Principles of Software Construction: Objects, Design, and Concurrency

Designing classes

Delegation and inheritance and behavioral subtyping (oh my)

Josh Bloch          Charlie Garrod
Administrivia

- Homework 2 due Thursday
Key concepts from Thursday...
What is a contract?

• Agreement between an object and its user
• Includes
  – Method signature (type specifications)
  – Functionality and correctness expectations
  – Performance expectations

• **What the method does**, not how it does it
  – Interface *(API)*, not implementation
The == operator vs. equals method

- For primitives you *must* use ==
- For object reference types
  - The == operator provides *identity semantics*
    - Exactly as implemented by Object.equals
    - Even if Object.equals has been overridden
    - This is seldom what you want!
  - you should (almost) always use .equals
    - Using == on an object reference is a bad smell in code
      
      if (input == "yes") // A bug!!!
The equals contract in English

- **Reflexive** – every object is equal to itself
- **Symmetric** – if `a.equals(b)` then `b.equals(a)`
- **Transitive** – if `a.equals(b)` and `b.equals(c)`, then `a.equals(c)`
- **Consistent** – equal objects stay equal unless mutated
- “**Non-null**” – `a.equals(null)` returns false
- Taken together these ensure that equals is a global *equivalence relation* over all objects
Learning goals for today

• Be able to explain inheritance and delegation
• Apply inheritance and delegation appropriately for reuse
  – Understand their tradeoffs
• Behavioral subtyping and implications for specification and testing
Today: Class-level reuse with delegation and inheritance

- Delegation
- Inheritance
  - Java-specific details for inheritance
- Behavioral subtyping: Liskov's Substitution Principle
- Thursday: Design patterns for improved class-level reuse
- Later in the course:
  - System-level reuse with libraries and frameworks
## The promise of reuse:

<table>
<thead>
<tr>
<th>Products</th>
<th>Cost (With reuse)</th>
<th>Cost (Without reuse)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
COMPOSITION AND DELEGATION
Recall our earlier sorting example:

Version A:

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

Version B':

```java
interface Comparator {
    boolean compare(int i, int j);
}
final Comparator ASCENDING = (i, j) -> i < j;
final Comparator DESCENDING = (i, j) -> i > j;

static void sort(int[] list, Comparator cmp) {
    ...
    boolean mustSwap =
        cmp.compare(list[i], list[j]);
    ...
}
```
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 Comparator
- Judicious delegation enables code reuse

```java
interface Comparator {
    boolean compare(int i, int j);
}
final Comparator ASCENDING = (i, j) -> i < j;
final Comparator DESCENDING = (i, j) -> i > j;

static void sort(int[] list, Comparator cmp) {
    ...
    boolean mustSwap =
        cmp.compare(list[i], list[j]);
    ...
}
```
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 Comparator
• Judicious delegation enables code reuse
  – Sorter can be reused with arbitrary sort orders
  – Comparators can be reused with arbitrary client code that needs to compare integers
Using delegation to extend functionality

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

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

• 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 Comparator
IMPLEMENTATION INHERITANCE AND ABSTRACT CLASSES
Variation in the real world: types of bank accounts

```java
public interface CheckingAccount {
    public long getBalance();
    public void deposit(long amount);
    public boolean withdraw(long amount);
    public boolean transfer(long amount, Account<?> target);
    public long getFee();
}

public interface SavingsAccount {
    public long getBalance();
    public void deposit(long amount);
    public boolean withdraw(long amount);
    public boolean transfer(long amount, Account<?> target);
    public double getInterestRate();
}
```
Interface inheritance for an account type hierarchy

```java
public interface Account {
    public long getBalance();
    public void deposit(long amount);
    public boolean withdraw(long amount);
    public boolean transfer(long amount, Account target);
    public void monthlyAdjustment();
}

public interface CheckingAccount extends Account {
    public long getFee();
}

public interface SavingsAccount extends Account {
    public double getInterestRate();
}

public interface InterestCheckingAccount extends CheckingAccount, SavingsAccount {
}
```
The power of object-oriented interfaces

