Object-Oriented Programming

CSCI-UA 0470-001
Class 15
Instructor: Randy Shepherd

Solution to in-class exercise

Questions

- *Why is the implementation of the constructor for Array and its __vtable initialization call in the header file?*

```
template <typename T>
struct Array;

template <typename T>
struct Array_VT;

template <typename T>
struct Array {
  Array_VT<T>* __vptr;
  const int32_t length;
  T* __data;

  // The constructor (defined inline).
  Array(const int32_t length)
    : __vptr(&__vtable), length(length), __data(new T[length]
  }

  // The function returning the class object for the array.
  static java::lang::Class __class();

  // The vtable for the array.
  static Array_VT<T> __vtable;
};

// The vtable for arrays.
template <typename T>
Array_VT<T> Array<T>::__vtable;
```

Questions

- *Why is the implementation of the constructor for Array and its __vtable initialization call in the header file?*

- A template is not a class or a function. A template is a “pattern” that the compiler uses to generate a family of classes or functions. Therefore its not an implementation.

- More here..
  https://isocpp.org/wiki/faq/templates#templates-defn-vs-decl

```
template <typename T>
struct Array;

template <typename T>
struct Array_VT;

template <typename T>
struct Array {
  Array_VT<T>* __vptr;
  const int32_t length;
  T* __data;

  // The constructor (defined inline).
  Array(const int32_t length)
    : __vptr(&__vtable), length(length), __data(new T[length]
  }

  // The function returning the class object for the array.
  static java::lang::Class __class();

  // The vtable for the array.
  static Array_VT<T> __vtable;
};

// The vtable for arrays.
template <typename T>
Array_VT<T> Array<T>::__vtable;
```

Questions

- *Why has the 'component' property at line 6 and the 'isArray' at line 20 method been added?*

```
1  struct __Class
2  {
3      __Class_VT* __vptr;
4      String name;
5      Class parent;
6      Class component;
7      bool primitive;
8
9      // The constructor.
10     __Class(String name,
11             Class parent,
12             Class component = (Class)__rt::null(),
13             bool primitive = false);
14
15     // The instance methods of java.lang.Class.
16     static String toString(Class);
17     static String getName(Class);
18     static Class getSuperclass(Class);
19     static bool isPrimitive(Class);
20     static bool isArray(Class);
21     static Class getComponentType(Class);
22     static bool isInstance(Class, Object);
23
24     // The function returning the class object representing
25     // java.lang.Class.
26     static Class __class();
27
28     // The vtable for java.lang.Class.
29     static __Class_VT __vtable;
30 };
```

Questions

- Why has the 'component' property at line 6 and the 'isArray' at line 20 method been added?

- Reminder: <u>covariance:</u> for an array with component type S, if S extends T, then S[] extends T[]

```
1  struct __Class
2  {
3      __Class_VT* __vptr;
4      String name;
5      Class parent;
6      Class component;
7      bool primitive;
8
9      // The constructor.
10     __Class(String name,
11             Class parent,
12             Class component = (Class)__rt::null(),
13             bool primitive = false);
14
15     // The instance methods of java.lang.Class.
16     static String toString(Class);
17     static String getName(Class);
18     static Class getSuperclass(Class);
19     static bool isPrimitive(Class);
20     static bool isArray(Class);
21     static Class getComponentType(Class);
22     static bool isInstance(Class, Object);
23
24     // The function returning the class object representing
25     // java.lang.Class.
26     static Class __class();
27
28     // The vtable for java.lang.Class.
29     static __Class_VT __vtable;
30 };
```

Questions

- Why has the 'component' property at line 6 and the 'isArray' at line 20 method been added?

- Since arrays in Java are covariant, we need to know the type contained in the array to determine if some array is a subclass of some other array.

- isArray exists as a convenience so that we easily recognize array types for special casing logic for them.

```
1  struct __Class
2  {
3      __Class_VT* __vptr;
4      String name;
5      Class parent;
6      Class component;
7      bool primitive;
8
9      // The constructor.
10     __Class(String name,
11             Class parent,
12             Class component = (Class)__rt::null(),
13             bool primitive = false);
14
15     // The instance methods of java.lang.Class.
16     static String toString(Class);
17     static String getName(Class);
18     static Class getSuperclass(Class);
19     static bool isPrimitive(Class);
20     static bool isArray(Class);
21     static Class getComponentType(Class);
22     static bool isInstance(Class, Object);
23
24     // The function returning the class object representing
25     // java.lang.Class.
26     static Class __class();
27
28     // The vtable for java.lang.Class.
29     static __Class_VT __vtable;
30 };
```

