Principles of Software Construction: Objects, Design, and Concurrency

Part 1: Designing classes

Design for reuse: delegation and inheritance

Charlie Garrod Chris Timperley

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- Homework 1 graded soon
- Reading due today: Effective Java, Items 17 and 50
  - Optional reading due Thursday
  - Required reading due next Tuesday
- Homework 2 due Thursday 11:59 p.m.

Design goals for your Homework 1 solution?

Functional correctness	Adherence of implementation to the specifications
Robustness	Ability to handle anomalous events
Flexibility	Ability to accommodate changes in specifications
Reusability	Ability to be reused in another application
Efficiency	Satisfaction of speed and storage requirements
Scalability	Ability to serve as the basis of a larger version of the application
Security	Level of consideration of application security

Source: Braude, Bernstein, Software Engineering. Wiley 2011

One Homework 1 solution…

```
class Document {
    private final String url;
    public Document(String url) {
        this.url = url;
    }

    public double similarityTo(Document d) {
        … ourText = download(url);
        … theirText = download(d.url);
        … ourFreq = computeFrequencies(ourText);
        … theirFreq = computeFrequencies(theirText);
        return cosine(ourFreq, theirFreq);
    }
    …
}
```

Compare to another Homework 1 solution…

```
class Document {
    private final String url;
    public Document(String url) {
        this.url = url;
    }

    public double s
        … ourText =
        … theirText
        … ourFreq =
        … theirFreq
        return cosi
    }
    …
}
```

```
class Document {
    private final … frequencies;
    public Document(String url) {
        … ourText = download(url);
        frequencies = computeFrequencies(ourText);
    }

    public double similarityTo(Document d) {
        return cosine(frequencies,
                      d.frequencies);
    }
    …
}
```

Using the Document class

```
For each url:
    Construct a new Document

For each pair of Documents d1, d2:
    Compute d1.similarityTo(d2)
    …
```

- What is the running time of this, for n urls?

Latency Numbers Every Programmer Should Know

Jeff Dean, Senior Fellow, Google

PRIMITIVE	LATENCY: ns	us	ms
L1 cache reference	0.5		
Branch mispredict	5		
L2 cache reference	7		
Mutex lock/unlock	25		
Main memory reference	100		
Compress 1K bytes with Zippy	3,000	3	
Send 1K bytes over 1 Gbps network	10,000	10	
Read 4K randomly from SSD*	150,000	150	
Read 1 MB sequentially from memory	250,000	250	
Round trip within same datacenter	500,000	500	
Read 1 MB sequentially from SSD*	1,000,000	1,000	1
Disk seek	10,000,000	10,000	10
Read 1 MB sequentially from disk	20,000,000	20,000	20
Send packet CA->Netherlands->CA	150,000,000	150,000	150

The point

- Constants matter
- Design goals sometimes clearly suggest one alternative

Key concepts from last Thursday

Key concepts from last Thursday

- Testing
  - Continuous integration, practical advice
  - Coverage metrics, statement coverage
- Exceptions in Java
- Behavioral subtyping
  - Liskov Substitution Principle
  - The `java.lang.Object` contracts

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. Subtypes must have:
  - 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*.

This Square is *not* a behavioral subtype of Rectangle

```
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;
   //@ ensures w==neww
             && h==old.h;
    void setWidth(int neww) {
        w=neww;
    }
}
```

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

   //@ requires neww > 0;
   //@ ensures w==neww
             && h==neww;
   @Override
   void setWidth(int neww) {
      w=neww;
      h=neww;
   }
}
```

Today

- Design for reuse: delegation and inheritance
  - Java-specific details for inheritance
- An exercise in equality

Recall our earlier sorting example:

Version A:

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

Version B':

```
interface Order {
  boolean lessThan(int i, int j);
}
final Order ASCENDING =  (i, j) -> i < j;
final Order DESCENDING = (i, j) -> i > j;

static void sort(int[] list, Order cmp) {
  …
  boolean mustSwap =
    cmp.lessThan(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 `Order`
- Judicious delegation enables code reuse

```
interface Order {
  boolean lessThan(int i, int j);
}
final Order ASCENDING =  (i, j) -> i < j;
final Order DESCENDING = (i, j) -> i > j;

static void sort(int[] list, Order cmp) {
  …
  boolean mustSwap =
    cmp.lessThan(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 `Order`
- Judicious delegation enables code reuse
  - `Sorter` can be reused with arbitrary sort orders
  - `Order`s can be reused with arbitrary client code that needs to compare integers

