Principles of Software Construction: Objects, Design, and Concurrency

Inheritance and delegation (leftovers)

Claire Le Goues **Bogdan Vasilescu**

Recall our intro lecture sorting example:

Version A:

```
static void sort(int[] list, boolean ascending) {
   …
   boolean mustSwap;
   if (ascending) {
      mustSwap = list[i] > lis
   } else {
      mustSwap = list[i] < lis
   }
   …
}
```

Version B':

```
interface Order {
  boolean lessThan(int i, int j);
}
class AscendingOrder implements Order {
  public boolean lessThan(int i, int j) { return i < j;
}
class DescendingOrder implements Order {
  public boolean lessThan(int i, int j) { return i > j;
}

static void sort(int[] list, Order order) {
  …
  boolean mustSwap =
    order.lessThan(list[j], list[i]);
  …
```

Delegation

- *Delegation* is simply when one object relies on another object for some subset of its functionality
  - e.g. here, the sorter is delegating functionality to some `Order`

```
interface Order {
  boolean lessThan(int i, int j);
}
class AscendingOrder implements Order {
  public boolean lessThan(int i, int j) { return i < j; }
}
class DescendingOrder implements Order {
  public boolean lessThan(int i, int j) { return i > j; }
}
…
static void sort(int[] list, Order order) {
  …
  boolean mustSwap =
    order.lessThan(list[j], list[i]);
```

Delegation

- Judicious delegation enables code reuse
  - The sorter can be reused with arbitrary sort orders
  - `Order` objects can be reused with arbitrary client code that needs to compare ints

```
interface Order {
  boolean lessThan(int i, int j);
}
class AscendingOrder implements Order {
  public boolean lessThan(int i, int j) { return i < j; }
}
class DescendingOrder implements Order {
  public boolean lessThan(int i, int j) { return i > j; }
}
…
static void sort(int[] list, Order order) {
  …
  boolean mustSwap =
    order.lessThan(list[j], list[i]);
```

Using delegation to extend functionality

- Consider the `java.util.List` (excerpted):

```
public interface List<E> {
       public boolean add(E e);
       public E       remove(int index);
       public void    clear();
       …
}
```

- Now suppose we want a list that logs its operations to the console …

Using delegation to extend functionality

- One solution:

```java
public class LoggingList<E> implements List<E> {
  private final List<E> list;
  public LoggingList<E>(List<E> list) { this.list = list; }
  public boolean add(E e) {
      System.out.println("Adding " + e);
      return list.add(e);
  }
  public E remove(int index) {
      System.out.println("Removing at " + index);
      return list.remove(index);
  }
  …
```