Questions

- *Describe in just a few words what the checkStore function does.*

```
template <typename T, typename U>
void checkStore(Array<T>* array, U object)
{
    if (null() != (java::lang::Object) object)
    {
        java::lang::Class t1 = array->__vptr->getClass(array);
        java::lang::Class t2 = t1->__vptr->getComponentType(t1);

        if (! t2->__vptr->isInstance(t2, (java::lang::Object)object))
        {
            throw java::lang::ArrayStoreException();
        }
    }
}
```

Questions

- *Describe in just a few words what the checkStore function does.*
- checkStore checks object of some type U is a legal type to be inserted into the array and if not throws an exception.

```
template <typename T, typename U>
void checkStore(Array<T>* array, U object)
{
    if (null() != (java::lang::Object) object)
    {
        java::lang::Class t1 = array->__vptr->getClass(array);
        java::lang::Class t2 = t1->__vptr->getComponentType(t1);

        if (! t2->__vptr->isInstance(t2, (java::lang::Object)object))
        {
            throw java::lang::ArrayStoreException();
        }
    }
}
```

Questions

- *Describe in just a few words what the checkStore function does.*

- checkStore checks object of some type U is a legal type to be inserted into the array and if not throws an exception.

- What is the definition of a legal type?

```
template <typename T, typename U>
void checkStore(Array<T>* array, U object)
{
    if (null() != (java::lang::Object) object)
    {
        java::lang::Class t1 = array->__vptr->getClass(array);
        java::lang::Class t2 = t1->__vptr->getComponentType(t1);

        if (! t2->__vptr->isInstance(t2, (java::lang::Object)object))
        {
            throw java::lang::ArrayStoreException();
        }
    }
}
```

Questions

- *Note the comment on line 10 of output/java_lang.cpp, what needs to happen there?*

```
1  bool __Class::isInstance(Class __this, Object o)
2  {
3      Class k = o->__vptr->getClass(o);
4  
5      do
6      {
7          if (__this->__vptr->equals(__this, (Object) k))
8            return true;
9  
10         // TODO: handle covariance of arrays
11 
12         k = k->__vptr->getSuperclass(k);
13     }
14     while ((Class) __rt::null() != k);
15 
16     return false;
17 }
```

Questions

- *Note the comment on line 10 of output/java_lang.cpp, what needs to happen there?*
- *Check if the component of the array types have a super/sub-class relationship, since for the component type, if S extends T, then S[] extends T[] and since arrays extend Object.*
- *I.e. an Array<String> is an instance of Array<Object>*

```
1  bool __Class::isInstance(Class __this, Object o)
2  {
3      Class k = o->__vptr->getClass(o);
4
5      do
6      {
7          if (__this->__vptr->equals(__this, (Object) k))
8            return true;
9
10         // TODO: handle covariance of arrays
11
12         k = k->__vptr->getSuperclass(k);
13     }
14     while ((Class) __rt::null() != k);
15
16     return false;
17 }
```

Questions

- *Briefly explain this code.*

```
// Template specialization for arrays of ints.
template<>
java::lang::Class Array<int32_t>::__class() {
  static java::lang::Class ik =
    new java::lang::__Class(
       __rt::literal("int"),
       (java::lang::Class)__rt::null(),
       (java::lang::Class)__rt::null(),
       true
    );

  static java::lang::Class k =
    new java::lang::__Class(
      literal("[I"),
      java::lang::__Object::__class(),
      ik);

  return k;
}
```

Questions

- *Briefly explain this code.*

- *This is the template specialization for arrays of int. Note that there is superclass for the component since ints don't extend Object.*

- *The component type must be constructed ad-hoc, since we don't have a java::lang::int*

```
template<>
java::lang::Class Array<int32_t>::__class() {
  static java::lang::Class ik =
    new java::lang::__Class(
       __rt::literal("int"),
       (java::lang::Class)__rt::null(),
       (java::lang::Class)__rt::null(),
       true
    );

  static java::lang::Class k =
    new java::lang::__Class(
      literal("[I"),
      java::lang::__Object::__class(),
      ik);

  return k;
}
```

