Principles of Software Construction: Objects, Design, and Concurrency

Inheritance and delegation

Christian Kästner **Vincent Hellendoorn**

Carnegie Mellon University School of Computer Science

institute for SOFTWARE RESEARCH

Notes on HW2

- Common over-testing:
  - MostMistakesFirstOrganizer: not stable
  - InMemoryCardOrganizer: getAllCards order not guaranteed
    - A lot of tests relied on the cards-file → CardLoader loop to get FlashCards
- Common under-testing:
  - FlashCard: reference answer trimming
  - Repeating: more observations than limit
  - Non-repeating organizer: test for a single ‘false’ answer

Quiz

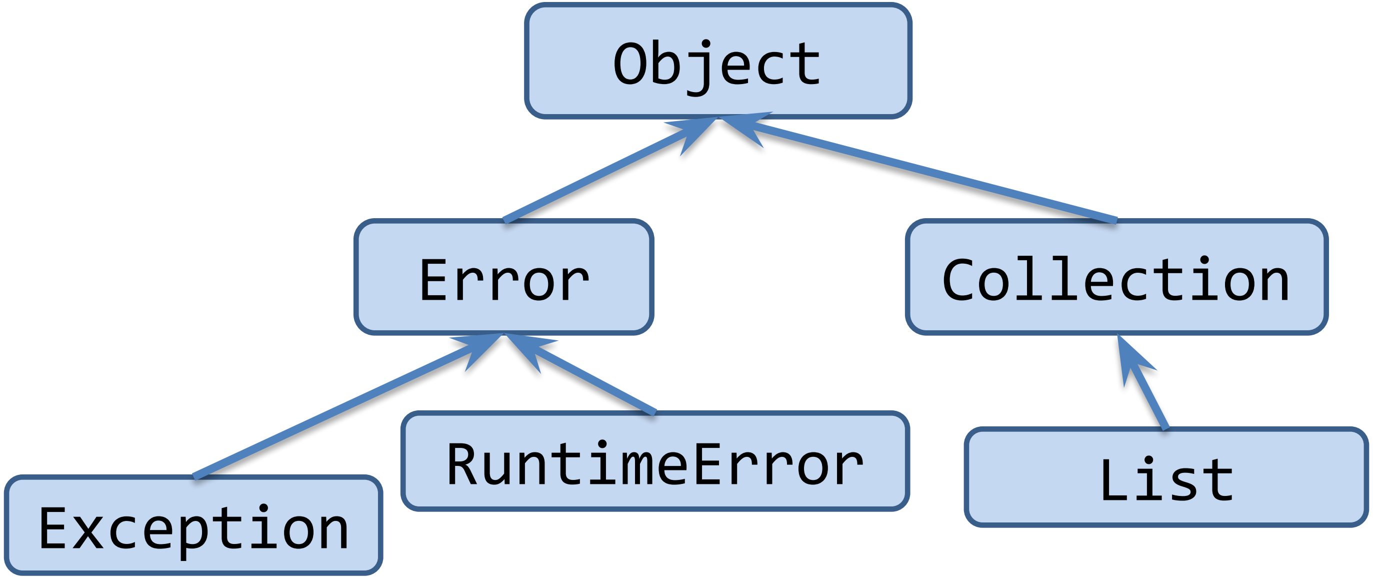
https://rb.gy/ffljay

Today

- Class Hierarchies
- Behavioral Subtyping
- Design Goals
  - Template Method Pattern
  - Reuse; relation to coupling
  - When to use inheritance, delegation
- A bit on refactoring

Class Hierarchy

In Java:



Class Hierarchy

Some 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 that
- Primitive types are not in the class hierarchy

Chime In

What does it mean to “extend” a class?

