Vecka 1. Introduktion

Programmering, grundkurs (pgk)

Björn Regnell

Datavetenskap, LTH, Lunds universitet
https://cs.lth.se/pgk

EDAA45, Lp1-2, HT2024

Kompilerad den 14 maj 2025

1 Introduktion
- Om kursen
- Att lära denna läsvecka `w01`
- Om programmering
- De enklaste beståndsdelarna: litteraler, uttryck, variabler
- Funktioner
- Logik
- Satser
- Kontrollstrukturer
- Veckans uppgifter

Om kursen

Vem går pgk?

- D1|C1: Nybörjare på Datateknik|Infocom, kursen är obligatorisk men du ska ändå självregistrera dig i Ladok.
- Dx|Cx: Äldre LTH-studenter som är omregistrerade i senare årskurs eller bytt till Datateknik|Infocom

Viktiga länkar

- Kursens **öppna** hemsida: `http://cs.lth.se/pgk`

  Här finns det mesta, t.ex. dessa bilder, annat kursmaterial, hur du installerar verktyg på din egen dator etc.

  Lär dig hitta på hemsidan redan nu.

- Kursens **slutna** sida bakom inloggningsvägg:
  `https://canvas.education.lu.se/courses/31677`

  Här finns administrativ information om efterbeställning av bokpaket, gruppindelning, föreläsningsdeltagarlista, hemlig invit till Discord-server etc.

  Håll koll på ”anslag” i Canvas så du inte missar något.

  Du måste ha studentkonto, läs noga här:
  `https://www.student.lth.se/ny-student/`

Vad lär du dig?

- Grundläggande principer för programmering: Sekvens, Alternativ, Repetition, Abstraktion (SARA)
  $\Longrightarrow$ Inga förkunskaper i programmering krävs!
- Implementation av algoritmer
- Tänka i abstraktioner, dela upp problem i delproblem
- Förståelse för flera olika angreppssätt:
  - **imperativ programmering**
  - **objektorientering**
  - **funktionsprogrammering**
- Det moderna programmeringsspråket **Scala**
- Utvecklingsverktyg (editor, kompilator, utvecklingsmiljö)
- Implementera, granska, testa, felsöka

Progression

Kursens koncept avancerar **steg för steg**:

- Kontrollstrukturer
- Funktioner
- Objekt
- Datastrukturer
- Algoritmer
- Nästlade strukturer
- Avancerade abstraktionsmekanismer
  - Komposition
  - Polymorfism
  - Kontextuella abstraktioner

Vi **itererar** över koncepten & **fördjupar** förståelsen **efter hand**.

Veckoöversikt

W	*Modul*	*Övn*	*Lab*
W01	Introduktion	expressions	kojo
W02	Program och kontrollstrukturer	programs	–
W03	Funktioner och abstraktion	functions	irritext
W04	Objekt och inkapsling	objects	blockmole
W05	Klasser och datamodellering	classes	blockbattle0
W06	Mönster och felhantering	patterns	blockbattle1
W07	Sekvenser och enumerationer	sequences	shuffle
TP	–	–	–
W08	Nästlade och generiska strukturer	matrices	life
W09	Mängder och tabeller	lookup	words
W10	Arv och komposition	inheritance	snake0
W11	Varians och kontextparametrar	context	snake1
W12	Fördjupning, Projekt	extra	Projekt0
W13	Repetition	examprep	Projekt1
W14	MUNTLIGT PROV	Munta	Munta
TP	VALFRI TENTAMEN	–	–

Kursutveckling och förnyelse

- **Scala** är förstaspråk på Datateknik (D) sedan **2016**.

  Den **största förnyelsen** av den inledande programmeringskursen sedan vi införde **Java 1997**.
- **Scala** är förstaspråk på InfoCom (C) sedan **2021**.
- **Scala 3** sedan 2021 med stora förenklingar för nybörjare + nya avancerade koncept för proffsen.
- **Ny examination** från 2021: muntligt prov + valfri tenta
- Kursmaterialet är **öppen källkod** och **fritt** tillgängligt.
- **Studentmedverkan** i kursutvecklingen:
  - Mer än 90 personer har bidragit på github.com/lunduniversity/introprog
- Se alla förbättringar sen förra året här: github.com/lunduniversity/introprog/commits
- Du är hjärtligt välkommen att bidra! Se instruktioner i kompendiet.

Historik förstaspråk på D & C vid LTH

Scala	**2016 (D), 2021 (C)**
Java	1997 (D), 2001 (C), InfoCom-programmet grundas
Simula	1990 (D)
Pascal	1982 (D), Datateknik-programmet grundas
Algol	(F, E, ...) förhistoria med hålkortsprogrammering

Historik förstaspråk på D & C vid LTH

Scala	**2016 (D), 2021 (C)**
Java	1997 (D), 2001 (C), InfoCom-programmet grundas
Simula	1990 (D)
Pascal	1982 (D), Datateknik-programmet grundas
Algol	(F, E, ...) förhistoria med hålkortsprogrammering

Scalas uppfinnare Professor Martin Odersky vid EPFL i Lausanne, Schweiz, har även skrivit stora delar av Java-kompilatorn, och var en gång i tiden doktorand för Prof. Niklaus Wirth, som låg bakom Algol, Pascal, Modula m.m. Första versionen av Scala kom 2004. `https://en.wikipedia.org/wiki/Scala_(programming_language)`

Varför Scala som förstaspråk?

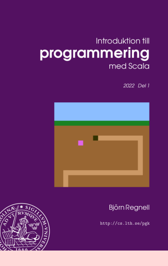
- Varför **Scala**?
  1. Enkel och enhetlig syntax => **lätt att skriva**
  2. Enkel och enhetlig semantik => **lätt att fatta**
  3. Kombinerar flera angreppssätt => **jämföra lösningar**
  4. Stark typning + statisk typning => **färre buggar**
  5. Typhärledning => **koncis kod**
  6. Scala Read-Evaluate-Print-Loop => **lätt att experimentera**
  7. Skalbart från lätt till avancerat => **nybörjare** + **fördjupning**
  8. Scala är **öppen källkod** + massor av fria kodbibliotek[1]
  9. Effektivitet: avancerad, mogen teknik => snabba program
  10. Stor industriell spridning: Netflix, LinkedIn, Twitter, Spotify, PayPal, Klarna, Sony, AirBNB, UBS, The Guardian, ...
  11. Scala och Java fungerar utmärkt tillsammans:
      - Java-bibliotek kan användas direkt i ditt Scala-program, och Java är det mest använda (?) språket på planeten Jorden

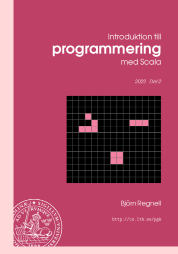
[1] `https://index.scala-lang.org/`

Hur lär du dig?

