Principles of Software Construction: Objects, Design, and Concurrency

Inheritance and delegation

Claire Le Goues Vincent Hellendoorn

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

No quiz today; a lot to cover!

Administrivia

HW2 grading comment.

First exam is next Thursday (1 week from today). It will be held in class.

We will release a sample midterm/study guide on Piazza.

- It is *much longer* than the actual exam will be!
- We will *not* be releasing sample answers.

You may bring a study guide of up to 4 pages, front and back, typed or handwritten. Otherwise, closed book etc.

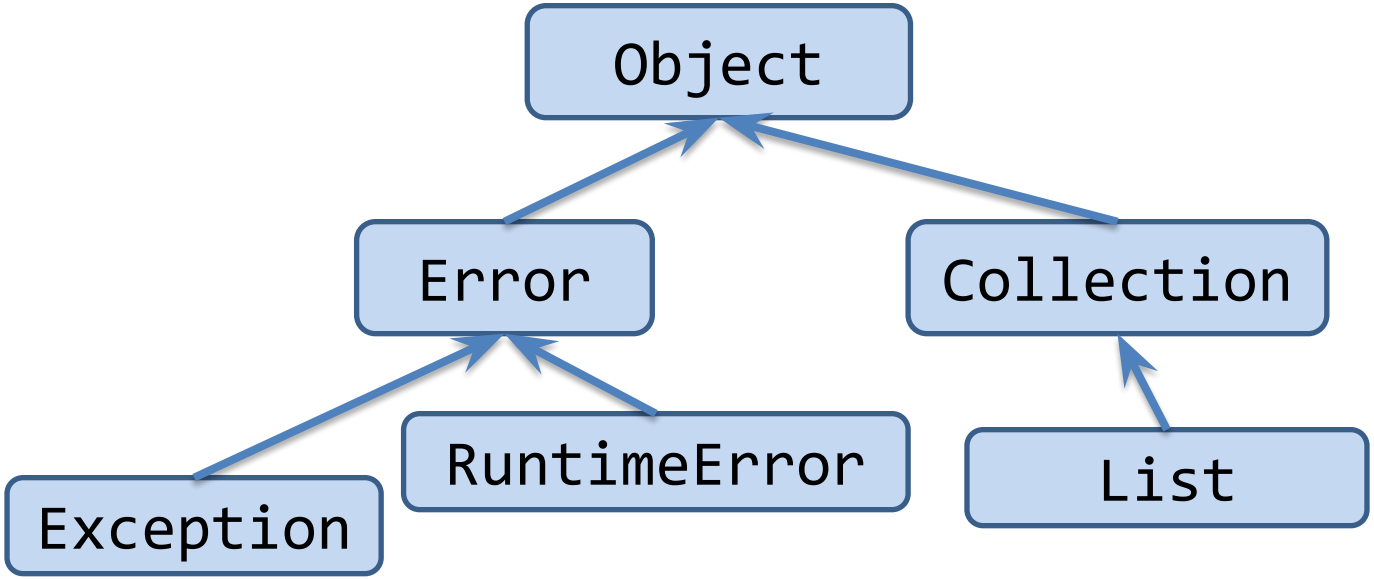
If you have covid (...or the flu or anything else…), reach out to us via email and we will accommodate you. (...don't come to the exam with covid…)

Learning goals

- Understand the role of class hierarchies in object-oriented languages.
- Define inheritance and understand some of the conditions under which it is appropriate/inappropriate to use in system design.
- Identify behavioral subtyping and use it to make good decisions about whether a class should extend another class.
- Achieve reuse via composition and delegation instead of inheritance.
- Contrast inheritance and composition and delegation in terms of benefits/drawbacks.
- If we get to it: describe the template method pattern and the situations in which it is useful, contrast with the strategy pattern.

