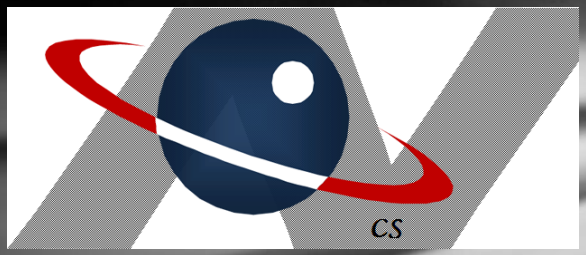
# O(N) CS LESSONS

*Lesson 10B – Class Design*




*By John B. Owen*

# Table of Contents

# Objectives

- In this lesson we will expand the design of the Student class, and will begin to explore the three primary aspects of OOP, namely
  - Encapsulation
  - Inheritance
  - Polymorphism

# The OOP "Tri-pod"

- As mentioned in the last slide, the three legs on which Object Oriented Programming stand are:
  - Encapsulation
  - Inheritance
  - Polymorphism

# Encapsulation

- This first "leg" of OOP refers to the all-encompassing nature of how a class is designed, including the data of the object, as well as any methods it needs.

# Encapsulation

- Abstraction and information hiding are often related to this aspect of designing OOP classes.
- Both refer to the idea that the user of the class may not always be aware of the inner workings of the class, but is still able to use it.

# Encapsulation

- In the Student class previously explored, the name and age instance fields, as well as three methods - the two constructors and the toString method - are the significant parts of the encapsulation for this class.

# Student class

Student.java

```
1  public class Student
2  {
3      String name;
4      int age;
5      public Student(String n, int a)
6      {
7          name = n;
8          age = a;
9      }
10     public Student()
11     {
12         name = "Joe";
13         age = 16;
14     }
15     public String toString()
16     {
17         return "Name="+name+" Age="+age;
18     }
19     public static void main(String [] args)
20     {
21         Student s = new Student();
22         System.out.println(s);
23         s = new Student("Sally",15);
24         System.out.println(s);
25
26     }
```

# Inheritance

- This second "leg" of OOP design refers to the fact that classes inherit features from previously defined classes, most significantly the Object class, which we explored earlier.

# Inheritance

- We discussed and demonstrated earlier how five of the Object class methods work – clone, equals, hashCode, finalize, and toString – and how the toString method could be customized for the Student class.

# Inheritance

- In the next lesson we will explore this further when we create a Person class, which will serve as a super class for the Student class.
- We will then design an Employee class which will also inherit the Person class.

# Composition

- This aspect of OOP is closely related to Inheritance and Encapsulation, and refers to the process of building a class that uses other, pre-defined class as components of the class.

# "Is A" vs "Has A"

- The key distinction between Inheritance and Composition is easily understood by using the phrases "is a" and "has a".

# "Is A" vs "Has A"

- For example, if a Student class inherits a Person class, the Student class "is a" Person.
- This is an example of Inheritance.

# "Is A" vs "Has A"

- If a Student uses another class, say a Grade class as a part of its data, then it can be said that the Student class "has a" Grade.
- This is an example of Composition.

# Polymorphism

- This aspect of OOP design refers to how methods in a class can take "many forms", an idea explored earlier when two constructors were designed for the Student class, a default constructor, and another two-parameter constructor.
- This is called "over-loading".

# Polymorphism

- We also customized the toString method inherited from the Object class in order to provide a better output for the Student class.
- This is called "over-riding".

# Information Hiding

- Let's now discuss the feature of Encapsulation where information from a class can be "hidden", or protected from the outside world, allowing access to it only in a controlled way.

