Vecka 11. Varians och kontextparametrar

Programmering, grundkurs (pgk)

Björn Regnell

Datavetenskap, LTH, Lunds universitet
https://lunduniversity.github.io/pgk

EDAA45, Lp1-2, HT2025

Kompilerad den 22 augusti 2025

11 Varians och kontextparametrar

- Repetition: typparametrar
- Generiska typgränser
- Varians: flexibla typparametrar
- Vad är ett bra api?
- Kontextuella abstraktioner
- Översikt av kursens avslutning

Repetition: typparametrar

Typparameter, generisk struktur, typkonstruktor

- Med hjälp av **typparametrar** (eng. *type parameters*) kan du skapa **generiska strukturer** (eng. *generic structures*).
- Typparametrar skrivs inom hakparenteser, exempelvis: `[T, U]`
- En generisk struktur fungerar för *godtycklig* typ, **okänd** vid **deklaration**.
- Kompilatorn säkerställer **korrekt användning** redan vid *kompilering*.
- På användningsplatsen i koden (vid anrop eller instansiering) **binds** typparametrar till ”verkliga” typer enligt typsystemets regler.

```
case class Pair[A, B](a: A, b: B):  // A och B är obundna (fria) inom []
  def swap: Pair[B, A] = Pair(b, a) // A och B bundna då swap saknar []
```

- Skriver du inte ut typerna försöker kompilatorn **härleda** (eng. *infer*) dem:

```
scala> val p = Pair("hej", 42)
val p: Pair[String, Int] = Pair(hej,42)

scala> p.swap
val res0: Pair[Int, String] = Pair(42,hej)
```

- Klassen `Pair` kallas **typkonstruktor** (eng. *type constructor*) då den ”färdiga” typen `Pair[String, Int]` ”konstrueras” vid användning.

Typparameter, generisk struktur, typkonstruktor

- Med hjälp av **typparametrar** (eng. *type parameters*) kan du skapa **generiska strukturer** (eng. *generic structures*).
- Typparametrar skrivs inom hakparenteser, exempelvis: `[T, U]`
- En generisk struktur fungerar för *godtycklig* typ, **okänd** vid **deklaration**.
- Kompilatorn säkerställer **korrekt användning** redan vid *kompilering*.
- På användningsplatsen i koden (vid anrop eller instansiering) **binds** typparametrar till ”verkliga” typer enligt typsystemets regler.

```
case class Pair[A, B](a: A, b: B):  // A och B är obundna (fria) inom []
  def swap: Pair[B, A] = Pair(b, a) // A och B bundna då swap saknar []
```

- Skriver du inte ut typerna försöker kompilatorn **härleda** (eng. *infer*) dem:

```
scala> val p = Pair("hej", 42)
val p: Pair[String, Int] = Pair(hej,42)

scala> p.swap
val res0: Pair[Int, String] = Pair(42,hej)
```

- Klassen `Pair` kallas **typkonstruktor** (eng. *type constructor*) då den ”färdiga” typen `Pair[String, Int]` ”konstrueras” vid användning.

Övning: Ändra klassen `Pair` ovan så att båda elementen i paret har samma typ.

Generiska typgränser

Olika sätt att begränsa generiska typer

Det finns i Scala flera olika sätt att att begränsa vilka typer du vill tillåta – kompilatorn att hjälper dig att kontrollera detta.

- **Övre gräns** (eng. *upper bound*) med `A <: B`
- **Undre gräns** (eng. *lower bound*) med `A >: B`
- **Egentyp** (eng. *self type*) begränsar hur du kan mixa in en trait.

Olika sätt att begränsa generiska typer

Det finns i Scala flera olika sätt att att begränsa vilka typer du vill tillåta – kompilatorn att hjälper dig att kontrollera detta.

- **Övre gräns** (eng. *upper bound*) med `A <: B`
- **Undre gräns** (eng. *lower bound*) med `A >: B`
- **Egentyp** (eng. *self type*) begränsar hur du kan mixa in en trait. Ge ett godtyckligt namn följt av en typanotering och en raket i klasskroppen:

```
trait MinTrait:
  self: ÄrDennaTyp =>

  // Kod här kan utgå från att denna instans är av typen ÄrDennaTyp
  // namnet, här self, är valfritt, men self är vanligaste valet
```

  Kompilatorn kontrollerar att inmixing sker med `ÄrDennaTyp`.
- Det finns också något som heter **kontextgräns** (eng. *context bound*) där `[A: B]` gör så att typkonstruktorn `B` blir en kontextparameter `B[A]` – mer om det senare i avsnittet om kontextuella abstraktioner.

Övre och undre typgräns

Med typoperatorerna `<:` och `>:` går det att begränsa vilka typer som kan bindas till en typparameter i en generiska struktur.

- Minnesregel för typgränser: **kolon på slutet**.

Antag att T är en **obunden** typparameter, medan U och L är **bundna**.

- med T `<:` U blir U en övre gräns (eng. *upper bound*) för T
  U är ”högsta” möjliga typ som T får vara (jämför ”mindre eller lika med”)
- med T `>:` L blir L en undre gräns (eng. *lower bound*) för T
  U är ”lägsta” möjliga typ som T får vara (jämför ”större eller lika med”)
- För alla typer `A` gäller att `A >: Nothing` och `A <: Any`

Övre och undre typgräns

Med typoperatorerna `<:` och `>:` går det att begränsa vilka typer som kan bindas till en typparameter i en generiska struktur.

- Minnesregel för typgränser: **kolon på slutet**.

Antag att T är en **obunden** typparameter, medan U och L är **bundna**.

