Inheritance

Jason Miller
Shepherd University

Goal: Become an Efficient Programmer

- Avoid writing the same code every month.
- Get faster every month!
- Re-use your own code to accomplish new tasks.
- Write code that other programmers can use for other tasks.

Techniques for efficient programming (part 1)

- Composition
  - Do not try to solve the entire problem with one method or even one class.
  - You'll be more efficient if you implement, design, and test one component at time.
  - You'll build powerful programs quickly by reusing previously build components.
  - Use the **has-a** relationship in Java programs!
- Delegation
  - No class should try to do everything by itself.
  - Class A **delegates** to class B when A lets B do some of the work.
  - Class A might even have **pass-thru methods** that simply call a similar method of B.
  - Exploit the **has-a** relationship. Avoid redundant code!

Techniques for efficient programming (part 2)

- Inheritance
  - If class A does 90% of what you need, don’t rewrite it, and don’t copy it either.
  - Write a class B that **extends** class A.
  - Class B **inherits** all the methods of A and can have extra methods too.
  - Even if A has a bug, it is better to one copy of the bug instead of two.
  - Avoid spreading bugs! Practice code reuse without copy-and-paste.
- Polymorphism
  - The roots of the word polymorphism mean many shapes.
  - In object-oriented programming, the word means letting **one class act like another**.
  - Java allows polymorphism through **inheritance** (also through interfaces, not covered here).
  - Write a class that purposely acts like another. This is **compile-time** polymorphism.
  - Reuse a class in ways it wasn’t designed for. This is **run-time** polymorphism.

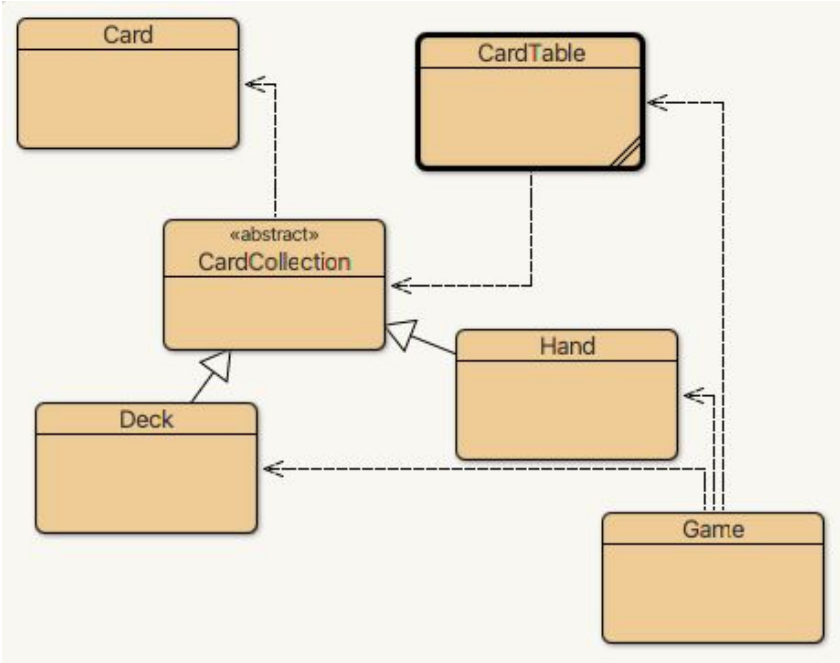
Composition

CardTable **has-a** CardCollection

- It actually has many: one deck and two hands.
- This is a one-to-many relation.

CardCollection **has-a** Card

- It actually has an array of Card.
- This is a one-to-many relation.
- This form of composition is called **aggregation**.