# Information Hiding

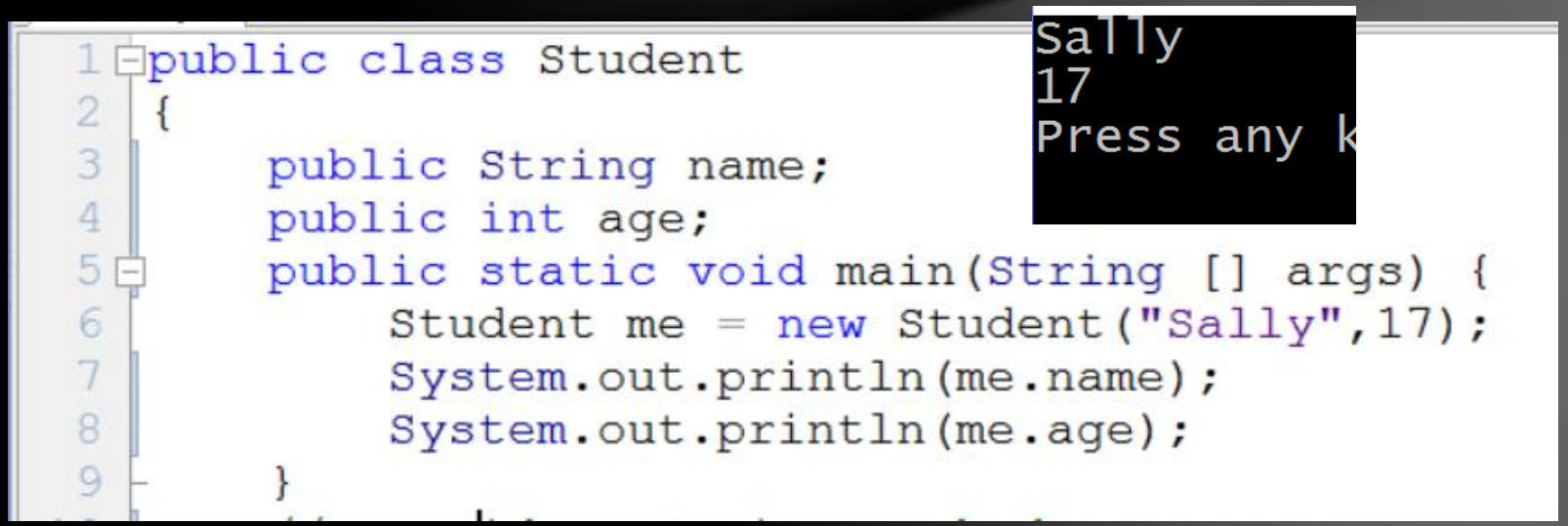
- This aspect of OOP design is crucial since most data is sensitive and important to protect, such as bank account balances, passwords, private information, medical records, etc.

# Information Hiding

- To make data secure, we use the Java reserved word "private" in front of each data item, which allows ONLY that particular object, or other member objects of that class, access to that data.
- Let's look at this with our sample program.

# Public access

- In the Student class we defined earlier, there were two instance fields, name and age.



```
1  public class Student
2  {
3      public String name;
4      public int age;
5      public static void main(String [] args) {
6          Student me = new Student("Sally",17);
7          System.out.println(me.name);
8          System.out.println(me.age);
9      }
```

```
Sally
17
Press any k
```

# Public access

- They are public by default, even if the word public is not included, as was the case in the original example (see slide 8).

```
1  public class Student
2  {
3      public String name;
4      public int age;
5      public static void main(String [] args) {
6          Student me = new Student("Sally",17);
7          System.out.println(me.name);
8          System.out.println(me.age);
9      }
```

```
Sally
17
Press any k
```

# Public access

- Public access simply means they can be accessed directly from any program portion external to the class definition.

```
1  public class Student
2  {
3      public String name;
4      public int age;
5      public static void main(String [] args) {
6          Student me = new Student("Sally",17);
7          System.out.println(me.name);
8          System.out.println(me.age);
9      }
```

```
Sally
17
Press any k
```

# Public access

- This public data can be accessed from main directly, a static method that belongs to the class, but not directly to an object.

```
1  public class Student
2  {
3      public String name;
4      public int age;
5      public static void main(String [] args) {
6          Student me = new Student("Sally",17);
7          System.out.println(me.name);
8          System.out.println(me.age);
9      }
```

```
Sally
17
Press any k
```

# Public access

- By simply stating me.name and me.age, the data in this object can be directly accessed, as you can see in the output statements below.

```
1  public class Student
2  {
3      public String name;
4      public int age;
5      public static void main(String [] args) {
6          Student me = new Student("Sally",17);
7          System.out.println(me.name);
8          System.out.println(me.age);
9      }
```

```
Sally
17
Press any k
```

# Public access

- Further evidence is shown below, where direct changes are made to the name and age fields.

