Vecka 6. Mönster och felhantering

Programmering, grundkurs (pgk)

Björn Regnell

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

EDAA45, Lp1-2, HT2023

Kompilerad den 7 november 2023

6 Mönster och felhantering

- Typhierarkier i Scala
- Mönstermatchning
- Hantera speciella värden med inkapsling
- Hantera saknade värden med `Option`
- Undantag
- Hantera undantag med `Try`
- Hantera undantag med `try-catch`
- För- och nackdelar med undantag
- Fördjupning: Implementera `equals`
- Veckans uppgifter

Typhierarkier i Scala

Bastypen för alla typer: Any

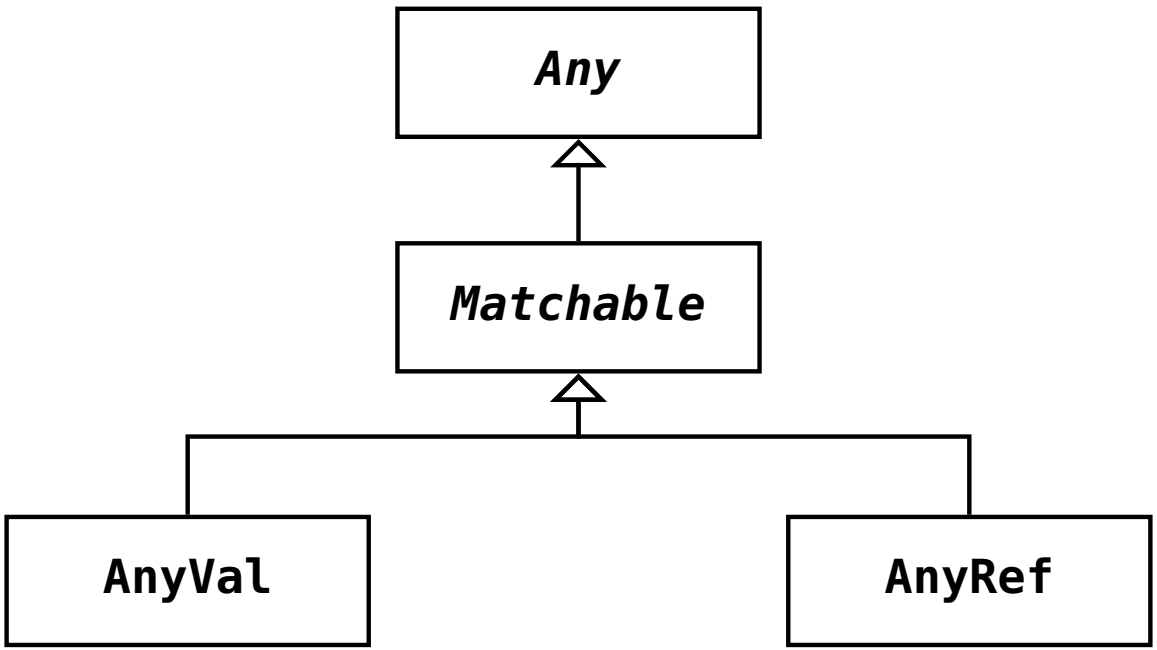
Scalas typsystem är **fullständigt**:

- Alla värden är objekt som har en typ.
- Alla typer är subtyper till bastypen `Any`.
- Typen `Any` kallas därför **topptyp**.
- Alla objekt har vissa grundläggande metoder, så som `toString` och `==`

```
trait Any:          // en förenklad beskrivning av Any
  def toString
  def ==
  def !=
  def equals
  def isInstanceOf
  def asInstanceOf
  def ##
  def hashCode
  def getClass

trait Matchable extends Any
```

Det kommer mer om abstrakta klasser och traits i veckan om arv.

Alla typer är subtyper till Any

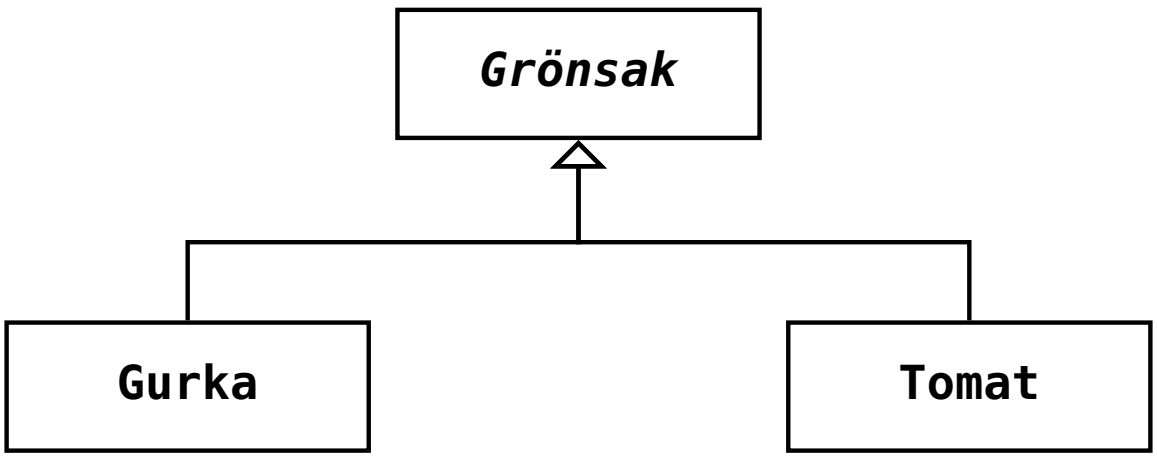


- Alla **värdetyper**, t.ex. `Int`, `Double`, `Boolean`, är subtyper till `AnyVal`
- Alla **referenstyper** t.ex. `String` är subtyper till `AnyRef`
- Värden av typen `Matchable` kan användas vid s.k mönstermatchning.
- (Det finns även s.k. opaka typer som inte kan mönstermatchas.)

Dina egna referenstyper är subtyper till `AnyRef`

Alla typer du skapar är subtyper till `AnyRef` utan att du behöver skriva det.

```
trait Grönsak:                                    // din egen bastyp
  def vikt: Int

case class Gurka(vikt: Int) extends Grönsak     // din egen subtyp
case class Tomat(vikt: Int) extends Grönsak     // en annan subtyp
```



Det kommer mer om typhierarkier och **`extends`** i veckan om arv.

Mönstermatchning

Vad är matchning?

- Matchning gör man då man vill jämföra ett värde mot andra värden och hitta överensstämmelse (eng. *match*) enligt olika **mönster**.
- Med mönster kan man även **plocka isär** objekt i sina beståndsdelar.

