オブジェクト指向言語における関数型イディオムの実態調査

田中 紘都[1,a)]　柗本 真佑[1,b)]　楠本 真二[1,c)]

概要：
プログラミング言語の進化は，プログラムを構成する要素や部品をどのように分解し組み合わせるかという考え，すなわちパラダイムの進化と隣り合わせである．代表的なパラダイムとしてはオブジェクト指向や関数型が広く知られており，近年ではこれらを組み合わせた実装方法，つまりマルチパラダイムな実装方法が浸透しつつある．その中でもオブジェクト指向言語として登場した Java は，2014 年に関数型に基づく様々な記法（イディオム）を採用した．これにより，Java 開発者にとってはオブジェクト指向に加え，関数型の採用という選択肢が増えたといえる．しかしながら，Java を用いた実際の開発現場において関数型が採用されているか，また関数型のイディオムがどのように利用されているかは明らかになっていない．本研究では，Java を用いたオープンソースプロジェクト 100 個の約 13 万リビジョンを対象に，3 種類の関数型イディオム（ラムダ式と Stream，Optional）の利用実態を調査する．調査により，実際のプロジェクトのうちラムダ式を採用しているプロジェクトは 18%である一方，Stream を採用しているプロジェクトは 5%であり，Stream の利用は浸透していない事が示された．また，関数型イディオムを採用する主な理由は，可読性やパフォーマンスの向上のためであり，一方で関数型イディオムを採用しない理由は，JDK6/7 に対する後方互換性や保守性の維持のためであるという調査結果が得られた．

キーワード：関数型イディオム，オブジェクト指向，Java，ラムダ式，Stream，Optional

1. はじめに

プログラミング言語は常に進化し続けている [1], [2], [3]．この言語の進化は，プログラムを構成する要素や部品をいかに分解し組み合わせるかという考え，すなわちプログラミングパラダイムの進化 [4] と常に隣り合わせである．よって言語の進化には，糖衣構文の追加や文法の拡張といったイディオム（記法）単位の変化に限らず，別種のプログラミングパラダイムを採用するといった，実装の方針やスタイルそのものに変化を生じさせるような劇的な変化も含まれる．代表的なパラダイムとしてオブジェクト指向と関数型があり，この 2 つのパラダイムを組み合わせた実装方法，つまりマルチパラダイムな実装方法が普及してきている．

しかし，これまでオブジェクト指向言語による開発を行ってきた開発者が，実際に関数型のイディオムを使っているかは明らかとなっていない．開発者にとっては，オブジェクト指向に加え関数型の実装方法を採用するという選択肢が増えたことになる．一方で，オブジェクト指向をベースとした言語では純粋な関数型の機能を導入できないという批判も存在する [5]．加えて，オブジェクト指向と関数型は，完全に排他的ではないが大きく異なる考え方である．オブジェクト指向においてはデータと処理をオブジェクトとしてまとめ，オブジェクト同士が相互に作用することで処理を進めていく．しかし関数型ではデータと処理は分離されており，出力結果は入力によってのみ決定するという考えの元，副作用のないプログラムを開発する．

本研究では，オブジェクト指向を基にした言語である Java を題材に，Java における関数型イディオムを対象として，その利用実態に関する調査を行う．Java は 2014 年にリリースされた Java 8 において関数型イディオムを導入しており，オブジェクト指向をベースとする言語に対しての関数型の導入としては比較的最近行われたものである．そのため，関数型イディオムが開発現場の中でどれほど浸透しているのか，なぜ使う/使わないのかという利用実態を調査する上で適切な題材であると考えられる．また，対象とする関数型イディオムは Java 8 で導入された代表的な関数型イディオムであるラムダ式と Stream，および Optional の 3 つである．調査では以下に示す 3 つの Research Question（RQ）に答えることを目的とする．

[1] 大阪大学大学院情報科学研究科
[a)] h-tanaka@ist.osaka-u.ac.jp
[b)] shinsuke@ist.osaka-u.ac.jp
[c)] kusumoto@ist.osaka-u.ac.jp

- RQ1: 関数型イディオムは受け入れられているか
- RQ2: 関数型イディオムを採用する理由/採用しない理由は何か
- RQ3: 関数型イディオムはどのように使われているか

2. 準備

2.1 Java 8 の関数型イディオム

2.1.1 ラムダ式

ラムダ式は単一のメソッドを持つインターフェース（関数型インターフェース）の機能を簡潔に表現するためのイディオムであり，無名関数の代わりに関数型インターフェースの機能を実装できる [6]．以下にラムダ式の例を示す．

```
List<Integer> list = ...;
list.forEach(s -> System.out.println(s));
```

例に示すように，ラムダ式は引数，->記号，処理の本体の 3 つの要素で構成される．この例では Collection 型の list 変数の全ての要素に対して，要素を標準出力に書き出すというラムダ式を適用している．

ラムダ式を用いることで，無名関数と比べて記述を簡素化でき可読性の向上が見込まれる．また，型推論を用いることでさらに簡潔な記述が可能となる．加えて，ラムダ式はメソッドの引数として渡すことができる，つまり値ではなく処理内容をメソッドの引数に渡すことができる．