- **Subtype polymorphism**
  - Different kinds of objects can be treated uniformly by client code
  - Each object behaves according to its type
    - e.g., if you add new kind of account, client code does not change:

```java
If today is the last day of the month:
    For each acct in allAccounts:
        acct.monthlyAdjustment();
```
Implementation inheritance for code reuse

```java
public abstract class AbstractAccount
    implements Account {
    protected long balance = 0;
    public long getBalance() {
        return balance;
    }
    abstract public void monthlyAdjustment();
    // other methods...
}

public class CheckingAccountImpl
    extends AbstractAccount
    implements CheckingAccount {
    public void monthlyAdjustment() {
        balance -= getFee();
    }
    public long getFee() { ... }
}
```
Implementation inheritance for code reuse

```java
public abstract class AbstractAccount
    implements Account {
    protected long balance = 0;
    public long getBalance() {
        return balance;
    }
    abstract public void monthlyAdjustment();
    // other methods...
}

public class CheckingAccountImpl
    extends AbstractAccount
    implements CheckingAccount {
    public void monthlyAdjustment() {
        balance -= getFee();
    }
    public long getFee() { ... }
}
```

- An abstract class is missing the implementation of one or more methods.
- Protected elements are visible in subclasses.
- An abstract method is left to be implemented in a subclass.
- No need to define `getBalance()` – the code is inherited from `AbstractAccount`.
Inheritance: a glimpse at the hierarchy

- Examples from Java
  - `java.lang.Object`
  - Collections library
Java Collections API (excerpt)
Benefits of inheritance

• Reuse of code
• Modeling flexibility
• A Java aside:
  – Each class can directly extend only one parent class
  – A class can implement multiple interfaces
Inheritance and subtyping

- Inheritance is for code reuse
  - Write code once and only once
  - Superclass features implicitly available in subclass

- Subtyping is for polymorphism
  - Accessing objects the same way, but getting different behavior
  - Subtype is substitutable for supertype
Typical roles for interfaces and classes

• An interface defines expectations / commitments for clients
• A class fulfills the expectations of an interface
  – An abstract class is a convenient hybrid
  – A subclass specializes a class's implementation
public abstract class AbstractAccount implements Account {
    protected long balance = 0;
    public boolean withdraw(long amount) {
        // withdraws money from account (code not shown)
        return true; // example return value
    }
}

public class ExpensiveCheckingAccountImpl extends AbstractAccount implements CheckingAccount {
    public boolean withdraw(long amount) {
        balance -= HUGE_ATM_FEE;
        boolean success = super.withdraw(amount);
        if (!success)
            balance += HUGE_ATM_FEE;
        return success;
    }
}
Java details: constructors with this and super

```java
public class CheckingAccountImpl extends AbstractAccount implements CheckingAccount {

    private long fee;

    public CheckingAccountImpl(long initialBalance, long fee) {
        super(initialBalance);
        this.fee = fee;
    }

    public CheckingAccountImpl(long initialBalance) {
        this(initialBalance, 500);
    }

    /* other methods... */
}
```

Invokes another constructor in this same class

Invokes a constructor of the superclass. Must be the first statement of the constructor.
Java 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 { ...
Note: type-casting in Java

- Sometimes you want a different type than you have
  - e.g.,
    
    ```java
    double pi = 3.14;
    int indianaPi = (int) pi;
    ```

- Useful if you know you have a more specific subtype:
  - e.g.,
    ```java
    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
Note: instanceof

- Operator that tests whether an object is of a given class
  
  ```java
  public void doSomething(Account acct) {
      long adj = 0;
      if (acct instanceof CheckingAccount) {
          checkingAcct = (CheckingAccount) acct;
          adj = checkingAcct.getFee();
      } else if (acct instanceof SavingsAccount) {
          savingsAcct = (SavingsAccount) acct;
          adj = savingsAcct.getInterest();
      }
      ...
  }
  ```

- Advice: avoid instanceof if possible
  - Never(?) use instanceof in a superclass to check type against subclass

Warning: This code is bad.
Java details: Dynamic method dispatch

1. (Compile time) Determine which class to look in
2. (Compile time) Determine method signature to be executed
   1. Find all accessible, applicable methods
   2. Select most specific matching method
Java details: Dynamic method dispatch

1. (Compile time) Determine which class to look in
2. (Compile time) Determine method signature to be executed
   1. Find all accessible, applicable methods
   2. Select most specific matching method
3. (Run time) Determine dynamic class of the receiver
4. (Run time) From dynamic class, locate method to invoke
   1. Look for method with the same signature found in step 2
   2. Otherwise search in superclass and etc.
Design with inheritance (or not)

