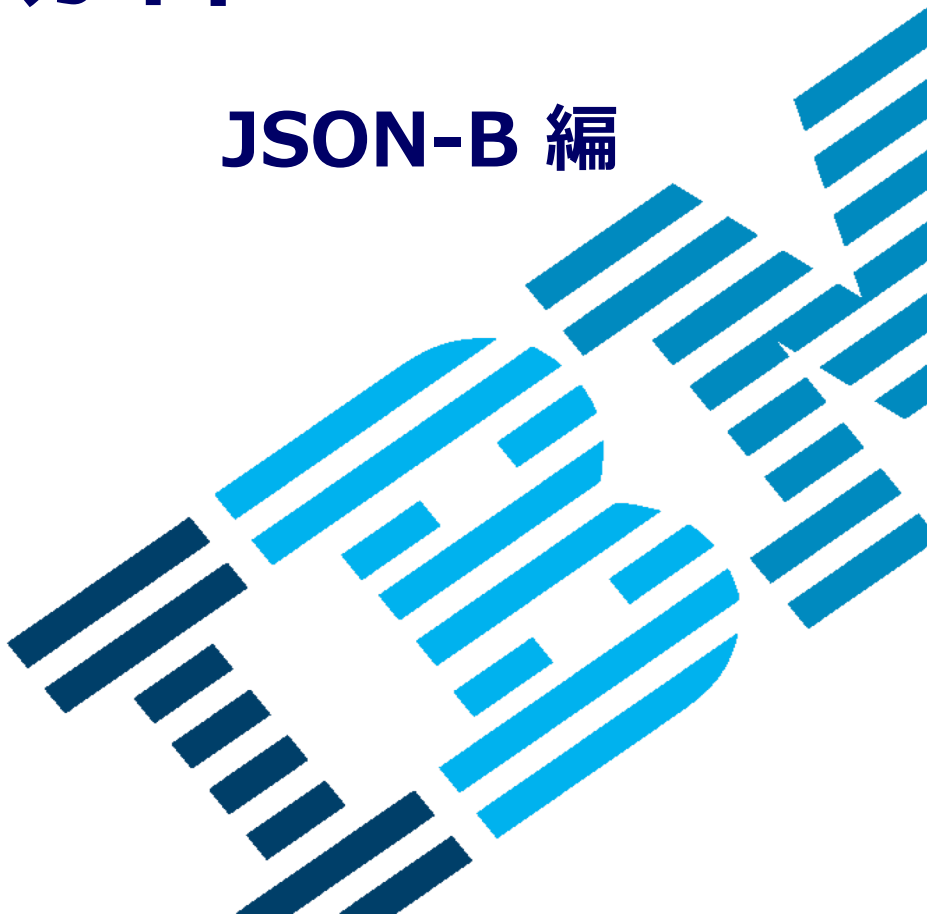
Java EE 8 アプリケーション設計ガイド

JSON-B 編



日本アイ・ビー・エム　システムズ・エンジニアリング株式会社

Disclaimer

- この資料は日本アイ・ビー・エム株式会社ならびに日本アイ・ビー・エム　システムズ・エンジニアリング株式会社の正式なレビューを受けておりません。
- 当資料は、資料内で説明されている製品の仕様を保証するものではありません。
- 資料の内容には正確を期するよう注意しておりますが、この資料の内容は2019年3月現在の情報であり、製品の新しいリリース、PTFなどによって動作、仕様が変わる可能性があるのでご注意下さい。
- 今後国内で提供されるリリース情報は、対応する発表レターなどでご確認ください。
- ＩＢＭ、ＩＢＭロゴおよびibm.comは、世界の多くの国で登録されたInternational Business Machines Corporationの商標です。他の製品名およびサービス名等は、それぞれＩＢＭまたは各社の商標である場合があります。現時点でのＩＢＭの商標リストについては、www.ibm.com/legal/copytrade.shtmlをご覧ください。
- 当資料をコピー等で複製することは、日本アイ・ビー・エム株式会社ならびに日本アイ・ビー・エム　システムズ・エンジニアリング株式会社の承諾なしではできません。
- 当資料に記載された製品名または会社名はそれぞれの各社の商標または登録商標です。
- JavaおよびすべてのJava関連の商標およびロゴは Oracleやその関連会社の米国およびその他の国における商標または登録商標です。
- Microsoft, Windows および Windowsロゴは、Microsoft Corporationの米国およびその他の国における商標です。
- Linuxは、Linus Torvaldsの米国およびその他の国における登録商標です。
- UNIXはThe Open Groupの米国およびその他の国における登録商標です。

本ガイドについて

- 本ガイドはJava EE 8における新機能、JSON-Bに関するガイドです
- 本ガイドにおけるサンプルはすべてLiberty 環境で実装しております
- 実装環境詳細
  - Eclipse Java EE IDE for Web Developers
    - version: 2018-12
  - Java version 1.8.0_201
  - Mac OS X
  - WebSphere Application Server 19.0.0.1
  - WAS Liberty with Java EE 8 Full Platform
  - Servlet 4.0を使用

- 利用方法：javaee-8.0 フィーチャーを記述することでjsonb-1.0も同時にインストールされる

server.xml フィーチャー

```
<featureManager>
    <feature>javaee-8.0</feature>
</featureManager>
```

サーバー起動

Console

```
CWWKF0012I: サーバーは次のフィーチャーをインストールしました。
... ,servlet-4.0 ...,jaxrs-2.1 ..., jsonb-1.0...
```

目次

1. JSON-B概要
    1-1. JSON-BのJavaEE 8における位置付け
    1-2. JSON-B概要
    1-3. JSON-Bの構成
    1-4. JSON-B 実装イメージ
    1-5. JSON-P と JSON-B

2. JSON-B機能
    2-1. シリアライズ
    2-2. デシリアライズ
    2-3. JSON-B のデフォルト設定

3. JSON-Bカスタマイズ
    3-1. カスタマイズで可能なこと
    3-2. カスタマイズ方法
    3-3. アノテーションによるカスタマイズ
    3-4. JsonbConfigインスタンスの利用
    3-5. JsonbAdapter, JsonbSerializerの実装
        3-5-1. JsonbSerializerの実装
        3-5-2. JsonbDeserializerの実装

1. JSON-B概要

1-1. JSON-BのJavaEE 8における位置付け

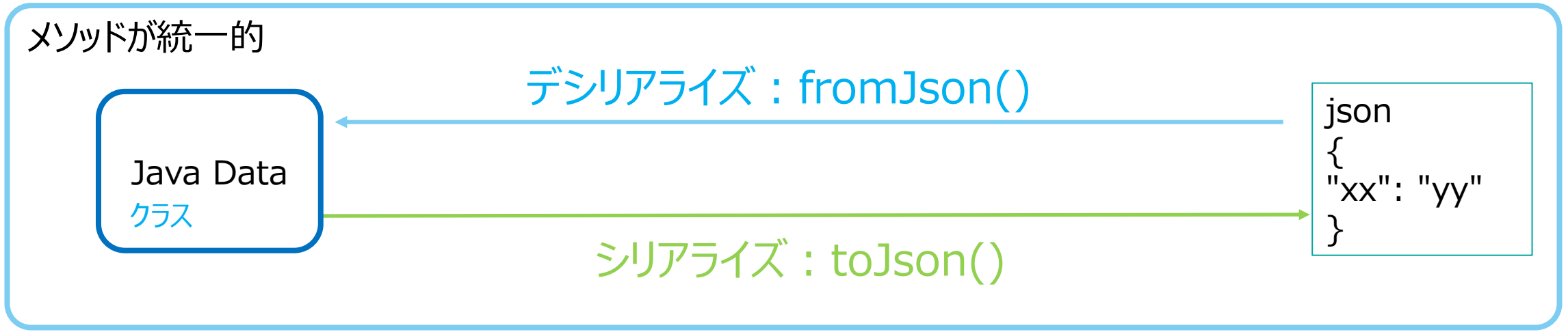
- **Java EE(今まで)**
  - クライアントとのデータのやり取りにはXML中心に扱っていた
  - JSONデータの組み込みには長けていなかった

- 必要とされる背景
  - 迅速にビジネスに対応していくことが必要
    - RESTfulなWebサービスが主流に
    - データのやりとりは、JSONフォーマットが主流に
    - JSONを扱う画面側の開発とサーバー(Java)から送信されるJsonの整合性に柔軟に対応する必要がある
  - **Javaのデータ型とJSON型で双方向に簡潔かつ柔軟に変換できる機能が必要**
- **Jackson -> JSON-B**
  - Jacksonのライブラリ等を導入することなく、標準でJSONを扱えるように
  - Javaデータと、JSONのバインディングの相互変換をワンステップで行うことが可能に
  - 一般的な業界の慣例と方法を体系化し、日常的な開発・実装のシナリオに適したものに
  - JSONデータ送信先との整合性を踏まえ、柔軟にカスタマイズすることが可能に

JSON-B：Java EE 8におけるJsonデータ処理のプラットホーム

1-2. JSON-B概要

- JavaデータからJSONドキュメントへの変換、JSONドキュメントからJavaデータへの逆変換を単純かつ統一的に行える
  - JSONをクラスに:fromJson(), クラスをJSONに:toJson()



- 直感的にわかりやすい形でサポートされており、JSONを扱ったことがない開発者も簡単に理解可能
- Javaクラス、JSONドキュメントの相互変換を自由にカスタムマッピングすることが可能