All object types exist in a *class hierarchy*

In Java:



Class hierarchy basic terminology

A class hierarchy is a tree

- Parent/child relation is called: superclass/subclass
- A class **`extends`** its superclass
- The root is “Object” -- if a class extends nothing explicitly, it extends Object implicitly

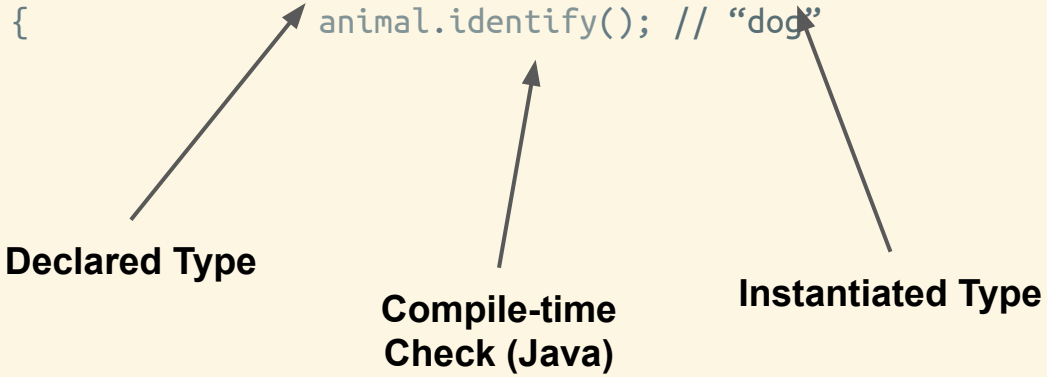
Primitive types are *not* in the class hierarchy

Discussion question: What does/should it mean to “extend” a class?

Inheritance enables Extension & Reuse

```
class Animal {
    final String name;

    public Animal(String name) {
        this.name = name;
    }

    public String identify() {
        return this.name;
    }
}
```

```
class Dog extends Animal {
    public Dog() {
        super("dog");
    }
}

Animal animal = new Dog();
animal.identify(); // “dog”
```



Is this Allowed?

```
class Animal {
    final String name;

    public Animal(String name) {
        this.name = name;
    }

    public String identify() {
        return this.name;
    }
}
```

```
class Dog extends Animal {
    public Dog() {
        super("dog");
    }

    public String bark() {
        return "Woof!";
    }
}

Dog dog = new Dog();
dog.bark();    // ??

Animal animal = new Dog();
animal.bark(); // ??
```

Is this allowed?

```
class Animal {
    final String name;

    public Animal(String name) {
        this.name = name;
    }

    public String identify() {
        return this.name;
    }
}
```

```
class Dog extends Animal {
    public Dog() {
        super("dog");
    }

    public String bark() {
        return "Woof!";
    }
}

Dog dog = new Dog();
dog.bark();     // “Woof”

Animal animal = new Dog();
animal.bark(); // No such method
```

Is this Allowed?

```
class Animal {
    final String name;

    public Animal(String name) {
        this.name = name;
    }

    public String identify() {
        return this.name;
    }
}
```

```
class Dog extends Animal {
    public Dog() {
        super("dog");
    }

    public Animal identify() {
        return this;
    }
}

Animal animal = new Dog();
animal.identify(); // ??
```

Is this Allowed?

```
class Animal {
    final String name;

    public Animal(String name) {
        this.name = name;
    }

    public String identify() {
        return this.name;
    }
}
```

```
class Dog extends Animal {
    public Dog() {
        super("dog");
    }

    public Animal identify() {
        return this;
    }
}

Animal animal = new Dog();
animal.identify(); // compile time error!*
```

*This error is already raised when *declaring* the highlighted method.

Is this Allowed?

```
class Animal {
    final String name;

    public Animal(String name) {
        this.name = name;
    }

    public Animal identify() {
        return this;
    }
}
```

```
class Dog extends Animal {
    public Dog() {
        super("dog");
    }

    public Animal identify() {
        return this;
    }
}

Animal animal = new Dog();
animal.identify(); // This is fine!
```

“Can I inherit from this type?”

Yes.

“Should I inherit from this type?”

…that’s a different/design question.

“Should I inherit from this type?”

Behavioral Subtyping gives a more formal principle behind when extension should be considered.

Subclasses should satisfy the expectations of clients accessing subclass objects through references of superclass type, both syntactically and behaviorally.

- Subtypes inherit attributes, *behavior* from their parents
- Subtypes can add new *behavior*, properties

Behavioral Subtyping gives a more formal principle behind when extension should be considered.

```
Animal dog = new Dog();
```

Roughly:

- anything an Animal does, a Dog should do
- You should be able to use a subtype as if it was its parent
- But, dog may be more specific

The **Liskov substitution principle** is one more specific definition**:**
“Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.”

