# M7 (b) - Inheritance

Jin L.C. Guo

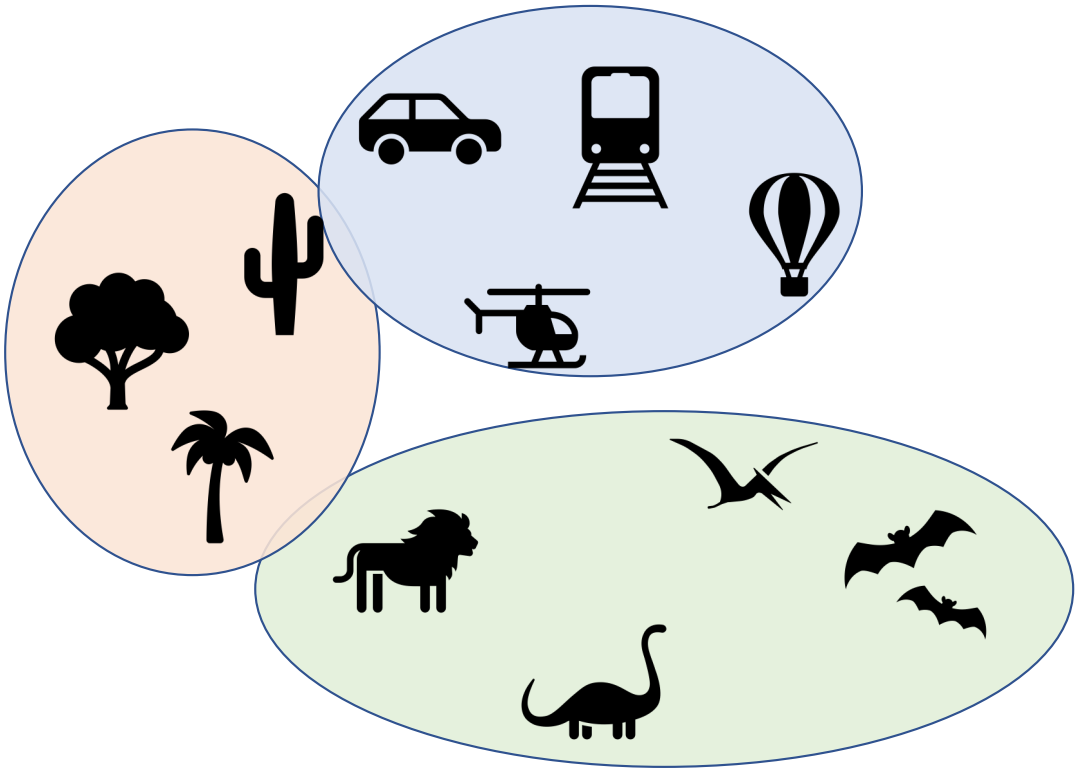
*Image source https://cdn.pixabay.com/photo/2015/01/11/21/30/cats-596782_1280.jpg*

# Objective

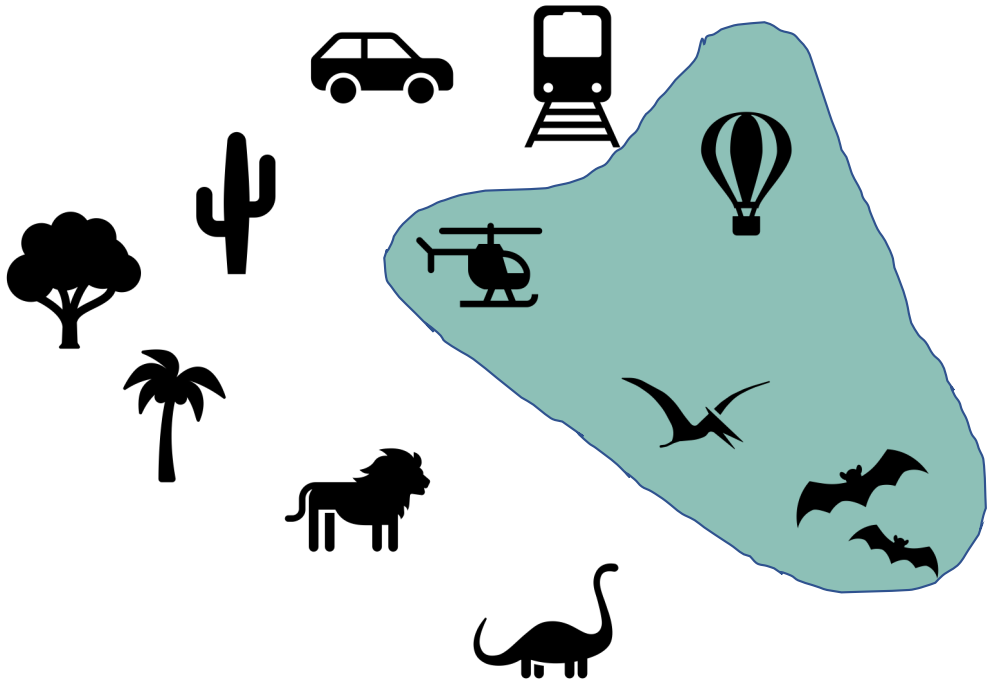
- Common problems/considerations of inheritance
- Liskov Substitution Principle

