プログラミング基礎

第10回

継承によるクラスの拡張とグループ化

今回のお題

- （クラスの）継承（教科書 下 p.44～）
- 継承を利用し，既存のクラスを拡張できる

なぜそんな機能があるのか？
⇒少ないプログラムで目的の機能を作るため

（プログラムが多くなれば相応にプログラムの動きを把握することが困難になり
エラー，バグの発生率が上がるから）

教科書（下） p.59「継承の目的」も参照

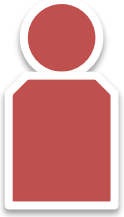
今回のお題

- 継承の利点
- 継承の書き方
- 継承とコンストラクタ
- オーバーライド
- 修飾子（public,protected,private）

継承の利点

継承を用いない場合

Aさん作成



Bさん

ファイルからデータを読み込み，
データを基にある処理Xを行う
クラスP

ファイルからデータを読み込み，
データを基にある処理Yを行う
クラスQを作りたい

解決方法：
Aさんからプログラムを
もらって書き直す

```
class P
{
    void read(){…}
    void X(){…}
}
```

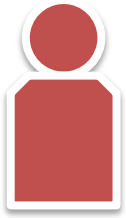
```
class Q
{
    void read(){…}
    void Y(){…}
}
```

クラスPとQでreadメソッドが
重複している（プログラムの量が増える）
⇒もし，バグが見つかれば両方直す必要がある

継承の利点

継承を用いた場合

Aさん作成



Bさん

ファイルからデータを読み込み，
データを基にある処理Xを行う
クラスP

ファイルからデータを読み込み，
データを基にある処理Yを行う
クラスQを作りたい

解決方法：
クラスPを継承しクラスQを作る

```
class P
{
    void read(){…}
    void X(){…}
}
```

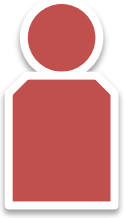
```
class Q extends P
{
    void Y(){…}
}
```

データを読み込む処理
(readメソッド)はクラスPのものを使いまわし，
クラスPに新たに処理Y（メソッドY）を付け加え，クラスQとする

継承の利点

継承を用いた場合

Aさん作成



Bさん

ファイルからデータを読み込み，
データを基にある処理Xを行う
クラスP

ファイルからデータを読み込み，
データを基にある処理Yを行う
クラスQを作りたい

解決方法：
クラスPを継承しクラスQを作る

```
class P
{
    void read(){…}
    void X(){…}
}
```

```
class Q extends P
{
    void Y(){…}
}
```

「親クラス」と呼ぶ

「子クラス」と呼ぶ

処理Y（メソッドY）が増えただけであり，
増えるプログラムの量が抑えられたことになる

継承の書き方

教科書 p.50～

1:継承が使えそうか吟味する

Web資料のコンパクト版で解説:
各学科各学生の成績を保存するためのクラスを考える
（まずは継承を使わない）

FI科学生用

```
class FIStudent
{
   String name;
   int english;
   int cg;
   FIStudent(String n)
   {
       name = n;
   }
   String getName()
   {
      return name;
   }
}
```

見比べると…

重複

重複

FR科学生用

```
class FRStudent
{
   String name;
   int english;
   int robot;
   FRStudent(String n)
   {
       name = n;
   }
   String getName()
   {
      return name;
   }
}
```

2つのクラスは「東京電機大学の学生」の枠でくくることができる
⇒親クラスとして「東京電機大学の学生」クラスを作る

継承の書き方

2:親クラスを作る

FI科学生用

```
class FIStudent
{
   String name;
   int english;
   int cg;
   FIStudent(String n)
   {
       name = n;
   }
   String getName()
   {
      return name;
   }
}
```

東京電機大学学生用

```
class TDUStudent
{
   String name;
   int english;

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

FR科学生用

```
class FRStudent
{
   String name;
   int english;
   int robot;
   FRStudent(String n)
   {
       name = n;
   }
   String getName()
   {
      return name;
   }
}
```

継承の書き方

2:重複箇所を削る

FI科学生用

```
class FIStudent
{
   int cg;

   FIStudent(String n)
   {
       name = n;
   }
}
```

東京電機大学学生用

```
class TDUStudent
{
   String name;
   int english;

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

FR科学生用

```
class FRStudent
{
   int robot;

    FRStudent(String n)
   {
       name = n;
   }
}
```

継承の書き方

2：親クラスを継承させる

FI科学生用

```
class FIStudent extends TDUStudent
{
   int cg;

   FIStudent(String n)
   {
       name = n;
   }
}
```