Barbara Liskov

Is this behavioral subtyping?

```
class Animal {

   final String name;

   public Animal(String name) {
       this.name = name;
   }

   public Animal me() {
       return this;
   }
}
```

```
class Dog extends Animal {

   public Dog() {
       super("dog");
   }

   public Dog me() {
       return this;
   }
}
```

Specifications/Invariants and Behavioral Subtyping.

Class invariants describe properties about an object's state/fields that should always hold before/after execution of public methods.

- Established by the constructor
- May be invalidated temporarily during method execution

There exists language extensions that allow you to annotated e.g., Java classes with invariants/specifications and will even check that they are maintained!

- We don't make you use them in this class.

Given that:

- Subtypes cannot have more restrictive (stronger) pre-conditions
  - That would prevent using the subclass as the parent-class
- But they can have stronger invariants and post-conditions!
  - And not just in terms of return type.

More concisely, subclasses should have:

Same or *stronger* **invariants** than super class

Same or *weaker* **preconditions** for all methods in super class

Same or *stronger* **postconditions** for all methods in super class

Is Square a behavioral subtype of Rectangle?

```
class Rectangle {

   int width;
   int height;

   public Rectangle(int width,
                    int height) {
       this.width = width;
       this.height = height;
   }
   public void scale(int factor) {
       width=width*factor;
       height=height*factor;
   }
}
```

```
public class Square extends Rectangle {

   public Square(int width) {
       super(width, width);
   }

}
```

How can we tell?

```
class Rectangle {
  //@ invariant height>0 && width>0;
   int width;
   int height;

   public Rectangle(int width,
                    int height) {
       this.width = width;
       this.height = height;
   }
   //@ requires factor > 0;
   public void scale(int factor) {
       width=width*factor;
       height=height*factor;
   }
}
```

```
public class Square extends Rectangle {
   //@ invariant height>0 && width>0;
   //@ invariant height==width;
   public Square(int width) {
       super(width, width);
   }

}
```

Is this behavioral subtyping?

```
class Rectangle {

   int width;
   int height;

   public Rectangle(int width,
                    int height) {
       this.width = width;
       this.height = height;
   }

   // Sets just the width.
   public void setWidth(int w) {
       this.width = w;
   }
}
```

```
public class Square extends Rectangle {

   public Square(int width) {
       super(width, width);
   }

}
```

It's technically fine, per our actual invariants, but the Square isn't a square anymore :-(

Is this better?

```
class Rectangle {

   int width;
   int height;

   public Rectangle(int width,
                    int height) {
       this.width = width;
       this.height = height;
   }

   //@ requires w > 0;
   //@ ensures width==neww && height==old.height;
   public void setWidth(int w) {
       this.width = w;
   }
}
```

```
public class Square extends Rectangle {

   public Square(int width) {
       super(width, width);
   }

   //@ requires w > 0;
   //@ ensures width==w && height==w;
     public void setWidth(int w) {
       this.width = w;
       this.height = w;
   }

}
```

Squares are square again, but they aren't behavioral subtypes of rectangles. :-(

Language enforcement of behavioral subtyping

The compiler won't always check this for you, clearly.

That said, there is some auto enforcement here, e.g., compiler enforced Java rules:

- Subtypes can add, but not remove methods
- Concrete class must implement all undefined methods
- Overriding method must return same type or subtype
- Overriding method must accept the same parameter types
- Overriding method may not throw additional exceptions

There are others language ways to enforce/restrict extension that we'll see moving forward (like `abstract` classes, can't be instantiated; `final` methods, can't be overridden (doesn't exist in TS)).

JS/TS has Classes

Since ES2016

```
class Square {
    width: number;

    constructor(width: number) {
        this.width = width;
    }

    printWidth() {
        console.log(this.width);
    }
}
```

```
let s1 = new Square(1);
let s2 = new Square(2);
s1.printWidth(); // 1
s2.printWidth(); // 2
```

Inheritance in JS/TS

```
class Animal {

    private name: string;

    constructor(name: string) {
        this.name = name;
    }
}
```

```
class Dog extends Animal {

    constructor() {
        super("dog");
    }
}


let dog = new Dog();
console.log(dog) // Dog { name: 'dog' }
```

TS has some language enforcement too…

```
class Number {
    value: number;

    constructor(value: number) {
        this.value = value;
    }
}
```

```
class LongerNumber extends Number {

    constructor(value: BigInt) {
        super(value);
    }
}
```

Inheritance vs. Subtyping

Inheritance is for polymorphism and code reuse

- Write code once and only once
- Superclass features implicitly available in subclass

```
class A extends B
```

Subtyping is for polymorphism

- Accessing objects the same way, but getting different behavior
- Subtype is substitutable for supertype

```
class A implements B
class A extends B
```

So why inheritance?