# Abstract Class vs Interface

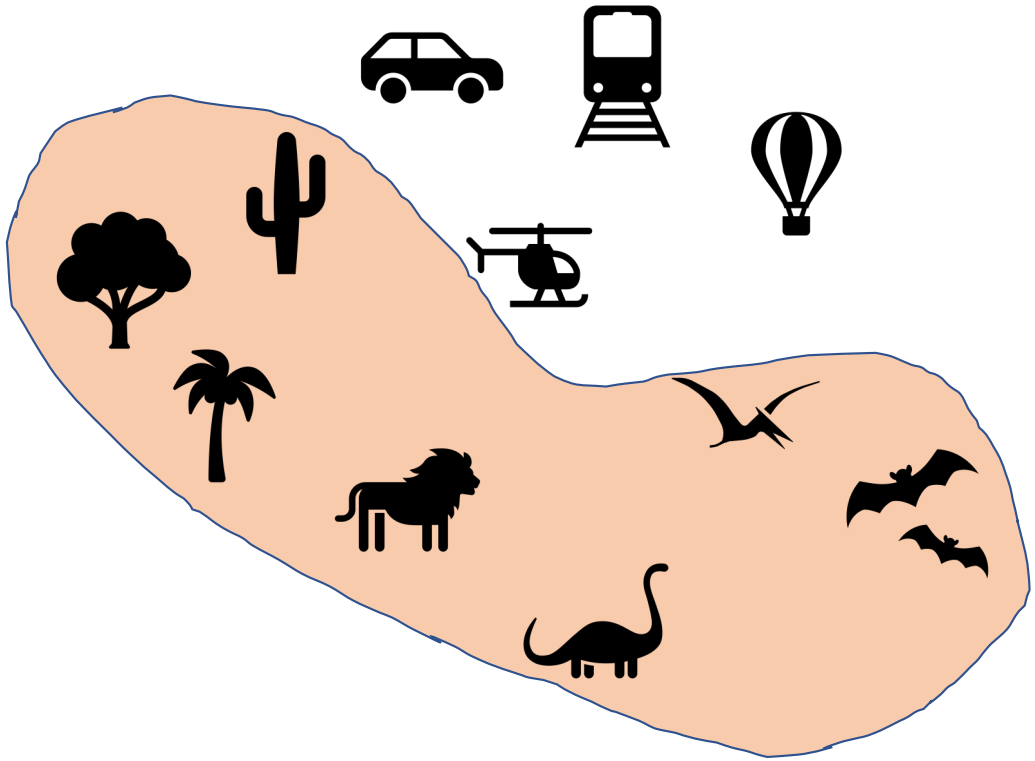
# Abstract Class vs Interface



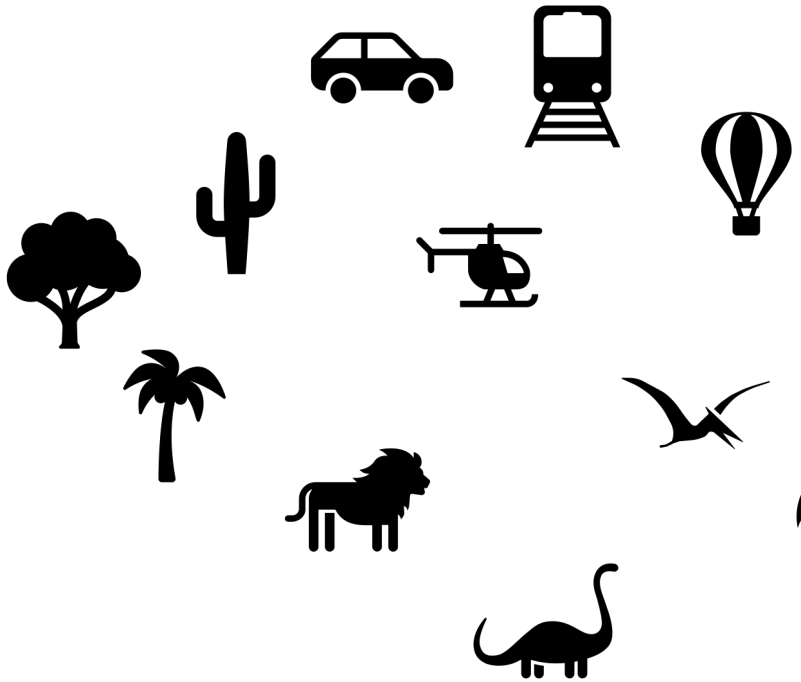
# Abstract Class vs Interface

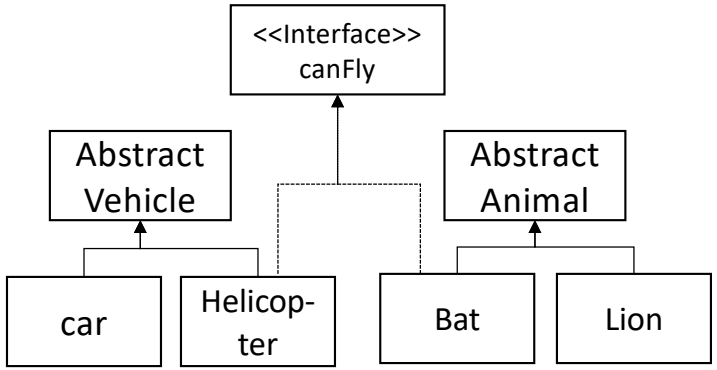


# Abstract Class vs Interface



# Abstract Class vs Interface





# Activity from last class:

- Using inheritance to design a class representing an unmodifiable list of Card that is constructed through a card array.
- What methods do you need to override?
- How to override them?

```java
public class CardList extends AbstractList<Card>
{
    private final Card[] aCards;

    CardList(Card[] pCards)
    {
        assert pCards!= null;
        aCards = pCards;
    }

    @Override
    public Card get(int index)
    {
        assert index>=0 && index<size();
        return aCards[index];
    }
    @Override
    public int size()
    {
        return aCards.length;
    }


}
```

```java
public static void main(String[] pArgs)
{
    Card[] cards = new Card[2];
    cards[0] = new Card(Rank.ACE, Suit.CLUBS);
    cards[1] = new Card(Rank.FIVE, Suit.DIAMONDS);
    CardList cardList = new CardList(cards);

    System.out.println(cardList.contains(cards[1]));

    for (Iterator<Card> iter=cardList.iterator();
        iter.hasNext(); )
    {
        Card element = iter.next();
        System.out.println(element);
    }
}
```

Unmodifiable list?

```
public class CardList extends AbstractList<Card>
{
    private final Card[] aCards;

    CardList(Card[] pCards)
    {
        assert pCards!= null;
        aCards = pCards;
    }

    @Override
    public Card get(int index)
    {
        assert index>=0 && index<size();
        return aCards[index];
    }
    @Override
    public int size()
    {
        return aCards.length;
    }


}
```

<- Returns the element at the specified position in this list.

<- Returns the number of elements in this list.