- med T `<:` U blir U en övre gräns (eng. *upper bound*) för T
  U är ”högsta” möjliga typ som T får vara (jämför ”mindre eller lika med”)
- med T `>:` L blir L en undre gräns (eng. *lower bound*) för T
  U är ”lägsta” möjliga typ som T får vara (jämför ”större eller lika med”)
- För alla typer `A` gäller att `A >: Nothing` och `A <: Any`

Exempel:

```
trait Grönsak { def vikt: Int }

def f[T <: Grönsak](x: T): Int = x.vikt
```

Kompilatorn använder den övre typgränsen för att konstatera att metoden `vikt` är tillgänglig via den generiska parametern `x`.

Exempel på övre och undre typgräns

```
class Djur
class Katt extends Djur
class Hund extends Djur
class Robothund extends Hund

def testUpperBound[T <: Hund](x: T) = println(x)
def testLowerBound[T >: Hund](x: T) = println(x)
```

```
scala> testUpperBound[Katt](Katt())
-- Error:
1 |testUpperBound[Katt](Katt())
  |               ^
  |               Type argument Katt does not conform to upper bound Hund

scala> testLowerBound[Robothund](Robothund())
-- Error:
1 |testLowerBound[Robothund](Robothund())
  |               ^
  |               Type argument Robothund does not conform to lower bound Hund
```

Varians: flexibla typparametrar

Vad är varians?

Vad är varians?

Är en kattbur också en djurbur??

Om vi tillåter **varians** så blir generiska strukturer mer **flexibla**.

Varför behövs varians?

```
trait Djur
case class Katt() extends Djur
case class Hund() extends Djur

case class Bur[A](a: A)
```

Varför behövs varians?

```
trait Djur
case class Katt() extends Djur
case class Hund() extends Djur

case class Bur[A](a: A)
```

Nedan fungerar inte! Buren ovan är **invariant** (oflexibel i sin typparameter).

```
1 scala> val djurbur: Bur[Djur] = Bur[Katt](Katt())
2 -- Error:
3 1 |val djurbur: Bur[Djur] = Bur[Katt](Katt())
4   |                        ^^^^^^^^^^^^^^^^^
5   |                        Found:    Bur[Katt]
6   |                        Required: Bur[Djur]
```

Varför behövs varians?

```
trait Djur
case class Katt() extends Djur
case class Hund() extends Djur

case class Bur[A](a: A)
```

Nedan fungerar inte! Buren ovan är **invariant** (oflexibel i sin typparameter).

```
scala> val djurbur: Bur[Djur] = Bur[Katt](Katt())
-- Error:
1 |val djurbur: Bur[Djur] = Bur[Katt](Katt())
  |                  ^^^^^^^^^^^^^^^^^
  |                  Found:    Bur[Katt]
  |                  Required: Bur[Djur]
```

Varför fungerar detta??

```
scala> val djur: Vector[Djur] = Vector[Katt](Katt())
val djur: Vector[Djur] = Vector(Katt())
```

Varför behövs varians?

```
trait Djur
case class Katt() extends Djur
case class Hund() extends Djur

case class Bur[A](a: A)
```

Nedan fungerar inte! Buren ovan är **invariant** (oflexibel i sin typparameter).

```
scala> val djurbur: Bur[Djur] = Bur[Katt](Katt())
-- Error:
1 |val djurbur: Bur[Djur] = Bur[Katt](Katt())
  |                  ^^^^^^^^^^^^^^^^^
  |                  Found:    Bur[Katt]
  |                  Required: Bur[Djur]
```

Varför fungerar detta??

```
scala> val djur: Vector[Djur] = Vector[Katt](Katt())
val djur: Vector[Djur] = Vector(Katt())
```

`Vector` är deklarerad som **kovariant** och därmed mer flexibel!

Kovarians (eng. *covariance*)

- För en **kovariant** typkonstruktor F gäller att:
  om `T <: U` **så** `F[T] <: F[U]` (subtypsflexibel ”på samma håll”)
- I Scala kan du få en kovariant typkonstruktor med + före typparametern:

```
trait Djur
case class Katt() extends Djur
case class Hund() extends Djur

case class Bur[+A](a: A)  // kovariant tack vare + före A
```

Kovarians (eng. *covariance*)

- För en **kovariant** typkonstruktor F gäller att:
  om `T <: U` **så** `F[T] <: F[U]` (subtypsflexibel ”på samma håll”)
- I Scala kan du få en kovariant typkonstruktor med + före typparametern:

```
trait Djur
case class Katt() extends Djur
case class Hund() extends Djur

case class Bur[+A](a: A)  // kovariant tack vare + före A
```

- Nu funkar det :)

```
1 scala> val djurbur: Bur[Djur] = Bur[Katt](Katt())
2 val djurbur: Bur[Djur] = Bur(Katt())
```

- `Bur[Katt]` är nu en **subtyp** till `Bur[Djur]`.

Kovarians (eng. *covariance*)

- För en **kovariant** typkonstruktor F gäller att:
  om `T <: U` **så** `F[T] <: F[U]` (subtypsflexibel ”på samma håll”)
- I Scala kan du få en kovariant typkonstruktor med + före typparametern:

```
trait Djur
case class Katt() extends Djur
case class Hund() extends Djur

case class Bur[+A](a: A)  // kovariant tack vare + före A
```

- Nu funkar det :)

```
1 scala> val djurbur: Bur[Djur] = Bur[Katt](Katt())
2 val djurbur: Bur[Djur] = Bur(Katt())
```

- `Bur[Katt]` är nu en **subtyp** till `Bur[Djur]`.
- **Oföränderliga** samlingar är ofta kovarianta, t.ex Vector, Option, List.

