Scala超入門

- 同値判定

```
object Equals {
  def main(args: Array[String]) {
    val s1 = new String("Hello")
    val s2 = new String("Hello")

    println(s1 == s2)
  }
}
```

valは変更できない(final相当)。varだと変更できる

true

- Javaでは==は参照の一致判定だが、Scalaでは同値判定（Javaのequals()相当）。
- 同値判定を間違って==で行ってしまうことが無い。

Scala超入門

- 全てがオブジェクト

```
object EverythingIsObject {
  def main(args: Array[String]) {
    val i = -123
    val j = 124

    println(i.abs)
    println(i.abs max j)
  }
}
```

- ScalaにはJavaのプリミティブ型(intなど)が無い。
- プリミティブ型を特別に扱う必要が無い。
- 必要に応じてコンパイラが内部でプリミティブに変換して高速化する。

Scala超入門

- 関数が第一級市民(First class citizen)

```
object Functions {
  def main(args: Array[String]) {
    def sayHello(msg: => String) {
      println("Hello " + msg)
    }

    sayHello("World")

    var count = 0
    sayHello {
      count += 1
      "Sekai" + count
    }
  }
}
```

関数の中に関数を作ることができる

関数を引数にできる。

関数を渡す。

実行結果

```
Hello World
Hello Sekai1
```

Scala超入門

- 関数リテラル

```
def print(i: Int, formatter: Int => String) =
  println(formatter(i))
print(123, i => i.toString)
print(123, _.toString)
```

関数をその場で作って渡す

実行結果

```
123
123
```

Scala超入門

- フィールドとメソッドを同じように扱える。

```
object FieldAndMethod {
  def main(args: Array[String]) {
    var count = 0
    def createNumber() = {
      count += 1
      count
    }

    abstract class Hello {
      def sayHello = "Hello " + message
      def message: String
    }
```

Scala超入門

```
  class HelloWorld extends Hello {
    override val message = "World " + createNumb
  }

  class HelloSekai extends Hello {
    override def message = "Sekai " + createNumb
  }

  val world = new HelloWorld
  println(world.sayHello)
  println(world.sayHello)

  val sekai = new HelloSekai
  println(sekai.sayHello)
  println(sekai.sayHello)
 }
}
```

定数でオーバーライド

関数でオーバーライド

実行結果

```
Hello World 1
Hello World 1
Hello Sekai 2
Hello Sekai 3
```

Scala超入門

- Case class

```
object ValueObject {
  def main(args: Array[String]) {
    case class Person(name: String, age: Int)

    val p1 = Person("Name1", 10)
    val p2 = Person("Name1", 10)

    if (p1 == p2) {
      println(p1 + " == " + p2)
    }
  }
}
```

equals, hashCode, toStringが自動生成される。
name, ageのゲッタが自動生成される。

true

Scala超入門

- 引数

```
object Arguments {
  def main(args: Array[String]) {
    case class Person(name: String = "", age: Int)

    val p1 = Person(age = 10)
    println(p1)
  }
}
```

デフォルト値

引数名指定

Scala超入門

- オペレータオーバーロード
  （実はオーバーロードではなくメソッドの作成）

```
object OperatorOverload {
  def main(args: Array[String]) {
    val b1 = BigDecimal("123.45")
    val b2 = BigDecimal("543.21")

    println(b1 + b2)

    case class Point(val x: Int, val y: Int) {
      def +(that: Point) = Point(x + that.x, y + that.y)
    }

    println(Point(1, 2) + Point(3, 4))
  }
}
```

b1.+(b2)と解釈

実行結果

```
666.66
Point(4,6)
```

Scala超入門

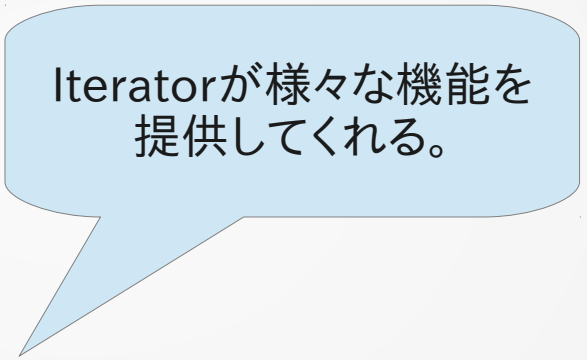
- トレイト ＝ 実装を持てるインターフェイス

```
object Trait {
  def main(args: Array[String]) {
    case class Person(name: String, age: Int) extends Ordered[Person] {
      def compare(that: Person) = age - that.age
    }

    val p1 = Person("Name1", 10)
    val p2 = Person("Name2", 12)

    println(p1 < p2)
```