Plocka isär ett objekt i sina beståndsdelar med mönster

```
scala> case class Point(x: Int, y: Int)

scala> val p = Point(1, 2)      // konstruera en punkt
val p: Point = Point(1,2)
```

Plocka isär ett objekt i sina beståndsdelar med mönster

```
scala> case class Point(x: Int, y: Int)

scala> val p = Point(1, 2)      // konstruera en punkt
val p: Point = Point(1,2)
```

```
scala> val Point(x, y) = p      // plocka isär en punkt
val x: Int = 1
val y: Int = 2
```

Plocka isär ett objekt i sina beståndsdelar med mönster

```
scala> case class Point(x: Int, y: Int)

scala> val p = Point(1, 2)      // konstruera en punkt
val p: Point = Point(1,2)
```

```
scala> val Point(x, y) = p      // plocka isär en punkt
val x: Int = 1
val y: Int = 2
```

`Point(x, y)` kallas ett **konstruktormönster**.
Namnen `x` och `y` blir nya variabler.
Det finns många olika sorters mönster.
Vanligaste användningen av mönster är i **`match`**-uttryck.

Kolla om det passar med nästlade if-uttryck

Ett vanligt problem:
att kolla vilket bland många värden som passar

Kan göras med nästlade **`if-then-else`**-uttryck:

```
val g = scala.io.StdIn.readLine("Ange en grönsak:")
val smak =
  if      g == "gurka"    then "gott!"
  else if g == "tomat"    then "jättegott!"
  else if g == "broccoli" then "ganska gott..."
  else "inte gott :("

println(g + " är " + smak)
```

Matchning är ungefär som att passa klossar i en låda

Kolla om det passar med `match`-uttryck

I stället för nästlade **`if`** kan du använda Scalas kraftfulla **`match`-uttryck**:

```
val g = scala.io.StdIn.readLine("Ange en grönsak: ")
val smak =
  g match
    case "gurka"    => "gott!"
    case "tomat"    => "jättegott!"
    case "broccoli" => "ganska gott..."
    case _ => "mindre gott..."
```

Kolla om det passar med `match`-uttryck

I stället för nästlade **`if`** kan du använda Scalas kraftfulla **`match`-uttryck**:

```
val g = scala.io.StdIn.readLine("Ange en grönsak: ")
val smak =
  g match
    case "gurka"    => "gott!"
    case "tomat"    => "jättegott!"
    case "broccoli" => "ganska gott..."
    case _ => "mindre gott..."
```

- Varje **`case`**-gren testas var för sig i tur och ordning **uppifrån och ned**.
- Det som står mellan **`case`** och **`=>`** kallas ett **mönster** (eng. *pattern*)
- Om ett mönster matchar så görs det som står efter **`=>`**
- **Inga efterföljande `case`-grenar testas efter lyckad match**.
- Ovan är exempel på matchning mot **konstant-mönster**, i detta fallet tre stycken strängkonstantmönster.
- Sista default-grenen ovan kallas **wildcard-mönster**: **`case _ =>`**
- Det finns många andra sätt att skriva mönster.

Syntax för match-uttryck

Ett **`match`**-uttryck består av godtyckligt många
`case ... => ...`

```
värdeAttUndersöka match {
  case mönster1 => resultat1
  case mönster2 => resultat2
  case mönster3 => resultat3
  case mönsterN => resultatN
}
```

- Klammerparenteser efter **`match`** valfria om **`case`** på ny rad.
- Varje resultat-uttryck kan bestå av många rader.
- Klammerparenteser behövs ej efter **`=>`** vid många rader.

Om många rader efter **`case`** så blir sista uttrycket resultatet.
Vi ska nu se exempel på många olika mönster

Matchning med gard

Man kan stoppa in en s.k **gard** (eng. *guard*) innan pilen **=>** för att villkora matchningen: (notera **if** utan **then**)

```
val g = scala.io.StdIn.readLine("Ange en grönsak: ")
val smak = g match
  case "gurka" if math.random() > 0.5 => "gott ibland!"
  case "tomat" => "jättegott!"
  case "broccoli" => "ganska gott..."
  case _ => "mindre gott..."
```

case-grenen med gard ger bara en lyckad matchning om uttrycket efter **if** är sant; annars provas nästa gren, etc.

Matchning med variabelmönster

Om det finns ett namn efter **`case`** som börjar med liten begynnelsebokstav, blir detta namn en variabel som automatiskt binds till uttrycket före **`match`**:

```
val g = scala.io.StdIn.readLine("Ange en grönsak: ")
val smak = g match
  case "gurka" if math.random() > 0.5 => "gott ibland!"
  case "tomat" => "jättegott!"
  case "broccoli" => "ganska gott..."
  case other => "smakar bakvänt: " + other.reverse
```

Ett **enkelt variabelmönster**, så som
`case` `other` **`=>`** `...`
i exemplet ovan, matchar **allt**!
`other` får alltså värdet av g om g **inte** är `"gurka"`, `"tomat"`, `"broccoli"`.

Matchning med eller-mönster

Om man har samma utfall för olika grenar kan dessa slås ihop och mönstret separeras med vertikalstreck: |

```
val g = scala.io.StdIn.readLine("Ange en grönsak: ")
val smak = g match
  case "gurka" => "gott"
  case "tomat" => "gott"
  case "lök"   => "gott"
  case _ => "inte gott"
```

Mer koncist med eller-mönster:

```
val g = scala.io.StdIn.readLine("grönsak:")
val smak = g match
  case "gurka" | "tomat" | "lök" => "gott"
  case _ => "inte gott"
```

Matchning med typade mönster

Med en typannotering efter en variabel får man ett **typat mönster** (eng. *typed pattern*). Om matchningen lyckas blir värdet **omvandlat** till den specifika typen och binds till variabeln.

```
def f() =
  if math.random() < 0.5 then 42 + math.random()
  else s"gurka ${math.random()}"

val i = f() match
  case x: Double => x.round.toInt
  case s: String => s.length
```

`f()` får typen `Matchable` som är subtyp till `Any`. Vilken typ får `i`?

Matchning med typade mönster

Med en typannotering efter en variabel får man ett **typat mönster** (eng. *typed pattern*). Om matchningen lyckas blir värdet **omvandlat** till den specifika typen och binds till variabeln.

```
def f() =
  if math.random() < 0.5 then 42 + math.random()
  else s"gurka ${math.random()}"

val i = f() match
  case x: Double => x.round.toInt
  case s: String => s.length
```

`f()` får typen `Matchable` som är subtyp till `Any`. Vilken typ får `i`? `Int`

Matchning mot specifika typer enl. ovan används i idiomatisk Scala hellre än `isInstanceOf` och `asInstanceOf` men man kan göra motsvarande ovan så här:

```
val i2: Int =
  val x = f()
  if x.isInstanceOf[Double] then x.asInstanceOf[Double].round.toInt
  else if x.isInstanceOf[String] then x.asInstanceOf[String].length
  else throw scala.MatchError(x)
```

Fördjupning: Unionstyper och typen Matchable

- Exempel: För de orelaterade typerna `String` och `Int` är den mest specifika typen som kan härledas `Int | String`, läses ”Int eller String” och kallas en **unionstyp** (eng. *union type*).

```
scala> def f() = math.random() match
         case a if a > 0.5 => 42
         case a if a < 0.2 => "hej"
       ;

def f(): Int | String
```