- Genom praktiskt **eget arbete**: **Lära genom att göra!**
  - Övningar: applicera koncept på olika sätt
  - Laborationer: kombinera flera koncept till en helhet
- Genom studier av kursens teori: **Skapa förståelse!**
- Genom samarbete med dina kurskamrater: **Gå djupare!**

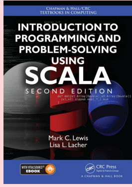
Kurslitteratur



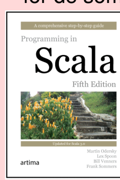


- **Kompendium**
  med övningar & laborationer,
  trycks till självkostnad enl. beställning,
  se info bakom Canvas-väggen.
- Föreläsningsbilder på kurshemsidan
  `http://cs.lth.se/pgk/`
- Nätresurser enl. länkar i kursmaterialet

Bra extra **överkursläsning**:
– för **nybörjare**:



– för de som **redan kan OO**:



Se `https://cs.lth.se/pgk/litteratur/`

Bokpaketet

UTHÄMTNING av beställda men ej uthämtade bokpaket sker på rasten i E:A i kl 14 i morgon tisdag, se info i Canvas.

- Kompendiet och snabbreferens trycks här i E-huset och säljs av institutionen till **självkostnadspris**. Se info i Canvas.
- **Kompendiet** finns i pdf för fri nedladdning enl. licensen CC-BY-SA, men det **rekommenderas starkt** att du även drar nytta av pappersversionen.
- För de flesta funkar det mycket bättre att ha övningar och labbar **på papper bredvid skärmen**, när du ska tänka, koda och plugga!
- **Snabbreferensen** finns också i pdf men du behöver ha en tryckt version eftersom det är **enda tillåtna hjälpmedlet** på skriftliga kontrollskrivningen och tentamen.
  Säljs separat, se info i Canvas ”Efterbeställning snabbreferens”.

Föreläsningsanteckningar

- Föreläsningbilder uppdateras under kursens gång.
- `http://cs.lth.se/pgk/foerelaesningar/`
- Fram till mån kl 13 aktuell vecka är de bilder som ligger ute under **pågående uppdatering** och kan således ändras.
- Latex-koden och ändringshistorik finns här: github.com/lunduniversity/introprog/tree/master/slides
- Kom gärna med förslag på innehåll och tips på förbättringar!

Personal 2024

Kursansvar: Prof. Björn Regnell, bjorn.regnell@cs.lth.se, E:2413

Handledare: **teknologer** anställda som s.k. **amanuenser**, 2024 (34 st) Adam Månsson, Alice Westerberg Zetterlund, André Philipsson Eriksson, Axel Langenskiöld, Axel Nilsson, Dag Hemberg, David Unelind, Edvin Norrman, Elias Åradsson, Erik Lundberg, Emil Sunnerdahl, Erik Heimdal, Esbjörn Stenberg, Hampus Edén, Hjalmar Rutberg, Johan Ekberg, Jon Mellerby, Johannes Nydahl, Karl Sellergren, Lovisa Löfgren, Marina Fridh-Cardoso, Maximilian Uggla, Moa Gassilewski, Naima Khan, Nils Klemming Nordenskiöld, Nils Melén, Noah Andreasson, Oscar Korpi, Rurik Hagman, Sander Holm, Sara Reimers, Tova Stengard, Tristan Farkas, Viggo Bergdahl

Kursadmin: Birger Swahn, rum E:2181, Ulrika Templing, rum E:2179
Karta till expeditionen och mer info här:
`http://cs.lth.se/kontakt/expedition/`

Kursmoment — varför?

- **Föreläsningar**: skapa översikt, ge struktur, förklara teori, svara på frågor, motivera varför.
- **Övningar**: bearbeta teorin steg för steg, **grundövningar** för alla, **extraövningar** om du vill/behöver öva mer, **fördjupningsövningar** om du vill gå djupare; **förberedelse inför laborationerna**.
- **Laborationer**: **obligatoriska**, sätta samman teorins delar i ett större program; lösningar redovisas för handledare; gk på alla för att få tenta.
- **Resurstider**: få hjälp med övningar och laborationsförberedelser av handledare, fråga vad du vill.
- **Samarbetsgrupper**: grupplärande genom samarbete, hjälpa varandra.
- **Kontrollskrivning**: **obligatorisk**, diagnostisk, kamraträttad; kan ge samarbetsbonuspoäng till valfria tentan.
- **Individuell projektuppgift**: **obligatorisk**, du visar att du kan skapa ett större program självständigt; redovisas för handledare.
- **Muntligt prov**: **obligatoriskt**, ska klaras för godkänt på kursen; du visar att du har tillräcklig förståelse för kursens koncept för att klara nästa kurs.
- **Tentamen**: Valfri för överbetyg men alla uppmuntras att försöka; skriftlig, enda hjälpmedel: snabbreferensen.`http://cs.lth.se/pgk/quickref`

Detta är bara början...

Exempel på efterföljande kurser som bygger vidare på denna:

- Programmeringsteknik – fördjupningskurs
- Objektorienterad modellering och design
- Programvaruutveckling i grupp
- Algoritmer, datastrukturer och komplexitet
- Funktionsprogrammering

Förkunskaper

- Förkunskaper $\neq$ Förmåga
- Varken kompetens eller personliga egenskaper är statiska
- ”Programmeringskompetens” är inte *en* enda enkel förmåga utan en komplex sammansättning av flera olika förmågor som **utvecklas** genom hela livet
- Ett innovativt utvecklar**team** behöver många olika kompetenser för att vara framgångsrikt

Stor spridning i förkunskaper

Har du programmerat tidigare?

Ja	130 respondenter	**74 %**	
Nej	45 respondenter	26 %	

Samarbetgrupper

- Ni delas in i **samarbetsgrupper** om `(4 to 6)` personer baserat på förkunskapsenkäten, så att olika förkunskapsnivåer sammanförs.
- En av laborationerna (`snake`) i lp2 är en mer omfattande **grupplabb** och kommer att göras i din samarbetsgrupp.
- Kontrollskrivningen i halvtid kan ge **samarbetsbonus** (max 5p) som adderas till valfria tentans poäng (max 100p) med medelvärdet av gruppmedlemmarnas individuella kontrollskrivningspoäng

  Bonus b för varje person i en grupp med n medlemmar med p_i poäng vardera på kontrollskrivningen:

  $$b = \sum_{i=1}^{n} \frac{p_i}{n}$$

Varför studera i samarbetsgrupper?

Huvudsyfte: **Djupinriktat lärande!**

- Pedagogisk forskning stödjer tesen att lärandet blir mer djupinriktat om det sker i utbyte med andra
- Ett studiesammanhang med **höga ambitioner** och **respektfull gemenskap** gör att **alla lär sig mer**
- Varför ska du som redan kan mycket aktivt dela med dig av dina kunskaper?
  - Förstå bättre själv genom att förklara för andra
  - Träna din pedagogiska förmåga
  - Förbered dig för ditt kommande yrkesliv som mjukvaruutvecklare

Samarbetskontrakt

Gör ett skriftligt **samarbetskontrakt** gärna med dessa och ev. andra punkter som ni också tycker bör ingå:

1. Återkommande mötestider per vecka
2. Kom i tid till gruppmöten
3. Var väl förberedd genom självstudier inför gruppmöten
4. Hjälp varandra att förstå, men ta inte över och lös allt
5. Ha ett respektfullt bemötande även om ni har olika åsikter
6. Inkludera alla i gemenskapen

Diskutera hur ni ska uppfylla dessa innan alla skriver på.
Var och en tar med en egen **pappersutskrift** av samarbetskontraktet där **alla skrivit på** och visar för handledare på första labben.