Inheritance enables Extension

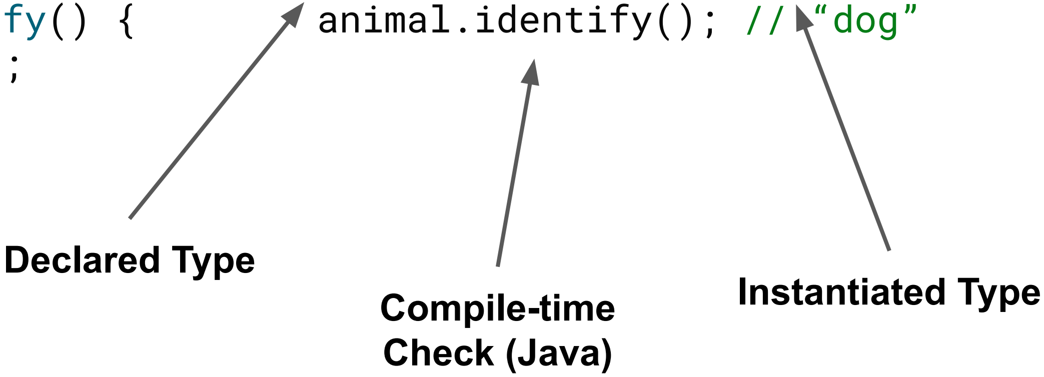
```
class Animal {
   final String name;

   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”
```

Inheritance enables Extension

```
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”
```



Declared Type

Compile-time Check (Java)

Instantiated Type

Is this Allowed?

```java
class Animal {
   final String name;

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

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

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

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

Behavioral Subtyping

- Formalizes notion of extension

The **Liskov substitution principle:**
“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

Behavioral Subtyping

- Formalizes notion of extension

```
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:**
“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

Behavioral Subtyping

```
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
```

Behavioral Subtyping

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

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;
   }
}
```

Is this behavioral subtyping?

```
class Number {
    value: number;

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

```
class LongerNumber extends Number {

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

Behavioral Subtyping

- Subtypes cannot have more restrictive (stronger) *pre-conditions*
    - That would prevent using the subclass as the parent-class
- But they can have *stronger post-conditions*
    - Not just in terms of return type

Is this behavioral subtyping?

```java
class Rectangle {

   int width;
   int height;

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

```java
public class Square extends Rectangle {

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

Is this behavioral subtyping?

```java
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;
   }
}
```

```java
public class Square extends Rectangle {

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

   public void setWidth(int w) {
       this.width = w;
       this.height = w;
   }
}
```

Behavioral Subtyping

- The compiler won’t always check this for you
- There are many ways to enforce/restrict extension
    - `abstract` classes, can’t be instantiated
        - But can have `abstract` methods that must be overridden
    - `final` methods, can’t be overridden
        - Does not exist in TS
    - Heavily language-specific

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
```

JS/TS has Classes

Since ES2016, but…

```
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

Square.prototype.printWidth = function () {
    console.log('nope!');
}

s1.printWidth(); // 'nope'
s2.printWidth(); // 'nope'
```

JS/TS has Classes

Since ES2016, but…

- No notion of static, private
  - TypeScript introduces keywords for these (and more).
- The definition of ‘this’ is tricky
  - Especially with inheritance
  - For those interested: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this

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' }
```

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(); }
}
```

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?

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

```java
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;
   }
```

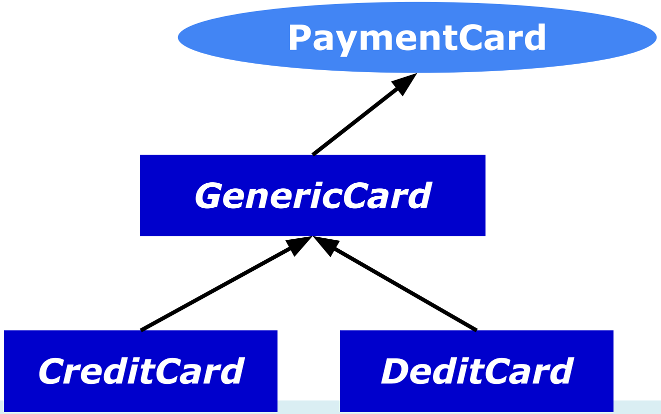
So why inheritance?

```
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;
     this.expirationDate = expirationDate;
     this.creditLimit = creditLimit;
     this.currentCredit = credit;
  }
```