- Alla värden som kan undersökas med **`match`** har typen `Matchable` .
- Typen `Matchable` är nästan lika generell som topptypen `Any`.

```
scala> (f().isInstanceOf[Matchable], f().isInstanceOf[Any])
val res0:  (Boolean, Boolean) = (true,true)
```

- `Matchable` infördes i Scala 3 med **opaka typalias** som garanterat aldrig boxas men inte kan mönstermatchas. (Ingår ej i denna kurs.)
- Fördjupning om `Matchable` och **`opaque type`** i Scala 3 finns här: `https://dotty.epfl.ch/docs/reference/other-new-features`

Konstruktormönster med case-klasser

En basklass med gemensamma delar och två subtyper:

```
trait Grönsak:
  def vikt: Int
  def ärRutten: Boolean

case class Gurka(vikt: Int, ärRutten: Boolean) extends Grönsak
case class Tomat(vikt: Int, ärRutten: Boolean) extends Grönsak
```

Konstruktormönster med case-klasser

En basklass med gemensamma delar och två subtyper:

```
trait Grönsak:
  def vikt: Int
  def ärRutten: Boolean

case class Gurka(vikt: Int, ärRutten: Boolean) extends Grönsak
case class Tomat(vikt: Int, ärRutten: Boolean) extends Grönsak
```

Tack vare case-klasserna kan man använda **konstruktormönster** (eng. *constructor pattern*) för att se vad som finns **inuti** en instans:

```
def testa(g: Grönsak): String = g match
  case Gurka(v, false) => "gott, väger " + v
  case Gurka(_, true)  => "inte gott"
  case Tomat(v, r)     => (if r then "inte " else "") + s"gott, väger $v"
  case _ => s"okänd grönsak: $g"
```

Konstruktormönster ”**plockar isär**” det som matchas och binder variabler till de attribut som finns i case-klassens konstruktor.

Plocka isär samlingar med djupa mönster

- Man kan plocka isär innehållet i en samling så här:

```
def visa(xs: Vector[Grönsak]): String = xs match
  case Vector()                => "tom grönsaksvektor"
  case Vector(Gurka(v, true)) => s"en rutten gurka som väger $v"
  case Vector(g)               => s"exakt en grönsak: $g"
  case Vector(g1, g2)          => s"exakt två grönsaker: $g1, $g2"
  case Vector(g, gs*)          => s"först en $g och sedan svansen: $gs"
```

- Vad händer om du byter ordning på 2:a och 3:e mönstret?
- `Vector(g, gs*)` kan också skrivas som `g +: gs`

Matchning på tupler

Det går fint att plocka isär tupler med mönstermatchning:[1]

```
var pair = ("hej", 42)

pair match
  case (a, b) if b == 42 => s"livets mening är funnen: $a"
  case (_, b)            => s"fattas mening: $b"
```

[1] https://youtu.be/aboZctrHfK8

Mönstermatchning och uppräkning med case-objekt

En bastyp och specifika singelobjekt av gemensam typ:

```
trait Färg
case object Spader  extends Färg // funkar utan case men vi vill ha najs toString
case object Hjärter extends Färg
case object Ruter   extends Färg
case object Klöver  extends Färg

def parallellFärg(f: Färg): Färg = f match
  case Spader  => Klöver
  case Klöver  => Spader
  case Hjärter => Ruter
```

Vilken case-gren har vi glömt? Kan kompilatorn hjälpa oss?

Mönstermatchning och uppräkning med case-objekt

En bastyp och specifika singelobjekt av gemensam typ:

```
trait Färg
case object Spader  extends Färg // funkar utan case men vi vill ha najs toString
case object Hjärter extends Färg
case object Ruter   extends Färg
case object Klöver  extends Färg

def parallellFärg(f: Färg): Färg = f match
  case Spader  => Klöver
  case Klöver  => Spader
  case Hjärter => Ruter
```

Vilken case-gren har vi glömt? Kan kompilatorn hjälpa oss?

```
1 scala> parallellFärg(Ruter)
2 scala.MatchError: Ruter
```

Undantag vid körtid : (

Mönstermatchning och förseglade typer

Med nyckelordet **`sealed`** får vi en kompileringsvarning.

```
sealed trait Färg //tryck Alt+Enter i REPL för tolkning av flera rader ett svep
case object Spader  extends Färg
case object Hjärter extends Färg
case object Ruter   extends Färg
case object Klöver  extends Färg

def parallellFärg(f: Färg): Färg = f match
  case Spader  => Klöver
  case Klöver  => Spader
  case Hjärter => Ruter
```

```
1 |def parallellFärg(f: Färg): Färg = f match
  |                                    ^
  |                           match may not be exhaustive.
  |
  |                           It would fail on pattern case: Ruter
def parallellFärg(f: Färg): Färg
```

Varning vid kompilering :) Sista raden visar att det bara är en varning!

Mönstermatcha enumeration

I stället för **`sealed trait ... case object ...`** kan du använda en **enumeration** (ä.k. uppräkning, uppräknad datatyp, (eng. *enumeration*)).

```
enum Färg:
  case Spader, Hjärter, Ruter, Klöver

def parallellFärg(f: Färg): Färg =
  import Färg.*
  f match
    case Spader  => Klöver
    case Klöver  => Spader
    case Hjärter => Ruter
```

Mönstermatcha enumeration

I stället för **`sealed trait ... case object ...`** kan du använda en **enumeration** (ä.k. uppräkning, uppräknad datatyp, (eng. *enumeration*)).

```
enum Färg:
  case Spader, Hjärter, Ruter, Klöver

def parallellFärg(f: Färg): Färg =
  import Färg.*
  f match
    case Spader  => Klöver
    case Klöver  => Spader
    case Hjärter => Ruter
```

```
1  def parallellFärg(f: Färg): Färg
2  3 |  f match
3    |  ^
4    |  match may not be exhaustive.
5    |
6    |  It would fail on pattern case: Ruter
```

Även här får vi hjälpsam varning vid kompilering :)

Stora/små begynnelsebokstäver vid matchning

Fallgrop: matcha **värde** som börjar med **liten** bokstav.

```
1  scala> val livetsMening = 42
2
3  scala> def ärLivetsMeningBuggig(svar: Int) = svar match
4           case livetsMening => true    // lokalt namn som matchar allt!
5           case _ => false
6
7  scala> ärLivetsMeningBuggig(43)
8  val res0: Boolean = true
9
10 scala> val LivetsMening = 42   // stor begynnelsebokstav
11
12 scala> def ärLivetsMening(svar: Int) = svar match
13          case LivetsMening => true    // funkar fint!
14          case _ => false
15
16 scala> ärLivetsMening(43)
17 val res1: Boolean = false
```

Stora/små begynnelsebokstäver vid matchning

Ett sätt att komma runt problemet med liten begynnelsebokstav: **backticks** to the rescue!

```
scala> val livetsMening = 42

scala> def ärLivetsMeningBackTicks(svar: Int) = svar match
         case `livetsMening` => true    // nu funkar det!
         case _ => false

scala> ärLivetsMeningBackTicks(43)
val res2: Boolean = false
```