Om arbetet i samarbetsgruppen inte fungerar ska ni mejla kursansvarig och boka mötestid för konsultation!

Dina frågor är viktiga!

- Det finns bättre och sämre frågor vad gäller hur mycket man kan lära sig av svaret, men **all undran är en chans** att i dialog utbyta erfarenheter och lärande.
- Den som frågar **vill veta** och berättar genom frågan något om nuvarande kunskapsläge.
- Den som svarar får chansen att **reflektera** över vad som kan vara svårt och olika vägar till djupare förståelse.
- I en hälsosam lärandemiljö är det **helt tryggt** att visa att man **ännu inte förstår**, att man gjort ”fel”, att man har mer att lära, etc.
- Det är viktigt att **våga försöka** även om det blir **”fel”**: **det är ju då man lär sig!**

Plagiatregler

- Läs dessa regler noga och diskutera i samarbetsgrupperna:
  - `http://cs.lth.se/utbildning/samarbete-eller-fusk/`
  - Föreskrifter angående obligatoriska moment
- Ni ska lära er genom **eget arbete** och **bra samarbete**.
- Samarbete gör att man lär sig bättre, men man lär sig **inte** av att kopiera andras lösningar.
- **Plagiering är förbjuden** och kan medföra **disciplinärende och avstängning**.
- Du får **INTE** lägga ut laborationslösningar öppet på **`github`** eller på annan plats där någon annan kan komma åt dem!
- Det är **INTE** tillåtet att använda **artificiell intelligens** i arbetet med laborationer och projekt i denna kurs.

En typisk kursvecka

1. Gå på **föreläsningar** på **måndag–tisdag**
2. **Jobba individuellt** med teori, övningar, labbförberedelser på **måndag–torsdag**
3. **Träffas** regelbundet i **samarbetsgruppen** och hjälp varandra att förstå mer och fördjupa lärandet, förslagsvis på återkommande tider varje vecka då alla i gruppen kan
4. Kom till **resurstiderna** och få hjälp och tips av handledare och kurskamrater på **onsdag–torsdag**
5. Genomför den obligatoriska **laborationen** på **fredag**

Se detaljerna och undantagen i schemat: cs.lth.se/pgk/schema

Övningar

- **Programmering lär man sig bäst genom att programmera...**
- Genom veckans övningarna i kompendiet bearbetar du teorins olika delar via egna **undersökningar** med **Scala REPL** som viktigaste verktyg.
- Fokusera på att **förstå** vad som händer när du kör din kod. Om du bara rusar vidare utan reflektion lär du dig inte alls lika bra.
- Till de flesta övningar finns **facit**. Titta inte på facit förrän du själv gjort ett försök. Det finns ofta många olika sätt att åstadkomma samma sak, och din lösningsvariant kan vara lika bra som den i facit – använd REPL för att verifiera att din lösning fungerar. Diskutera med handledare om du är osäker på vad som är en mer eller mindre bra lösning.
- Skapa en miljö för **koncentration** och lärande **på djupet**. Stäng telefon, be kompisar i datorsalen att inte prata för högt, etc.

Laborationer

- På laborationerna **sammanför** du veckans koncept till en **helhet** i ett större program och kollar att du **kan grunderna** inför kommande veckor.
- Labbarna är **individuella** (utom en) och **obligatoriska**.
- Gör **övningarna** och **labbförberedelserna** noga **innan** själva labben – detta är ofta helt nödvändigt för att du ska hinna klart. Dina labbförberedelser kontrolleras av handledare under labben.
- Är du **sjuk?** Anmäl det **före** labben till `bjorn.regnell@cs.lth.se`, få hjälp på resurstid och redovisa på resurstid (eller labbtid, när handledaren har tid över)
- Hinner du inte med hela labben? Se till att handledaren noterar **”kompletteras”**, och fortsätt på resurstid och ev. uppsamlingstider.
- Läs noga kapitel ”**Anvisningar**” i kompendiet!
- Laborationstiderna är gruppindelade enligt schemat. Du ska gå till den tid och den sal som motsvarar din grupp som visas i TimeEdit.
- **Du hittar vilken grupp du tillhör i Canvas.**

Ge din **hjärna** rätt förutsättningar för programmering

Att lära denna läsvecka `w01`

Att lära denna läsvecka w01

Modul **Introduktion**: Övn **expressions** → Labb **kojo**

- ☐ sekvens
- ☐ alternativ
- ☐ repetition
- ☐ abstraktion
- ☐ editera
- ☐ kompilera
- ☐ exekvera
- ☐ datorns delar
- ☐ virtuell maskin
- ☐ litteral
- ☐ värde
- ☐ uttryck
- ☐ identifierare
- ☐ variabel
- ☐ typ
- ☐ tilldelning
- ☐ namn
- ☐ val
- ☐ var
- ☐ def
- ☐ definiera och anropa funktion
- ☐ funktionshuvud
- ☐ funktionskropp
- ☐ procedur
- ☐ inbyggda grundtyper
- ☐ println
- ☐ typen Unit
- ☐ enhetsvärdet ()
- ☐ stränginterpolatorn s
- ☐ aritmetik
- ☐ slumptal
- ☐ logiska uttryck
- ☐ de Morgans lagar
- ☐ if
- ☐ true
- ☐ false
- ☐ while
- ☐ for
- ☐ dod: operativsystem

Om programmering

Att skapa koden som styr världen

I stort sett **alla** delar av samhället är beroende av programkod:

- kommunikation
- transport
- byggsektorn
- statsförvaltning
- finanssektorn
- media & underhållning
- sjukvård
- övervakning
- integritet
- upphovsrätt
- miljö & energi
- sociala relationer
- utbildning
- ...

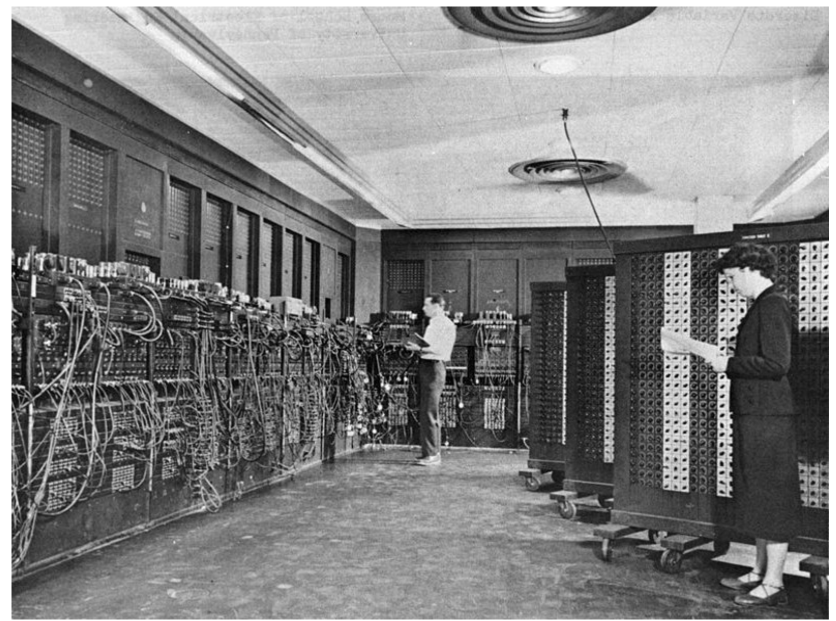
Hur blir ditt framtida yrkesliv som systemutvecklare?

