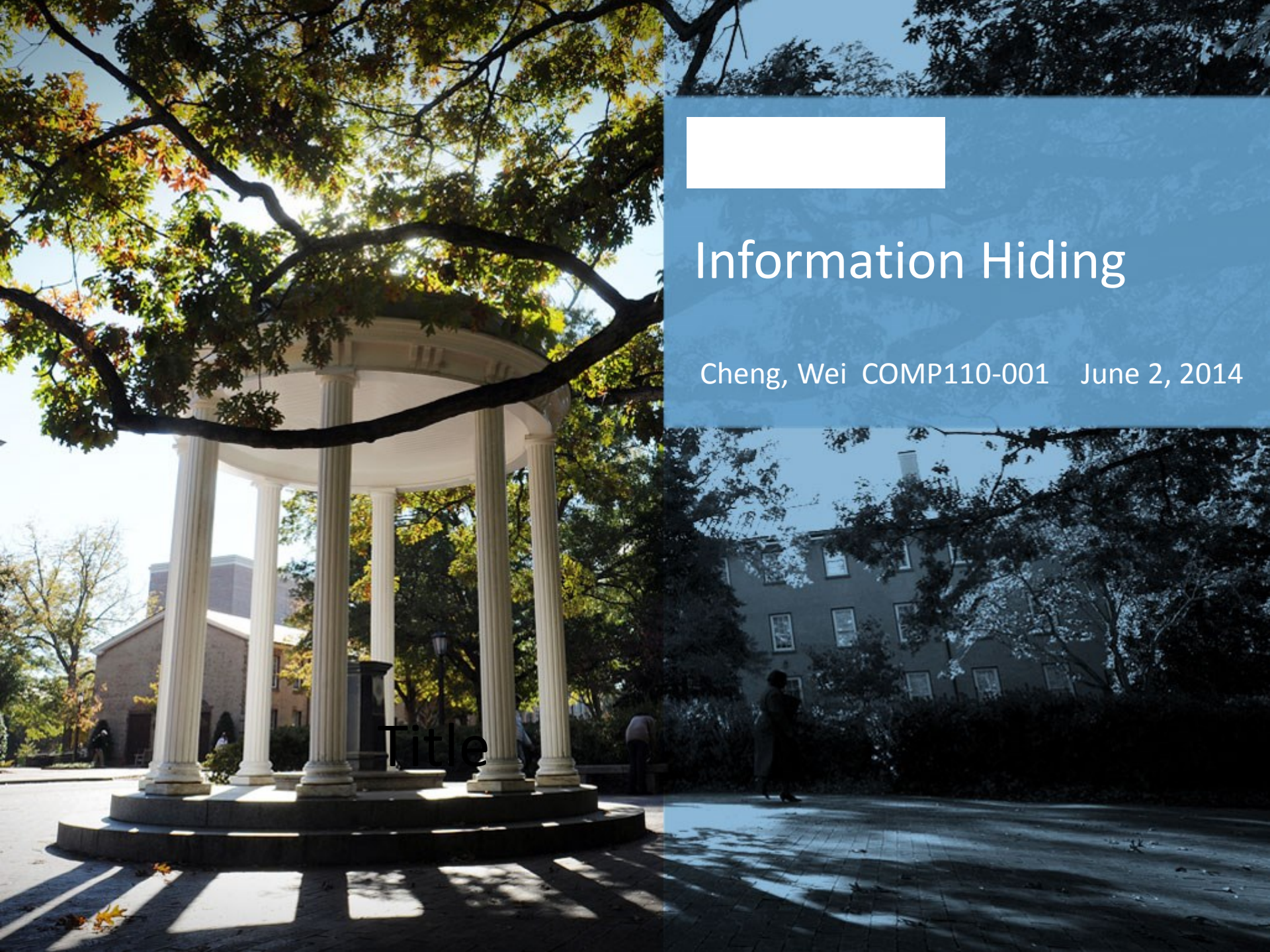
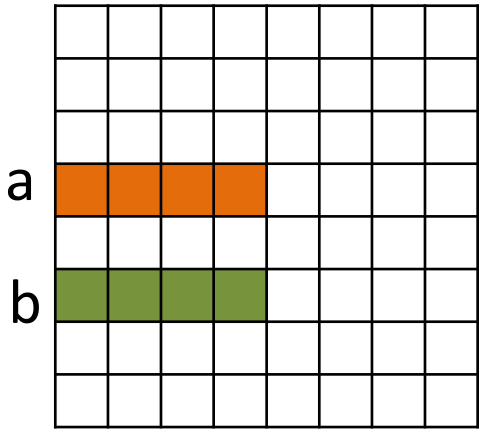