東京電機大学学生用

```
class TDUStudent
{
   String name;
   int english;

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

FR科学生用

```
class FRStudent extends TDUStudent
{
   int robot;

    FRStudent(String n)
   {
       name = n;
   }
}
```

演習

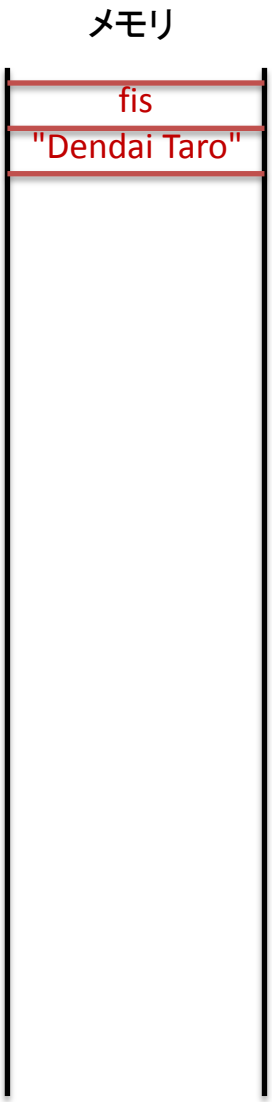
- 実際にクラスを作成し動作を確かめよ
  （プログラム名：StudentTest）

```
//TDUStudentクラスを作成
// TDUStudentを継承しFIStudentクラスを作成
// TDUStudentを継承しFRStudentクラスを作成
public class StudentTest
{
   static public void main(String[] args)
   {
      FIStudent fis = new FIStudent("Dendai Taro");
      FRStudent frs = new FRStudent("Dendai Hanako");

      System.out.println(fis.getName());
      System.out.println(frs.getName());
   }
}
```

どう動いたのか

```
public class StudentTest
{
   static public void main(String[] args)
   {
      FIStudent fis = new FIStudent("Dendai Taro");
      FRStudent frs = new FRStudent("Dendai Hanako");

      System.out.println(fis.getName());
      System.out.println(frs.getName());
   }
}
```

まず変数fis,
引数に指定されている
文字列ができる

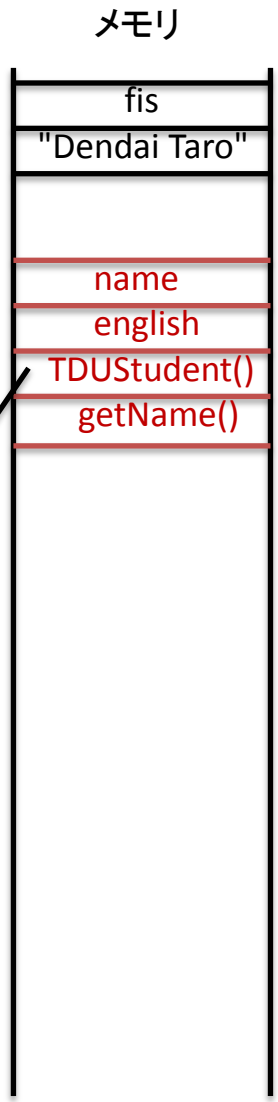


どう動いたのか

```
public class StudentTest
{
   static public void main(String[] args)
   {
      FIStudent fis = new FIStudent("Dendai Taro");
      FRStudent frs = new FRStudent("Dendai Hanako");

      System.out.println(fis.getName());
      System.out.println(frs.getName());
   }
}
```

FIStudentはTDUStudentを継承しているので，まずTDUStudentのインスタンスができる

＊デフォルトコンストラクタ



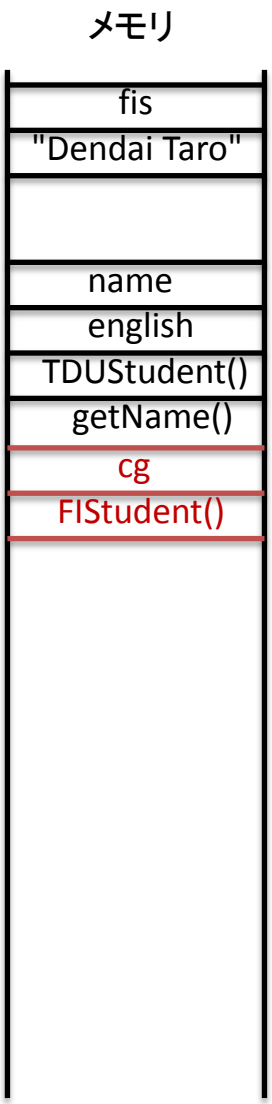

