Object-Oriented Programming

4. Composition, Inheritance and Encapsulation

Filip Křikava

https://courses.fit.cvut.cz/BI-OOP/

Questions?

Warmup?

Arrays in Java

- All generic types in Java are invariant and implemented using type erasure
- Except for arrays that are covariant and implemented via type reification


```
Cat[] cats = new Cat[]{ new Cat("Garfield"), new Cat("Arlene") };
Animal[] animals = cats;
animals[0] = new Dog("Odie");
Cat c = cats[0];
```

throws at runtime
ArrayStoreException(...)

- What are the consequences?
  - Java type system is unsound!
  - But, because of this exception, Java runtime is safe!
    - Very important to avert malicious attacks

Type safety and soundness

- What is type safety?
- What is a type checker?
- What is type soundness?

Type safety

- No operation will be performed on meaningless data, e.g., `42("Hello World")`
- Does not fix any implementation strategy
  - dynamically using runtime checks
  - statically by a type checker

Type checker

- A program that takes another program as input
- It makes a prediction about the program's behavior:
  - when it states that a particular term has type Number,
    it is predicting that when run, that term will produce a numeric value.
- How do we know this prediction is sound, i.e., that the type checker never lies?
- First, we need a type-system for which we can prove *soundness*.
  - If the type checker concludes that

    `e : t`

    either we can take a step `e ↦ e′` and `e′: t` or `e` is value `v : t`
  - Caveats
    - Errors (e.g., it might not even finish execution)
      - yet we could see that each step has the same type as the whole expression
    - Undecidability (e.g., some things cannot be checked until runtime)
      - array bounds, so there is almost always a list of permissible exceptions

Type soundness

- In the case of statically typed languages
- If a given expression `e` has a type `T`

  then at runtime when we run `e` and get a value `v`, it is of type `T`
- Has to be proven (typically using progress and preservation)
- The tradeoff - by typing my program I know that certain bad things won't happen
  - plus get extra documentation
  - better tooling support (code navigation, code completion, refactoring)

Types vs Safety

	Safe	Unsafe
Statically Typed	Scala, Java, Go	C, C++
Dynamically Typed	Python, Racket, JavaScript	machine code

Summary

- Make sure you understand the difference between type safety and type soundness
- Do not use the "untyped language"
  - Besides machine code, there is hardly any language that has just one type
- Try to avoid terms "strongly" / "weakly" typed language (unless you define them)
  - The problem of typing has nothing to do with strength of the type checker, but rather with the runtime that back them.
  - Pretty much we always want a safe language.
  - A language that is statically typed should ideally be sound so that the type checker does not lie.

Except for arrays, is Java Type system really sound?


```
class Unsound {

      static <T,U> U coerce(T t) {
            …
      }
      public static void main(String[] args) {
            String zero = Unsound.<Integer,String>coerce(0);
      }
}
```

Except for arrays, is Java Type system really sound?

```
class Unsound {
      static class Type<A> {
            class Constraint<B extends A> extends Type<B> {}
            <B> Constraint<? super B> bad() { return null; }
            <B> A coerce(B b) {
                  return pair(this.<B>bad(), b).value;
            }
      }
      static class Sum<T> {
            Type<T> type;
            T value;
            Sum(Type<T> t, T v) { type = t; value = v; }
      }
      static <T> Sum<T> pair(Type<T> type, T value) {
            return new Sum<T>(type, value);
      }
      static <T,U> U coerce(T t) {
            type<U> type = new Type<U>();
            return type.<T>coerce(t);
      }
      public static void main(String[] args) {
            String zero = Unsound.<Integer,String>coerce(0);
      }
}
```

- Java and Scala's Type Systems are Unsound by Nada Amin, Ross Tate, OOPSLA'16
- https://dl.acm.org/doi/10.1145/2983990.2984004

throws at runtime **ClassCastException(...)**

Encapsulation

Encapsulation

Encapsulation is about hiding the implementation, controlling access to state, and ensuring objects always stay in a valid, consistent condition.

One of the key component to battle complexity

- if something is hidden it can be changed later
- once you expose something, there is no way back and deprecating is painful!

