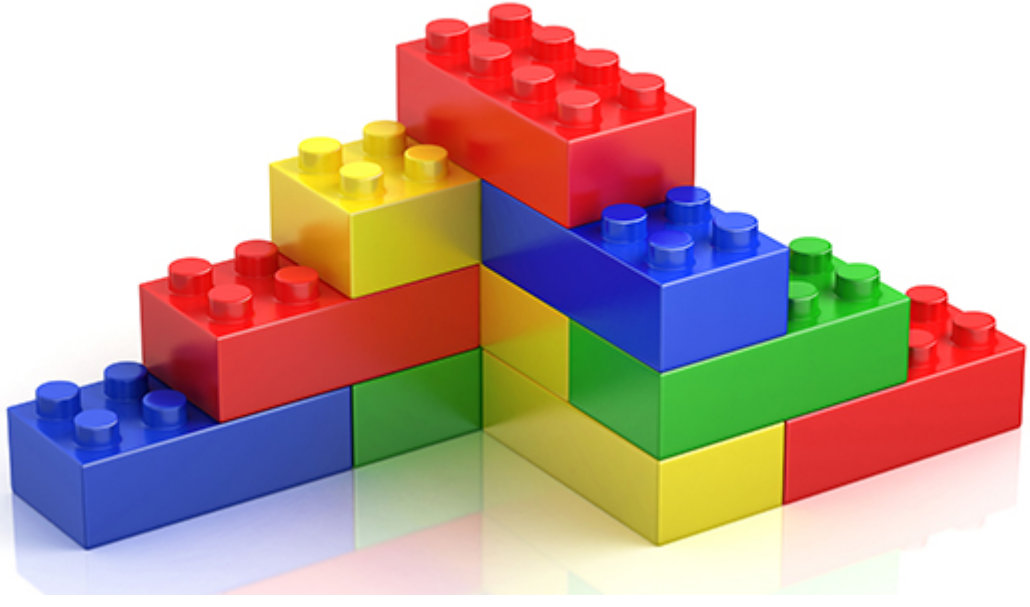
PRACTICAL TYPE SAFETY



Jacob Wang

HELLO

- Scala Developer at 

- @jatcwang

THIS TALK

- **Type Safety** - What and why?
- Principles
- Practical techniques for Scala

STATIC TYPES & TYPE SAFETY

STATIC TYPES & TYPE SAFETY

- Static types
  - Prevent incorrect behaviour, without running the program

STATIC TYPES & TYPE SAFETY

- Static types
  - Prevent incorrect behaviour, without running the program
- Type Safety is “how much” we take this approach

THE GOAL

What do we want as software developers?

THE GOAL

What do we want as software developers?

- Write **correct** & **maintainable** software **faster**!

THE GOAL

What do we want as software developers?

- Write **correct** & **maintainable** software **faster**!
- I believe that a language with good **static type systems** really helps!

THE GOAL

What do we want as software developers?

- Write **correct** & **maintainable** software **faster**!
- I believe that a language with good **static type systems** really helps!
  - Tooling: Autocomplete, code browsing, refactoring

THE GOAL

What do we want as software developers?

- Write **correct** & **maintainable** software **faster**!
- I believe that a language with good **static type systems** really helps!
  - Tooling: Autocomplete, code browsing, refactoring
  - Understanding the code

THE GOAL

What do we want as software developers?

- Write **correct** & **maintainable** software **faster**!
- I believe that a language with good **static type systems** really helps!
  - Tooling: Autocomplete, code browsing, refactoring
  - Understanding the code
  - Design & Prototyping

BUT..

BUT..

- Consider the trade-offs

BUT..

- Consider the trade-offs
  - Ergonomics (in the language you're using)

BUT..

- Consider the trade-offs
  - Ergonomics (in the language you’re using)
  - Implementation Complexity / Time

BUT..

- Consider the trade-offs
  - Ergonomics (in the language you’re using)
  - Implementation Complexity / Time
  - Performance

BUT..

- Consider the trade-offs
  - Ergonomics (in the language you’re using)
  - Implementation Complexity / Time
  - Performance
  - Is it understandable?

BUT..