```
class TDUStudent
{
   String name;
   int english;

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

どう動いたのか

```
public class StudentTest
{
   static public void main(String[] args)
   {
      FIStudent fis = new FIStudent("Dendai Taro");
      FRStudent frs = new FRStudent("Dendai Hanako");

      System.out.println(fis.getName());
      System.out.println(frs.getName());
   }
}
```

次に，先ほどの
インスタンスに
FIStudentのデータ
が追加される

継承を使用することで，
親クラスの内容を含んだ
インスタンスが作成される

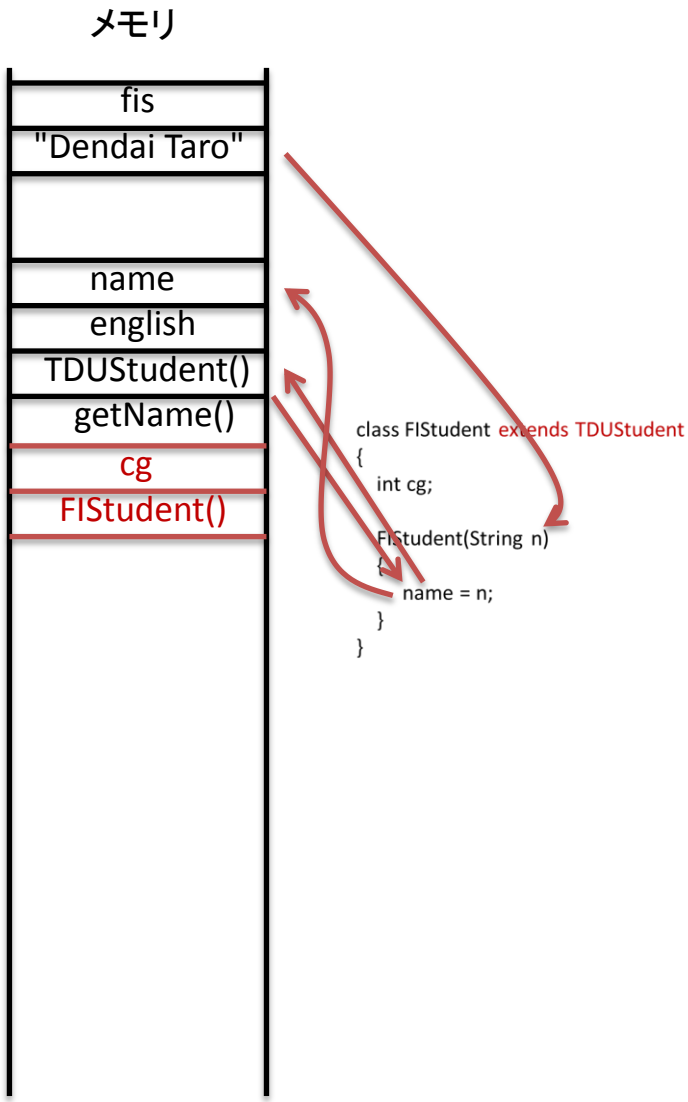



```
class FIStudent extends TDUStudent
{
   int cg;

   FIStudent(String n)
   {
      name = n;
   }
}
```

どう動いたのか

```
public class StudentTest
{
   static public void main(String[] args)
   {
      FIStudent fis = new FIStudent("Dendai Taro");
      FRStudent frs = new FRStudent("Dendai Hanako");

      System.out.println(fis.getName());
      System.out.println(frs.getName());
   }
}
```

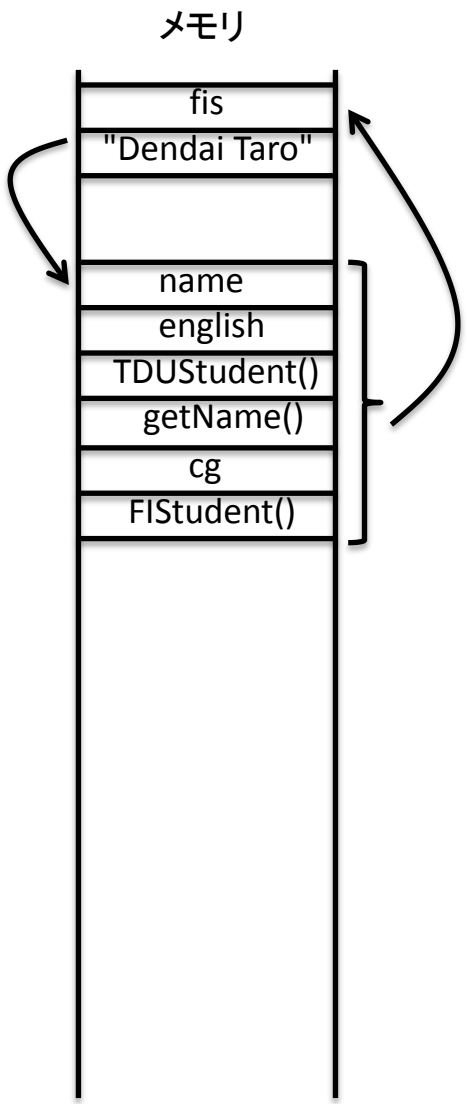
FIStudentコンストラクタが実行される
⇒実は中で勝手に，TDUStudentコンストラクタも実行している
（理由は後述）



