慶応義塾大学文学部

応用情報処理V 2017

クラスを活用する様々な機能

Chapter 10

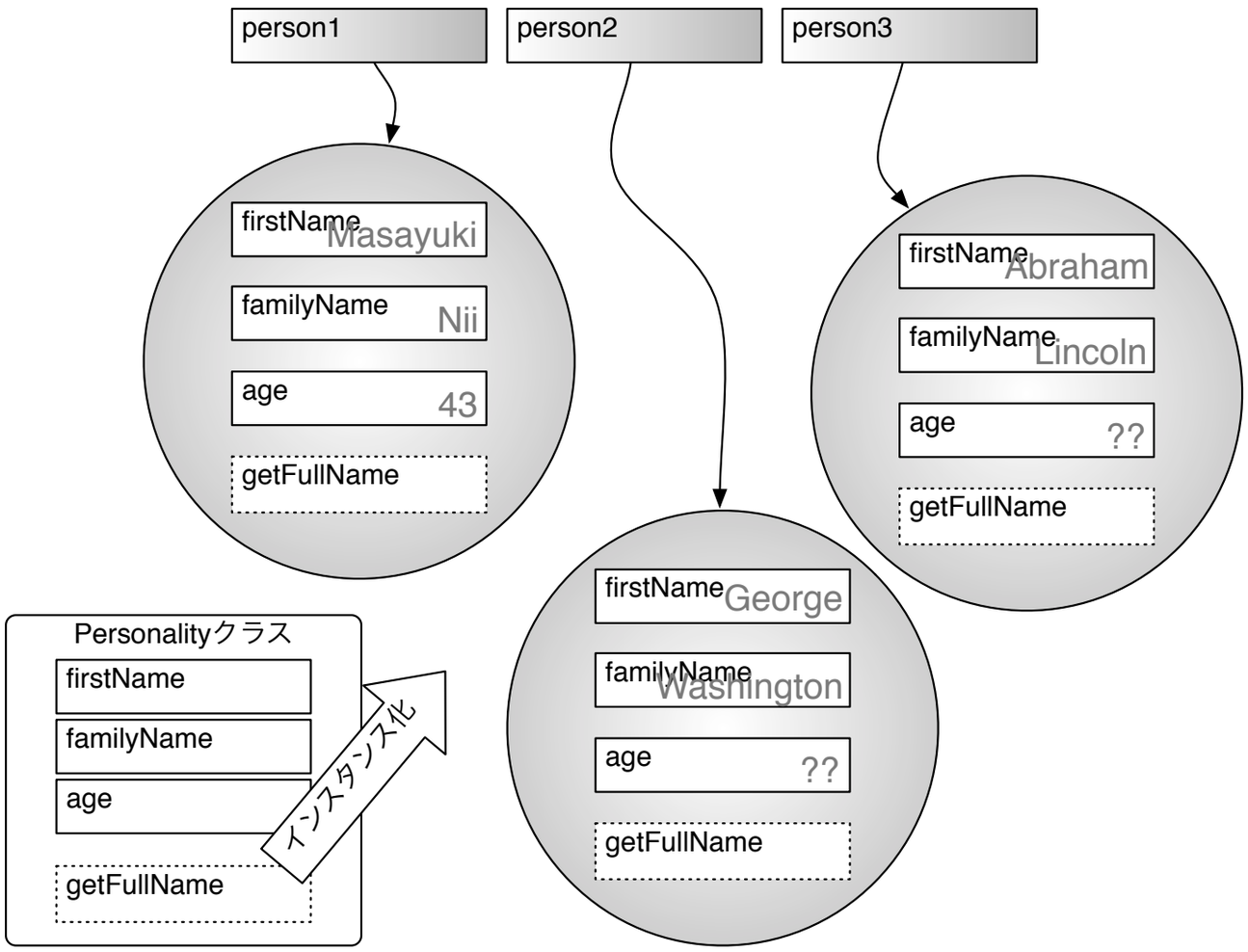
2017-12-08

新居雅行 Masayuki Nii Ph.D. in Engineering

 nii@msyk.net  msyknii  msyk_nii

複数のインスタンス

- **複数のインスタンスを生成した…**
  - 当然のこととして、参照先はそれぞれ違う
  - 1つの変数では覚えられない。原則としてオブジェクトの数だけ参照は必要