- We already have interfaces; why not:

```
interface Rectangle {
    getWidth(): number;
    getHeight(): number;
}

class Square implements Rectangle {
    width: number;
    constructor(width: number) {
        this.width = width;
    }
    getWidth(): number {
        return this.width * this.width;
    }
    getHeight(): number { return getWidth(); }
}
```

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

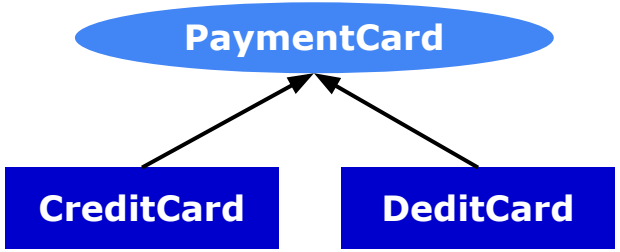
```java
public interface PaymentCard {
   String getCardHolderName();
   BigInteger getDigits();
   Date getExpiration();
   int getValue();
   boolean pay(int amount);
}

class DebitCard implements PaymentCard {
   private final String cardHolderName;
   private final BigInteger digits;
   private final Date expirationDate;
   private int debit;

   public DebitCard(String cardHolderName,
         BigInteger digits, Date expirationDate,
         int debit) {
      this.cardHolderName = cardHolderName;
      this.digits = digits;
      this.expirationDate = expirationDate;
      this.debit = debit;
   }
```

```java
// . . . continued . . .

@Override
public String getCardHolderName() {
    return this.cardHolderName;
}

@Override
public BigInteger getDigits() {
    return this.digits;
}

@Override
public Date getExpiration() {
    return this.expirationDate;
}
```

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

class CreditCard implements PaymentCard {
   private final String cardHolderName;
   private final BigInteger digits;
   private final Date expirationDate;
   private final int creditLimit;
   private int currentCredit;

   public CreditCard(String cardHolderName,
         BigInteger digits, Date expirationDate,
         int creditLimit, int credit) {
       this.cardHolderName = cardHolderName;
       this.digits = digits;
 // . . . continued . . .
```

```java
// . . . continued . . .
       this.expirationDate = expirationDate;
       this.creditLimit = creditLimit;
       this.currentCredit = credit;
   }

@Override
 public String getCardHolderName() {
       return this.cardHolderName;
 }

 @Override
 public BigInteger getDigits() {
       return this.digits;
 }

 @Override
 public Date getExpiration() {
       return this.expirationDate;
 }
```

Design option 1



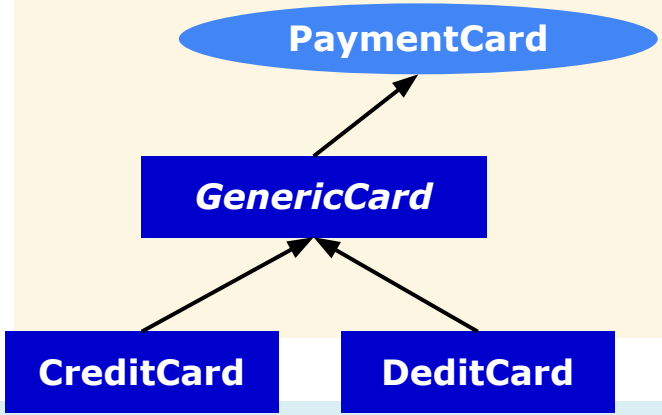