2.1.2 Stream

Stream は順次および並列なコレクション操作を実装するためのイディオムである．以下に Stream の例を示す．

```
List<Integer> list = ...;
list.stream()
    .filter(i -> i > 0)
    .forEach(i -> System.out.println(i));
```

Stream は 1 つの生成操作と複数の中間操作（Stream の中身を変更しない操作），1 つの終端操作（Stream の中身に作用する操作）から成る．例に示すように，Collection 型の変数に対して生成操作である `stream()` メソッドを呼び出すことで，Stream API を用いた Stream の利用が可能となる．さらに，生成した Stream に対し中間操作である `filter()` によって正の値のみを抽出後，終端操作により要素を出力している．

Stream を用いることで，簡潔かつ可読性の高い記述によって並列処理を実装することが可能となる．また，コードの抽象度が高くなるためコードの再利用が容易となる．

2.1.3 Optional

Optional は null もしくは null 以外のデータを持つオブジェクトである．以下に，Optional の例を示す．

```
List<Integer> list = ...;
int val = Optional.ofNullable(list.get(3))
                  .orElseGet(0);
```

この例では，変数 `list` の 3 番目の要素が null である可能性を明示しつつ，null の場合は 0 を，そうでない場合はその値そのものを val 変数に格納する．

Optional を用いることで，null になる可能性があることを明示することができる．また，値が存在しない場合の処理を強制することができる．つまり Optional を使うことで安全なプログラムを書くことができる．

2.2 Java の関数型イディオムに対する批判

デバッグが困難になる [7], [8]：Java の関数型イディオムには，使用することでメソッドの呼び出し系列（スタック）が深くなり，デバッグが困難になるという批判がある．

速度低下を招く場合がある [9]：Stream は処理内容によっては実行速度の低下を招くという批判がある．これは複数の処理を Stream を用いた並列処理で行う際に，一部の処理が他の処理に比べて実行時間がかかる場合に生じる問題である．こうした場合に Stream による並列処理を行うと，実行時間がかかる一部の処理以外で実行時間が増加してしまい，結果として全体の実行時間が増加してしまう．

メモリの観点では非効率である [10]：ラムダ式と Stream の利用はメモリという観点では非効率であるといった批判もある．ラムダ式や Stream を用いることで，イテレータや forEach を用いる場合に比べて，実行時間 1 秒あたりに発生するガベージコレクションは増加する．このようなガベージコレクションの増加はプログラムのパフォーマンスが損なわれる事に繋がってしまう．

純粋な関数型プログラミングの機能を実装できていない [5]：Java はオブジェクト指向というパラダイムを基本としているため，純粋な関数型プログラミングの機能を実装できていないという批判もある．Java に実装されていない機能として，関数型プログラミングのタプルという機能が存在する．タプルとは，様々な型のデータを持つことのできるデータ型であり，関数型プログラミングの代表的な機能である．しかしオブジェクト指向を基本とする Java では，タプルの導入によって抽象化が行われなくなる可能性から実装されていない．

3. Research Questions

本研究では，Java の関数型イディオムが実際の現場でどのように捉えられ扱われているのかを調査する．調査を行うにあたり以下に示す 3 つの Research Question を設定した．

RQ1: 関数型イディオムは受け入れられているか

2 節で述べたように，Java の関数型イディオムには利点だけでなく様々な批判も存在する．しかし，実際の開発現

場で Java の関数型イディオムがどの程度採用されているのかについては明らかにされていない．この RQ に答えることで，Java 8 リリース後約 4 年が経った現在，実際の Java プロジェクト開発現場で関数型イディオムがどの程度普及しているのかを知ることができる．

RQ2: 関数型イディオムを採用する理由/採用しない理由は何か

Java の関数型イディオムに対する批判がある中でも，関数型イディオムを利用している実際の開発プロジェクトは存在すると考えられる．こうしたプロジェクトがどのような理由で関数型イディオムを利用しているのかを定性的に調査する．また，Java の関数型イディオムに対して言われている批判が，実際の開発現場においてどの程度考慮されているのかも明らかではない．この調査の結果は，関数型イディオムの採用を検討するプロジェクトへの一つの指針となり得る．

RQ3: 関数型イディオムはどのように使われているか

Java を使う開発者が実際の開発現場でどのように関数型イディオムを使っているのかは明らかでない．実際の開発現場での関数型イディオムの使われ方を調査することで，他の開発者が関数型イディオムを使う場合の参考になると考えられる．

4. 調査方法

4.1 調査全体の流れ

全 RQ に対する調査の流れを図 1 に示す．調査は以下に示す手順に従って行う．

1. 調査対象 100 プロジェクトの抽出
2. 調査対象の全リビジョンから関数型イディオムを検出
3. 関数型イディオムの使用密度（詳細は後述）を算出
4. 全リビジョンを通しての使用密度の遷移をグラフ化
5. 最新リビジョンの使用密度に注目して RQ1 について調査
6. 使用密度が大きく変化する時期のコミットメッセージや issue に注目して RQ2 について調査
7. 最新リビジョンの関数型イディオムを抽出
8. 関数型イディオムの使い方に注目して RQ3 について調査

