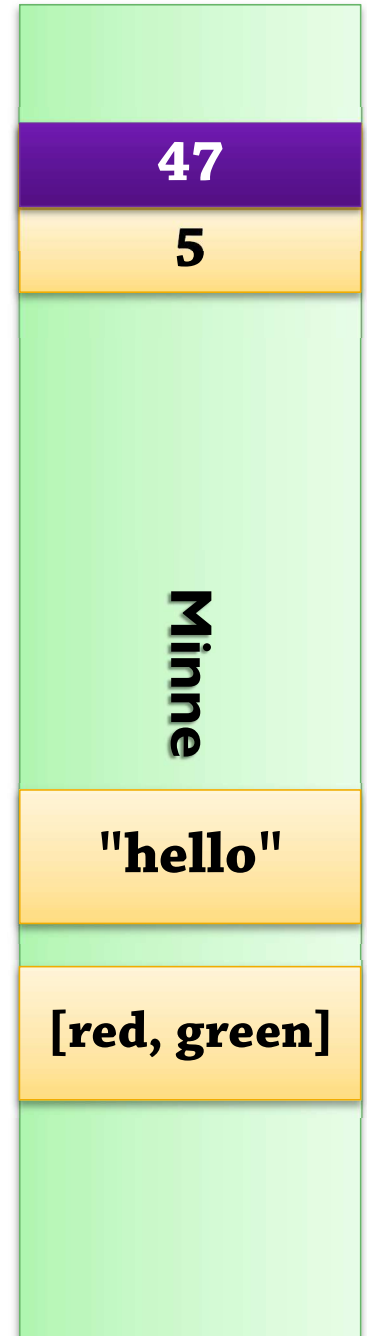
- längd = 10  
höjd = 5  
hälsning = "hello"  
färger = [red, green]

|       |             |
|-------|-------------|
| längd | 10000-10003 |
| höjd  | 10004-10007 |

- längd = 47  
*# Samma variabel,*  
*# samma lagringsplats,*  
*# samma minnesadress (som vi oftast inte vet / bryr oss om!),*  
*# nytt värde...*

|          |             |
|----------|-------------|
| hälsning | 40000-40024 |
|----------|-------------|

|        |             |
|--------|-------------|
| färger | 50000-50020 |
|--------|-------------|





# Variabler: Sammanfattning

- Så: En **variabel** används för att **lagra** ett värde, och består av:
  - En **lagringsplats** i minnet, där ett värde kan placeras
  - Ett **symboliskt namn** på lagringsplatsen, som används i koden

```
Python
längd    = 10
höjd     = 5
hälsning = "hello"
färger   = [red, green]
```



**Variabel = en "låda" för ett värde:  
Värdet kan bytas ut (längd = 22),  
men det är fortfarande samma variabel**

**Typer:**  
**För värden och variabler**

- Varje värde har en typ – heltal, decimaltal, ...
  - Vissa språk *håller inte reda* på typen

## Assembler: *Ingen* typkontroll

```
// Lagra 32-bitars heltal (L = Long)  
MOVE.L #10, längd  
// Läs som om det vore 32-bitars flyttal  
FMOVE.S längd, FP1
```