Information Hiding

Cheng, Wei COMP110-001 June 2, 2014

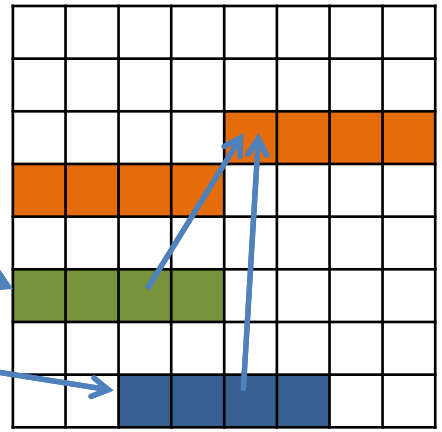
Variable of Primitive Types

```
int a = 10;
int b = a;


b = b + 1;


System.out.println( a );
```



Variable of Class Types

```
Student anna = new Student();
anna.PID = 1234;
anna.year = 3;

Student a_copy = anna;
a_copy.year = 4;

System.out.println( anna.year );
```

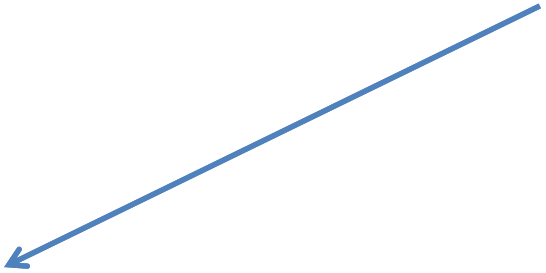


Pass-by-Value

- When a method with parameter of primitive type is called?

```
public void increaseByOne( int num ) {
      num = num + 1;
}

public void doSth () {
      int someNum = -2;
      increaseByOne( someNum );
      System.out.println( someNum );
}
```

What do you get?

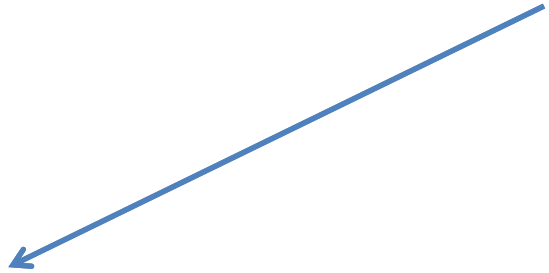


Pass-by-Value

- When a method with parameter of Class type is called?

```
public void increaseByOne( Student s) {
      s.year = s.year + 1;
}

public void doSth () {
      Student anna = new Student();
      anna.PID = 1234;
      anna.year = 3;
      increaseByOne(  anna );
      System.out.println( anna.year );
}
```

What do you get?



