Variabler, värden och typer

Viktigt att förstå på djupet:

– För programmering i många språk, t.ex. Java

– För kommande objektorientering!

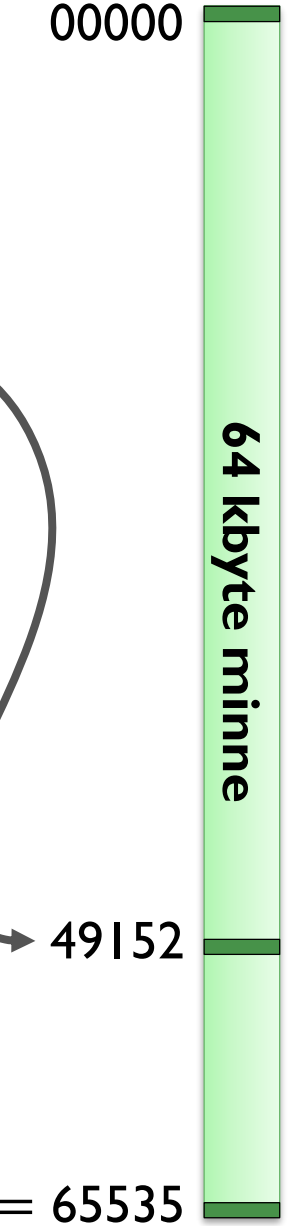
Fråga – kommentera – avbryt!

Intro till variabler (1)

jonkv@ida

- 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

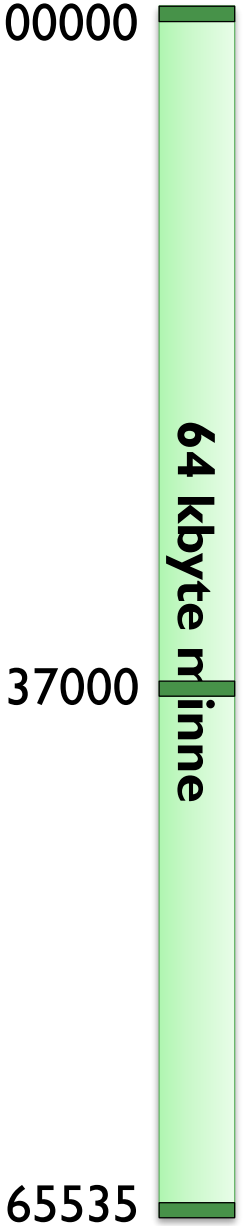
(På denna sida: 6502-assemblerkod)



Intro till variabler (2)

jonkv@ida

- **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* // **Oops**, undrar vad färgnumret betyder
    // när det tolkas som en *instruktion*!