### *Method Summary*

All Methods | Instance Methods | Abstract Methods | Concrete Methods

| Modifier and Type | Method and Description |
|---|---|
| boolean | **add**(**E** e)<br>Appends the specified element to the end of this list (optional operation). |
| void | **add**(int index, **E** element)<br>Inserts the specified element at the specified position in this list (optional operation). |
| boolean | **addAll**(int index, **Collection**<? extends **E**> c)<br>Inserts all of the elements in the specified collection into this list at the specified position (optional operation). |
| void | **clear**()<br>Removes all of the elements from this list (optional operation). |
| boolean | **equals**(**Object** o)<br>Compares the specified object with this list for equality. |
| abstract **E** | **get**(int index)<br>Returns the element at the specified position in this list. |
| int | **hashCode**()<br>Returns the hash code value for this list. |
| int | **indexOf**(**Object** o)<br>Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element. |
| **Iterator**<**E**> | **iterator**()<br>Returns an iterator over the elements in this list in proper sequence. |
| int | **lastIndexOf**(**Object** o)<br>Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element. |
| **ListIterator**<**E**> | **listIterator**()<br>Returns a list iterator over the elements in this list (in proper sequence). |
| **ListIterator**<**E**> | **listIterator**(int index)<br>Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list. |
| **E** | **remove**(int index)<br>Removes the element at the specified position in this list (optional operation). |
| protected void | **removeRange**(int fromIndex, int toIndex)<br>Removes from this list all of the elements whose index is between `fromIndex`, inclusive, and `toIndex`, exclusive. |

```java
public class CardList extends AbstractList<Card>
{
    private final Card[] aCards;

    CardList(Card[] pCards)
    {
        assert pCards!= null;
        aCards = pCards;
    }

    @Override
    public Card get(int index)
    {
        assert index>=0 && index<size();
        return aCards[index];
    }
    @Override
    public int size()
    {
        return aCards.length;
    }


}
```

```java
public static void main(String[] pArgs)
{
    Card[] cards = new Card[2];
    cards[0] = new Card(Rank.ACE, Suit.CLUBS);
    cards[1] = new Card(Rank.FIVE, Suit.DIAMONDS);
    CardList cardList = new CardList(cards);

    System.out.println(cardList.contains(cards[1]));

    for (Iterator<Card> iter=cardList.iterator();
        iter.hasNext(); )
    {
        Card element = iter.next();
        System.out.println(element);
    }

    cardList.set(0,
        new Card(Rank.ACE, Suit.CLUBS));
}
```

```
public abstract class AbstractList<E> extends AbstractCollection<E> implements List<E>
{
      … …



      public E set(int index, E element) {
             throw new UnsupportedOperationException();
         }
      … …
 }
```

# java.util.AbstractList

This class provides a skeletal implementation of the List interface to minimize the effort required to implement this interface backed by a "random access" data store (such as an array).

To implement an unmodifiable list, the programmer needs only to extend this class and provide implementations for the get(int) and size() methods.

To implement a modifiable list, the programmer must additionally override the set(int, E) method (which otherwise throws an UnsupportedOperationException). If the list is variable-size the programmer must additionally override the add(int,E) and remove(int) methods.

… …

```
public class CardList extends AbstractList<Card>
{
    private final Card[] aCards;

    @Override
    public Card get(int pIndex)
    {
        assert pIndex>=0 && pIndex<size();
        return aCards[pIndex];
    }

     public List<Card> getRange(int pStartIndex, int
     pEndIndex)
     {
         assert pStartIndex>=0 && pEndIndex<size();
         List<Card> cards = new ArrayList<>();
         for (int i = pStartIndex;i <= pEndIndex;  i++)
         {
             cards.add(aCards[i]);
         }
         return cards;
     }
}
```

Abby

# Extend CardList to count list element access

```
public class AccessCountCardList extends CardList
{

    private int count =0;
    AccessCountCardList(Card[] pCards)
    {
        super(pCards);
    }


    @Override
    public Card get(int pIndex)
    {
        assert pIndex>=0 && pIndex<size();
        Card card = super.get(pIndex);
        count ++;
        return card;
    }
```

Pat

# Extend CardList to count member access

```
public class AccessCountCardList extends CardList
{

    … …
    @Override
    public List<Card> getRange(int pStartIndex, int pEndIndex)
    {
        assert pStartIndex>=0 && pEndIndex<size();
        List<Card> cards = super.getRange(pStartIndex, pEndIndex);
        count += cards.size();
        return cards;
    }

    public void printAccessCount()
    {
        System.out.printf("Total Access Count: %d", count);
    }
```