Kovarians (eng. *covariance*)

- För en **kovariant** typkonstruktor F gäller att:
  om `T <: U` **så** `F[T] <: F[U]` (subtypsflexibel ”på samma håll”)
- I Scala kan du få en kovariant typkonstruktor med + före typparametern:

```
trait Djur
case class Katt() extends Djur
case class Hund() extends Djur

case class Bur[+A](a: A)  // kovariant tack vare + före A
```

- Nu funkar det :)

```
1 scala> val djurbur: Bur[Djur] = Bur[Katt](Katt())
2 val djurbur: Bur[Djur] = Bur(Katt())
```

- `Bur[Katt]` är nu en **subtyp** till `Bur[Djur]`.
- **Oföränderliga** samlingar är ofta kovarianta, t.ex Vector, Option, List.
- Generiska enumerationer **behöver** vara kovarianta för att de olika fallen ska få en flexibel typparameter. `Ledig` är en `Toalett[Nothing]` här:
  `enum Toalett[+T] { case Upptagen(x: T); case Ledig }`

Kontravarians

Kontravarians

Är en kattveterinär också en djurveterinär?

Ibland vill vi ha variansen på andra hållet: En veterinär som bara kan behandla katter ska inte få behandla vilket djur som helst.

Kontravarians (eng. *contravariance*)

- För en **kontravariant** typkonstruktor F gäller att:
  om `T <: U` **så** `F[U] <: F[T]` (subtypsflexibel ”på fel håll”)
- Du skapar en kontravariant typkonstruktor med `-` före typparametern:

```
trait Djur
case class Katt() extends Djur
case class Hund() extends Djur

class Veterinär[-A]:    // kontravariant med -
  def behandla(x: A) = println(s"$this har behandlat $x")
```

Kontravarians (eng. *contravariance*)

- För en **kontravariant** typkonstruktor F gäller att:
  om `T <: U` **så** `F[U] <: F[T]` (subtypsflexibel ”på fel håll”)
- Du skapar en kontravariant typkonstruktor med `-` före typparametern:

```
trait Djur
case class Katt() extends Djur
case class Hund() extends Djur

class Veterinär[-A]:    // kontravariant med -
  def behandla(x: A) = println(s"$this har behandlat $x")
```

- Nu funkar det ”baklänges” (men inte på andra hållet):

```
scala> val kattveterinär: Veterinär[Katt] = Veterinär[Djur]()
val kattveterinär: Veterinär[Katt] = Veterinär@77b6d94c

scala> val kattveterinär: Veterinär[Djur] = Veterinär[Katt]()
-- Error:
1 |val kattveterinär: Veterinär[Djur] = Veterinär[Katt]()
  |                                     ^^^^^^^^^^^^^^^^^
  |                                     Found:    Veterinär[Katt]
  |                                     Required: Veterinär[Djur]
```

Variansproblem – tack kompilatorn!

- En typkonstruktor kan **inte** vara kovariant om typparametern används som parametertyp för metoder (så kallad *kontravariant position*):

```
trait Djur
case class Katt() extends Djur
case class Hund() extends Djur
```

```
scala> case class Bur[+A](a: A):
         def bytTill(x: A): Bur[A] = Bur(x)
-- Error:
2 |  def bytTill(x: A): Bur[A] = Bur(x)
  |           ^^^^
  |  covariant type A occurs in contravariant position in type A of parameter x
```

Variansproblem – tack kompilatorn!

- En typkonstruktor kan **inte** vara kovariant om typparametern används som parametertyp för metoder (så kallad *kontravariant position*):

```
trait Djur
case class Katt() extends Djur
case class Hund() extends Djur
```

```
scala> case class Bur[+A](a: A):
         def bytTill(x: A): Bur[A] = Bur(x)
-- Error:
2 |  def bytTill(x: A): Bur[A] = Bur(x)
  |          ^^^^
  |  covariant type A occurs in contravariant position in type A of parameter x
```

- Här måste typen tillåtas variera ”uppåt” med hjälp av en **undre gräns**:

```
case class Bur[+A](a: A):
  def bytTill[B >: A](x: B): Bur[B] = Bur(x)
```

```
scala> Bur[Katt](Katt()).bytTill(Hund())
val res1: Bur[Djur] = Bur(Hund())
```

När använda vilken slags varians?

- Oföränderliga generiska klasser som har metoder som är **producenter** av nya oföränderliga generiska typer passar ofta bäst som **kovarianta**, t.ex. `Vector[+T]`, `Option[+T]` .
- En typkonstruktor av `T`, som har metoder som är **konsumenter** av `T`, kan vara **kontravariant**, t.ex. `Veterinär[-T]`.
  Den kan också vara **kovariant** om konsumerande metoder **vidgar** inparametertypen till `B` där `B >: T`
- En typkonstruktor med flera parametrar kan vara **både** kontravariant **och** kovariant, t.ex. `Function1[-A, +B]`
  ett parametervärde `: A` *konsumeras* och ett returtvärde `: B` *produceras*
  (`Function1[-A, +B]` är egentliga typen av ”syntaktiska sockret” `A => B`

När använda vilken slags varians?

- Oföränderliga generiska klasser som har metoder som är **producenter** av nya oföränderliga generiska typer passar ofta bäst som **kovarianta**, t.ex. `Vector[+T]`, `Option[+T]` .
- En typkonstruktor av T, som har metoder som är **konsumenter** av T, kan vara **kontravariant**, t.ex. `Veterinär[-T]`.
  Den kan också vara **kovariant** om konsumerande metoder **vidgar** inparametertypen till B där `B >: T`
- En typkonstruktor med flera parametrar kan vara **både** kontravariant **och** kovariant, t.ex. `Function1[-A, +B]`
  ett parametervärde `:A` *konsumeras* och ett returtvärde `:B` *produceras*
  (`Function1[-A, +B]` är egentliga typen av ”syntaktiska sockret” `A => B`
- **Förändringsbara** strukturer måste vara **invarianta**, annars väntar **kaos**!
  Antag att det finns en `Bur` som kan ändras på plats via metoden `bytTill` *och samtidigt* vore kovariant (varning för känsliga kodare):

```
ejscala> val kattBur: Bur[Katt] = Bur(Katt())

ejscala> val djurBur: Bur[Djur] = kattBur  // en kovariant referens till samma Bur

ejscala> djurBur.bytTill(Hund())           // ändra på plats

ejscala> val katt: Katt = kattBur.släppUt  // KAOS! typosäkert hundkatt-monster :(
```