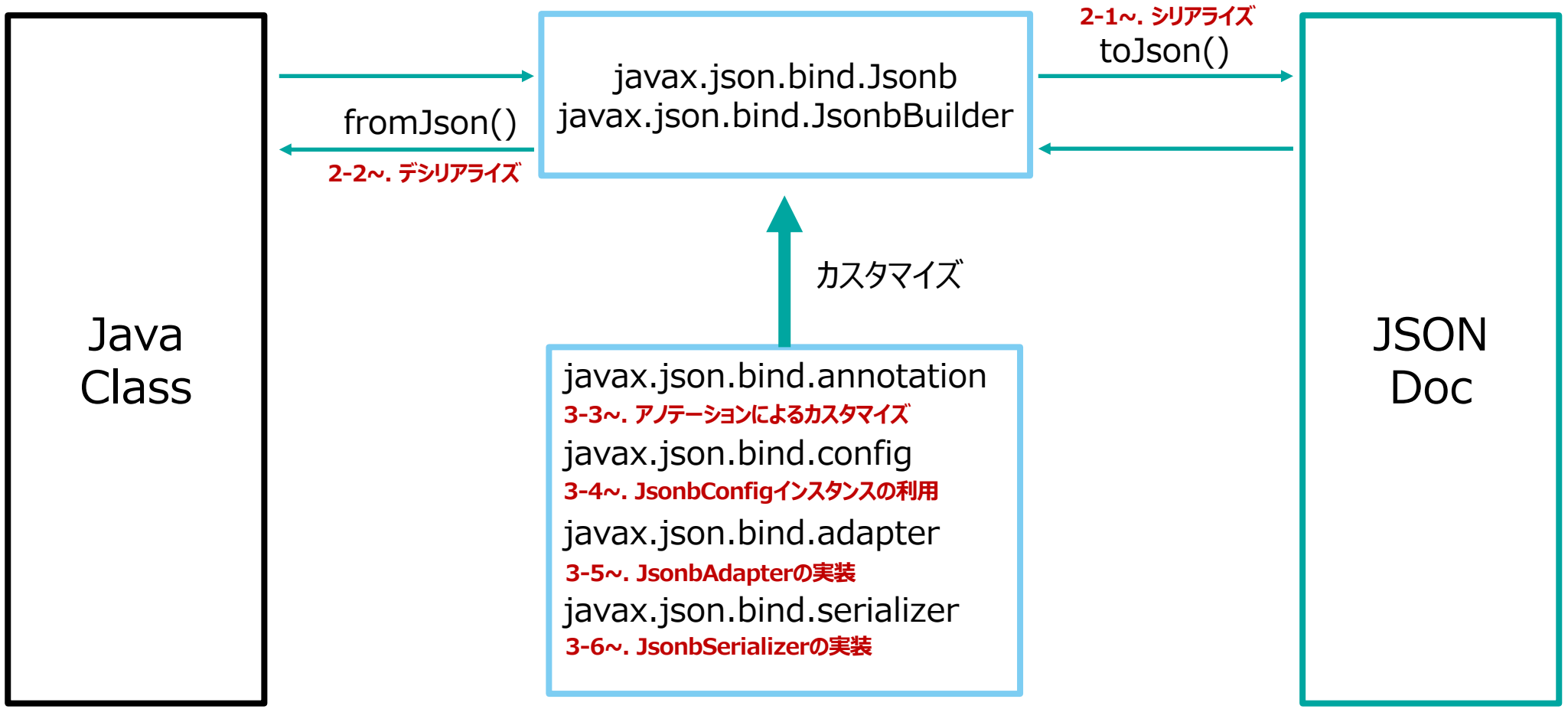
1-3. JSON-Bの構成

- 下記のようなパッケージで構成されている

Package	API(Exceptionは省略)	Description
javax.json.bind	Interface: Jsonb, JsonbBuilder	toJson()やfromJson()といったシリアライズ、デシリアライズするためのInterface
javax.json.bind.adapter	Interface: JsonbAdapter	JsonbAdapter Interfaceを継承し実装することで、カスタムマッピングを行う
javax.json.bind.annotation	Interface: JsonbAnnotation	アノテーションを付与することで出力形式をカスタマイズする
javax.json.bind.config	Interface: PropertyNamingStrategy, PropertyVisibility Strategy	出力順序やフィールドの可視性をカスタマイズする
javax.json.bind.serializer	Interface: DeserializeContext, JsonbDeserializer<T> JsonbSerializer<T>, SerializationContext	JsonbSerializerまたはJsonbDeserializer Interfaceを継承し、実装することでカスタムマッピングを.
javax.json.bind.spi	JsonbProvider (本資料では解説省略)	JsonbBuilderの機能を補完するためのプラグイン

1-4. JSON-B 実装イメージ

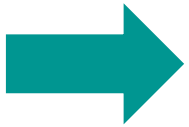
- 基本的にJsonb、JsonbBuilderインターフェースを用いてシリアライズ(toJson()),デシリアライズ(fromJson())を行う
- 出力形式をカスタマイズしたい時はadapter, serializer,annotation, configパッケージを用いる



1-5. JSON-PとJSON-B

- JSON-P(JSON-Processing)
  - JSONデータを0から生成をする

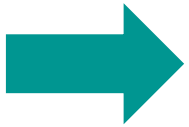
```
JsonObject js = Json.createObjectBuilder()
    .add("name", "Fukui")
    .add("age", 22)
    .add("isPassed", Boolean.FALSE)
    .build();
  String result = js.toString();

  System.out.println(result);
```



```
{
   "name":"Fukui",
    "age":22,
    "isPassed":false
}
```

- JSON-B(JSON-Binding)
  - JSON-Pを元に設計されている
  - JavaオブジェクトとJSONドキュメントを紐付ける
  - JSON-Pと組み合わせることで、JSONドキュメントのマッピングを柔軟にカスタマイズ可能

```
public class Test {
   public String dog = "chihuahua";
   public int angle = 45;
   public String bread = "naan";
   public String car = "Ford";
}
```

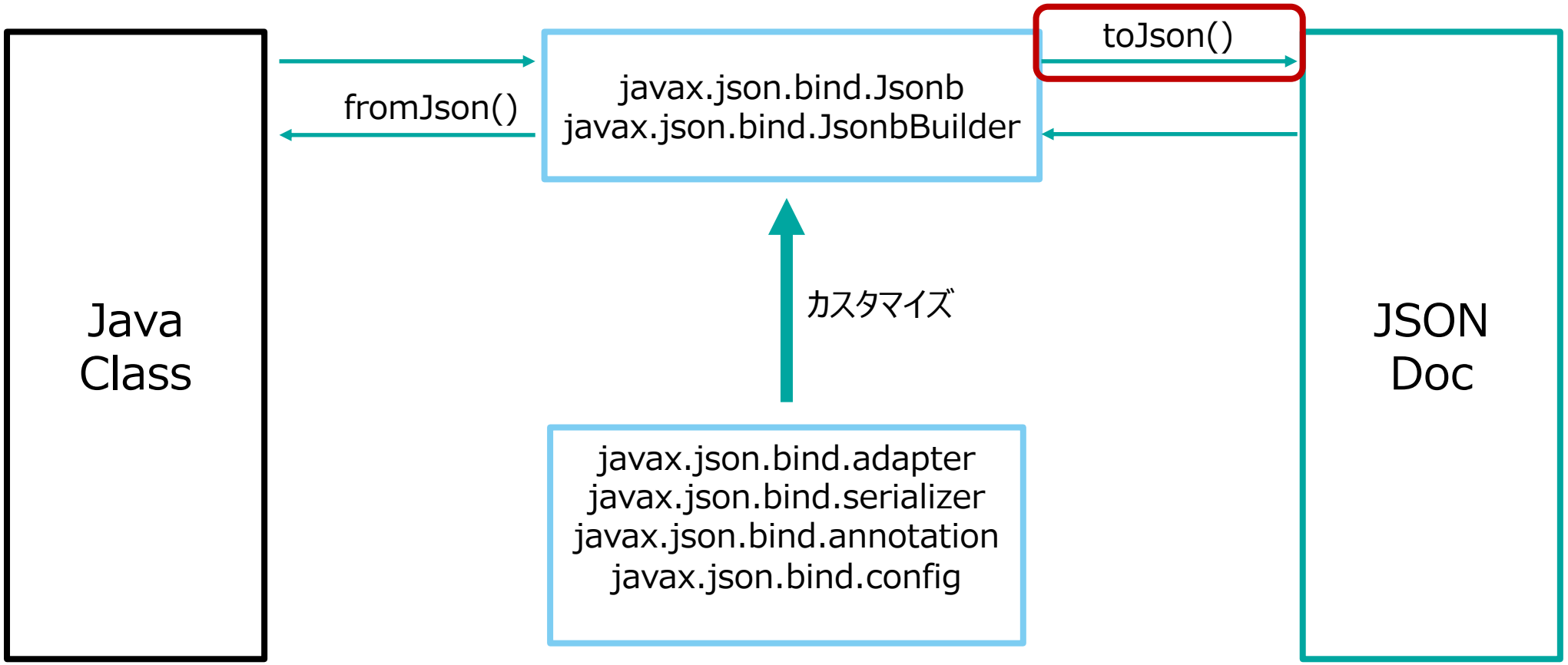


```
{
   "angle":45,
   "bread":"Chihuahua",
   "car":"Ford",
   "dog":"Labradoodle"
}
```

JSON-B機能

2-1. シリアライズ

- JavaクラスからtoJson()を用いてJSONドキュメントにマッピング
- Jsonbインターフェース、JsonbBuilderインターフェースを用いる



2-1. シリアライズ

- シリアライズの流れ

① シリアライズするJavaクラス、インスタンスを作成

ex）Employeeクラスのインスタンスをシリアライズする

Empoloyee クラス		
フィールド		インスタンス化
String	id	a1234
String	name	fukui
Stiring	department	cloud

② JsonbBuilder のstatic メソッド, create()を利用しJsonb型のインスタンスを作成

```
Jsonb jsonb = JsonbBuilder.create()
```

③ ②で作成したJsonbインスタンスが持つtoJson()メソッドの引数に①のインスタンスを与える

```
Jsonb.toJson( Empoloyee インスタンス )
```

```
{
    "department": "cloud",
    "id": "a1234",
    "name": "fukui"
}
```

2-1. シリアライズ(1/2)

- **実装例**

① シリアライズするためのJavaクラス、インスタンスを作成
② Jsonbインスタンスを作成し、JsonBuilderのcreate()を使用
③ toJson(obj)を使用(obj はシリアライズ対象のオブジェクト)

クラス

```
public class Employee {

    private String id;
    private String name;
    private String department;

    public Employee() {
    }

    public Employee(String id, String name, String department) {
        this.id = id;
        this.name = name;
        this.department = department;
    }
...
(セッター、ゲッターは省略(privvateフィールドでは必須))
```

① ← private String id; / private String name; / private String department;

メソッド定義

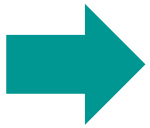
```
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
...

public class SampleMethod {
                シリアライズを行うメソッド定義
public String serializeEmployee(Employee emp) {

        Jsonb jsonb = JsonbBuilder.create();   → ②
        String json = jsonb.toJson(emp);       → ③
        return json;

    }                          インスタンス格納
}
```

2-1. シリアライズ(2/2)

- 出力

```
protected void doGet(HttpServletRequest request, HttpServletResponse
 response) throws ServletException, IOException {

    SampleMethod sample = new SampleMethod();
    ⟶ シリアライズメソッドを持つクラスをインスタンス化

   Employee emp = new Employee("X12345", "fukui", "cloud");
    ⟶ インスタンス作成
    SampleMethod sample = new SampleMethod();
    String jsonEmployee = sample.serializeEmployee(emp);
    ⟶ 前ページのメソッドの利用

 String view = "WEB-INF/view/result.jsp";
     RequestDispatcher dispatcher1=request.getRequestDispatcher(view);
              dispatcher1.forward(request, response);
          ⟶ 画面に文字列を描画
}
```