Pat

How many Card get accessed?

```
public static void main(String[] pArgs)
{
    Card[] cards = new Card[3];
    cards[0] = new Card(Rank.ACE, Suit.CLUBS);
    cards[1] = new Card(Rank.FIVE, Suit.DIAMONDS);
    cards[2] = new Card(Rank.EIGHT, Suit.HEARTS);
    AccessCountCardList cardList =
        new AccessCountCardList(cards);

    for (Card card: cardList.getRange(0, 1))
    {
        System.out.println(card);
    }

    cardList.printAccessCount();

}
```

# CardList gets refactored…

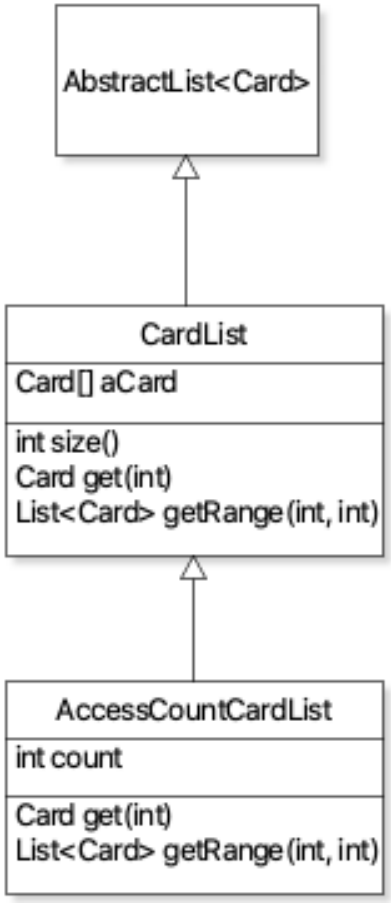

```
public class CardList extends AbstractList<Card>
{
    private final Card[] aCards;
    ……
    @Override
    public List<Card> getRange(int pStartIndex, int pEndIndex)
    {
        assert pStartIndex>=0 && pEndIndex<size();
        List<Card> cards = new ArrayList<>();
        for (int i = pStartIndex;i <= pEndIndex;  i++)
        {
            cards.add(get(i)); ←  aCards[i]
        }
        return cards;
    }
}
```

How many Card get accessed?

```
public static void main(String[] pArgs)
{
    Card[] cards = new Card[3];
    cards[0] = new Card(Rank.ACE, Suit.CLUBS);
    cards[1] = new Card(Rank.FIVE, Suit.DIAMONDS);
    cards[2] = new Card(Rank.EIGHT, Suit.HEARTS);
    AccessCountCardList cardList =
        new AccessCountCardList(cards);

    for (Card card: cardList.getRange(0, 1))
    {
        System.out.println(card);
    }

    cardList.printAccessCount();

}
```

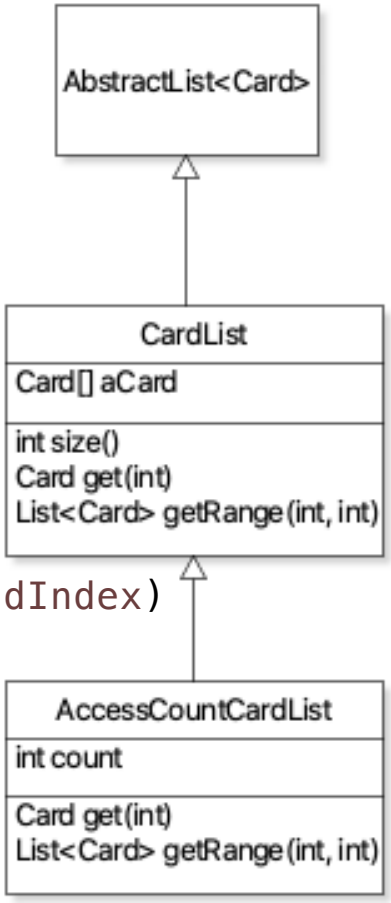
# Demo in Intellij

```
AccessCountCardList cardList =
  new AccessCountCardList(cards);

 cardList.getRange(0, 1))
```



# Demo in Intellij

```
public class CardList extends AbstractList<Card>
{
    private final Card[] aCards;
    ……
    @Override
    public List<Card> getRange(int pStartIndex, int pEndIndex)
    {
        assert pStartIndex>=0 && pEndIndex<size();
        List<Card> cards = new ArrayList<>();
        for (int i = pStartIndex;i <= pEndIndex;  i++)
        {
            cards.add(get(i));
        }
        return cards;
    }
}
```



# Inheritance violates encapsulation

- A subclass depends on the implementation details of its superclass for its proper function.

