Principles of Software Construction: Objects, Design, and Concurrency

Designing classes

Inheritance and delegation

Josh Bloch  Charlie Garrod
Administrivia

- Homework 2 due tonight
- Homework 3 due Sunday, February 7th
Key concepts from Thursday...
The equals contract

The equals method implements an **equivalence relation**. It is:

- **Reflexive**: For any non-null reference value `x`, `x.equals(x)` must return true.
- **Symmetric**: For any non-null reference values `x` and `y`, `x.equals(y)` must return true if and only if `y.equals(x)` returns true.
- **Transitive**: For any non-null reference values `x`, `y`, `z`, if `x.equals(y)` returns true and `y.equals(z)` returns true, then `x.equals(z)` must return true.
- **Consistent**: For any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
- For any non-null reference value `x`, `x.equals(null)` must return false.
Testing and defensive programming

- Assume clients will try to destroy invariants
  - May actually be true (malicious hackers)
  - More likely: honest mistakes

- Ensure class invariants survive any inputs
  - Defensive copying
  - Minimizing mutability
Assertions

• Statement containing boolean expression that programmer believes to be true:

```c
assert speed <= SPEED_OF_LIGHT;
```
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 inheritance and delegation

- 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</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without reuse</td>
<td>With reuse</td>
</tr>
</tbody>
</table>

- Graph showing the cost of products with and without reuse.
COMPOSITION AND DELEGATION
Recall our earlier sorting example:

Version A:
```java
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;

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
  – In the previous example, the Sorter is delegating functionality to some Comparator implementation
Delegation

• _Delegation_ is simply when one object relies on another object for some subset of its functionality
  – In the previous example, the Sorter is delegating functionality to some Comparator implementation

• Judicious delegation enables code reuse
Delegation

• *Delegation* is simply when one object relies on another object for some subset of its functionality
  – In the previous example, the Sorter is delegating functionality to some Comparator implementation

• 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):

        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.
Aside?: A sequence diagram for the LoggingList
Delegation and design

- Small interfaces with clear contracts
- Classes to encapsulate algorithms, behaviors
  - E.g., the Comparator, the strategy pattern
IMPLEMENTATION INHERITANCE
AND ABSTRACT CLASSES
Variation in the real world: types of bank accounts

<table>
<thead>
<tr>
<th>«interface» CheckingAccount</th>
<th>«interface» SavingsAccount</th>
</tr>
</thead>
<tbody>
<tr>
<td>getBalance() : long</td>
<td>getBalance() : long</td>
</tr>
<tr>
<td>deposit(amount : long)</td>
<td>deposit(amount : long)</td>
</tr>
<tr>
<td>withdraw(amount : long) : boolean</td>
<td>withdraw(amount : long) : boolean</td>
</tr>
<tr>
<td>transfer(amount : long, target : Account) : boolean</td>
<td>transfer(amount : long, target : Account) : boolean</td>
</tr>
<tr>
<td>getFee() : long</td>
<td>getInterestRate() : double</td>
</tr>
</tbody>
</table>
Interface inheritance for an account type hierarchy

- CheckingAccount extends Account. All methods from Account are inherited (copied to CheckingAccount).
- SavingsAccount is a subtype of Account. Account is a supertype of SavingsAccount.

```
«interface» Account

getBalance() : long
deposit(amount : long)
withdraw(amount : long) : boolean
transfer(amount : long, target : Account) : boolean
monthlyAdjustment()

«interface» CheckingAccount

getFee() : long

«interface» SavingsAccount

getInterestRate() : double

«interface» InterestCheckingAccount
```

If we know we have a CheckingAccount, additional methods are available.

Multiple interface extension
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:

```
If today is the last day of the month:
  For each acct in allAccounts:
    acct.monthlyAdjustment();
```
An implementation with poor reuse

```
«interface» Account
getBalance() : long
deposit(amount : long)
withdraw(amount : long) : boolean
transfer(amount : long,
    target : Account) : boolean
monthlyAdjustment()

«interface» CheckingAccount
getFee() : long

«interface» SavingsAccount
getInterestRate() : double

CheckingAccountImpl
...

«interface» InterestCheckingAccount

InterestCheckingAccountImpl
...

SavingsAccountImpl
...
```
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() { ... }
}
Better: Inheritance to reuse abstract account code

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

class A extends B

class A implements I
class A extends B
Typical roles for interfaces and classes

- An interface defines expectations / commitments for clients
- A class fulfills the expectations of an interface
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
Java details: extended reuse with super

```java
public abstract class AbstractAccount implements Account {
    protected long balance = 0;
    public boolean withdraw(long amount) {
        // withdraws money from account (code not shown)
    }
}

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;
    }
}
```

Overrides withdraw but also uses the superclass withdraw method.
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 a constructor of the superclass. Must be the first statement of the constructor.

Invokes another constructor in this same class
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.,
    double pi = 3.14;
    int indianaPi = (int) pi;

• Useful if you know you have a more specific subtype:
  – e.g.,
    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 in a nutshell

• If `Cowboy.draw()` overrides `Circle.draw()` somebody gets hurt!
Behavioral subtyping (Liskov Substitution Principle)

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 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
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

Coming Thursday:
- More Liskov's Substitution Principle
- Design patterns for reuse