Principles of Software Construction: Objects, Design, and Concurrency
(Part 1: Designing Classes)

Design for Reuse (class level)

Christian Kästner  Bogdan Vasilescu
Administrivia

• Homework 2 due today
• Homework 3:
  – Out tonight / tomorrow
  – Due Thursday, February 9
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

```java
public class Sorter {
    void sort(int[] list, Comparator cmp) {
        ...
        boolean mustswap;
        mustswap = cmp.compare(list[i], list[j]);
        ...
    }
}

interface Comparator {
    boolean compare(int i, int j);
}

class UpComparator implements Comparator {
    boolean compare(int i, int j) { return i<j; }
}

class DownComparator implements Comparator {
    boolean compare(int i, int j) { return i>j; }
}
```
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.
Interface inheritance for an account type hierarchy

<table>
<thead>
<tr>
<th>«interface» <strong>CheckingAccount</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>getBalance() : float</td>
</tr>
<tr>
<td>deposit(amount : float)</td>
</tr>
<tr>
<td>withdraw(amount : float) : boolean</td>
</tr>
<tr>
<td>transfer(amount : float, target : Account) : boolean</td>
</tr>
<tr>
<td>getFee() : float</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>«interface» <strong>SavingsAccount</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>getBalance() : float</td>
</tr>
<tr>
<td>deposit(amount : float)</td>
</tr>
<tr>
<td>withdraw(amount : float) : boolean</td>
</tr>
<tr>
<td>transfer(amount : float, target : Account) : boolean</td>
</tr>
<tr>
<td>monthlyAdjustment()</td>
</tr>
<tr>
<td>getInterestRate() : float</td>
</tr>
</tbody>
</table>

**CheckingAccount** extends **Account**. All methods from **Account** are inherited (copied to **CheckingAccount**).
Implementation inheritance for code reuse

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

«interface» CheckingAccount
- `getFee() : float`

«interface» SavingsAccount
- `getInterestRate() : float`

CheckingAccountImpl
- ...
- `getBalance()`
- ...

SavingsAccountImpl
- ...
- `getBalance()`
- ...

InterestCheckingAccountImpl
- ...
- `getBalance()`
- ...

Duplication at code level
public abstract class AbstractAccount
    implements Account {
    protected float balance = 0.0;
    public float getBalance() {
        return balance;
    }
    abstract public void monthlyAdjustment();
    // other methods…
}

public class CheckingAccountImpl extends AbstractAccount
    implements CheckingAccount {
    public void monthlyAdjustment() {
        balance -= getFee();
    }
    public float getFee() { /* fee calculation */ }
}

Better: Reuse abstract account code
Alternatively: Reuse via composition and delegation

public class CheckingAccountImpl
implements CheckingAccount {
    BasicAccountImpl basicAcct = new(...);
    public float getBalance() {
        return basicAcct.getBalance();
    }
    // ...

CheckingAccountImpl is composed of a BasicAccountImpl
(Almost always) Prefer composition over inheritance

- **Tight coupling:**
  - Changes to superclass $\rightarrow$ changes to subclass implementation

- **Base class breaks encapsulation:**
  - Exposes implementation details to subclasses (protected members)

- **Inherited implementation can't change at runtime**

- **Inheritance stack may get very deep and confusing**

- **Inheritance: IS-A**
  - Can use subclass where superclass is expected
  - E.g., Cessna biplane “is a” Airplane

- **Composition: HAS-A**
  - Only want some of the behavior of the superclass
  - E.g., Bird could (but shouldn’t) inherit from Airplane, they both fly()
THE DECORATOR DESIGN PATTERN
Limitations of inheritance

- Suppose you want various extensions of a Stack data structure...
  - UndoStack: A stack that lets you undo previous push or pop operations
  - SecureStack: A stack that requires a password
  - SynchronizedStack: A stack that serializes concurrent accesses
Limitations of inheritance

• Suppose you want various extensions of a Stack data structure...
  – UndoStack: A stack that lets you undo previous push or pop operations
  – SecureStack: A stack that requires a password
  – SynchronizedStack: A stack that serializes concurrent accesses
  – SecureUndoStack: A stack that requires a password, and also lets you undo previous operations
  – SynchronizedUndoStack: A stack that serializes concurrent accesses, and also lets you undo previous operations
  – SecureSynchronizedStack: ...
  – SecureSynchronizedUndoStack: ...