Questions

- How will you add primitive array specializations?
  - Implement 'by-hand' specializations into java_lang.cpp
- How could you add specializations for objects of arbitrary types in your translator?
  - The translator will have to be smart enough to recognize arrays of types that are being translated and generate a specialization for it, as you did by hand for A in this exercise.

```
namespace inputs
{
  namespace javalang
  {
    __A::__A() : __vptr(&__vtable) {}

    String __A::toString(A __this) {
      return new __String("A");
    }

    Class __A::__class() {
      static Class k =
        new __Class(__rt::literal("inputs.javalang.A"), (Class) __rt::null());
      return k;
    }

    __A_VT __A::__vtable;
  }
}

namespace __rt
{
  // Template specialization for arrays of A
  template<>
  java::lang::Class Array<inputs::javalang::A>::__class()
  {
    static java::lang::Class k =
        new java::lang::__Class(literal("[Linputs.javalang.A;"),
                                java::lang::__Object::__class(),
                                inputs::javalang::__A::__class());
    return k;
  }
}
```

```java
public static void main(String[] args) {
 int[] ints = new int[2];
 System.out.println(ints[1]);

 float[] floats = new float[2];
 System.out.println(floats[1]);

 A[] array = new A[5];

 for (int i = 0; i < array.length; ++i) {
     A a = new A();
     array[i] = a;
 }

 for (int i = 0; i < array.length; ++i) {
     A a = array[i];
     System.out.println(a.toString());
 }

 try {
     System.out.println(array[128]);
 } catch (ArrayIndexOutOfBoundsException e) {
     System.out.println("Caught ArrayIndexOutOfBoundsException");
 }

 try {
     Object[] o = array;
     o[2] = new Object();
 } catch (ArrayStoreException e) {
     System.out.println("Caught ArrayStoreException");
 }
}
```

```cpp
int main(void)
{
  __rt::Array<int32_t>* ints = new __rt::Array<int32_t>(2);
  __rt::checkNotNull(ints);
  __rt::checkIndex(ints, 1);
  cout << ints->__data[1] << endl;

  __rt::Array<float>* floats = new __rt::Array<float>(2);
  __rt::checkNotNull(floats);
  __rt::checkIndex(floats, 1);
  cout << fixed << floats->__data[1] << endl;

  __rt::Array<A>* array = new __rt::Array<A>(5);
  for(int i = 0 ; i < array->length ; ++i) {
     A a = new __A();
     __rt::checkNotNull(array);
     __rt::checkIndex(array, i);
     __rt::checkStore(array, a);
     array->__data[i] = a;
  }

  for(int i = 0 ; i < array->length ; ++i) {
    __rt::checkNotNull(array);
    __rt::checkIndex(array, i);
    A a = array->__data[i];
    cout << a->toString(a)->data << endl;
  }

  try {
    __rt::checkNotNull(array);
    __rt::checkIndex(array, 128);
    std::cout << array->__data[128] << std::endl;
  } catch (const ArrayIndexOutOfBoundsException& ex) {
    std::cout << "Caught ArrayIndexOutOfBoundsException"<< std::endl;
  }

  try {
    __rt::Array<Object>* o = (__rt::Array<Object>*) array;
    String s = __rt::literal("Hello");
    __rt::checkNotNull(o);
    __rt::checkIndex(o, 2);
    __rt::checkStore(o, s);
  } catch (const ArrayStoreException& ex) {
    std::cout << "Caught ArrayStoreException"<< std::endl;
  }

  return 0;
}
```

Inheritance is terrible..

Inheritance is terrible..

(sometimes)

- *Class* inheritance is a powerful way to achieve code reuse, but not always best tool for job.
- It is safe to use class inheritance when extending classes specifically designed for it.
- Inheriting from ordinary, concrete classes is often a bad approach.

Advantages of Class Inheritance

- Substitution principle.
- Easy code reuse, just inherit.
- Easy to modify or extend the implementation being reused.

Disadvantages of Class Inheritance

- *Complicated inheritance hierarchies* can lead to code which is difficult to read and maintain.

    - Undisciplined, poorly applied subtype polymorphism leads to a large gap between reading the static code and what actually happens at runtime.

    - Bugs become very difficult to find (and sometimes recreate). Mental gymnastics required to debug.

Disadvantages of Class Inheritance

- *Breaks encapsulation*; subclass exposed to implementation details of superclass.
  - Known as “white-box” reuse. Good abstractions are “black-box”.
  - Subclasses may have to be changed if superclass is changed.
  - Tight coupling between super and subclasses.