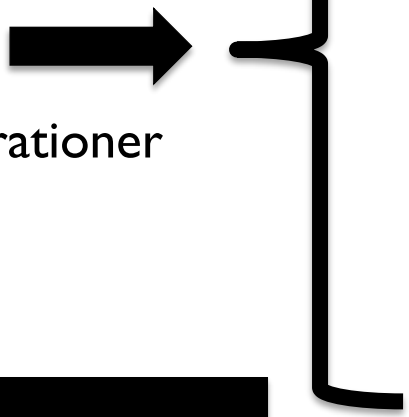


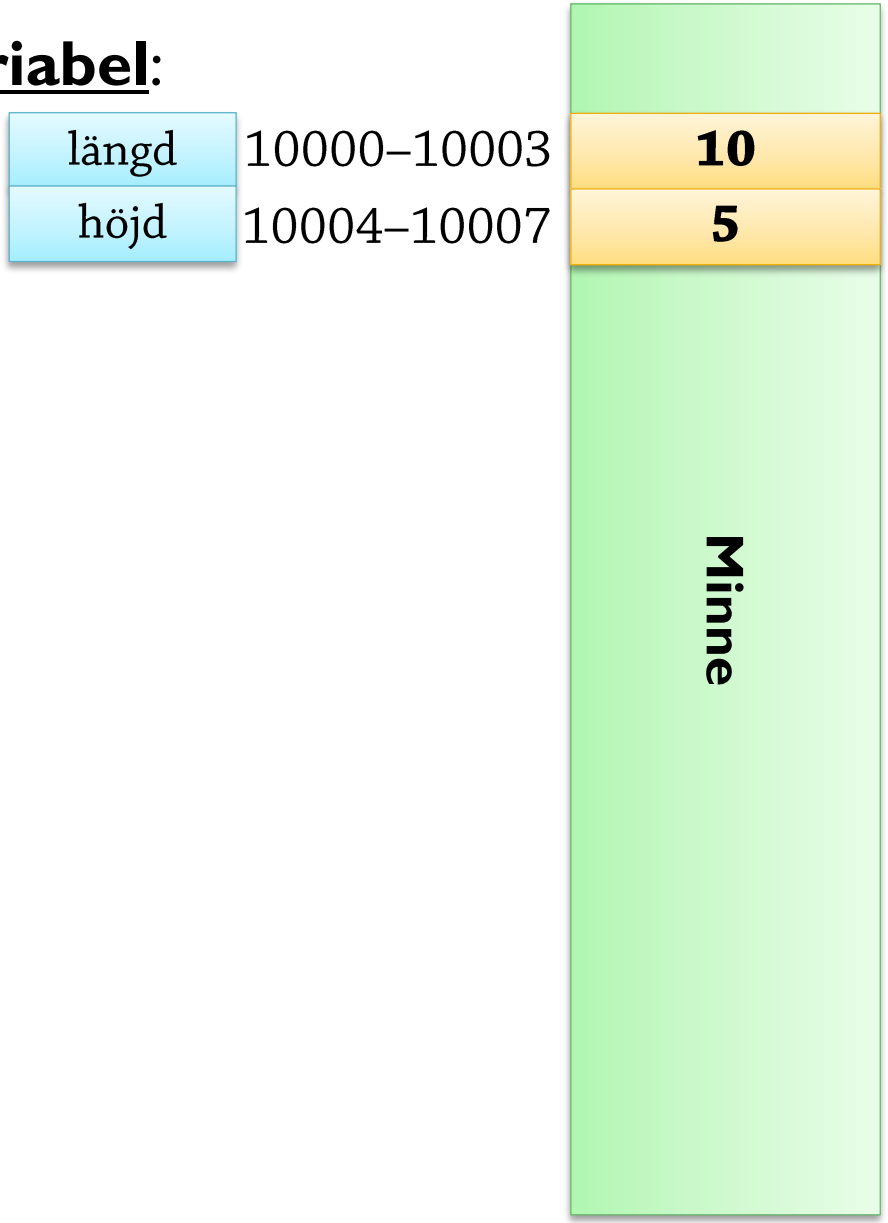
En första nivå av abstraktion!

Konkret adress, 37000 → **Abstrakt namn, colornum**

Intro till variabler (3)

- Abstrahera språket mer ➔ en **variabel**:
  - En **lagringsplats** (en eller flera bytes)
  - Ett **symboliskt namn** på lagringsplatsen
    - längd = 10
      höjd = 5
  - Bara "vettiga" operationer är möjliga…
    - ~~JMP colornum;~~





Minnesadressen kan bli osynlig i språket!