Mönster på andra ställen än i `match`

Mönster i **deklarationer**:

```
scala> case class Point(x: Int, y: Int)

scala> val p = Point(0, 1)

scala> val Point(x, y) = p          // konstruktormönster med case-klass
val x: ???
val y: ???

scala> val (x, y, z) = (0, 1, 2)    // konstruktormönster med tupel
val x: ???
val y: ???
val z: ???
```

Mönster i **for-uttryck**:

```
scala> val xs = for (x, y) <- Vector((1,2), (3,4)) yield x
val xs: ???
```

Mönster på andra ställen än i `match`

Mönster i **deklarationer**:

```
scala> case class Point(x: Int, y: Int)

scala> val p = Point(0, 1)

scala> val Point(x, y) = p          // konstruktormönster med case-klass
val x: Int = 0
val y: Int = 1

scala> val (x, y, z) = (0, 1, 2)    // konstruktormönster med tupel
val x: Int = 0
val y: Int = 1
val z: Int = 2
```

Mönster i **for-uttryck**:

```
scala> val xs = for (x, y) <- Vector((1,2), (3,4)) yield x
val xs: Vector[Int] = Vector(1, 3)
```

Mönsterdelar och variabelt antal argument

Met två olika specialtecken går det att

- binda variabler till **mönsterdelar** med **@**
  ```
  case Vector(xs@Vector(a), Vector(42)) => ...
  ```
- matcha **variabelt antal argument** med ∗
  ```
  case Vector(a, _, c) => ...
  ```
  matchar om 3 element, _ kvittar
  ```
  case Vector(a, svans*) => ...
  ```
  matchar om minst ett element
  ```
  case Vector(a, _*) => ...
  ```
  intresserad av första, svans kvittar

Partiella funktioner och metoden `collect`

- En **partiell funktion** är, till skillnad från en **total funktion**, inte definierad för alla parametervärden.
- Partiella funktioner kan skapas med **`case`** utan **`match`**:

```
val pf: PartialFunction[Int, Double] = { case z if z != 0 => 1.0 / z }
```

- Funktionen är inte definierad för argumentet `0`:

```
scala> pf(0)
scala.MatchError: 0
```

- Detta är användbart tillsammans med samlingsmetoden `collect` som applicerar en partiell funktion endast på definierade värden:

```
scala> Vector(1, 2, 0, 4).collect(pf)
val res0: Vector[Double] = Vector(1.0, 0.5, 0.25)

scala> Vector(1 -> 2, 0 -> 3, 42 -> 0).collect{ case (a,b) if a > 0 => a }
val res1: Vector[Int] = Vector(1, 42)
```

- Notera att **krullparentes behövs** vid ensamt **`case`**.

Fördjupning: metoden `unapply`

När du deklarerar en case-klass kommer kompilatorn att **automatiskt generera en metod** med namnet **`unapply`**.

```
1  scala> case class Gurka(vikt: Int, ärRutten: Boolean)
2
3  scala> Gurka.unapply // tryck ENTER för att se typen
4  val res0: Gurka => Gurka = Lambda1914/0x00000008408cf840@b0e7bde
5
6  scala> val g = Gurka(100, false)
7
8  scala> Gurka.unapply(g)
9  val res1: Gurka = Gurka(100,false)
```

Vad ska detta vara bra för?

Fördjupning: metoden `unapply`

När du deklarerar en case-klass kommer kompilatorn att **automatiskt generera en metod** med namnet **`unapply`**.

```
1  scala> case class Gurka(vikt: Int, ärRutten: Boolean)
2
3  scala> Gurka.unapply // tryck ENTER för att se typen
4  val res0: Gurka => Gurka = Lambda1914/0x00000008408cf840@b0e7bde
5
6  scala> val g = Gurka(100, false)
7
8  scala> Gurka.unapply(g)
9  val res1: Gurka = Gurka(100,false)
```

Vad ska detta vara bra för? Metoden `unapply` genereras av kompilatorn och används internt vid matchning och det är den metoden som gör att case-klasser kan användas i konstruktormönster. Principen är generell: Man kan skapa **egna** s.k. **extraktorer** (eng. *extractors*) som kan plocka isär ett värde med mönstermatchning, även utan case-klass.

För den nyfikne: `https://docs.scala-lang.org/scala3/reference/changed-features/pattern-matching.html`

Hantera speciella värden med inkapsling

Inkapsling av speciella värden så att krasch kan undvikas

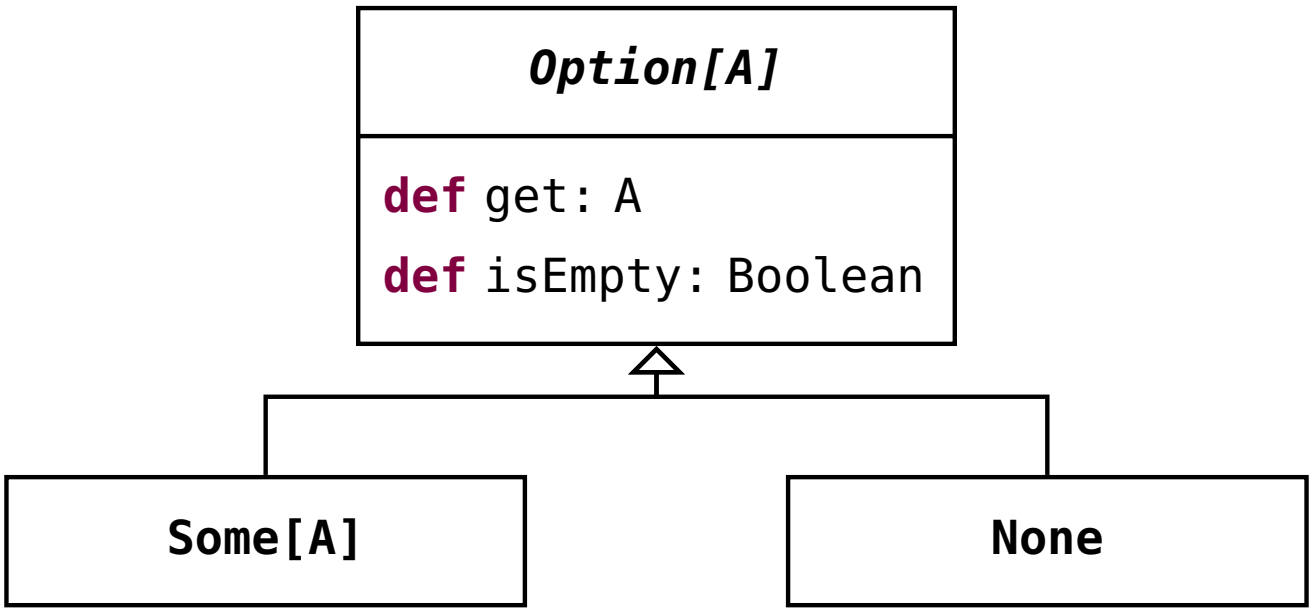
Hantera saknade värden med `Option`

Hur hantera saknade värden?