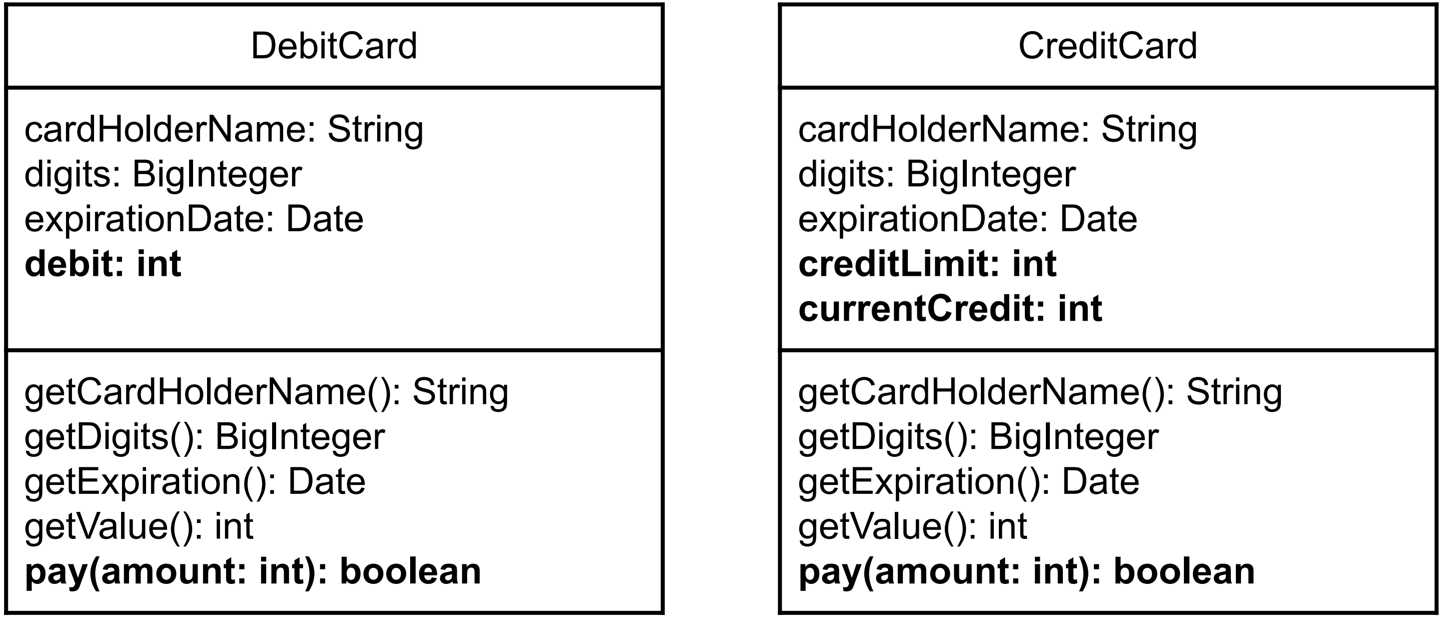
The `LoggingList` *is composed of* a `List`, and delegates (the non logging) functionality to that `List`

Delegation and design

- Small interfaces with clear contracts
- Classes to encapsulate algorithms, behaviors
  - E.g., the `Order`

Another example

Variation in the real world: types of bank cards



DebitCard

cardHolderName: String
digits: BigInteger
expirationDate: Date
debit: int

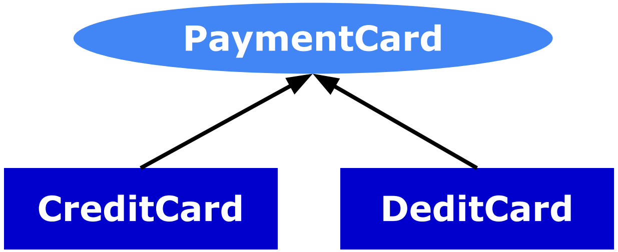
getCardHolderName(): String
getDigits(): BigInteger
getExpiration(): Date
getValue(): int
pay(amount: int): boolean

CreditCard

cardHolderName: String
digits: BigInteger
expirationDate: Date
creditLimit: int
currentCredit: int

getCardHolderName(): String
getDigits(): BigInteger
getExpiration(): Date
getValue(): int
pay(amount: int): boolean

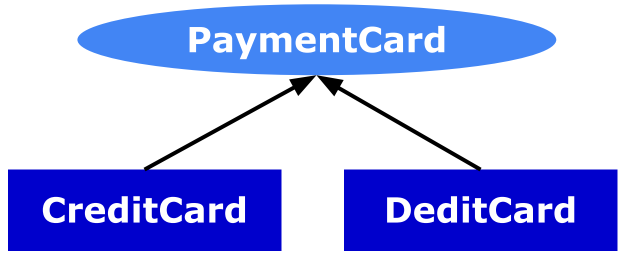
Design option 1



```
public interface PaymentCard {
   String getCardHolderName();
   BigInteger getDigits();
   Date getExpiration();
   int getValue();
   boolean pay(int amount);
}
```

```
class CreditCard implements PaymentCard {

    …

}
class DebitCard implements PaymentCard {

    …

}
```