Goal: arbitrarily composable extensions
Limitations of inheritance

- Extensions not combinable
- Middle extension not optional
Workarounds?

- Combining inheritance hierarchies
  - Combinatorial explosion
  - Massive code replication

Multiple inheritance
  - Diamond problem
The *Decorator* design pattern

- **Problem**: Need arbitrary / dynamically composable extensions to individual objects.

- **Solution**:
  - Implement common interface
  - Delegate primary responsibility to an underlying object.

- **Consequences**:
  - More flexible than static inheritance
  - Customizable, cohesive extensions
  - Breaks object identity, self-references
Decorators use both subtyping and delegation

```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);
    }
    ...
```
Using the Decorator for our Stack example

The abstract forwarding class

```java
public abstract class StackDecorator implements IStack {
    private final IStack stack;

    public StackDecorator(IStack stack) {
        this.stack = stack;
    }

    public void push(Item e) {
        stack.push(e);
    }

    public Item pop() {
        return stack.pop();
    }

    ...}
```
Using the Decorator for our Stack example

A concrete decorator class

```java
public class UndoStack
    extends StackDecorator
    implements IStack {
    private final UndoLog log = new UndoLog();
    public UndoStack(IStack stack) {
        super(stack);
    }
    public void push(Item e) {
        log.append(UndoLog.PUSH, e);
        super.push(e);
    }
    ...}
```
Using the Decorator for our Stack example

Using the decorator classes

- To construct a plain stack:
  \[\text{Stack } s = \text{new Stack();}\]

- To construct an plain undo stack:
  \[\text{UndoStack } s = \text{new UndoStack(new Stack());}\]

- To construct a secure synchronized undo stack:
  \[\text{SecureStack } s = \text{new SecureStack(new SynchronizedStack(new UndoStack(new Stack()))});\]
Decorators from java.util.Collections

• Turn a mutable list into an immutable list:
  
  static List<T> unmodifiableList(List<T> lst);
  static Set<T> unmodifiableSet(Set<T> set);
  static Map<K,V> unmodifiableMap(Map<K,V> map);

• Similar for synchronization:
  
  static List<T> synchronizedList(List<T> lst);
  static Set<T> synchronizedSet(Set<T> set);
  static Map<K,V> synchronizedMap(Map<K,V> map);
The UnmodifiableCollection (simplified excerpt)

```java
public static <T> Collection<T> unmodifiableCollection(Collection<T> c) {
    return new UnmodifiableCollection<>(c);
}

class UnmodifiableCollection<E> implements Collection<E>, Serializable {
    final Collection<E> c;
    UnmodifiableCollection(Collection<> c) {this.c = c; }
    public int size() {return c.size();}
    public boolean isEmpty() {return c.isEmpty();}
    public boolean contains(Object o) {return c.contains(o);}
    public Object[] toArray() {return c.toArray();}
    public <T> T[] toArray(T[] a) {return c.toArray(a);}
    public String toString() {return c.toString();}
    public boolean add(E e) {throw new UnsupportedOperationException();}
    public boolean remove(Object o) { throw new UnsupportedOperationException();
        public boolean containsAll(Collection<?> coll) { return c.containsAll(coll);
        public boolean addAll(Collection<? extends E> coll) { throw new UnsupportedOperationException();
        public boolean removeAll(Collection<?> coll) { throw new UnsupportedOperationException();
        public boolean retainAll(Collection<?> coll) { throw new UnsupportedOperationException();
    public void clear() { throw new UnsupportedOperationException();
```
The decorator pattern vs. inheritance

• Decorator composes features at run time
  – Inheritance composes features at compile time

• Decorator consists of multiple collaborating objects
  – Inheritance produces a single, clearly-typed object