staticキーワードによるクラス定義

- **static(静的)キーワードがあると特別な意味を持つ**
  - static class someThing
    - インスタンス化しなくても必ず1つのインスタンスが使える状態になっている
    - クラス名そのもので、そのインスタンスを参照できる
  - staticプロパティ、staticメソッド
    - staticクラスで利用できるプロパティやメソッド
    - クラス自体がstaticはなくても、同じように、必ず1つのインスタンスが使える状態になっている
- **staticインスタンスと動的なインスタンス**
  - 相互利用は（ある意味では）できない
  - 設計上staticにするかどうかは難しい判断

staticを利用している場所

- **アプリケーション実行の最初のとっかかり**

```
class Personality{
    String firstName;
    String familyName
    int age;

    Personality( ){ }
    Personality( String initFirstName, String initFamilyN
        this.firstName = initFirstName;
        this.familyName = initFamilyName;
        this.age = initAge;
    }
    String getFullName() {
        String fullName = this.firstName + " " + this.family
        return fullName;
    }
    // ****** アプリケーションを起動すると、ここが呼び出される ******
    //static なメソッドで、自分自身をインスタンス化している
    public static void main( String args[] ) {
        Personality person1 = new Personality("Masayuki", "Nii", 43);
        System.out.println( person1.getFullName() + "さんの年齢は" + person1.age );
    }
}
```

staticなメソッドがあるので、Personalityで参照できる特別なインスタンスが作られ、mainメソッドの呼び出しができる

Javaは起動時にクラスを指定する

起動直後に指定したクラスの「public static void main(String[])」メソッドを呼び出すのがランタイムの仕様。つまり、Personality.main(…) を実行

このメソッドが最初に実行される

ここのnewにより、「もう1つのPersonalityクラスのインスタンス」が生成される

起動時の処理はなんとかならないか

- **起動時に自分自身を生成する書き方は違和感がある**
  - その理由を初心者に説明する術がない
- **なぜそう書くのか**
  - 簡単なプログラムではクラスを1つに集約させたい？
- **起動時に呼び出す専用クラスを定義するのが自然**
  - public static void main(String[])だけのクラスを作る
  - class Starter {

```
    public static void main( String args[] ) {
        Personality person1 = new Personality("Masayuki", "Nii",
43);
        System.out.println( person1.getFullName() +
            "さんの年齢は" + person1.age );
    }
}
```

クラスの継承

- **あるクラスの定義を引き継ぎ、新たなクラスを作る**
  - 何でも1から定義するのではなく、既存の定義を再利用する手法を一般化したもの
  - 「差分を記述するだけでOK」という考え方でOK
  - 元のクラスで定義されたプロパティやメソッドの追加だけでなく、置き換え、削除など様々な再構成ができる
- **「継承」**
  - クラスAがあり、それを元にクラスBを定義すること
  - Aから見たBをサブクラス、Bから見たAをスーパークラス

継承が便利な事例

- **Personalityクラスにより一人の人をデータ化できる**
  - getFullNameメソッドでフルネームが得られるが、名前が先に来ている。日本人だったら姓が先に来て欲しい
  - アメリカ人なら、姓名のスペルの最初の文字を使ってイニシャルが作成できるではないか
- **継承したクラスの定義**
  - **class JapanesePersonality extends Personality** {

```
    String getFullName() {
        String fullName = this.familyName + " " + this.firstName;
        return fullName;
    }
}
```

  - **class AmericanPersonality extends Personality** {

```
    String getInitial(){
        String initial = this.firstName.substring(0, 1) +
                         "." + this.lastName.substring(0, 1);
        return initial;
    }
}
```

それぞれのクラスで利用できるプロパティとメソッド

	Personality	JapanesePersonality	AmericanPersonality
プロパティ	firstName	firstName	firstName
	familyName	familyName	familyName
	age	age	age
コンストラクター	Personality()	JapanesePersonality()	AmericanPersonality()
	Personality(String, string, int)		
メソッド	getFullName()	**getFullName()**	getFullName()
			getInitial()

