INTRODUCTION TO

FUNCTIONAL PROGRAMMING

HISTORY

- 1900. Halting problem. David Hilbert.
- 1928. Entscheidungsproblem (decision problem). David Hilbert.
- 1932. Lambda calculus. Alonzo Church.
- 1936. Turing machine. Alan Turing.
- 1954. Fortran. John Warner Backus.
- 1958. LISP. John McCarthy.
- 1967. Simula 67. Kristen Nygaard, Ole-Johan Dahl.
- 1970. Smalltalk. Alan Curtis Kay.
- 1973. ML. Robin Milner.
- 1985. C++. Bjarne Stroustrup.
- 1990. Haskell. FPCA ’87.
- 1995. Java. James Gosling.

HISTORY

```
<function> := λ <name> . <expression>

<expression> := <name> | <function> | <application>

<application> := <expression> <expression>
```

HISTORY

<function> := λ <name> . <expression>

<expression> := <name> | <function> | <application>

<application> := <expression> <expression>

id = λx.x

<function> := λ <name> . <expression>

<expression> := <name> | <function> | <application>

<application> := <expression> <expression>

id = λx.x

0 = λf.λx.x
1 = λf.λx.(f x)
2 = λf.λx.(f (f x))
3 = λf.λx.(f (f (f x)))

<function> := λ <name> . <expression>

<expression> := <name> | <function> | <application>

<application> := <expression> <expression>

id = λx.x

0 = λf.λx.x

1 = λf.λx.(f x)

2 = λf.λx.(f (f x))

3 = λf.λx.(f (f (f x)))

divide = (λn.((λf.(λx.x x) (λx.f (x x))) (λc.λn.λm.λf.λx.(λd.(λn.n (λx. (λa.λb.b)) (λa.λb.a)) d ((λf.λx.x) f x) (f (c d m f x))) ((λm.λn.n (λn.λf.λx.n (λg.λh.h (g f)) (λu.x) (λu.u)) m) n m))) ((λn.λf.λx. f (n f x)) n))

PURE FUNCTIONS & IMMUTABILITY

> Functional programming is a programming paradigm – a style of building the structure and elements of computer programs – that treats computation as the evaluation of mathematical functions and avoids changing - state and mutable data.

Wikipedia

PURE FUNCTION

- Given the same input, will always return the same output
- Produces no side effects

PURE FUNCTIONS

```java
int max(int a, int b) {
    return a > b ? a : b;
}
```

PURE FUNCTIONS

```java
int max(int a, int b) {
    return a > b ? a : b;
}
```

```scala
def average(a: Int, b: Int): Double = {
  (a + b) / 2
}
```

PURE FUNCTIONS

```
int max(int a, int b) {
    return a > b ? a : b;
}
```

```
def average(a: Int, b: Int): Double = {
  (a + b) / 2
}
```

```
square r = pi * r^2
```

PURE FUNCTIONS

```
int max(int a, int b) {
    return a > b ? a : b;
}
```

```
def average(a: Int, b: Int): Double = {
  (a + b) / 2
}
```

```
square r = pi * r^2
```

IMPURE FUNCTIONS

```
int nextRandom(int value) {
    return (int) (Math.random() * value);
}
```

PURE FUNCTIONS & IMMUTABILITY

PURE FUNCTIONS

```java
int max(int a, int b) {
    return a > b ? a : b;
}
```

```scala
def average(a: Int, b: Int): Double = {
  (a + b) / 2
}
```

```haskell
square r = pi * r^2
```

IMPURE FUNCTIONS

```java
int nextRandom(int value) {
    return (int) (Math.random() * value);
}
```

```scala
def updateBalance(charge: Double): Unit = {
  balance -= charge
  saveNewBalance(balance)
}
```

PURE FUNCTIONS & IMMUTABILITY

PURE FUNCTIONS

```
int max(int a, int b) {
    return a > b ? a : b;
}
```

```
def average(a: Int, b: Int): Double = {
  (a + b) / 2
}
```

```
square r = pi * r^2
```