- Det är sedan lång tid en **skriande brist** på utvecklare och det blir bara värre...
  CIO 2018-11-09
- Störst brist är det på **kvinnliga** utvecklare:
  SVT 2016-12-03
- Global kompetensmarknad
  CIO 2016-02-01
  CS 2015-06-14
  CS 2016-07-14

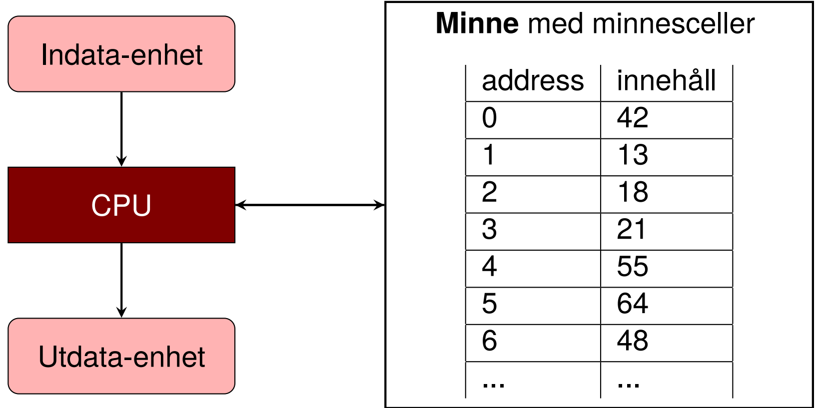
Utveckling av mjukvara i praktiken

- **Inte bara kodning:** kravbeslut, releaseplanering, design, test, versionshantering, kontinuerlig integration, driftsättning, återkoppling från dagens användare, ekonomi & investering, gissa om morgondagens användare, ...
- **Teamwork:** Inte ensamma hjältar utan autonoma team i decentraliserade organisationer med innovationsuppdrag
- **Snabbhet:** Att koda innebär att hela tiden uppfinna nya ”byggstenar” som ökar organisationens förmåga att snabbt skapa värde med hjälp av mjukvara. **Öppen källkod**. Skapa kraftfulla API:er. (API=application programming interface, byggstenar för att bygga appar)
- **Livslångt lärande:** Lär nytt och dela med dig hela tiden. Exempel på pedagogisk utmaning: hjälp andra förstå och använda ditt API $\Longrightarrow$ **Samarbetskultur**

Vad är en dator?



Hur fungerar en dator?



address	innehåll
0	42
1	13
2	18
3	21
4	55
5	64
6	48
...	...

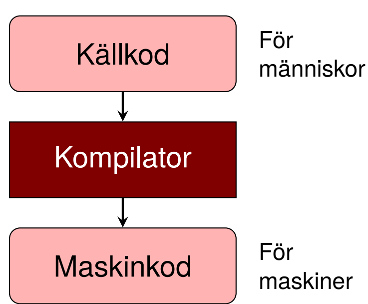
Minnet innehåller endast **heltal** som representerar **data och instruktioner**.

Vad är programmering?

- Programmering innebär att ge instruktioner till en maskin.
- Ett **programmeringsspråk** används av människor för att skriva **källkod** som kan översättas av en **kompilator** till **maskinspråk** som i sin tur **exekveras** av en dator.
- Ada Lovelace publicerade det första programmet redan på 1800-talet ämnat för en kugghjulsdator.

- sv.wikipedia.org/wiki/Programmering
- en.wikipedia.org/wiki/Computer_programming
- Ha picknick i Ada Lovelace-parken på Brunnshög!

Vad är en kompilator?

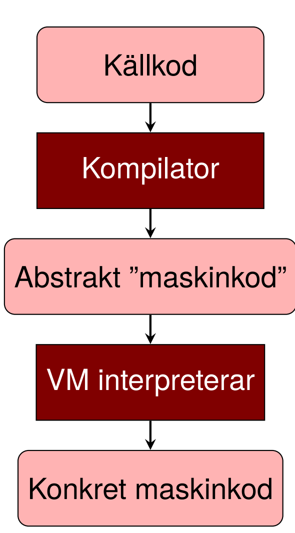


Grace Hopper uppfann kompilatorn 1952.

en.wikipedia.org/wiki/Grace_Hopper

Virtuell maskin (VM) == abstrakt hårdvara

- En VM är en ”dator” implementerad i mjukvara som kan tolka en abstrakt ”maskinkod” som **översätts under körning** till den **verkliga** maskinens konkreta maskinkod.
- Med en VM blir källkoden **plattformsoberoende** och fungerar på många olika maskiner.
- Exempel JVM: **Java Virtual Machine**



Vad består ett program av?

- Text som följer entydiga språkregler (grammatik):
  - **Syntax**: textens konkreta utseende
  - **Semantik**: textens betydelse (vad maskinen gör/beräknar)
- **Nyckelord**: ord med speciell betydelse, t.ex. `if`, `while`
- **Deklarationer**: definitioner av nya ord: `def gurka = 42`
- **Satser** är instruktioner som **gör** något: `print("hej")`
- **Uttryck** är instruktioner som beräknar ett **resultat**: `1 + 1`
- **Data** är information som behandlas: t.ex. heltalet 42
- Instruktioner ordnas i kodstrukturer: **SARA**
  - **Sekvens**: ordningen spelar roll för vad som händer
  - **Alternativ**: olika resultat beroende på uttrycks värde
  - **Repetition**: instruktioner upprepas många gånger
  - **Abstraktion**: nya byggblock skapas för att återanvändas

Exempel på programmeringsspråk

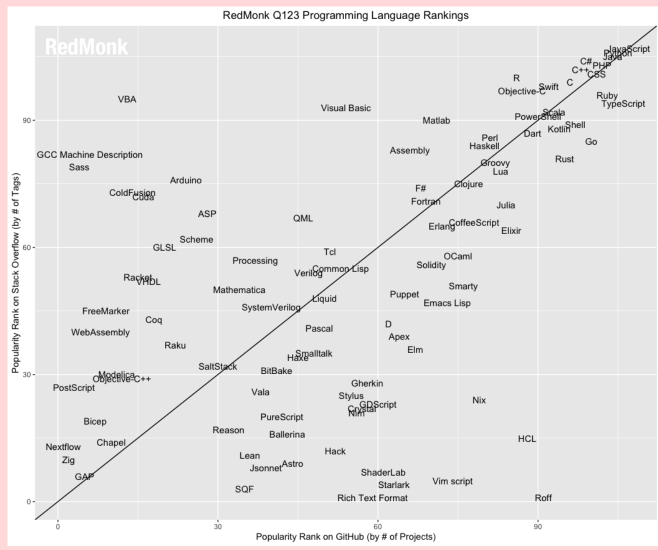
Det finns massor med olika språk och det kommer ständigt nya.