Breaking Encapsulation

- Best understood through example:
  - Suppose we have a Java program that uses the java.util.HashSet data structure. (Which is an implementation of a Set).
  - In order to tune performance, we want to monitor the number of attempted element insertions.
  - Note that this is different than simply querying the HashSet for size, we want to consider any removed elements in our count or duplicates.

Adapted from Joshua Bloch's book "Effective Java"

Breaking Encapsulation

- In our first attempt, we write an extension to HashSet that keeps count of the insertion attempts.
- HashSet contains two methods capable of adding elements, `add` and `addAll`
- These two methods are overridden.

Adapted from Joshua Bloch's book "Effective Java"

InstrumentedHashSet

- This looks pretty reasonable, right?

```java
public class InstrumentedHashSet extends HashSet {
  // The number of attempted element insertions
  private int addCount = 0;

  public InstrumentedHashSet() { super(); }

  public InstrumentedHashSet(Collection c) { super(c); }

  public InstrumentedHashSet(int initCap, float loadFactor) {
    super(initCap, loadFactor);
  }

  public boolean add(Object o) {
    addCount++;
    return super.add(o);
  }

  public boolean addAll(Collection c) {
    addCount += c.size();
    return super.addAll(c);
  }

  public int getAddCount() {
    return addCount;
  }
}
```

Adapted from Joshua Bloch's book "Effective Java"

InstrumentedHashSet

- This looks pretty reasonable, right?
- Suppose we do this…

```
HashSet hs = new InstrumentedHashSet();
hs.addAll(new String[] ("1", "2", "3"))
System.out.println(hs.getAddCount());
```

- What should we *expect* as output?

```
public class InstrumentedHashSet extends HashSet {
  // The number of attempted element insertions
  private int addCount = 0;

  public InstrumentedHashSet() { super(); }

  public InstrumentedHashSet(Collection c) { super(c); }

  public InstrumentedHashSet(int initCap, float loadFactor) {
    super(initCap, loadFactor);
  }

  public boolean add(Object o) {
    addCount++;
    return super.add(o);
  }

  public boolean addAll(Collection c) {
    addCount += c.size();
    return super.addAll(c);
  }

  public int getAddCount() {
    return addCount;
  }
}
```

Adapted from Joshua Bloch's book "Effective Java"

InstrumentedHashSet

- This looks pretty reasonable, right?
- Suppose we do this…

```
HashSet hs = new InstrumentedHashSet();
hs.addAll(new String[] ("1", "2", "3"))
System.out.println(hs.getAddCount());
```

- What should we *expect* as output?
- 3 right? Well, it prints 6.

```
public class InstrumentedHashSet extends HashSet {
  // The number of attempted element insertions
  private int addCount = 0;

  public InstrumentedHashSet() { super(); }

  public InstrumentedHashSet(Collection c) { super(c); }

  public InstrumentedHashSet(int initCap, float loadFactor) {
    super(initCap, loadFactor);
  }

  public boolean add(Object o) {
    addCount++;
    return super.add(o);
  }

  public boolean addAll(Collection c) {
    addCount += c.size();
    return super.addAll(c);
  }

  public int getAddCount() {
    return addCount;
  }
}
```

Adapted from Joshua Bloch's book "Effective Java"

InstrumentedHashSet

- The problem is that the `addAll` method of the class HashSet is implemented by calling the `add` method for each element in the Collection argument.

- Since add is overridden by the subclass every inserted element is counted twice.

```
public class InstrumentedHashSet extends HashSet {
  // The number of attempted element insertions
  private int addCount = 0;

  public InstrumentedHashSet() { super(); }

  public InstrumentedHashSet(Collection c) { super(c); }

  public InstrumentedHashSet(int initCap, float loadFactor) {
    super(initCap, loadFactor);
  }

  public boolean add(Object o) {
    addCount++;
    return super.add(o);
  }

  public boolean addAll(Collection c) {
    addCount += c.size();
    return super.addAll(c);
  }

  public int getAddCount() {
    return addCount;
  }
}
```

Adapted from Joshua Bloch's book "Effective Java"

“Fixing” the Problem

- We could “fix” this by removing our implementation of *addAll.*
- While that would work, our data structure’s correct functioning would depend on the implementation of its superclass.
- There is no guarantee that this implementation detail would hold true in future versions of Java.
- Encapsulation is broken.