Access modifiers - public

```
class Super:
  def f() = println("f")

class Sub extends Super:
  f()

class Other:
  new Super().f()
```

- Scala default (differs from Java)
- Public members are accessible from everywhere
  - they can be called
  - they can be overridden (unless defined `final`)

Access modifiers - protected

```
class Super:
  protected def f() = println("f")

class Sub extends Super:
  f()

class Other:
  new Super().f() // error: f is not accessible
  new Sub().f()   // error: f is not accessible
```

- Protected members are only accessible from subclasses of the defining class (differs from Java)
  - they can be called
  - they can be overridden (unless defined `final`)
- When would I want to use `final` with `protected`?
  - stop the ability to override in subclasses

Access modifiers - protected

```
class Super {
  protected def f() = { println("f") }
}

class Sub extends Super {
  override def f() = { println("g") }
}

class Other {
  new Super().f() // error: f is not accessible
  new Sub().f()   // prints "g"
}
```

- Protected members are only accessible from subclasses of the defining class (differs from Java)
  - they can be called
  - they can be overridden (unless defined `final`)
  - can broaden the access - make it public

Access modifiers - private

```
class Super:
  private def f() = { println("f") }
  f() // ok

class Sub extends Super:
  f() // symbol f is inaccessible from this point

class Other:
  new Super().f() // symbol f is inaccessible from this point
  new Sub().f()   // symbol f is inaccessible from this point
```

- Private members are accessible only from
  - inside the class or
  - companion object

Access modifiers - private

```
class Outer:
  class Inner:
    private def f() = println("f")
      f() // OK
      class InnerMost:
        f() // OK

  new Inner().f() // error: f is not accessible
```

- Private members are accessible only from
  - inside the class or
  - companion object

Access modifiers - private

```
class Super:
  private def f() = println("f")

class Sub extends Super:
  override def f() = println("g") // f overrides nothing

class Other:
  new Super().f() // symbol f is inaccessible from this point
  new Sub().f()   // symbol f is inaccessible from this point
```

- Private members are not visible outside of the defining class (trait)
  - are not inherited
  - cannot be overriden

Protecting invariants

```
class Person(var name: String, var age: Int)

val alice = Person("Alice", 30)
```

```
alice.age = -100
```

```
val bob = Person("Bob", -100)
```

- exposing the age, breaks the class invariant
- anyone can set to an arbitrary value, which breaks the invariant and breaks the domain logic

Protecting invariants

```
final class Person (val name: String, private var _age: Int):
  checkAge(_age) // check class invariant at object construction

  // public getter
  def age: Int = _age

  // public setter
  def age_=(value: Int): Unit =
    checkAge(value) // check class invariant at mutation point
    _age = value

  private def checkAge(age: Int) =  // invariant check
    require(age >= 0, s"Age cannot be negative (got $age)")
```

```
val alice = Person("Alice", 30)        // OK
val alice = Person("Alice", -1)        // RUNTIME ERROR: IllegalArgumentException: requirement
                                       //   failed: Age cannot be negative (got -1)

alice.age = -1                         // RUNTIME ERROR: IllegalArgumentException: requirement
                                       //   failed: Age cannot be negative (got -1)
```

Protecting invariants

```
import java.util.Date

final class Person (val name: String, private var _dob: Date):
  checkAge(_dob) // check class invariant at object construction

  // public getter
  def dateOfBirth: Date = _dob
  def age: Int = computeAge(_dob)

  // public setter
  def dateOfBirth_=(value: Date): Unit =
    checkAge(value) // check class invariant at mutation point
    _dob = value

  private def checkAge(dob: Date) = require(computeAge(dob) >= 0)  // invariant check
  private def computeAge(dob: Date) = ???
```

```
val alice = Person("Alice", ymd(2024, 11 , 6)) // OK
alice.age() // OK: 1
alice.dateOfBrith.setTime(ymd(2026, 11 , 6).getTime()) // OK
alice.age() // ERROR: -1
```

- the problem is that Date class is mutable
- with public getter anyone can set to an arbitrary value, which breaks the invariant and breaks the domain logic

https://scastie.scala-lang.org/20fn37KaS9uiXaOCyZo0aQ

Protecting invariants

```
import java.util.Date

final class Person (val name: String, private var _dob: Date):
  checkAge(_dob) // check class invariant at object construction

  // public getter returning a copy
  def dateOfBirth: Date = new Date(_dob.getTime())
  def age: Int = computeAge(_dob)

  // public setter
  def dateOfBirth_=(value: Date): Unit =
    checkAge(value) // check class invariant at mutation point
    _dob = value

  private def checkAge(dob: Date) = require(computeAge(dob) >= 0)  // invariant check
  private def computeAge(dob: Date) = ???
```

```
val alice = Person("Alice", ymd(2024, 11 , 6)) // OK
alice.dateOfBrith.setTime(ymd(2026, 11 , 6))   // OK, changes the copy, not the _dob field in alice
alice.age() // OK: 1
```

