

# 計算機科学第一

2013年度第3回

---

日付

# 本日の内容

---

- ❖ オブジェクトを作成するための new を直接, 用いないテクニック
  - ❖ Static Factory Method
  - ❖ Builder Object



# 課題を出しました

OCW-i を確認して下さい

---

日付

# 1. Static Factory Method

---



# 質問

---

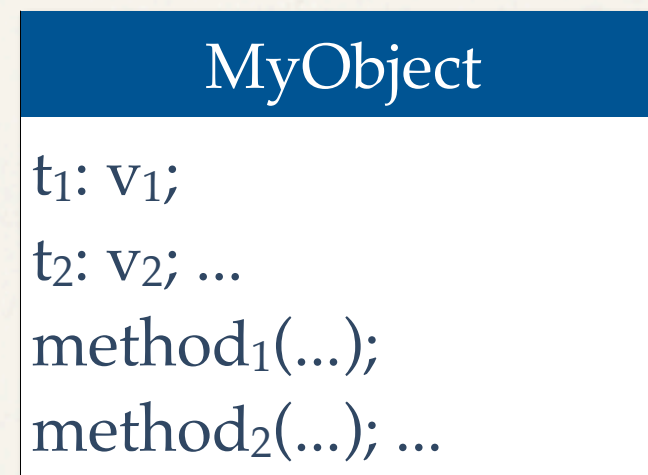
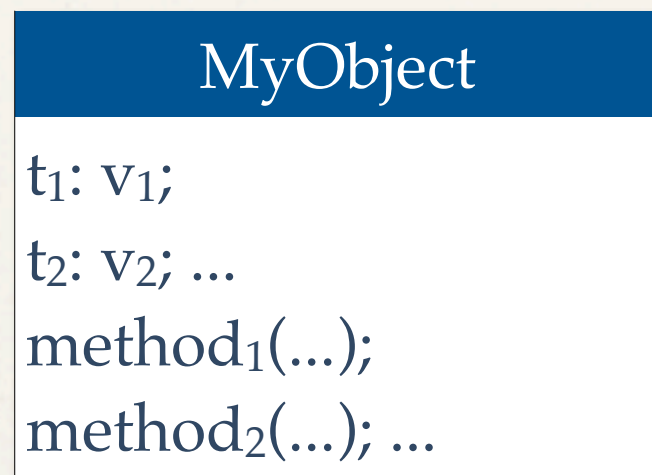
- ❖ あなたはどのようにオブジェクトを生成しますか？

# public constructor

---

❖ **public** MyObject(*arg*<sub>1</sub>, *arg*<sub>2</sub>, ...)

❖ **new** MyObject(*e*<sub>1</sub>, *e*<sub>2</sub>, ...)





# もうひとつの作り方

---

- ❖ public static **factory** method
  - ❖ public: 公開された
  - ❖ **static** method:  
クラスが提供するメソッド  
MyObject.method(...)
  - ❖ factory: (オブジェクト生成の) 工場

# 例 1 : Boolean:valueOf

---

```
public static Boolean valueOf(boolean b) {  
    return b ? Boolean.TRUE :  
            Boolean.FALSE;  
}
```



# 例: BigInteger

---

- ❖ java.math.BigInteger クラス
- ❖ 「無限長精度」整数のクラス
- ❖ 円周率の世界記録に挑むのに使う？

# BigIntegerのコンストラクタ

---

BigInteger(Byte[] val)

BigInteger(int signum, byte[] magnitude)

BigInteger(int length, int certainty, Random r)

BigInteger(int numBits, Random r)

BigInteger(String val)

BigInteger(String val, int radix)



BigInteger(Byte[] val)

BigInteger(int signum, byte[] magnitude)

BigInteger(int length, int certainty, Random r)

BigInteger(int numBits, Random r)

BigInteger(String val)

BigInteger(String val, int radix)



さて 12345 に対応する BigInteger を作りたい. どうする?