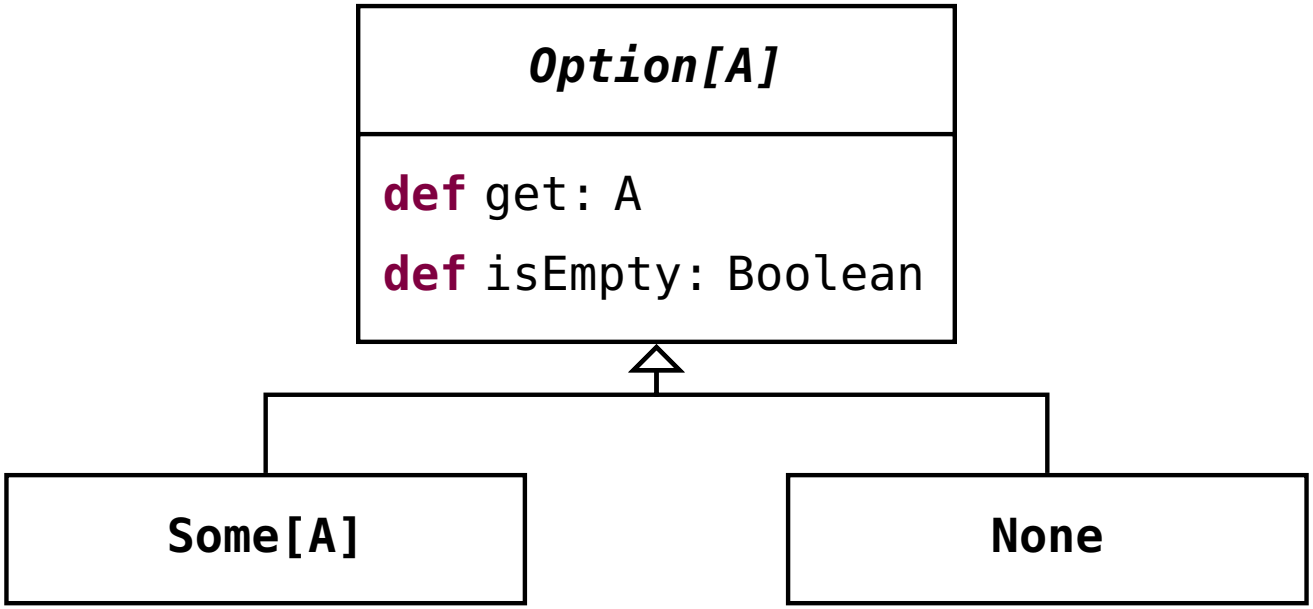
Olika sätt att hantera saknade värden:

- Hitta på ett specialvärde: exempel -1 för saknat värde
- `null` om värde saknas (vanligt i Java m.fl. språk, mkt ovanligt i Scala)
- Använd en samling och låt tom samling representera saknat värde:
  ```
  val sums = Vector(Vector(42),Vector(32),Vector(),Vector(21))
  ```
- `Option[A]` gemensam bastyp för:
  `None` som representerar **saknat värde**, och
  `Some[A]` som representerar att **värde finns**

En gemensam bastyp för ett värde som kanske saknas



En gemensam bastyp för ett värde som kanske saknas




```
scala> var x: Option[Int] = Some(42)

scala> x.isEmpty
val res0: Boolean = false

scala> x = None

scala> x.isEmpty
val res1: Boolean = true
```

Option för hantering av ev. saknade värden

Alla vill inte berätta för Facebook vad de har för kön.
Förbättra Facebooks kod med ett litet Scala-program:

```
enum Gender:
  case Male, Female

case class Person(name: String, gender: Option[Gender])
```

Option för hantering av ev. saknade värden

Alla vill inte berätta för Facebook vad de har för kön.
Förbättra Facebooks kod med ett litet Scala-program:

```
enum Gender:
  case Male, Female

case class Person(name: String, gender: Option[Gender])
```

```
1  scala> val p1 = Person("Björn",  Some(Gender.Male))
2  scala> val p2 = Person("Sandra", Some(Gender.Female))
3  scala> val p3 = Person("Kim",  None)
4  scala> val g2 = p2.gender
5  scala> def show(g: Option[Gender]): String = g match {
6           case Some(x) => x.toString
7           case None    => "unknown"
8         }
9  scala> show(g2)
10 scala> show(p3.gender)
11 scala> val ps = Vector(p1,p2,p3)
12 scala> ps.map(_.gender).map(show)   // None ignoreras av map
```

Några smidiga metoder på `Option`

Metoden `getOrElse` gör att man ofta kan undvika matchning.

```
var opt: Option[Int] = None

val x = opt.getOrElse(42)   // get the value, give default if missing
```

Flera av de vanliga samlingsmetoderna funkar, t.ex. `foreach` och `map`.

```
opt.foreach(x => println(x)) // only done if value exists

opt.map(x => x + 1)          // only done if value exists

opt = Some(42)               // change opt to now have some value

opt.foreach(x => println(x)) // done as value now exists

opt.map(x => x + 1)          // done as value now exists
```