Protecting invariants

```
import java.time.LocalDate

final class Person (val name: String, private var _dob: LocalDate):
  checkAge(_dob) // check class invariant at object construction

  // public getter returning a reference, LocalDate is immutable
  def dateOfBirth: LocalDate = _dob
  def age: Int = computeAge(_dob)

  // public setter
  def dateOfBirth_=(value: LocalDate): Unit =
    checkAge(value) // check class invariant at mutation point
    _dob = value

  private def checkAge(dob: LocalDate) = require(computeAge(dob) >= 0)  // invariant check
  private def computeAge(dob: LocalDate) = ???
```

- instead of Date, use LocalDate which is immutable

Hiding internal collections

```
import scala.collection.mutable.ArrayBuffer

final class Role(val name: String, var active: Boolean)

final class Person(val name: String):
  private val _roles: ArrayBuffer[Role] = ArrayBuffer()

  def addRole(role: Role): Unit =
    checkRole(role)
    _roles += role

  def roles: ArrayBuffer[Role] = _roles
```

```
val p = Person("Alice")
p.addRole(Role("adnim", true))

p.roles += Role("admin", false) // Anyone can manipulate roles directly, breaks the invariant
```

Hiding internal collections

```
import scala.collection.mutable.ArrayBuffer

final class Role(val name: String, var active: Boolean)

final class Person(val name: String):
  private val _roles: ArrayBuffer[Role] = ArrayBuffer()

  def addRole(role: Role): Unit =
    checkRole(role)
    _roles += role

  def roles: Seq[Role] = _roles.toSeq
```

```
val p = Person("Alice")
p.addRole(Role("adnim", true))

p.roles += Role("admin", false) // COMPILE ERROR
```

https://scastie.scala-lang.org/yIxuEcoEQ3OqA2hJNwHOqw

Hiding internal collections

```
import scala.collection.mutable.ArrayBuffer

final class Role(val name: String, var active: Boolean)

final class Person(val name: String):
  private val _roles: ArrayBuffer[Role] = ArrayBuffer()

  def addRole(role: Role): Unit =
    checkRole(role)
    _roles += role

  def roles: Seq[Role] = _roles.toSeq
```

```
val p = Person("Alice")
p.addRole(Role("adnim", true))
p.roles                                                 // Seq(Role(admin, true))
```

```
p.roles.find(_.name == "adnim").foreach(_.active = false)
p.roles                                                 // Seq(Role(admin, false))
```

- We only return shallow copy of roles, need to be careful as it might possibly expose internal state as well

Prefer opaque types

```
def schedule(delay: Long): Unit = …


schedule(5) // is that ms? seconds? frames?
```

```
object Time:
  opaque type Millis = Long

  object Millis:
    def apply(n: Long): Millis =
      require(n >= 0); n

  extension (m: Millis) def toLong: Long = m

def schedule(delay: Time.Millis): Unit = ...

schedule(Time.Millis(5))
```

- Encodes meaning
- Enforces invariants
- No runtime cost
- Keeps representation private

Use companion objects for controlled construction

```
final class Email(val value: String)
```

- Immutable, but no validation

```
final class Email private (val value: String)

object Email:
  // don't worry too much about the regex
  private val Rx = "^[^@]+@[^@]+\\.[^@]+$".r

  def apply(s: String): Option[Email] =
    s match
      case Rx(_*) => Some(new Email(s))
      case _      => None
```

- Representation is hidden
- Enforces invariants
- Can normalize values

Use companion objects for controlled construction

```
final class Email(val value: String)
```

- Immutable, but no validation

```
final class Email private (val value: String)

object Email:
  //  RFC 5322
  private val Rx =
"""(?i)\A(?:[a-z0-9!#$%&'*+\x2f=?^_`\x7b-\x7d~\x2d]+(?:\.[a-z0-9!
#$%&'*+\x2f=?^_`\x7b-\x7d~\x2d]+)*|"(?:[\x01-\x08\x0b\x0c\x0e-\x1
f\x21\x23-\x5b\x5d-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])*")@(?:(?
:[a-z0-9](?:[a-z0-9\x2d]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9\x2d]*[a
-z0-9])?|\[(?:(?:(2(5[0-5]|[0-4][0-9])|1[0-9][0-9]|[1-9]?[0-9]))\
.){3}(?:(2(5[0-5]|[0-4][0-9])|1[0-9][0-9]|[1-9]?[0-9])|[a-z0-9\x2
d]*[a-z0-9]:(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]|\\[
\x01-\x09\x0b\x0c\x0e-\x7f])+)\])\z""".r

  def apply(s: String): Option[Email] =
    s match
      case Rx(_*) => Some(new Email(s))
      case _      => None
```