- Java
- C
- C++
- C#
- Python
- JavaScript
- Scala

Några topplistor:

- Redmonk
- PYPL
- TIOBE

Redmonk Language Rankings: Github, Stackoverflow



Olika programmeringsparadigm

- Det finns många olika programmeringsparadigm (sätt att programmera på), till exempel:
  - **imperativ programmering:** programmet är uppbyggt av satser som påverkar systemets tillstånd
  - **objektorienterad programmering:** en sorts imperativ programmering där programmet består av objekt som kapslar in data och erbjuder operationer som bearbetar dessa data
  - **funktionsprogrammering:** programmet är uppbyggt av samverkande funktioner som undviker förändringar av data
  - **deklarativ programmering, logikprogrammering:** programmet är uppbyggt av logiska uttryck som beskriver olika fakta eller villkor och exekveringen utgörs av en bevisprocedur som söker efter värden som uppfyller fakta och villkor

Denna kurs behandlar de tre första.

Hello world

Kör rad för rad i Scala REPL (Read-Evaluate-Print-Loop):

```
> scala repl
Welcome to Scala 3.3.0 (17.0.8, Java OpenJDK 64-Bit Server VM).
Type in expressions for evaluation. Or try :help.

scala> println("Hello World!")
Hello World!
```

Hello world

Kör rad för rad i Scala REPL (Read-Evaluate-Print-Loop):

```
> scala repl
Welcome to Scala 3.3.0 (17.0.8, Java OpenJDK 64-Bit Server VM).
Type in expressions for evaluation. Or try :help.

scala> println("Hello World!")
Hello World!
```

`@main` framför valfri funktion anger var ett fristående program ska starta:

```
@main def hi = println("Hello world!")
```

Spara texten ovan i filen `hello.scala` och kompilera ditt program:

```
> scala compile hello.scala
```

Kör ditt program med `scala run` som kompilerar automatiskt vid behov.

```
> scala run hello.scala
Hello World!
```

Utvecklingscykeln

editera; kompilera; hitta fel och förbättringar; editera; kompilera; hitta fel och förbättringar; editera; kompilera; hitta fel och förbättringar; editera; kompilera; hitta fel och förbättringar; editera; kompilera; hitta fel och förbättringar; editera; kompilera; hitta fel och förbättringar; ...

```
upprepa(1000){
  editera
  kompilera
  testa
}
```

Utvecklingsverktyg

- Din verktygskunskap är mycket viktig för din produktivitet.
- Lär dig kortkommandon för vanliga handgrepp.
- Alla verktyg som behövs finns förinstallerade på **LTH:s linuxdatorer**.
  Om din egen burk krånglar: kör på skolans burkar så du ej fördröjs!
- Verktyg vi använder i kursen:
  - Scala **REPL**: från övn 1
  - Barnvänlig Scala-programmering med Kojo: Lab 1
  - **Texteditor** för kod, t.ex **VS code**: från övn 2
  - Kompilera och kör fristående program med `scala`: från övn 2
- Andra verktyg som är bra att lära sig:
  (ingår i EDAA60 Datorer och datoranvändning)
  - Git för versionshantering
  - GitHub för kodlagring – men **inte** av lösningar till labbar!
  - Linux/Ubuntu och nyttiga terminalkommando

Installera verktyg på din egen dator

När du ska skriva kod i en editor, kompilera i terminalen och köra ditt program som en **fristående applikation**, så behövs:

- En editor: **VS Code** med tillägget **Scala (Metals)**
- Körmiljön **OpenJDK**
- Kommandoverktyg för terminalen: **`scala`**
- Se instruktioner här: `http://cs.lth.se/pgk/verktyg`
- Läs mer i Appendix C.
- Tips om du kör Windows: installera nya Windows Terminal
- Installationshjälp:
  1. Drop-in: kl 12-13,
     4/9 E:3336, 5/9 E:3336, 6/9 E:2116, 9/9 E:2116,
     10/9 E:3336, 11/9 E:2116, 12/9 E:2116, 13/9 E:2116
  2. Pluggkvällar som SRD ordnar.
  3. `#frågor-och-svar` på vår Discord-server
  4. Fråga handledare på resurstid (i mån av tid).

Scala Command Line Interface (CLI)

- Utvecklingen av ett nytt kommandogränssnitt (eng. *Command Line Interface (CLI)*) för Scala startades 2022 i ett öppenkällkodsprojekt som leds av Virtuslab.
- I augusti 2024 blev **`scala-cli`** det nya **`scala`**-kommandot.[2]
- Läs mer i Appendix C och F, samt här: `https://scala-cli.virtuslab.org/`
- Du kan se vad Scala CLI kan göra via hjälp-optionen:

```
> scala help
```

`scala-cli help` är ”gamla” kommandot som också ingår i Scala-installationen.

[2]När kompendiet trycktes hade skiftet ännu inte skett. Nu kan du ersätta alla förekomster av `scala-cli` med det kortare `scala`.

Tips och trix med `scala` i terminalen

- Skriv `:help` i REPL så får du se vilka **kommando** som finns.
- Du kan **avsluta** REPL med `:q` eller trycka Ctrl+D.
- Ett **vertikalstreck visas** om du trycker ENTER mitt i en ofullständigt rad. Detta indikerar att du kan fortsätta skriva på ny rad innan tolkning sker.
- Om du vill att REPL ska vänta att tolka raden du skrivit och istället ge dig **ännu en rad**, så tryck först ner ESC-tangenten och sedan ENTER.
- Om du vill förhindra att REPL ger ny rad efter ENTER vid ofullständig rad, så skriv ett **semikolon** och tryck ENTER.
- Starta `repl` med punkt efter blanktecken om du vill ha tillgång till koden i alla scala-filer i **aktuell katalog** i din REPL-session:
  `scala repl .`
- Kör med punkt efter blanktecken så kompileras och exekveras alla scala-filer i **aktuell katalog och** eventuella **underkataloger**:
  `scala run .`

De enklaste beståndsdelarna: litteraler, uttryck, variabler

Litteraler

- En litteral representerar ett fixt **värde** i koden och används för att skapa **data** som programmet ska bearbeta.
- Exempel:

`42`	heltalslitteral
`42.0`	decimaltalslitteral
`'!'`	teckenlitteral, omgärdas med ’enkelfnuttar’
`"hej"`	stränglitteral, omgärdas med ”dubbelfnuttar”
`true`	litteral för sanningsvärdet ”sant”

- Literaler har en **typ** som avgör vad man kan göra med dem.

Exempel på inbyggda datatyper i Scala

- Alla värden, uttryck och variabler har en **datatyp**, t.ex.:
  - `Int` för heltal
  - `Long` för *extra* stora heltal (tar mer minne)
  - `Double` för decimaltal, så kallade flyttal med flytande decimalpunkt
  - `String` för strängar
- Kompilatorn håller reda på att uttryck kombineras på ett **typsäkert** sätt. Annars blir det **kompileringsfel**.
- Scala och Java är s.k. **statiskt typade** språk, vilket innebär att kontroll av typinformation sker vid kompilering (eng. *compile time*)[3].
- Scala-kompilatorn gör **typhärledning**: man **slipper skriva typerna** om kompilatorn kan lista ut dem med hjälp av typerna hos deluttrycken.