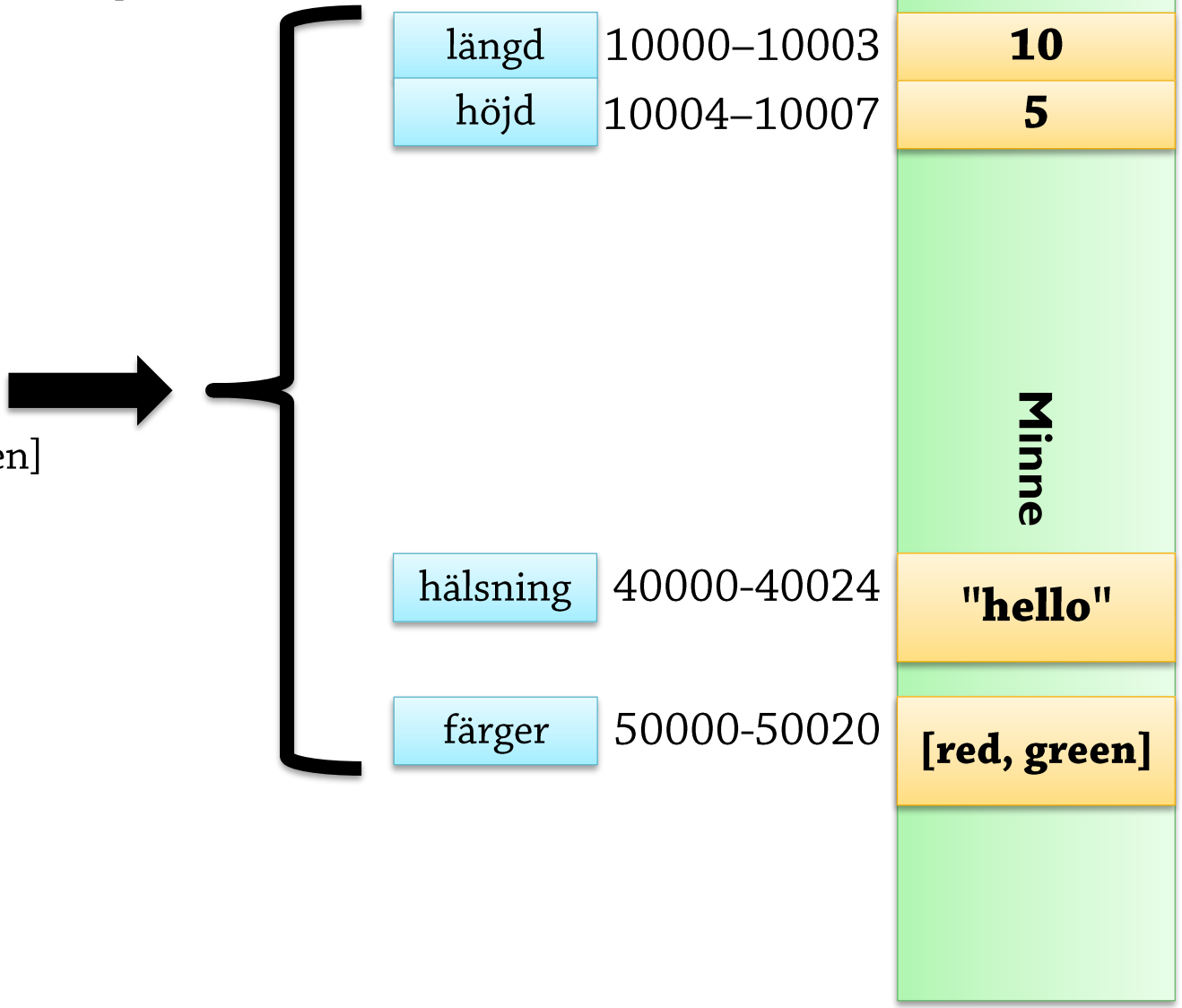
Men den *finns* ändå där…

Intro till variabler (4)