Adapted from Joshua Bloch's book "Effective Java"

“Fixing” the Problem

- We could again try to “fix” this by implementing our own version of `addAll`, where we iterate over the Collection ourselves and add each element.

- This is slightly better, but also has problems.

- We are defeating a primary purpose of inheritance (code reuse) possibly introducing bugs.

- And this is not always possible, since data required for this technique, in some cases may be private.

Adapted from Joshua Bloch's book "Effective Java"

“Fixing” the Problem

- This problem stems from method overriding.
- Ok, lets not override methods then. This sounds safer. (It is in fact, to a degree).
- So we add a `addWithInsertionCount` method that returns an *int*.
- However, what if a future version of HashSet has its own `addWithInsertionCount` method… and it returns a `long`?

Adapted from Joshua Bloch's book "Effective Java"

Fixing the Problem

- Luckily there is a solution and it is a design technique with wide applicability.
- Wrap HashSet inside a new class that delegates all work to the HashSet and its only logic is only counts the number of attempted insertions.
- This approach is called ‘*composition*’.
- By using composition we can make the wrapper class independent of the concrete implementation of HashSet.

Adapted from Joshua Bloch's book "Effective Java"

Composition

- This technique is called 'composition' because the existing class becomes a *component* of the new class.

- Each instance method method in the new class either 'decorates' or 'forwards to' an existing Set implementation.

- Note how we take an implementation of the Set interface as an argument at construction.

```java
public class InstrumentedSet implements Set {
  private final Set s;
  private int addCount = 0;

  public InstrumentedSet(Set s) { this.s = s; }

  public boolean add(Object o) {
    addCount++;
    return s.add(o);
  }

  public boolean addAll(Collection c) {
    addCount += c.size();
    return s.addAll(c);
  }

  public int getAddCount() { return addCount; }

  public void clear() {  s.clear(); }

  // Other forwarding methods for the remaining
  // methods of the Set interface
  // ...
}
```

Adapted from Joshua Bloch's book "Effective Java"

Composition

- This approach is extremely flexible.
- Unlike the inheritance-based approach, our new class can work for any Set implementation.
- We are completely decoupled from the set implementation. Encapsulation is preserved.
- This is known as the *'Decorator Pattern'*

```
public class InstrumentedSet implements Set {
  private final Set s;
  private int addCount = 0;

  public InstrumentedSet(Set s) { this.s = s; }

  public boolean add(Object o) {
    addCount++;
    return s.add(o);
  }

  public boolean addAll(Collection c) {
    addCount += c.size();
    return s.addAll(c);
  }

  public int getAddCount() { return addCount; }

  public void clear() {  s.clear(); }

  // Other forwarding methods for the remaining
  // methods of the Set interface
  // ...
}
```

Adapted from Joshua Bloch's book "Effective Java"

Another Disadvantage of Class Inheritance

- *Implementations* inherited from superclasses *can not be changed at run-time*.
    - To change the way a subclass behaves you must modify the source code.

Implementations Fixed at Runtime

- Another example is necessary…
    - Suppose we want to model the business logic of an airline.
    - One component of the software handles transactions with passengers such as ticket reservation and purchase. Another component handles payroll.
    - The ticketing component involves *passengers* and *agents*, while the payroll component involves only *agents*.
    - The ticketing component needs to keep track of the name and address of each *passenger*. The payroll component needs to maintain the *same information* for each *agent*.

Implementations Fixed at Runtime

- A design for the two components might involve the following class hierarchy:
  - A class `Person` is used to store the name and address of each person.
  - There are two classes, `Passenger` and `Agent`, to model agents and passengers, respectively.
  - Both of these classes are subclasses of `Person`.

Person Hierarchy

- What's wrong with this design?

```java
public abstract class Person {
  private String name;
  private Address address;
  //.. Accessors, etc.
}

public class Agent extends Person {
  // Agent-related methods...
}

public class Passenger extends Person {
  // Passenger-related methods...
}
```

Person Hierarchy

- What's wrong with this design?
- The roles of a person may change over time.
- A person may be a passenger at one point in time and an agent at a another point in time, or even both at the same time.
- This is not reflected in the static class hierarchy.

```java
public abstract class Person {
  private String name;
  private Address address;
  //.. Accessors, etc.
}

public class Agent extends Person {
  // Agent-related methods...
}

public class Passenger extends Person {
  // Passenger-related methods...
}
```