```
public interface PaymentCard {
   String getCardHolderName();
   BigInteger getDigits();
   Date getExpiration();
   int getValue();
   boolean pay(int amount);
}
```

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

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

Inheritance Facilitates Reuse!

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

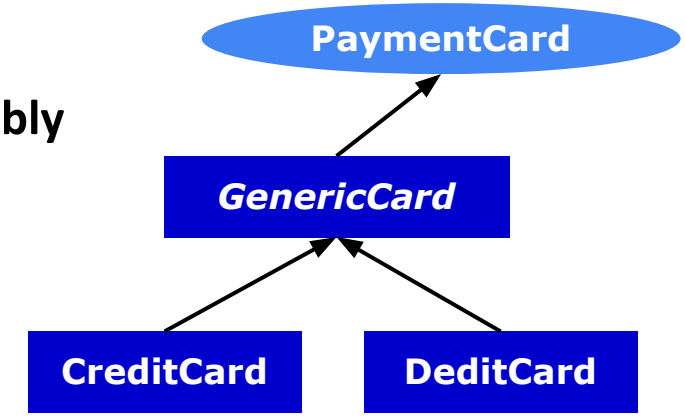



```java
class GenericCard implements PaymentCard {
   private final String cardHolderName;
   private final BigInteger digits;
   private final Date expirationDate;

   public GenericCard(String cardHolderName,
         BigInteger digits, Date expirationDate) {
      this.cardHolderName = cardHolderName;
      this.digits = digits;
      this.expirationDate = expirationDate;
   }

   @Override
   public String getCardHolderName() {
      return this.cardHolderName;
   }
```

Design option 2

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

Alternative to inheritance: *composition + delegation.*

- When classes relate closely, it can be nice to share functionality via inheritance.
- However, inheritance is not the only way!

Alternative: *composition + delegation*

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`

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:

```
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`

Payment card with delegation.

```
public interface PaymentCard {
   CardData getCardData();
   int getValue();
   boolean pay(int amount);
}
```

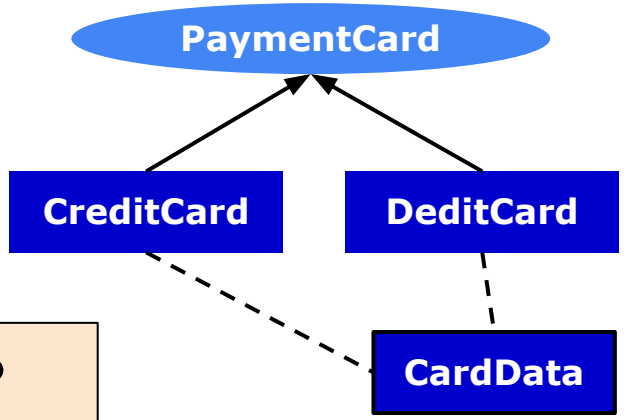
Payment card with delegation.

```
public interface PaymentCard {
    CardData getCardData();
    int getValue();
    boolean pay(int amount);
}

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

    public CardData(String cardHolderName,
         BigInteger digits, Date expirationDate) {
        this.cardHolderName = cardHolderName;
        this.digits = digits;
        this.expirationDate = expirationDate;
    }

    @Override
    public String getCardHolderName() {
        return this.cardHolderName;
    }
   // . . . other getters . . .
}
```

Payment card with delegation.

```
public interface PaymentCard {
   CardData getCardData();
   int getValue();
   boolean pay(int amount);
}

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

   public CardData(String cardHolderName,
        BigInteger digits, Date expirationDate) {
       this.cardHolderName = cardHolderName;
       this.digits = digits;
       this.expirationDate = expirationDate;
   }
   @Override
   public String getCardHolderName() {
       return this.cardHolderName;
   }
  // . . . other getters . . .
}
class CreditCard implements PaymentCard {
    private CardData cardData = new(…);
    public BigInteger getDigits() {
        return cardData.getDigits();
    }
    public boolean pay(int amount) {
      …
}
class DebitCard implements PaymentCard {
      …
}
```

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() {…}
}
```

Is this better?