Värdetyp "sparas" inte,  
kontrolleras aldrig

Inget fel signaleras

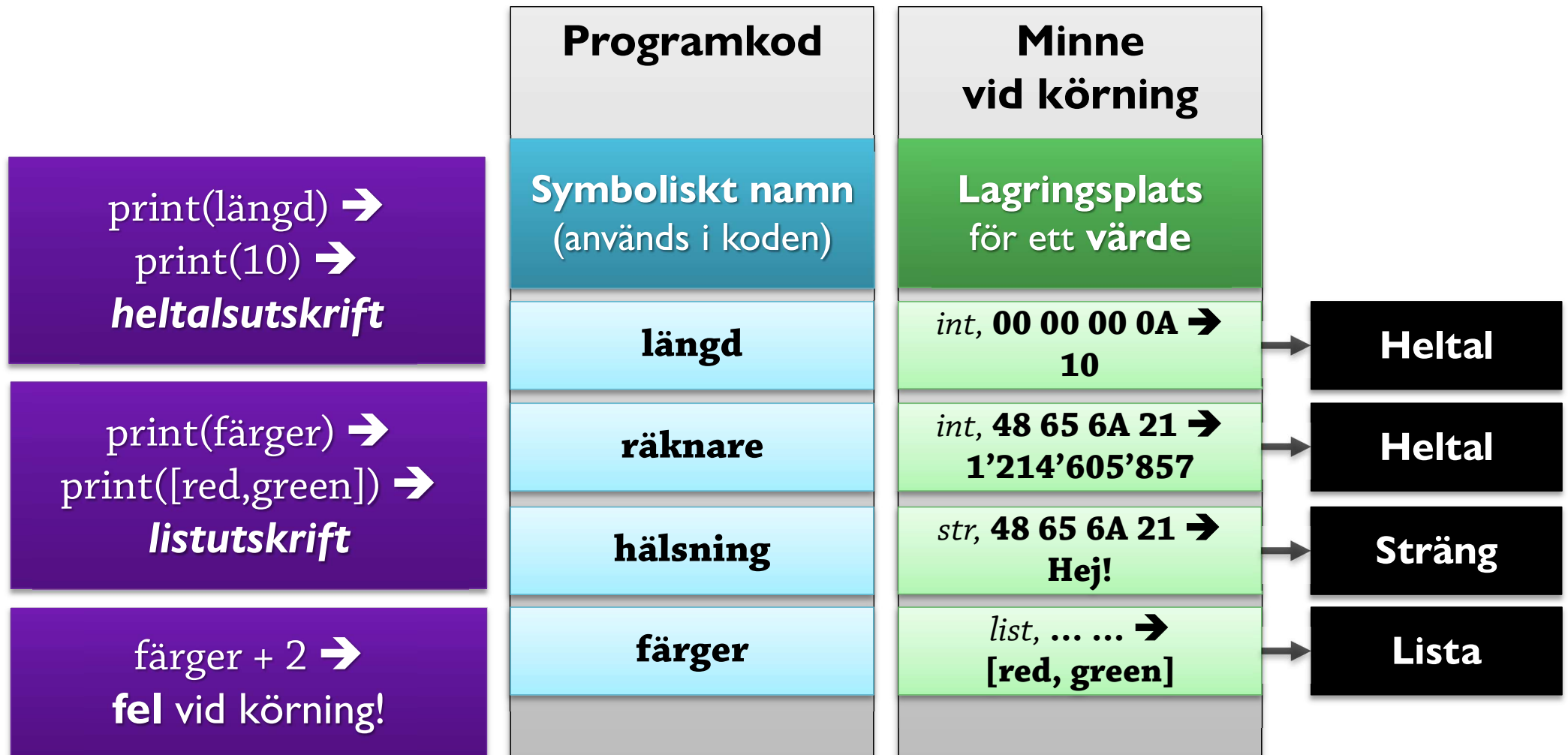
Resultat: "Skumma värden"