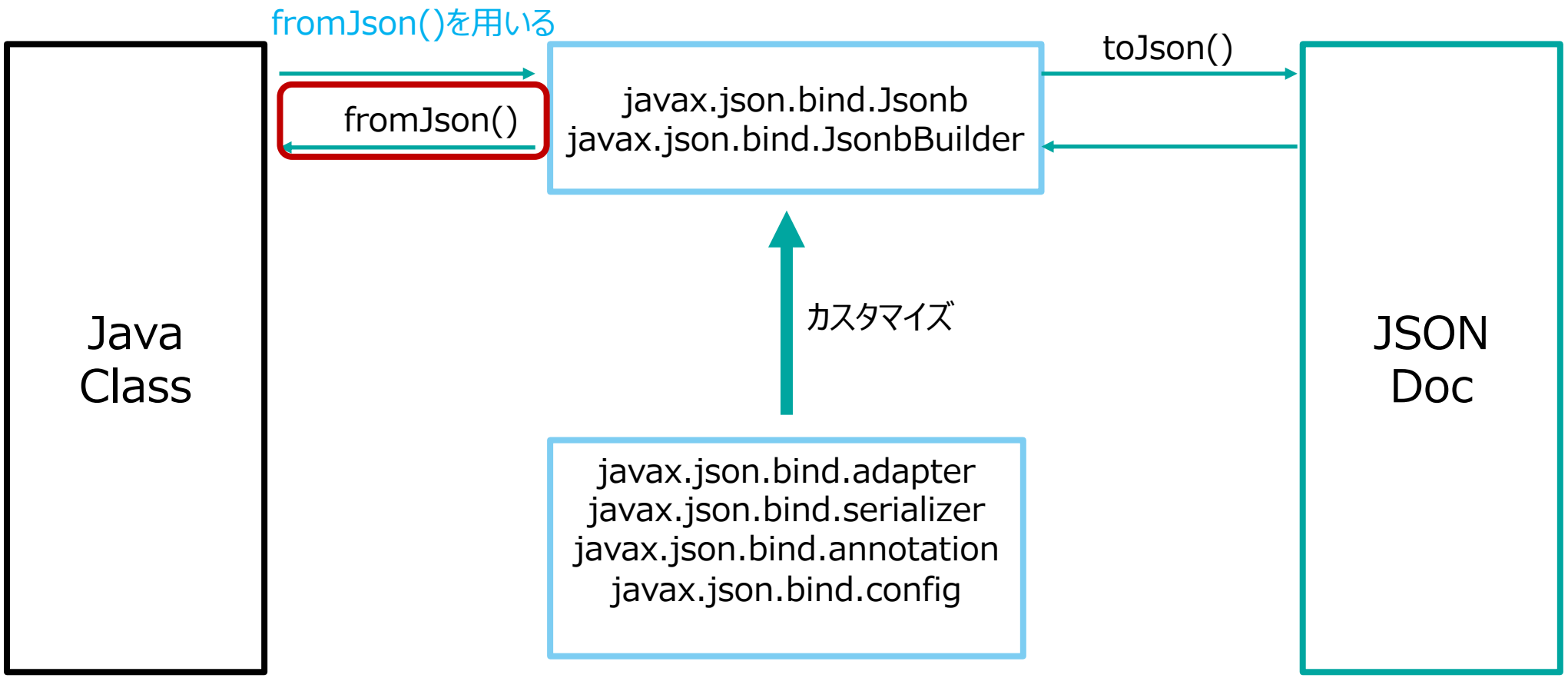
出力例



{"department":"cloud","id":"X12345","name":"fukui"}

2-2. デシリアライズ

- シリアライズ同様、Jsonbインターフェース、JsonbBuilderインターフェースを用いる



2-1.デシリアライズ

- デシリアライズの流れ
  - シリアライズと流れは同様

①, シリアライズするJSONを用意

```
{
    "department": "cloud",
    "id": "a1234",
    "name": "fukui"
}
```

②, JsonbBuilder のstatic メソッド, create()を利用したJsonb型のインスタンスを作成する。
->デシリアライズと同様

```
Jsonb jsonb = JsonbBuilder.create()
```

③, ②で作成したJsonbインスタンスが持つfromJson()メソッドの第一引数にJSONドキュメント、第二引数に該当クラスを指定

```
Jsonb.fromJson( JSON ドキュメント , Employee.class )
```

2-2. デシリアライズ(1/2)

- **シリアライズと流れは同じ**

① Javaクラス、インスタンスを作成、デシリアライズするJson文字列を取得
② Jsonbインスタンスを作成し、JsonBuilderのcreate()を使用
③ fromJson(json, Obj.class)を使用(json は文字列, Objは帰属されるクラス)

クラス

```
public class Employee {

    private String id;
    private String name;
    private String department;

    public Emp() {}

...(コンストラクタ、セッター、ゲッター省略)

 public String toString() {
        return "Employee {" +
                "id='" + id + '¥'' +
                ", name='" + name + '¥'' +
                ", department='" + department +'¥'' +
                '}';
    }
```

① ブロック

デフォルトコンストラクタ必須!

クラスを文字列として出力する細工

メソッド定義

```
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
...

public class Sample {

public Employee deserializeEmployee(String jsonEmployee) {
        Jsonb jsonb = JsonbBuilder.create();
        Employee empClass =
                jsonb.fromJson(jsonEmployee, Employee.class);
        return empClass;
    }
}
```

② JsonbBuilder.create()

③ fromJson(jsonEmployee, Employee.class)

JSON文字列

帰属されるクラス

2-2. デシリアライズ(2/2)

- **変換(出力)**

```
protected void doGet(HttpServletRequest request, HttpServletResponse
 response) throws ServletException, IOException {

    SampleMethod sample = new SampleMethod();
        ⟶ デシリアライズメソッドを持つクラスをインスタンス化
String json
 = "{¥"id¥":¥"a12345¥",¥"name¥":¥"fukui¥",¥"department¥":¥"cloud¥"}";
        ⟶ JSON文字列
        ⟶ 帰属されるクラスのフィールド変数名と一致させないとで該当フィールドはnull値となる
Employee classEmployee = sample.deserializeEmployee(json);
        ⟶ 前ページのメソッドの利用

String view = "WEB-INF/view/result.jsp";
     RequestDispatcher dispatcher1=request.getRequestDispatcher(view);
              dispatcher1.forward(request, response);
        ⟶ 画面に文字列を描画
}
```

出力例

Employee {id='a12345', name='fukui', department='cloud'}

2-2(参考). JAX-RS 2.1から使用する場合

- MediaType.APPLICATION_JSONで@Produceしたメソッドで，オブジェクトを返すとシリアライズされる

```
public class Employee {

    private String id;
    private String name;
    private String department;

    public Emp() {}
...(コンストラクタ、セッター、ゲッター省略)
}
```

```
import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("/api")
public class Rest extends Application{

}
```

ApplicationPassの設定

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

import overview.domain.Employee;
@Path("/test")
public class EndPoint {
        @GET
        @Produces(MediaType.APPLICATION_JSON)
        public Employee getEmployee() {
                return new Employee("a12345", "fukui","cloud");
        }

}
```

Pathの設定

http://localhost:9080/<プロジェクト名>/api/test

```
{"department":"cloud","id":"a12345","name":"fukui"}
```

2-3. JSON-Bのデフォルト設定

- 前ページまでで基本的なシリアライズ、デシリアライズの方法を解説
  - シリアライズ
    1. シリアライズするためのJavaクラス、インスタンスを作成
    2. Jsonbインスタンスを作成し、JsonBuilderのcreate()を使用
    3. toJson(obj)を使用(obj はシリアライズ対象のオブジェクト)

  - デシリアライズ
    1. Javaクラス、インスタンスを作成、デシリアライズするJson文字列を取得
    2. Jsonbインスタンスを作成し、JsonBuilderのcreate()を使用
    3. fromJson(json, Obj.class)を使用(json は文字列, Objは帰属されるクラス)

- 以降ではJsonBによりシリアライズ、デシリアライズされる際に**デフォルトでどのようにマッピングされるかを解説**
  - 出力形式、順番等

2-3. JSON-Bのデフォルト設定

- 基本型のマッピング
  - 基本データ型についてはデフォルトで下記のようにマッピング
    - String, char, はJSONで文字列に解釈される
    - Float, double, int, AtomicInteger, Byte型は数値として解釈される
    - boolean型は真偽値で解釈される

```
public class BasicTypes {

        private String name;

        private char initial;

        private Float height;

        private double weight;

        private int age;

        private boolean isPassed;

        private AtomicInteger count;

        private Byte version;
(セッター・ゲッター省略)
}
```

クラス

```
BasicTypes basic = new BasicTypes();

basic.setName("Fukui");

basic.setInitial('F');

basic.setHeight(169.99f);

basic.setWeight(60.000);

basic.setAge(20);

basic.setIsPassed(true);

basic.setCount(new AtomicInteger(4));

basic.setVersion((byte) 5);
```

値を格納

```
{
        "name":"Fukui",
        "initial":"F",
        "height":169.99,
        "weight":60.0,
        "age":20,
        "isPassed":true
        "count":4,
        "version":5,
}
```

出力(実際には辞書式で出力)

2-3. JSON-Bのデフォルト設定

- 配列・コレクション型のシリアライズ
  - List型, 固定長配列ともに、JSONドキュメントとしては配列として解釈される
  - Map<Key, Value>はネストされたオブジェクトとして{"key": "value"}が生成される
  - MapやList型のインスタンスにnull値が含まれる場合は、配列にnullと表示される
    - 注) カスタムクラスのフィールドのnull値はJSONドキュメントにするとそのフィールド自体が出力されない(P.26)

```
public class ArraysAndCollections {

    private int[] ints = new int[5];

    private int[][] ints2d = new int[5][];

    private String[] strs = new String[3];

    private List<String> list;

    private Map<String, Integer> map;

    private Set<String> set;



(セッター・ゲッター省略)
}
```

```
ArraysAndCollections ac = new ArraysAndCollections();

ac.setInts(new int[]{1, 2, 3, 4, 5});
ac.setInts2d(new int[][]{{1, 2}, {3, 4}, {5, 6}, {7, 8}});
ac.setStrs(new String[]{"aaa", "bbb" , null});
ac.setList(new ArrayList<String>() {{
          add("Java EE");
          add("Python");
       }});
ac.setMap(new HashMap<String, Integer>() {{
          put("Fukui", 26);
          put("Problem", null);
       }});
ac.setSet(new HashSet<String>() {{
          add("Red");
       }});
```

```
{

      "ints": [1,2,3,4,5],

      "ints2d": [[1,2],[3,4],[5,6],[7,8]],

      "strs": ["aaa","bbb",null]