4.2 調査対象

調査対象として，GitHub でのスター順検索上位 100 個の Java プロジェクトを用いる．スター順検索の上位プロジェクトを対象とすることで，広く知られており，かつ開発規模の大きなプロジェクトに対して調査が行えると考えた．各プロジェクトにおける対象リビジョンは Java 8

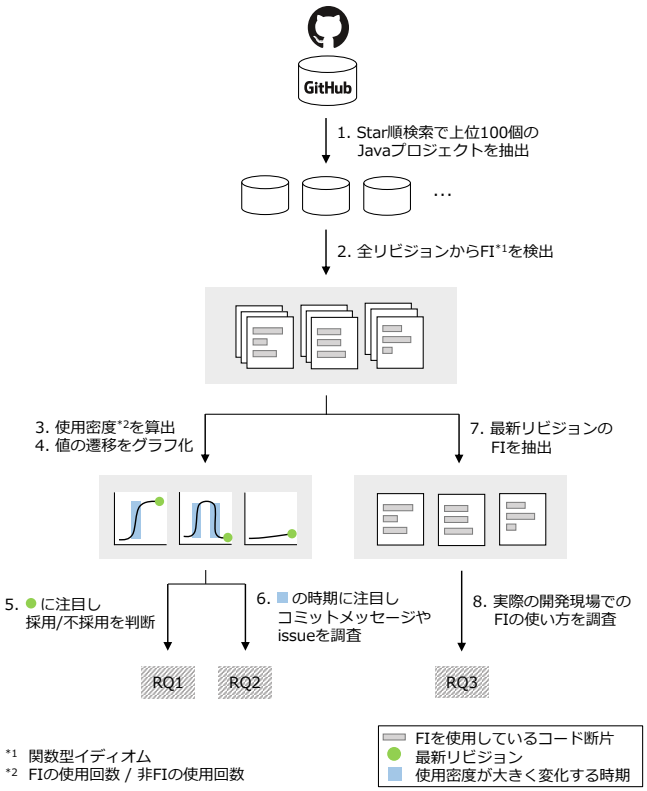

図 1 調査の流れ

リリース日の半年前（2013 年 9 月 18 日）から調査実施日（2018 年 4 月 10 日）までの全リビジョンである．対象とするファイルは各プロジェクトの各リビジョンにおける全 Java ファイルである．

4.3 指標

本研究では使用密度という指標を用いて調査を行う．使用密度は，3 種類の関数型イディオム（ラムダ式と Stream, Optional）それぞれについて算出する．算出は，関数型イディオムの使用回数を非関数型イディオムの使用回数で正規化することで行う．関数型イディオムと，それに対応する非関数型イディオムを表 1 に示す．以降，この使用密度を D_{idiom} と表記することとする．

4.4 RQ1 の調査の詳細

RQ1 の調査では，調査対象とするプロジェクトから算出した D_{idiom} を基に，最新リビジョンでの関数型イディオムの利用状況と，調査対象とした全リビジョンにおける D_{idiom} の値の遷移を調査する．

表 1 関数型イディオムと非関数型イディオムの対応

関数型イディオム	非関数型イディオム
ラムダ式	無名関数
Stream	for 文, while 文
Optional	null チェックを行う文

4.5 RQ2 の調査の詳細

RQ1 の結果に基づいて，前リビジョンから D_{idiom} が大きく変化するリビジョン（ラムダ式は 3%増減, Stream と Optional は 1%増減するリビジョンとする）のコミットメッセージや issue を定性的に調べることで，各プロジェクトでの関数型イディオムを利用する理由，利用しない理由を調査する．調査対象リビジョンは，ラムダ式で 87 リビジョン，Stream で 53 リビジョン，Optional で 69 リビジョンである．また，調査は目視によって行う．

上記の方法に加えて，さらに広く調査を行うために，GitHub の検索クエリを組み合わせてコミットメッセージや issue，コメント文を調査する．具体的には，下記に示すようなクエリの組み合わせにより調査を行った．

```
Scope  = ("java 8" OR "java8")
Fi     = ("lambda" OR "stream" OR "optional")
Action = ("use" OR "accept" OR "remove" OR
          "replace" OR "reason")
Query  = (Scope OR Fi) AND Action
```

なお，この方法においては，GitHub 上の全 Java プロジェクトに対してクエリを用いた検索を行い，その検索結果から 100 個のコミットを調査対象とする．

4.6 RQ3 の調査の詳細

RQ3 の調査では，最新リビジョンにおいて抽出された関数型イディオムについて，どのような使い方をしているのか調査する．最新リビジョンのみに注目するのは，最新状態での使い方が，プロジェクトにおいて受け入れられている使い方であると考えたためである．調査は，AST による静的ソースコード解析によって行う．AST を構築するに際して，IDE の一つである Eclipse においてプラグインとして提供されている JDT（Java Development Tools）を使用している．