jonkv@ida

- Kan stödja fler **datatyper**:
  - Strängar
  - Listor
  - …

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

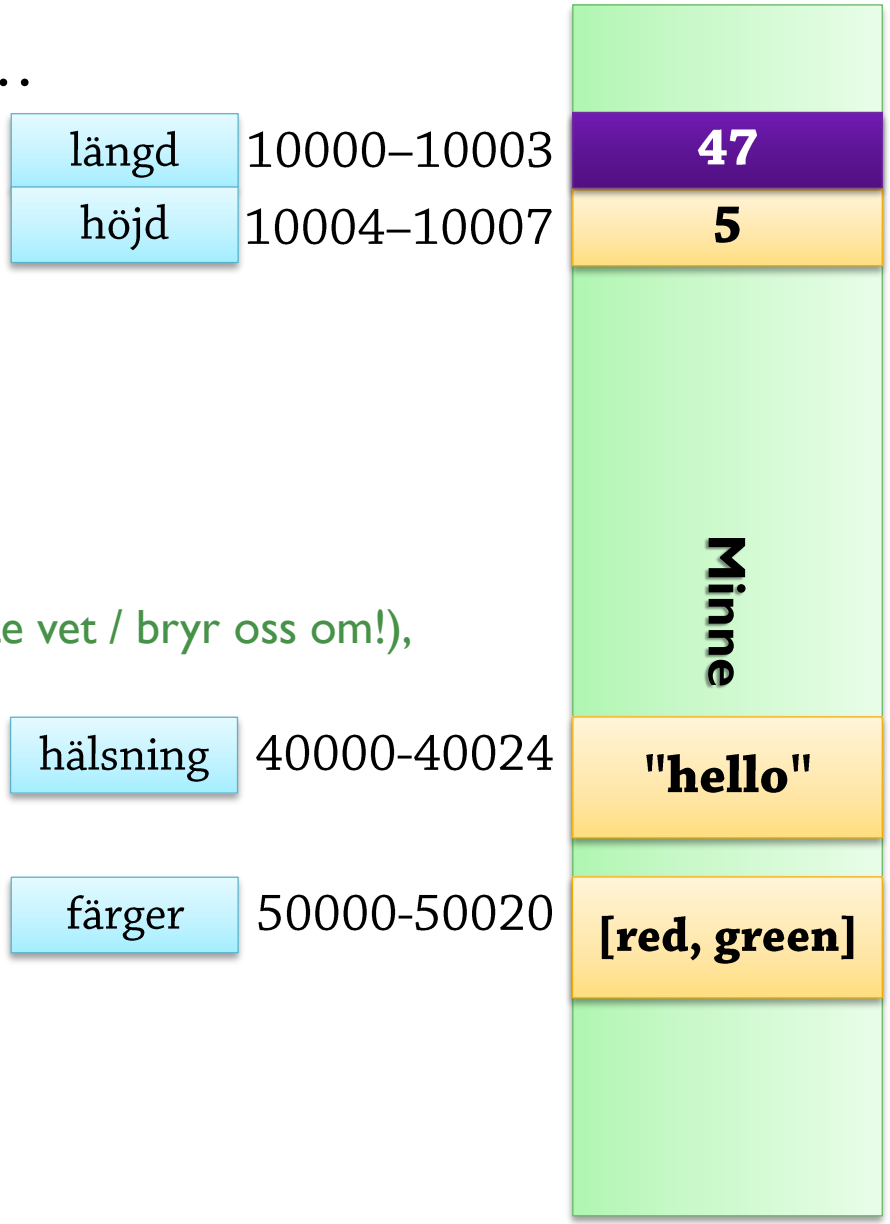


Intro till variabler (5)

jonkv@ida

- Vi kan **skriva över** gamla värden…
  - längd = 10
    höjd = 5
    hälsning = "hello"
    färger = [red, green]