Orderedトレイトを実装すれば、<や>が使用可能になる。

Scala超入門

```
class Numbers(start: Int, end: Int) extends Iterator[Int] {
  private var i = start
  def hasNext = i <= end
  def next() = try {
    i
  }
  finally {
    i += 1
  }
}

(new Numbers(1, 10)) foreach println
```



実行結果

```
1
2
3
4
5
6
7
8
9
10
```

Scala超入門

- traitは多重継承ではない

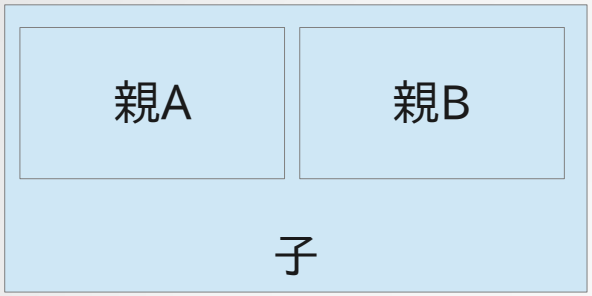
```
trait A {
  override def toString = super.toString + ”[A]”
}

trait B {
  override def toString = super.toString + ”[B]”
}

object ToString {
  def main(args: Array[String]) {
    println((new AnyRef with A with B).toString)
    println((new AnyRef with B with A).toString)
  }
}
```

実行結果

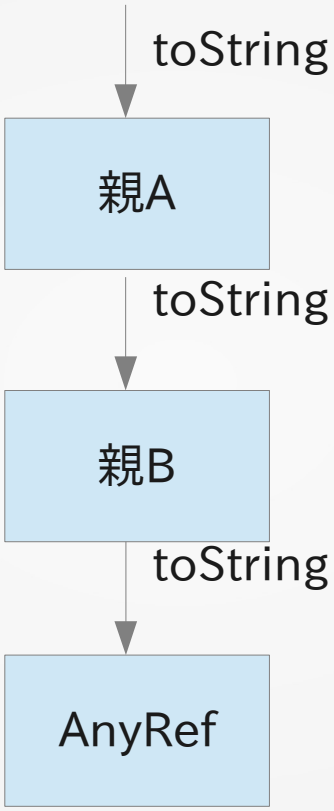
```
ToString$$anon$1@f7e6a96[A][B]
ToString$$anon$2@272d7a10[B][A]
```

Scala超入門

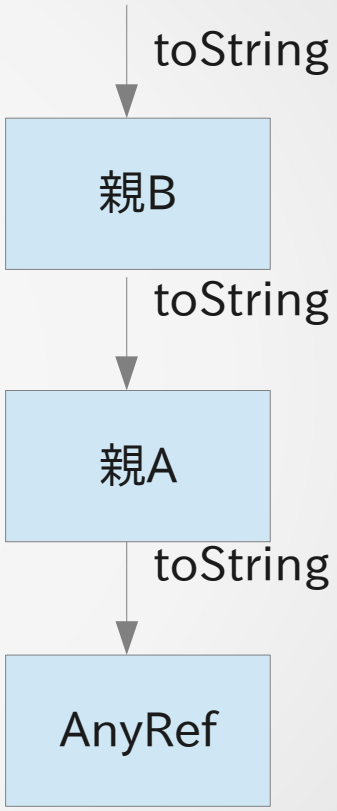
多重継承のイメージ



traitのイメージ



traitのイメージ



new AnyRef with B with A

new AnyRef with A with B

必ず単一継承になる。

Scala超入門

- トレイトを利用したミックスイン

```
object MixIn {
  def main(args: Array[String]) {
    trait Shape
    case class Circle(x: Int, y: Int, r: Int) extends Shape

    trait Widget extends Shape {
      def select()
    }

    class CircleWidget(x: Int, y: Int, r: Int) extends Circle(x, y, r) with Widget {
      def select() {
      }
    }
  }
}
```