- Consider the trade-offs
  - Ergonomics (in the language you're using)
  - Implementation Complexity / Time
  - Performance
  - Is it understandable?
  - Compile time

PRINCIPLES & TECHNIQUES

I. GUARD AGAINST CHANGE

I. GUARD AGAINST CHANGE

- Code is written once, and changed many times

I. GUARD AGAINST CHANGE

- Code is written once, and changed many times
  - Change in requirement

I. GUARD AGAINST CHANGE

- Code is written once, and changed many times
  - Change in requirement
  - Change in code architecture (refactoring)

I. GUARD AGAINST CHANGE

- Code is written once, and changed many times
  - Change in requirement
  - Change in code architecture (refactoring)
- **Want**: Compile errors where our previous assumptions no longer holds

I. GUARD AGAINST CHANGE

- Code is written once, and changed many times
  - Change in requirement
  - Change in code architecture (refactoring)
- **Want**: Compile errors where our previous assumptions no longer holds
  - Prompt us to reconsider the logic

I. GUARD AGAINST CHANGE

- Code is written once, and changed many times
  - Change in requirement
  - Change in code architecture (refactoring)
- **Want**: Compile errors where our previous assumptions no longer holds
  - Prompt us to reconsider the logic
- “If I change a basic assumption I’m making here, will the compiler tell me?”

I. GUARD AGAINST CHANGE

Avoid catch-alls in pattern matching

```
sealed trait User {
  def name: String
}
final case class Guest(name: String) extends User
final case class Member(name: String) extends User
final case class Admin(name: String, level: Int) extends User

def canEdit(role: User): Boolean = {
  role match {
    case _: Guest => false
    case _      => true
  }
}
```

I. GUARD AGAINST CHANGE

Avoid catch-alls in pattern matching

```
sealed trait User {
  def name: String
}
final case class Guest(name: String) extends User
final case class Member(name: String) extends User
final case class Admin(name: String, level: Int) extends User

def canEdit(role: User): Boolean = {
  role match {
    case _: Guest => false
    case _     => true
  }
}
```

Bug if we add another role that isn’t allowed to edit!

I. GUARD AGAINST CHANGE

Avoid catch-alls in pattern matching

```
def canEditV2(role: User): Boolean = {
  role match {
    case _: Guest => false
    case u @ (_: Member | _: Admin) => {
      // u is a User
      // useful if you want to access fields on User class
      true
    }
  }
}
```

I. GUARD AGAINST CHANGE

Avoid default parameters

I. GUARD AGAINST CHANGE

Avoid default parameters

```scala
// Version 1
final case class User(name: String, age: Int)

// Version 2
final case class User(name: String, age: Int, isAdmin: Boolean = false)
```

I. GUARD AGAINST CHANGE

Avoid default parameters

```
// Version 1
final case class User(name: String, age: Int)

// Version 2
final case class User(name: String, age: Int, isAdmin: Boolean = false)
```

```
def createFromLegacyUser(legacyUser: LegacyUser): User = {
  User(legacyUser.name, legacyUser.age)
}
```

I. GUARD AGAINST CHANGE

Avoid default parameters

Use explicitly named values/functions

```
final case class User(name: String, age: Int, isAdmin: Boolean)

object User {
  val defaultUser: User = ...
  // or
  def notAdmin(name: String, age: Int, isAdmin: Boolean = true): User
}
```

I. GUARD AGAINST CHANGE

Avoid toString

I. GUARD AGAINST CHANGE

Avoid toString

- Pervasive use of **toString** leads to bugs and bad logs because **toString** is callable on everything!

I. GUARD AGAINST CHANGE

Avoid toString

- Pervasive use of **toString** leads to bugs and bad logs because **toString** is callable on everything!

```
final case class Vacancy(job: String, since: Instant)

// Usage
Url(s"company.com/create_vacancy/${vacancy.job}").post.send()
// POST https://company.com/create_vacancy/engineer
```

I. GUARD AGAINST CHANGE

Avoid toString

- Pervasive use of **toString** leads to bugs and bad logs because **toString** is callable on everything!

```
final case class Vacancy(job: String, since: Instant)

// Usage
Url(s"company.com/create_vacancy/${vacancy.job}").post.send()
// POST https://company.com/create_vacancy/engineer
```

```
final case class Job(name: String, category: JobCategory)
final case class Vacancy(job: Job, since: Instant)

// Oops
// POST https://company.com/create_vacancy/Job(engineer, 2019-01-01T00:00:00Z)
```

I. GUARD AGAINST CHANGE

Avoid toString

Solution 1:

I. GUARD AGAINST CHANGE

Avoid toString

Solution 1:

- Use a custom interpolator that only allows `String` parameters,

I. GUARD AGAINST CHANGE

Avoid toString

Solution 1:

- Use a custom interpolator that only allows `String` parameters,
- Explicit conversion to String