Inheritance Facilitates Reuse

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



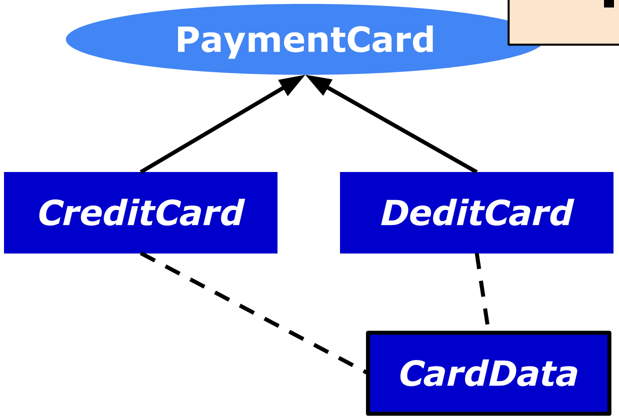
```
class GenericCard implements PaymentCard {
   private final String cardHolderName;
   private final BigInteger digits;
   private final Date expirationDate;

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

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

Inheritance Facilitates Reuse

- When classes relate closely, it is nice to share functionality
  - That doesn’t *necessitate* inheritance

Reuse does not Require Inheritance

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



Is this better?

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

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

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

Reuse does not Require Inheritance

- When classes relate closely, it is nice to share functionality
  - That doesn’t *necessitate* inheritance
- But inheritance can enable **substantial** reuse
  - When strong coupling is reasonable

Template Method Pattern

```java
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;
   }
}
```

```java
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;
   }
```

Template Method Pattern

```java
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();
}
```

Must be implemented

Template Method 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.
   }
}
```