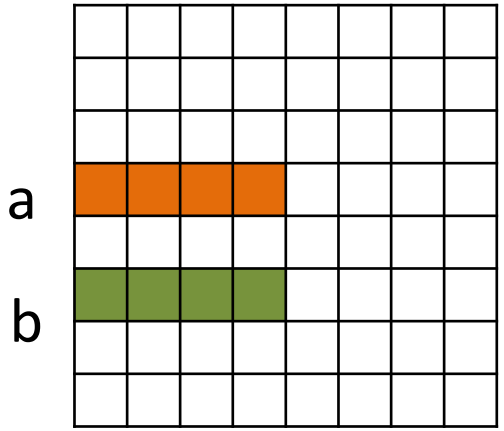
== vs .equals()

== is a built-in operator to compare the values directly associated to variables:

- **primitive type: the value stored in variable**
- class type: the address

```
int a = 10;
int b = a;
```

a == b? (orange vs green)



== vs .equals()

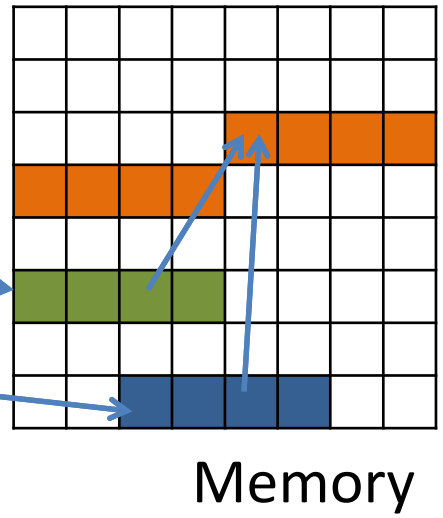
== is a built-in operator to compare the values directly associated to variables:

- primitive type: the value stored in variable
- **class type: the address**

Student anna = new Student();

Student a_copy = anna;

anna == a_copy? (green vs blue)



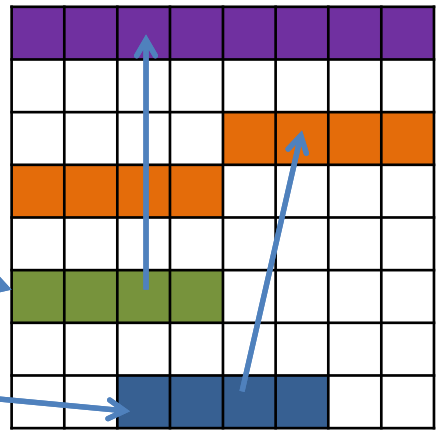
== vs .equals()

== tests whether two class variables are pointing to the same object location in memory. It does not examine object content.

```
Student anna = new Student();
anna.pid = 1234;

Student a_copy = new Student();
a_copy.pid = 1234;
```

anna == a_copy ? (green vs blue)

anna.equals(a_copy) ? (we want Purple vs Orange)



Memory

== vs .equals()

- The == operator does the same thing for all classes.
- The .equals() is class-specific (depends on how your implement it).
- For Student class, we can do:

```
class Student {
      int PID;
      int year;
      public boolean equals( Student s ) {
            return this.PID == s.PID; // assume that PID is unique
      }
}
```

Another Example

This is how .equals() is implemented in Java String class:

```
1012        public boolean equals(Object anObject) {
1013            if (this == anObject) {
1014                return true;
1015            }
1016            if (anObject instanceof String) {
1017                String anotherString = (String)anObject;
1018                int n = count;
1019                if (n == anotherString.count) {
1020                    char v1[] = value;
1021                    char v2[] = anotherString.value;
1022                    int i = offset;
1023                    int j = anotherString.offset;
1024                    while (n-- != 0) {
1025                        if (v1[i++] != v2[j++])
1026                            return false;
1027                    }
1028                    return true;
1029                }
1030            }
1031            return false;
1032        }
```

Summary

For primitive types, use == to test equality

For class types, == only tests equality of references. Generally, in this course, you should not use it. To examine content, use the .equals() method provided in class.