- Representation is hidden
- Enforces invariants
- Can normalize values

Encapsulation - Actions

- Hide internal representation
  - Internal data structures and state should not be directly exposed.
  - Clients should not depend on how data is stored, only on what the object does.
- Control access through a stable interface
  - Provide methods or properties that define the allowed interactions.
  - The interface is the contract; the implementation can change without breaking users.
- Preserve invariants
  - Objects should always remain in a valid state.
  - All modification points must enforce invariants (e.g., age ≥ 0, sorted list stays sorted).
- Protect against incorrect external mutation
  - Never expose mutable state directly (e.g., arrays, buffers, maps).
  - Return copies, immutable views, or controlled adapters instead.
- Keep data consistent by centralizing Logic
  - Logic that updates internal state should live in the class, not scattered across the program.
  - The object knows how to maintain itself correctly.
- Minimize the public API
  - Only expose what is necessary to use the object.
  - Everything else should be private or protected.

Encapsulation - Actions

- Place data and the operations that perform on that data in the same class
- Use responsibility-driven design to determine the grouping of data and operations into classes
  - high cohesion
  - low coupling
- Don't expose data items
- Don't expose the difference between stored data and derived data
- Don't expose a class's internal structure
- Don't expose implementation details of a class
- Always think whether you need mutability, often you do not

[1] https://www.infoworld.com/article/2075271/encapsulation-is-not-information-hiding.html

Encapsulation - Benefits

- Encapsulation enables refactoring
  - Because implementation details are hidden, they can change without requiring changes in clients.
  - This is crucial for evolving and maintaining codebases.

- Improves understandability
  - Users of a class do not need to understand how it works internally.
  - They only need to know what it does.

- Encapsulation enables stronger abstractions
  - Once representation is hidden, you can introduce richer behavior (caching, lazy evaluation, validation) without impacting callers.

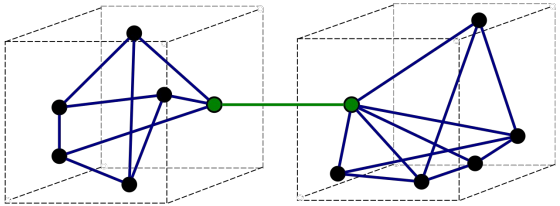
- Encapsulation supports testing and debugging
  - Invariants and controlled mutation points make bugs easier to track.
  - Well-encapsulated classes behave more predictably.

Cohesion and coupling

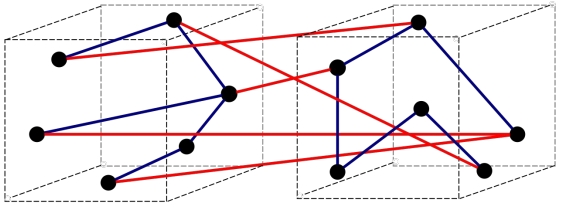
- Cohesion
  - how much elements inside a module belong together?
  - small methods, related activities
- Coupling
  - what is the interdependence between modules?

- Aim at high cohesion and low coupling

- Why?
  - A change in one module usually forces a ripple effect of changes in other modules.
  - Increased effort to assemble dependent modules
  - Harder to reuse and/or test because dependent modules must be included.



a) Good (loose coupling, high cohesion)



b) Bad (high coupling, low cohesion)

Algebraic Data Types

Motivation

- You want to create an encapsulated class that represent a role in a system
- A role has a name and a flag indicating whether it is active or not

```java
import java.util.Objects;

public class Role {
    private String name;
    private boolean active;

    public Role(String name, boolean active) {
        this.name = name;
        this.active = active;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public boolean isActive() {
        return active;
    }

    public void setActive(boolean active) {
        this.active = active;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Role role = (Role) o;
        return active == role.active && Objects.equals(name, role.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, active);
    }

    @Override
    public String toString() {
        return "Role{" +
                "name='" + name + '\'' +
                ", active=" + active +
                '}';
    }
}
```

```scala
class Role(val name: String, active: Boolean) {

  private def canEqual(other: Any): Boolean =
    other.isInstanceOf[Role]

  override def equals(other: Any): Boolean =
    other match {
      case that: Role =>
        that.canEqual(this) && name == that.name
      case _ => false
    }

  override def hashCode(): Int = {
    val state = Seq(name)
    state.map(_.hashCode())
         .foldLeft(0)((a, b) => 31 * a + b)
  }

  override def toString = s"Role($name, $active)"
}

object Role {
  def apply(name: String, active: Boolean): Role =
    new Role(name, active)
}
```