調査項目と，各調査項目に対しての具体的な調査方法を説明する．

ラムダ式のステートメント数：ラムダ式の処理内容である本体部分中からステートメントを検出し，その数を計測する．カウントするステートメントは，JDT を用いて作成される AST のノードの中で，Statement という単語を含むノードを対象とする．

Stream API における各メソッドの使用事例数：AST を用いた静的ソースコード解析により，Stream API で提供されているメソッドを検出し，それぞれのメソッドが使用されている事例数を計測する．

Stream のメソッドチェーン数：1 つの Stream に対してメソッドチェーンを用いて呼び出されているメソッドの数を計測する．ただし，呼び出されたメソッドの引数中で用いられているメソッドチェーンについては計測対象ではないとする．また，`stream()` メソッドや `of()` メソッドのような Stream を生成するメソッドについては計測対象外とし，`stream()` メソッドや `of()` メソッドの次に呼び出されているメソッドから計測開始する．

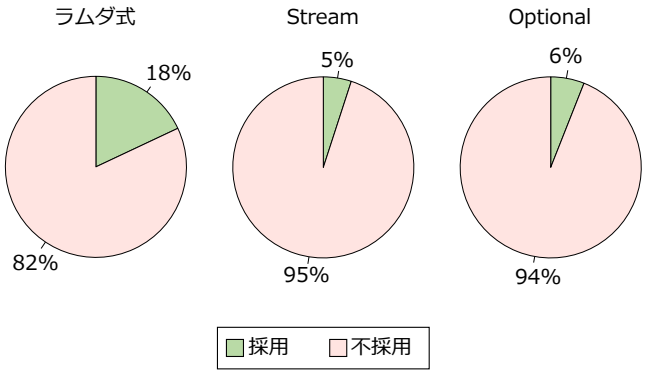

図 2　100 プロジェクトの最新リビジョンにおける採用/不採用の割合

Optional におけるメソッドの使用事例数：Java の Optional パッケージとして提供されているメソッドの使用回数をそれぞれカウントし，各メソッドが使用されている事例数を計測する．

5. 調査結果

5.1 RQ1 の調査結果

RQ1: 関数型イディオムは受け入れられているか

最新リビジョンでの D_{idiom}：調査対象とした 100 プロジェクトを採用/不採用に分類した結果を図 2 示す．「採用」の定義は，最新リビジョンでの D_{idiom} が 1%以上のプロジェクトとし，それ以外のプロジェクトを「不採用」とする．つまり，最新リビジョンにおいて 1 か所でも関数型イディオムを使用している場合は，そのプロジェクトを採用とみなす．図に示す円グラフは左から，ラムダ式，Stream，Optional についての採用と不採用の割合を示している．

図 2 に示す結果より，採用プロジェクトの割合が最も高いのはラムダ式であり 18%である．一方，Stream の採用プロジェクトは 5%であり，調査対象とする関数型イディオムの中で最も採用の割合が低く，ラムダ式の 3 分の 1 以下の値である．

各リビジョンにおける D_{idiom} の遷移：各リビジョンにおける 3 つの関数型イディオムそれぞれの D_{idiom} の遷移を図 3 に示す．ただし，図 3 には最新リビジョンにおける D_{idiom} の値が高い上位 5 プロジェクトの結果のみを示している．いずれの図も横軸は各リビジョンに対応する日付を，縦軸は D_{idiom} を示している．

まず，ラムダ式は他 2 つの関数型イディオムに比べて，いずれの時期においても D_{idiom} の値が高く，広く使用されて

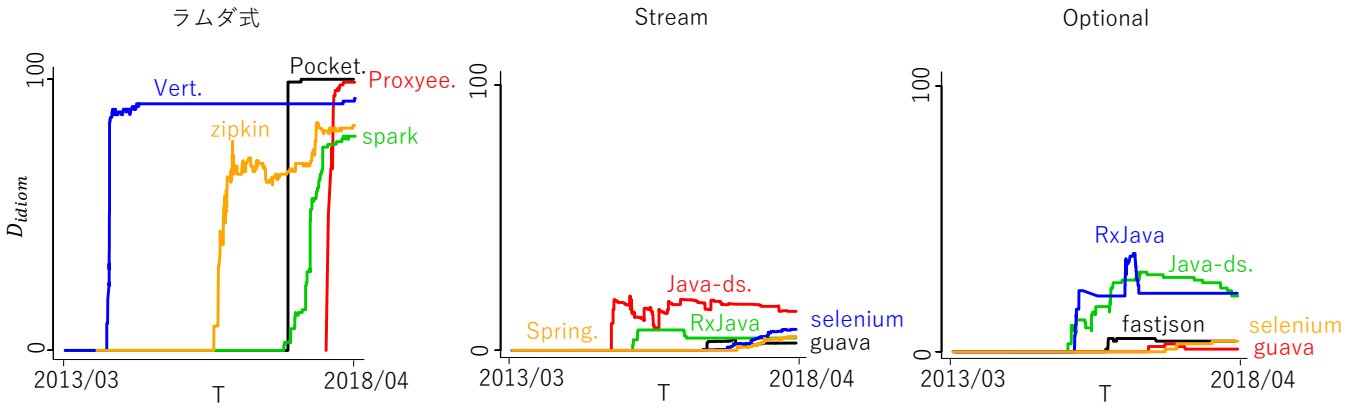

図 3 調査対象プロジェクトの各リビジョンにおける D_{idiom}
(最新リビジョンでの D_{idiom} 上位 5 つのプロジェクトのみ記載)