Design option 1

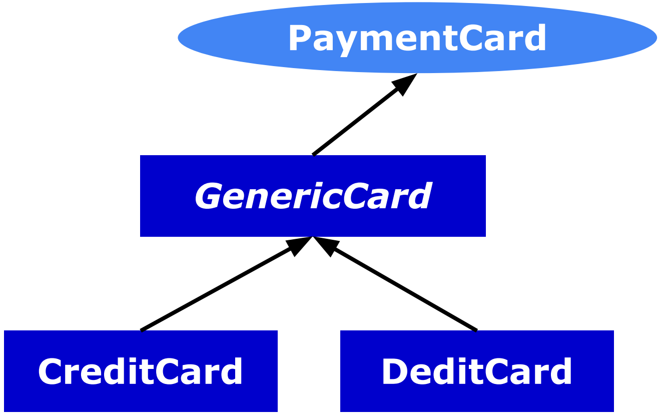


```
public interface PaymentCard {
   String getCardHolderName();
   BigInteger getDigits();
   Date getExpiration();
   int getValue();
   boolean pay(int amount);
}
```

```
class CreditCard implements PaymentCard {
   …
}
class DebitCard implements PaymentCard {
   …
}
```

Lots of duplicated code: many common fields and methods that need to be implemented twice

Design option 2



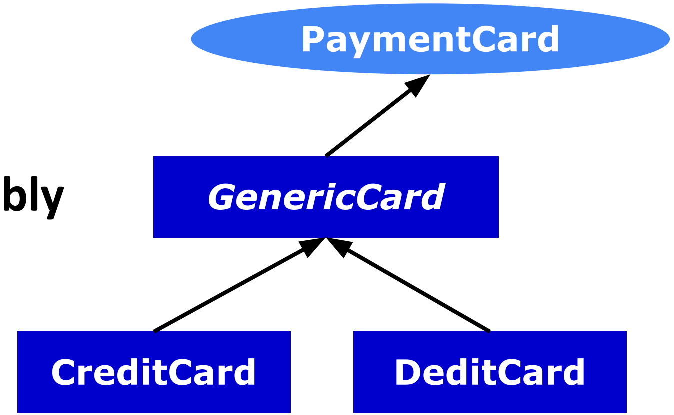
```
abstract class AbstractGenericCard
            implements PaymentCard {
   …
   public String getCardHolderName() {
       return this.cardHolderName;
   }
   public BigInteger getDigits() {
       return this.digits;
   }
   public Date getExpiration() {
       return this.expirationDate;
   }
   abstract boolean pay(int amount);
}
```

```
class CreditCard extends AbstractGenericCard {
   @Override
   public boolean pay(int amount) {
      …
   }
}
class DebitCard extends AbstractGenericCard {
   @Override
   public boolean pay(int amount) {
      …
   }
}
```

Design option 2

```
abstract class AbstractGenericCard
            implements PaymentCard {
   …
   public String getCardHolderName() {
       return this.cardHolderName;
   }
   public BigInteger getDigits() {
       return this.digits;
   }
   public Date getExpiration() {
       return this.expirationDate;
   }
   abstract boolean pay(int amount);
}
```

Much more reuse; inheritance is probably a good choice here. But not always.



```
class CreditCard extends AbstractGenericCard {
   @Override
   public boolean pay(int amount) {
       …
   }
}
class DebitCard extends AbstractGenericCard {
   @Override
   public boolean pay(int amount) {
       …
   }
}
```

Inheritance limits information hiding!

```
public class InstrumentedHashSet<E> extends HashSet<E> {

   public int addCount = 0;

   @Override
   public boolean add(E a) {
       addCount += 1;
       return super.add(a);
   };

   @Override
   public boolean addAll(Collection<? extends E> c) {
       addCount += c.size();
       return super.addAll(c);
   }
}
```

```
public static void main(String[] args) {
   InstrumentedHashSet<String> set = new
InstrumentedHashSet<String>();

   set.addAll(List.of("A", "B", "C"));

   System.out.println(set.addCount);
}
```

What will this print?