      "list": ["Java EE","Python"],

      "map": {
               "Fukui": 26,
               "Problem": null
      },

       "set": ["Red"],

}
```

2-3. JSON-Bのデフォルト設定

- その他のデータ型のマッピング(1/2)
- BigInteger, Optional, Date 型
  - BigInteger, Optional<Integer>, OptionalDouble, OptionalLongそれぞれ数値として解釈される
  - Date型は文字列として解釈される

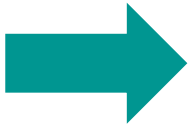
```
public class JavaTypes {

        public BigInteger bigInteger = new BigInteger("10");

        public Optional<Integer> valueOptInt = Optional.of(10);

        public OptionalDouble valueOptDbl = OptionalDouble.of(3.14);

        public OptionalLong valueOptLong = OptionalLong.of(314666666);

        public Date date = Date.valueOf("1992-12-31");
}
```

```
{
        "bigInteger":10,
        "valueOptInt":10,
        "valueOptDbl":3.14,
        "valueOptLong":314666666
        "date":"1992-12-30Z",
}
```

2-3. JSON-Bのデフォルト設定

- その他のデータ型のマッピング(2/2)
- 作成したクラスに別の作成したクラスが含まれる場合
  - ネストして新たなオブジェクトとして生成される

```
public class EmpWithOtherClass {
                private String id;
                private String name;
                private String department;
                private Skills skills;
…
}

public class Skills {
                private boolean java;
                private boolean python;
                private boolean javascript;
…
}
```



```
{
                "department":"cloud",
                "id":"a1234",
                "name":"fukui",
                "skills":{
                                "java":true,
                                "javascript":true,
                                "python":false
                                }
}
```

"field名": 新たなJSONオブジェクト

2-3. JSON-Bのデフォルト設定

- デシリアライズ(1/2)
  - Jsonオブジェクトをカスタムクラスに帰属させない場合は基本的にMap型で帰属可能

JSONの値	Javaデータ型
オブジェクト	java.util.Map<String, Object>
配列	java.util.List<Object>
ストリング	java.lang.String
数値	java.math.BigDecimal
true, false	java.lang.Boolean
null	null

ex)
以下のJson文字列を扱う場合

```
String json = "{\"name\":\"Fukui\",\"height\":175.78,\"health problem\":null}";
```

```
public Map<String, Object> deserializeMap(String json){
    Map map = JsonbBuilder.create().fromJson(json, Map.class);
    return map;
    }
```

デシリアライズ

```
[Fromjson(Default)]
{health problem=null, name=Fukui, height=175.78}
```

2-3. JSON-Bのデフォルト設定

- デシリアライズ(2/2)
  - Jsonオブジェクトをカスタムクラスに帰属させる場合
    - JSONフィールドのキー名とJavaクラスのフィールド変数名に一致するもののみがインスタンス化される（大文字小文字も区別される）
    - JSONのキー名とJavaクラスのキー名が異ならないように注意

ex)

デシリアライズするJSON

```
{
    "department":"cloud",
    "fake": "fake"
    Javaクラスのフィールドに存在しないデータ
    "id":"a1234",
    "Name":"fukui"
    キーが大文字
}
```

帰属するクラス

```
public class Employee {
    private String id;
    private String name;
    フィールド変数が小文字
    private String department;
..
.(セッター、ゲッター、コンストラクタ)
}
```

格納される値

```
{
    department=cloud ,
    id=a1234,
    name=null
    Nameとnameは区別
    されるため、マッチするも
    のがないとみなされnull
    値が入る
}
```

例示した“fake”はフィールドに存在しないため無視される

2-3. JSON-Bのデフォルト設定

- ストラテジー ・・・JSONドキュメントにマッピングされるときの法則
  - 命名ストラテジー
    - JSONのkeyはフィールド名が引き継がれる

```
public class Test {
   public String dog = "chihuahua";
   public int angle = 45;
   public String bread = "naan";
   public String car = "Ford";
}
```

```
{
   "angle":45,
   "bread":"naan"
   "car":"Ford",
   "dog":"chihuahua"
}
```

  - 可視性ストラテジー
    - public フィールド、public メソッドのみにアクセス。
      - カプセル化を行なった場合シリアライズの際にpublicなgetterメソッドが必要
    - null値を持つフィールドは無視されて出力されるが、配列・コレクション型に対してはnull値が含まれる場合、null値として設定される

```
public class Test {
   String dog = "chihuahua";
   int angle = 45;
   String bread = "naan";
   String car = "Ford";
   String error = "null";
}
```

```
{
   "angle":45,
   "bread":"naan",
   "car":"Ford",
   "dog":"chihuahua"
}
```

nullフィールドは無視される

※ P.23で用いたクラス

```
ArraysAndCollections ac = new ArraysAndCollections();
ac.setStrs(new String[]{"aaa", "bbb" , null });
```

```
{
   "strs": ["aaa","bbb", null ]
}
```

2-3. JSON-Bのデフォルト設定

- ストラテジー(2/2)
  - 順序
    - 辞書式

ex)

```
public class Test {
   public String dog = "chihuahua";
   public int angle = 45;
   public String bread = "naan";
   public String car = "Ford";
}
```

フィールド名のアルファベット順

```
{
   "angle":45,
   "bread":"chihuahua",
   "car":"Ford",
   "dog":"naan"
}
```

  - バイナリーデータストラテジー
    - バイト配列を使用してバイナリーデータを符号化

3. JSON-Bカスタマイズ

3-1. カスタマイズで可能なこと

- 以下の様なカスタマイズが可能
  - データ構造のカスタマイズ
    - ネストしていたものを並列にする
    - 新たなフィールドを追加する
  - フィールドの可視性をカスタマイズ
    - null値の扱いを変更する
    - 特定のフィールドを非表示に
  - 出力順序のカスタマイズ
    - 辞書式とは逆の順序で出力
    - 指定した順序で出力
  - 命名ストラテジーのカスタマイズ
    - 先頭の文字を大文字にするなど
  - 日付の出力形式や、数値のフォーマットのカスタマイズ
    - 2019-05-27 -> 2019/05/27
    - 45.000001 -> 45.00

カスタマイズ例

デフォルトでの出力例

```
{
   "department":"cloud",
   "id":"a1234",
   "name":"fukui",
   "skills":{
      "java":true,
      "javascript":true,
      "python":false
      }
}
```

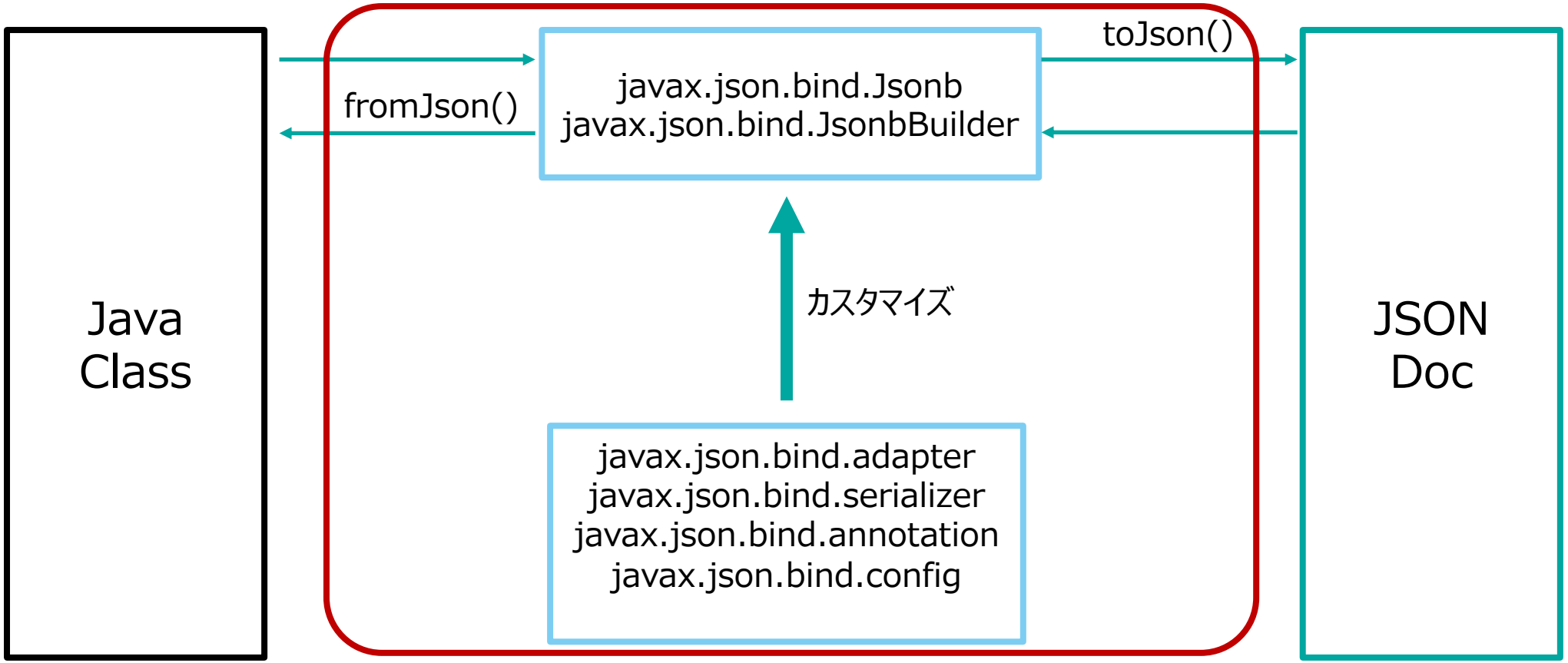
・指定の順序での出力
・性と名を分割
・キー名を変更
・スキルフィールドを削除し、
　Java, Python, JavaScriptキーを新規生成

カスタマイズした出力例