## Exempel:

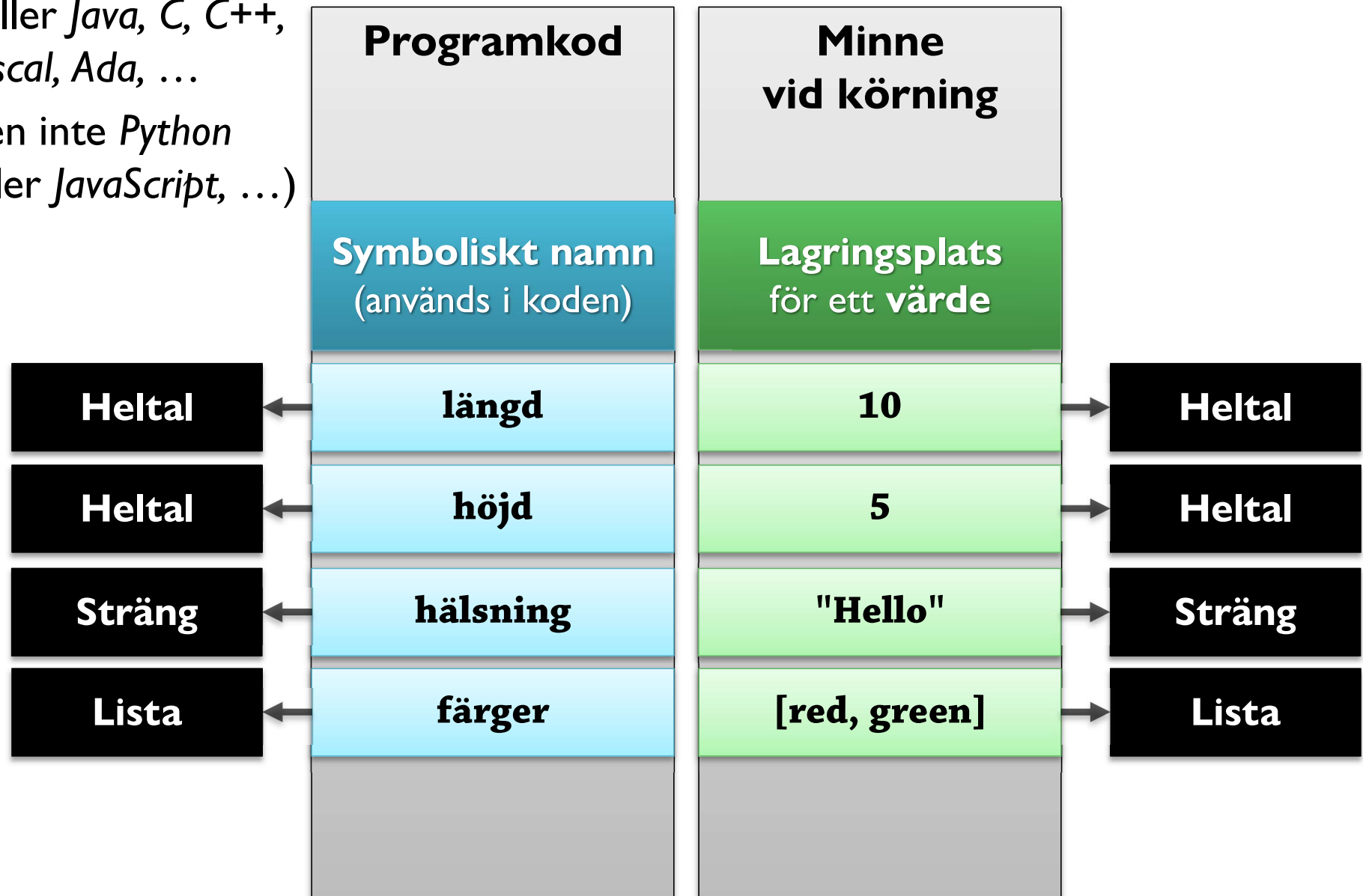
Lagra decimaltal 3.5 → 4 bytes, hex 40 60 00 00

Läs som heltal, 4 bytes, hex 40 60 00 00 → heltal 1'080'033'280

- Men de flesta, även Python, håller reda på värdets typ
  - Annars skulle inte **isinstance()** fungera... Och mycket annat



- I många språk har även variabeln en typ
  - Gäller *Java*, *C*, *C++*, *Pascal*, *Ada*, ...
  - Men inte *Python* (eller *JavaScript*, ...)



**Java: Manifest typsystem  
(variabeltyp måste anges  
explicit i koden)**

**int längd = 10;**  
→ längd är en *heltalsvariabel*  
som **alltid** innehåller ett heltal

**Python: Latent typsystem  
(bara värdet har en typ)**

**längd = 10**  
→ längd är en *vad-som-helst-variabel*  
som **just nu** innehåller ett heltal



Varför variabeltyper? Varför ange dem explicit?