# Activity 1: Fix ideas?

```
public class AccessCountCardList extends CardList
{

    @Override
    public Card get(int pIndex)
    {
        assert pIndex>=0 && pIndex<size();
        Card card = super.get(pIndex);
        count ++;
        return card;
    }
    @Override
    public List<Card> getRange(int pStartIndex, int pEndIndex)
    {
        assert pStartIndex>=0 && pEndIndex<size();
        List<Card> cards = super.getRange(pStartIndex, pEndIndex);
        count += cards.size();
        return cards;
    }
}
```

```
public class DelegatedAccessCountCardList extends AbstractList<Card>
{
	private CardList aCardList;
	private int count = 0;

	……
	@Override
	public Card get(int pIndex)
	{
		assert pIndex>=0 && pIndex<size();
		Card card = aCardList.get(pIndex);
		count++;
		return card;
	}

	public List<Card> getRange(int pStartIndex, int pEndIndex)
	{
		assert pStartIndex>=0 && pEndIndex<size();
		List<Card> cards = aCardList.getRange(pStartIndex, pEndIndex);
		count += cards.size();
		return cards;
	}
```

```
    final
public class CardList extends AbstractList<Card>
{
    private final Card[] aCards;

    @Override
    public Card get(int pIndex)
    {
        assert pIndex>=0 && pIndex<size();
        return aCards[pIndex];
    }
    public List<Card> getRange(int pStartIndex, int
    pEndIndex)
    {
        assert pStartIndex>=0 && pEndIndex<size();
        List<Card> cards = new ArrayList<>();
        for (int i = pStartIndex;i <= pEndIndex;  i++)
        {
            cards.add(aCards[i]);
        }
        return cards;
    }

}
```

# Change inheritance to composition

- Delegate duties using interface
- Decoupled implementation between two classes

**Inheritance**

**code reuse, subtype**

# Liskov Substitution Principle

- If S is a subtype of T, then objects of type T may be replaced with objects of type S without altering any of the desirable properties of the program.

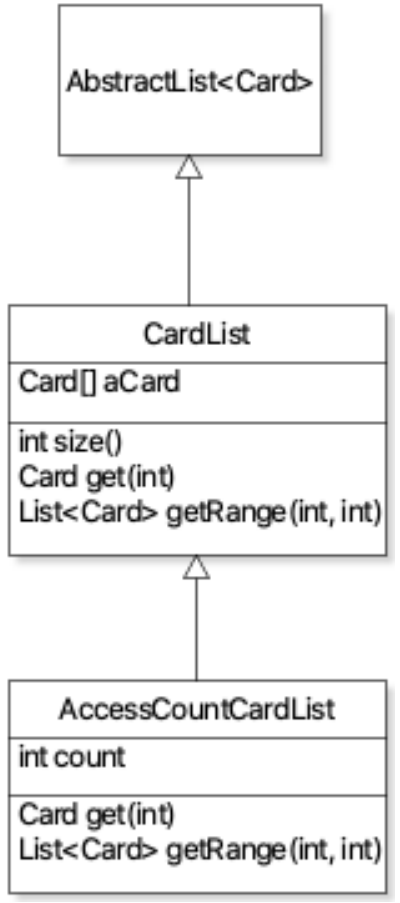
*S is substitutable of T*

Barbara Liskov, Computer Scientist at MIT

# Proper use of Inheritance

- Inherited methods in subclass
  - Cannot have stricter preconditions
  - Cannot have less strict postconditions
  - Cannot take more specific types as parameters
  - Cannot make the method less accessible (e.g. public -> protected)
  - Cannot throw more checked exceptions
  - Cannot have a less specific return type

Cannot surprise the client



# Rectangle Class

```
public class Rectangle
{
    protected int aWidth;
    protected int aHeight;
    public void setSize(int pWidth, int pHeight)
    {
        aWidth = pWidth;
        aHeight = pHeight;
    }

    public int getArea()
    {
        return aWidth*aHeight;
    }
}
```