Case class

The val keyword is implicit in case classes

```
final case class Role(name: String, active: Boolean)
```

```
final class Role(val name: String, val active: Boolean) {
  def toString: String = ...
  def equals(x: Any): Boolean = ...
  def hashCode(x: Any): Int = ...
  def copy(
    name: String = this.name,
    active: Boolean = this.active
  ): Role = ...

}

object Role {
  def apply(name: String, active: Boolean): Role = ...
  def unapply(c: Role): Option[(String, Boolean)] = ...
}
```

- Model plain data aggregates with less ceremony
- Scala's take on product part of Algebraic Data Types
- Immutable (strongly recommended)
- Should not be extended (make them final)
- Next to a new type definition, the following gets generated
  - `toString()`
  - `equals()/hashCode()`
  - `copy()`
  - `product*()`
  - a factory method
  - extractor for pattern matching

Case class

- Model ordinary data aggregates
- Essentially a nominal tuple: a transparent, shallowly immutable carrier for a specific ordered sequence of elements
- They provide
  - a type
  - a constructor
  - `toString` implementation
  - `equals`, `hashCode` implementation
  - `copy` implementation
  - `product*` implementation (case class implements `Product`)

Case classes - summary

- Non-encapsulated data structures
- Useful for data modelling
- They play especially well with pattern matching

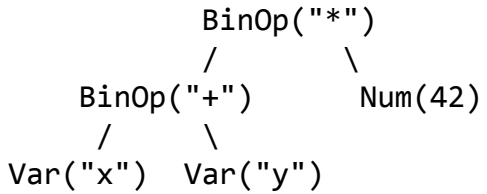
- In Scala the way to do Algebraic Data Types
  - Essentially types formed by combining other types
  - Aggregates several concepts in a new type
  - Product (tuple / record)
    - all values is the cartesian product of the sets of all possible values of its field
    - a case class in Scala
  - Sum (tagged / disjoint union)
    - all values is the disjoint union of the sets of all possible values of its variants
    - a sealed trait in Scala

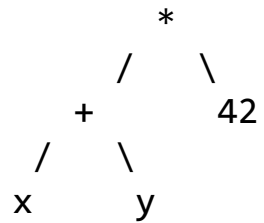
Algebraic Data Types

```
sealed abstract class Expr

final case class Num(num: Double) extends Expr
final case class Var(name: String) extends Expr
final case class BinOp(op: String, left: Expr, right: Expr) extends Expr


val expr = BinOp("*", BinOp("+", Var("x"), Var("y")), Num(42))  // (x + y) * 42
```



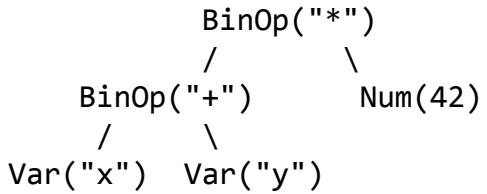


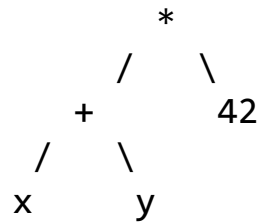
Algebraic Data Types - Scala 3 syntax

```
enum Expr:
  case Num(num: Double)
  case Var(name: String)
  case BinOp(op: String, left: Expr, right: Expr)


import Expr._

val expr = BinOp("*", BinOp("+", Var("x"), Var("y")), Num(42))  // (x + y) * 42
```





Pattern Matching

Pattern matching on values

```
val x = 3

val result = x match
  case 1 => "one"
  case 2 => "two"
  case 3 => "three"
  case _ => "something else"

println(result) // "three"
```

Pattern matching on values

```
def describe(x: Any) =
  x match {
    case 5 => "five"
    case true => "truth"
    case "hello" => "hi!"
    case Nil => "the empty list"
    case _ => "something else"

describe(5) // five
```

Pattern matching on values

```
def describe(x: Any) =
  x match {
    case 5 => "five"
    case true => "truth"
    case "hello" => "hi!"
    case Nil => "the empty list"
    case _ => "something else"

describe(5) // five
```