```
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

- Inheritance can enable **substantial** reuse when strong coupling is reasonable
  - Sometimes a natural fit for reuse -- look for “is-a” relationships.
  - Does not mean “no delegation”
- That said, good design typically favors composition + delegation
  - Enables reuse, encapsulation by programming against interfaces
  - Delegation supports information hiding; inheritance violates it
  - Usually results in more *testable* code.
  - Composition facilitates adding multiple behaviors
    - Multiple inheritance exists, but gets messy

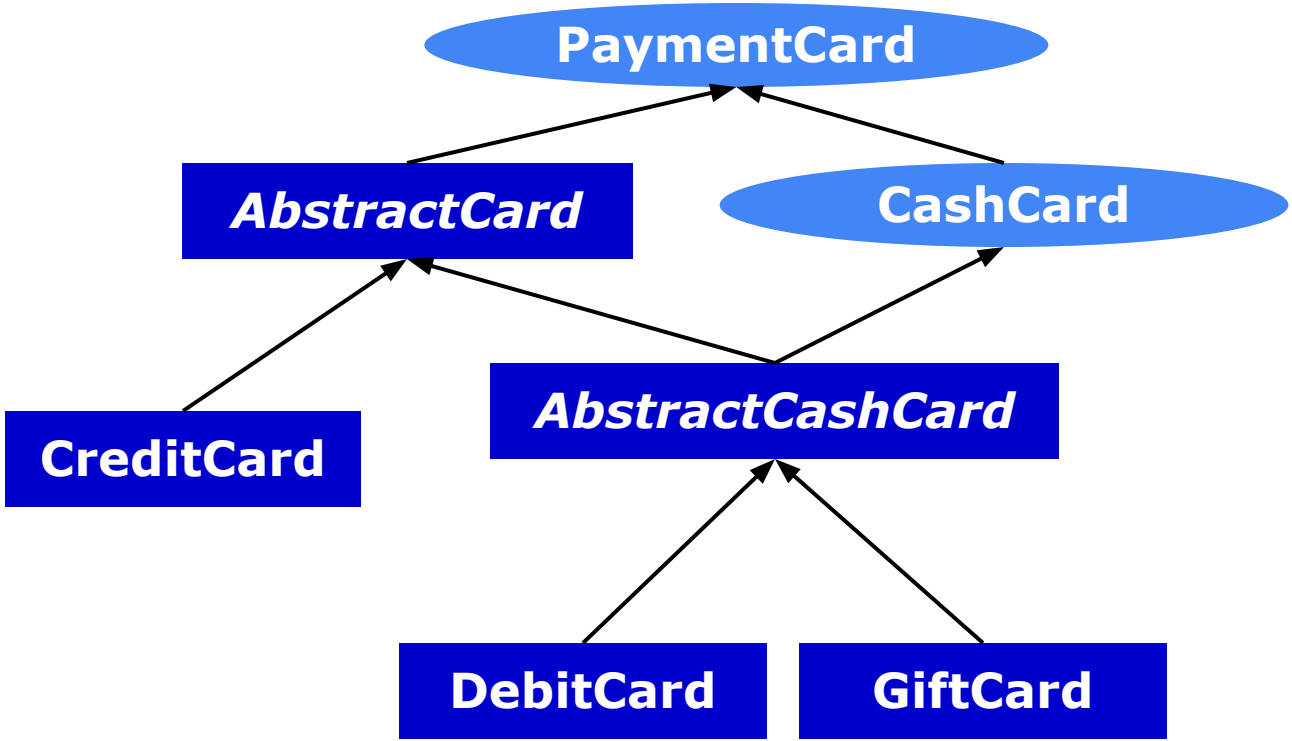
Designing with Inheritance in Mind

- Try to avoid it when composition+delegation is available
- Document contracts for inheritance
  - The compiler won't inforce all invariants
  - Document requirements for overriding methods
  - Test with subclasses!
- Enforce or prohibit inheritance where possible
  - In Java: `final` & `abstract`

Interface Inheritance exists (both Java and TS)

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

interface CashCard extends PaymentCard {
   boolean pay(int amount);
   int getBalance();
   void addCash(int amount);
}
```

Payment Card Hierarchy (one example)



Summary so far

- Inheritance is a powerful tool
  - …That takes coupling to the extreme
  - And deserves careful consideration, to be used/designed well.
- Subtyping and inheritance are related, but not the same
- Composition & Delegation are often the right tools instead
  - Not mutually exclusive

Going back to where inheritance is good, actually…

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

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

Going back to where inheritance is good, actually…

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

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

```
class DebitCard implements PaymentCard {
   private int balance;
   private int fee;
   public DebitCard(int balance,
                    int transactionFee) {
       this.balance = balance;
       this.fee = fee;
   }

