M6 (b) - Inheritance

Jin L.C. Guo

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

RA position at

- Swevo lab
- Contact Professor Martin Robillard this week.

Recap

- Conceptual foundations of inheritance
- Inheritance in Java
- Abstract Class
- Template Method Pattern

Objective

- Common problems of inheritance
- Liskov Substitution Principle

Activity from last class:

- What methods do you need to override for implementing an unmodifiable list of Card that is constructed through a card array.
- How to override them?

Unmodifiable list?

```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);
    }
}
```

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 an modifiable list, the programmer must additionally override the set(int, E) method (which otherwise throws an UnsupportedOperationException).

… …

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

Inheritance violates encapsulation

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

```java
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;
    }
}
```

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;
    }
```

Extend CardList to count member access

```
public class AccessCountCardList extends CardList
{

    … …

    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);
    }
```

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;
    ……

    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();

}
```

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;
    }
    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;
    }
}
```

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 pWeight, int pHeight)
    {
        aWidth = pWeight;
        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

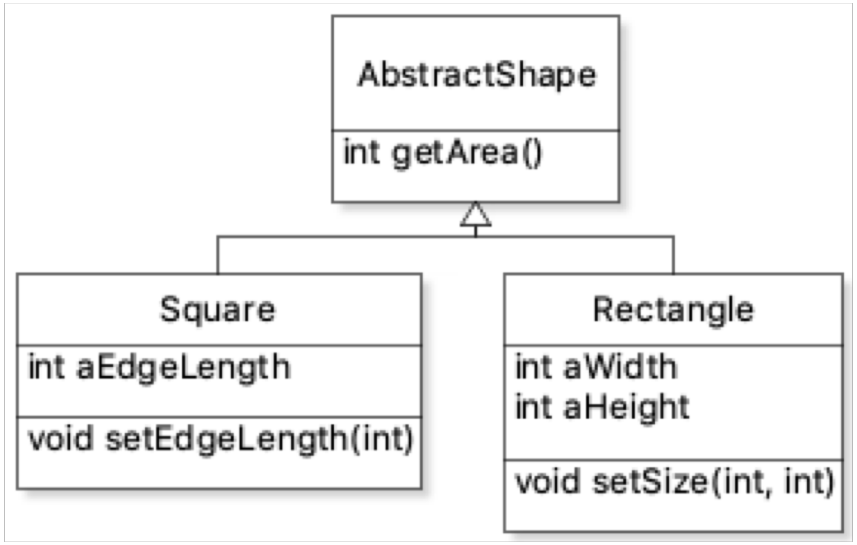
- If square is a subtype of rectangle

  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 the equal

Square and Rectangle are not related



```
public class DrawTwoDecorator implements CardSource
{
    private final CardSource aCardSource;

    @Override
    public Card draw()
    {
        assert size()>0;
        Card card1 = aCardSource.draw();
        if (aCardSource.size() > 0)
        {
            Card card2 = aCardSource.draw();
            if (card1.compareTo(card2)>0)
            {
                return card1;
            }
            return card2;
        }
        return card1;
    }
}
```

```
public class DrawTwoDecorator implements CardSource
{
    private final CardSource aCardSource;

    @Override
    public Card draw()
    {
        assert size()>1;
        Card card1 = aCardSource.draw();
        Card card2 = aCardSource.draw();
        if (card1.compareTo(card2)>0)
        {
            return 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 **UndergradCourse** violates the Liskov Substitution Principle ?

```
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.