IMPURE FUNCTIONS

```
int nextRandom(int value) {
    return (int) (Math.random() * value);
}
```

```
def updateBalance(charge: Double): Unit = {
  balance -= charge
  saveNewBalance(balance)
}
```

```
void printSum(int a, int b) {
    System.out.println(a + b);
}
```

PURE FUNCTIONS & IMMUTABILITY

PURE

```scala
def updateBalance(balance: Double, charge: Double): Double
```

IMPURE

```scala
def updateBalance(charge: Double): Unit
```

PURE FUNCTIONS

- Easier to reason about
- Easier to test
- Easier to combine
- Easier to parallelize
- Easier to refactor
- Lazy evaluation
- Memoization

IMMUTABILITY

- Thread safety
- Prevent inconsistent state
- No temporal couplings
- Easier to cache
- Simpler to construct, test, and use

PURE FUNCTIONS & IMMUTABILITY

```java
public static void main(String... args) {
    List<Score> scores = Arrays.asList(
            new Score("Albert Einstein", 75),
            new Score("Isaac Newton", 98),
            new Score("Charles Darwin", 65));

    scoreService.publishBest(scores);
    scoreService.publishFirst(scores);
}
```

PURE FUNCTIONS & IMMUTABILITY

```java
public static void main(String... args) {
    List<Score> scores = Arrays.asList(
            new Score("Albert Einstein", 75),
            new Score("Isaac Newton", 98),
            new Score("Charles Darwin", 65));

    scoreService.publishBest(scores);
    scoreService.publishFirst(scores);
}


public void publishBest(List<Score> scores) {
    scores.sort(scoreComparator);
    Score result = scores.get(0);
    scoreTableDao.saveBest(result);
}

public void publishFirst(List<Score> scores) {
    Score result = scores.get(0);
    scoreTableDao.saveFirst(result);
}
```