# BigInteger の

## static factory method: valueOf

---

### valueOf

```
public static BigInteger valueOf(long val)
```

値が指定された long の値と等しい BigInteger を返します。この「static  
ファクトリメソッド」は、よく使われる BigInteger を再利用できるようにするために、(long) コンストラクタの代わりに提供されます。

パラメータ:

val - 返される BigInteger の値

戻り値:

指定値を使った BigInteger



# BigInteger::valueOfの使用例

---

❖ new BigInteger(1), new BigInteger(2);

```
private void run() {  
    BigInteger power100 = BigInteger.valueOf(1);  
    BigInteger two = BigInteger.valueOf(2);  
    for (int i = 1; i <= 100; i++) {  
        power100 = power100.multiply(two);  
        System.out.printf("2^%d = %s\n", i, power100);  
    }  
}
```

$$\begin{aligned}2^1 &= 2 \\2^2 &= 4 \\2^3 &= 8 \\2^4 &= 16 \\2^5 &= 32 \\2^6 &= 64 \\2^7 &= 128 \\2^8 &= 256 \\2^9 &= 512 \\2^{10} &= 1024\end{aligned}$$

$$\begin{aligned}2^{95} &= 39614081257132168796771975168 \\2^{96} &= 79228162514264337593543950336 \\2^{97} &= 158456325028528675187087900672 \\2^{98} &= 316912650057057350374175801344 \\2^{99} &= 633825300114114700748351602688 \\2^{100} &= 1267650600228229401496703205376\end{aligned}$$



# 因にBigIntegerを使わないと

---

$$2^{28} = 268435456$$

$$2^{29} = 536870912$$

$$2^{30} = 1073741824$$

$$2^{31} = -2147483648 \quad \leftarrow \text{オーバーフロー}$$

$$2^{32} = 0$$

$$2^{33} = 0$$

$$2^{34} = 0$$

# SFM vs. new

static factory method	コンストラクタ (new)
1. 意味のある名前	単なるクラス名 ↑ 分りにくい
2. オブジェクトを再利用してもよい	必ず新しいオブジェクトを生成 ← メモリの無駄
3. subtypeのインスタンスを返してもいい	指定されたクラスのインスタンス ← 柔軟性↓
4. parameterized typeも扱える	難しい



# 1. 意味のある名前

---

<code>static <u>BigInteger</u></code>	<code><u>valueOf</u>(long val)</code>
	値が指定された long の値と等しい BigInteger を返します。

<code>static <u>BigInteger</u></code>	<code><u>probablePrime</u>(int bitLength, <u>Random</u> rnd)</code>
	指定されたビット長で正の BigInteger (おそらく素数) を返します。

# 1. 意味のある名前

---

- ❖ `new Person(String, String, String)`
- ❖ `Person.createFromNameFamilyPhone(  
String /* Name */,  
String /* Family */,  
String /* Phone */)`
- ❖ `Person.createFromNameFamilyEmail(  
String /* Name */,  
String /* Family */,  
String /* Email */)`



## 2. オブジェクトを作らなくていい

---

- ❖ 無駄なオブジェクト生成を抑制し、性能向上

`Boolean.valueOf`

- ❖ *instance-controlled class* を実装できる

- ❖ オブジェクトを新規に生成するか否かを制御するクラス

```
private void test1() {  
    at_start("[standard constructor]");  
    for (int i = 0; i < bools.length; i++)  
        bools[i] = new Boolean(i % 2 == 0);  
    at_stop();  
}  
  
private void test2() {  
    at_start("[static factory method]");  
    for (int i = 0; i < bools.length; i++)  
        bools[i] = Boolean.valueOf(i % 2 == 0);  
    at_stop();  
}
```



[new] :	計算時間	-	31ms ,	メモリ使用量	-	17423104B
[sfm] :	計算時間	-	32ms ,	メモリ使用量	-	0B
[new] :	計算時間	-	18ms ,	メモリ使用量	-	17091168B
[sfm] :	計算時間	-	29ms ,	メモリ使用量	-	0B
[new] :	計算時間	-	17ms ,	メモリ使用量	-	16976400B
[sfm] :	計算時間	-	31ms ,	メモリ使用量	-	0B
[new] :	計算時間	-	18ms ,	メモリ使用量	-	16793280B
[sfm] :	計算時間	-	28ms ,	メモリ使用量	-	0B