Student.java

```
1  public class Student
2  {
3      public String name;
4      public int age;
5      public static void main(String [] args) {
6          Student me = new Student("Sally",17);
7          System.out.println(me.name);
8          System.out.println(me.age);
9          me.name = "Joe";
10         me.age = 15;
11         System.out.println(me);
12     }
```

```
Sally
17
Name: Joe Age: 15
Press any key to c
```

# Public access

- It stands to reason that this is NOT a good thing, especially for sensitive data.

Student.java

```
1  public class Student
2  {
3      public String name;
4      public int age;
5      public static void main(String [] args) {
6          Student me = new Student("Sally",17);
7          System.out.println(me.name);
8          System.out.println(me.age);
9          me.name = "Joe";
10         me.age = 15;
11         System.out.println(me);
12     }
```

```
Sally
17
Name: Joe Age: 15
Press any key to c
```

# Private access

- To fix this, public is changed to private. This should result in a compile error, but doesn't.

Student.java

```
1  public class Student
2  {
3      private String name;
4      private int age;
5      public static void main(String [] args) {
6          Student me = new Student("Sally",17
7          System.out.println(me.name);
8          System.out.println(me.age);
9          me.name = "Joe";
10         me.age = 15;
11         System.out.println(me);
12     }
```

```
Sally
17
Name: Joe Age: 15
Press any key to c
```


# Private access

- Here's why…since main is a member of this class definition, it can "see" all of the private data

Student.java

```
1  public class Student
2  {
3      private String name;
4      private int age;
5      public static void main(String [] args) {
6          Student me = new Student("Sally",17
7          System.out.println(me.name);
8          System.out.println(me.age);
9          me.name = "Joe";
10         me.age = 15;
11         System.out.println(me);
12     }
```

```
Sally
17
Name: Joe Age: 15
Press any key to c
```


# Private access

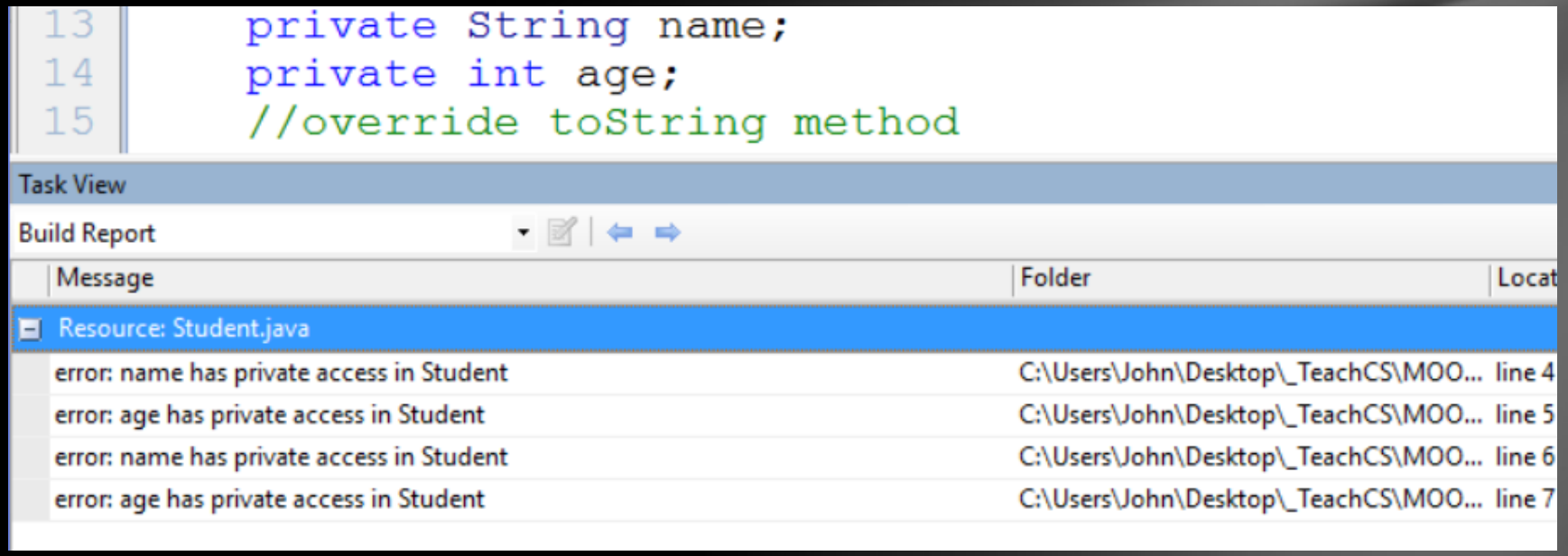
- main is now in a separate class, no longer a member of the Student class.