Shapeとミックスイン可能

Scala超入門

- nullを排除可能

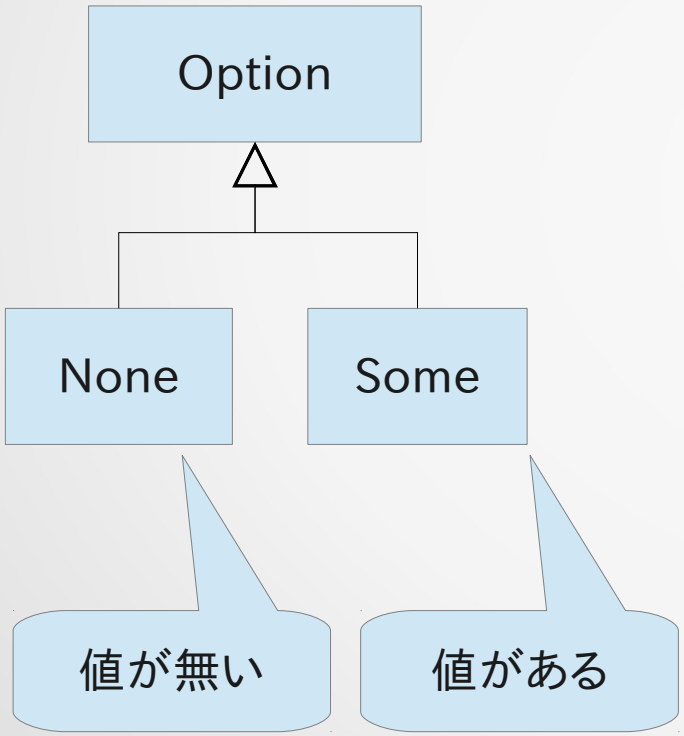
マーカtrait

```
object NonNull {
  def main(args: Array[String]) {
    case class Person(name: String, age: Int) extends NotNull

//    val p1: Person = null
  }
}
```

コンパイルエラー

Scala超入門

- 値があるか無いか分からないケースにはOptionを使用する。




```
List(0, 1, 2) find(_ > 0) match {
  case Some(n) => println("Found " + n)
  case None => println("Not found")
}
```

実行結果

```
Found 1
```

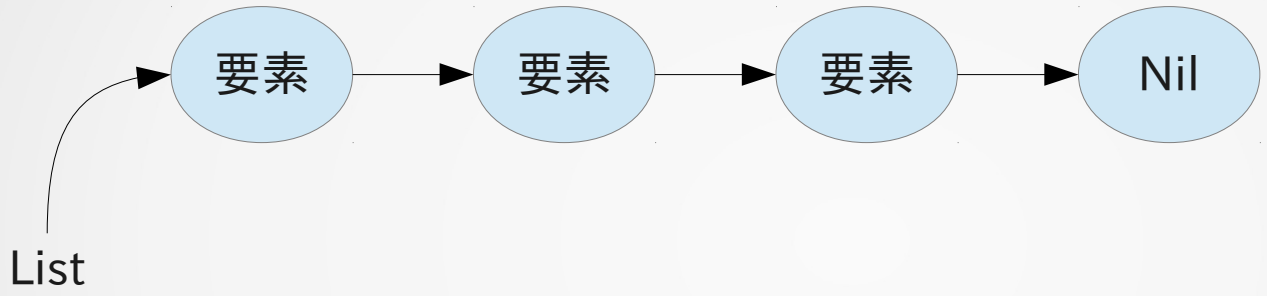
```
println(List(0, 1, 2) find(_ > 3) getOrElse("Not found"))
println(List(0, 1, 2) find(_ > 1) getOrElse("Not found"))
```