Typjoker: varning för gränslösa typer

Invarianta klasser har oflexibel typparameter, vilket begränsar användningen.

```
scala> class Box[A](val value: A)

scala> val b: Box[Any] = Box[Int](42)
-- Error:
1 |val b: Box[Any] = Box[Int](42)
  |                  ^^^^^^^^^^^
  |                  Found:    Box[Int]
  |                  Required: Box[Any]
```

Tveksam räddning: En **typjoker** (eng. *wildcard type*) skrivs med ett frågetecken och ger en helt okänd typ som **ej kontrolleras av kompilatorn**.

```
scala> var whatever: Box[?] = Box[Int](42)
var whatever: Box[?] = Box@70d7a49b

scala> whatever = Box("hej")
whatever: Box[?] = Box@43120a77
```

Använd bara typpjoker om det *verkligen* behövs.

Mer om varians för den nyfikne

- https://docs.scala-lang.org/scala3/book/types-variance.html
- https://en.wikipedia.org/wiki/Covariance_and_contravariance_-(computer_science)
- https://www.artima.com/pins1ed/type-parameterization.html

Egentyp + anonym klass för att ”injektera” beroenden

- Det går att få ett explicit, valfritt namn, t.ex. `self`, på ”mig själv”, alltså denna instans, genom att skriva `self =>` i kroppen på en trait eller klass.
- En typbegränsning på den egna instansen kallas **egentyp** (eng. *self type*).
- Användbart vid inmixning: du kan begränsa med vad denna trait får mixas in.

```
trait Grönsak { def vikt: Int }

trait KanSkalas:
  self: Grönsak =>
  def viktEfterSkalning = 0.99 * vikt
```

- Notera att *även* om `KanSkalas` *inte* gör **`extends`** så är *ändå* `vikt` tillgänglig i dess kropp, eftersom vi med egentypen kräver att `KanSkalas` är en `Grönsak`.

Egentyp + anonym klass för att ”injektera” beroenden

- Det går att få ett explicit, valfritt namn, t.ex. `self`, på ”mig själv”, alltså denna instans, genom att skriva `self =>` i kroppen på en trait eller klass.
- En typbegränsning på den egna instansen kallas **egentyp** (eng. *self type*).
- Användbart vid inmixning: du kan begränsa med vad denna trait får mixas in.

```
trait Grönsak { def vikt: Int }

trait KanSkalas:
  self: Grönsak =>
  def viktEfterSkalning = 0.99 * vikt
```

- Notera att *även* om `KanSkalas` *inte* gör **`extends`** så är *ändå* `vikt` tillgänglig i dess kropp, eftersom vi med egentypen kräver att `KanSkalas` är en `Grönsak`.
- Du kan du kombinera anonym klass och dynamisk inmixning med **`with`**:

```
scala> val g = new Grönsak with KanSkalas { val vikt = 100 }
val g: Grönsak & KanSkalas = anon1@70a91d72

scala> g.viktEfterSkalning
val res0: Double = 99.0
```

- Detta är ett elegant sätt att **injektera beroenden** (eng. *dependency injection*). `https://en.wikipedia.org/wiki/Dependency_injection`

Vad är fördelen med egentyper i stället för arv?

Arv tillåter **inte** ömsesidiga (cykliska) beroenden...

```
scala> trait Tulpan extends Ros; trait Ros extends Tulpan
-- [E110] Syntax Error: ----------------------------------------------------------------
1 |trait Tulpan extends Ros; trait Ros extends Tulpan
  |                      ^^^
  |                      Cyclic inheritance: trait Tulpan extends itself
  |
```

...medan egentyper *kan* vara **ömsesidigt beroende**:

```
trait Tulpan:
  self: Ros =>

trait Ros:
  self: Tulpan =>
```

```
scala> val tulipanaros = new Tulpan with Ros
val tulipanaros: Tulpan & Ros = anon1@1c63b39a
```

https://sv.wikipedia.org/wiki/Tulipanaros

Vad är ett bra api?

Vad är ett bra api?

- Ett api (eng. *Application Programming Interface*) är ett gränssnitt för applikationsprogrammering.
- Med ”gränssnitt” menas att det (i koden) finns en gräns mellan vad som syns utåt och vad som finns innanför ”under huven”.
- Det finns många kvalitetsaspekter som är önskvärda för ett api:

Vad är ett bra api?

- Ett api (eng. *Application Programming Interface*) är ett gränssnitt för applikationsprogrammering.
- Med ”gränssnitt” menas att det (i koden) finns en gräns mellan vad som syns utåt och vad som finns innanför ”under huven”.
- Det finns många kvalitetsaspekter som är önskvärda för ett api:
  - Enkelt att använda på utsidan även om insidan är komplex.
  - Vadå ”enkelt att använda”?
    - Lätt att lära
    - Lätt att komma ihåg
    - Lätt att begripa
    - Najs upplevelse (subjektivt)
  - Löser ett (generellt) problem på ett bra sätt. Vadå ”bra”?
    - Snabbt (hög prestanda)
    - Snålt (effektiv användning av resurser, t.ex. minne)
    - ...
- Intressant föredrag av Joshua Bloch (Google), om god api-design:
  - `https://research.google.com/pubs/archive/32713.pdf`
  - `https://youtu.be/aAb7hSCtvGw`

Api-desgin med Scala

- Kombinera paradigm OO+FP för att välja bäst lämpade lösningen.
- Avancerade abstraktionsmekanismer som kan vara utmanande för api-konstruktören men samtidigt bli enkelt för api-användaren
- Gör det möjligt att skapa ett api som fungerar i flera körmiljöer:
  - På desktop och i back-end med JVM och Graal VM
  - I front-end i webbläsaren med Scala JS
  - Direkt till plattformsspecifik maskinkod med Scala Native