- String.equals() compares “this” string and an input string character by character

When you write your own class, you should consider how to implement a .equals() method if equality test is needed.

this

- Within a class definition, this is a name for the current receiving object
  - `this.age`
  - `this.major`
  - `this.getAge()`
- Frequently omitted, but understood to be there
- See book for details

Example

```
class Student {
        int PID;
        int year;

        public void setPID( int PID  ) {
                this.PID = PID;
        }
}
```

Information Hiding

- Software:
  - usually efforts of many engineers
  - Divided into multiple components
  - Each component interacts with other components
  - Each component has its internal data/logic that are not supposed to be visible to outside
  - We will see examples later

Information Hiding

Design a method so that it can be used without any need to understand the fine detail of the code is called information hiding.

```
/**
    Precondition: The instance variables of the calling
    object have values.
    Postcondition: The data stored in (the instance variables
    of) the receiving object have been written to the screen.
*/
public void writeOutput()

/**
    Precondition: years is a nonnegative number.
    Postcondition: Returns the projected population of the
    receiving object
    after the specified number of years.
*/
public int predictPopulation(int years)
```

Access Control Modifiers

public : attributes/methods that can be used (invoked) by any other classes without restriction

--- object interaction is done through these attributes/methods

protected: not covered in this course

default (no modifier): covered later

private: attributes/methods that is only available within the class (i.e., cannot be invoked from outside)

Example

```
public class Student
{
    public int classYear;
    private String major;
}

Student anna = new Student();

anna.classYear = 1;

anna.major = “Computer Science”;
```

OK, *classYear* is *public*

Error!!! *major* is *private*

Generally, Instance variables should be private

- Force users of the class to access instance variables only through methods
  - Gives you control of how programmers use your class
  - Embed logic in accessing variables
- Exceptions

Information Hiding

- Example: consider the two following designs

```
class Student {
    public int PID;
    public int year;
    ....
}
```

```
class Student {
    private int PID;
    private int year;

    public void setPID( int PID ) {
        ....
    }

    public int getPID() {
        ....
    }
}
```

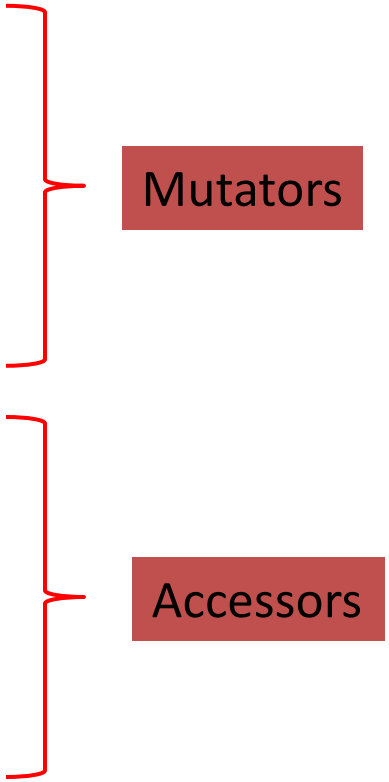
Which is better?

Accessors and mutators

- How do you access private instance variables?
- Accessor methods (a.k.a. get methods, getters)
  - Allow you to look at data in private instance variables
- Mutator methods (a.k.a. set methods, setters)
  - Allow you to change data in private instance variables

Example: Person

```
public class Person
{
    private String name;
    private int age;

    public void setName(String name)
    {
        this.name = name;
    }

    public void setAge(int age)
    {
        this.age = age;
    }

    public String getName()
    {
        return this.name;
    }

    public int getAge()
    {
        return this.age;
    }
}
```



Okay, but why make methods private?

- Helper methods that will only be used from inside a class should be private
  - External users have no need to call these methods

- **Encapsulation**

Example

Not directly accessible to outside

Only used internally

```
class Student {
        private int PID;
        private int year;

        private boolean checkPID( int newPID ) {
                …
        }

        public void setPID( int PID  ) {
                if (checkPID( PID) ) this.PID = PID;
        }
}
```

Example: driving a car

- Accelerate with the accelerator pedal
- Decelerate with the brake pedal
- Steer with the steering wheel
- Does not matter if:
  - You are driving a gasoline engine car or a hybrid engine car
  - You have a 4-cylinder engine or a 6-cylinder engine
- You still drive the same way