‘Pay’ is already implemented

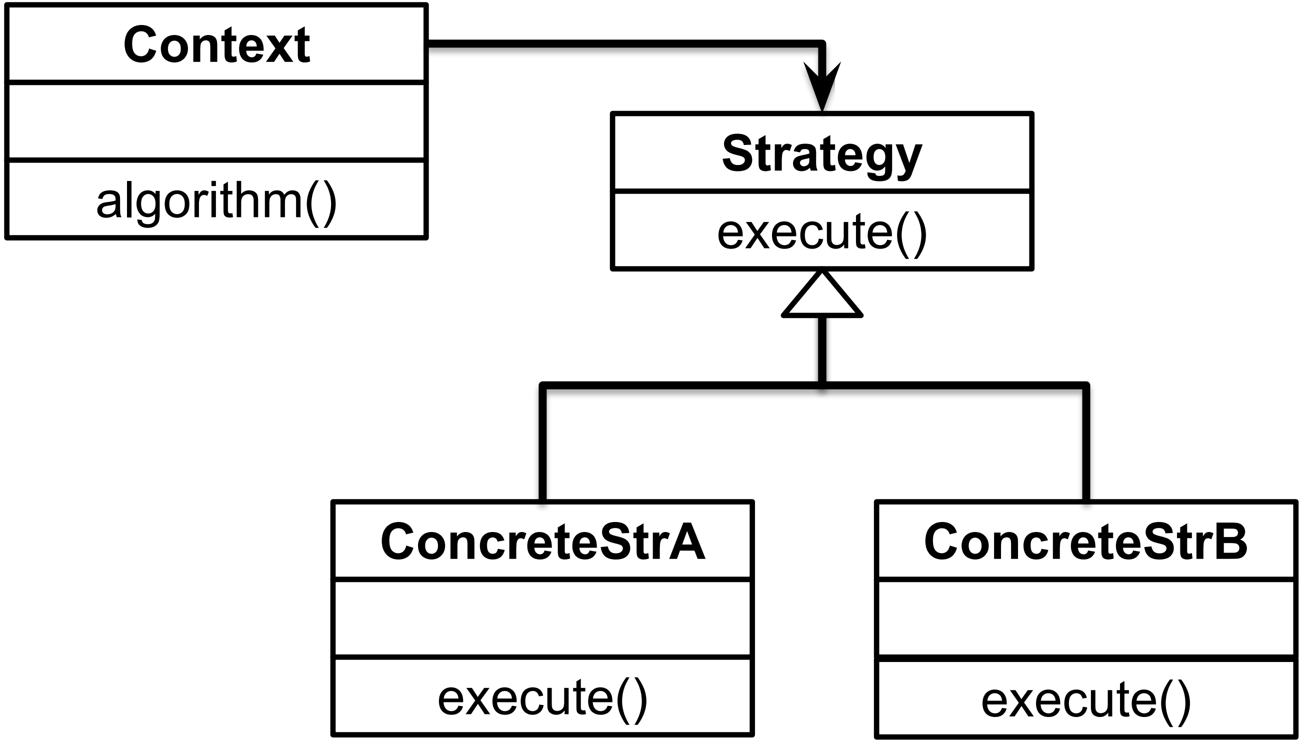
Template Method Pattern

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

   public boolean pay
       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?

Strategy Pattern



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

Inheritance vs. Composition + Delegation

- A lot of good design uses composition + delegation
  - Enables reuse, encapsulation by programming against interfaces
  - Composition facilitates adding multiple behaviors
    - Multiple inheritance exists, but gets messy
- Inheritance implies strong coupling
  - Sometimes a natural fit for reuse -- look for “is-a” relationships.
  - Much reduced encapsulation
  - Does not mean “no delegation”

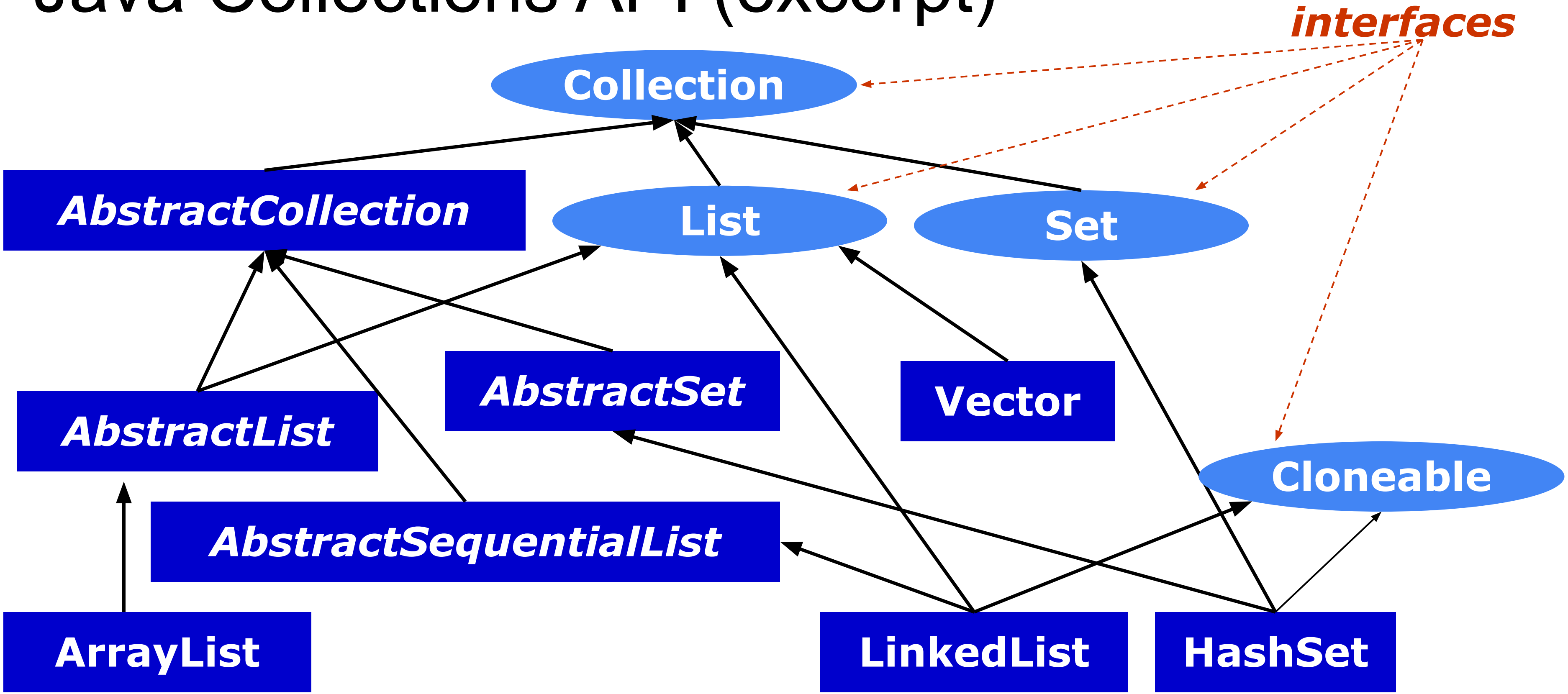
Inheritance vs. Composition + Delegation