どう動いたのか

```
public class StudentTest
{
   static public void main(String[] args)
   {
      FIStudent fis = new FIStudent("Dendai Taro");
      FRStudent frs = new FRStudent("Dendai Hanako");

      System.out.println(fis.getName());
      System.out.println(frs.getName());
   }
}
```

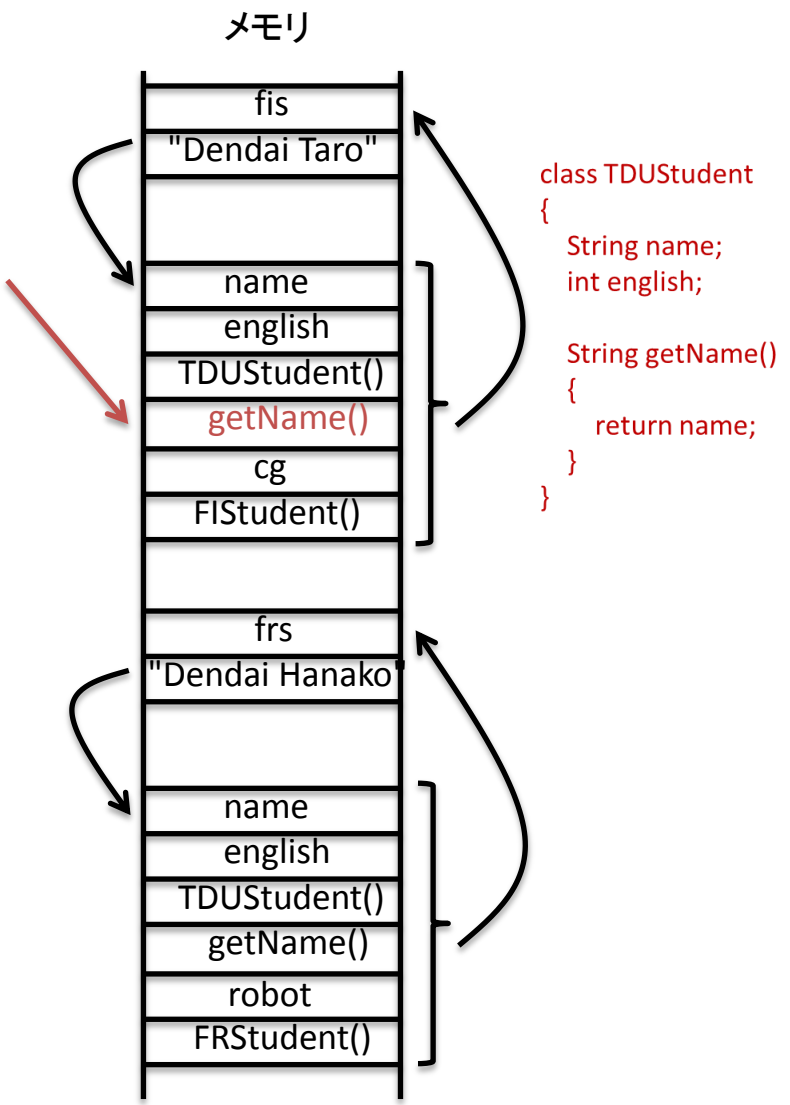
作られた
FIStudentインスタンス
へのアドレスが
fisに代入される



どう動いたのか

```
public class StudentTest
{
   static public void main(String[] args)
   {
      FIStudent fis = new FIStudent("Dendai Taro");
      FRStudent frs = new FRStudent("Dendai Hanako");

      System.out.println(fis.getName());
      System.out.println(frs.getName());
   }
}
```

fisのアドレスが指すインスタンスのgetNameメソッドを使って，名前を取得



継承とコンストラクタ

教科書 p.52～

FI科学生用

```
class FIStudent extends TDUStudent
{
   int cg;

   FIStudent(String n)
   {
      name = n;
   }
}
```