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

   public void addCash(int amount) {
```

Opportunity to reuse even more

```
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;
           chargeFee();
           return true;
       }
       return false;
   }
   abstract void chargeFee();
}
```

```
class GiftCard extends AbstractCashCard {
   @Override
   void chargeFee() {
      return; // Do nothing.
   }
}
```

'Pay' is already implemented!

Must be implemented

Opportunity to reuse even more

```
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;
           chargeFee();
           return true;
       }
       return false;
   }
   abstract void chargeFee();
}
```

```
class GiftCard extends AbstractCashCard {
   @Override
   void chargeFee() {
      return; // Do nothing.
   }
}
```

```
class DebitCard extends AbstractCashCard {
   @Override
   void chargeFee() {
      this.balance -= this.fee;
   }
}
```

This is the Template Method Design Pattern!

```
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;
           chargeFee();
           return true;
       }
       return false;
   }
   abstract void chargeFee();
}
```

```
class GiftCard extends AbstractCashCard {
   @Override
   void chargeFee() {
      return; // Do nothing.
   }
}
```

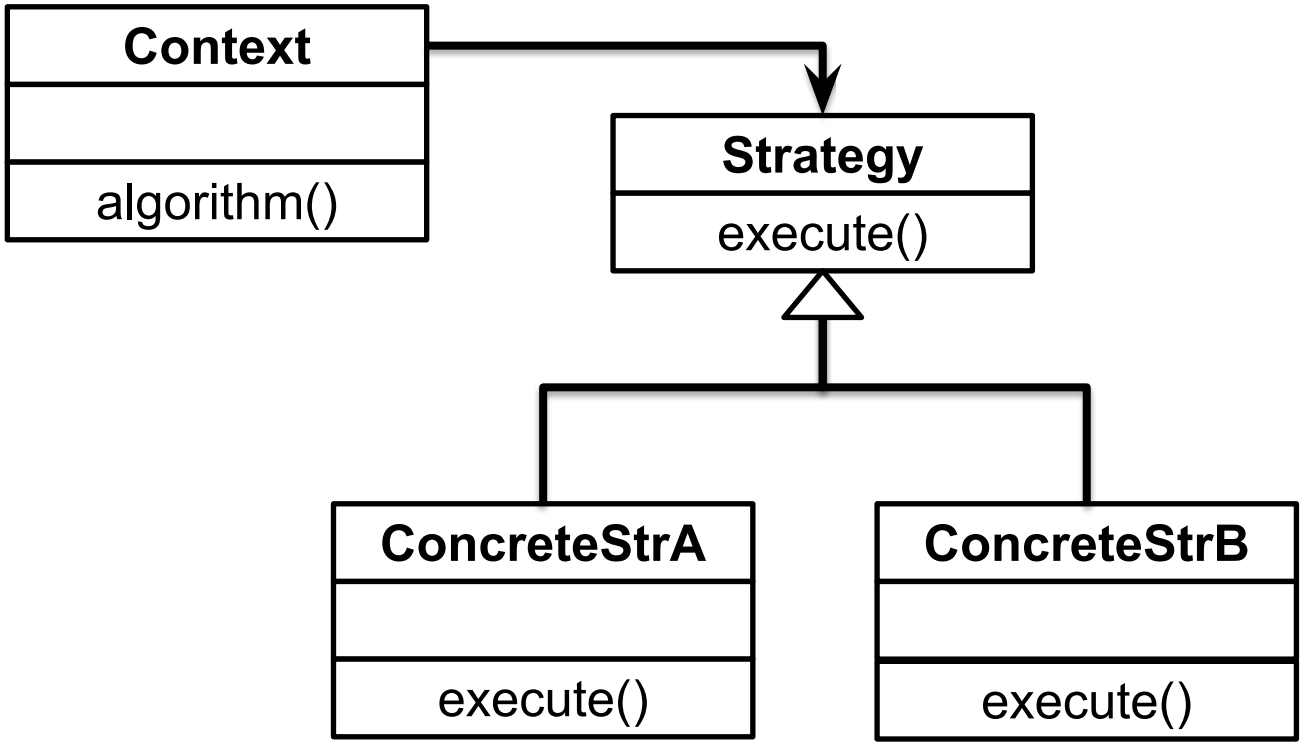
```
class DebitCard extends AbstractCashCard {
   @Override
   void chargeFee() {
      this.balance -= this.fee;
   }
}
```

Design Tradeoffs?

Template Method vs. Strategy Pattern

- Template method uses *inheritance* to vary part of an algorithm
  - Template method implemented in supertype, primitive operations implemented in subtypes
- Strategy pattern uses *delegation* to vary the entire algorithm
  - Strategy objects are reusable across multiple classes
  - Multiple strategy objects are possible per class
  - Where have we seen this?

Strategy Pattern in UML.



Supplement 1: Language/Implementation Details

Dynamic Dispatch

In Java:

- (Compile time) Determine which class to look in
- (Compile time) Determine method signature to be executed
  - Find all accessible, applicable methods
  - Select most specific matching method
- (Run time) Determine dynamic class of the receiver
- (Run time) From dynamic class, determine method to invoke
  - Execute method with the same signature found in step 2 (from dynamic class or one of its supertypes)

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

Supplement 2: More behavioral subtype examples

Is Car a behavioral subtype of Vehicle?