Pattern matching on types

```
def describe(value: Any): String =
  value match
    case n: Int     =>  s"Int: $n"
    case s: String  =>  s"String: $s"
    case b: Boolean =>  s"Boolean: $b"
    case other      =>  s"Other: $other"

describe(10)      // Int: 10
describe("hello") // String: hello
```

Pattern matching on ADT

```
def onOption(x: Option[Int]) =
  x match
    case Some(x) => s"has value $x"
    case None    => "no value"

onOption(Some(1)) // has value 1
```

Pattern matching on ADT

```
def onOption(x: Option[Int]) =
  x match
    case Some(x) if x > 10 => s"has value $x > 10"
    case None    => "no value"

onOption(Some(11)) // has value 11 > 10
```

```
Warning:
  Line 2: match may not be exhaustive.
          It would fail on pattern case: Some(_)
```

Pattern matching with guards

```
def onOption(x: Option[Int]) =
  x match
    case Some(x) if x > 10 => s"has value $x > 10"
    case Some(_) => "has some other value"
    case None    => "no value"

onOption(Some(11)) // has value 11 > 10
```

Pattern matching with guards

```
def onOption(x: Option[Int]) =
  val A = 10
  x match
    case Some(A) => "has value 10"
    case Some(a) => s"has some other value $a"
    case None    => "no value"

onOption(Some(10)) // has value 10
onOption(Some(11)) // has some other value 11
```

Pattern matching and subtyping

```
class A
class B extends A
class C extends B

val a: A = C()

a match                  // 1
  case x: A => 1
  case x: B => 2
  case x: C => 3

a match
  case x: C => 3         // 3
  case x: B => 2
  case x: A => 1
```

```
Warning: Unreachable case
Warning: Unreachable case
```

Pattern matching on lists

```
def sum(xs: List[Int], acc: Int = 0) =
  xs match
    case Nil     => acc
    case x :: xs => sum(xs, acc + x)

sum(List(1, 2, 3))
sum(1 :: 2 :: 3 :: Nil)
```

The `::` has multiple meanings in this case:

- In `x :: xs` it represents a type

  ```
  final case class ::[+A](head: A, var next: List[A]) extends List[A]
  ```

- In `1 :: 2 :: 3 :: Nil` it represents a right-associative method defined on `List`

  ```
  def ::[B >: A](elem: B): List[B]
  ```

Pattern matching on sequences

```
val xs = List(Num(1), Num(2), Var("x"))

xs match
  case Num(1) :: Num(x) :: _ :: Nil => s"Has three elements, two numbers, the second is ${x}"
  case Num(1) :: _ :: _ :: Nil      => s"Three elements list"
  case Num(1) :: _ :: _ :: _        => s"At least 3 elements list"
  case Nil                          => "empty list"
  case _                            => "some list"


xs match
  case Seq(Num(1), Num(x), _) => s"Has three elements, two numbers, the second is $x"
  case Seq(Num(1), _, _)      => s"Three elements seq"
  case Seq(Num(1), _, _, _*)  => s"At least 3 elements seq"
  case Seq()                  => "empty seq"
  case _                      => "some seq"
```

Pattern matching everywhere

```
val pair = (Some(1), (2, Some(2)))
val (Some(a), (b, Some(c))) = pair

val xs = List(Some(1), Some(2), Some(3))

for Some(x) <- xs do
  println(x)

val capitals = Map("Czechia" -> "Prague", "Japan" -> "Tokyo")
for ((country, city) <- capitals)
  println("The capital of " + country + " is " + city)


val results = List(Some("apple"), None, Some("orange"))
for (Some(fruit) <- results) println(fruit)
```

```
Error: pattern's type Some[String] is more specialized than the right
hand side expression's type Option[String]
```

Pattern Matching - variable binding

```
val expr = BinOp("+", BinOp("*", Num(1), Num(2)), Num(3))

expr match
  case BinOp("+", left @ BinOp(_, Num(1), _), right) =>
    s"LHS of the + is a binary op ${left.op} with ${left.right} RHS"
  case _ =>
    "something else"

// LHS of the + is a binary op * with Num(2.0) RHS
```

Pattern matching outside Scala

- Pattern matching is extremely useful
- Traditionally present in FP languages
- But today many mainstream languages have that
  - being backported to languages
    - Java 17+ records and switch expression and binding instanceof
    - Python 3.10+ match / case structural pattern matching
    - C++ with P1371, P2688, ...