# När vet man värdetyper?



- **Latent** typsysteem:

Python

```
def send(x):  
    # Kommer x att vara heltal här? Flyttal? Sträng, lista, ...?  
    # Ingen aning förrän programmet körs – får kontrolleras dynamiskt
```

- **Manifest** typsysteem:

Java

```
public void send(int x) {  
    // Här är x ett heltal, och det vet vi vid kompilering – statiskt  
}
```

**Kompilatorn vet mer (kan optimera mer → effektivare)**

**Kodanalysen vet mer (kan tala om när vi gör fel)**

**Vi vet mer (typerna är *dokumentation*)**

# Men har inte Python type hinting?



- Type hinting i Python:

Python

```
def send(x: int):  
    # Kommer x att vara heltal här? Flyttal? Sträng, lista, ...?  
    # Ingen aning förrän programmet körs – får kontrolleras dynamiskt
```

- Kallas hinting, för det är inget krav
  - Kan skicka in vad som helst
  - Kan vara felaktiga "hints"
  - Kan alltså inte optimera utifrån detta



## Python: Dynamisk typkontroll

```
def doSomething(x):  
    y = x + 10  
    ...
```

Gäller även med type hinting: `...(x: int)`

## Java: Statisk typkontroll

```
void doSomething(int x) {  
    int y = x + 10;  
    ...  
}
```

Dessutom: Typdeklarationer är *dokumentation!*

Går det att addera?

Kolla värdetyp vid körning  
Klarar inte "+": Signalera fel

```
>>> langd=10  
>>> halsning="Hello"  
>>> halsning+langd  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: cannot concatenate 'str'  
and 'int' objects
```

Går det att addera?

Kolla variabeltyp vid kompilering  
Klarar inte "+": Signalera fel

Tidigare upptäckt av problem  
→ effektivare programmering,  
färre krascher

Mindre typkontroll vid körning  
→ effektivare

dynamic type checking  
static type checking

# Konsekvenser: Val av operation



## Python: *Dynamisk* typkontroll

```
def doSomething(x):  
    y = x + 10;
```

## Java: *Statisk* typkontroll

```
void doSomething(int x) {  
    int y = x + 10;  
    ...  
}
```

**Måste kolla vid körning:**

**x heltal?**

**Addera 10 direkt...**

**x flyttal?**

**Konvertera 10 till 10.0, addera**

**...**

**x annat?**

**Signalera fel!**

**x är heltal!  
Addera direkt**

**(Kompileta direkt till  
heltalsaddition i maskinkod!)**

## Python: *Dynamisk* typkontroll

```
def doSomething(x):  
    y = x + 10;
```

## Java: *Statisk* typkontroll

```
void doSomething(int x) {  
    int y = x + 10;  
    ...  
}
```

### Hur kolla typen hos x:s värde?

Måste lagra typen med värdet



→ delvis därför  
kan ett heltal ta 24 bytes

### Hur kolla typen hos x:s värde?

Variabelns typ är int,  
värdet måste ju ha *samma* typ...



→ Ett heltal tar 4 bytes (int),  
8 bytes (long)

## Bättre kodanalys – t.ex. för *komplettering* (ctrl-shift-space)

```
import java.util.Random;

public final class RandomInteger {

    public static void main(String[] args){
        log("Generating 10 random integers in range 0..99.");

        Random randomGenerator = new R...
        for (int idx = 1; idx <= 10; idx++){
            int randomInt = randomGenerator.nextInt(100);
            log("Generated : " + randomInt);
        }

        log("Done.");
    }

    private static void log(String aMessage){
        System.out.println(aMessage);
    }
}
```

