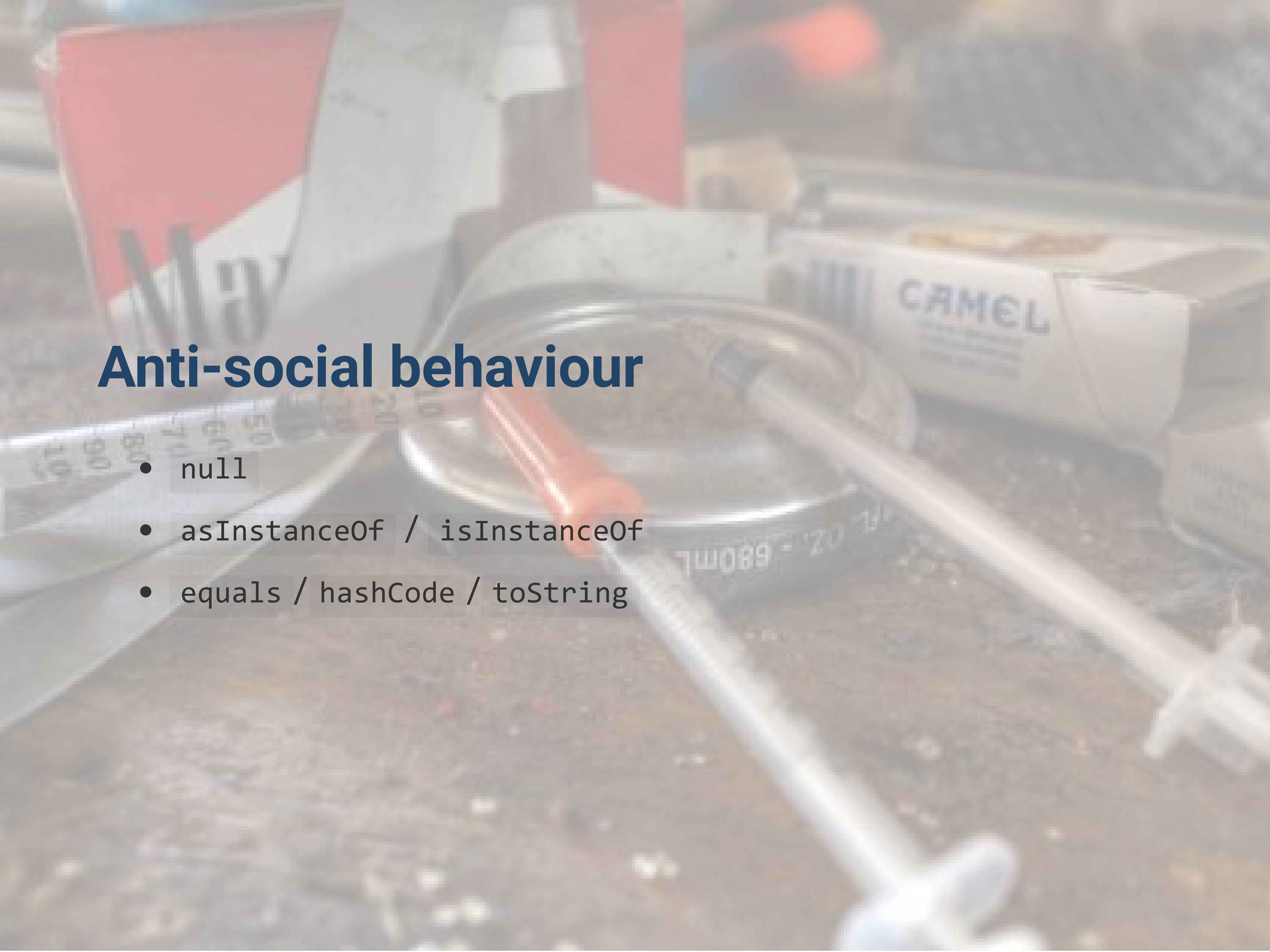
Types Inhabitants

Anti-social behaviour is prohibited



Anti-social behaviour

- `null`
- `asInstanceOf` / `isInstanceOf`
- `equals` / `hashCode` / `toString`

Anti-social behaviour

- `throw`
- runtime reflection
- side-effects

No `equals`???

Unsafe

```
def update[A](before: A, after: A): Option[A] =
  if (before == after) None else Some(after)
```

No `equals`???

Safe: Scalaz

```
import scalaz._, Scalaz._

def update[A](before: A, after: A)
             (implicit e: Equal[A]): Option[A] =
  if (e.equal(before, after)) None else Some(after)

def update[A: Equal](before: A, after: A): Option[A] =
  if (before === after) None else Some(after)
```

No `equals`???

Safe: UnivEq

```
import japgolly.univeq._

def update[A: UnivEq](before: A, after: A): Option[A] =
  if (before == after) None else Some(after)
```

Inhabitants

Boolean

?