Några samlingsmetoder som ger en `Option`, övning

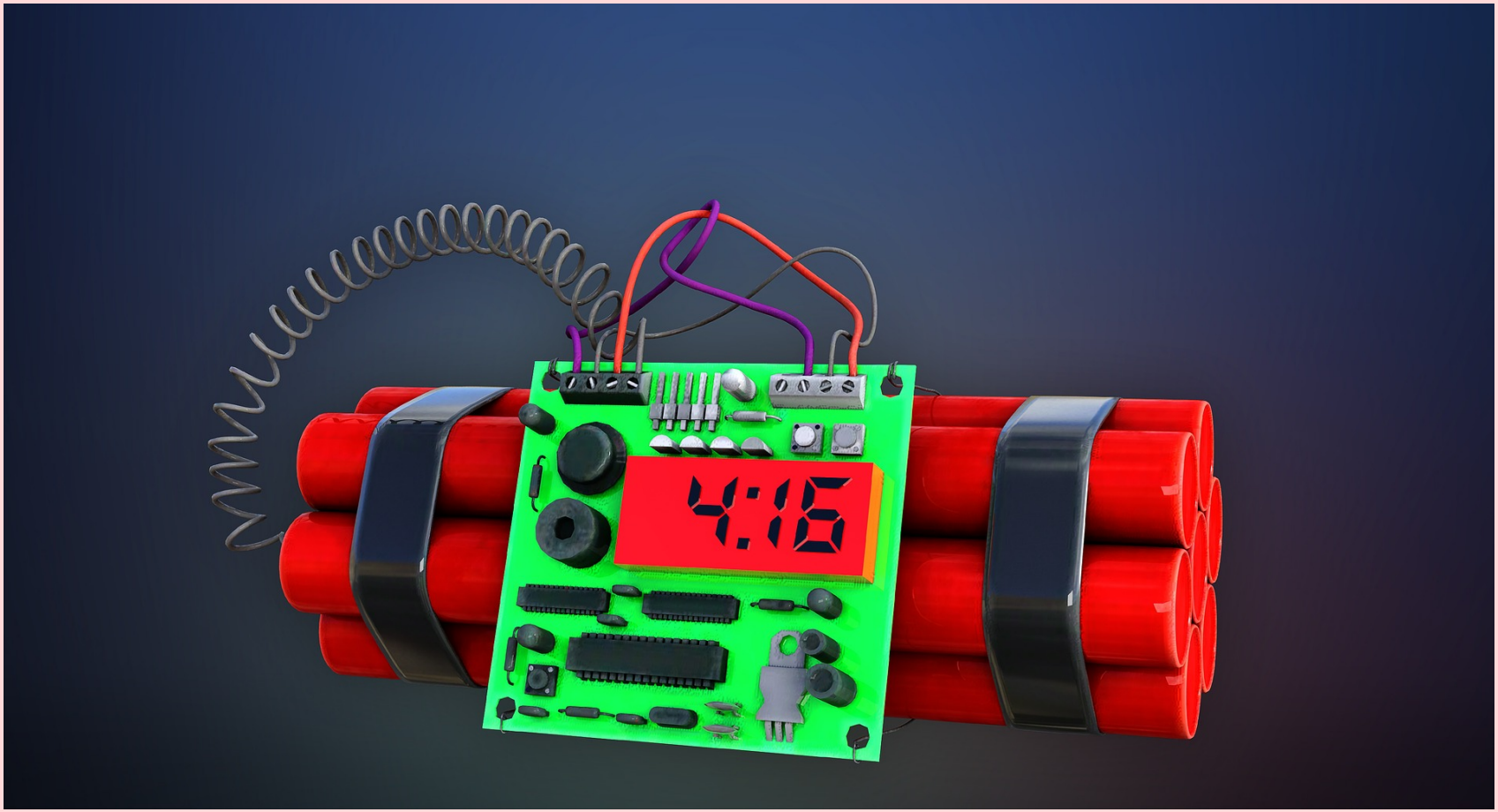
```
scala> val (xs, ys) = (Vector(1,2,3), Vector())

scala> xs.headOption
res0: ???

scala> ys.headOption
res1: ???

scala> xs.find(_ > 1)
res2: ???

scala> xs.find(_ > 5)
res3: ???

scala> val huvudstad = Map("Sverige" -> "Sthlm", "Skåne" -> "Malmö")

scala> huvudstad.get("Skåne")
res4: ???

scala> huvudstad.get("Danmark")
res5: ???
```

Några samlingsmetoder som ger en `Option`, svar

```
scala> val (xs, ys) = (Vector(1,2,3), Vector())

scala> xs.headOption
res0: Option[Int] = Some(1)

scala> ys.headOption
res1: Option[Nothing] = None

scala> xs.find(_ > 1)
res2: Option[Int] = Some(2)

scala> xs.find(_ > 5)
res3: Option[Int] = None

scala> val huvudstad = Map("Sverige" -> "Sthlm", "Skåne" -> "Malmö")

scala> huvudstad.get("Skåne")
res4: Option[String] = Some(Malmö)

scala> huvudstad.get("Danmark")
res5: Option[String] = None
```

Undantag

Undantag kan orsaka krasch...



Undantag orsakar ingen krasch om inkapslad i en Try

Vad är ett undantag (eng. *exception*)?

Undantag representerar ett fel eller ett onormalt tillstånd som upptäcks under exekvering och som behöver hanteras på särskilt sätt vid sidan av det normala exekveringsflödet.

sv.wikipedia.org/wiki/Undantagshantering

Exempel på undantag:

Vad är ett undantag (eng. *exception*)?

Undantag representerar ett fel eller ett onormalt tillstånd som upptäcks under exekvering och som behöver hanteras på särskilt sätt vid sidan av det normala exekveringsflödet.

sv.wikipedia.org/wiki/Undantagshantering

Exempel på undantag:

- Indexering utanför vektorns indexgränser.
- Läsning bortom filens slut.
- Försök att öppna en fil som inte finns.
- Minnet är slut.
- Heltalsdivision med noll ger `java.lang.ArithmeticException`.
- `"hej".toInt` ger `java.lang.NumberFormatException`

Orsaka undantag indirekt med `require` och `assert`

- Med funktionen `require(b)` skapas ett `IllegalArgumentException("requirement failed")` om b är **`false`**
- `require` används om man vill begränsa vilka argument som är giltiga
- Med funktionen `assert(b)` skapas ett `AssertionError("assertion failed")` om b är **`false`**
- `assert` används om man vill förhindra ogiltiga tillstånd

Se implementationen av `require` här:

`https://github.com/scala/scala/blob/v2.13.6/src/library/scala/Predef.scala#L315`

Kasta dina egna undantag med `throw`

Man kan själv generera ett undantag med **`throw`**, vilket kallas att **kasta** ett undantag som (om det inte **fångas**), gör att exekveringen **avbryts**.

```
1  scala> def pang = throw Exception("PANG!")
2  pang: Nothing
3
4  scala> pang
5  java.lang.Exception: PANG!
```

Kasta dina egna undantag med `throw`

Man kan själv generera ett undantag med **`throw`**, vilket kallas att **kasta** ett undantag som (om det inte **fångas**), gör att exekveringen **avbryts**.

```
1  scala> def pang = throw Exception("PANG!")
2  pang: Nothing
3
4  scala> pang
5  java.lang.Exception: PANG!
```