```
{
      "ID":"a1234",
      "氏":"fukui",
      "名":"yoshiharu",
      "部門":"cloud",
      "Java":true,
      "Python":true,
      "JavaScript":false
}
```

3-2. カスタマイズ方法

- 基本的なシリアライズ、デシリアライズ同様Jsonb,JsonbBuilderインターフェースを用いる
- adapter, serializer, annotation, configパッケージを利用してカスタマイズを行う



3-2. カスタマイズ方法

- 基本的には以下のような方法でカスタマイズを行う
  - ① アノテーションを利用した、**出力形式**のカスタマイズ
    - アノテーションを付与するだけで簡単にカスタマイズ可能
    - アノテーションによりフォーマットや特定のフィールド名の変更、順序変更などを行う

利用インターフェース：javax.json.bind.annotation

  - ② JsonbConfigインスタンスを用いた、**出力形式**のカスタマイズ
    - JsonbConfig インスタンスに用意されているデフォルトの命名ストラテジーや、順序ストラテジー、フィールドの可視性ストラテジーのメソッドを利用してを適用して、出力形式を変更する
    - フォーマットや順序、フィールドの可視性の変更を行える

利用インターフェース：javax.json.bind.config

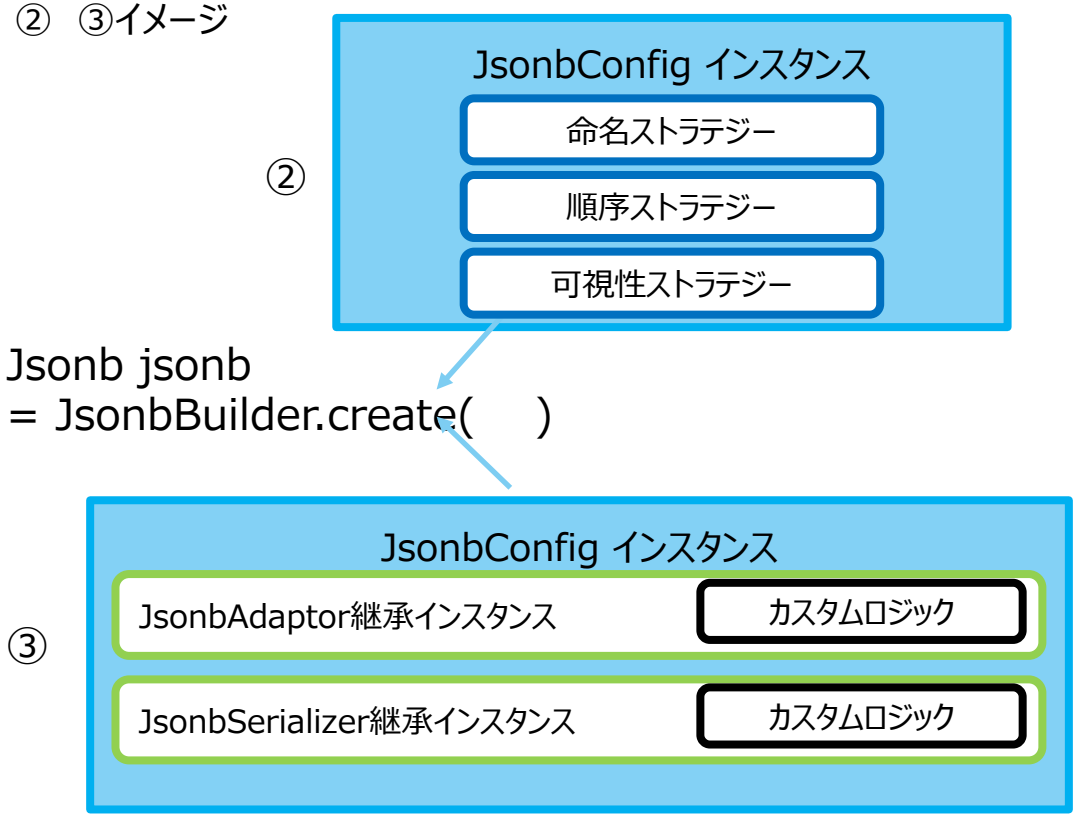
  - ② JsonbConfigインスタンスを用いた**データ構造**のカスタマイズ
    - JsonbAdapterやJsonSerializer, JsonDeserializer Interface を継承したクラスに出力されるJsonのデータ構造そのものをカスタマイズするカスタマイズロジックを記述する
    - 上記のクラスをJsonbConfigに登録することで、カスタマイズロジックに応じたカスタマイズを行う

利用インターフェース：
JsonbAdapter, JsonbSerializer, JsonbSerializer

①例

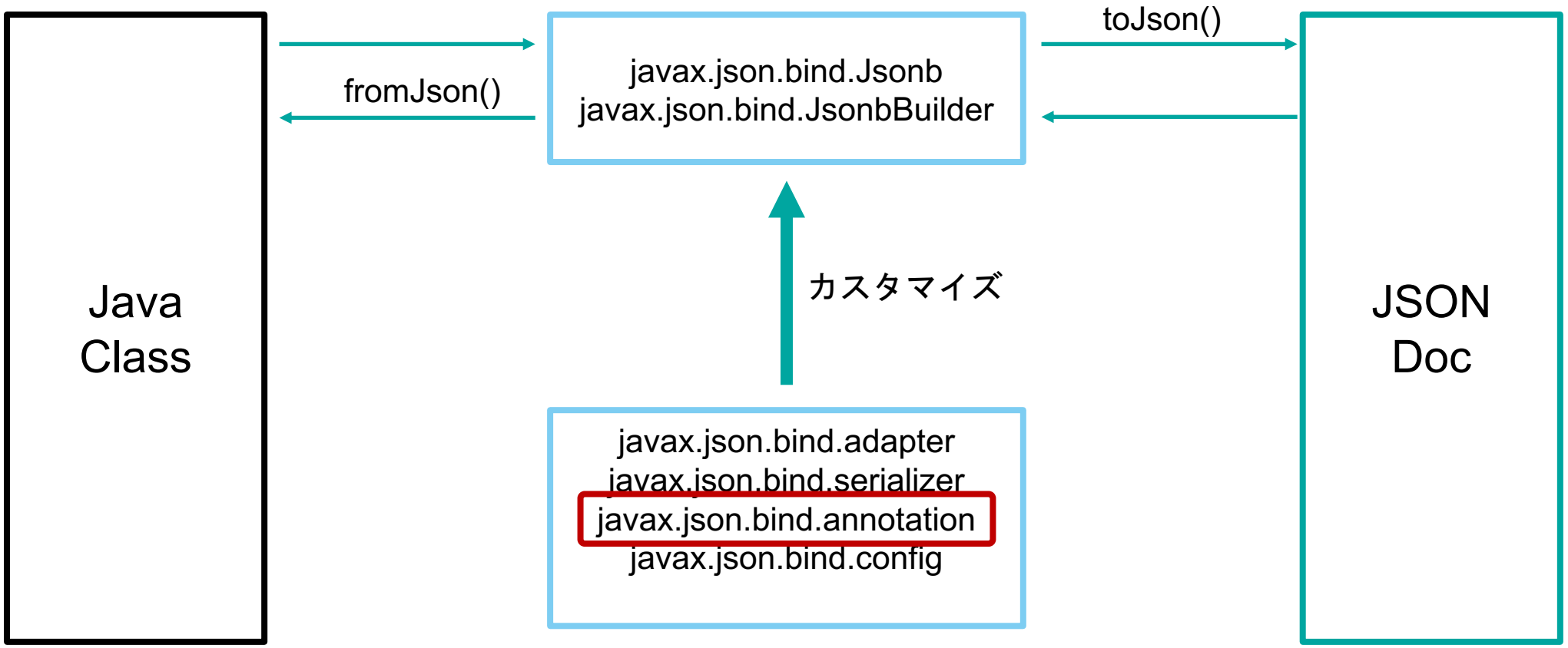
```
public class Test {
@JsonbTransient
   public String id = "a12345";

}
```

② ③イメージ



3-3. アノテーションによるカスタマイズ

- クラス上部や、フィールド上部、メソッド上部等にアノテーションを付与することで、手軽にカスタマイズを行う



3-3. アノテーションによるカスタマイズ

- 次のようにJsonbでカスタマイズする際にアノテーションを付与する場所は3パターン存在する
  - public class 上部
  - フィールド変数上部
  - メソッド上部

- アノテーションの対象範囲については次ページ参照

```
@JsonbPropertyOrder(value = {"id", "angle", "bread" ,"car", "dog"})
public class Test {

   private String id = "a12345";
@JsonbTransient
   private String dog = "Chihuahua";

…
 @JsonbDateFormat(value = "yyyy年MM月dd日", locale = "Locale.JAPANESE")
public LocalDate getWhen() {
                return when;
   }
}
```

→ public class 上部

→ フィールド変数上部や変数直前

→ メソッド上部

3-3. アノテーションによるカスタマイズ

- アノテーション一覧

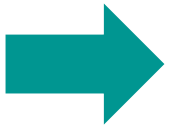
Annotation Type	Description	適用範囲
JsonbAnnotation	用意されているアノテーションが利用できるようになる	-
JsonbCreator	コンストラクターに出力させる変数を指定し、指定したもののみをマッピングする	@Target({ANNOTATION_TYPE,METHOD,CONSTRUCTOR})
JsonbDateFormat	日付のフォーマットを変更する	@Target({ANNOTATION_TYPE,FIELD,METHOD,TYPE,PARAMETER,PACKAGE})
JsonbNillable	フィールドにnull値が含まれるか判断する	@Target({ANNOTATION_TYPE,TYPE,PACKAGE})
JsonbNumberFormat	数値のフォーマットを変更する	@Target({ANNOTATION_TYPE,FIELD,METHOD,TYPE,PARAMETER,PACKAGE})
JsonbProperty	フィールドの変数名を変更する	@Target({ANNOTATION_TYPE,METHOD,FIELD,PARAMETER})
JsonbPropertyOrder	出力順序を変更する	@Target({ANNOTATION_TYPE,TYPE})
JsonbTransient	JSONにマッピングされないようにする	@Target({ANNOTATION_TYPE,FIELD,METHOD})
JsonbTypeAdapter	フィールドレベルでJsonbAdapterを利用する	@Target({ANNOTATION_TYPE,TYPE,FIELD,METHOD})
JsonbTypeDeserializer	フィールドレベルでDeserializerを利用する	@Target({ANNOTATION_TYPE,TYPE,FIELD,METHOD})
JsonbTypeSerializer	フィールドレベルでSerializerを利用する	@Target({ANNOTATION_TYPE,TYPE,FIELD,METHOD})
JsonbVisibility	可視性をカスタマイズする	@Target({ANNOTATION_TYPE,TYPE,PACKAGE})

3-3. アノテーションによるカスタマイズ(フィールドの可視性のカスタマイズ)

- 使用例1
  - @JsonbTransient
    - 特定のフィールドをJSONとして出力させない

出力させたくないフィールドの上部に配置

```
public class Test {
@JsonbTransient
   public String id = "a12345";
   public String dog = "Chihuahua";
   public int angle = 45;
   public String bread = "naan";
   public String car = "Ford";

}
```



```
{

      "angle":45,
      "bread":"naan",
      "car":"Ford",
      "dog":"Chihuahua"

      idが出力されない

}
```

3-3. アノテーションによるカスタマイズ(フィールドの可視性のカスタマイズ)

- 使用例2
  - @JsonbProperty
    - フィールド名をカスタマイズ
    - null値の扱いもカスタマイズ可能