---

[3] Andra språk, t.ex. Python och Javascript är **dynamiskt typade** och där skjuts typkontrollen upp till körningsdags (eng. *run time*)
Vilka är för- och nackdelarna med statisk vs. dynamisk typning?

Grundtyper i Scala

Dessa **grundtyper** (eng. *basic types*) finns inbyggda i Scala:

Svenskt namn	*Engelskt namn*	**Grundtyper**
heltalstyp	integral type	`Byte`, `Short`, `Int`, `Long`, `Char`
flyttalstyp	floating point number types	`Float`, `Double`
numeriska typer	numeric types	heltalstyper och flyttalstyper
strängtyp (teckensekvens)	string type	`String`
sanningsvärdestyp (boolesk typ)	truth value type	`Boolean`

Grundtypernas omfång

Grundtyp	Antal bitar	Omfång: minsta & största värde
`Byte`	8	$-2^7 \dots 2^7 - 1$
`Short`	16	$-2^{15} \dots 2^{15} - 1$
`Char`	16	$0 \dots 2^{16} - 1$
`Int`	32	$-2^{31} \dots 2^{31} - 1$
`Long`	64	$-2^{63} \dots 2^{63} - 1$
`Float`	32	$\pm 3.4028235 \cdot 10^{38}$
`Double`	64	$\pm 1.7976931348623157 \cdot 10^{308}$

Grundtypen `String` lagras som en *sekvens* av 16-bitars tecken av typen `Char` och kan vara av godtycklig längd (tills minnet tar slut).

Uttryck

- Ett **uttryck** består av en eller flera delar som efter **evaluering** ger ett **resultat**.
- Delar i ett uttryck kan t.ex. vara:
  litteraler (42), operatorer (+), funktioner (sin), ...
- Exempel:
  - Ett enkelt uttryck:
    ```
    42.0
    ```
  - Sammansatta uttryck:
    ```
    40 + 2
    (20 + 1) * 2
    sin(0.5 * Pi)
    "hej" + " på " + "dej"
    ```
- När programmet tolkas sker **evaluering** av uttrycket, vilket ger ett resultat i form av ett **värde** som har en **typ**.

Variabler

- En **variabel** kan tilldelas värdet av ett enkelt eller sammansatt uttryck.
- En variabel har ett **variabelnamn**, vars utformning följer språkets regler för s.k. **identifierare**.
- En ny variabel införs i en **variabeldeklaration** och då den kan ges ett värde, **initialiseras**. Namnet användas som **referens** till värdet.
- Exempel på variabeldeklarationer i Scala, notera **nyckelordet** **`val`**:

```
val a = 0.5 * Pi
val length = 42 * sin(a)
val exclamationMarks = "!!!"
val greetingSwedish = "Hej på dej" + exclamationMarks
```

- Vid exekveringen av programmet lagras variablernas värden i minnet och deras respektive värde hämtas ur minnet när de **refereras**.
- Variabler som deklareras med **`val`** kan endast tilldelas ett värde **en enda gång**, vid den initialisering som sker vid deklarationen.

Regler för identifierare

- **Enkel** identifierare: t.ex. `gurka2Tomat`
  - Börja med bokstav
  - ...följt av bokstäver eller siffror
  - Kan även innehålla understreck
- **Operator**-identifierare, t.ex. `+:`
  - Börjar med ett **operatortecken**, t.ex. `+ - * / : ? ~ #`
  - Kan följas av fler operatortecken
- En identifierare får **inte** vara ett **reserverat ord**, se snabbreferensen för alla reserverade ord i Scala.
- **Bokstavlig** identifierare: ``` `kan innehålla allt` ```
  - Börjar och slutar med **backticks** ``` ` ` ```
  - Kan innehålla vad som helst (utom backticks)
  - Kan användas för att undvika krockar med reserverade ord: ``` `val` ```

Att bygga strängar: konkatenering och interpolering

- Man kan **konkatenera** strängar med operatorn +
  `"hej" + " på " + "dej"`
- Efter en sträng kan man konkatenera vilka uttryck som helst; uttryck inom parentes evalueras först och värdet görs sen om till en sträng före konkateneringen:

```
val x = 42
val msg = "Dubbla värdet av " + x + " är " + (x * 2) + "."
```

- Man kan i Scala få hjälp av kompilatorn att övervaka bygget av strängar med **stränginterpolatorn s**:

```
val msg = s"Dubbla värdet av $x är ${x * 2}."
```

Heltalsaritmetik

- De fyra räknesätten skrivs som i matematiken (vanlig precedens):

```
scala> 3 + 5 * 2 - 1
res0: Int = 12
```

- **Parenteser** styr **evalueringsordningen**:

```
scala> (3 + 5) * (2 - 1)
val res1: Int = 8
```

- **Heltalsdivision** sker med **decimaler avkortade**:

```
scala> 41 / 2
val res2: Int = 20
```

- **Moduloräkning** med restoperatorn `%`

```
scala> 41 % 2
val res3: Int = 1
```

Flyttalsaritmetik

- Decimaltal representeras med s.k. **flyttal** av typen `Double`:

```
scala> math.Pi
val res4: Double = 3.141592653589793
```

- Stora tal så som $\pi * 10^{12}$ skrivs:

```
scala> math.Pi * 1E12
val res5: Double = 3.141592653589793E12
```

- Det finns **inte** oändligt antal decimaler vilket ger problem med **avvrundingsfel**:

```
scala> 0.1 + 0.2
val res6: Double = 0.30000000000000004

scala> 1E10 + 0.0000000000001
val res7: Double = 1.0E10

scala> BigDecimal("0.1") + BigDecimal("0.2")  // BigDecimal funkar
val res8: BigDecimal = 0.3
```

Läs mer här: `https://0.30000000000000004.com`

Funktioner

Definiera namn på uttryck

- Med nyckelordet **def** kan man låta ett **namn** betyda samma sak som ett **uttryck**.
- Exempel:

```
def gurklängd = 42 + x
```

- Uttrycket till höger evalueras **varje** gång **anrop** sker, d.v.s. varje gång namnet används på annat ställe i koden.

```
gurklängd
```

Funktion, argument, parameter

- En **funktion** räknar ut **resultat** baserat på indata som kallas **argument**.
- Argument ges namn genom deklaration av **parametrar**.
- Exempel på deklaration av en funktion med en parameter:

```scala
def dubblera(x: Int) = 2 * x
```

- Parametrarnas typ **måste** beskrivas efter **kolon**.
- Kompilatorn kan härleda **returtypen**, men den kan också med fördel, för tydlighetens skull, anges **explicit**:

```scala
def dubblera(x: Int): Int = 2 * x
```

- Observera att namnet x blir ett "nytt fräscht" **lokalt namn** som **bara finns och syns "inuti" funktionen** och har inget med ev. andra x utanför funktionen att göra.
- Beräkningen sker först vid **anrop** av funktionen:

```
1  scala> dubblera(42)
2  res1: Int = 84
```