いることが分かる．また，Vert. は Java 8 リリース後まもなく急激にラムダ式を導入しており，Pocket. や Proxyee. はプロジェクト開始時点からラムダ式を急速に導入している．このようなプロジェクトは，ラムダ式に対して強い関心を持っているプロジェクトであると言える．

Stream において最も高い D_{idiom} の値を示しているプロジェクトは Java-ds. であるが，その値は 15%程度である．また，Java-ds. は D_{idiom} の値が大きく変動する時期が複数存在する．このような時期に，プロジェクトの方針として良しとしない関数型イディオムの使い方をリファクタリングしているのではないかと考えられる．

Optional に関するグラフの中でも，RxJava は 2015 年ごろから Optional を使用し続けていたが，ある時期に一部の使用を取り消していることが分かる．このような時期には，Optional を使用しない理由をプロジェクト内で議論しているのではないかと考えられる．

ラムダ式において D_{idiom} の値が 100%近いプロジェクトが多い理由は，無名関数は比較的最近導入されたイディオムであり，使用される事例が他イディオムに比べて少ないことが原因である可能性がある．一方で，Stream や Optional に対応する非関数型イディオムは for,while 文と null チェック文，つまり Java 8 以前から普及しているイディオムであるため，D_{idiom} の値は低くなると考えられる．

5.2 RQ2 の調査

> RQ2: 関数型イディオムを採用する理由/採用しない理由は何か

関数型イディオムの採用理由：RQ2 の調査に対する調査結果として，関数型イディオムを採用する理由を表 2 に示す．guava がラムダ式と Stream を採用する理由は「Stream を使うことで Guava のパフォーマンスを向上させるため」であるが，具体的にどのようなパフォーマンスを向上させるのかは明記されていない．realm-java がラムダ式を採用する理由は，realm-java がサポートしようとしている RxJava2 が Java の関数型イディオムを使用していることにより，realm-java でも関数型イディオムを使う必要があるためである．selenium がラムダ式と Stream を採用する理由と，spark がラムダ式を使う理由は，「コードの記述量を減らすため」である．また，selenium がラムダ式を使うもう一つの理由として，「最終的な jar ファイルのサイズを大きくし過ぎないため」ということも挙げている．ラムダ式は無名関数と異なり，コンパイル時に新たなクラスを作成することがない．そのため，ラムダ式に書き換えることで jar ファイルのサイズが小さくなる．retrofit が Optional を採用する理由は，retrofit が内部で使用するツールにおいて Optional でラップした値を扱うことから，Optional で値をラップするコンバータを作るためである．

関数型イディオムの不採用理由：RQ3 の調査に対する調査結果として，関数型イディオムを不採用とする理由を表 3 に示す．GraalVM が Stream を採用しない理由は「スタックオーバーフローの発生が早くなる」である．Stream は他の繰り返し処理と比べて発生させるスタックフレームが多く，スタックのオーバーフローを早く起こしてしまうため GraalVM では Stream を採用していない．guava はラムダ式を使わない場合があり，その理由として「ラムダ式を使うことで例外処理の扱いが複雑になるため」と挙げている．ラムダ式を用いる場合，ラムダ式の中で起きた例外はラムダ式の外から直接扱うことはできない．そのため，一度ラムダ式の中で例外をラップし，ラムダ式の外でラップされた例外を読み取って処理を行う必要がある．このような複雑な記述を防ぐために，guava ではラムダ式を使わない場合がある．Hystrix と lottie-android, RxJava が関数型イディオムを採用しない理由は，JDK6 や JDK7 に対する後方互換性を維持するためである．selenium は，メソッド引数内で使用される Optional は記述を複雑にすることを理由に，Optional の使用を制限している．Optional を用いる利点の一つとして値が null であることによるエラー