Mycket annat börjar på R,  
men bara dessa har rätt typ

- En brasklapp:
  - **Terminologin** för typsystem är **ofta otydlig och omtvistad**
  - Många termer brukar blandas ihop
    - **Statisk typning**
    - **Statisk typkontroll**
    - **Manifest typning**
    - **Stark typning**
    - ...
  - **Det viktigaste är *begreppen och deras konsekvenser***

**Även dynamisk typning (Python) har fördelar!**

Mindre att skriva

Mer flexibilitet ("på eget ansvar")

# Mer Java: Egenskaper hos primitiva datatyper

Primitiv: **grundläggande, odelbar, ...**

# Primitiva (grundläggande) typer i Java



Heltalstyper – lika på alla plattformar!

|              |          | <u>minsta värde</u>   | <u>största</u>       |  |
|--------------|----------|-----------------------|----------------------|--|
| <b>byte</b>  | 8 bitar  | -128                  | 127                  | } Används sällan<br>Sparar minne i array |
| <b>short</b> | 16 bitar | -32768                | 32767                |  |
| <b>int</b>   | 32 bitar | -2147483648           | 2147483647           | Vanligast!                               |
| <b>long</b>  | 64 bitar | -9223372036854775808L | 9223372036854775807L |  |

"L" indikerar "långt heltal"

Två flyttalstyper – skiljer i *precision* och *storlek*

|               |          |                               |                              |
|---------------|----------|-------------------------------|------------------------------|
| <b>float</b>  | 32 bitar | $\pm 3.40282347E-45$          | $\pm 3.40282347E+38$         |
| <b>double</b> | 64 bitar | $\pm 4.9406564584124654E-324$ | $\pm 1.797769313486...E+308$ |

Övrigt

|                |                          |
|----------------|--------------------------|
| <b>boolean</b> | false, true              |
| <b>char</b>    | tecken (värdet 0..65535) |

```
public class JavaTest {  
    public static void main(String[ ] args) {  
        int massor = 131072 * 131072;  
        System.out.println("128k * 128k är: " + massor);  
    }  
}
```

Operationer på heltal kan ge overflow – "översvämning"!

- **Operander** av typ `int`: [-2147483648, 2147483647]
- → Multiplikation av 32-bitarstal:
  - $0b10000000000000000000 * 0b10000000000000000000 = 0b1000$
- → 128k \* 128k är: 0

**32 bitar slutresultat**



```
public class JavaTest {  
    public static void main(String[ ] args) {  
        int massor = 131072 * 131072;  
        System.out.println("128k * 128k är: " + massor); // Skriver ut 0  
    }  
}
```

## Varför overflow i Java, men inte Python?

- **Historiskt...**
  - "Så var det ju i C och C++"
- **Effektivitet!**
  - Java: 32-bitars multiplikation, *klar*. Använd **BigInteger** för väldigt stora tal.
  - Python: Testa storlek, allokeras så mycket minne som behövs för resultat,  
...

```
public class JavaTest {  
    public static void main(String[ ] args) {  
        long massor = 131072 * 131072;  
        System.out.println("128k * 128k är: " + massor);  
    }  
}
```

Beräkningar använder den **största** av **operandernas** typer

- 131072 är en *int* (inget "L")
  - $0b10000000000000000000 * 0b10000000000000000000 = 0b1000000000000000000000000000000000$

**32 bitar slutresultat**

- Sedan** expanderas detta till 64 bitar
  - $128k * 128k$  är: 0

Korkat? Mer förutsägbart:  
Resultat beror bara på  
*operandernas typer*,  
"bottom up"

# Använd större datatyp (2)



```
public class JavaTest {  
    public static void main(String[ ] args) {  
        long massor = 131072L * 131072;  
        System.out.println("128k * 128k är: " + massor);  
    }  
}
```

Beräkningar använder den största av operandernas typer

- Största operanden är long
  - Expandera den andra "131072" till long
  - Utför 64-bitars multiplikation
  - 128k \* 128k is: 17179869184  
( $2^{34}$ )

Expanding  
tappar aldrig information  
→ sker automatiskt!