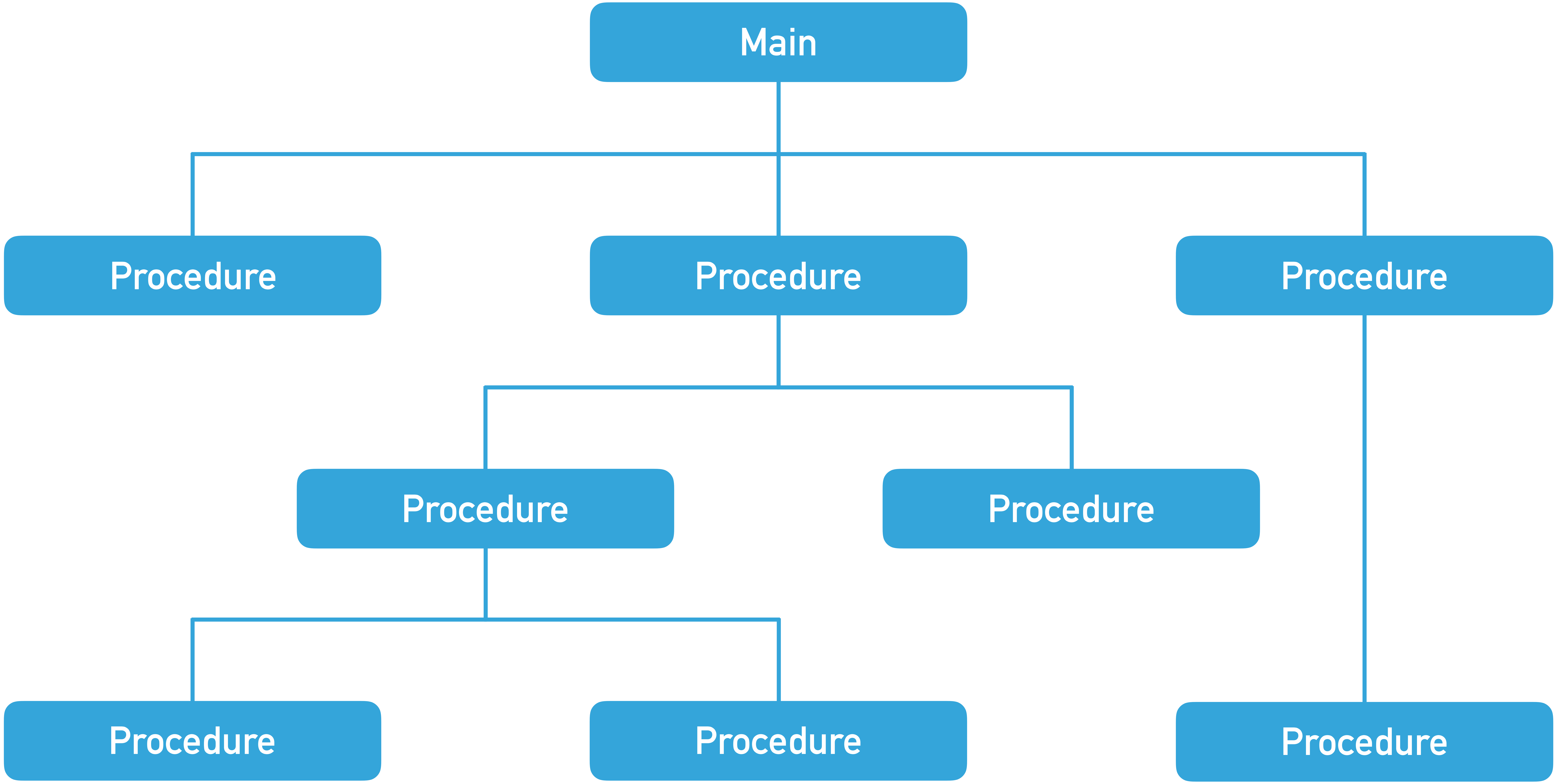
PURE FUNCTIONS & IMMUTABILITY

```java
public static void main(String... args) {
    List<Score> scores = Arrays.asList(
            new Score("Albert Einstein", 75),
            new Score("Isaac Newton", 98),
            new Score("Charles Darwin", 65));

    scoreService.publishBest(scores);
    scoreService.publishFirst(scores);
}


public void publishBest(List<Score> scores) {
    scores.sort(scoreComparator);
    Score result = scores.get(0);
    scoreTableDao.saveBest(result);
}

public void publishFirst(List<Score> scores) {
    Score result = scores.get(0);
    scoreTableDao.saveFirst(result);
}
```