I. GUARD AGAINST CHANGE

Avoid toString

Solution 1:

- Use a custom interpolator that only allows `String` parameters,
- Explicit conversion to String
- Don’t use toString unless you’re forced to (e.g. Int)

I. GUARD AGAINST CHANGE

Avoid toString

Solution 1:

- Use a custom interpolator that only allows `String` parameters,
- Explicit conversion to String
- Don’t use toString unless you’re forced to (e.g. Int)

```
// Compile error! vacancy.job is not a String!
Url(str"company.com/create_vacancy/${vacancy.job}").post.send()

// Compiles, all interpolated values are type String!
Url(str"company.com/create_vacancy/${vacancy.job.name}").post.send()
```

I. GUARD AGAINST CHANGE

Avoid toString

Solution 2: Custom typeclass

I. GUARD AGAINST CHANGE

Avoid toString

Solution 2: Custom typeclass

- Write custom typeclass e.g. `UrlShow`

I. GUARD AGAINST CHANGE

Avoid toString

Solution 2: Custom typeclass

- Write custom typeclass e.g. `UrlShow`
- Provide typeclasses instances for types that are safe to be printed

I. GUARD AGAINST CHANGE

Avoid toString

Solution 2: Custom typeclass

- Write custom typeclass e.g. `UrlShow`
- Provide typeclasses instances for types that are safe to be printed
- Scala allow custom string interpolators (For example see `cats.Show`)

I. GUARD AGAINST CHANGE

Avoid toString

Solution 2: Custom typeclass

- Write custom typeclass e.g. `UrlShow`
- Provide typeclasses instances for types that are safe to be printed
- Scala allow custom string interpolators (For example see `cats.Show`)

```
Url(url"company.com/jobs/${vacancy.job.value}?page=${someNumber}").get.send()
```

II. MAKE IT EASY TO DO THE RIGHT THING

…and hard to make mistakes

II. MAKE IT EASY TO DO THE RIGHT THING

…and hard to make mistakes

- Use types & language features as guard rails

II. MAKE IT EASY TO DO THE RIGHT THING

Newtypes

Wrap existing types in a new class

II. MAKE IT EASY TO DO THE RIGHT THING

Newtypes

Wrap existing types in a new class

- Readability - Domain concepts!
- Avoid mistakes
- Improves refactoring
- Enforce additional constraints

II. MAKE IT EASY TO DO THE RIGHT THING

Newtypes

Which is easier to use and understand?

```
final case class Directory(
  id: UUID,
  ownerId: UUID,
  parentId: Option[UUID],
  name: String
)
```

```
final case class Directory(
  id: DirectoryId,
  ownerId: UserId,
  parentId: Option[DirectoryId],
  name: DirectoryName
)
```

II. MAKE IT EASY TO DO THE RIGHT THING

Newtypes

How?

II. MAKE IT EASY TO DO THE RIGHT THING

Newtypes

How?

- Scala 2: **Case class + AnyVal**, or newtype library
- Scala 3: **Opaque Type**

II. MAKE IT EASY TO DO THE RIGHT THING

Newtypes

How?

- Scala 2: **Case class + AnyVal**, or newtype library
- Scala 3: **Opaque Type**

```scala
final case class DirectoryId(uuid: UUID) extends AnyVal

// Use it like any other case class
DirectoryId(someUuid)
```

II. MAKE IT EASY TO DO THE RIGHT THING

Newtypes

How?

- Scala 2: **Case class + AnyVal**, or newtype library
- Scala 3: **Opaque Type**

```scala
final case class DirectoryId(uuid: UUID) extends AnyVal

// Use it like any other case class
DirectoryId(someUuid)
```

- **AnyVal** will avoid allocating the wrapper class*
  - Reduce allocations (GC) and indirection
  - *Allocations are still incurred in some cases
- **newtype** library does not suffer from this issue

II. MAKE IT EASY TO DO THE RIGHT THING

Newtypes with constraints

Enforce constraints using wrapper classes

II. MAKE IT EASY TO DO THE RIGHT THING

Newtypes with constraints

Enforce constraints using wrapper classes

- We want:
  - No direct construction (`new`, `apply`) nor `copy`
  - `unapply`, `hashCode` and `equals`

II. MAKE IT EASY TO DO THE RIGHT THING

Newtypes with constraints

Enforce constraints using wrapper classes

- We want:
  - No direct construction (`new`, `apply`) nor `copy`
  - `unapply`, `hashCode` and `equals`

```
final class private DirectoryName // Need to implement equals manually :(
final case class DirectoryName private (str: String) // .copy still accessible :(
```

II. MAKE IT EASY TO DO THE RIGHT THING

Newtypes with constraints

Introducing `sealed abstract case class`!!