東京電機大学学生用

```
class TDUStudent
{
   String name;
   int english;

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

FR科学生用

```
class FRStudent extends TDUStudent
{
   int robot;

   FRStudent(String n)
   {
      name = n;
   }
}
```

見直してみるとやってることが同じ

継承とコンストラクタ

ではこう書けないか？

FI科学生用

```
class FIStudent extends TDUStudent
{
   int cg;
}
```

東京電機大学学生用

```
class TDUStudent
{
   String name;
   int english;

   TDUStudent(String n)
   {
      name = n;
   }

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

FR科学生用

```
class FRStudent extends TDUStudent
{
   int robot;
}
```

⇒コンストラクタは継承されない

（mainメソッドで，
new FIStudent("Dendai Taro")
と書けなくなる）

継承とコンストラクタ

FI科学生用

```
class FIStudent extends TDUStudent
{
   int cg;
}
```

東京電機大学学生用

```
class TDUStudent
{
   String name;
   int english;

   TDUStudent(String n)
   {
      name = n;
   }

   String getNa
   {
      return name;
   }
}
```

FR科学生用

```
class FRStudent extends TDUStudent
{
   int robot;
}
```

⇒どうせ子クラスで同じことを書くのだから使いまわしたい

継承とコンストラクタ

解決方法：子クラスのコンストラクタで親クラスのコンストラクタを使う（superの利用）

FI科学生用

```
class FIStudent extends TDUStudent
{
   int cg;

   FIStudent(String n)
   {
      super(n);
   }
}
```

superを利用することによって，
親クラスの属性，メソッドを
利用することが出来る

東京電機大学学生用

```
class TDUStudent
{
   String name;
   int english;

   TDUStudent(String n)
   {
      name = n;
   }

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

FR科学生用

```
class FRStudent extends TDUStudent
{
   int robot;

    FIStudent(String n)
   {
      super(n);
   }
}
```

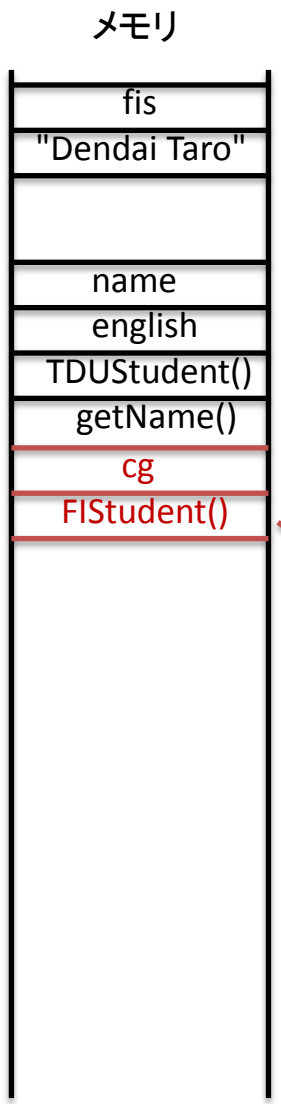
演習

- 実際にsuperを利用し動作を確かめよ
（プログラム名：StudentTest）

どう動いたのか

```
public class StudentTest
{
   static public void main(String[] args)
   {
      FIStudent fis = new FIStudent("Dendai Taro");
      FRStudent frs = new FRStudent("Dendai Hanako");

      System.out.println("FI:" + fis.getName());
      System.out.println("FR:" + frs.getName());
   }
}
```

FIStudentコンストラクタが実行される
⇒super(n)によって，TDUStudentのコンストラクタが実行される




```
class TDUStudent
{
   String name;
   int english;

   TDUStudent(String n)
   {
      name = n;
   }

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

```
class FIStudent extends TDUStudent
{
   int cg;

   FIStudent(String n)
   {
      super(n);
   }
}
```

継承とコンストラクタ

- Javaにおける**子クラスの**コンストラクタのルール
  - 最初の1行で，親クラスのコンストラクタ（オーバーロードされている（複数ある）場合はいずれか1つ）を**必ず**実行しなければならない

```
class FIStudent extends TDUStudent
{
   int cg;

   FIStudent(String n)
   {
      name = n;
   }
}
```