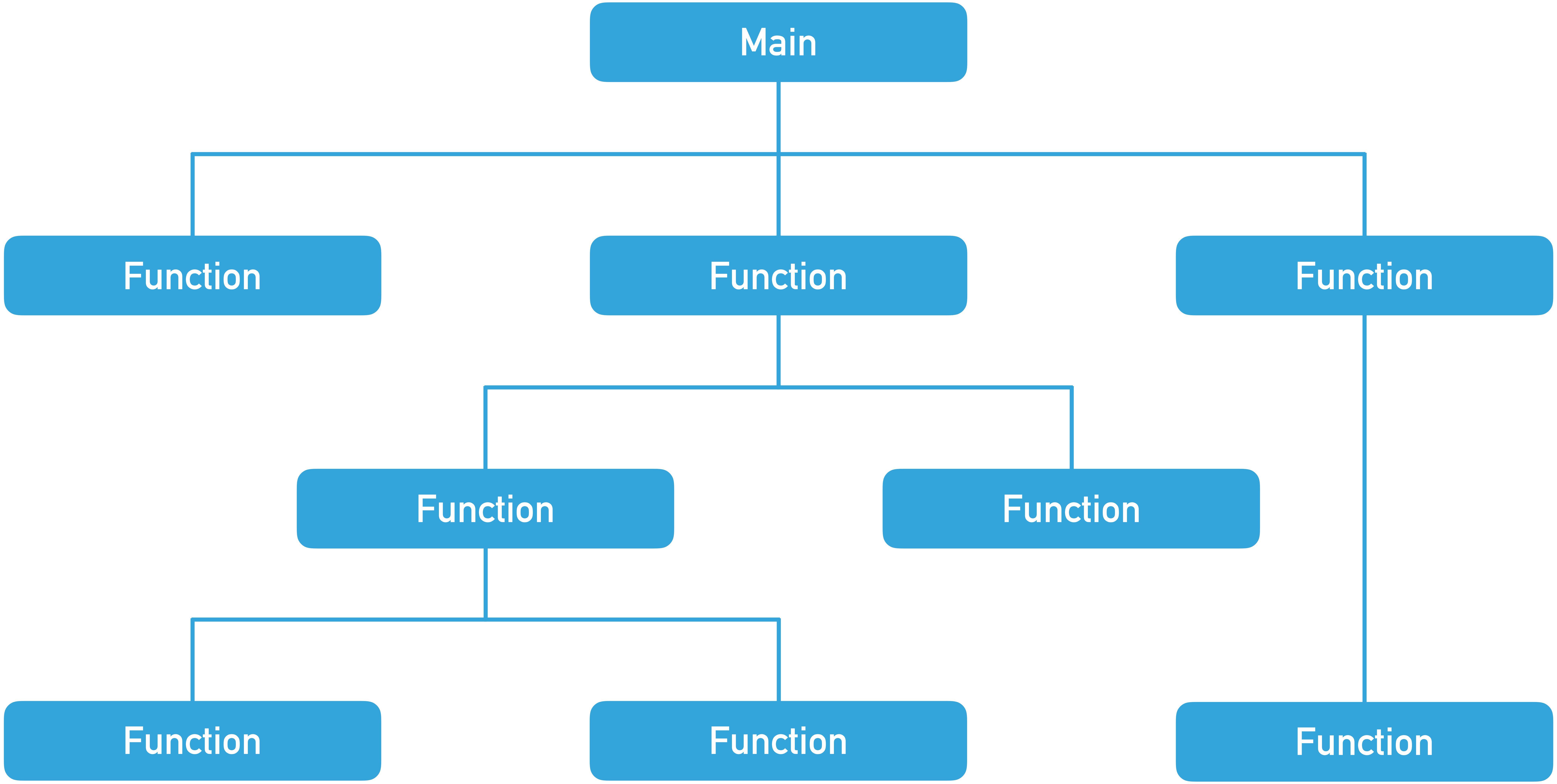
```
Best score: Isaac Newton 98
First: Isaac Newton 98
```