Fixing the problem

- One viable solution is to use subclassing and composition together:
    - Introduce a new class PersonRole that has a reference to a Person.
    - Define Passenger and Agent as subclasses of PersonRole.

```java
public abstract class PersonRole {
  protected Person p;
  // Role-related methods...
}

public class Agent extends PersonRole {
  // Agent-related methods...
}

public class Passenger extends PersonRole {
  // Passenger-related methods...
}

public class Person {
  private String name;
  private Address address;
  //.. Accessors, etc.
}
```

Advantages of Composition

- Contained objects are accessed by the containing class solely through their interfaces.
- "Black-box" reuse, since internal details of contained objects are not visible.
- Good encapsulation.
- Fewer implementation dependencies.
- Each class is focused on just one task.
- The composition can be defined dynamically at run-time through objects acquiring references to other objects of the some type.

Disadvantages of Composition

- Resulting systems tend to have more objects.
    - Though the JVM is good at dealing w/ short-lived objects.
- Lots of ”boiler plate" code for forwarding methods that delegate work to the contained objects.
    - Nothing new in Java.
- Interfaces must be carefully defined in order to use many different objects as composition blocks.
    - This is where you get into trouble.

So when should one use class inheritance?

- **Coad's Rules**: Use class inheritance when the following criteria are satisfied:

    1. A subclass expresses *"is a special kind of"* and not *"is a role played by a"* (and also not *"has a"*).

    2. An instance of a subclass never needs to become an object of another class.

    3. A subclass extends, rather than overrides or nullifies, the responsibilities of its superclass.

    4. A subclass does not extend the capabilities of what is merely a utility class.

    5. For a class in the problem domain, the subclass specializes a role, transaction, or device.

And when you do use it..

- Beware the pitfalls!
- The relationships between objects in the problem domain do not always carry over to an OO design.
- For example, suppose we want to model geometric objects such as rectangles, squares, circles, etc.
- Since a square is a special kind of a rectangle, we might want to implement squares as a subclass of Rectangle:

False Hierarchies

```java
public class Rectangle {
  protected double width;
  protected double height;

  public Rectangle(double w, double h) {
    width = w;
    height = h;
  }

  public double getWidth() { return width; }
  public double getHeight() { return height; }

  public void setWidth(double w) { width = w; }
  public void setHeight(double h) { height = h; }

  public double area() {return (width * height); }
}
```

```java
public class Square extends Rectangle {
  public Square(double s) { super(s, s); }

  // ...

  public void setHeight(double h) {
    height = h;
    width = h;
  }

  public void setWidth(double w) {
    height = w;
    width = w;
  }

  //...
}
```

- Note that we override the setHeight and setWidth to maintain the invariant that the height and width of a square coincide.

False Hierarchies

```java
public class Rectangle {
  protected double width;
  protected double height;

  public Rectangle(double w, double h) {
    width = w;
    height = h;
  }

  public double getWidth() { return width; }
  public double getHeight() { return height; }

  public void setWidth(double w) { width = w; }
  public void setHeight(double h) { height = h; }

  public double area() {return (width * height); }
}
```

```java
public class Square extends Rectangle {
  public Square(double s) { super(s, s); }

  // ...

  public void setHeight(double h) {
    height = h;
    width = h;
  }

  public void setWidth(double w) {
    height = w;
    width = w;
  }

  //...
}
```

- A Square object does not behave like a Rectangle object because the methods setHeight and setWidth of class Square modify both height and width.

False Hierarchies

```java
public class Rectangle {
  protected double width;
  protected double height;

  public Rectangle(double w, double h) {
    width = w;
    height = h;
  }

  public double getWidth() { return width; }
  public double getHeight() { return height; }

  public void setWidth(double w) { width = w; }
  public void setHeight(double h) { height = h; }

  public double area() {return (width * height); }
}
```

```java
public class Square extends Rectangle {
  public Square(double s) { super(s, s); }

  // ...

  public void setHeight(double h) {
    height = h;
    width = h;
  }

  public void setWidth(double w) {
    height = w;
    width = w;
  }

  //...
}
```

- We cannot safely use a Square object when a Rectangle object is expected, even though the subclass relationship suggests otherwise. This is a violation of the *substitution principle*.

Final note…

- Everything we have talked about today relates to *class* inheritance. What's known as "implementation inheritance".

- There is another kind of inheritance that has fewer disadvantages, called "definition inheritance".

- Have you heard of interfaces? That is an example. We will get to that later in the semester.