Färdiga matte-funktioner i paketet `scala.math`

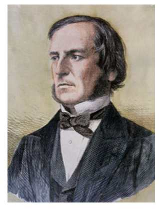
- I paketet **`scala.math`** finns många användbara funktioner: t.ex. `math.random()` ger slumptal mellan `0.0` och `0.9999999999999999`

```
scala> val x = math.random()
x: Double = 0.27749191749889635

scala> val length = 42.0 * math.sin(math.Pi / 3.0)
length: Double = 36.373066958946424
```

- Studera dokumentationen här:
  `https://www.scala-lang.org/api/current/scala/math.html#`
- Paketet `scala.math` delegerar ofta till Java-klassen **`java.lang.Math`** som är dokumenterad här:
  `https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/Math.html`

Logik

Logiska uttryck



- Datorn kan ”räkna” med sanning och falskhet: s.k. booelsk algebra efter George Boole
- Enkla logiska uttryck: (finns bara två stycken)

```
true
false
```

- Sammansatta logiska uttryck med logiska operatorer:
  && och, || eller, ! icke, == likhet, != olikhet, relationer: > < >= <=
- Exempel:

```
true && true
false || true
!false
42 == 43
42 != 43
(42 >= 43) || (1 + 1 == 2)
```

De Morgans lagar

De Morgans lagar beskriver vad som händer om man **negerar** ett logiskt uttryck. Kan användas för att göra **förenklingar**.

- I alla deluttryck sammanbundna med && eller ||, ändra alla && till || och omvänt.
- Negera alla ingående deluttryck. En relation negeras genom att man byter == mot !=, < mot >=, etc.

Exempel på förenkling där de Morgans lagar används upprepat:

```
! (a < b || (a == 1 && b == 1))              ⟺
! (a < b) && ! (a == 1 && b == 1)            ⟺
! (a < b) && (! (a == 1) || ! (b == 1))      ⟺
a >= b && (a != 1 || b != 1)
```

Alternativ med if-uttryck

- Ett if-uttryck börjar med nyckelordet **if**, följt av ett logiskt uttryck (villkor) inom parentes och två grenar.

```
def slumpgrönsak = if math.random() < 0.8 then "gurka" else "tomat"
```

- Uttrycket efter **then** blir resultatet om villkoret är **true**
- Uttrycket efter **else** blir resultatet om villkoret är **false**

```
scala> slumpgrönsak
res13: String = gurka

scala> slumpgrönsak
res14: String = gurka

scala> slumpgrönsak
res15: String = tomat
```

Satser

Uttryck eller sats?

Skillnad mellan uttryck och sats:

- Ett uttryck ger ett **resultat**. Exempel: 1+1
- En sats har en **effekt**.
  Exempel: utskrift, spara på fil, tilldela variabel nytt värde.

Skriv ett **uttryck** när du är intresserad av **värdet** som beräknas.
Skriv en **sats** när du vill att något ska **göras**.

Både satser och uttryck kan i sin tur innehålla satser och uttryck i godtyckligt komplexa **nästlade strukturer** (mer om det senare).

Variabeldeklaration och tilldelningssats

- En **variabeldeklaration** medför att **plats i datorns minne** reserveras så att värden av den typ som variabeln kan referera till får plats där.
- Vid deklaration ska variabeln **initialiseras** med ett startvärde.
- En **`val`**-deklaration ger en variabel som efter initialisering inte kan ändras.

Dessa deklarationer...

```
var x = 42
val y = x + 1
```

... ger detta innehåll någonstans i minnet:

x	42
y	43

- Med en **tilldelningssats** ges en tidigare **`var`**-deklarerad variabel ett nytt värde:

```
x = 13
```

- Det gamla värdet försvinner för alltid och det nya värdet lagras istället:

x	13
y	43

Observera att y här inte påverkas av att x ändrade värde.

Tilldelningssatser är *inte* matematisk likhet

- Likhetstecknet används alltså för att **tilldela** variabler nya värden och det är **inte** samma sak som matematisk likhet. Vad händer här?

```
x = x + 1
```

- Denna syntax är ett arv från de gamla språken C, Fortran mfl.
- I andra språk används t.ex.

  `x := x + 1` eller `x <- x + 1`

- Denna syntax visar kanske bättre att tilldelning är en **stegvis process**:
  1. Först beräknas **uttrycket till höger** om tilldelningstecknet.
  2. Sedan **ersätts värdet** som variabelnamnet refererar till av det beräknade uttrycket. Det gamla värdet **försvinner för alltid**.

Förkortade tilldelningssatser

- Det är vanligt att man vill tilldela en variabel ett nytt värde som beror av det gamla, så som i
  `x = x + 1`
- Därför finns **förkortade tilldelningssatser** som gör så att man sparar några tecken och det blir tydligare (?) vad som sker (när man vant sig vid detta skrivsätt):

```
x += 1
```

- Uttrycket ovan expanderas av kompilatorn till `x = x + 1`

Exempel på förkortade tilldelningssatser

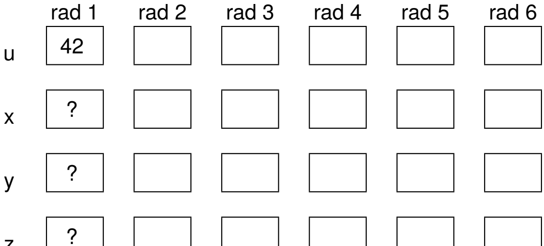
```
scala> var x = 42
val x: Int = 42

scala> x *= 2

scala> x
val res0: Int = 84


scala> x /= 3

scala> x
val res1: Int = 28
```

Övning: Tilldelningar i sekvens

Rita hur minnet ser ut efter varje rad nedan:

```
1  var u = 42
2  var x = 10
3  var y = 2 * x + 1
4  x = 20
5  var z = y + x + y - x
6  x += 1; y *= 2
```

En variabel som ännu inte **initierats** har ett **odefinierat** värde, anges nedan med frågetecken.



Variabler som ändrar värden kan vara knepiga

- Kod som innehåller variabler som **förändras** över tid är ofta svårare att läsa och begripa.
- Många buggar beror på att variabler av misstag förändras på felaktiga och oanade sätt.
- Föränderliga värden blir speciellt svåra i kod som körs jämlöpande (parallellt).
- I kod som körs i skarpt läge med många användare (s.k. produktionskod) är därför **`val`** att föredra, medan **`var`** endast används om det **verkligen** behövs.
- Alltså: räkna hellre ut nya värden än förändra befintliga.

Kontrollstrukturer

Kontrollstrukturer: alternativ och repetition

Används för att kontrollera (förändra) sekvensen och skapa **alternativa** vägar genom koden. Vägen bestäms vid körtid.

- if-sats:

```
if math.random() < 0.8 then println("gurka") else println("tomat")
```

Olika sorters **loopar** för att repetera satser. Antalet repetitioner ges vid körtid.

- `while`-sats: bra när man **inte vet hur många gånger** det kan bli.

```
while math.random() < 0.8 do println("gurka")
```

- `for`-sats: bra när man **vill ange antalet repetitioner**:

```
for i <- 1 to 10 do println(s"gurka nr $i")
```

Scala-2-syntax för kontrollstrukturer fungerar i Scala 3

I Scala 2 användes en gammal syntax för kontrollstrukturer som liknar mer C/C++/Java. Den är tillåten i Scala 3, men nya mer lättlästa syntaxen är att föredra.