PURE FUNCTIONS & IMMUTABILITY

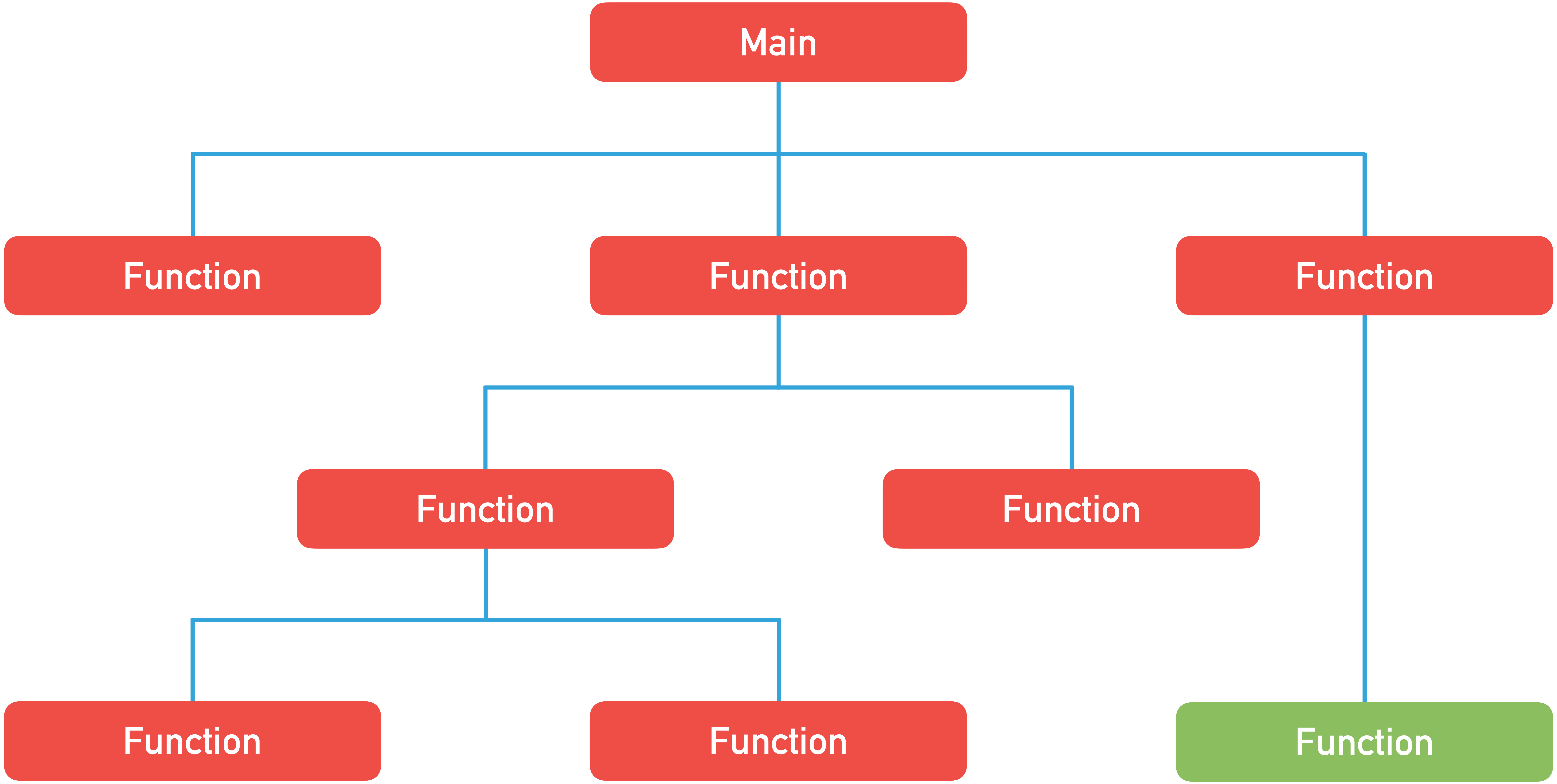
```
public static void main(String... args) {
    List<Score> scores = Arrays.asList(
            new Score("Albert Einstein", 75),
            new Score("Isaac Newton", 98),
            new Score("Charles Darwin", 65));

    scoreService.publishBest(scores);
    scoreService.publishFirst(scores);
}


public Optional<Score> getBest(Collection<Score> scores) {
    return scores.stream().max(scoreComparator);
}

public Optional<Score> getFirst(Collection<Score> scores) {
    return scores.stream().findFirst();
}

public void publishBest(List<Score> scores) {
    getBest(scores).ifPresent(scoreTableDao::saveBest);
}

public void publishFirst(List<Score> scores) {
    getFirst(scores).ifPresent(scoreTableDao::saveFirst);
}
```


Main
Procedure
Procedure
Procedure
Procedure
Procedure
Procedure
Procedure
Procedure