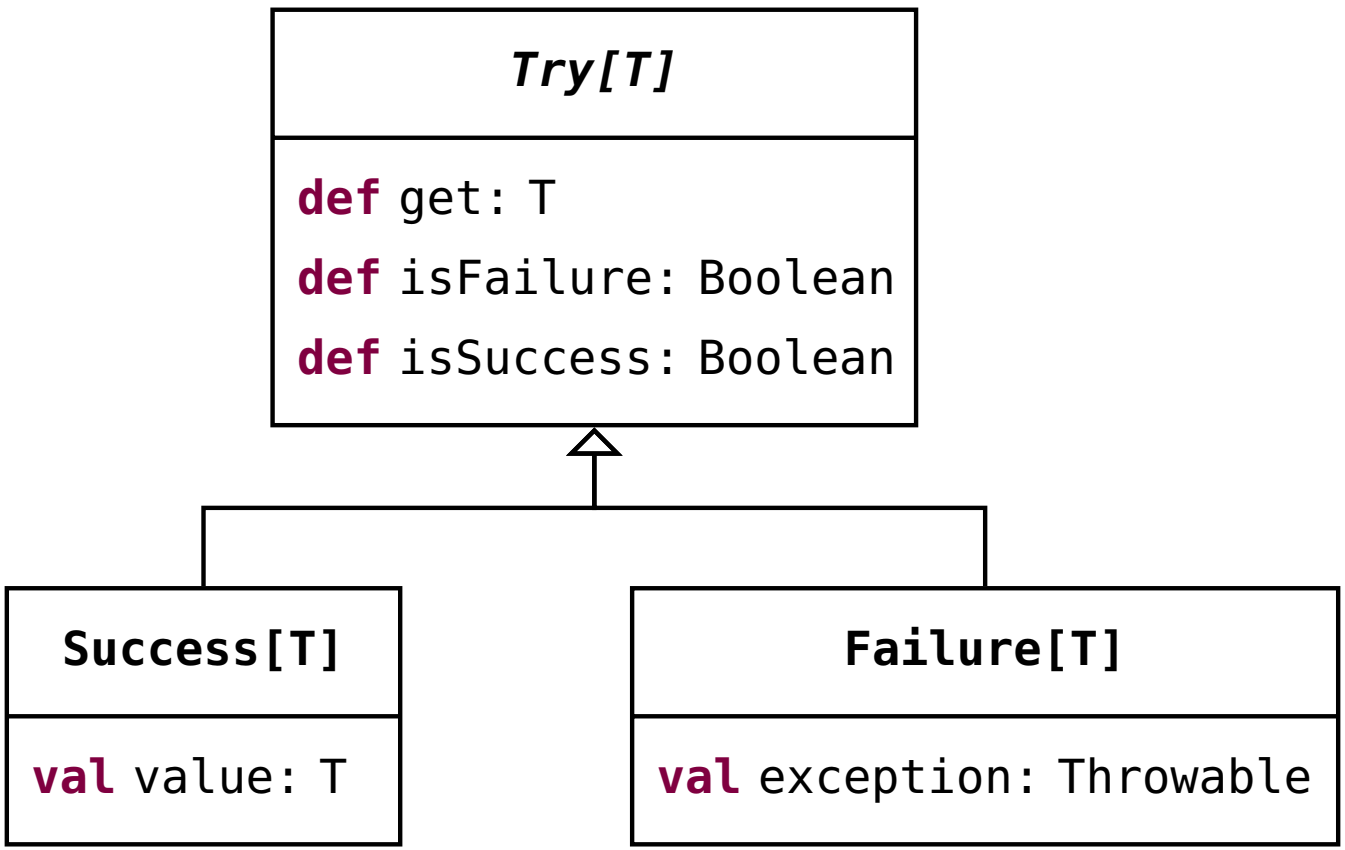
Olika sätt att hantera undantag och förhindra att exekveringen avbryts:

- **`try catch`**-uttryck omvandlar undantag till ngt lämpligt värde.
- `scala.util.Try` **kapslar in** kod som kan ge undantag.

Hantera undantag med `Try`

En gemensam bastyp för något som kan misslyckas

```
import scala.util.{Try, Success, Failure}
```



Hantera undantag med Try

```
scala> def pang = throw new Exception("PANG!")

scala> def kanskePang = if math.random() < 0.5 then 42 else pang

scala> import scala.util.{Try, Success, Failure}

scala> def försök = Try { kanskePang }

scala> val xs = Vector.fill(15){försök}

scala> val trettonde = xs(12) match
         case Success(value) => value
         case Failure(e) => println(e); -1

scala> (xs(12).isSuccess, xs(12).isFailure)

scala> xs(12).getOrElse(0)

scala> xs(12).toOption

scala> försök.foreach(println)

scala> försök.map(_ + 1)

scala> for Success(x) <- xs yield x
```

Hantera undantag med `try-catch`

try-catch-uttryck

Man kan fånga undantag med ett **try ... catch**-uttryck:

```
def carola =
  try
    if math.random() > 0.5 then throw Exception("stormvind")
    42
  catch
    case e: Exception =>
      println("Fångad av en " + e.getMessage)
      -1
```

try-catch-uttryck

Man kan fånga undantag med ett **try ... catch**-uttryck:

```
def carola =
  try
    if math.random() > 0.5 then throw Exception("stormvind")
    42
  catch
    case e: Exception =>
      println("Fångad av en " + e.getMessage)
      -1
```

```
1  scala> Vector.fill(5)(carola)
2  Fångad av en stormvind
3  Fångad av en stormvind
4  Fångad av en stormvind
5  val res0: Vector[Int] = Vector(-1, 42, 42, -1, -1)
```

För- och nackdelar med undantag

Unvik undantag om det går

Fördelar med undantag:

- Vid allvarliga fel då det inte är mycket att göra än att starta om, t.ex. `OutOfMemoryException`, är det bra att få veta vad som är fel.
- Onormala fall som uppkommer sällan kan hanteras separat (t.ex. i huvudprogrammet) utan att koden för normalfallet blir tillkrånglad.

Nackdelar med undantag:

- Ett slags ”goto” som gör exekveringsflödet svårt att följa.
- Skapa stack-trace tar tid; undantag som sker ofta påverkar prestanda.

Unvik undantag om det går

Fördelar med undantag:

- Vid allvarliga fel då det inte är mycket att göra än att starta om, t.ex. `OutOfMemoryException`, är det bra att få veta vad som är fel.
- Onormala fall som uppkommer sällan kan hanteras separat (t.ex. i huvudprogrammet) utan att koden för normalfallet blir tillkrånglad.

Nackdelar med undantag:

- Ett slags ”goto” som gör exekveringsflödet svårt att följa.
- Skapa stack-trace tar tid; undantag som sker ofta påverkar prestanda.

Exempel: undantagslösa `toIntOption` är både säker och snabb!

```
scala> def time(op: => Unit): Long = {val t0 = System.nanoTime; op; System.nanoTime - t0}

scala> def min(op: => Unit, n: Int = 1000): Long = Seq.fill(n)(time(op)).drop(n / 20).min

scala> min(util.Try("hello".toInt))
val res0: Long = 3549

scala> min(try "hello".toInt catch (_: Throwable) => ())
val res1: Long = 3046

scala> min("hello".toIntOption)
val res2: Long = 157
```

Fördjupning: Kontrollerade undantag

- Det finns möjligheter i Scala att låta kompilatorn kontrollera om undantag hanteras.
- Läs mer här: `https://docs.scala-lang.org/scala3/reference/experimental/canthrow.html`

Fördjupning: Implementera `equals`

Fördjupning: Implementera `equals` med `match`

Det visar sig att **innehållslikhet** är **förvånansvärt komplicerat** att implementera, speciellt i samband med arv.

- Det enklare fallet: Gör fördjupningsuppgift *”Metoden `equals`”* och implementera `equals` för innehållslikhet utan arv.
  En bra träning på att använda **`match`**!
- Svårare: Gör fördjupningsuppgifterna *”Överskugga `equals`”* och *”Överskugga equals vid arv”* om du vill se hur en **komplett** `equals` ska se ut som fungerar **i alla lägen**.

Det krävs i denna kurs inte att du själv ska kunna implementera en generellt fungerande `equals`. Men du ska förstå skillnaden mellan referenslikhet och innehållslikhet. Mer om `equals` i fortsättningkursen, men en liten inblick i problemet nu...