[new] :	計算時間	-	31ms ,	メモリ使用量	-	17423104B
[sfm] :	計算時間	-	32ms ,	メモリ使用量	-	0B
[new] :	計算時間	-	18ms ,	メモリ使用量	-	17091168B
[sfm] :	計算時間	-	29ms ,	メモリ使用量	-	0B
[new] :	計算時間	-	17ms ,	メモリ使用量	-	16976400B
[sfm] :	計算時間	-	31ms ,	メモリ使用量	-	0B
[new] :	計算時間	-	18ms ,	メモリ使用量	-	16793280B
[sfm] :	計算時間	-	28ms ,	メモリ使用量	-	0B



# Boolean SFMの中身

---

```
class Boolean {  
    public static final Boolean TRUE = new Boolean(true);  
    public static final Boolean FALSE = new Boolean(false);  
    public static Boolean valueOf(boolean b) {  
        return b ? Boolean.TRUE : Boolean.FALSE;  
    }  
}
```

# Instance-controlled class

---

- ❖ singleton (来週)
- ❖ non-instantiable (来週)
- ❖ immutable



# 3. subtypeを返せる

- ❖ 基本的な集合オブジェクトに機能(`checked`, `synchronized`, ...)を追加するためのAPI

<code>checkedCollection</code>	<code>emptyMap</code>	<code>synchronizedCollection</code>	<code>unmodifiableMap</code>
<code>checkedList</code>	<code>emptySet</code>	<code>synchronizedList</code>	<code>unmodifiableSet</code>
<code>checkedMap</code>	<code>list</code>	<code>synchronizedMap</code>	<code>unmodifiableSortedMap</code>
<code>checkedSet</code>	<code>newSetFromMap</code>	<code>synchronizedSet</code>	<code>unmodifiableSortedSet</code>
<code>checkedSortedMap</code>	<code>singleton</code>	<code>synchronizedSortedMap</code>	
<code>checkedSortedSet</code>	<code>singletonList</code>	<code>synchronizedSortedSet</code>	
<code>emptyList</code>	<code>singletonMap</code>	<code>unmodifiableCollection</code>	

# Collections Frameworkでの効果

---

- ❖ 32のcollections APIたちをCollectionsクラスのstatic factory methodに押し込めてAPIを単純化
  - ❖ checked / empty / singleton / synchronized / unmodifiable
  - ❖ sorted / or not
  - ❖ collection / list / map / set
- ❖ APIが類似しているため、使い方を学ぶのが容易 (conceptual weight が小さい)



## 4. Generics との相性 (1/3)

---

```
private void run2() {  
    Map<String, List<String>> map = U.hashMap();  
    {  
        List<String> list = U.arrayList();  
        list.add("カレーライス");  
        list.add("コーヒー");  
        map.put("昼食", list);  
    }  
}
```

## 4. Generics との相性 (2/3)

---

```
private void run2() {  
    List<String> list = null;  
    Map<String, List<String>> map = U.hashMap();  
    {  
        list = U.arrayList();  
        list.add("カレーライス");  
        list.add("コーヒー");  
        map.put("昼食", list);  
    }  
}
```



## 4. Generics との相性 (3/3)

---

```
public static <V> List<V> arrayList() { return new ArrayList<V>(); }
```

```
public static <V> Set<V> treeSet() { return new TreeSet<V>(); }
```

```
public static <V> Set<V> hashSet() { return new HashSet<V>(); }
```

```
public static <K, V> Map<K, V> hashMap() { return new HashMap<K, V>(); }
```

```
public static <K, V> Map<K, V> treeMap() { return new TreeMap<K, V>(); }
```



