Инкапсуляция. Практическое применение.

Создаем класс Point(). Данные не инкапсулированы.

**public class** Point {  
 **public int x**;  
 **public int y**;  
}

Соответственно получаем доступ к данным создаваемых объектов (координаты **x,y**) с любой точки программы:

Point point = **new** Point();  
 point.x = 3;  
 point.**y** = 5;  
 **int** coordinateX = point.x;  
 **int** coordinateY = point.**y**;

( см. пример 01\_Encapsulation\_01 )

В чем может быть проблема?

Существует весьма распространённое мнение, что основная проблема в данном случае в безопасности кода: мол, необходимо защищать классы от "неправильного" использования. Т.е. если какой-нибудь программист по злому умыслу или случайно установит этим полям неверные значения (изменит координаты), то после таких изменений объект станет работать неправильным образом и могут появиться ошибки. Соответственно для защиты таких полей рекомендуется объявить их private.

**public class** Point {  
 **private int x**;  
 **private int y**;  
  
 **public int** getX() {  
 **return x**;  
 }  
  
 **public void** setX(**int** x) {  
 **this**.**x** = x;  
 }  
  
 **public int** getY() {  
 **return y**;  
 }  
  
 **public void** setY(**int** y) {  
 **this**.**y** = y;  
 }  
}

Давайте посмотрим, так ли это.

Закрываем данные модификатором private. Создаем методы доступа к полям ( getters & setters ). Смотрим на безопасность. Серьезно ее улучшить не удалось. Т.е. с одной стороны получить прямой доступ к полям не представляется возможным, однако в классе появились специализированные методы, которые предоставляют сравнимый с ранее получаемым напрямую доступом, доступ к координатам точки. Другими словами, программист, как и раньше, по злому умыслу или случайно может изменить этим полям значения на ошибочные:

Point point = **new** Point();  
point.setX(3);  
point.setY(5);  
**int** coordinateX = point.getX();  
**int** coordinateY = point.getY();

Т.е. безопасностью кода объяснить необходимость инкапсулировать данные не получается. Соответственно должна быть другая причина для этого.

( см. пример 01\_Encapsulation\_02 )

Представим ситуацию, что данный класс принадлежит достаточно большой программе, где координаты точек **x** и **y**, со временем, стали использоваться во множестве других объектов. И, в какой-то момент, возникла необходимость изменения типа данных координат точек **x** и **y** с **int** на **double**:

**private** */\*int\*/* **double x**;  
**private** */\*int\*/* **double y**;

Соответственно, при не инкапсулированных данных разработчикам программы пришлось бы во всех местах, где ранее использовались координаты точек **x** и **y** произвести изменения данных на соответствующий тип (в данном случае на **double**), что могло оказаться очень ресурсо – и капитало - емко. При этом, особо хочется отметить, что таких изменений в реальности не требовалось, т.к. в тех местах, где ранее были использованы координаты точек **x** и **y**, они спокойно могли продолжаться использоваться в целочисленном варианте.

Что позволяет нам реализовать с минимумом изменений инкапсулированные данные:

**private** */\*int\*/* **double x**;  
**private** */\*int\*/* **double y**;  
  
**public int** getX() {  
 **return** (**int**)**x**;  
}  
  
**public void** setX(**int** x) {  
 **this**.**x** = x;  
}  
  
**public int** getY() {  
 **return** (**int**)**y**;  
}  
  
**public void** setY(**int** y) {  
 **this**.**y** = y;  
}

после таких изменений все места, где ранее использовались координат точек **x** и **y** в **int** виде не требуют изменений. Осталось только создать новые соответствующие методы, которые предоставят нам возможность использования данных в **double** виде:

**public double** getDoubleX() {  
 **return x**;  
}  
  
**public void** setDoubleX(**double** x) {  
 **this**.**x** = x;  
}  
  
**public double** getDoubleY() {  
 **return y**;  
}  
  
**public void** setDoubleY(**double** y) {  
 **this**.**y** = y;  
}

,что позволит разработчикам программы в дальнейшем использовать координат точек **x** и **y** в **double** виде:

point.setDoubleX(4.);  
point.setDoubleY(7.);  
**double** coordinateDoubleX = point.getDoubleX();  
**double** coordinateDoubleY = point.getDoubleY();

( см. пример 01\_Encapsulation\_03 )

Особо стоит отметить тот факт, что в случае инкапсулированных данных, при изменении формата данных, все изменения происходили только в соответствующем классе.

И, в заключение, пример, который еще раз показывает пользу от инкапсулирования данных:

**class** First {  
 **int value** = 3;  
 **void** addFive() {  
 **value** += 5;  
 System.***out***.println(**"first "**);  
 }  
}  
**class** Second **extends** First{  
 **int value** = 2;  
 **void** addFive () {  
 **value** += 5;  
 System.***out***.println(**"second "**);  
 }  
}  
**public class** Main {  
 **public static void** main(String[] args) {  
First f = **new** Second();  
 f.addFive();  
 System.***out***.println(f.**value**);  
 }  
}

Здесь ответ:

second

3

Что вызывает иногда непонимание. Дело в том, данные вызываются по ссылке (она у нас типа First), а методы по типу класса (он у нас Second).

Поэтому и получается 3 (значение **value** из класса First), в то время как в классе Second **value** = 7 (2 + 5).

Соответственно, это еще одна причина инкапсулировать данные.