• Can mix and match multiple decorations
  – Multiple inheritance has conceptual problems

• Risk when multiple clients: Danger of inconsistent references, possibility of different perspectives
CLASS INVARIANTS
Recall: Data Structure Invariants (cf. 122)

struct list {
    elem data;
    struct list* next;
};

struct queue {
    list front;
    list back;
};

bool is_queue(queue Q) {
    if (Q == NULL) return false;
    if (Q->front == NULL || Q->back == NULL) return false;
    return is_segment(Q->front, Q->back);
}
Recall: Data Structure Invariants (cf. 122)

- Properties of the Data Structure
- Should always hold before and after method execution
- May be invalidated temporarily during method execution

```java
void enq(queue Q, elem s)
//@requires is_queue(Q);
//@ensures is_queue(Q);
{ ... }
```
Class Invariants

• Properties about the fields of an object
• Established by the constructor
• Should always hold before and after execution of public methods
  – May be invalidated temporarily during method execution
Class Invariants

- Properties about the fields of an object
- Established by the constructor
- Should always hold before and after execution of public methods

```java
public class SimpleSet {
    int contents[];
    int size;

    //@ invariant sorted(contents);
    SimpleSet(int capacity) { ... }

    //@ requires sorted(contents);
    //@ ensures sorted(contents);
    boolean add(int i) { ... }

    //@ requires sorted(contents);
    //@ ensures sorted(contents);
    boolean contains(int i) { ... }
}
```
BEHAVIORAL SUBTYPING

“SHOULD I BE INHERITING FROM THIS TYPE?”
(Almost always) Prefer composition over inheritance

- **Tight coupling:**
  - Changes to superclass $\rightarrow$ changes to subclass implementation

- **Base class breaks encapsulation:**
  - Exposes implementation details to subclasses (protected members)

- **Inherited implementation can't change at runtime**

- **Inheritance stack may get very deep and confusing**

- **Inheritance: IS-A**
  - Can use subclass where superclass is expected
  - E.g., Cessna biplane “is a” Airplane

- **Composition: HAS-A**
  - Only want some of the behavior of the superclass
  - E.g., Bird could (but shouldn’t) inherit from Airplane, they both fly()
Behavioral subtyping (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

- Applies to specified behavior:
  - Same or stronger invariants
  - Same or stronger postconditions for all methods
  - Same or weaker preconditions for all methods

- 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

This is called the Liskov Substitution Principle.
Behavioral subtyping in a nutshell

- If `Cowboy.draw()` overrides `Circle.draw()` somebody gets hurt!
Car is a behavioral subtype of Vehicle

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

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
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
Is this Square a behavioral subtype of Rectangle?

Yes!
- Subclass fulfills the same invariants (and additional ones)
- Overridden methods: NA
Is this Square a behavioral subtype of Rectangle?

Yes!
- Subclass fulfills the same invariants (and additional ones)
- Overridden methods: NA
Is this Square a behavioral subtype of Rectangle?

```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;
    }
    //@ 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);
    }
}
```

No!

- Invalidates stronger invariant (w==h) in subclass

class GraphicProgram {
    void scale(Rectangle r, int factor) {
        rsetWidth(r.getWidth() * factor);
    }
}
Is this Square 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;
    }
    //@ ensures getArea = h*w
    float getArea() {
        return(this.h * this.w);
    }
    //@ requires factor > 0;
    //@ ensures w == \old(w)*factor
    //    && h == \old(h)
    void scaleW(int factor) {
        w=w*factor;
    }
}

No!