最初の例では，親クラスのコンストラクタを実行していないのに動いていたが？

実は，省略した場合は，自動的に親クラスの**デフォルトコンストラクタ**を呼ぶように書き換えられている

```
class FIStudent extends TDUStudent
{
   int cg;

   FIStudent(String n)
   {
      super();
      name = n;
   }
}
```

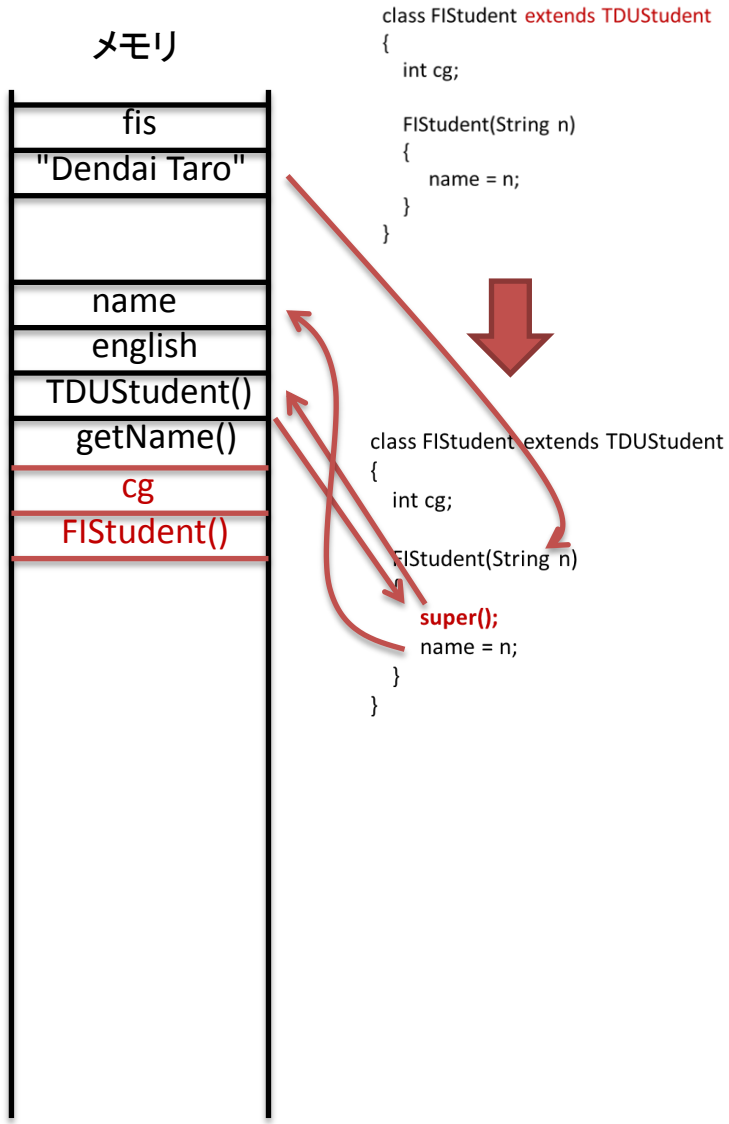
どう動いたのか

```
public class StudentTest
{
   static public void main(String[] args)
   {
      FIStudent fis = new FIStudent("Dendai Taro");
      FRStudent frs = new FRStudent("Dendai Hanako");

      System.out.println("FI:" + fis.getName());
      System.out.println("FR:" + frs.getName());
   }
}
```

FIStudentコンストラクタが実行される
⇒実は中で勝手に，TDUStudentコンストラクタも実行している
（理由は後述）

これがさっきの”理由”

実は，省略した場合は，
自動的に親クラスの
デフォルトコンストラクタ
を呼ぶように書き換えられている



継承とコンストラクタ

というわけで，以下はコンパイルエラーになるので注意

```
class FIStudent extends TDUStudent
{
   int cg;

   FIStudent(String n)
   {
      name = n;
   }
}
```

親クラスのコンストラクタの記述を省略しているので，コンパイル時，自動的にsuper();が追加される

```
class FIStudent extends TDUStudent
{
   int cg;

   FIStudent(String n)
   {
      super();
      name = n;
   }
}
```

しかし，親クラスにデフォルトコンストラクタが存在しないのでコンパイルエラー

```
class TDUStudent
{
   String name;
   int english;

   TDUStudent(String n)
   {
      name = n;
   }

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

分かりづらいエラーなので注意

オーバーライド

教科書 p.60～

getNameメソッドを改良し，名前の後ろに所属の文字列を付け加えたいとする

単純に記述した場合（各クラスで所属の文字列を加えるgetNameメソッドを作る）