Api-desgin med Scala

- Kombinera paradigm OO+FP för att välja bäst lämpade lösningen.
- Avancerade abstraktionsmekanismer som kan vara utmanande för api-konstruktören men samtidigt bli enkelt för api-användaren
- Gör det möjligt att skapa ett api som fungerar i flera körmiljöer:
  - På desktop och i back-end med JVM och Graal VM
  - I front-end i webbläsaren med Scala JS
  - Direkt till plattformsspecifik maskinkod med Scala Native
- Avancerade saker som vi inte gått in på i kursen men som kan hjälpa api-konstruktörer att göra lättanvänt api:
  - Kontextfunktioner `?=>`
  - Opaka typer `opaque type`
  - Typklassderivering `derives`
  - Metaprogrammering `inline`
  - Typ-lambda och match-typer `=>>`
  - ... (forskning pågår)

Köpa ett api för 55 miljarder?

Ericssons jätteaffär

Ericsson köper amerikanska Vonage, en plattform för app-utveckling, för 55 miljarder kronor. Strategin är att växa inom trådlösa företagslösningar.

–Vi är oerhört glada och uppåt här på Ericsson i dag, säger finanschefen Carl Mellander.

Mindre glada var aktieägarna. Att dra fördel av och få lönsamhet i sådana här stora affärer har historiskt visat sig vara svårt. Köpet är ett av de större i svensk företagshistoria.

Hade ni inte kunnat välja att samarbeta bara?

–Vi har utvärderat det. Kunde vi utveckla det här själv, ingå partnerskap? Vi landade i att förvärv var det bästa, den enda möjligheten att växa in i det här snabbt, säger Mellander.

Att plocka in mer mjukvara i Ericssonkoncernen bejon registrerade utvecklare globalt. Speciellt det sistnämnda är själva nyckeln, enligt Carl Mellander.

–Vad det här förvärvet handlar om, vi pratar om API:er, det är den mjukvara som gör att system och applikationer kan prata med varandra, säger Carl Mellander.

Han förklarar det som att Vonage är bryggan mellan app-utvecklaren och vad 5G-nätverken kan erbjuda.

FAKTA

Vonage i siffror

- Intäkterna under tredje kvartalet var 358 miljoner dollar, vilket under en rullande tolvmånadersperiod innebär årliga intäkter på 1,4 miljarder dollar.
- Efter skatt landade resultatet på minus 2 miljoner dollar under tredje kvartalet. Det justerade resultatet före avskrivningar blev plus 51 miljoner dollar.
- Aktien hade, innan affären med Ericsson annonserades, under året lyft med nästan 30

sta halvåret 202
har i dag över 5(
kronor netto i ka

Jonas Olavi, fon
på Alpcot, anser
är strategisk. Ha
att Ericsson har
på börstajminge
siktet inställt pa
ska generera int
fram i tiden.

–Affären är li
kande, eftersor
har varit några
kulationer innar
man tänker till

Sydsvenskan 2021-11-23

Kontextuella abstraktioner

Sammanhanget är avgörande när du kodar!

- **Kontexten**, alltså sammanhanget, styr vilka namn som syns var.

Sammanhanget är avgörande när du kodar!

- **Kontexten**, alltså sammanhanget, styr vilka namn som syns var.
- **Objektorientering** (OO) skapar sammanhang med:
  - **Namnrymder**
  - **Tillstånd**
  - **Subtypspolymorfism**

Sammanhanget är avgörande när du kodar!

- **Kontexten**, alltså sammanhanget, styr vilka namn som syns var.
- **Objektorientering** (OO) skapar sammanhang med:
  - **Namnrymder**
  - **Tillstånd**
  - **Subtypspolymorfism**
- **Funktionsprogrammering** (FP) skapar sammanhang med:
  - **Parametrar**
  - **Funktionsvärden**
  - **Parametrisk polymorfism**

Sammanhanget är avgörande när du kodar!

- **Kontexten**, alltså sammanhanget, styr vilka namn som syns var.
- **Objektorientering** (OO) skapar sammanhang med:
  - **Namnrymder**
  - **Tillstånd**
  - **Subtypspolymorfism**
- **Funktionsprogrammering** (FP) skapar sammanhang med:
  - **Parametrar**
  - **Funktionsvärden**
  - **Parametrisk polymorfism**
- **Kombinationen** av OO och FP är **extra** kraftfull och flexibel!

Sammanhanget är avgörande när du kodar!

- **Kontexten**, alltså sammanhanget, styr vilka namn som syns var.
- **Objektorientering** (OO) skapar sammanhang med:
  - **Namnrymder**
  - **Tillstånd**
  - **Subtypspolymorfism**
- **Funktionsprogrammering** (FP) skapar sammanhang med:
  - **Parametrar**
  - **Funktionsvärden**
  - **Parametrisk polymorfism**
- **Kombinationen** av OO och FP är **extra** kraftfull och flexibel!
- Men det finns **svårigheter**:
  - OO: ibland svårt att resonera om ändrade tillstånd och dynamisk bindning
  - FP: kan bli många parametrar som måste upprepas vid nästlade anrop

Sammanhanget är avgörande när du kodar!

- **Kontexten**, alltså sammanhanget, styr vilka namn som syns var.
- **Objektorientering** (OO) skapar sammanhang med:
  - **Namnrymder**
  - **Tillstånd**
  - **Subtypspolymorfism**
- **Funktionsprogrammering** (FP) skapar sammanhang med:
  - **Parametrar**
  - **Funktionsvärden**
  - **Parametrisk polymorfism**
- **Kombinationen** av OO och FP är **extra** kraftfull och flexibel!
- Men det finns **svårigheter**:
  - OO: ibland svårt att resonera om ändrade tillstånd och dynamisk bindning
  - FP: kan bli många parametrar som måste upprepas vid nästlade anrop
- Till vår räddning: fler **coola grejer** i Scala:
  - **Kontextparametrar**: möjliggör att **givna** (implicita) värden kan framkallas automatiskt av kompilatorn.
  - **Ad hoc polymorfism**: möjliggör olika implementationer beroende på **statisk** typ utan att det krävs en arvsrelation eller dynamisk bindning.