Boolean

$2 = \{True, False\}$

Byte

?

Byte

$2^8 = 256$

Char

?

Char

$2^{16} = 65,536$

Int

?

Int

$2^{32} = 4,294,967,296$

String

?

String

∞ (bound by memory in practice)

Unit

?

Unit

1: `()`

Singleton / `object`

?

Singleton / `object`

1

Nothing

?

Nothing

0

Generic `A`

Generic `A`

```
def gimme[A]: A
```

Generic `A`

0 - without access

Can only reference, not **create**.

```
// 0
def nope[A]: A

// 1
def one[A](a: A): A

// 2
def two[A](a1: A, a2: A): A
```

Generic `A`

Cheating:

```
def gimme[A]: A = ???

def gimme[A]: A = 0.asInstanceOf[A]

def gimme[A >: Null]: A = null

def gimme[A <: AnyRef]: A = null
```

ADT time!

Sum types (coproducts)

```haskell
-- Haskell
data Either a b = Left a | Right b
```

```scala
// Scala
sealed trait Either[+A, +B]
case class Left [+A](value: A) extends Either[A, Nothing]
case class Right[+B](value: B) extends Either[Nothing, B]
```

Sum types (coproducts)

$a + b$

Sum types (coproducts)

Either[Unit, Unit] = 1 + 1 = 2

```
Left(())
Right(())
```

Sum types

Either[Boolean, Boolean] = 2 + 2 = 4

```
Left(true)
Left(false)
Right(true)
Right(false)
```

`Option[A]`

?

`Option[A]`

$1 + a$

```
data Option a = None | Some a
```

Products

Tuples.

```
(A, B)
```

Products

a.b

Products

(Unit, Unit) = 1 x 1 = 1

```
((), ())
```

(Boolean, Boolean) = 2 x 2 = 4

```
(true , true)
(false, true)
(false, false)
(true , false)
```

Functions

```
A => B
```

Functions

```
A => B
```

b^a

Functions

`Boolean => Boolean` : $2^2 = 4$

- `a => a`
- `a => !a`

Functions

`Boolean => Boolean` : $2^2 = 4$

- `a => a`
- `a => !a`
- `_ => true`
- `_ => false`

Functions

`Unit => Boolean` : $2^1 = 2$

- `_ => true`
- `_ => false`

Functions

`Boolean => Unit` : $1^2 = 1$

- `_ => ()`

Functions

`Option[Boolean] => (Boolean, Boolean)` : ?

Functions

`Option[Boolean] => (Boolean, Boolean)` : $4^3 = 64$

Functions

Side note: `xmap`

```
class Omg[A] {
  def xmap[B](f: A => B)(g: B => A): Omg[B]
}
```

```scala
case class JsonFormat[A](
  read : JsValue => JsResult[A],
  write: A       => JsValue) {

  def xmap[B](f: A => B)(g: B => A): JsonFormat[B] =
    JsonFormat[B](read.andThen(_ map f), write compose g)
}
```

```scala
case class UserId(value: Long) extends AnyVal

object UserId {
  implicit val jsonFormat: JsonFormat[UserId] =
    implicitly[JsonFormat[Long]].xmap(UserId)(_.value)
}
```

ADTs

Algebraic data types

ADTs

```scala
sealed trait Thing[+A]

case class Stuff[A](value: A) extends Thing[A]

case object NoStuff extends Thing[Nothing]

case class OtherStuff[+A](enabled: Boolean, o: Option[A])
  extends Thing[A]
```

ADTs

```scala
sealed trait Thing[+A]

case class Stuff[A](value: A) extends Thing[A]

case object NoStuff extends Thing[Nothing]

case class OtherStuff[+A](enabled: Boolean, o: Option[A])
  extends Thing[A]
```

Thing[A] = A + 1 + (2 x (1 + A))

Thing[A] = A + 1 + 2 + 2A

Thing[A] = 3A + 3

Thing[Unit] = 6

Thing[Boolean] = 9

Isomorphisms

Isomorphisms exist where inhabitant counts match
(and no side-effects are involved).

Boolean ≅ Either[Unit, Unit]

```
true  = Left(())
false = Right(())
```

Boolean ≅ Either[Unit, Unit]

```
true  = Right(())
false = Left(())
```

Either[Boolean, Boolean] ≅ (Boolean, Boolean)

```
Left (true ) = (true , true )
Left (false) = (true , false)
Right(true ) = (false, true )
Right(false) = (false, false)
```

Currying

Types

- `(A, B) ⇒ C`
- `(B, A) ⇒ C`
- `A ⇒ (B ⇒ C)`

Inhabitants

- c^{ab}
- c^{ba}
- $(c^b)^a$

Currying

Types

- `A ⇒ (B ⇒ C)`
- `B ⇒ (A ⇒ C)`

Inhabitants

- $(c^b)^a$
- $(c^a)^b$

Why is this interesting?

Use the isomorphism to craft more convenient shapes.