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

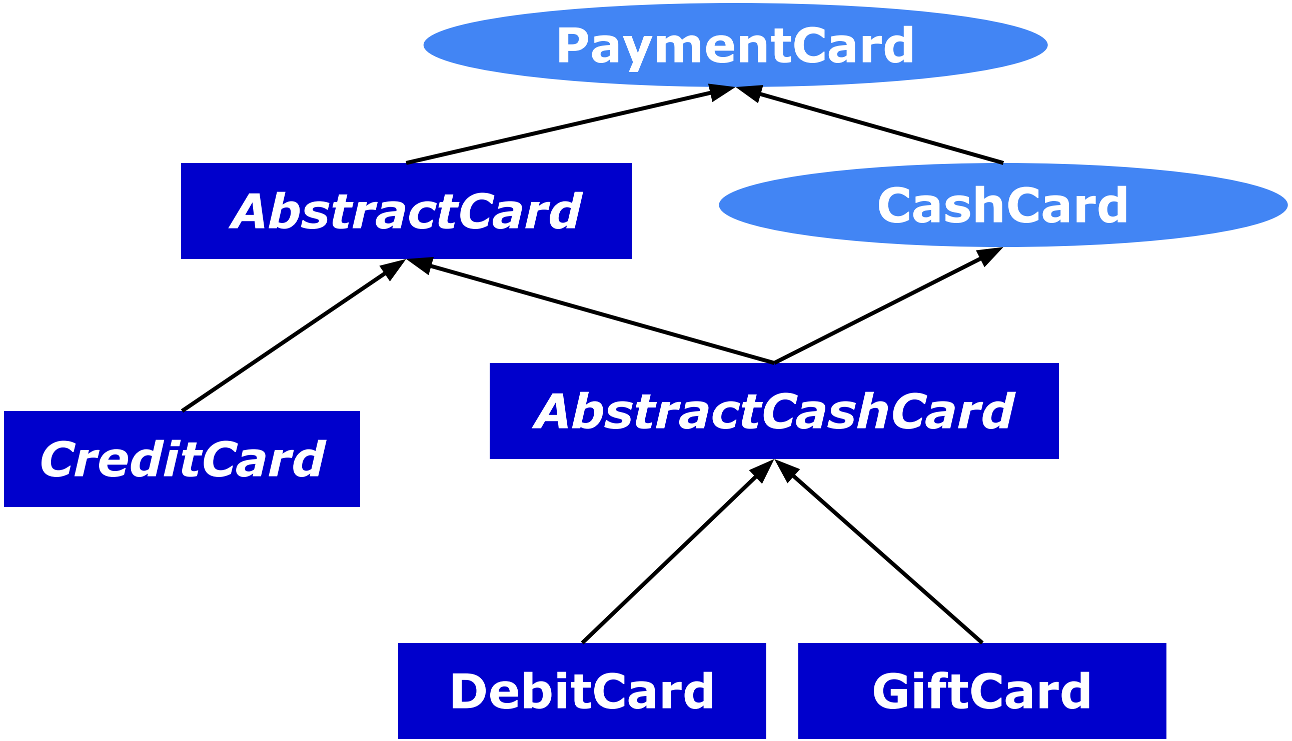
Interface Inheritance

```java
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);
}
```

Java Collections API (excerpt)



Payment Card Hierarchy (example)



Payment Card with Inheritance

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

```
abstract class AbstractCard implements PaymentCard {
    private final String cardHolderName;
    private final BigInteger digits;
    private final Date expirationDate;

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

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

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)

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

Inheritance Reuse w/o Inversion of Control

```java
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;
   }
}
```

```java
class DebitCard extends AbstractCashCard

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

Works because of the order of invocation.
But is it good?