```
class FIStudent extends TDUStudent
{
   int cg;

   FIStudent(String n)
   {
      super(n);
   }

   String getName()
   {
      return name + ":TDU" + ":" + "FI";
   }
}
```

```
class FRStudent extends TDUStudent
{
   int robot;

    FIStudent(String n)
   {
      super(n);
   }

   String getName()
   {
      return name + ":TDU" + ":" + "FR";
   }
}
```

親クラス（TDUStudent）のsetNameメソッドが子クラスのsetNameメソッドで上書きされる
⇒オーバーライドされる

オーバーライド

getNameメソッドを改良し，名前の後ろに所属の文字列を付け加えたいとする

単純に記述した場合（各クラスで所属の文字列を加えるgetNameメソッドを作る）

```
class FIStudent extends TDUStudent
{
   int cg;

   FIStudent(String n)
   {
      super(n);
   }

   String getName()
   {
      return name + ":TDU" + ":" + "FI";
   }
}
```

```
class FRStudent extends TDUStudent
{
   int robot;

    FIStudent(String n)
   {
      super(n);
   }

   String getName()
   {
      return name + ":TDU" + ":" + "FR";
   }
}
```

⇒また見直してみると
プログラムが
重複している

親クラス（TDUStudent）
へもっていく

オーバーライド

getNameメソッドを改良し，名前の後ろに所属の文字列を付け加えたいとする

```
class TDUStudent
{
   String name;
   int english;

   TDUStudent(String n)
   {
      name = n;
   }

   String getName()
   {
      return name + ":TDU" + ":" ;
   }
}
```

⇒あとは各クラスで，学科名を付け加えればよい

オーバーライド

getNameメソッドを改良し，名前の後ろに所属の文字列を付け加えたいとする

```
class FIStudent extends TDUStudent
{
   int cg;

   FIStudent(String n)
   {
      super(n);
   }

   String getName()
   {
      return name + ":TDU" + ":" + "FI";
   }
}
```

```
class FRStudent extends TDUStudent
{
   int robot;

    FIStudent(String n)
   {
      super(n);
   }

   String getName()
   {
      return name + ":TDU" + ":" + "FR";
   }
}
```

⇒ここは親クラスの
getNameメソッド
を使いたい

オーバーライド

getNameメソッドを改良し，名前の後ろに所属の文字列を付け加えたいとする

各クラスでは親クラスのsetNameメソッドを利用するように書き換える
⇒superを利用する

```
class FIStudent extends TDUStudent
{
   int cg;

   FIStudent(String n)
   {
      super(n);
   }

   String getName()
   {
       double str = super.getName();
       return str + "FI";
   }
}
```

```
class FRStudent extends TDUStudent
{
   int robot;

    FIStudent(String n)
   {
      super(n);
   }

   String getName()
   {
      double str = super.getName();
       return str + "FR";
   }
}
```

演習

- 実際にオーバーライドを利用し動作を確かめよ
  （プログラム名 : StudentTest）

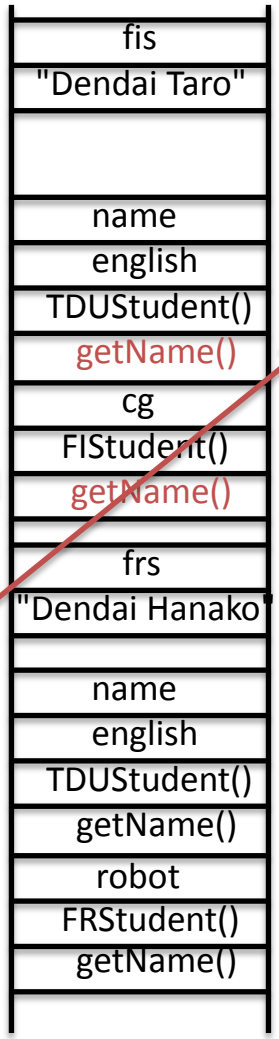
どう動いたのか

```
public class StudentTest
{
   static public void main(String[] args)
   {
      FIStudent fis = new FIStudent("Dendai Taro");
      FRStudent frs = new FRStudent("Dendai Hanako");

      System.out.println(fis.getName());
      System.out.println(frs.getName());
   }
}
```

オーバーライドされているので，まずFIStudentのgetNameメソッドが実行される

```
class FIStudent extends TDUStudent
{
   int cg;