II. MAKE IT EASY TO DO THE RIGHT THING

Newtypes with constraints

Introducing `sealed abstract case class`!!

- No `copy`, `new` nor `apply`

II. MAKE IT EASY TO DO THE RIGHT THING

Newtypes with constraints

Introducing `sealed abstract case class`!!

- No `copy`, `new` nor `apply`
- `unapply`, `equals`, `hashCode` still works

II. MAKE IT EASY TO DO THE RIGHT THING

Newtypes with constraints

Introducing `sealed abstract case class`!!

- No `copy`, `new` nor `apply`
- `unapply`, `equals`, `hashCode` still works

Recommendation:

- A validated construction function
  - Returns **Either / Validated (cats)**
- An unsafe construction (explicitly marked unsafe)
  - Throws exception if an invalid input is provided

II. MAKE IT EASY TO DO THE RIGHT THING

Newtypes with constraints

II. MAKE IT EASY TO DO THE RIGHT THING

Newtypes with constraints

```scala
sealed abstract case class DirectoryName(strValue: String)

object DirectoryName {
  def fromString(str: String): Either[DirectoryNameError, DirectoryName] = {
    if (str.isEmpty)
      Left(DirectoryNameError.StringIsEmpty)
    else
      // Use anonymous subclass (allowed only in this file) to create an instance
      Right(new DirectoryName(str) {}) // ###
  }

  import cats.syntax.either._ // Provides .valueOr extension method
  // For tests or parsing from trusted/validated sources (e.g. Database)
  def fromStringUnsafe(str: String): DirectoryName = {
    fromString(str).valueOr(e  => throw e)
  }
}
```

II. MAKE IT EASY TO DO THE RIGHT THING

Named Parameters

Named parameters improves readability and help spot mistakes

```
def calculateTotal(
  items: List[Item],
  addTax: Boolean,
  addServiceCharge: Boolean
)
```

II. MAKE IT EASY TO DO THE RIGHT THING

Named Parameters

```
def printReceipt(items: List[Item], options: PrintOption)
calculateTotal(
  items,
  options.addServiceCharge, // bug
  options.addTax,
)
```

```
def printReceipt(items: List[Item], options: PrintOption)
calculateTotal(
  items,
  addServiceCharge = options.addTax, // Aha!
  addTax = options.addServiceCharge,
)
```

III. BUILD ABSTRACTIONS WITH TYPES

III. BUILD ABSTRACTIONS WITH TYPES

- Use types to help build abstractions and communicate intent…and don’t lie!

III. BUILD ABSTRACTIONS WITH TYPES

- Use types to help build abstractions and communicate intent…and don’t lie!
- Good abstractions

III. BUILD ABSTRACTIONS WITH TYPES

- Use types to help build abstractions and communicate intent…and don’t lie!
- Good abstractions
  - Reduces cognitive overhead when reading code

III. BUILD ABSTRACTIONS WITH TYPES

- Use types to help build abstractions and communicate intent…and don’t lie!
- Good abstractions
  - Reduces cognitive overhead when reading code
  - Allows aggresive refactoring

III. BUILD ABSTRACTIONS WITH TYPES

Parametricity

III. BUILD ABSTRACTIONS WITH TYPES

Parametricity

- Obeying what we know about a type

III. BUILD ABSTRACTIONS WITH TYPES

Parametricity

- Obeying what we know about a type
- No reflections

III. BUILD ABSTRACTIONS WITH TYPES

Parametricity

- Obeying what we know about a type
- No reflections

```
def f[T](list: List[T]): List[T]

f(List(1,2,3))
// List(1)

f(List(1.0, 2.0, 3.0))
// List(1.0)

f(List("1", "2", "3"))
// Nil ??!!!
```

III. BUILD ABSTRACTIONS WITH TYPES

Parametricity

- Obeying what we know about a type
- No reflections

```
def f[T](list: List[T]): List[T]

f(List(1,2,3))
// List(1)

f(List(1.0, 2.0, 3.0))
// List(1.0)

f(List("1", "2", "3"))
// Nil ??!!!
```

```
// Don't do this!
list match {
  case (s: String) :: rest => Nil
  case s :: rest => List(s)
  // ...
}
```

III. BUILD ABSTRACTIONS WITH TYPES

Parametricity

Solution: Use typeclasses! (or subtype constraints)

```
def f[T](list: List[T])(implicit monoid: Monoid[T]): T = {
  // Might use "empty" and "combine" from the Monoid typeclass instance of T
}
```

III. BUILD ABSTRACTIONS WITH TYPES

Use an IO type!