UML Class Diagram

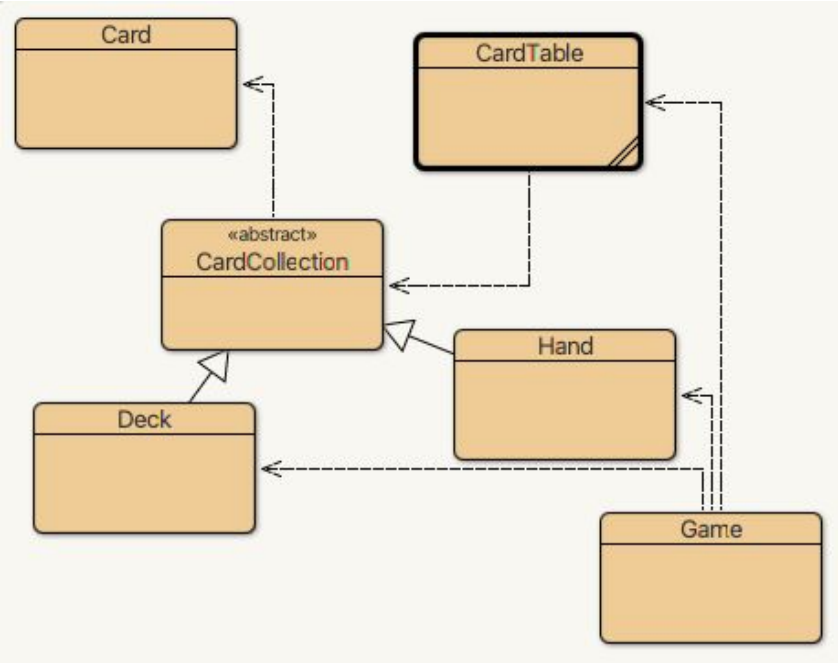
Delegation

CardCollection **delegates** to Card

- When CardCollection.**sort()** needs to know which is the higher card, it calls card1.compareTo(card2).
- Thus CardCollection delegates some decision making to Card.

CardTable **delegates** to CardCollection

- CardTable owns several CardCollections: deck, hand1, hand2.
- The CardTable.draw() method delegates to the draw() method of each CardCollection it owns.



UML Class Diagram

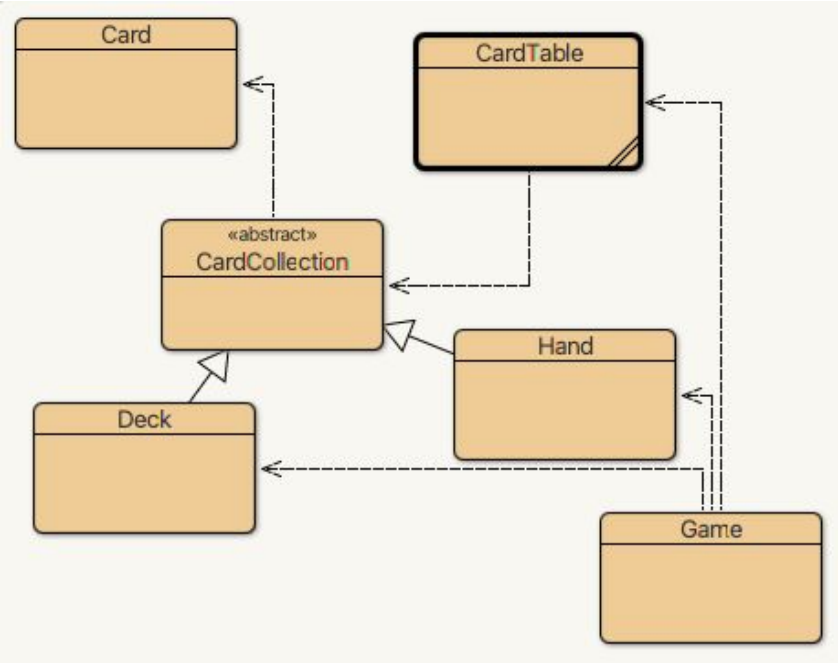
Inheritance

Hand **is-a** CardCollection

- Hand **extends** CardCollection.
- Hand inherits every method that CardCollection has.
- The developer of Hand wrote a great class while only adding a little code.