Student.java

```
1  class Tester{
2      public static void main(String [] args) {
3          Student me = new Student("Sally",17);
4          System.out.println(me.name);
5          System.out.println(me.age);
6          me.name = "Joe";
7          me.age = 15;
8          System.out.println(me);
9      }
10 }
11 public class Student
12 {
13     private String name;
14     private int age;
15     //override toString method
```

# Private access

- The result is several compile error messages, as you can see below.



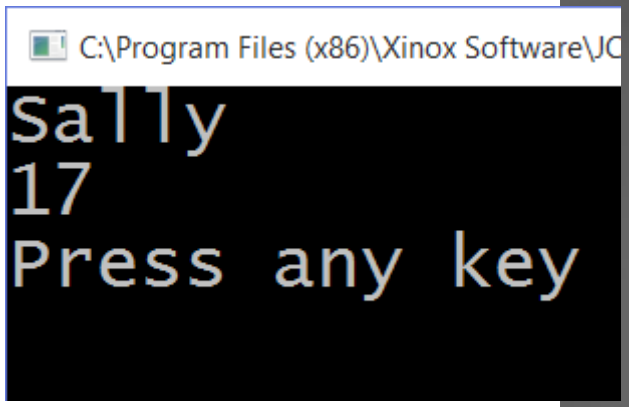
# Special access methods

- The proper way to access these private data fields is by using special methods called accessors and modifiers (mutators)
- These methods are often named using the prefixes set and get.
- Study the code on the next slide carefully.

# Accessor methods – "get"

Student.java

```
1  class Tester{
2      public static void main(String [] args) {
3          Student me = new Student("Sally",17);
4          System.out.println(me.getName());
5          System.out.println(me.getAge());
6  //      me.name = "Joe";
7  //      me.age = 15;
8  //      System.out.println(me);
9      }
10 }
11 public class Student
12 {
13     private String name;
14     private int age;
15     //Accessor methods
16     public String getName(){
17         return name;
18     }
19     public int getAge(){
20         return age;
21     }
```




# Accessor methods – "get"

- The "get" methods shown are simple in nature, and may seem like extra work, but are crucial to the information hiding process.

```
15      //Accessor methods
16      public String getName(){
17          return name;
18      }
19      public int getAge(){
20          return age;
21      }
```

# Accessor methods – "get"

- The format is simple –
  - Start with the word public
  - …then the type of data

```
15      //Accessor methods
16      public String getName(){
17          return name;
18      }
19      public int getAge(){
20          return age;
21      }
```

# Accessor methods – "get"

- …then a method name that starts with the word get, followed by the capitalized name of the field, with an empty parentheses at the end.

```
15      //Accessor methods
16      public String getName(){
17          return name;
18      }
19      public int getAge(){
20          return age;
21      }
```

# Accessor methods – "get"

- The only statement required is a return statement with the name of the field.

```
15      //Accessor methods
16      public String getName(){
17          return name;
18      }
19      public int getAge(){
20          return age;
21      }
```

# Accessor methods – "get"

- If desired, extra access protection could be provided inside this method by requiring some kind of password value through the parameter list.

```
15      //Accessor methods
16      public String getName(){
17          return name;
18      }
19      public int getAge(){
20          return age;
21      }
```