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



Variabler: Sammanfattning

jonkv@ida

- 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]
```

Programkod	Minne vid körning
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

Värdetyper

jonkv@ida

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

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"

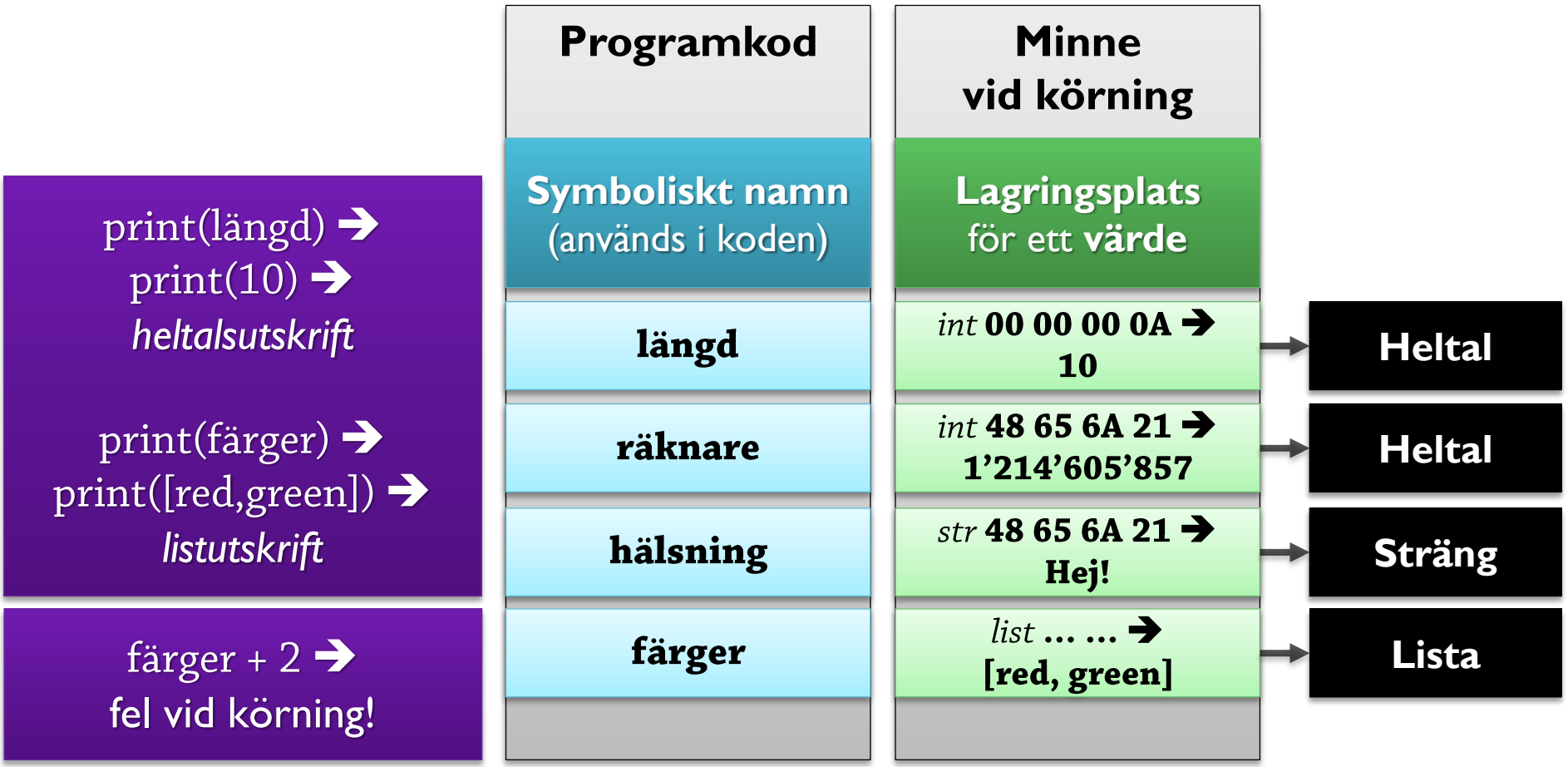
Exempel:

Lagra decimaltal 3.5 → 4 bytes, hex 40 60 00 00
Läs som heltal, 4 bytes, hex 40 60 00 00 → heltal I'080'033'280

Värdetyper

jonkv@ida

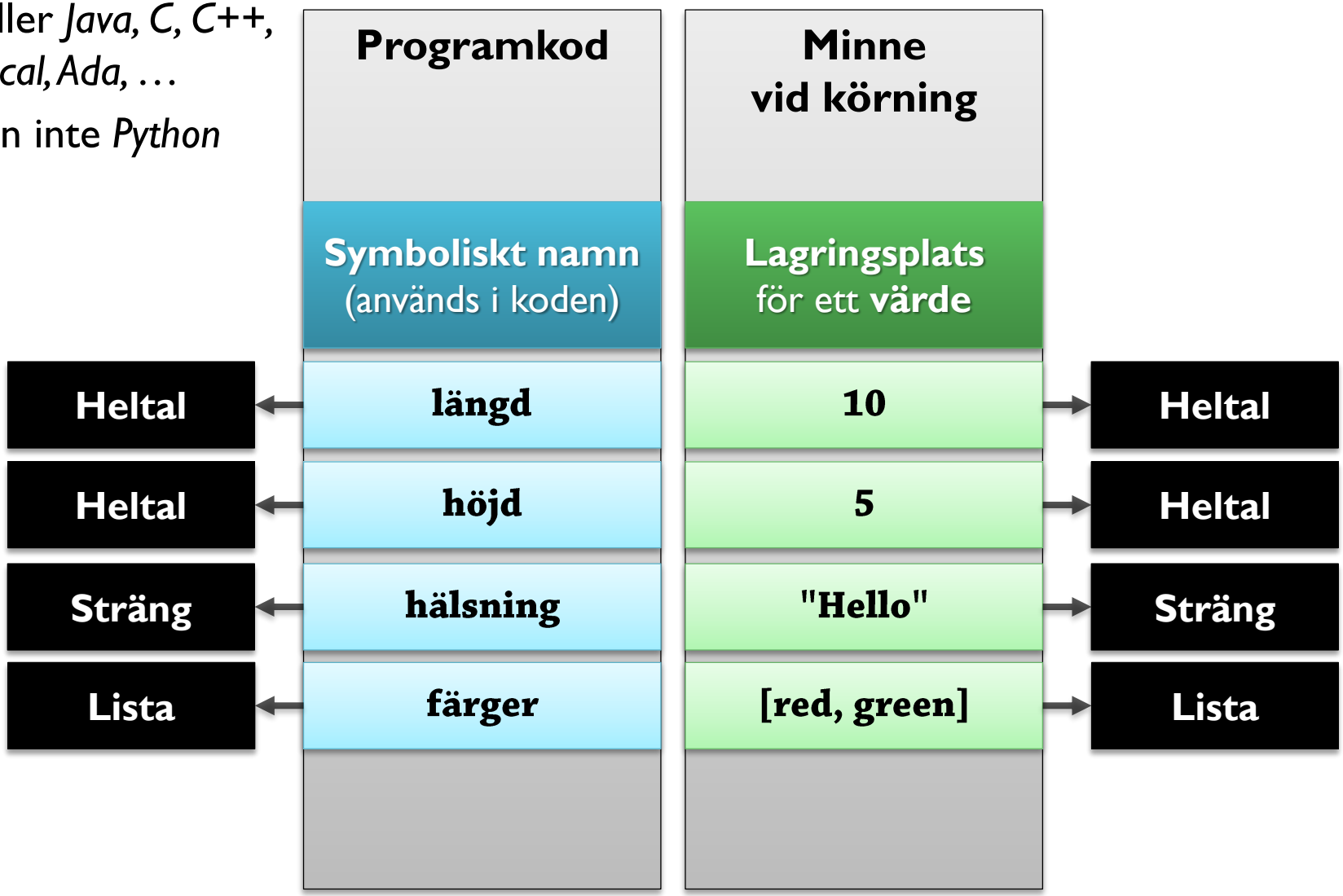
- Men de flesta, även Python, **håller reda på** värdets typ
  - Även om man sällan *skriver* typnamnet



Variabeltyper

jonkv@ida

- I många språk har även **variabeln** en **typ**
  - Gäller *Java, C, C++, Pascal, Ada, …*
  - Men inte *Python*



Variabeltyper

Java: *Manifest* typsystem
(*variabeltyp* 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

<u>Varför</u> variabeltyper? Varför ange dem explicit?

När vet man värdetypen?

jonkv@ida

- Latent typsystem:

Python

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

- Manifest typsystem:

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)
Vi vet mer (typerna är *dokumentation*)

Konsekvenser: Typkontroll

jonkv@ida

Python: *Dynamisk* typkontroll

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