```
public class Test {
@JsonbProperty("ID")
   public String id = "a1234";
   public String dog = "Chihuahua";
   public int angle = 45;
}
```

キー名を変更したいフィールドの上部に配置

"id" が "ID"として出力される

```
{
   "angle":45,
   "bread":"naan",
   "ID":"a1234"

}
```

(デフォルト)

```
public class Test {
   public String id = null;
   public String dog = "Chihuahua";
   public int angle = 45;
}
```

デフォルトではnullフィールドは出力されない

```
{
   "angle":45,
   "bread":"naan",

}
```

(@JsonbProperty(nillable = true))

```
public class Test {
  @JsonbProperty(nillable = true)
   public String id = null;
   public String dog = "Chihuahua";
   public int angle = 45;
}
```

null値を出力させたいフィールドの上部に配置

デフォルトではnullフィールドが出力される

```
{
   "angle":45,
   "bread":"naan",
   "id":null

}
```

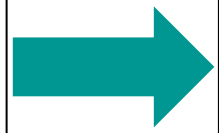
3-3. アノテーションによるカスタマイズ(順序のカスタマイズ)

- 使用例 3
  - @JsonbPropertyOrder
    - 出力順序を指定できる

クラス上部に配置

```
@JsonbPropertyOrder(value = {"id", "angle", "bread" ,"car", "dog"})
public class Test {
   public String id = "a12345";
   public String dog = "Chihuahua";
   public int angle = 45;
   public String bread = "naan";
   public String car = "Ford";

}
```

JSONが出力される順序



```
{
   "id":"a12345",
   "angle":45,
   "bread":"naan",
   "car":"Ford",
   "dog":"Chihuahua"
}
```

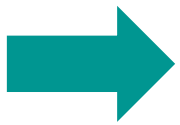
辞書式順序ではなく、アノテーションで指定した順序に

3-3. アノテーションによるカスタマイズ(日付、時刻フォーマットのカスタマイズ)

- 使用例4
  - @JsonbDateFormat
    - Date型を返すゲッターの上部に配置し、日付、時刻フォーマットを指定できる

```
LocalDate date1 = LocalDate.now();        フィールドにdate1(現在の日時)を追加
Test test = new Test("a12345", "Chihuahua", 45, "naan","Ford", date1 );
```

```
public class Test {
   private String id;
   private String dog;
...
   private LocalDate when;
 ...
 @JsonbDateFormat(value = "yyyy年MM月dd日", locale = "Locale.JAPANESE")
public LocalDate getWhen() {  ⟶  ゲッターでフォーマットを指定
                return when;
   }
```



カスタマイズされた出力例
{"id":"a12345","angle":45,"bread":"naan","car":"Ford","dog":"Chihuahua","when":"2019年05月23日"}

カスタマイズなしの場合の出力例
{"id":"a12345","angle":45,"bread":"naan","car":"Ford","dog":"Chihuahua","when":"2019-05-23"}

3-3. アノテーションによるカスタマイズ

- 数値フォーマットのカスタマイズ
  - @JsonbNumberFormat

```
public class Test {
   private String id;
   private String dog;
   @JsonbNumberFormat("#.00")
   private double angle;
   private String bread;
   private String car;
   private LocalDate when;
 ...
```

フォーマットを変更したいフィールドの上部に配置

フォーマットの指定：小数点2桁まで切り捨て

(インスタンス)

```
Test test = new Test("a12345", "Chihuahua", 45.00001, "naan","Ford", date1);
```

カスタマイズした場合の出力例

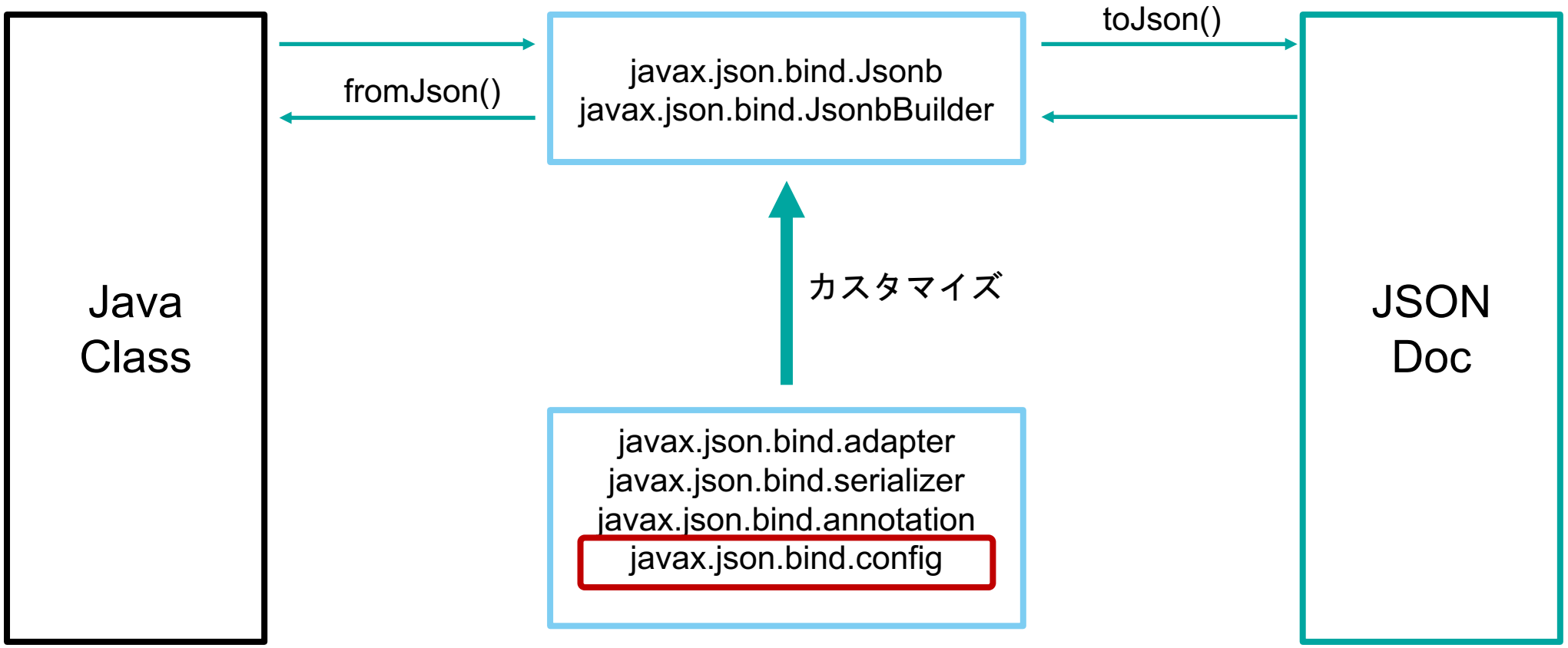
{"id":"a12345","angle":"45.00","bread":"naan","car":"Ford","dog":"Chihuahua","when":"05_23_2019"}

カスタマイズ無しの場合の出力例

{"id":"a12345","angle":45.00001,"bread":"naan","car":"Ford","dog":"Chihuahua","when":"05_23_2019"}

3-4. JsonbConfigインスタンスの利用

- JsonbConfigインスタンスを用いて、出力順序(順序ストラテジー), 出力名(命名ストラテジー),可視性ストラテジーを設定し、カスタマイズを行う

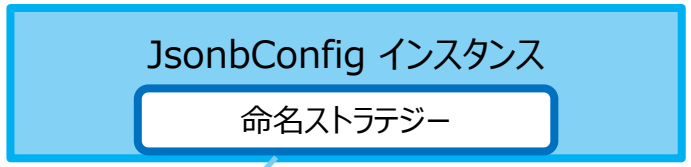


3-4. JsonbConfigインスタンスの利用

- JsonbConfigの基本的な使い方

① JsonbConfigのインスタンスを作成し、ストラテジーを決定

② Jsonbインスタンスのcreateメソッドの引数に①を代入



Jsonb jsonb = JsonbBuilder.create( )

```
public String serializeTest(Test test) {

JsonbConfig jsonbConfig = new JsonbConfig()
            .withPropertyNamingStrategy(
                PropertyNamingStrategy.UPPER_CAMEL_CASE);

Jsonb jsonb = JsonbBuilder.create(jsonbConfig);
String json = jsonb.toJson(test);
return json;

}
```

①JsonbConfigインスタンスの作成
①命名ストラテジーエオ変更
②フィールド名の先頭文字が大文字になる
ストラテジーを設定
ストラテジーを設定したJsonbConfigインスタンスを代入

(インスタンス)

```
Test test = new Test("Chihuahua", 45, "naan","Ford");
```

{"Angle":45,"Bread":"naan","Car":"Ford","Dog":"Chihuahua"}

フィールド名の先頭が大文字に

3-4. JsonbConfigインスタンスの利用

- 命名ストラテジーに加えて、順序ストラテジーも変更可能

```
public String serializeTest(Test test) {

JsonbConfig jsonbConfig = new JsonbConfig()
                                .withPropertyNamingStrategy(
                                        PropertyNamingStrategy.UPPER_CAMEL_CASE);

        jsonbConfig.withPropertyOrderStrategy(PropertyOrderStrategy.REVERSE);
                                                    → 命名ストラテジーに加えて順序ストラテジーも追加
        Jsonb jsonb = JsonbBuilder.create(jsonbConfig);
                                                → JsonbConfigを利用
    String json = jsonb.toJson(test);
    return json;
}
```

```
{"Dog":"Chihuahua","Car":"Ford","Bread":"naan","Angle":45}
```

3-4. JsonbConfigインスタンスの利用

- 6つの命名ストラテジーと、3つの順序ストラテジー
  - 命名ストラテジー

Modifier and Type	Field	Description	例 (マッピングされるJSONのキー)
static String	CASE_INSENSITIVE	デフォルトの通りにマッピングする	authorName -> authorName
static String	IDENTITY	プロパティ名が変わらないようにする	
static String	LOWER_CASE_WITH_DASHES	単語ごとの先頭が小文字に変換され、単語間にダッシュが挿入される (最初の文字以降に大文字がある時に別単語として認識される)	authorName->author-name authorname->authorname
static String	LOWER_CASE_WITH_UNDERSCORES	単語ごとの先頭が小文字に変換され、単語間でアンダーバーが挿入される	authorName->author_name authorname->authorname
static String	UPPER_CAMEL_CASE	単語ごとの先頭が大文字に変換され、先頭文字が大文字になる	authorName ->AuthorName authorname->authorname
static String	UPPER_CAMEL_CASE_WITH_SPACES	単語ごとの先頭が大文字に変換され、単語間に　スペースが入る	authorName ->Author Name authorname ->authorname

  - 順序ストラテジー

Modifier and Type	Field	Description
static String	ANY	順序は保障されない
static String	LEXICOGRAPHICAL	辞書式のままの順序で出力される
static String	REVERSE	辞書式とは逆の順序で出力される

3-4. JsonbConfigインスタンスの利用

- 日付のカスタマイズ
  - アノテーションで日付フォーマットを変更したのと同様の操作をJsonbConfigインスタンスを用いても行える

```
LocalDate date1 = LocalDate.now();
Test test = new Test("a12345", "Chihuahua", 45, "naan","Ford", date1);
```

```
public String serialize(Test test) {
        JsonbConfig cfg = new JsonbConfig().withDateFormat(" yyyy年MM月dd日", Locale.JAPANESE);
                                                    → withDateFormat()を用いる
        Jsonb jsonb = JsonbBuilder.create(cfg);
        String json = jsonb.toJson(test);
          return json;
}
```

→ {"id":"a12345","angle":45,"bread":"naan","car":"Ford","dog":"Chihuahua","when":" 2019年05月23日"}

3-4. JsonbConfigインスタンスの利用

- フィールドの可視性をカスタマイズ
  - PropertyVisibilityStrategy インスタンスを利用することで設定可能

```
blic class Test {

    private String id;
    private String dog;
    private int angle;
    private String bread;
    private String car;

...
     public String getBread () {
              return id;
     }
...