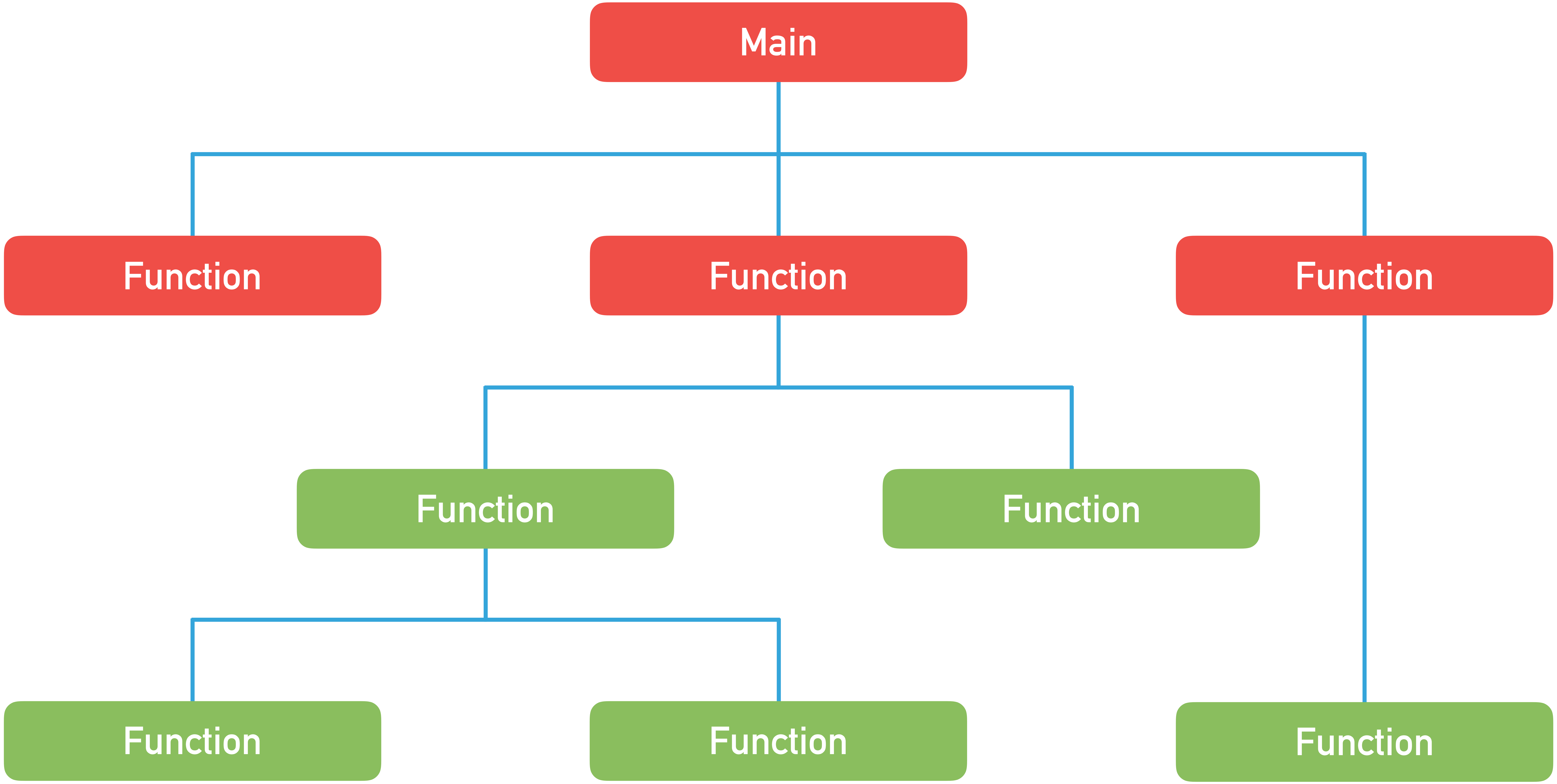
PURE FUNCTIONS & IMMUTABILITY



PURE FUNCTIONS & IMMUTABILITY



PURE FUNCTIONS & IMMUTABILITY



> OO makes code understandable by encapsulating moving parts. FP makes code understandable by minimizing moving parts.

Michel Feathers

RICH

```java
class Person {
    private int x;
    private int y;

    public void moveToStartPoint() {
        this.x = 0;
        this.y = 0;
    }
}
```

ANEMIC

```java
class Person {
    private int x;
    private int y;

    public int getX();
    public void setX(int x);
    public int getY();
    public void setY(int y);
}

class MovementService {
    void movePersonToStartPoint(Person person) {
        person.setX(0);
        person.setY(0);
    }
}
```

TYPE DRIVEN DEVELOPMENT

> A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute.