# "Farliga" typkonverteringar



```
public class JavaTest {  
    public static void main(String[] args) {  
        int sqrtPi = Math.sqrt(Math.PI);  
        System.out.println("Sqrt( $\pi$ ) är: " + sqrtPi);  
    }  
}
```

Konstanter och funktioner  
i Math-klassen

Kompileringsfel! `Math.sqrt` returnerar en *double*

- Att konvertera *double* till *int* kan tappa information – "farligt"
- Måste uttryckligen **be om** konvertering!

```
public class JavaTest {  
    public static void main(String[ ] args) {  
        int sqrtPi = (int) Math.sqrt(Math.PI);  
        System.out.println("Sqrt( $\pi$ ) är: " + sqrtPi);  
    }  
}
```

Konvertera med en *cast*

## Beräkning, sedan trunkering (avhuggning)

- Beräknar värdet: 1,7724538509055160272981674833411
- (int) *trunkerar* detta till: 1
- Mer:
  - `int i = (int) 271828.1828;` // OK — `i = 271828` (trunkerat)
  - `short s = (short) 271828;` // OK — `s = 9684` (lägsta 16 bitarna)
  - `0b1000010010111010100`

```
public class JavaTest {  
    public static void main(String[ ] args) {  
        double sqrtPi = Math.sqrt(Math.PI);  
        System.out.println("Sqrt( $\pi$ ) är: " + sqrtPi);  
    }  
}
```

Byt variabeltyp...

## Beräkningar

- Beräknar värdet: 1,7724538509055160272981674833411
- Skriver ut det

# Vad är sant / falskt?



## Python: Auto-konvertering

### Falska värden:

False, None, 0, 0.0, "", (), [], {}, ...

### Sanna värden:

Allt annat

```
längd = 10
```

```
if längd: # Om inte 0, (), False, ...
```

```
    print(längd)
```

Ofta bekvämt  
Kan leda till misstag

## Java: Bara boolean-värden

Falska värden: false

Sanna värden: true

Allt annat: inte sanningsvärde!

Operationer: ! && || (och fler)

```
int längd = 10;
```

```
if (längd) ... // Fel!
```

```
if (längd != 0) {
```

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

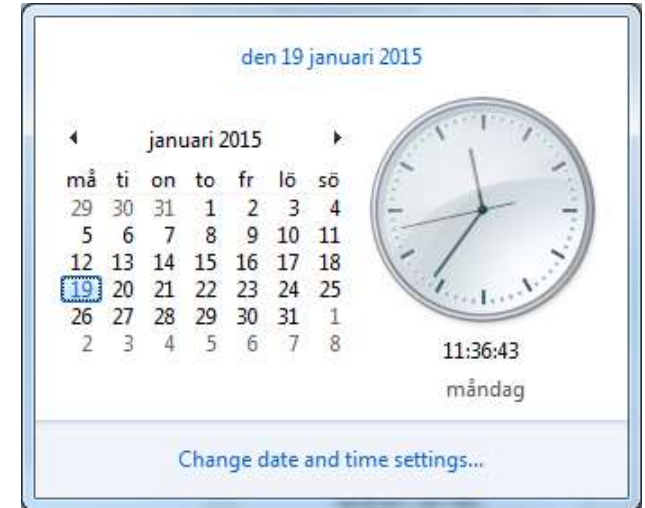
```
}
```

Mer att skriva  
Ibland tydligare  
Kan förhindra misstag

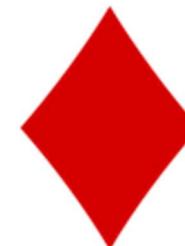
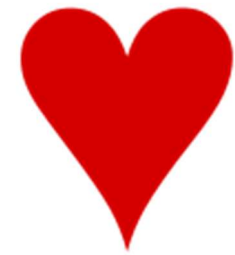
Våra första egna typer:  
Uppräknings typer



- Vissa typer ska bara ha några få fördefinierade värden
  - Day: Monday, Tuesday, ..., Sunday



- Suit: Clubs, Diamonds, Hearts, Spades



- Kan emuleras på många sätt, t.ex. med heltalskonstanter

- **final** int MONDAY = 0, TUESDAY = 1, ..., SUNDAY = 6;
- **public** void setDayOfWeek(int day) { ... }

final: Ändra inte värdet

- Inte typsäkert!