実行結果

```
Not found
2
```

Scala超入門

- List（イミュータルブル。Scalaにはミュターブル、イミュータブル、両方のコレクションがある）




```
object Lists {
  def main(args: Array[String]) {
    val list = 1::9::2::3::5::8::Nil
    println(list.filter(_ % 2 == 0))
    println((0 /: list)(_ + _))
    println(list map (_ + 1))
  }
}
```

実行結果

```
List(2, 8)
28
List(2, 10, 3, 4, 6, 9)
```

Scala超入門

- タプル

```
object Tuples {
  def main(args: Array[String]) {
    val t = (1, "Hello", 3f)
    println(t._1)
  }
}
```

任意の型の値をまとめる。

_1が先頭

実行結果

```
1
```

Scala超入門

- Map（イミュータルブル）

```
object Maps {
  def main(args: Array[String]) {
    val map = Map(1 -> "One", 2 -> "Two", 3 -> "Three")
    println(map(1))
    println(map.size)
  }
}
```

2要素のタプル

実行結果

```
One
3
```

Scala超入門

- パターンマッチ

```
object PatternMatch {
  def main(args: Array[String]) {
    "Hello" match {
      case "Hello" => println("it is Hello")
      case _ => println("not Hello")
    }
  }
}
```

Javaのswitchに似ているが、caseには定数以外でも指定できるし、型が限定されない

実行結果

```
it is Hello
```

Scala超入門

- パターンマッチ

```
case class Person(name: String, age: Int)
Person("Taro", 20) match {
  case Person(name, age) => println("%s, %d%n".format(name, age))
}
```

実行結果

```
Taro, 20
```

特定の条件を満たしたクラスは、
パターンとして使用できる。

```
Person("Taro", 20) match {
  case Person("Taro", age) => println("You are Taro.")
  case _ => println("Not Taro")
}
```

name = “Taro”の
Personにのみマッチ。

Scala超入門

- パターンマッチ

Listもパターンマッチに使用できる。先頭要素が0のListにマッチ

```
List(0, 1, 2) match {
  case List(0, _*) => println("Top is zero")
}
```

実行結果

```
Top is zero
```

変数名 @を付けてキャプチャすることもできる

```
List(0, 1, 2) match {
  case List(0, rest @ _*) => println("Top is zero. Rest is " + rest)
}
```

実行結果

```
Top is zero. Rest is List(1, 2)
```

Scala超入門

- パターンマッチ

ifを付加して条件を追加可能

```
List(7, 8, 9) match {
  case List(n, _*) if n < 10 => println("Top(=" + n + ") is less than 10")
}
```

実行結果

```
Top(=7) is less than 10
```

変数宣言でパターンマッチが使える

```
val a::b::Nil = List(1, 2)
println("a = " + a + ", b = " + b)
```

```
val (a, b) = (1, 2)
println("a = " + a + ", b = " + b)
```

実行結果

```
a = 1, b = 2
```

Scala超入門

- カリー化されたメソッド

```
object Functions2 {
  def main(args: Array[String]) {
    def unless(cond: Boolean)(body: => Unit) {
      if (!cond) body
    }

    unless (true) {
      println("True")
    }

    unless (false) {
      println("False")
    }
  }
}
```

実行結果

```
False
```

Scala超入門

- XMLの扱い

```
object Xml {
  def main(args: Array[String]) {
    case class Person(name: String, age: Int)
    val persons = Person("name1", 10)::Person("name2", 12)::Nil
    val xml =
      <persons>
        {for (p <- persons) yield {
          <person>
            <name>{p.name}</name>
            <age>{p.age}</age>
          </person>
        }}
      </persons>
    println(xml)
```

実行結果

```
<persons>
      <person>
        <name>name1</name>
        <age>10</age>
      </person><person>
        <name>name2</name>
        <age>12</age>
      </person>
    </persons>
```

Scala超入門

- XMLの扱い

```
val deserialized = (xml ¥ "person") map { p =>
  Person(name = (p ¥ "name").text, age = (p ¥ "age").text.toInt)
}
println(deserialized)
```