Deck **is-a** CardCollection

- CardCollection is the **superclass**. Deck is the **subclass**.
- Fixing a bug in the superclass fixes it for all the subclasses.
- Inheritance avoids copy-and-paste, which would lead to multiple copies of every bug.



UML Class Diagram

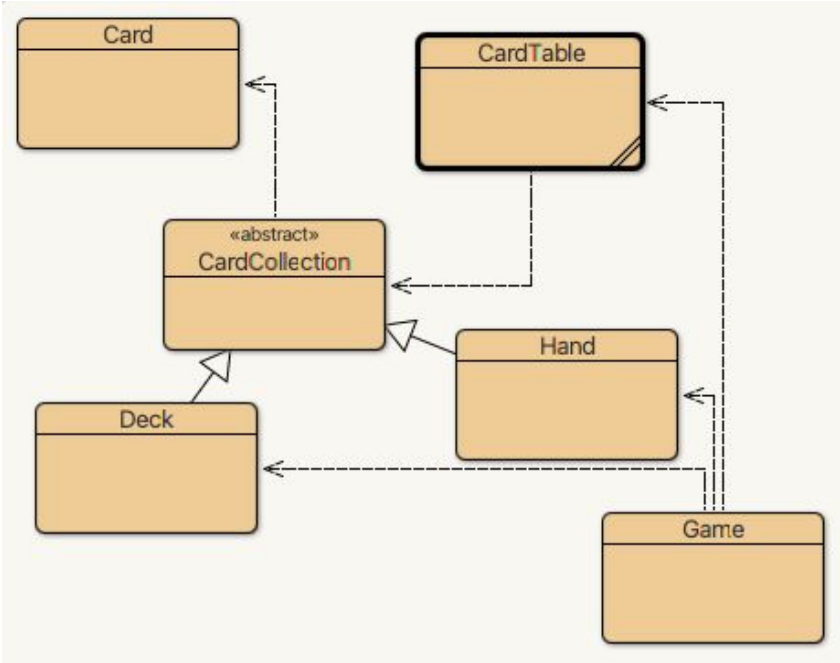
Polymorphism

Hand **is-a** CardCollection

- You can pass a Hand object to any method that takes CardCollection.
- The Hand developer was aware of the duality, so this is **compile-time** polymorphism.

Hand **is-a** CardCollection

- CardTable was written before Hand existed. But the code works with Hand as well as Deck!
- The CardTable developer wasn't aware of the duality, so this is **run-time** polymorphism.



UML Class Diagram

Java keywords for inheritance: 1/3

```
Class1 implements Interface2
```

The interface is a contract. It promises which methods will be present but it provides no implementations. The compiler demands that Class1 implement every method of Interface2. The compiler allows instances of the class to be used wherever the interface is called for. Class1 "is-a" Interface2. Classes may implement zero to many interfaces.

Java keywords for inheritance: 2/3

```
abstract Class3
```

The abstract class provides code for a few methods, but the class is meant to be extended not used directly. The compiler won't allow the constructor to be called. The compiler allows "abstract Class3 implements Interface2" even if Class3 does not fulfill the contract.

Java keywords for inheritance: 3/3

```
Class5 extends Class4
```

The child class inherits all the methods of the parent. The parent may be abstract or concrete. The child may override the parent's methods. The child may implement additional methods. The compiler allows child instances to be used anywhere the parent is called for. Class5 "is-a" Class4. In Java, each child has exactly one parent. All classes extend Object, which implements equals() and toString().

Review questions

- Why are we studying efficiency techniques?
- How are composition and delegation related?
- How are inheritance and polymorphism related?
- Compare and contrast: composition and inheritance.
- Which efficiency techniques lead to fewer lines of code?
- What is the danger associated with code copy & paste?
- What's wrong with writing the monolith?
- In a UML class diagram, what does a solid arrow indicate?
- What does a dashed arrow indicate?
- Explain the differences between two types of polymorphism.