 }
```

```
public String serialize(Test test) {
  PropertyVisibilityStrategy vis = new PropertyVisibilityStrategy() {
       @Override
        public boolean isVisible(Field field) {
                    return Modifier.isProtected(field.getModifiers());
                }
       @Override
        public boolean isVisible(Method method) {
                   return !method.getName().contains('Bread');
                }
   };
 JsonbConfig jsonbConfig = new JsonbConfig().withPropertyVisibilityStrategy(vis)
            Jsonb jsonb = JsonbBuilder.create(jsonbConfig);
            String json = jsonb.toJson(test);
            return json;
}
```

PropertyVisibilityStrategyインスタンスを作成

IsVisibleの実装必須

ゲッターのメソッド名にBreadが含まれる場合は非表示

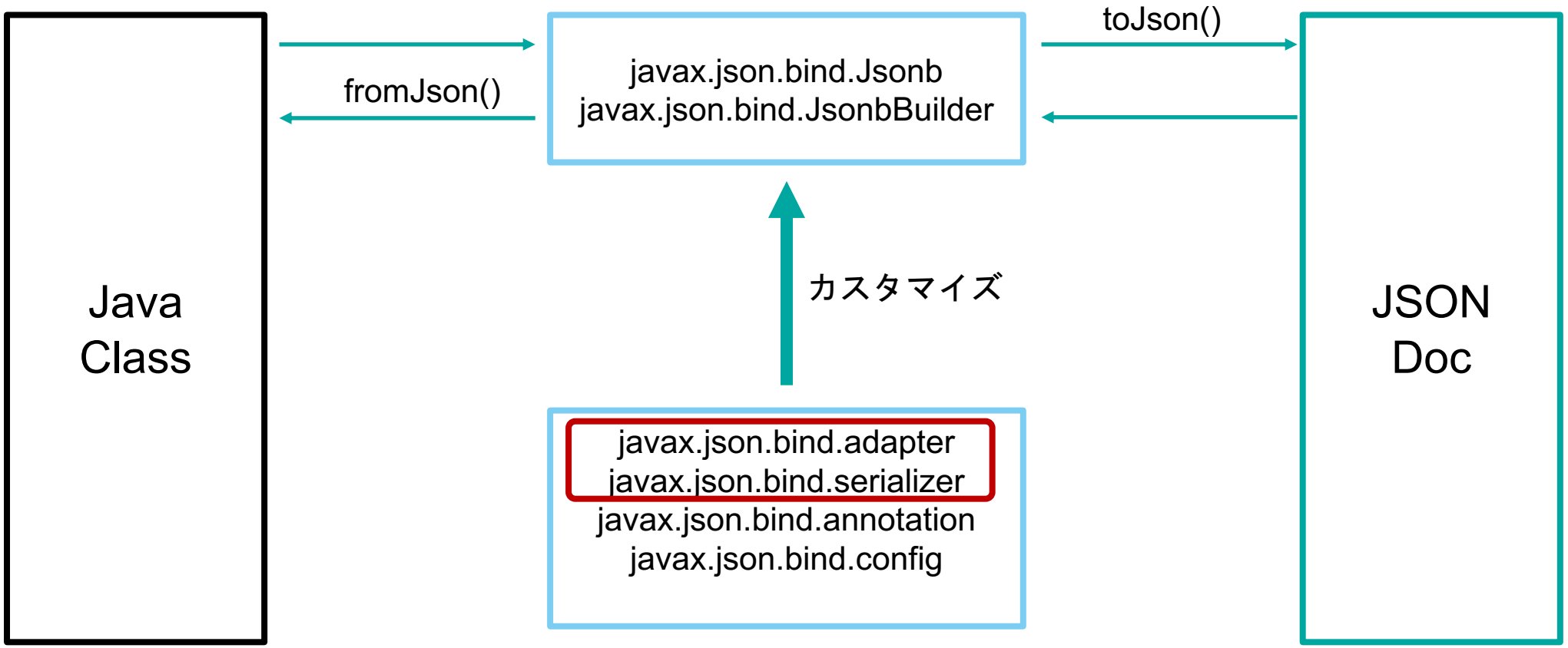
PropertyVisibilityStrategyインスタンスを引数に代入

```
Test test = new Test("a12345", "Chihuahua", 45, "naan","Ford", date1);
```

```
{"angle":45,"car":"Ford","dog":"Chihuahua","id":"a12345"}
```

3-5. Jsonbアダプター、JsonbSerializerの実装

- 今までのカスタマイズ方法では、範囲に限りがある
  - データ構造などをカスタマイズすることはできない
- adapter, serializerインターフェースを用いてより柔軟にカスタマイズを行う



3-5-1. Jsonbアダプターの実装

- JsonbAdapterや、JsonbSerializerを用いるメリット
  - JSONへのマッピングを好きなように構成できる
    - JSONにシリアライズする際に、新たなフィールドを追加する
    - 別クラスから得る情報を追加することも可能
    - シリアライズする際の表示形式等を変更することももちろん可能
  - 下記のようなニーズに対応できる
    - クライアントサイドの開発者から、サーバーから送信される既存のデータ項目に別のデータ項目を追加してほしいと要求される
      - 一からクラスの設計をやり直す必要がない
    - クライアントサイドの開発者からやはり氏名の"氏"と"名"で分割してほしいと要求される
      - クラスの設計や、DBの設計などを行う必要がない

クライアントサイド開発者とサーバーサイド開発者の連携がスムーズになる

3-5-1. Jsonbアダプターの実装

- カスタマイズ例
  - JsonbAdapterを用いることで下記のようにデータ構造や出力形式まで包括的にカスタマイズ可能

```
public class EmpWithOtherClass {
                private String id;
                private String name;
                private String department;
                private Skills skills;
}
…
public class Skills {
                private boolean java;
                private boolean python;
                private boolean javascript;
…
}
```

カスタマイズ無し

```
{
        "department":"cloud",
        "id":"a1234",
        "name":"fukui",
        "skills":{
                "java":true,
                "javascript":true,
                "python":false
                }
}
```

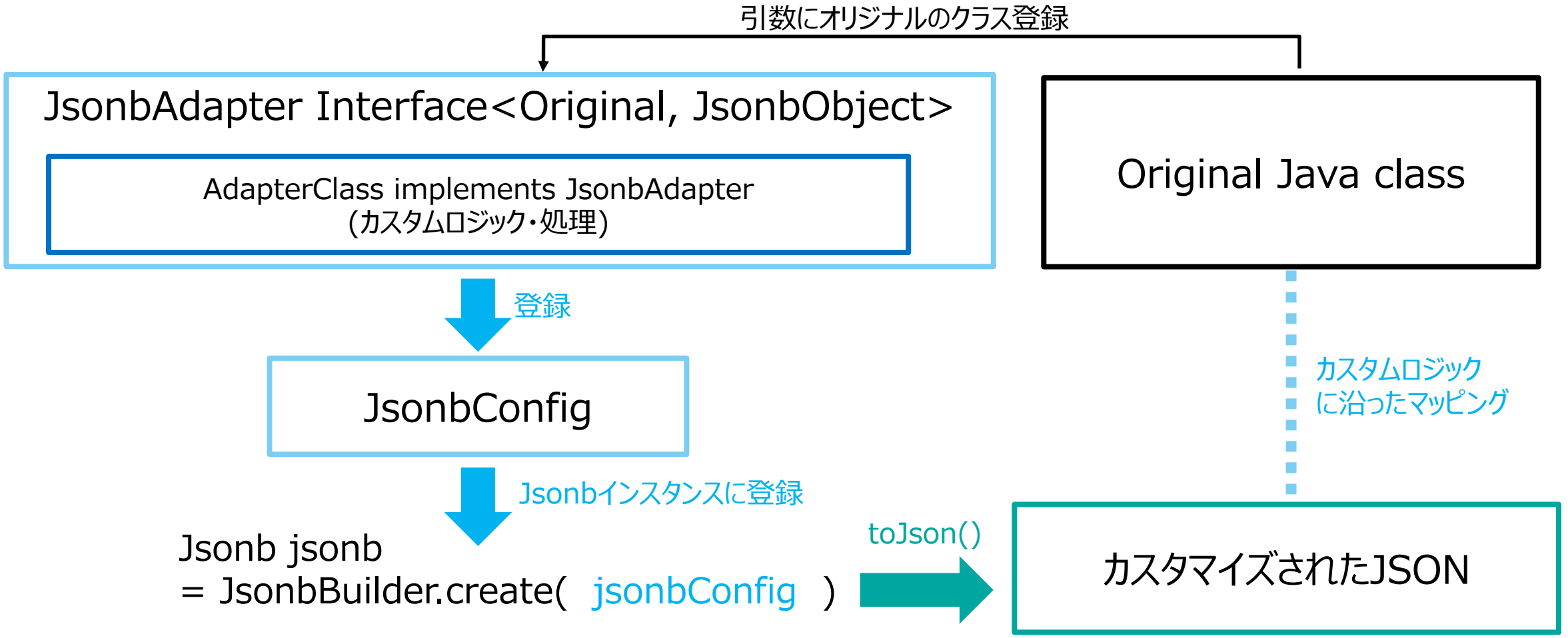
カスタマイズ(例)

・指定した順序での出力
・性と名を分割
・キー名を変更
・スキルフィールドを削除
・Java, Python, JavaScript キーを新規生成