実行結果

```
List(Person(name1,10), Person(name2,12))
```

Scala超入門

- 末尾再帰

```
object TailRecursion {
  def main(args: Array[String]) {
    def sum(from: Int, to: Int): Int = sum2(from, to, 0)

    def sum2(from: Int, to: Int, work: Int): Int =
      if (from > to) work
      else sum2(from + 1, to, from + work)

    println(sum(1, 10))
  }
}
```

自分自身を呼び出す形式になっていればループに展開される

実行結果

```
55
```

再帰で記述すると変数が不要になる。

Scala超入門

- implicit

```
implicit def str2int(str: String) = str.toInt
val i: Int = "123"
println(i)
```

実行結果

```
123
```

暗黙の型変換
Intが要求される場所で
Stringが指定された場
合、変換関数があれば
変換される

```
case class Person(name: String, age: Int)
def printPerson(implicit person: Person) {
  println(person)
}

implicit val person = Person("Taro", 20)
printPerson
printPerson(Person("Hanako", 18))
```

暗黙の引数

実行結果

```
Person(Taro,20)
Person(Hanako,18)
```

Scala超入門

- implicit

```
class Repeat(str: String) {
  def x(times: Int): String = x(times, new StringBuilder())
  def x(times: Int, buf: StringBuilder): String =
    if (times < 1) buf.toString
    else x(times - 1, buf.append(str))
}
implicit def strToRepeat(str: String) = new Repeat(str)

println("Hello" x 5)
```

Stringにはxというメソッドはないが、StringからRepeatへの暗黙の変換が定義されていて、Repeatがxを持っていると変換される。

実行結果

```
HelloHelloHelloHelloHello
```

Scala超入門

- implicit

```
trait ConnectionFactory {
  def apply(): Connection
}

object DriverManagerConnectionFactory extends
ConnectionFactory {
  def apply(): Connection = {
    null
  }
}

object DataSourceConnectionFactory extends
ConnectionFactory {
  def apply(): Connection = {
    null
  }
}
```

つづく...

Scala超入門

```
object Implicit3 {
  def main(args: Array[String]) {
    implicit val f = DriverManagerConnectionFacto
    queryDb
  }

  def queryDb(implicit connectionFactory:
ConnectionFactory) {
    println("factory = " + connectionFactor
  }
}
```