# SFMで注意を要する点

---

- ❖ 完全に public/protected constructor を除去してしまうと継承できなくなる。  
→ それはそれで構わないという説もある。
- ❖ JavaDoc のサポートがないので、ドキュメントから static factory method を探しにくい。(constructorなら簡単)  
→ Naming convention で対応する？  
valueOf, of, getInstance, newInstance, getType, newType



# 2. Builder

---

# あなたならどうする？

---

- ❖ プログラムが複雑になるにつれて、クラスのフィールドが増えて、コンストラクタの引数がどんどん増えて、わけがわからなくなってきました。



# 名簿ソフトを作ろうとした

---



Nanashi Gombei  
**名無し ゴンベイ**  
“gombei”

勤務先 0123-45-6789  
携帯電話 090-8765-4321

勤務先 gombei\_nanashi@somewhere.in.the.heaven  
mobile g\_1985\_jul\_7\_ombei@docomo.com

誕生日 July 7, 1985

勤務先 123-4567  
わからん県 よく知らん市  
でも、とっても 1-1-6-5  
日本

# 項目

---

- ❖ 氏名, シメイ, あだ名
- ❖ 電話: 自宅、勤務先、携帯<sub>1</sub>、 携帯<sub>2</sub>
- ❖ 所在地: 自宅、勤務先
- ❖ メール: 自宅、勤務先、 携帯<sub>1</sub>、 携帯<sub>2</sub>
- ❖ 誕生日



# 項目: **必須**、オプション

---

- ❖ **氏名, シメイ**, あだ名
- ❖ 電話: 自宅、勤務先、携帯<sub>1</sub>、携帯<sub>2</sub>
- ❖ 所在地: 自宅、勤務先
- ❖ メール: 自宅、勤務先、携帯<sub>1</sub>、携帯<sub>2</sub>
- ❖ 誕生日

# 素朴な方法

---

❖ Person(  
    String 氏名,  
    String シメイ,  
    String あだ名,  
    String 電話自宅,  
    String 電話勤務先,  
    String 電話携帯1,  
    String 電話携帯2, ...)



# 1. 望遠鏡方式

---

❖ Person(氏名, シメイ)

氏名

シメイ

あだ名

電話

❖ Person(氏名, シメイ, あだ名)

❖ Person(氏名, シメイ, あだ名, 電話自宅)

❖ ...

# 1. 望遠鏡方式

---

```
Person(氏名, シメイ, あだ名, 電話自宅 ){  
    this(氏名, シメイ, あだ名);  
    this.電話番号 = 電話番号;  
}
```



# 1. 望遠鏡方式

---

電話番号を何も知らない人の誕生日を設定する場合

`new Person(“名無しのゴンベイ”, “gombei”,`

`null, null, null, null,`  
`null, null, null, null,`  
`“Jul 7, 1985”)`

← 電話番号はすべて不明

← メールアドレスもすべて不明

# 2. JavaBeans方式

---

- ❖ 引数なしのconstructor
- ❖ 全フィールドに対する setter メソッド



## 2. JavaBeans方式

---

電話番号を何も知らない人の誕生日を設定する場合

```
Person gombei = new Person();
```

```
gombei.氏名("名無しのゴンベイ");
```

```
gombei.あだ名("gombei");
```

```
gombei.誕生日(Jul 7, 1985");
```

## 2. JavaBeans方式の問題点

---

- ❖ 引数間の一貫性の検査ができない  
例: 〒と市町村の整合
- ❖ immutable (不変、代入できない)オブジェクトが作れない



# 3. Builder方式

---

- ✧ 以下を両立するのが目的
  - ✧ 望遠鏡方式の安全性（一貫性検査）
  - ✧ JavaBeans方式の可読性

# PersonBuilder

---

- ❖ Personクラスのインスタンスを作るためのBuilderクラスとして PersonBuilderを作成し、これに対してJavaBeans方式でフィールドの初期値を設定する。
- ❖ Builderクラスのbuildメソッドを使って、Personクラスのインスタンスを生成



# Builderの利点

---

- ❖ 名前つきオプションパラメター
- ❖ 柔軟性
  - ❖ Builderの使い回し
  - ❖ デフォルト値の自動挿入