Java: *Statisk* typkontroll

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

Dessutom: Typdeklarationer är *dokumentation*!

Går det att addera?

Kolla värdetyp *vid körning*
Om inte numeriskt: 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*
Om inte numeriskt: Signalera fel

Tidigare upptäckt av problem
➔ effektivare programmering, färre krascher

Mindre typkontroll vid körning
➔ effektivare körning

dynamic type checking
static type checking

Konsekvenser: Val av operation

jonkv@ida

Python: *Dynamisk* typkontroll

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

Måste kolla *vid körning*:

x heltal?
Addera 10 direkt…

x flyttal?
Konvertera 10 till 10.0, addera

…

x annat?
Signalera fel!

Java: *Statisk* typkontroll

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

x är heltal!
Addera direkt

Konsekvenser: Minnesutrymme

jonkv@ida

Python: *Dynamisk* typkontroll

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

Java: *Statisk* typkontroll

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

Hur kolla typen hos b:s värde?

Måste lagra typen med värdet

➔ delvis därför
kan ett heltal ta 24 bytes

Hur kolla typen hos b:s värde?

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

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

Konsekvenser: För utvecklingsmiljöer

jonkv@ida

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 <=
            int randomInt = rand
            log("Generated : " +
        }

        log("Done.");
    }

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

Random (java.util)
SecureRandom (java.security)

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

http://jetbrains.dzone.com/articles/top-20-code-completions-in-intellij-idea

Brasklapp

- En brasklapp:
  - **<u>Terminologin</u>** för typsystem är **<u>ofta otydlig och omtvistad</u>**
  - Många termer brukar blandas ihop
    - Statisk typning
    - Statisk typkontroll
    - Manifest typning
    - Stark typning
    - …
  - **Det viktigaste är *begreppen* och dess *konsekvenser***

Även dynamisk typning (Python) har fördelar!

Mindre att skriva

Mer flexibilitet i vissa fall

Mer Java:
Egenskaper hos primitiva datatyper

Primitiva (grundläggande) typer i Java

jonkv@ida

Heltalstyper – lika på alla plattformar!

		min	max
byte	8 bitar	-128	127
short	16 bitar	-32768	32767
int	32 bitar	-2147483648	2147483647
long	64 bitar	-9223372036854775808L	9223372036854775807L

Används sällan

Vanligast!

"L" indikerar "långt heltal"

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

float	32 bitar	±3.40282347E-45	±3.40282347E+38
double	64 bitar	±4.9406564584124654E-324	±1.797769313486...E+308

Övrigt

boolean	false, true
char	tecken (värden 0..65535)

Stora tal