Every square is a rectangle, so…

# How to design the setSize for Square?

- Option1:

```
public class Square extends Rectangle
{
    @Override
    public void setSize(int pWidth, int pHeight)
    {
        assert pWidth == pHeight;
        aWidth = pWidth;
        aHeight = pHeight;
    }

}
```

# How to design the setSize for Square?

- Option2:

```
public class Square extends Rectangle
{
    public void setSize(int pEdgeLength) {
        aWidth = pEdgeLength;
        aHeight = pEdgeLength;
    }

    @Override
    public void setSize(int pWidth, int pHeight)
    {
        throw new UnsupportedOperationException("Invalid operation
            for Square.");
    }

}
```

# Square and Rectangle are not related

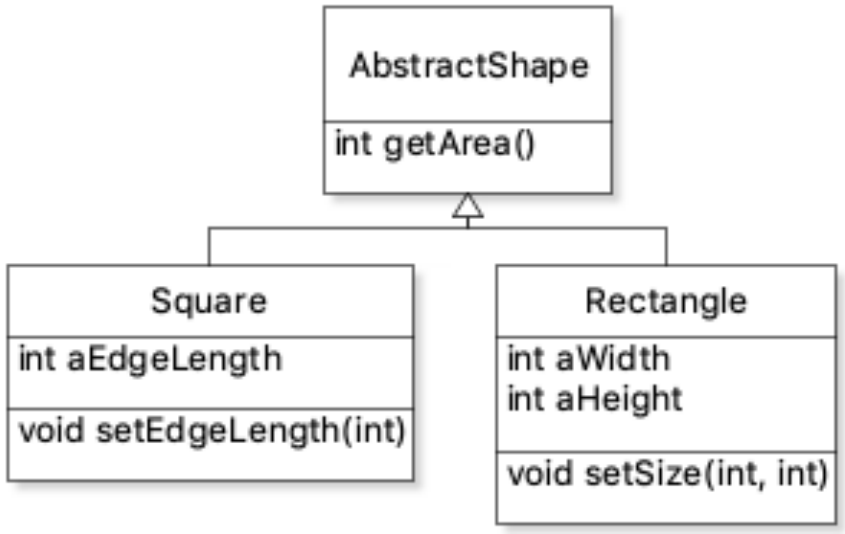
- If square is a subtype of rectangle

  Client will be surprised when they find out that width and height cannot be changed independently

- If rectangle is a subtype of square

  Client will be surprised when they find out that width and height are not equal

# Square and Rectangle are not related



```
public class Deck extends CardSource
{
    protected final CardStack aCard = new CardStack();

    public Card draw(){ return aCard.pop();}

    public boolean isEmpty() { return aCard.isEmpty();}

}
```

```
public class DrawBestDeck extends Deck
{

    @Override
    public Card draw()
    {
        Card card1 = aCard.pop();
        Card card2 = aCard.pop();
        if (card1.compareTo(card2)>0)
        {
           aCard.push(card2);
            return card1;
        }
         aCard.push(card1);
        return card2;

    }
}
```

## Violation of Liskov Substitution Principle

Precondition becomes stricter

# Activity2:

- Given the **Student** class, and **UndergradCourse** is a true subtype of **Course**. Which of the methods in **UndergradStudent** violates the Liskov Substitution Principle and why?

```
public class Student{
   public Course recommend(Course pCourseID);}
```

```
public class UndergradStudent extends Student {
```

1. `public Course recommend(UndergradCourse pCourseID);`
2. `public UndergradCourse recommend(Course pCourseID);`
3. `public UndergradCourse recommend(Object pCourseID);`
4. `public Course recommend(Course pCourseID) throw SomeCheckedException;`

# Summary

- Consider using composition rather than inheritance when using “foreign” classes.
- Reason if a true subtype relationship exist;
  *“If it looks like a duck and quacks like a duck but it needs batteries, you probably have the wrong abstraction.”*
- Document self-use of overridable methods when designing classes to be inherited. Write subclasses to test;
- Prohibit subclassing when it’s not safe. “final” class, or restrict accessibility;
- Refactoring.