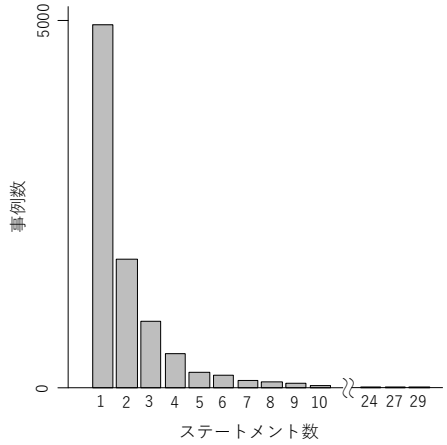

図 4 Lambda のステートメント数

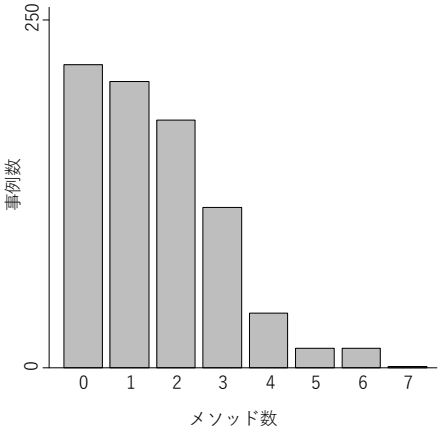

図 5 Stream においてメソッドチェーンにより呼び出されるメソッド数

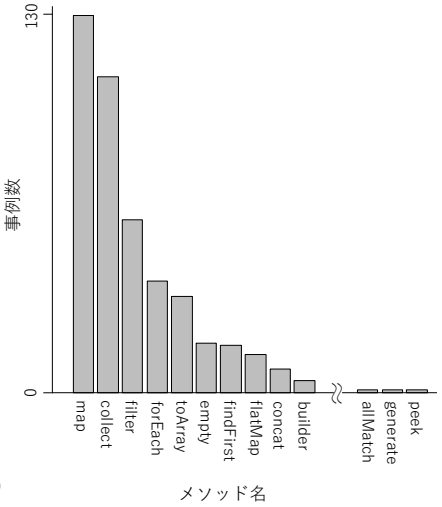

図 6 Stream API の各メソッドの使用事例数

を防ぐ事が挙げられるが，メソッド引数内で Optional を使用した場合はこのエラーを防ぐことができない．そのため，記述を簡素化するために selenium ではメソッド引数内で Optional を使用していない．

5.3 RQ3 の調査結果

RQ3: 関数型イディオムはどのように使われているか

ラムダ式のステートメント数：ラムダ式の本体部分中に存在するステートメント数を計測した結果を図 4 に示す．図 4 に示すグラフの横軸はステートメント数，縦軸は各ステートメント数に該当する事例の数である．この結果から，ラムダ式の本体部分が 1 行のステートメントで構成されている事例が一番多く，二番目に多い事例である 2 行ステートメントで構成されるラムダ式の 2 倍以上の事例数であることが分かる．また，最大で 29 個のステートメントで構成されるラムダ式も 1 件存在する．

Stream のメソッドチェーン数：図 5 に Stream においてメソッドチェーンを用いて呼び出されるメソッドの数を計測した結果を示す．グラフの横軸はメソッドチェーンによって呼び出されたメソッド数，縦軸は各メソッド数の事例数である．調査結果より，メソッドチェーンによって呼

*1 https://groups.google.com/forum/#!topic/guava-announce/o954PqvaXLY/discussion
*2 https://github.com/realm/realm-java/commit/9ac68
*3 https://github.com/square/retrofit/commit/e985d
*4 https://github.com/google/guava/issues/1670
*5 https://github.com/SeleniumHQ/selenium/issues/4867
*6 https://github.com/SeleniumHQ/selenium/commit/4c38c0
*7 https://github.com/oracle/graal/commit/bca7c
*8 https://github.com/google/guava/issues/1670
*9 https://github.com/Netflix/Hystrix/commit/e102e
*10 https://github.com/airbnb/lottie-android/commit/fa239
*11 https://github.com/ReactiveX/RxJava/commit/000a1
*12 https://github.com/SeleniumHQ/selenium/commit/4c38c

表 2 関数型イディオムの採用理由

プロジェクト	関数型イディオム	日付	採用理由
guava	ラムダ式，Stream	2016/11/05	Stream を使うことで guava のパフォーマンスを向上させるため*1
realm-java	ラムダ式	2017/09/12	関数型イディオムを使用している RxJava2 に合わせるため*2
retrofit	Optional	2017/03/12	Optional へのコンバータを作るため*3
selenium	ラムダ式	2014/11/01	最終的な jar ファイルのサイズを大きくし過ぎないため*4
selenium	ラムダ式，Stream	2017/12/03	コードを簡潔に書くため*5
spark	ラムダ式	2014/04/07	コードを簡潔にするため*6

表 3 関数型イディオムの不採用理由

プロジェクト	関数型イディオム	日付	不採用理由
GraalVM	Stream	2014/09/09	スタックをオーバーフローさせるのが早いため*7
guava	ラムダ式	2014/11/01	ラムダ式を使うことで例外処理の扱いが複雑になるため*8
Hystrix	ラムダ式, Stream, Optional	2016/08/19	JDK6/7 でビルド可能にするため*9
lottie-android	ラムダ式	2017/04/08	ビルドやインストールで問題が発生しないようにするため*10
RxJava	ラムダ式, Stream, Optional	2016/02/04	JDK6 への互換性のため*11
selenium	Optional	2017/03/30	引数内での Optional はメソッドを呼び出しにくくするだけ*12