implicit引数は明示的に渡さなくて良い

implicit引数

実行結果:

```
factory =
DriverManagerConnectionFactory$@5101a031
```

Scala超入門

- 定数埋め込み問題

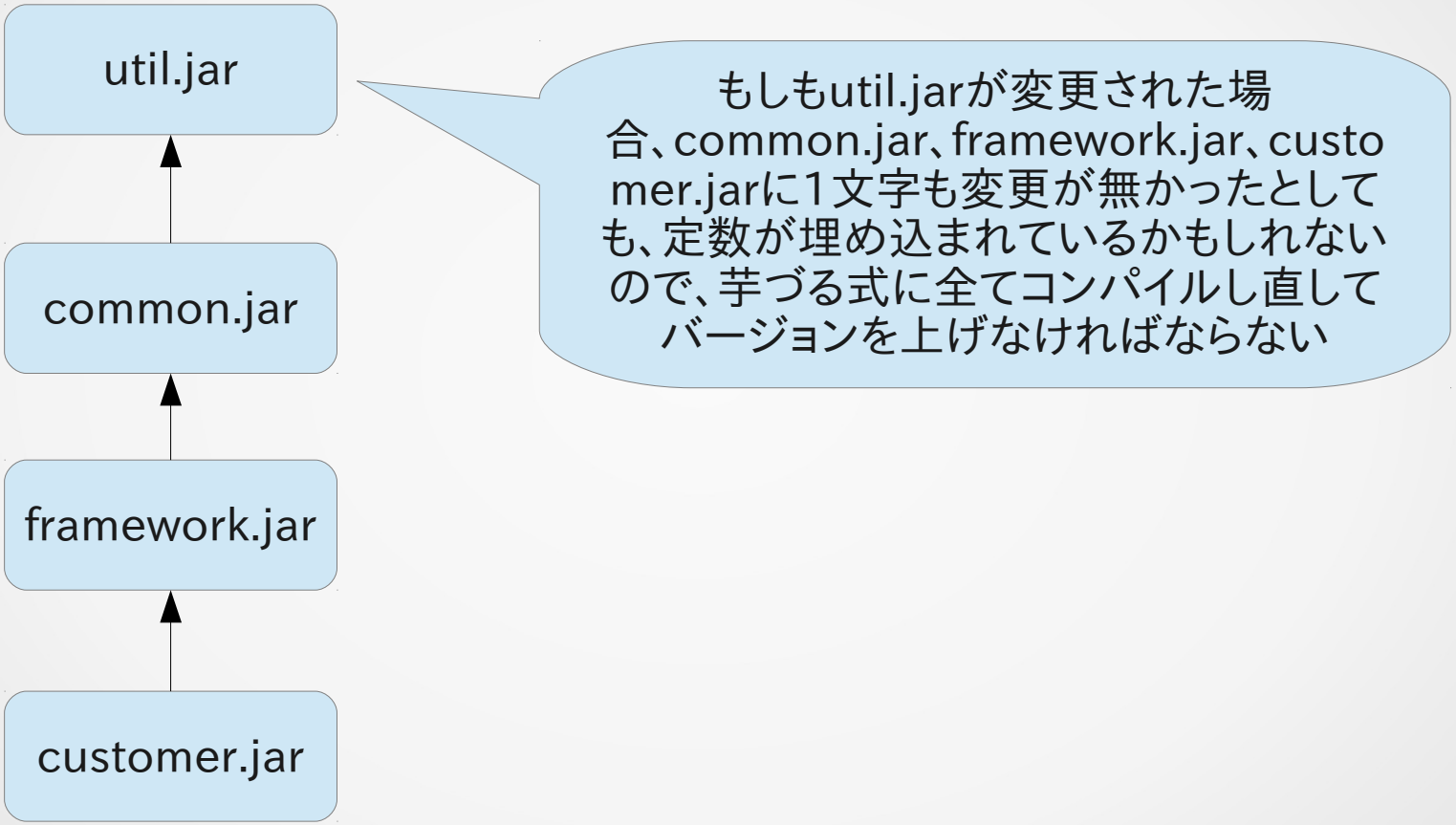
```
public class A {
    public static final int CONST_VALUE = 1;
}
 public class B {
     int i = A.CONST_VALUE;
 }
```

Javaには定数埋め込みという特性があるので大規模開発ではモジュール管理が大変になる

```
  public B();
    flags: ACC_PUBLIC
    Code:
      stack=2, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."<init>":()V
         4: aload_0
         5: iconst_1
         6: putfield      #2                  // Field i:I
         9: return
      LineNumberTable:
        line 1: 0
        line 2: 4
```

A.CONST_VALUEではなく、定数1がそのまま使用されている。

Scala超入門


util.jar
common.jar
framework.jar
customer.jar
もしもutil.jarが変更された場合、common.jar、framework.jar、customer.jarに1文字も変更が無かったとしても、定数が埋め込まれているかもしれないので、芋づる式に全てコンパイルし直してバージョンを上げなければならない

Scala超入門

```
object A {
    val ConstValue = 1;
}

class B {
    def foo() = A.ConstValue;
}
```

Scalaの場合、定数は関数として実装されるため定数が埋め込まれることはない。

```
 public int foo();
   flags: ACC_PUBLIC
   Code:
     stack=1, locals=1, args_size=1
        0: getstatic     #16                 // Field A$.MODULE$:LA$;
        3: invokevirtual #19                 // Method A$.ConstValue:()I
        6: ireturn
```

Scala超入門

- Javaと比べて、どのくらい短くなるのか？
  - 感覚的に1/3くらいになる
- 速度のペナルティは？
  - ほとんど無いが、1-2割ほど遅いと言われている。
  - Javaとの親和性が高いため、速度が重要な部分のみJavaで書くことも可能。