```
interface Order {
  boolean lessThan(int i, int j);
}
final Order ASCENDING =  (i, j) -> i < j;
final Order DESCENDING = (i, j) -> i > j;

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

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

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

- 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);
  }
  …
```

Delegation and design

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

institute for SOFTWARE RESEARCH

Today

- Design for reuse: delegation and inheritance
  - Java-specific details for inheritance
- An exercise in equality

Consider: types of bank accounts

```
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

```
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:

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

Implementation inheritance for code reuse

```
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

```
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

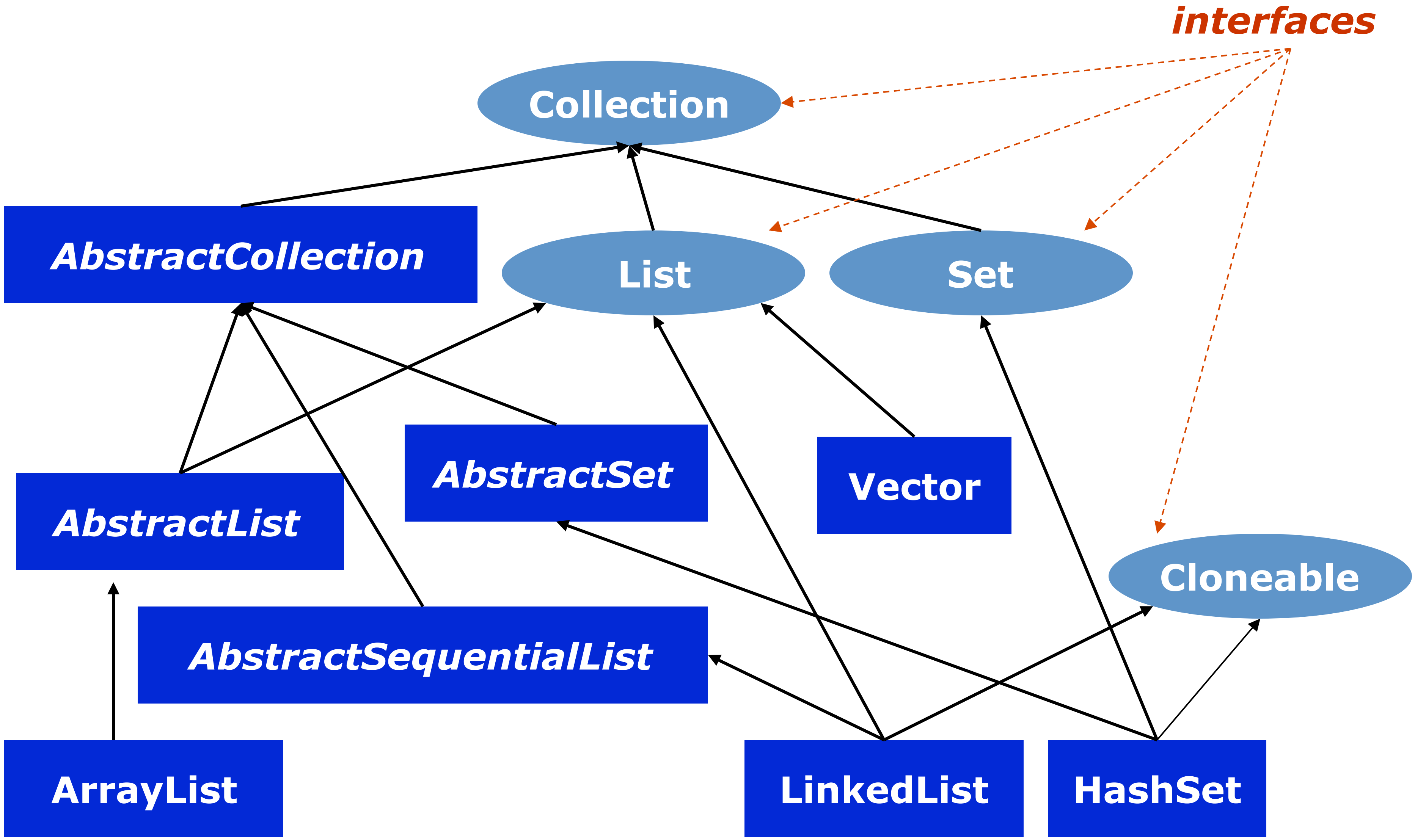
an abstract method is left to be implemented in a subclass

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)



The abstract `java.util.AbstractList<E>`