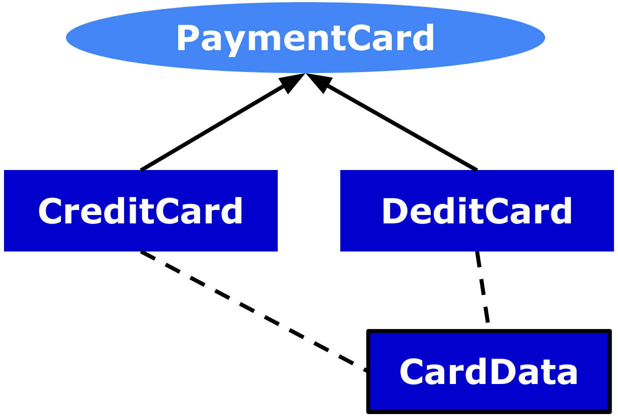
Designing with inheritance in mind

- Document contracts for inheritance
  - The compiler won’t inforce all invariants

- Try to avoid it when composition+delegation is available
  - Delegation reduces coupling

- Enforce or prohibit inheritance where possible
  - In Java: `final` & `abstract`

Design option 3



```
class CardData {
   private final String cardHolderName;
   private final BigInteger digits;
   private final Date expirationDate;

   public CardData(…) {…}
   public String getCardHolderName() {…}
   public BigInteger getDigits() {…}
   public Date getExpiration() {…}
}
```

```
class CreditCard implements PaymentCard {
      private CardData cardData = new(…);
      public BigInteger getDigits() {
            return cardData.getDigits();
      }
      …
}
class DebitCard implements PaymentCard {
      …
}
```

You can still achieve good reuse with composition+delegation

Inheritance vs. Composition + Delegation

- A lot of good design favors composition/delegation over inheritance
    - Delegation supports information hiding
    - Inheritance violates information hiding
- Design and document for inheritance, or prohibit it
    - Document requirements for overriding any method
    - `protected` hooks / helper methods
    - Test with subclasses

Inheritance vs. Composition + Delegation

- It’s not an either/or question
  - Interfaces provide contracts
  - Inheritance provides reuse, strong coupling

Language/Implementation Details

Details: `final`

- A final field: prevents reassignment to the field after initialization
- A final method: prevents overriding the method
- A final class: prevents extending the class
  - e.g., `public final class CheckingAccountImpl { …`
- Not present in TypeScript
  - Called “sealed” in some languages

Details: `abstract`

- An abstract method:
  - must be overridden by a non-abstract subclass
- An abstract class:
  - only classes allowed to have abstract members

Details: `super`

- Similar to `this`
- Refers to any (recursive) parent
  - Depending on what is accessed
- In TS, must call `super();` before using ‘this’
  - Initializes the class
- In Java, super call needs to be first statement in constructor

Example: `super`

```
abstract class AbstractCashCard
            implements PaymentCard {
   private int balance;
   public AbstractCashCard(int balance) {
       this.balance = balance;
   }

   public boolean pay(int amount) {
       if (amount <= this.balance) {
           this.balance -= amount;
           return true;
       }
       return false;
   }
}
```

```
class DebitCard extends AbstractCashCard {

   @Override
   public boolean pay(int amount) {
      boolean success = super.pay(amount);
      if (success)
         this.balance -= this.fee;
      return success;
   }
}
```

Details: type-casting

- Sometimes you want a different type than you have
  - e.g.,

```
double pi = 3.14;
int indianaPi = (int) pi;
```

In TS:

```
(dog as Animal).identify()
```

- Useful if you know you have a more specific subtype:

```
Account acct = …;
CheckingAccount checkingAcct = (CheckingAccount) acct;
long fee = checkingAcct.getFee();
```

  - Will get a `ClassCastException` if types are incompatible
- Advice: avoid downcasting types
  - Never(?) downcast within superclass to a subclass

Summary

- Inheritance is a powerful tool
  - That takes coupling to the extreme
  - And deserves careful consideration
  - Template method pattern enforces reuse, limits customization
- Subtyping and inheritance are related, but not the same
  - Composition & Delegation are often the right tools
  - Not mutually exclusive