Scala超入門

- コード例：
  - Personクラスと、その集合を管理するクラスをJavaとScalaで比較してみます。

Java版 Person

```
public final class Person {
    final private String name;
    final private int age;
    final private int hashCode;

    public Person(String name, int age) {
        if (name == null) throw new NullPointerException();
        this.name = name;

        if (age < 0)
            throw new IllegalArgumentException("age(=" + age + ") should not be negative.");
        this.age = age;
        hashCode = name.hashCode() ^ age;
    }
```

値オブジェクト

Scala超入門

```
public String getName() {
    return name;
}

public int getAge() {
    return age;
}
```

Scala超入門

```
    @Override public boolean equals(Object o) {
        if (o == this) return true;
        if (o == null) return false;
        if (o.getClass() != getClass()) return false;

        Person that = (Person)o;
        return name.equals(that.name) && age == that.age;
    }

    @Override public int hashCode() {
        return hashCode;
    }

    @Override public String toString() {
        return "Person(name: '" + name + "', age: " + age + ")";
    }
}
```

equals,
hashCode,
toStringは自分で
定義が必要

Scala超入門

Java版 PersonBuilder

ビルダクラスも自分で
定義が必要

```
public class PersonBuilder {
    private String name;
    private int age;

    public PersonBuilder() {
        name = "";
        age = 0;
    }

    public PersonBuilder(Person person) {
        name = person.getName();
        age = person.getAge();
    }

    public PersonBuilder withName(String name) {
        if (name == null) throw new NullPointerException();
        this.name = name;
        return this;
    }
```

Scala超入門

```
    public PersonBuilder withAge(int age) {
        if (age < 0)
            throw new IllegalArgumentException("age(=" + age + ") should not be negative.");
        this.age = age;
        return this;
    }

    public Person toPerson() {
        return new Person(name, age);
    }
}
```

同じエラー処理が重複する。

Scala超入門

Java版 PersonRepository

```
public class PersonRepository {
    private List<Person> repository = new ArrayList<>();

    public PersonRepository addPerson(Person person) {
        if (person == null) throw new NullPointerException();
        repository.add(person);
        return this;
    }

    public Iterable<Person> all() {
        return new ArrayList<>(repository);
    }

    public Iterable<Person> distinct() {
        return new HashSet<>(repository);
    }
```

ArrayListは不変ではない
ため防御的コピー必須

Scala超入門

```
    public boolean exists(Person person) {
        if (person == null) throw new NullPointerException();
        for (Person p: repository) {
            if (p.equals(person)) return true;
        }

        return false;
    }

    public Iterable<Person> havingName(String name) {
        if (name == null) throw new NullPointerException();
        List<Person> ret = new ArrayList<>();
        for (Person person: repository) {
            if (person.getName().equals(name))
                ret.add(person);
        }

        return ret;
    }
}
```

Personはnullかもしれない
のでnullチェック必須

Scala超入門

Scala版 Person

Personはnullにはならない

```
case class Person(name: String, age: Int) extends NotNull {
   require(name != null)
   require(age < 0, "age(=" + age + ") should not be
negative.")

  def set(
    name: String = this.name,
    age: Int = this.age
  ) = Person(name, age)
}
```

名前付き引数とデフォルト値を用いる
ことでビルダクラスは不要に

equals, hashCode,
toStringは自動生成される

Scala超入門

Scala版 PersonRepository

```
class PersonRepository {
  private var repository = List[Person]()

  def addPerson(person: Person) = {
    repository = person::repository
    this
  }

  def all = repository
  def distinct = HashSet(repository)
  def exists(person: Person) = repository.exists(_ == person)
  def havingName(name: String) = {
    require(name != null)
    repository.filter(_.name == name)
  }
}
```