Encapsulation

- The *interface* is the same
- The underlying *implementation* may be different

Encapsulation in classes

- A *class interface* tells programmers all they need to know to use the class in a program
- The *implementation* of a class consists of the private elements of the class definition
  - private instance variables and constants
  - private methods
  - bodies of public methods

Example: two implementations of Rectangle

```
public class Rectangle
{
    private int width;
    private int height;
    private int area;

    public void setDimensions(
        int newWidth,
        int newHeight)
    {
        width = newWidth;
        height = newHeight;
        area = width * height;
    }

    public int getArea()
    {
        return area;
    }
}
```

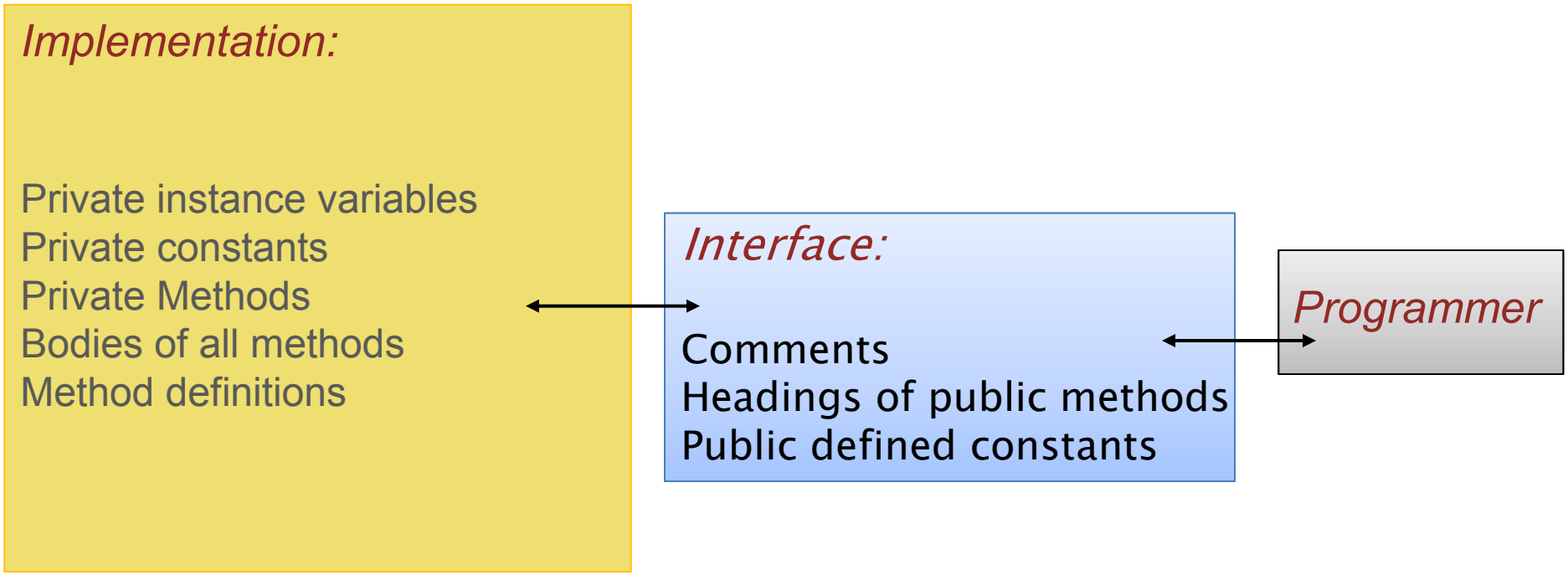
```
public class Rectangle
{
    private int width;
    private int height;

    public void setDimensions(
        int newWidth,
        int newHeight)
    {
        width = newWidth;
        height = newHeight;
    }

    public int getArea()
    {
        return width * height;
    }
}
```

Encapsulation

- Implementation should not affect behavior described by interface
  - Two classes can have the same behavior but different implementations

Well encapsulated

Imagine a wall between interface and implementation



Next Class

- Constructors and Static Methods
- Read Section 6.1-6.2