- setDayOfWeek(42); // **Accepteras av kompilatorn...**  
// **Men vi vill ha tidiga varningar!**
- final int BOLD = 2;  
setFont("Times", 14, BOLD); // **Vilken betyder 14 punkter fetstil?**  
setFont("Times", BOLD, 14); // **Vilken betyder 2 pt blinkande?**

- Stödjer nonsensoperationer

- int blah = TUESDAY \* SUNDAY + WEDNESDAY

- Efter kompilering finns bara heltalen kvar – svårare att tolka

- Ska 4 betyda torsdag eller fredag? Eller grönt, spader, giraff...?

- Java har stöd för uppräkningstyper (enumerated types)
  - Man räknar upp vilka värden som finns

```
public enum Day {  
    MONDAY,  
    TUESDAY,  
    ...,  
    SUNDAY  
}
```

Sju *enum*-konstanter,  
namn i ALL\_CAPS

Inga nonsens-operationer  
tillgängliga

- Distinkt typ (inte *int*) → typsäkerhet

```
public void setDayOfWeek(Day day) {  
    ...  
}  
setDayOfWeek(Day.TUESDAY);
```

```
System.out.println(Day.THURSDAY);
```

Håller reda på namn:  
Skriver ut "THURSDAY", inte 4

- Vi kan få ut ett värde med *givet namn* och en *lista på alla värden*

```
String dayname = getInputFromUserOrFile(); // "TUESDAY"  
Day d = Day.valueOf(dayname);
```

```
for (Day d : Day.values()) { // Iterera över array (förklaras senare)  
    System.out.println("En av dagarna är " + d);  
}
```

## Sämsta lösningen: Konstanter utan namn

```
if (state == 4) {  
    if (player.hitWall()) state = 1;  
    else ...;  
}
```

## Gammal lösning: Namngivna heltal

```
final static int STATE_STANDING = 1;  
final static int STATE_RUNNING = 4;  
if (state == STATE_RUNNING) {  
    if (player.hitWall()) state = STATE_STANDING;  
}
```

## Bäst, vid fixerade värden: Uppräkningsbar typ

```
enum State { RUNNING, STANDING }  
if (state == State.RUNNING) {  
    if (player.hitWall()) state = State.STANDING;  
    ...  
}
```

**Arrayer**

# Java's arrays: Som listor, men på lägre nivå

## Begränsad funktionalitet, fixerad storlek

```
Circle[] circles;  
int[] numbers;
```

För varje typ T  
finns en arraytyp T[]

```
Circle[] circles = new Circle[10];  
int[] numbers = new int[rnd()];
```

Arrayer skapas med **new**  
Storleken sätts när de skapas, ändras aldrig

```
int second = numbers[1];
```

Indexering börjar på 0. Ingen slicing, n[1:5]!

```
for (int i = 0; i < numbers.length; i++) {  
    System.out.println(numbers[i]);  
}
```

Index: 0...arr.length-1

**numbers**

|                |        |
|----------------|--------|
| Object header: | (data) |
| length         | 42     |
| [0]            | 0      |
| [1]            | 0      |
| [41]           | 0      |

Tomma platser  
skapas och  
*initialiseras*:  
0, 0.0, false,  
null (pekare!)

```
int val = numbers[42000];
```

Indexkontroll: Ger  
**ArrayIndexOutOfBoundsException**



- Arrayer: Begränsad funktionalitet
  - Används mest för att **implementera andra datatyper**
  - Användarkod använder oftare **listor**, som **ArrayList<Circle>** -- senare!
- Lite biblioteksfunktioner:
  - <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Arrays.html>
  - Jämförelser, sökning, sortering, ...