Benjamin C. Pierce

TYPE DRIVEN DEVELOPMENT

```scala
case class Customer(
  id: Int,
  firstName: String,
  middleName: String
  lastName: String,
  email: String,
  emailVerifyDate: LocalDateTime,
  emailVerified: Boolean
)
```

TYPE DRIVEN DEVELOPMENT

```
case class Customer(
  id: Int,
  firstName: String,
  middleName: String
  lastName: String,
  email: String,
  emailVerifyDate: LocalDateTime,
  emailVerified: Boolean
)

def changeCustomerName(id: Int, firstName: String, middleName: String, lastName: String): Unit

def sendVerificationRequest(email: String): Unit

def validateEmailAddress(email: String): Boolean
```

TYPE DRIVEN DEVELOPMENT

```
case class Customer(
  id: CustomerId,
  firstName: String,
  middleName: String
  lastName: String,
  email: String,
  emailVerifyDate: LocalDateTime,
  emailVerified: Boolean
)

case class CustomerId(id: Int)

def changeCustomerName(id: CustomerId, firstName: String, middleName: String, lastName: String): Unit

def sendVerificationRequest(email: String): Unit

def validateEmailAddress(email: String): Boolean
```

TYPE DRIVEN DEVELOPMENT

```scala
case class Customer(
  id: CustomerId,
  name: PersonalName,
  email: String,
  emailVerifyDate: LocalDateTime,
  emailVerified: Boolean
)

case class CustomerId(id: Int)

case class PersonalName(
  firstName: String,
  middleName: String,
  lastName: String
)

def changeCustomerName(id: CustomerId, newName: PersonalName): Unit

def sendVerificationRequest(email: String): Unit

def validateEmailAddress(email: String): Boolean
```

TYPE DRIVEN DEVELOPMENT

```scala
case class Customer(
  id: CustomerId,
  name: PersonalName,
  email: String,
  emailVerifyDate: LocalDateTime,
  emailVerified: Boolean
)

case class CustomerId(id: Int)

case class PersonalName(
  firstName: String,
  middleName: Option[String],
  lastName: String
)

def changeCustomerName(id: CustomerId, newName: PersonalName): Unit

def sendVerificationRequest(email: String): Unit

def validateEmailAddress(email: String): Boolean
```

TYPE DRIVEN DEVELOPMENT

```scala
case class Customer(
  id: CustomerId,
  name: PersonalName,
  email: EmailContactInfo
)

case class CustomerId(id: Int)

case class PersonalName(
  firstName: String,
  middleName: Option[String],
  lastName: String
)

case class EmailContactInfo(
  email: String,
  verifyDate: LocalDateTime,
  verified: Boolean
)

def changeCustomerName(id: CustomerId, newName: PersonalName): Unit

def sendVerificationRequest(email: EmailContactInfo): Unit

def validateEmailAddress(email: String): Boolean
```

TYPE DRIVEN DEVELOPMENT

```scala
case class Customer(
  id: CustomerId,
  name: PersonalName,
  email: EmailContactInfo
)

case class CustomerId(id: Int)

case class PersonalName(
  firstName: String,
  middleName: Option[String],
  lastName: String
)

case class EmailContactInfo(
  email: EmailAddress,
  verifyDate: LocalDateTime,
  verified: Boolean
)

case class EmailAddress(email: String)

def changeCustomerName(id: CustomerId, newName: PersonalName): Unit

def sendVerificationRequest(email: EmailContactInfo): Unit

def validateEmailAddress(email: String): EmailAddress
```

TYPE DRIVEN DEVELOPMENT

```scala
case class Customer(
  id: CustomerId,
  name: PersonalName,
  email: EmailContactInfo
)

case class CustomerId(id: Int)

case class PersonalName(
  firstName: String,
  middleName: Option[String],
  lastName: String
)

case class EmailAddress(email: String)

sealed trait EmailContactInfo
case class VerifiedEmail(email: EmailAddress, verifyDate: LocalDateTime) extends EmailContactInfo
case class UnverifiedEmail(email: EmailAddress) extends EmailContactInfo

def changeCustomerName(id: CustomerId, newName: PersonalName): Unit

def sendVerificationRequest(email: UnverifiedEmail): Unit

def validateEmailAddress(email: String): EmailAddress
```

THE ADVANTAGES OF STATIC TYPING

- Code documentation
- Error detection
- Abstraction
- Performance optimisation
- Tooling support

CONCLUSIONS