• Favor composition over inheritance
  – Inheritance violates information hiding

• Design and document for inheritance, or prohibit it
  – Document requirements for overriding any method
BEHAVIORAL SUBTYPING
Behavioral subtyping

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

- e.g., Compiler-enforced rules in Java:
  - 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
Behavioral subtyping

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

• e.g., Compiler-enforced rules in Java:
  – 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

• Also applies to specified behavior:
  – Same or stronger invariants
  – Same or stronger postconditions for all methods
  – Same or weaker preconditions for all methods

This is called the Liskov Substitution Principle.
Behavioral subtyping (Liskov Substitution Principle)

abstract class Vehicle {
    int speed, limit;
    //@ invariant speed < limit;
    void brake();
//@ requires speed != 0;
//@ ensures speed < \old(speed)
}

class Car extends Vehicle {
    int fuel;
    boolean engineOn;
    //@ invariant speed < limit;
    //@ invariant fuel >= 0;
    //@ requires fuel > 0 && !engineOn;
    //@ ensures engineOn;
    void start() {
        ...}
    void accelerate() {
        ...
    }
    void brake() {
        ...
    }
//@ requires speed != 0;
//@ ensures speed < \old(speed)
}

Subclass fulfills the same invariants (and additional ones)
Overridden method has the same pre and postconditions
Behavioral subtyping (Liskov Substitution Principle)

```java
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
Behavioral subtyping (Liskov Substitution Principle)

class Rectangle {
    int h, w;

    Rectangle(int h, int w) {
        this.h = h; this.w = w;
    }

    // methods
}

class Square extends Rectangle {
    Square(int w) {
        super(w, w);
    }
}

Is this Square a behavioral subtype of Rectangle?
Is this Square a behavioral subtype of Rectangle? (Yes.)
Is this Square a behavioral subtype of Rectangle?
Behavioral subtyping (Liskov Substitution Principle)

class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;

    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }

    //methods
}

class Square extends Rectangle {
    //@ invariant h>0 && w>0;
    //@ invariant h==w;
    Square(int w) {
        super(w, w);
    }
}

Is this Square a behavioral subtype of Rectangle? (Yes.)
Behavioral subtyping (Liskov Substitution Principle)

```java
class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;
    Rectangle(int h, int w) {
        this.h = h;
        this.w = w;
    }
    //@ requires factor > 0;
    void scale(int factor) {
        w = w * factor;
        h = h * factor;
    }
}

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

Is this Square a behavioral subtype of Rectangle?
Behavioral subtyping (Liskov Substitution Principle)

```java
class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;

    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }

    //@ requires factor > 0;
    void scale(int factor) {
        w=w*factor;
        h=h*factor;
    }
}

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

Is this Square a behavioral subtype of Rectangle? (Yes.)
Behavioral subtyping (Liskov Substitution Principle)

class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;

    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }

    //@ requires factor > 0;
    void scale(int factor) {
        w=w*factor;
        h=h*factor;
    }

    //@ requires neww > 0;
    void setWidth(int neww) {
        w=neww;
    }
}

class Square extends Rectangle {
    //@ invariant h>0 && w>0;
    //@ invariant h==w;
    Square(int w) {
        super(w, w);
    }
}

Is this Square a behavioral subtype of Rectangle?
Behavioral subtyping (Liskov Substitution Principle)

```java
class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;

    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }

    void scale(int factor) {
        w=w*factor;
        h=h*factor;
    }

    void setWidth(int neww) {
        w=neww;
    }
}

class Square extends Rectangle {
    //@ invariant h>0 && w>0;
    //@ invariant h==w;
    Square(int w) {
        super(w, w);
    }
}

class GraphicProgram {
    void scaleW(Rectangle r, int factor) {
        r.setWidth(r.getWidth() * factor);
    }
}
```

Yes? (But the Square is not actually a square...)

← Invalidates stronger invariant (w==h) in subclass
Summary: Designing reusable classes

- Reusable implementations with simple, clear contracts
- Inheritance for reuse, its pitfalls, and its alternatives
- Liskov's Substitution Principle for behavioral subtyping