Fördjupning: `equals` som fungerar för finala klasser

Recept för implementation av `equals` som fungerar för typer som **inte** har några subtyper:

```
final class Gurka(val vikt: Int, val ärÄtbar: Boolean):
  override def equals(other: Any): Boolean = other match
    case that: Gurka => vikt == that.vikt && ärÄtbar == that.ärÄtbar
    case _ => false

  override def hashCode: Int = (vikt, ärÄtbar).## // ger bra hashcode
```

- Du **måste** alltid överskugga `hashCode` också om du överskuggar `equals` annars funkar inte gurksamlingar (lång story ...)
- Notera typen `Any` – detta följer hur man valde att göra i Java (tyvärr?).

Fördjupning: `equals` som fungerar för finala klasser

Recept för implementation av `equals` som fungerar för typer som **inte** har några subtyper:

```
final class Gurka(val vikt: Int, val ärÄtbar: Boolean):
  override def equals(other: Any): Boolean = other match
    case that: Gurka => vikt == that.vikt && ärÄtbar == that.ärÄtbar
    case _ => false

  override def hashCode: Int = (vikt, ärÄtbar).## // ger bra hashcode
```

- Du **måste** alltid överskugga `hashCode` också om du överskuggar `equals` annars funkar inte gurksamlingar (lång story ...)
- Notera typen `Any` – detta följer hur man valde att göra i Java (tyvärr?).
- Ett **typsäkrare** innehållslikhetstest som **garanterat** bara jämför en gurka med en gurka och inget annat:

```
def ===(other: Gurka): Boolean =
  vikt == other.vikt && ärÄtbar == other.ärÄtbar
```

Fördjupning: Recept i 8 steg för arvssäker `equals`

1. Inför denna metod: `def canEqual(other: Any): Boolean`
   Observera att typen på parametern ska vara `Any`. Om subklass behövs `override`.
2. Metoden `canEqual` ska ge `true` om `other` är av samma typ som `this`, t.ex.:
   `override def canEqual(other: Any): Boolean = other.isInstanceOf[Gurka]`
3. Inför metoden `equals` och var noga med att parametern har typen `Any`:
   `override def equals(other: Any): Boolean`
4. Implementera metoden `equals` med ett match-uttryck som börjar så här:
   `other match { ... }`
5. Match-uttrycket ska ha två grenar. Den första grenen ska ha ett typat mönster för den klass som ska jämföras, t.ex.:
   `case that: Gurka =>`
6. Om du implementerar `equals` i den klass som inför `canEqual`, börja med:
   `(that canEqual this) &&`
   och skapa därefter en fortsättning som baseras på innehållet i klassen, t.ex.:
   `this.vikt == that.vikt && this.längd == that.längd`
   Om du överskuggar equals vill du nog börja med `super.equals(that) &&`
7. Den andra grenen i matchningen ska vara: `case _ => false`
8. Överskugga `hashCode`, t.ex. med tupel av attributvärden och metoden ##:
   `override def hashCode: Int  = (vikt, längd).##`
   `http://www.artima.com/pins1ed/object-equality.html`

Fördjupning: Säkrare likhetstest i Scala 3

- **Problem**: `equals` tar värden av vilken typ som helst.
- Detta kallas **universell likhet**.

```
scala> case class Hund(namn: String)
scala> case class Katt(namn: String)
scala> Hund("bob") == Katt("bob") // knasig jämförelse; kan aldrig bli sant
val res0: Boolean = false         // men kompilatorn låter dig göra likhetstestet
```

- I Scala 3 kan du få typsäker likhetstest med `derives CanEqual`
- Detta kalla **multiversell likhet**.

```
scala> case class Hund(namn: String) derives CanEqual
scala> Hund("bob") == Katt("bob")   // tack kompilatorn för fel:
-- Error:
1 |Hund("bob") == Katt("bob")
  |^^^^^^^^^^^^^^^^^^^^^^^^^
  |Values of types Hund and Katt cannot be compared with == or !=
```

- Du **slipper** skriva `derives CanEqual` om du gör:
  `import scala.language.strictEquality`
- Läs mer här: `https://docs.scala-lang.org/scala3/reference/contextual/multiversal-equality.html`

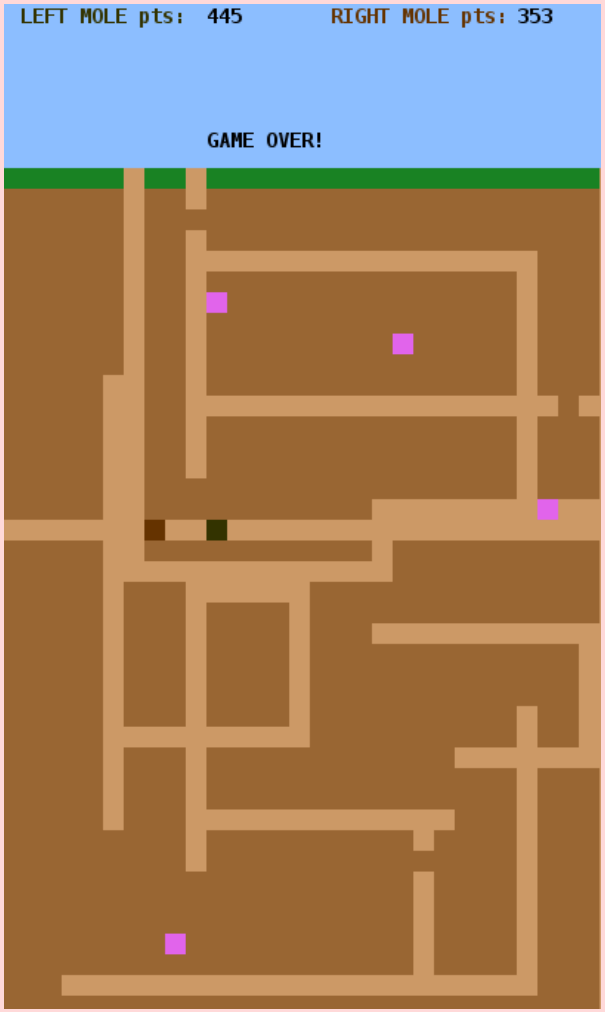
Veckans uppgifter

Veckans övning: `patterns`

Mål: Träna på matchning (och undantag)

- Uppg. 1–7: Matchning, `Option`
- Uppg. 8–10: Undantag, `Try`
- Fördjupn. 11: Matchning eller dynamisk bindning?
- Fördjupn. 12: avgöra likhet med matchning, `equals` utan arv
- Fördjupn. 13–23: diverse fördjupningsuppgifter om matchning, undantag, hash-koder, likhet vid arvshierarki

Lab `blockbattle` läsvecka 5–6



Tips på hur du kan träna på att använda mönstermatchning:

- Händelsehantering med **`match`** i stället för nästlade **`if`**
- Ändra så att `waitForKeyNonBlocking` returnerar `Option[String]`
- Låt metoderna `setBlock` och `getBlock` genererar undantag om position är utanför fönstret. Använd `require` för att åstadkomma detta.
- Använd `Try` när du anropar metoder som kan ge undantag och undvik att din app kraschar, men ge bra felmeddelande som underlättar avlusning.