# Sammanstatta datatyper: Poster / records / struct / ...

Generellt – inte Java

- Vi vill lagra information om **bilar**
  - **Många egenskaper** för varje bil
    - *Längd – flyttal*
    - *Höjd – flyttal*
    - *Toppfart – heltal*
    - *Färgkod – heltal*
  - Ska ses som **en enhet**
    - En funktion ska ta *en bil* som *en parameter*, inte 4:  
`void addToDatabase(Car c) { ... }`
    - Behöver **sammansatta** datatyper, som NamedTuple i Python

sammansatta datatyper =  
composite datatypes

## Bil

Längd – flyttal

Höjd – flyttal

Toppfart – heltal

Färgkod – heltal

- Försök 1: Använd **generella** sammansatta datatyper
  - Exempel från tidigare år i Python-kursen: **Listor till allt**

```
def skapa_bil(längd, höjd, toppfart, färg):  
    return ["bil", längd, höjd, toppfart, färg]  
minbil = skapa_bil(...)
```

- Problem:
  - Ingen riktig typ: Datatypen är inte **bil** utan **lista** – "*lista av vad som helst*"
  - Måste själv implementera typkontroller, mm.
  - "Delarna" har inga namn:  
minbil[3] = 200; # Använd index 3 för att hitta toppfarten

- Försök 2: Använd en mappning / dictionary

```
def skapa_bil(längd, höjd, toppfart, färg):  
    return { "type": "bil",  
            "length": längd, "height": höjd,  
            "topSpeed": toppfart, "color": färg }
```

```
minbil = skapa_bil(...)  
minbil["topSpeed"] = 200 # Ändra toppfart
```

- **Ger namngivna värden, men:**
  - Ingen egen typ (allt är ett dictionary)
  - Ingen standard för vilka data som finns om alla bilar

- Försök 3: Skapa nya typer med fält, som NamedTuple i Python
  - Pascal: Poster (records)

```
Type bil = Record  
  längd, höjd, toppfart: Real;  
  färg: Color;  
End;
```

Posten har namngivna  
medlemmar (fält)

Detta är "ritningen"  
för typen

```
Var minbil : bil;
```

Kan *deklarera* en variabel av  
typ *bil*, inte av typ *lista*  
Tillåter *statisk typkontroll*

```
minbil.toppfart := 200
```

Egenskapers värden  
tas fram via *namn*,  
inte via *index*

post = record  
medlem = member  
fält = field

# Sammanstatta datatyper (4)



- Fler exempel på sammanstatta datatyper
  - Vanliga, "fundamentala" typer
    - `String`
  - Generella behållare
    - `List`, `Queue`, `Stack`
  - Specialiserade typer för I/O, GUI, databaser, ...
    - `InputStream`, `Window`, `Connection`, ...
  - Och typer specifika för ett visst program
    - `Driver`, `Car`, `Wheel`
    - `Shape`, `Rectangle`, `Circle`
    - `Customer`, `BankAccount`, `Transaction`
    - `Literal`, `Conjunction`, `Disjunction`, `Quantifier`

## Viktigt: Skapar nya meningsfulla begrepp!

- Vi **samlar ihop** information – och ger den också ett **namn**
  - → Skapar nya ord i språket (**Customer**, **BankAccount**);  
sitter inte fast i språkets egna begrepp (lista, dict, ...)
  - → Vi kan lättare **skriva** kod, **förstå** existerande kod:  
**Begreppen** kan anpassas mer till **hur vi tänker**  
("en kund", inte "en lista med dessa 17 kundrelaterade värden")
- **Söndra och härska** (divide and conquer)!
  - Dela upp programmeringen i **delar av lämplig storlek**
  - Se till att varje enskild del kan **förstås i detalj**

I Java används klasser... Dags för objektorientering!