```
abstract class Vehicle {
        int speed, limit;

        //@ invariant speed < limit;









        //@ requires speed != 0;
        //@ ensures speed < \old(speed)
        void brake();
}
```

```
class Car extends Vehicle {
        int fuel;
        boolean engineOn;
        //@ invariant speed < limit;
        //@ invariant fuel >= 0;

        //@ requires fuel > 0 && !engineOn;
        //@ ensures engineOn;
        void start() { … }

        void accelerate() { … }

        //@ requires speed != 0;
        //@ ensures speed < \old(speed)
        void brake() { … }
}
```

Car is a behavioral subtype of Vehicle

```
abstract class Vehicle {
          int speed, limit;

          //@ invariant speed < limit;









          //@ requires speed != 0;
          //@ ensures speed < \old(speed)
          void brake();
}
```

```
class Car extends Vehicle {
          int fuel;
          boolean engineOn;
          //@ invariant speed < limit;
          //@ invariant fuel >= 0;

          //@ requires fuel > 0 && !engineOn;
          //@ ensures engineOn;
          void start() { … }

          void accelerate() { … }

          //@ requires speed != 0;
          //@ ensures speed < \old(speed)
          void brake() { … }
}
```

- **Subclass fulfills the same invariants (and additional ones)**
- **Overridden method `brake` has the same pre and postconditions**

Is Hybrid a behavioral subtype of Car?

```
class Car extends Vehicle {
      int fuel;
      boolean engineOn;
      //@ invariant fuel >= 0;

      //@ requires fuel > 0 && !engineOn;
      //@ ensures engineOn;
      void start() { … }

      void accelerate() { … }

      //@ requires speed != 0;
      //@ ensures speed < old(speed)
      void brake() { … }
}
```

```
class Hybrid extends Car {
      int charge;
      //@ invariant charge >= 0;

      //@ requires (charge > 0 || fuel > 0)
                              && !engineOn;
      //@ ensures engineOn;
      void start() { … }

      void accelerate() { … }

      //@ requires speed != 0;
      //@ ensures speed < \old(speed)
      //@ ensures charge > \old(charge)
      void brake() { … }
}
```

Hybrid is a behavioral subtype of Car

```
class Car extends Vehicle {
      int fuel;
      boolean engineOn;
      //@ invariant fuel >= 0;

      //@ requires fuel > 0 && !engineOn;
      //@ ensures engineOn;
      void start() { … }

      void accelerate() { … }

      //@ requires speed != 0;
      //@ ensures speed < old(speed)
      void brake() { … }
}
```

```
class Hybrid extends Car {
      int charge;
      //@ invariant charge >= 0;

      //@ requires (charge > 0 || fuel > 0)
                              && !engineOn;
      //@ ensures engineOn;
      void start() { … }

      void accelerate() { … }

      //@ requires speed != 0;
      //@ ensures speed < \old(speed)
      //@ ensures charge > \old(charge)
      void brake() { … }
}
```

- **Subclass fulfills the same invariants (and additional ones)**
- **Overridden method `start` has weaker precondition**
- **Overridden method `brake` has stronger postcondition**