An Exercise

Strictly played, Klondike is hard to win. One way to make the game more winnable is to loosen the rule for playing cards on the tableau. The game Thumb and Pouch does just that. Thumb and Pouch has the same rules as Klondike *except* that a card may be played on a table pile if the pile's top card is of the next higher rank and of any suit but its own.

Modify our Solitaire game to implement Thumb and Pouch.

Here is the relevant piece of code:

```
class TablePile extends CardPile
{
   ...
   public boolean canTake (Card aCard)
   {
      if ( isEmpty() )
         return aCard.rank() == 12;
      Card topCard = top();
      return (aCard.color() != topCard.color())    &&
             (aCard.rank() == topCard.rank() - 1);
   }
   ...
}
```

We could modify this method directly—but that would make it difficult to fire up a game of Klondike when we felt tougher.

The better option would be to use inheritance to add a new kind of pile:

- Create a subclass of `TablePile` for Thumb and Pouch:

- Create a subclass of `Solitare` that overrides the `init()` method to use `ThumbAndPouchTablePile`s instead of plain `TablePile`s.

.

A Solution

```
class ThumbAndPouchTablePile extends TablePile
{
   ThumbAndPouchTablePile (int x, int y, int c)
   {
      super(x, y, c);
   }

   public boolean canTake (Card aCard)
   {
      if ( isEmpty() )
         return aCard.rank() == 12;
      Card topCard = top();
      return (aCard.suit() != topCard.suit())    &&
             (aCard.rank() == topCard.rank() - 1);
   }

public class ThumbAndPouchSolitare extends Solitare
{
   public ThumbAndPouchSolitare ()
   {
      super();
   }

   public void init ()
   {
      // first allocate the arrays...

      // then fill them in...
      allPiles[0] = deckPile ...;
      allPiles[1] = discardPile ...;
      for (int i = 0; i < 4; I++) ... fill suitPiles
      for (int i = 0; i < 7; i++)
          allPiles[6+i] = tableau[i] =
             new ThumbAndPouchTablePile( ... );
   }
}
.
```

Inheritance versus Composition

Both inheritance and composition are techniques for reusing code.

- We use inheritance to model "**is a**" relationships:

  - A suit pile *is a* card pile.
  - A pin ball *is a* ball.
  - A bounded ball *is a* movable ball, which *is a* ball.

- We use composition to model "**has a**" relationships:

  - A solitaire game *has a* solitaire frame to display itself in.
  - A pin ball game *has a* collection of pin ball targets.
  - A movable ball *has a*n x-differential and a y-differential.
  - A ball *has a* color, a radius, and x- and y- coordinates.

.

When should we use which? Here are some guidelines:

- Try to identify the kind of relationship ("**is a**" and "**has a**") that exists in your application. Use the technique that fits the relationship.

- If you are ever in doubt, ask yourself, "Would I ever want to use an <x> in place of a <y>?" If so, consider using inheritance.

  If not, *definitely* use composition.

- Use inheritance when you are building a prototype, since it allows you to make a working system more quickly than composition. But be willing to refactor the system to use composition when building the deliverable.

- Use composition whenever inheritance would violate the principle of substitutability.

Inheritance versus Composition

Inheritance is more powerful than composition. It gives you stuff for free.

But if you don't *need* the stuff, then inheritance may be overkill.

And if you don't ***want*** the stuff, then inheritance can be dangerous.

Learning to use inheritance effectively will make you a more productive programmer. But the key is to use it effectively.

But in Java and other object-oriented languages, I have to inherit from *some* class...

True, but you can always just inherit from `Object`!

The Principle of Substitutability

The power of inheritance lies in the principle of substitutability. Using this technique, we can write a new class that fits seamlessly into an existing program.

- Think back to Day 2 and my example of drawing programs.
- Think about adding a new `CardPile` to our `Solitaire` game.
- Think about adding fancy kinds of balls to our ball games.