- Scala-2-syntax för alternativ: parenteser men inget **then**

```
if (math.random() < 0.8) println("gurka") else println("tomat")
```

Scala-2-syntax för repetition:

- **while**-sats: parenteser men inget **do**

```
while (math.random() < 0.8) println("gurka")
```

- **for**-sats: parenteser men inget **do**

```
for (i <- 1 to 10) println(s"gurka nr $i")
```

- Kojo Desktop funkar ännu bara med Scala 2 och gamla syntaxen, men Kojo kan även köras med Scala 3 (se hur i kompendiet).

Repetera många satser

Om du vill göra flera saker i sekvens inne i en repetition så kan du skriva flera satser inom **klammer-parenteser**:

```
while math.random() < 0.8 do {
  println("gurka")
  println("tomat")
}
println("Repetitionen är klar!")
```

Repetera många satser

Om du vill göra flera saker i sekvens inne i en repetition så kan du skriva flera satser inom **klammer-parenteser**:

```
while math.random() < 0.8 do {
  println("gurka")
  println("tomat")
}
println("Repetitionen är klar!")
```

Du kan efter vissa nyckelord (t.ex. **`do`**, **`then`**, **`else`**) välja bort klammer-parenteser (eng. *optional braces*).

```
while math.random() < 0.8 do
  println("gurka")
  println("tomat")

println("Repetitionen är klar!")
```

Då är det **indenteringen** som avgör vilka satser som ingår. Detta fungerar i Scala 3 (men inte i Scala 2).

Procedurer

- En **procedur** är en funktion som **gör** något intressant, men som **inte** lämnar något intressant returvärde.
- Exempel på procedur i standardbiblioteket: `println("hej")`
- Du **deklarerar egna procedurer** genom att ange **`Unit`** som returvärdestyp. Då returneras värdet **`()`** som betyder ”inget”.

```
scala> def hej(x: String): Unit = println(s"Hej på dej $x!")
hej: (x: String)Unit

scala> hej("Herr Gurka")
Hej på dej Herr Gurka!

scala> val x = hej("Fru Tomat")
Hej på dej Fru Tomat!
x: Unit = ()
```

- Det som **görs** kallas (sido)**effekt**. Ovan är utskriften själva effekten.
- Även funktioner kan ha sidoeffekter. De kallas då **oäkta** funktioner.

Problemlösning: nedbrytning i abstraktioner som sen kombineras

- En av de allra viktigaste principerna inom programmering är **funktionell nedbrytning** där **underprogram** i form av funktioner och procedurer skapas för att bli byggstenar som kombineras till mer avancerade funktioner och procedurer.
- Genom de namn som definieras skapas **återanvändbara abstraktioner** som kapslar in det funktionen gör till ett ”byggblock”.
- Bra ”byggblock” gör det lättare att lösa svåra programmeringsproblem.
- Abstraktioner som beräknar eller gör **en enda, väldefinierad sak** är enklare att använda, jämfört med de som gör många, helt olika saker.
- Abstraktioner med **välgenomtänkta namn** är enklare att använda, jämfört med kryptiska eller missvisande namn.

Veckans uppgifter

Övning `expressions` och labb `kojo`

- På övningen kör du Scala REPL för att träna på SARA. **Läs i Appendix** och på kursens hemsida under ”Verktyg” om hur du installerar och får igång Scala REPL.
- På laborationen använder du barnvänliga **Kojo** för träna på SARA, med fokus på abstraktion.
- Det finns två olika sätt att använda Kojo:
  1. Grafikbiblioteket **`kojolib`** i ett fristående Scala program med hjälp av en professionell kodeditor och kompilering och exekvering i terminalen. **Fungerar fint med nya Scala 3**.
  2. Skrivbordsappen **Kojo Desktop** med inbyggd barnvänlig editor (endast Scala 2, gammal syntax etc).
  3. Webbappen **`http://kojo.lu.se/`** direkt i webbläsare; rekommenderas ej – endast Scala 2, mer begränsad.

Alternativ 1 rekommenderas, men om du försenas av tekniskt strul, så kom igång med 2 så länge tills du fått hjälp.

Köa med Sigrid

För att köa till handledare på plats i sal i pgk använd Sigrid.

(Se hemlig länk i Canvas, sprid ej länken på internet så vi slipper bottar).

- Direkt när undervisningspasset **börjar**: starta en session med ditt förnamn, kurskod EDAA45 och rummets namn. Gör detta även om du inte behöver hjälp från start! Då kan **ambulanser** se antal studenter i varje rum.
- Inget lösenord behövs.
- Två olika köer i varje rum: **hjälpkö** och **redovisningskö**
  - Ställ dig i hjälpkö om du vill få vägledning och ställa frågor
  - Ställ dig i redovisningskö om du är klar att redovisa en labb
- Du måste klicka på **Uppdatera** – annars händer inget!
- OBS! **Köar inte**+**Uppdatera** så fort handledare anländer!
- Om du går på extra pass i mån av plats så kan du se vilket rum som har kortast kö använd Sigrid Monitor.

Sigrid in action

Så här ser det ut när student står i hjälpkö efter att först ha klickat på **Hjäälp!!!** *och* sedan på **Uppdatera**-knappen:

GLÖM INTE **Köar inte** + **Uppdatera** när handledare *anländer*!

Om veckans övning: `expressions`

- Förstå vad som händer när satser exekveras och uttryck evalueras.
- Förstå sekvens, alternativ och repetition.
- Känna till litteralerna för enkla värden, deras typer och omfång.
- Kunna deklarera och använda variabler och tilldelning, samt kunna rita bilder av minnessituationen då variablers värden förändras.
- Förstå skillnaden mellan olika numeriska typer, kunna omvandla mellan dessa och vara medveten om noggrannhetsproblem som kan uppstå.
- Förstå booleska uttryck och värdena **`true`** och **`false`**, samt kunna förenkla booleska uttryck.
- Förstå skillnaden mellan heltalsdivision och flyttalsdivision, samt användning av rest vid heltalsdivision.
- Förstå precedensregler och användning av parenteser i uttryck.
- Kunna använda **`if`**-satser och **`if`**-uttryck.
- Kunna använda **`for`**-satser och **`while`**-satser.
- Kunna använda `math.random()` för att generera slumptal i olika intervaller.
- Kunna beskriva skillnader och likheter mellan en procedur och en funktion.

Om veckans labb: `kojo`

- Kunna tillämpa och kombinera principerna sekvens, alternativ, repetition, och abstraktion i skapandet av egna program om minst 20 rader kod.
- Kunna förklara vad ett program gör i termer av sekvens, alternativ, repetition, och abstraktion.
- Kunna formatera egna program så att de blir lätta att läsa och förstå.
- Kunna förklara vad en variabel är och kunna deklarera oföränderliga och förändringsbara variabler, samt göra tilldelningar.
- Kunna genomföra upprepade varv i cykeln *editera-exekvera-felsöka/förbättra* för att stegvis bygga upp allt mer utvecklade program.