Repetition: default-argument

- Vi har tidigare sett att kompilatorn, med **default-argument**, kan **fylla i** värden, som vi inte måste skriva, vid funktionsanrop:

```
scala> def f(x: Int, y: Int = 42) = x + y
def f(x: Int, y: Int): Int

scala> f(1, 2)         // explicit y-värde
val res0: Int = 3

scala> f(1)           // vi kan också skippa y-värdet
val res1: Int = 43    // då används default-argumentet
```

Repetition: uppdelade parameterlistor

- Vi har tidigare sett uppdelade parameterlistor, som möjliggör stegvis applicering (s.k. Curry-funktioner):

```
scala> def f(x: Int)(y: Int = 42) = x + y
def f(x: Int)(y: Int): Int

scala> val g = f(1)  // en ny funktion utan default-arg
val g: Int => Int = Lambda1352/0x08406d0840@dbc7e0a

scala> f(1)()   // y-värdet fylls i av kompilatorn
val res4: Int = 43
```

Repetition: uppdelade parameterlistor

- Vi har tidigare sett uppdelade parameterlistor, som möjliggör stegvis applicering (s.k. Curry-funktioner):

```
scala> def f(x: Int)(y: Int = 42) = x + y
def f(x: Int)(y: Int): Int

scala> val g = f(1)  // en ny funktion utan default-arg
val g: Int => Int = Lambda1352/0x08406d0840@dbc7e0a

scala> f(1)()   // y-värdet fylls i av kompilatorn
val res4: Int = 43
```

- Kan vi ta detta ett steg till och **frikoppla** deklarationen av funktionen **från** det givna värdet, och ge detta på annan plats baserat på typen?

Repetition: uppdelade parameterlistor

- Vi har tidigare sett uppdelade parameterlistor, som möjliggör stegvis applicering (s.k. Curry-funktioner):

```
scala> def f(x: Int)(y: Int = 42) = x + y
def f(x: Int)(y: Int): Int

scala> val g = f(1)  // en ny funktion utan default-arg
val g: Int => Int = Lambda1352/0x08406d0840@dbc7e0a

scala> f(1)()   // y-värdet fylls i av kompilatorn
val res4: Int = 43
```

- Kan vi ta detta ett steg till och **frikoppla** deklarationen av funktionen **från** det givna värdet, och ge detta på annan plats baserat på typen?
- JA! I Scala 3 görs detta med **givna** värden och **kontextparametrar**, som skapas med nyckelorden **`given`** respektive **`using`**
  (i Scala 2 användes gamla nyckelordet **`implicit`**)

Givna värden + kontextparameter

Exmpel på användning av **`given`** och **`using`**:

```
case class Default(value: Int)

object Default:
  given d: Default = Default(0)  // Använd detta om inget annat givet värde finns lokalt
```

```
scala> def f(x: Int)(using d: Default) = x + d.value

scala> f(1)(using Default(2))    //explicit värde används alltid först om sådant finns
val res0: Int = 3

scala> f(1)  // kompilatorn framkallar ett givet värde för Default ur kompanjonsobjektet
val res1: Int = 1

scala> given d: Default = Default(42)  // låt d vara givna värdet i denna kontext
lazy val given_Default: Default

scala> f(1)       // kompilatorn framkallar nu det givna värdet i denna lokala kontext
val res2: Int = 43
```

Man kan utelämna namnet på värdet, eftersom det oftast inte behövs:

`given` `Default = Default(42)` eller ännu kortare: **`given`** `Default(42)`

Går det inte lika bra att ha en global variabel?

Varning för känsliga kodare!

- Ett försök att skapa ett default-värde med en global **`var`**-variabel:

```
scala> var globalVar = Default(42)
var ctx: Int = 42

scala> def f(x: Int, d: Default = globalVar) = x + d.value

scala> f(1)
val res3: Int = 43

scala> globalVar = Default(0) // FÖRÄNDRINGEN SLÅR GLOBALT I HELA DITT PROGRAM!!

scala> f(1)
val res4: Int = 1
```

- Här utgörs ”kontexten” av referensen till ett föränderligt värde som bestäms vid **körtid** i den globala namnrymden. Funktionen `f` är ej äkta!
- Denna lösning kan **inte** erbjuda **olika** kontextberoende default-värden; alla funktionsanrop använder **ett** globalt föränderliga värde. **Buggrisk**!

Givna värden med **`given`** härleds istället vid **kompileringstid** ur en speciell **implicit namnrymd** (eng. *implicit scope*) enligt speciella regler.