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

Designing with Inheritance in Mind

- Try to avoid it when composition+delegation is available
  - Delegation reduces coupling
  - Inheritance limits *information hiding*
- Document contracts for inheritance
  - The compiler won't inforce all invariants
- Enforce or prohibit inheritance where possible
  - In Java: `final` & `abstract`

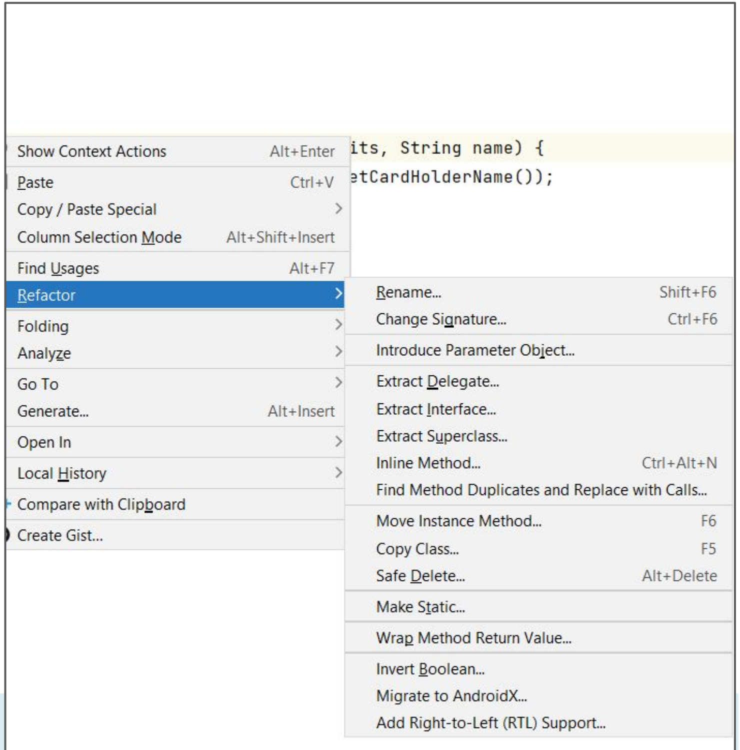
Refactoring

Refactoring

- Any functionality-preserving restructuring
  - Typically automated by IDE
  - Ideas?

Refactoring

- Rename class, method, variable to something not in-scope
- Extract method/inline method
- Extract interface
- Move method (up, down, laterally)
- Replace duplicates



```
its, String name) {
etCardHolderName());
```

Show Context Actions Alt+Enter
Paste Ctrl+V
Copy / Paste Special
Column Selection Mode Alt+Shift+Insert
Find Usages Alt+F7
Refactor
Folding
Analyze
Go To
Generate... Alt+Insert
Open In
Local History
Compare with Clipboard
Create Gist...
Rename... Shift+F6
Change Signature... Ctrl+F6
Introduce Parameter Object...
Extract Delegate...
Extract Interface...
Extract Superclass...
Inline Method... Ctrl+Alt+N
Find Method Duplicates and Replace with Calls...
Move Instance Method... F6
Copy Class... F5
Safe Delete... Alt+Delete
Make Static...
Wrap Method Return Value...
Invert Boolean...
Migrate to AndroidX...
Add Right-to-Left (RTL) Support...

Refactoring and Anti-Patterns

- Often, all the functionality is correct, but the organization is bad
    - High coupling, high redundancy, poor cohesion, god classes, …
- Refactoring is the principal tool to improve structure
    - Automated refactorings even guarantee correctness
        - But you can’t always count on those being right
    - A series of refactorings is usually enough to introduce design patterns

Refactoring and Anti-Patterns

- Often, all the functionality is correct, but the organization is bad
    - High coupling, high redundancy, poor cohesion, god classes, …
- Refactoring is the principal tool to improve structure
    - Automated refactorings even guarantee correctness
        - But you can’t always count on those being right
    - A series of refactorings is usually enough to introduce design patterns
- HW4 involves analyzing such a system and making primarily refactoring changes
    - “primarily”, because sometimes you do need to alter things slightly.

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