コンストラクターは継承されないが、引数なしのものは自動的に組み込まれる

メソッドは同名だが処理は書き換えられた

処理はPersonalityと同じ

新たに追加されたメソッドは、このクラスのみで利用可能

superによる親クラスの参照

- **スーパークラスのコンストラクタを利用する**

```
class Personality{
    Personality( ){ }
    Personality( String initFirstName, String initFamilyName, int initAge )
    {
        this.firstName = initFirstName;
        this.familyName = initFamilyName;
        this.age = initAge;
    }
}
class JapanesePersonality extends Personali
    JapanesePersonality(
        String initFamilyName, String initFirstName, int initAge) {
        super(initFirstName, initFamilyName, initAge);
    }
}
```

同じクラスのコンストラクターの参照は this(…)

引数のないコンストラクターがない場合には、以下のコンストラクターが自動的に作られる

```
JapanesePersonality() {
    super();
}
```

super(…)はコンストラクターの最初でしか呼び出せない

superがない場合は、super()が自動的にコンストラクタの最初に挿入

オブジェクト指向で利用される手法

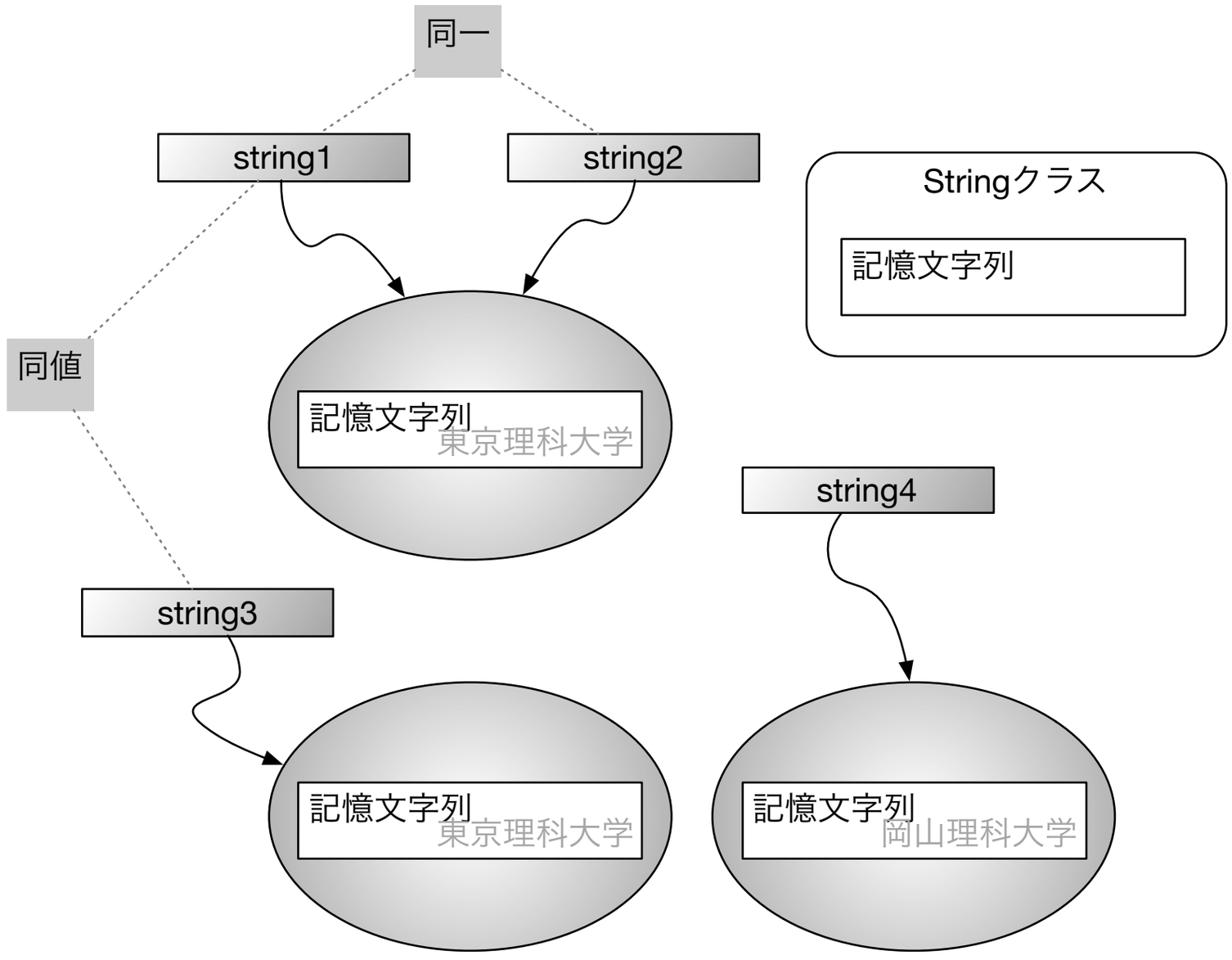
- **引数パターンの異なる複数のメソッドを定義できる**
  - オーバーロード
- **スーパークラスで定義したメソッドを再定義**
  - オーバーライド
  - そのままだと、スーパークラスのメソッドは呼び出されないが、「super.メソッド名(…)」で呼び出すこともできる
- **特殊なクラス**
  - 抽象クラス
    - 半完成クラス。一部のメソッドを、サブクラスで「必ず」実装しないといけない。インスタンスは作れない
  - インタフェース
    - メソッド定義やだけを持つクラス。実装と仕様を分離する目的で使われる。プロパティも定義できるが「定数」的なものになる