Import av kontextparameter

```
object EnNamnrymd:
  given enGivenSträng: String = "ett givet värde i EnNamnrymd"
  def framkalla(using s: String) = s     //kontextparametrar märks med using
```

Det är den **lokala** kontexten vid **användning** som styr vad som kan framkallas (och inte den namnrymd där kontextparametern deklarerades):

```
scala> EnNamnrymd.framkalla
-- [E172] Type Error: ----------------------------------------------------------
1 |EnNamnrymd.framkalla
  |                   ^
  | No given instance of type String was found for parameter s of method framkalla
  | in object EnNamnrymd. The following import might fix the problem:
  | import EnNamnrymd.enGivenSträng
```

Du kan importera givna värden med speciell syntax där typen anges istället för namnet:

```
scala> import EnNamnrymd.given String    // speciell import-syntax baserat på typ

scala> EnNamnrymd.framkalla
val res0: String = ett givet värde i EnNamnrymd
```

Framkalla värde med `summon`

I standardbiblioteket för Scala 3 finns en **generisk** variant av metoden `framkalla` i föregående exempel, som är definierad så här:

```
def summon[T](using x: T) = x
```

Funktionen `summon` kan användas för att testa vilket värde kompilatorn framkallar i en viss kontext:

```
object EnAnnanNamnrymd:
  given String = "tagen för givet"    // namn behövs ej, det räcker med typ
```

```
scala> summon[String]
-- Error:
1 |summon[String]
  |              ^
  | no implicit argument of type String was found for parameter x of
  | method summon in object Predef. The following import might fix the problem:
  | import EnAnnanNamnrymd.given_String

scala> import EnAnnanNamnrymd.given    // importerar alla givna värden i MinKontext

scala> summon[String]
val res0: String = tagen för givet
```

Prioritetsordning vid framkallning av givna värden

- Det kan finnas flera **olika** givna värden av **samma** typ om de finns i olika namnrymder.
- Det får **inte** vara **tvetydigt** vilket värde som ska framkallas.
- Därför finns det speciella prioriteringsregler:
  1. **Explicita** argument till kontextparametrar märkta med `using`
  2. `given` och `import given ...` i aktuell namnrymd (eng. *current scope*)
  3. `given`-värden i **kompanjonsobjekt** för den använda typen.
  4. ... (fler regler i speciella fall som vi inte går in på här)
- *Specialregel:* Om flera givna värden kan framkallas för typer som ingår i en gemensam **arvshierarki** så väljer kompilatorn det givna värdet som är av den **mest generella** typen.

Framkallning av givna värden har flera namn på engelska:

to summon, eller *term inference*, eller *implicit resolution*.

Ad hoc polymorfism

- Med så kallad **Ad hoc polymorfism** kan du ha *olika* implementationer av en funktion för *olika* typer *utan* att du behöver använda arv och dynamisk bindning.
- Detta uppfanns i språket ML och vidareutvecklades i språket Haskell.
- I Haskell skapas Ad hoc polymorfism med en s.k. ”**typklass**” (eng. *type class*).
- Detta görs i Scala genom att kombinera typparametrar och kontextparametrar.
- En ”typklass” i Scala är en tillståndslös `trait` med minst en typparameter och minst en abstrakt metod.

```
trait Parser[T]: //en typklass för tolkning och omvandling av strängar till godtycklig typ
  def fromString(value: String): Option[T]

object Parser:            //implementationer för en viss typ kan erbjudas som givna värden
  given Parser[Int] with        //nyckelordet with behövs då abstrakt medlem implementeras
    def fromString(value: String): Option[Int] = value.toIntOption
```

- Den specifika implementationen framkallas vid anrop med kontextparameter:

```
scala> def användParser[T](s: String)(using p: Parser[T]) = p.fromString(s)

scala> användParser[Int]("12")        // hittar givet värde i kompanjonsobjektet
val res0: Option[Int] = Some(12)
```

Hur få typklassen `Parser` att funka för fler typer?

```
scala> användParser[java.awt.Color]("Color(120,10,0)")
-- Error:
1 |användParser[java.awt.Color]("Color(120,10,0)")
  |                                        ^
  | no implicit argument of type Parser[java.awt.Color] was found
  | for parameter p of method användParser
```

```scala
given Parser[java.awt.Color] with
  def fromString(value: String): Option[java.awt.Color] =
    if !value.startsWith("Color(") then None else
      val trimmed = value.trim.stripPrefix("Color(").stripSuffix(")")
      trimmed.split(",").map(_.toIntOption) match
        case Array(Some(r),Some(g),Some(b)) => Some(java.awt.Color(r, g, b))
        case _ => None
```

```
scala> användParser[java.awt.Color]("Color(120,10,0)")
val res1: Option[java.awt.Color] = Some(java.awt.Color[r=120,g=10,b=0])
```

Hur få typklassen `Parser` att funka för fler typer?

```
scala> användParser[java.awt.Color]("Color(120,10,0)")
-- Error:
1 |användParser[java.awt.Color]("Color(120,10,0)")
  |                                        ^
  | no implicit argument of type Parser[java.awt.Color] was found
  | for parameter p of method användParser
```

```
given Parser[java.awt.Color] with
  def fromString(value: String): Option[java.awt.Color] =
    if !value.startsWith("Color(") then None else
      val trimmed = value.trim.stripPrefix("Color(").stripSuffix(")")
      trimmed.split(",").map(_.toIntOption) match
        case Array(Some(r),Some(g),Some(b)) => Some(java.awt.Color(r, g, b))
        case _ => None
```

```
scala> användParser[java.awt.Color]("Color(120,10,0)")
val res1: Option[java.awt.Color] = Some(java.awt.Color[r=120,g=10,b=0])
```

Detta ger **flexibilitet**: Jag kan ge givna värden för mina **egna** typer!

Namnet på kontextparametrar kan utelämnas

- Namnet på det givna värdet behövs ofta inte – det är ju *typen* som är det viktiga.
- Därför är det tillåtet att utelämna parameternamnet vid **`using`**.
- Det givna värdet kan istället framkallas med `summon` vid behov:

```
def användParser[T](s: String)(using Parser[T]) =
  summon[Parser[T]].fromString(s)
```

Kontextgräns

Denna form av **`using`**-parameter med en typkonstruktor...

```
def användParser[T](s: String)(using Parser[T])
```

...är så vanlig i Scala att det finns kortare skrivsätt som kallas **kontextgräns**:

```
def användParser[T: Parser](s: String)
```

Om `F[A]` är en typkonstruktor och du skriver `[T: F]` så blir `F` en så kallad **kontextgräns** (eng. *context boundary*). Kompilatorn expandera detta automatiskt till kontextparametern (`using F[T]`)