Use the isomorphism to transform shapes.

The big one...

Use to reduce the problem & solution space.

Make illegal states unrepresentable.

When we write code...

- Usually only one correct implementation.
- Therefore, N-1 incorrect implementations are possible.
- Reducing N reduces chance of error.

Bonus:

- Less mental effort required of devs.
- Safer refactoring and future extension.
- Less testing required; compile-time > runtime.

Examples

(Just two.)

Example #1. Emptiness

```
// Seq[Item] = [0 .. ∞]
def createItemsApi(items: Item*): Result =
```

Usually ok.

Maybe API call is expensive. Network cost...

You could (provably) prevent useless calls:

```
// (Item, Seq[Item]) = [1ᵢ .. ∞]
def createItemsApi(item1: Item, items: Item*): Result =
```

Use Scalaz or Cats, or create your own...

```scala
case class NonEmptyList[+A](head: A, tail: List[A])
```

```scala
case class OneAnd[F[_], A](head: A, tail: F[A])

type NonEmptyList  [A] = OneAnd[List  , A]
type NonEmptyVector[A] = OneAnd[Vector, A]
```

0+

```scala
case class Rule(targets: List[Target])
```

1+

```scala
case class Rule(target1: Target, targetN: List[Target]) {
  def targets: List[Target] =
    target1 :: targetN
}
```

```scala
case class Rule(targets: NonEmptyList[Target])
```

Example #2: Validation

```
case class Validator(
  validate: String => String \/ String)
```

```
val v = Validator { s =>
  val corrected = s.toUpperCase.trim.replaceAll(" +", " ")
  if (corrected.isEmpty)
    -\/("Name required.")
  else
    \/-(corrected)
}
```

```
v.validate(" david   barri") // \/-("DAVID BARRI")
v.validate("  ")             // -\/("Name required.")
```

Example #2: Validation

```
case class Validator(
  validate: String => String \/ String)
```

```
def validateTwo(v1: Validator,
                v2: Validator,
                s1: String,
                s2: String
              ): String \/ (String, String)
```

```scala
def validateTwo(v1: Validator,
                v2: Validator,
                s1: String,
                s2: String
               ): String \/ (String, String)
```

```
val validate = ∞ + ∞²

vn: (∞ + ∞²)
sn: ∞

inputs = 2∞(∞ + ∞²)
       = 2∞³ + 2∞²
output = (∞ + ∞²)

validateTwo = (∞ + ∞²) ^ (2∞³ + 2∞²)
```

```scala
case class Validator[Input, Validated, Error](
  validate: Input => Error \/ Validated)
```

```scala
def validateTwo[E, I1, V1, I2, V2](
    v1: Validator[I1, V1, E],
    v2: Validator[I2, V2, E],
    i1: I1,
    i2: I2
  ): E \/ (V1, V2)
```

```scala
case class Validator[Input, Validated, Error](
  validate: Input => Error \/ Validated)
```

```scala
def validateTwo[E, I1, V1, I2, V2](
    v1: Validator[I1, V1, E],
    v2: Validator[I2, V2, E],
    i1: I1,
    i2: I2
  ): E \/ (V1, V2)
```

Result inhabitants:

```
E = 2, V1 = 1, V2 = 1

= 2 \/ (1, 1)
= 2 + 1
= 3
```

```scala
def validateTwo[E, I1, V1, I2, V2](
    v1: Validator[I1, V1, E],
    v2: Validator[I2, V2, E],
    i1: I1,
    i2: I2): E \/ (V1, V2) =
  for {
    result1 <- v1.validate(i1)
    result2 <- v2.validate(i2)
  } yield (result1, result2)
```

- Impossible to pass input to wrong validator.
- If both validations pass, there is no `E`.
  Result can **only** be `\/-(V1,V2)`
- If either validation fails, `\/-(V1,V2)` is impossible.

Only two possible implementations will compile:

```
for {
  result1 <- v1.validate(i1)
  result2 <- v2.validate(i2)
} yield (result1, result2)
```

```
for {
  result2 <- v2.validate(i2)
  result1 <- v1.validate(i1)
} yield (result1, result2)
```

When *both* validations fail, should the v1 or v2 error be returned?

Constraints liberate.
Liberties constrain.

-- Runar Bjarnason

https://www.youtube.com/watch?v=GqmsQeSzMdw

Constraints liberate.
Liberties constrain.

- String version
  - **Liberty:** Infinite possible return values.
  - **Liberty:** Infinite possible implementations.
  - **Constraint:** One 1 way to use.

Constraints liberate.
Liberties constrain.

- String version
  - **Liberty:** Infinite possible return values.
  - **Liberty:** Infinite possible implementations.
  - **Constraint:** One 1 way to use.
- Generic version
  - **Constraint:** Only 3 possible return values.
  - **Constraint:** Only 2 possible implementations.
  - **Liberty:** Infinite ways to use.

Thank you!