An Exercise

Suppose that you are writing a software system to manage a video rental store. We will almost certainly need to implement a `Video` class.

Design a `Video` class. This involves:

- Identifying the state of a `Video` object. *What are its instance variables?*
- Identifying the behavior of a `Video` object. *What are its methods?*

Remember the design technique we learned about several weeks ago: Walk through several scenarios involving videos at the store. From the needs of each scenario, try to figure out what the video needs to know and what the video needs to do.

A video needs to knows its name and its copy number, at the very least. It probably should know its price and rental period. For a really useful system, we may want videos to know many details of the movie, for searching purposes.

A video needs to be able to check itself out, to check itself in, and answer queries about itself—including its location.

These behaviors indicate that the video also needs to know its location: on the shelf, checked out to a patron, or somewhere in between.

Toward A Solution

How can we represent the video's location? Over time a particular video sits on the shelf, sits in the returned bin, and is checked out to a patron.

1. We might try composition. Add a `location` instance variable that serves as a flag, with 0 = on the shelf, 1 = checked out, and 2 = in between.

The advantages of this approach:	Simple design.
	Transparent to clients.
The disadvantages:	Cluttered implementation.
	Combined responsibilities.
	Hard to add new locations.

2. Instead, we might try inheritance. Create subclasses of `Video`, one for each of the different "kind" of `Video`:

The advantages of this approach:	Easy to add new locations.
	Cleaner implementation.
	Divided responsibilities.
	Simple enough design.

   So, are we there yet?

Closer to a Solution

The problem with this solution is that a particular `Video` needs to behave like a `RentedVideo`, a `ReturnedVideo`, and a `ShelvedVideo` at different times — repeatedly — throughout its life.

Most OO languages do not allow us to change an object's class at run time. Even when they do, the operation is quite expensive.

How might we solve this problem?

We could simply create an instance of the now-appropriate class, copy the common data from the old object, assign the new object to the `Video` variable holding the old object, and destroy the old object.

But...

- We will have to do this repeatedly throughout the life of every video in the store.
- Doing this repeatedly is quite expensive.
- There may be many `Video` variables holding the old object. We will have to know every variable, so that we can update it.

Even worse, this solution is **not** transparent to clients. One of our chief goals in programming, especially OOP, is to hide implementation details from the users of our classes.

This location thing is all implementation detail. The client code simply wants to ask questions of the video, check it out, and check it in.

Can we achieve the advantages of this approach while still hiding details from client code?

A Good Solution: Use Inheritance and Composition Together

The advantages we seek came from using inheritance.
Hiding details comes best from using composition.

Why not try using both??

Solution: Separate the object from the roles it plays. Implement the roles using an inheritance hierarchy. Let the object contain an instance of a role.

In the `Video` scenario, we end up with something like this:

Now, a `Video` can behave like a `RentedVideo`, a `ShelvedVideo`, and a `ReturnedVideo` at different times — repeatedly — through its life. It does so by using *substitutable* subclasses that have the same interface.

We don't have to worry about making the `Video` object change its class at run time. So our clients are protected.

We do have to create and destroy instances repeatedly, but they are smaller and thus less expensive. (And this change is hidden from our clients.)

The Substitution Design Pattern

This is a common problem when building large systems. Many database objects need to play different roles over time. We can also think of many simple examples:

- Budd gives the example of frogs maturing over time.
- In Data Structures, you learned that an empty binary search trees can become non-empty if we insert a value, and a non-empty search tree can become empty if we remove its last value.

Our solution is a common recipe for solving this common problem, which occurs in many different contexts.

In OOP lingo, we call this solution the *Substitution Pattern* or the *Roles Played Pattern*. Budd refers to the idea as *dynamic composition* and devotes Section 10.5.1 to it.

In software design lingo, a pattern is a common recipe for solving a common problem that occurs in many different contexts. Budd devotes a whole chapter to some of the ones you should know upon leaving this course, and we will discuss these and other design patterns throughout the semester.