Ännu smidigare typklass med extensionsmetod

Det får att kombinera kontextgräns med extensionsmetoder:

```
extension [T: Parser](s: String)
  def parseOrElse(default: T): T =
    summon[Parser[T]].fromString(s).getOrElse(default)
```

Nu har vi smidig punktnotation:

```
scala> "12".parseOrElse(default = 42)
val res2: Int = 12

scala> "gurka".parseOrElse(default = 42)
val res3: Int = 42

scala> "Color(100,100,0)".parseOrElse(default = java.awt.Color.black)
val res4: java.awt.Color = java.awt.Color[r=100,g=100,b=0]

scala> "fel färg".parseOrElse(default = java.awt.Color.black)
val res5: java.awt.Color = java.awt.Color[r=0,g=0,b=0]
```

Sortera samlingar med given ordning

```
scala> case class Gurka(namn: String, vikt: Int)

scala> val xs = Vector(Gurka("a", 100), Gurka("b", 50), Gurka("c", 100))
val xs: Vector[Gurka] = Vector(Gurka(a,100), Gurka(b,50), Gurka(c,100))

scala> xs.sorted
-- Error:
1 |xs.sorted
  | ^
  | No implicit Ordering defined for B
  | where: B is a type variable with constraint >: Gurka
```

Sortera samlingar med given ordning

```
scala> case class Gurka(namn: String, vikt: Int)

scala> val xs = Vector(Gurka("a", 100), Gurka("b", 50), Gurka("c", 100))
val xs: Vector[Gurka] = Vector(Gurka(a,100), Gurka(b,50), Gurka(c,100))

scala> xs.sorted
-- Error:
1 |xs.sorted
  | ^
  | No implicit Ordering defined for B
  | where: B is a type variable with constraint >: Gurka
```

Detta kan fixas genom att tillhandahålla en given ordning för `Gurka`:

```
given Ordering[Gurka] with
  def compare(x: Gurka, y: Gurka): Int =
    if (x == y) then 0
    else if x.vikt < y.vikt then -1
    else 1
```

```
scala> xs.sorted   // nu funkar det :)
val res0: Vector[Gurka] = Vector(Gurka(b,50), Gurka(a,100), Gurka(c,100))
```

Sortera samlingar med ännu smidigare given ordning

- Det är vanligt att man vill definiera egna ordningsrelationer.
- Därför finns en smidig hjälpmetod i kompanjonsobjektet för typklassen `Ordering` som heter `fromLessThan`:

```
given Ordering[Gurka] =
  Ordering.fromLessThan((g1, g2) => g1.vikt < g2.vikt)
```

- Den tar som inparameter en funktion som tar två instanser som ska ordnas och ger **`true`** om den första ska anses **mindre** (alltså kommer **före**) enligt valfri definition.
- `fromLessThan` returnerar en `Ordering` som du kan låta vara givet.
- Prova gärna detta på veckans fördjupningsövningar.
- Läs mer om typklasser i Scala här: `https://docs.scala-lang.org/scala3/reference/contextual/type-classes.html`

Förslag på användning av kontextparameter i snake-labben

Antag att klassen Snake har två kontextparametrar:

```
class Snake (
  val initPos: Pos,
  val initDir: Dir,
  val headColor: Color,
  val tailColor: Color,
  )(using ctx: SnakeGame, settings: Settings) extends CanMove
```

- Var erbjuda default-värden för kontext-paramterar?

Förslag på användning av kontextparameter i snake-labben

Antag att klassen Snake har två kontextparametrar:

```
class Snake (
  val initPos: Pos,
  val initDir: Dir,
  val headColor: Color,
  val tailColor: Color,
  )(using ctx: SnakeGame, settings: Settings) extends CanMove
```

- Var erbjuda default-värden för kontext-paramterar?
  - Ett bra ställe att placera `given default: Settings = ???` är i kompanjonsobjekt till klassen `Settings`.
  - Du kan i kroppen på klassen `SnakeGame` ha en `given SnakeGame = this`
    Då kommer kompilatorn använda aktuell instans av `SnakeGame` som kontext-parameter när spelet i sin tur skapar instanser av `Snake`.

Förslag på användning av kontextparameter i snake-labben

Antag att klassen Snake har två kontextparametrar:

```
class Snake (
  val initPos: Pos,
  val initDir: Dir,
  val headColor: Color,
  val tailColor: Color,
  )(using ctx: SnakeGame, settings: Settings) extends CanMove
```

- Var erbjuda default-värden för kontext-paramterar?
  - Ett bra ställe att placera `given default: Settings = ???` är i kompanjonsobjekt till klassen `Settings`.
  - Du kan i kroppen på klassen `SnakeGame` ha en `given SnakeGame = this`
    Då kommer kompilatorn använda aktuell instans av `SnakeGame` som kontext-parameter när spelet i sin tur skapar instanser av `Snake`.

Notera denna text i labben, avsnitt *Tips och förslag*: ”Det är tillåtet att ändra, ta bort och lägga till, så länge de obligatoriska designkraven uppfylls.”

Översikt av kursens avslutning

Översikt av kursens avslutning

Återstående moment:

- W11: `snake` Alla ska vara aktiva på redovisningen.
- W12-W13: Projekt Individuellt arbete, välj gärna `bank`.
- W14: Munta På schematider endast ons-tors.
  - Om du är väl förberedd för muntan kan du göra den redan W11–W13 så snart alla labbar t.om. snake är klara.
  - Prata med handledare om din pluggplan.
  - Förbered dig noga med hjälp av `https://fileadmin.cs.lth.se/pgk/muntabot`
  - Läs om muntan i kompendiet.
- Tentaperiod januari: Valfri tentamen för överbetyg. Anmälan enl. LTH:s normala rutiner.

Se tentaschema i TimeEdit.
Mer info i kompendiet.