Personはnullにならないので、nullチェック不要

Listは不変なので防御的コピー不要

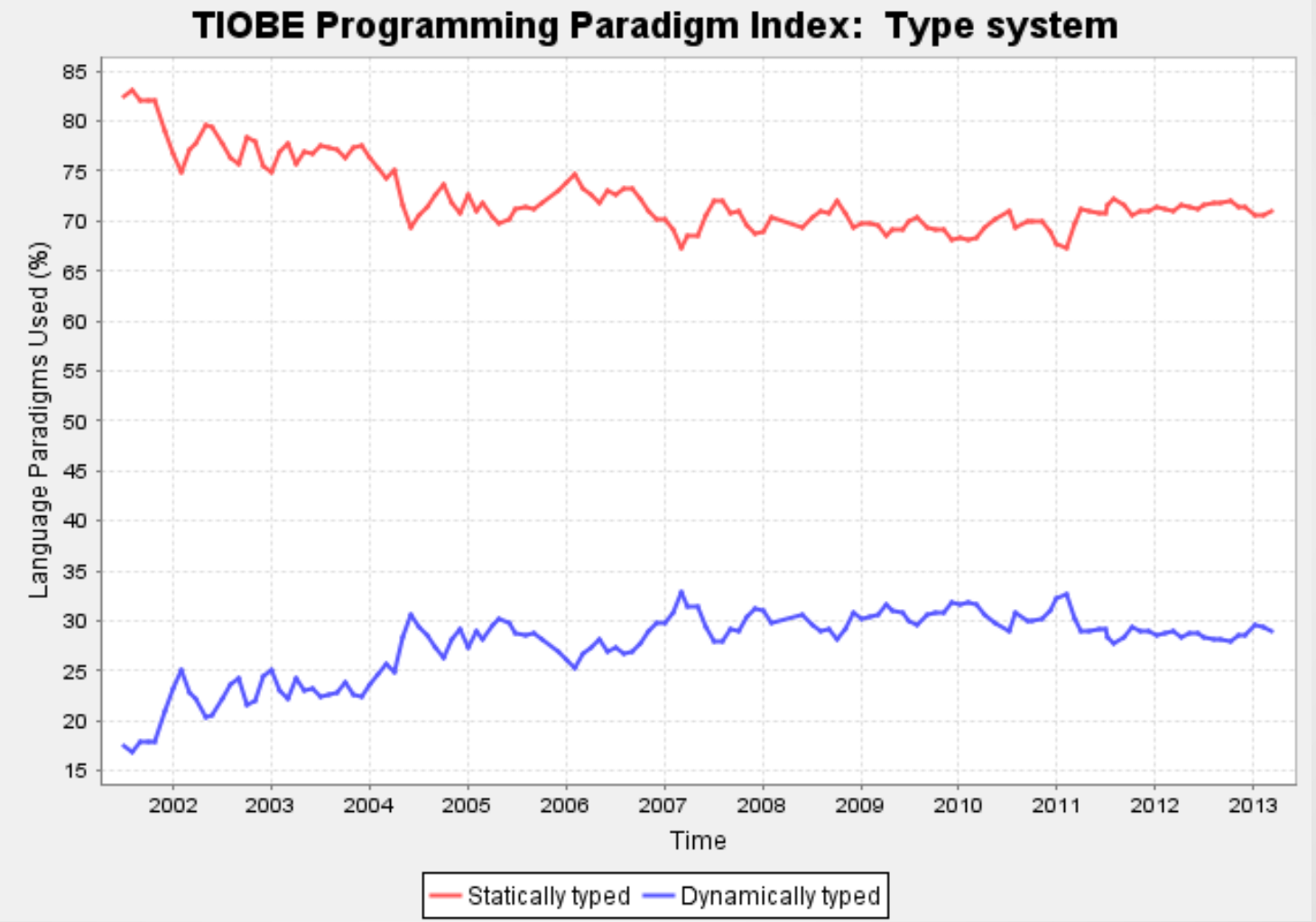
関数型プログラミングで簡潔に実装

Scala まとめ

- 動的言語のように簡単な記述が可能
  - 型推論
  - 関数型
  - 全てがオブジェクト
  - パターンマッチ
  - 演算子の名前を持ったメソッド定義
- オブジェクト指向支援機能
  - 不変オブジェクト（不変コレクション）
  - 値オブジェクト（case class）
  - nullの扱いの回避

型付けが動的な言語と静的な言語

- トレンド



http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html