   FIStudent(String n)
   {
      super(n);
   }

   String getName()
   {
      double str = super.getName();
      return str + "FI";
   }
}
```

super.getName()により，TDUStudent（親クラス）のgetName()メソッドが実行される




```
class TDUStudent
{
   String name;
   int english;

   TDUStudent(String n)
   {
      name = n;
   }

   String getName()
   {
      return name + ":TDU" + ":" ;
   }
}
```

TDUStudentのgetNameを実行後また，FIStudentのgetNameメソッドへ戻る

オーバーライド（上書き）という表現をしているがオーバーライドしても親クラスのメソッドは使えることに注意

多態性とオーバーライド 教科書 p.62～

- ある講義でFIとFRの学生が混じっていたとする，その講義の名簿データを作るには…
- 単純な方法：

```
FIStudent[] fis = new FIStudent[NUM_FI];
FRStudent[] frs = new FRStudent[NUM_FR];
```

この場合，受講者の一覧を表示するには…

```
for(int i = 0; i < fis.length; i++)
{
   System.out.println(fis[i].getName());
}
for(int i = 0; i < frs.length; i++)
{
   System.out.println(frs[i].getName());
}
```

しかし，さらに他学科の学生が増えていくと
⇒似たようなプログラムがどんどん増えていく

多態性とオーバーライド

- そこで多態性を利用する

```
TDUStudent[] students = new TDUStudent[NUM_STUDENTS];

//学生登録時
students[0] = new FIStudent("Dendai Taro");
students[1] = new FRStudent("Dendai Hanako");
…

この場合，受講者の一覧を表示するには…
for(int i = 0; i < students.length; i++)
{
   System.out.println(students[i].getName());
}
```

FIStudent，FRStudentは，TDUStudentの子クラスなので，TDUStudentとして扱うことが出来る
⇒多態性

FIStudent，FRStudentをTDUStudentとして扱っているが実際には，子クラスによってオーバーライドされたgetName（各学科の文字列付）が自動的に実行される

多態性とオーバーライド

- 多態性を用いることで，場合分けの処理が不要になる
- これが多態性の大きな利点

演習

- 実際に多態性を利用し動作を確かめよ
（プログラム名：StudentTest）

```
public class StudentTest
{
   static public void main(String[] args)
   {
      TDUStudent[] students = new TDUStudent[2];
      students[0] = new FIStudent("Dendai Taro");
      students[1] = new FRStudent("Dendai Hanako");

      for(int i = 0; i < students.length; i++)
      {
         System.out.println(students[i].getName());
      }
   }
}
```

修飾子（public,protected,private）

教科書 p.65～

属性，メソッドの使える（参照できる）範囲を
指定するために使う

public：いかなるクラスからも使う（参照する）ことができる

protected：子クラスからは使う（参照する）ことができる
その他のクラスからは不可

private：自クラス内のみで使う（参照する）ことができる

何も書かない：同じファイル内から使う（参照する）ことができる

修飾子（public,protected,private）

属性，メソッドの使える（参照できる）範囲を指定するために使う

親クラス

```
class Parent
{
   public metodA(){…}
   protected methodB(){…}
   private methodC(){…}
}
```

子クラス

```
class Child extends Parent
{
   void methodD()
   {
      methodA(); //OK
      methodB(); //OK
      methodC(); //NG
   };
}
```

```
void main(String[] args)
{
   Parent p = new Parent();
   Child c = new Child();
   p.methodA(); //OK
   p.methodB(); //NG
   p.methodC(); //NG
   c.methodA(); //OK
   c.methodB(); //NG
   c.methodC(); //NG
}
```

修飾子（public,protected,private）

利用例：

属性，メソッドの使える（参照できる）範囲を
指定するために使う

親クラス

```
class Parent
{
   private Data[] datas; //子クラスでむやみに直接データを編集されると困るのでprivate
   protected addData(){…}  //子クラスではデータの追加しかできないようにしておく
                           //また，その他のクラスからはデータを追加させない
   public getData(){…} //その他のクラスからデータの閲覧だけは可能
}
```

子クラス

```
class Child extends Parent
{
   public ReadDataFromText()
   {
       addData(…);
   }
}
```

修飾子（public,protected,private）

属性，メソッドの使える（参照できる）範囲を
指定するために使う

子クラスでは，どの箇所を拡張できるのか，
どのような拡張ができるのかを，
明確に指示するために使用する
（親クラスを作った人が想定しない使い方
で拡張されることを抑制するため）

継承は万能ではなく，使い方が悪ければ，
逆に分かりにくいプログラムを生み，
プログラム量増加の原因にもなりかねない