```
abstract E   get(int i);
abstract int size();
boolean      set(int i, E e);          // pseudo-abstract
boolean      add(E e);                 // pseudo-abstract
boolean      remove(E e);              // pseudo-abstract
boolean      addAll(Collection<? extends E> c);
boolean      removeAll(Collection<?> c);
boolean      retainAll(Collection<?> c);
boolean      contains(E e);
boolean      containsAll(Collection<?> c);
void         clear();
boolean      isEmpty();
Iterator<E>  iterator();
Object[]     toArray()
<T> T[]      toArray(T[] a);
…
```

Using `java.util.AbstractList<E>`

```
public class ReversedList<E> extends java.util.AbstractList<E>
             implements java.util.List<E> {
    private final List<E> list;

    public ReversedList(List<E> list) {
        this.list = list;
    }

    @Override
    public int size() {
        return list.size();
    }

    @Override
    public E get(int index) {
        return list.get(size() - index - 1);
    }
}
```

Benefits of inheritance

- Reuse of code
- Modeling flexibility

Inheritance and 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
```

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

```
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`

```
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

```
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();
    }
    …
}
```

Warning: This code is bad.

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

Delegation vs. inheritance summary

- Inheritance can improve modeling flexibility
- Usually, favor composition/delegation over inheritance
  - Inheritance violates information hiding
  - Delegation supports information hiding
- Design and document for inheritance, or prohibit it
  - Document requirements for overriding any method

Today

- Design for reuse: delegation and inheritance
  - Java-specific details for inheritance
- An exercise in equality

An `Object` method exercise

Provide all code needed for a reasonable equals method:

```
public final class Name {
    private final String first, last;
    public Name(String first, String last) {
        if (first == null || last == null)
            throw new NullPointerException();
        this.first = first; this.last = last;
    }
    …
}
```

What does the following code print?

```
public final class Name {
      private final String first, last;
      public Name(String first, String last) {
          if (first == null || last == null)
              throw new NullPointerException();
          this.first = first; this.last = last;
      public boolean equals(Name o) {
          return first.equals(o.first) && last.equals(o.last);
      }
      public int hashCode() {
          return 31 * first.hashCode() + last.hashCode();
      }
      public static void main(String[] args) {
          Set<Name> s = new HashSet<>();
          s.add(new Name("Mickey", "Mouse"));
          System.out.println(
              s.contains(new Name("Mickey", "Mouse")));
      }
}
```

(a) `true`
(b) `false`
(c) It varies
(d) None of the above

What does it print?

(a) `true`

(b) `false`

(c) It varies

(d) None of the above

The `Name` class overrides `hashCode` but not `equals`!

The two `Name` instances are thus unequal.

`Object.equals` has not been overridden

```
public final class Name {
      private final String first, last;
      public Name(String first, String last) {
          if (first == null || last == null)
              throw new NullPointerException();
          this.first = first; this.last = last;
      public boolean equals(Name o) {  // Accidental overloading
          return first.equals(o.first) && last.equals(o.last);
      }
      public int hashCode() {
          return 31 * first.hashCode() + last.hashCode();
      }
      public static void main(String[] args) {
          Set<Name> s = new HashSet<>();
          s.add(new Name("Mickey", "Mouse"));
          System.out.println(
              s.contains(new Name("Mickey", "Mouse")));
      }
}
```

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.

A correct `equals` implementation

```
@Override
public boolean equals(Object o) {
    if (!(o instanceof Name))
        return false;
    Name n = (Name) o;
    return n.first.equals(first) && n.last.equals(last);
}
```

The lesson

- Always override `hashCode` iff you override `equals`
- Always use `@Override` if you intend to override a method
  - or let your IDE generate these methods for you…