```
{
        "ID":"a1234",
        "氏":"fukui",
        "名":"yoshiharu",
        "部門":"cloud",
        "Java":true,
        "Python":true,
        "JavaScript":false
}
```

3-5-1. Jsonbアダプターの実装

- JsonbAdapterを用いたカスタマイズイメージ
  - JsonbAdapter Interfaceにオリジナルクラスのカスタムロジックを記述
  - 上記をJsonbConfigに登録して、toJson(Original original)を行う



3-5-1 Jsonbアダプターの実装

- 次ページ以降で下記のようなカスタマイズの実装例を示す

```
public class EmpWithOtherClass {
                private String id;
                private String name;
                private String department;
                private Skills skills;

(ゲッター、セッターコンストラクタ省略)
}

// ↓別クラス

public class Skills {
                private boolean java;
                private boolean python;
                private boolean javascript;

(ゲッター、セッターコンストラクタ省略)
}
```

カスタマイズ無し

```
{
    "department":"cloud",
    "id":"a1234",
    "name":"fukui",
    "skills":{
            "java":true,
            "javascript":true,
            "python":false
            }
}
```

カスタマイズ有り

```
{
    "ID":"a1234",
    "氏":"fukui",
    "名":"yoshiharu",
    "部門":"cloud",
    "Java":true,
    "Python":true,
    "JavaScript":false
}
```

3-5-1. Jsonbアダプターの実装

- JsonbAdapter の実装例

```
public class EmpAdapter implements JsonbAdapter<EmpWithOtherClass, JsonObject> {

        @Override
        public JsonObject adaptToJson(EmpWithOtherClass emp) {

                return Json.createObjectBuilder()
                                .add("ID", emp.getId())
                                .add("氏", emp.getName().split(" ")[1])
                                .add("名", emp.getName().split(" ")[0])
                                .add("部門", emp.getDepartment())
                                .add("Java", skill.getJava())
                                .add("Python", skill.getPython())
                                .add("JavaScript", skill.getJavascript())
                                .build();

        }

        @Override
        public EmpWithOtherClass adaptFromJson(JsonObject json) throws Exception {

                // TODO Auto-generated method stub
                EmpWithOtherClass emp = new EmpWithOtherClass(json.getString("ID"),json.getString("氏")+" "+json.getString("名"),json.getString("部門"),
                                new Skills(json.getBoolean("Java"), json.getBoolean("Python"), json.getBoolean("JavaScript")));
                return emp;
        }

}
```

JsonbAdapterインターフェースを継承

カスタマイズするクラス

Json-Pで用意されている
JsonbObjectBuilderメソッドを利用

自在に編集

adaptToJsonとadaptFromJson
共に実装必要

引数にデシリアライズされるJSON

jsonのキーから値を取得して、インスタンス化

3-5-1. Jsonbアダプターの実装

- JsonbConfigへの登録

```
public void adapterSerialize() {
        JsonbConfig jsonbConfig = new JsonbConfig().withAdapters(new EmpAdapter());

        Skills skills = new Skills(true, false, true);
        EmpWithOtherClass emp = new EmpWithOtherClass("a1234", "yoshiharu fukui", "cloud", skills);

        String actualJson = JsonbBuilder.create(jsonbConfig).toJson(emp);
        System.out.print(actualJson);
}
public void adaptorDeserialize() {
        String json = "{¥"ID¥":¥"a1234¥",¥"氏¥":¥"fukui¥",¥"名¥":¥"yoshiharu¥",¥"部門¥":¥"cloud¥",¥"Java¥":true,¥"Python¥":true,¥"JavaScript¥":false}";
        JsonbConfig jsonbConfig = new JsonbConfig().withAdapters(new EmpAdapter());
        EmpWithOtherClass emp = JsonbBuilder.create(jsonbConfig).fromJson(json, EmpWithOtherClass.class);
        System.out.print(emp);
}
```

P.50で作成したEmpAdapterインスタンスの登録

JsonbConfig configを登録

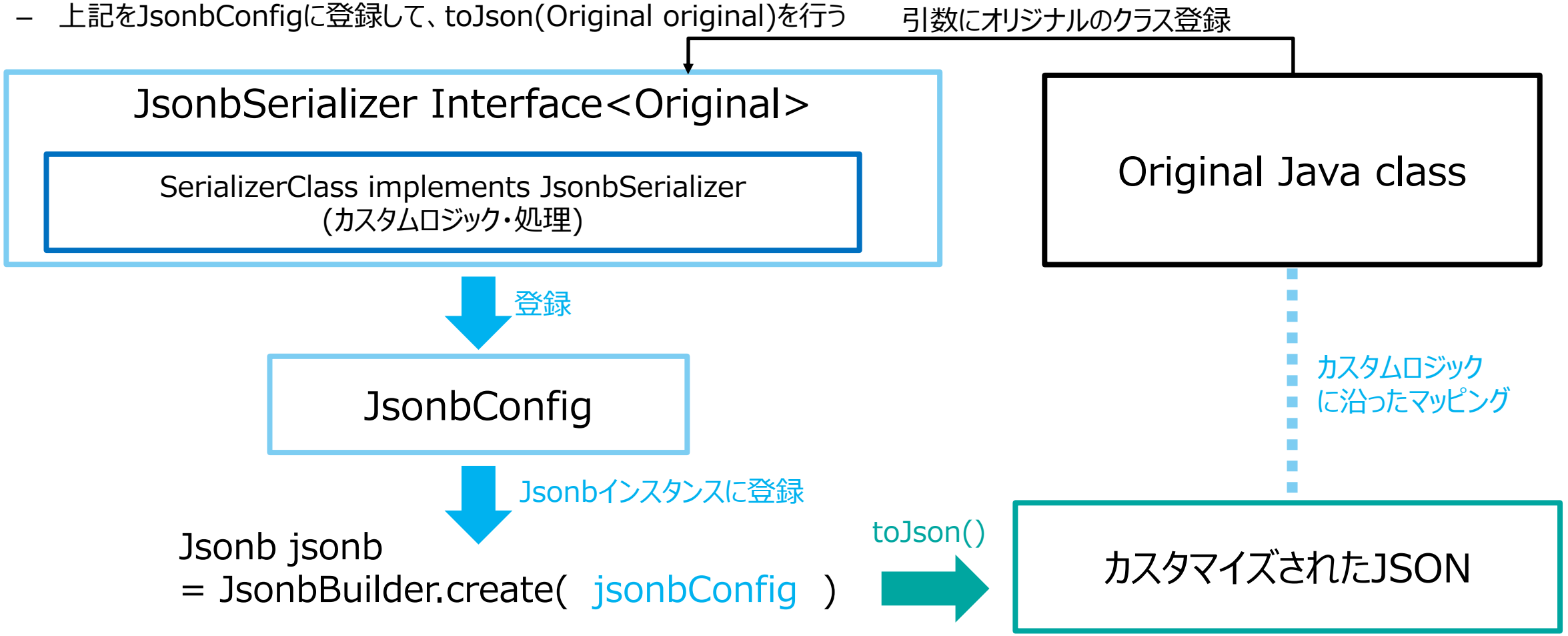
デシリアライズも同様

シリアライズ

```
{
    "ID":"a1234",
    "氏":"fukui",
    "名":"yoshiharu",
    "部門":"cloud",
    "Java":true,
    "Python":true,
    "JavaScript":false
}
```

3-5-2. JsonbSerializerの実装

- JsonbSerializerを用いたカスタマイズイメージ
  - JsonbAdapterと利用イメージは同様
  - JsonbSerializer Interfaceにオリジナルクラスのカスタムロジックを記述
  - 上記をJsonbConfigに登録して、toJson(Original original)を行う



3-5-2. JsonbSerializerの実装

- 使用例
  - JsonbAdapter同様データ構造や出力形式まで包括的にカスタマイズ可能
  - 次ページ以降で下記のようなカスタマイズ実装例を示す

元クラス

元クラス
```
public class Emp {
        private String id;
        private String name;
        private String department;
(セッター、ゲッター、コンストラクタ省略)
}
```
インスタンス
```
new Emp("a1234", "Yoshiharu Fukui", "cloud");
```

カスタマイズ無し

```
{
    "department": "cloud",
    "id": "a1234",
    "name": "Yoshiharu Fukui"
}
```

カスタマイズ(例)

・指定した順序での出力
・性と名を分割
・キー名を変更
・新たなフィールドを追加

```
{
    "ID": "a1234",
    "氏": "Fukui",
    "名": "Yoshiharu",
    "部門": "cloud",
    "入社日": "Tue May 01 10:33:17 JST 2018"
}
```

3-5-2. JsonbSerializerの実装

- JsonbSerialiserの実装例

```
public class EmpSerializer implements JsonbSerializer<Emp> {
        @Override
        public void serialize ( Emp emp, JsonGenerator jsg, SerializationContext arg2) {

                Calendar c = Calendar.getInstance();
                c.set(2018, 4, 1);

                jsg.writeStartObject();
                jsg.write("ID", emp.getId());
                jsg.write("氏", emp.getName().split(" ")[1]);
                jsg.write("名", emp.getName().split(" ")[0]);
                jsg.write("部門", emp.getDepartment());
                jsg.write("入社日", c.getTime().toString());
                jsg.writeEnd();
        }
}
```

JsonbSerializerの継承

第二引数必要(編集不要)

JSON-Pで用意されているクラス

カスタマイズするオリジナルクラス

自在に編集

3.6. JsonbSerializerの実装

- JsonbConfigへの登録

```
public void serializeWithSerializer() {
        Emp emp = new Emp("a1234", "Yoshiharu Fukui", "cloud");

        JsonbConfig jsonbConfig = new JsonbConfig().withSerializers(new EmpSerializer());

        Jsonb jsonb = JsonbBuilder.newBuilder().withConfig(jsonbConfig).build();

        String actualJson = jsonb.toJson(emp);

        System.out.print(actualJson);
}
```

前ページで作成したEmpserializer インスタンスの登録

JsonbConfig jsonbConfigを登録

```
{
    "ID": "a1234",
    "氏": "Fukui",
    "名": "Yoshiharu",
    "部門": "cloud",
    "入社日": "Tue May 01 10:33:17 JST 2018"
}
```

まとめ

まとめ

- Jsonb Interfaceのcreate(), toJson() メソッドを利用することで、ワンステップでオリジナルクラスをシリアライズ可能
- アノテーションにより手軽にカスタマイズ可能
- デフォルトの設定でも、カスタマイズを設定しても、最終的にはtoJson(), fromJson()を用いるため、理解しやすい
- JsonbAdapter, JsonbSerializerにより、データ構造を含め柔軟にカスタマイズすることが可能
  - クライアントサイド開発者とサーバーサイド開発者の連携がスムーズになる

END