オブジェクト指向の利点

- **ソフトウェアの再利用を促進する**
  - 継承：共通の処理がスーパークラス、サブクラスではクラスごとに違う処理だけを組み込めば良い
  - 何階層にも継承できる。正しく設計されれば、階層的に整理されることが期待できる
- **カプセル化が促進される**
  - 必要な情報だけに絞り、一部の情報を隠蔽することで、より理解が容易になるということを期待する
- **ポリモーフィズム**
  - メソッドなどが複数の型で利用できる仕組みのこと
  - 総称化、リスコフの置換原則などの実現

オブジェクトに対する同一と同値

- **同一: Identity**
  - 同じオブジェクト
  - ==演算子で判定
- **同値: Equivalence**
  - 違うオブジェクトでもありうる
  - equalsメソッドなど、判定のメソッドをクラスに用意する
  - 文字列の場合、大文字小文字の扱いなど
  - 同値という概念が状況によって異なる



Stringの特殊性

- **“東京理科大学” だけで、Stringオブジェクトが生成**
  - つまり、 “…“ は、裏では new String(“…”) をしている
  - 記述を容易にするための工夫、C言語の文字列に近い感覚
- **コード中に同一の文字リテラルがあるとき**
  - それぞれをインスタンス化せず、単一のStringオブジェクトを共通に使い回す
  - Stringクラスは変更不可能（Immutable）なので、問題ない
- **ありがちな間違い**
  - 原則！！　中身のチェックはequalsメソッドなど、同値かどうかの判定可能なメソッドを使う
  - ==は同一かどうかの判定になる
  - （次のスライドを参照）

Stringの特殊性を確かめるサンプル

赤いボックスは誤解を示す

newにより新たなオブジェクトが生成されている。同値判断はequalsメソッドなどを使う

同じ文字列かどうかは==で判定できるはずだ

Java

View

Untitled.java

```
1  class IdentityAndEquality {
2      public static void main(String[] args) {
3          System.out.println("tokyo" == new String("tokyo"));
4          // false と出力
5
6          String a="a",b="a";
7          System.out.println(a==b);
8          // true と出力
9      }
10 }
```

なんだ、==で文字列が同じかどうか判定できるね

“a” で生成したオブジェクトを使い回すので、変数aと変数bは同一オブジェクトを参照している

それぞれの文字列リテラルに対するStringオブジェクトが生成されるなら、変数aと変数bは違うオブジェクトを参照しているはずだ

```
false
true
```

Run Succeeded | Time 391 ms | main | Tab Size: 4 | Line 9, Column 14

講義のまとめ

- 1つのクラスから複数のオブジェクトが生成されることは一般的である
- staticは、クラス名そのもので参照できる特殊なオブジェクトと関連している
  - public static void main(String args[])の意味
- 定義されたクラスを元に、差分だけを記述することで新たなクラスを定義できる
  - 一般には抽象的なものからより具体的なものを定義
- 同一値の概念