```
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:
  - **0b**10000000000000000 * **0b**10000000000000000 = **0b**10000000000000000000000000000000
    32 bitar slutresultat
- ➔ 128k * 128k är: 0

Overflow


```
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*. Annars: **BigInteger**
  - Python: Testa storlek, allokera minne för resultat, …

Använd större datatyp

jonkv@ida

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

Beräkningar använder den <u>största</u> av <u>operandernas</u> typer

- 131072 är en *int* (inget "L")
  - **0b**10000000000000000 * **0b**10000000000000000 = **0b**100000000000000000000000000000000
  - **32 bitar slutresultat**
- **<u>Sedan</u>** expanderas detta till 64 bitar
  - 128k * 128k är: 0

Kan verka korkat…
…men är mer förutsägbart:
Resultat beror bara på
operandernas typer

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 <u>största</u> av <u>operandernas</u> typer

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

Expandering
tappar aldrig information
➔ sker automatiskt!

"Farliga" typkonverteringar

jonkv@ida

```
public class JavaTest {
   public static void main(String[ ] args) {
      int sqrtPi = Math.sqrt(Math.PI);
      System.out.println("Sqrt(π) 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!

Typkonvertering: Casting


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

Konvertera med en *cast*

Beräkning, sedan trunkering (avhuggning)

- Beräknar värdet: 1,772453850905516027298167483341 1
- (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)
    - 0b100001001011101010 0

Utan casting

jonkv@ida

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

Byt variabeltyp…

Beräkningar

- Beräknar värdet: 1,772453850905516027298167483341 1
- Skriver ut det

Vad är sant / falskt?

jonkv@ida

Python: Automatisk 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!**

```
int längd = 10;
if (längd) … // Fel!
if (längd != 0) {
   System.out.println(längd);
}
```

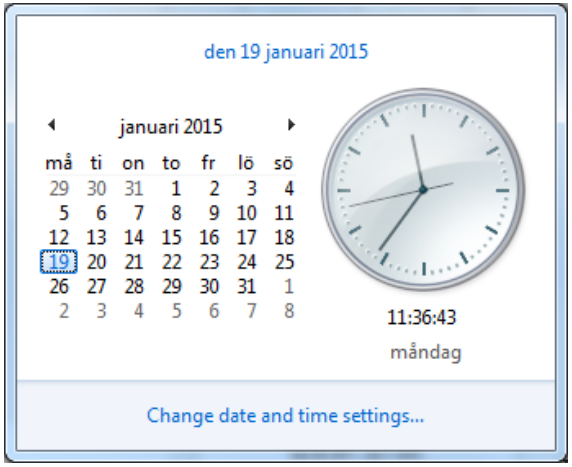
Mer att skriva
Ibland tydligare
Kan förhindra misstag

Våra första egna typer:
Uppräkningstyper

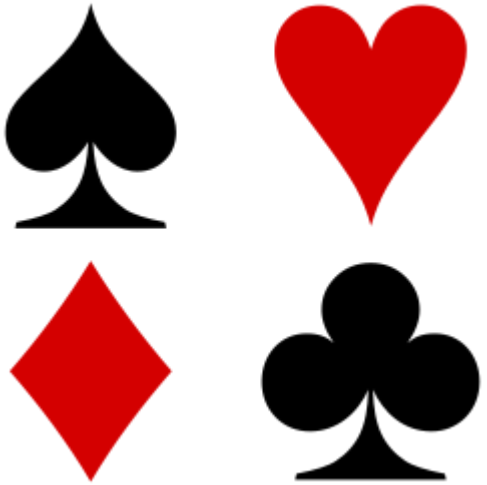
Bakgrund

jonkv@ida

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



  - **Suit**: Clubs, Diamonds, Hearts, Spades



Emulering

jonkv@ida

- Kan **emuleras** på många sätt, t.ex. med **heltalskonstanter**
  - `int MONDAY = 0, TUESDAY = 1, ..., SUNDAY = 6;`
  - `public void setDayOfWeek(int day) { ... }`

- Inte typsäkert!
  - `setDayOfWeek(42);` **// Accepteras av kompilatorn…**
    // Men vi vill ha *tidiga varningar*!
  - `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
  - Värdet är 4 – betyder det torsdag eller fredag?
    Eller kanske grönt, spader eller giraff?

Uppräkningstyper

jonkv@ida

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

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

Sju *enum-konstanter*

Inga nonsens-operationer tillgängliga

Namngivningskonvention: ALL_CAPS för *konstanter*

enum-konstant = enum constant

Uppräkningstyper: Typsäkerhet

jonkv@ida

- Distinkt typ (inte *int*) ➔ typsäkerhet

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

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

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

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

Uppräkningstyper: Index

jonkv@ida

- Vi kan få ut *index* för ett värde

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

```
public void setDayOfWeek(Day day) {
   int index = day.ordinal();
}
setDayOfWeek(Day.TUESDAY);  // index 1 (börjar på 0)
```

Uppräkningstyper

jonkv@ida

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

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

```
Day d = Day.valueOf("TUESDAY");
```

```
for (Day d : Day.values()) {
   System.out.println("En av dagarna är " + d);
}
```

Jämförelse

jonkv@ida

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;
   ...
```