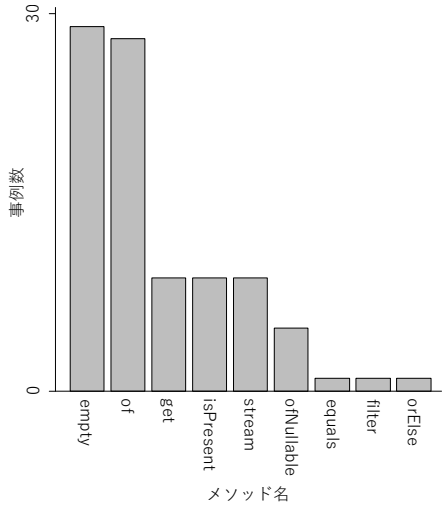

図 7 Optional における各メソッドの使用回数

び出されるメソッド数が 0, つまり `stream()` メソッドや `of()` メソッドによって Stream を生成するだけの事例が 218 件，次いで 1 つだけメソッドを呼び出す事例が 206 件であることが分かる．一方で，7 つのメソッドからなるメソッドチェーンが 1 件あることが分かる．

Stream API における各メソッドの使用事例数：図 6 に Stream API の各メソッドの使用事例数を計測した結果を示す．グラフの横軸はメソッド名，縦軸は各メソッドの事例数である．この調査結果から，Stream の中間操作では `map()` メソッドの事例が最大の 129 件，終端操作では `collect()` メソッドの事例が最大の 108 件であることが分かる．一方で，中間操作では `peek()` メソッドの事例が最小で 1 件，終端操作では `allMatch()` メソッドの事例が最小で 1 件であることが分かる．`peek()` メソッドは，メソッドが呼び出された時点での Stream をそのまま返すメソッドであるが，これはデバッグのために用意されたメソッドである．そのため，最終的なソースコードには現れにくく，使用事例が少ないのではないかと考えられる．また，`allMatch()` メソッドは Stream 中の全要素が条件に合致する場合に `true` を，それ以外の場合に `false` を返すメソッドである．ただし `allMatch()` メソッドは Stream の要素が空の場合は `true` を返す仕様であるため，開発者の想定とは異なる動作を起こす可能性がある．これにより，使用している事例が少ないのではないかと考えられる．

Optional におけるメソッドの使用割合：図 7 に，Optional の各メソッドの使用回数を計測した結果を示す．図 7 に示すグラフの横軸はメソッド名，縦軸は各メソッドの事例数である．調査結果より，Optional におけるメソッドのうち，最も使用されているのは `empty()` メソッドで 29 件の事例が存在した．使用事例数が 1 件のみの `equals()` メソッドは，2 つの Optional オブジェクトが等しいかを判定するメソッドである．使用事例が少ない原因として，Optional オブジェクト同士でなく，Optional から取り出した要素同士で等価判定を行う場合が多いのではないかと考えられる．同様に 1 件の事例のみ使用していた `orElse()` メソッドは，Optional の要素が存在しない場合に引数として与えられた値を返すメソッドである．しかし，`orElse()` メソッドは Optional の値が存在する場合にも呼び出されてしまうため，予期しない結果が生じる可能性がある．そのため，使用事例数が少ないのではないかと考えられる．

6. 議論

6.1 RQ1 に対する議論

図 2 に示す結果から，各関数型イディオムを採用しているプロジェクトの割合は，最も大きな値でもラムダ式を採用しているプロジェクトの 18%であり，Stream と Optional を採用しているプロジェクトは 100 個のプロジェクト中の 6%以下である．以上の事実から，関数型イディオムは実際の開発現場で広く受け入れられているとは言いきれない．

> **RQ1 の結論**
> 実際の Java プロジェクトの開発現場において関数型イディオムが浸透しているとは言えない．

6.2 RQ2 に対する議論

5.2 節に示す調査の結果より，関数型イディオムを採用しているプロジェクトの採用理由は 3 種類ある．1 つ目の採用理由は，「プログラムを扱いやすくするため」であると言える．これは，selenium と spark が採用理由として挙げている「コードの記述量を減らす」という理由や，selenium がラムダ式を採用するもう一つの理由である「jar ファイルのサイズを大きくし過ぎない」といった事から言える．2 つ目は，guava が Stream を採用する理由として示している「パフォーマンスを向上させるため」という理由である．一方で，「Java の Stream は速度が遅い」という批判もある [9]．guava の示した採用理由には具体的にどういったパフォーマンスの向上を目的としているのかは記述されていないため，速度以外でのパフォーマンスが向上するのではないかと考えられる．3 つ目の採用理由として，realm-java や retrofit のような「関数型イディオムを使用している他のツールとの互換性のため」という理由が挙げられる．

一方で，関数型イディオムを採用しない理由も 3 種類ある．1 つ目は JDK 6 や JDK 7 もしくはそれらを使用しているツールをサポートするため，つまり「後方互換性のため」であると言える．2 つ目は関数型イディオムを使用することによりスタックが深くなり，デバッグが行いにくくなることを防ぐ，つまり保守性を維持するためであると言える．また，「保守性（デバッグの行いやすさ）の維持」という不採用理由は，Java の関数型イディオムを使うことでスタックが深くなりデバッグが困難になるという批判 [7]