Use a referentially transparent IO type as found in cats-effect, Monix or ZIO

III. BUILD ABSTRACTIONS WITH TYPES

Use an IO type!

Use a referentially transparent IO type as found in cats-effect, Monix or ZIO

- Refactor freely because side-effect definition is separate from construction!

III. BUILD ABSTRACTIONS WITH TYPES

Use an IO type!

Use a referentially transparent IO type as found in cats-effect, Monix or ZIO

- Refactor freely because side-effect definition is separate from construction!

```
// Refactoring this will change the behaviour :(
for {
  _ <- Future { println("hi") }
  _ <- Future { println("hi") }
} yield ()
```

III. BUILD ABSTRACTIONS WITH TYPES

Use an IO type!

Use a referentially transparent IO type as found in cats-effect, Monix or ZIO

- Refactor freely because side-effect definition is separate from construction!

```
// Refactoring this will change the behaviour :(
for {
  _ <- Future { println("hi") }
  _ <- Future { println("hi") }
} yield ()
```

- Cancellation, Retries, Parallelism

III. BUILD ABSTRACTIONS WITH TYPES

Use an IO type!

Use a referentially transparent IO type as found in cats-effect, Monix or ZIO

- Refactor freely because side-effect definition is separate from construction!

```
// Refactoring this will change the behaviour :(
for {
  _ <- Future { println("hi") }
  _ <- Future { println("hi") }
} yield ()
```

- Cancellation, Retries, Parallelism

```
// Usine ZIO
def task1: IO[Int] = IO { /* Side effect here */ }
def tasks: List[IO[Int]] = List(task1, task2, task3)

ZIO.sequence(tasks)    // Sequential execution
ZIO.sequencePar(tasks) // Parallel execution
task1.race(task2)      // First success is returned, other is cancelled
```

RECAP

RECAP

- Guard against change

RECAP

- Guard against change
- Make it easy to do the right thing

RECAP

- Guard against change
- Make it easy to do the right thing
- Build abstractions with types

RECAP

- Guard against change
- Make it easy to do the right thing
- Build abstractions with types

In practice:

RECAP

- Guard against change
- Make it easy to do the right thing
- Build abstractions with types

In practice:

- Avoid default parameters

RECAP

- Guard against change
- Make it easy to do the right thing
- Build abstractions with types

In practice:

- Avoid default parameters
- Avoid catch-alls in pattern matching

RECAP

- Guard against change
- Make it easy to do the right thing
- Build abstractions with types

In practice:

- Avoid default parameters
- Avoid catch-alls in pattern matching
- Avoid toString

RECAP

- Guard against change
- Make it easy to do the right thing
- Build abstractions with types

In practice:

- Avoid default parameters
- Avoid catch-alls in pattern matching
- Avoid toString
- Newtypes

RECAP

- Guard against change
- Make it easy to do the right thing
- Build abstractions with types

In practice:

- Avoid default parameters
- Avoid catch-alls in pattern matching
- Avoid toString
- Newtypes
- Used named parameters

RECAP

- Guard against change
- Make it easy to do the right thing
- Build abstractions with types

In practice:

- Avoid default parameters
- Avoid catch-alls in pattern matching
- Avoid toString
- Newtypes
- Used named parameters
- Parametricity

RECAP

- Guard against change
- Make it easy to do the right thing
- Build abstractions with types

In practice:

- Avoid default parameters
- Avoid catch-alls in pattern matching
- Avoid toString
- Newtypes
- Used named parameters
- Parametricity
- Use IO

FINAL THOUGHTS

FINAL THOUGHTS

- Learn, but verify - “Does it really solve a problem for me?”

FINAL THOUGHTS

- Learn, but verify - “Does it really solve a problem for me?”
- The principles applies to other languages too!

FINAL THOUGHTS

- Learn, but verify - “Does it really solve a problem for me?”
- The principles applies to other languages too!
- Always keep the trade-offs in mind

REFERENCES

- Constraints Liberate, Liberties Constrain by Rúnar Bjarnason
- Cats Effect: The IO Monad for Scala by Gabriel Volpe
- Scaluzzi - Linting rules with Scalafix

THANK YOU!