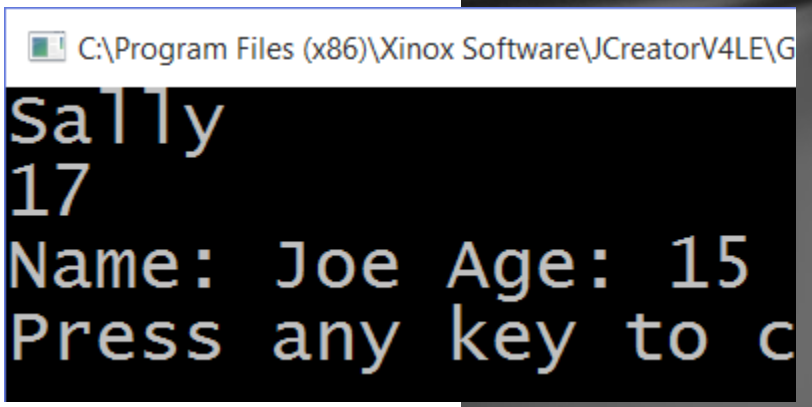
# Modifier, or mutator methods – "set"

- On the next slide see examples of modifier methods.

# Modifier, or mutator methods – "set"

Student.java

```
2       public static void main(String [] args) {
3           Student me = new Student("Sally",17);
4           System.out.println(me.getName());
5           System.out.println(me.getAge());
6           me.setName("Joe");
7           me.setAge(15);
8           System.out.println(me);
9       }
10  }
11  public class Student
12  {
13      private String name;
14      private int age;
15      //Modifiers, or mutators
16      public void setName(String n){
17          name = n;
18      }
19      public void setAge(int a){
20          age = a;
21      }
```



# Modifier, or mutator methods – "set"

- This format is simple, but different –
  - Start with the word public
  - …then the word void – this is NOT a return method – it just performs an action

```
//Modifiers, or mutators
public void setName(String n){
    name = n;
}
public void setAge(int a){
    age = a;
}
```

# Modifier, or mutator methods – "set"

- …then the method name, starting with the word "set", and the capitalized name of the field.

```
//Modifiers, or mutators
public void setName(String n){
    name = n;
}
public void setAge(int a){
    age = a;
}
```

# Modifier, or mutator methods – "set"

- The formal parameter must match the type of data, and can be named anything, usually an abbreviation of the matching instance field.

```
//Modifiers, or mutators
public void setName(String n){
    name = n;
}
public void setAge(int a){
    age = a;
}
```

# Modifier, or mutator methods – "set"

- It is NOT a good idea to use exactly the same name as the field, as this provides some confusion and requires extra code to clarify the scope.

```
//Modifiers, or mutators
public void setName(String name){
    name = name;
}
public void setAge(int age){
    age = age;
}
//Accessor methods
public String getName(){
    return name;
}
```

# Modifier, or mutator methods – "set"

- The only statement required is the assignment statement that gives the value of the parameter to the instance field.

```
//Modifiers, or mutators
public void setName(String n){
    name = n;
}
public void setAge(int a){
    age = a;
}
```

# Lesson Summary

- In this lesson, you learned all about the three main aspects of OOP, namely Encapsulation, Inheritance, and Polymorphism
- You also explored the notion of public vs private, and how to create access methods.

# Lab

- Create a Tester class as shown in this lesson and move the main method to that class.
- Using the Student class, add one more private instance field called gpa, which will be a decimal value.
- Adjust the two constructors and the toString method to include the new field

# Lab

- Write the get and set method for the new field.
- Expand the main method to include access to this new field, in the same way as the two other fields are accessed and modified.

# CONGRATULATIONS!

- Now you know how to protect the private data in your program by using access methods in your class definition.
- Now move on to the next lesson as we further explore the aspect of Inheritance.

# Thanks, and have fun!

To order supplementary materials for all the lessons in this series, including lesson examples, lab solutions, quizzes, tests, and unit reviews, visit the O(N)CS Lessons website, or contact me at

John B. Owen
captainjbo@gmail.com