を支持する事例であると言える．3 つ目の不採用理由として，guava が挙げているような「例外処理への対応を簡単にするため」という理由である．このことから，関数型イディオムを用いる場合，正常系の処理は簡潔に記述できるが，異常系の場合は複雑な記述になると言える．

RQ2 の結論
関数型イディオムを採用する理由

- プログラムを扱いやすくするため
- パフォーマンス（速度以外）を向上させるため
- 関数型イディオムを使用しているツールへの互換性のため

関数型イディオムを採用しない理由

- JDK 7 以前への後方互換性のため
- デバッグ面での保守性を維持するため
- 異常系の処理が複雑になるのを防ぐため

6.3 RQ3 に対する議論

図 4 に示す結果より，単一ステートメントで構成されるラムダの事例が多い．一般的にはラムダ式はネストが深くなく，なおかつ 1 行で書かれることが望ましいと言われているため [11]，実際の開発者も簡潔にラムダ式を書くことを意識していると考えられる．しかし一方で 20 個以上のステートメント数で構成されるラムダ式も複数存在する．また，図 7 の結果より，一般的には使わない方が良いとされている `get()` メソッドや `isPresent()` メソッドであるが [12], [13]，実際には使用されている事例が複数存在する．このような推奨されていない使い方について，コードの書き換えを提案するツールが必要であると考えられる．

RQ3 の結論
ラムダのステートメント数や Stream のメソッドチェーン数は，短く簡潔な記述を意識した使用が多い．ただし，一般的に使用を避けるべきであるとされるメソッドの使用は一定の事例数が存在するため，開発者に書き換えを提案する仕組みが必要である．

7. おわりに

本研究では，実際の開発現場における Java の関数型イディオムの調査として，3 つの RQ を設定して分析を行った．調査の結果，プロジェクトの方針として重点を置くポイントが，パフォーマンスやプロジェクトの扱いやすさなのか，保守性を低下させないことなのかによって関数型イディオムの利用を選択すればよいと言える．また，開発者は関数型イディオムを使用する際に簡潔な記述を意識する一方で，使わない方が良いメソッドにも注意すべきである．

今後の課題として，実際の開発現場での関数型イディオムの使い方をさらに調査することで，良くない関数型イディオムの使い方を定義することが考えられる．これにより，開発者に書き換えを提案するツールの作成が可能となる．また，関数型イディオムを使った場合のスタックトレースについて，関数型イディオムによって裏側で呼び出されたメソッドによるスタックと，そうでないスタックを区別できる仕組みの開発が考えられる．これによりスタックが深くなることでのデバッグが行いにくくなる問題を解決できる可能性があると考える．

謝辞 本研究の一部は，日本学術振興会科学研究費補助金基盤研究（B）（課題番号：16H02908，18H03222）の助成を得て行われた．

参考文献

[1] Favre, J. M.: Languages evolve too! Changing the software time scale, *Proceedings of the 8th International Workshop on Principles of Software Evolution*, pp. 33–42 (2005).
[2] Landin, P. J.: The Next 700 Programming Languages, *Communications of the Association for Computing Machinery*, Vol. 9, No. 3, pp. 157–166 (1966).
[3] Spinellis, D., Louridas, P. and Kechagia, M.: The Evolution of C Programming Practices: A Study of the Unix Operating System 1973–2015, *Proceedings of the 38th International Conference on Software Engineering*, pp. 748–759 (2016).
[4] Simmonds, D. M.: The Programming Paradigm Evolution, *Computer*, Vol. 45, No. 6, pp. 93–95 (2012).
[5] Saumont, P.: What's Wrong in Java 8, Part V: Tuples - DZone Performance, `https://dzone.com/article/whats-wrong-java-8-part-v`.
[6] Warburton, R.: *Java 8 Lambda Functional Programming for the Masses*, O'Reilly Media (2014).
[7] Fischer, R.: *Java Closures and Lambda*, chapter 7, Apress. (2015).
[8] Weiss, T.: The Dark Side Of Lambda Expressions in Java 8 — OverOps Blog, `https://blog.takipi.com/the-dark-side-of-lambda-expressions-in-java-8/`.
[9] Zhitnitsky, A.: The 6 biggest problems of Java 8 - JAXenter, `https://jaxenter.com/java-8-problems-112279.html`.
[10] Cheon, Y. and Torre, A.: Impacts of Java Language Features on the Memory Performances of Android Apps, Technical report, University of Texas at El Paso (2017).
[11] Subramaniam, V.: Java 8 idioms: Why the perfect lambda expression is just one line, `https://www.ibm.com/developerworks/library/j-java8idioms6/index.html`.
[12] Winnicki, M.: Optional isPresent() Is Bad for You - DZone Java, `https://dzone.com/articles/optional-ispresent-is-bad-for-you`.
[13] Gioiosa, M. P.: Java 8 Optional - Replace Your Get() Calls - DZone Java, `https://dzone.com/articles/java-8-optional-replace-your-get-calls`.