• Invariants OK
• Preconditions OK
• Overridden method scaleW has incompatible postcondition

class Square extends Rectangle {
    //@ invariant h>0 && w>0;
    //@ invariant h==w;
    Square(int w) {
        super(w, w);
    }
    //@ ensures getArea = h*w
    float getArea() {
        return(this.h * this.w);
    }
    //@ requires factor > 0;
    //@ ensures w == \old(w)*factor
    //    && h == \old(h)*factor
    void scaleW(int factor) {
        w=w*factor;
        h=h*factor;
    }
}

Rectangle alwaysTrue(Rectangle r) {
    double initialArea = r.getArea();
    double finalArea = r.scaleW(2).getArea();
    return(finalArea == 2*initialArea);
}
Is this Rectangle a behavioral subtype of Square?

```
Is this Rectangle a behavioral subtype of Square?

class Rectangle extends Square {
    //@ invariant h>0 && w>0;
    int h, w;
    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }
    //@ ensures getArea = h*w
    float getArea()
        return(this.h * this.w);
    //@ requires factor > 0;
    //@ ensures w == \old(w)*factor
    // && h == \old(h)
    void scaleW(int factor) {
        w=w*factor;
    }
}

No!
• Rectangle doesn’t fulfil Square invariant h==w
• Preconditions OK
• Overridden method scaleW has incompatible postcondition

class Square {
    //@ invariant h>0 && w>0;
    //@ invariant h==w;
    Square(int w) {
        super(w, w);
    }
    //@ ensures getArea = h*w
    float getArea()
        return(this.h * this.w);
    //@ requires factor > 0;
    //@ ensures w == \old(w)*factor
    // && h == \old(h)*factor
    void scaleW(int factor) {
        w=w*factor;
        h=h*factor;
    }

Square alwaysTrue(Square s) {
    double intialArea = s.getArea();
    double finalArea = s.scaleW(2).getArea();
    return(finalArea == 4*initialArea);}
```
Summary: Designing reusable classes

• Favor composition over inheritance
  – Inheritance violates information hiding

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

• Reusable implementations with simple, clear contracts
• Inheritance for reuse, its pitfalls, and its alternatives
• Liskov's Substitution Principle for behavioral subtyping
CLASS INVARIANTS AND DEFENSIVE COPYING
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
This class is **not** robust

```java
public final class Period {
    private final Date start, end; // Invariant: start <= end

    /**
     * @throws IllegalArgumentException if start > end
     * @throws NullPointerException if start or end is null
     */
    public Period(Date start, Date end) {
        if (start.after(end))
            throw new IllegalArgumentException(start + " > " + end);
        this.start = start;
        this.end = end;
    }

    public Date start() { return start; }
    public Date end()   { return end; }
    ... // Remainder omitted
}
```
The problem: Date is mutable

// Attack the internals of a Period instance
Date start = new Date();  // (The current time)
Date end   = new Date();  //   "     "      "   "
Period p = new Period(start, end);
end.setYear(78);  // Modifies internals of p!
The solution: defensive copying

// Original constructor
public Period(Date start, Date end) {
    if (start.after(end))
        throw new IllegalArgumentException(start + " > " + end);
    this.start = start;
    this.end = end;
}

// Repaired constructor - defensively copies parameters
public Period(Date start, Date end) {
    this.start = new Date(start.getTime());
    this.end = new Date(end.getTime());
    if (this.start.after(this.end))
        throw new IllegalArgumentException(start + " > " + end);
}
A few important details

- Copies made *before* checking parameters
- Validity check performed on copies
- Eliminates *window of vulnerability* between parameter check and copy
- Thwarts multithreaded TOCTOU attack
  - Time-Of-Check-To-Time-Of-U

```java
// BROKEN - Permits multithreaded attack!
public Period(Date start, Date end) {
    if (start.after(end))
        throw new IllegalArgumentException(start + " > " + end);
    // Window of vulnerability
    this.start = new Date(start.getTime());
    this.end = new Date(end.getTime());
}
```
Another important detail

- Used constructor, not clone, to make copies
  - Necessary because Date class is nonfinal
  - Attacker could implement malicious subclass
    - Records reference to each extant instance
    - Provides attacker with access to instance list
- But who uses clone, anyway? [EJ Item 11]
Unfortunately, constructors are only half the battle

// Accessor attack on internals of Period
Period p = new Period(new Date(), new Date());
Date d = p.end();
p.end.setYear(78); // Modifies internals of p!
The solution: more defensive copying

// Repaired accessors - defensively copy fields
public Date start() {
    return new Date(start.getTime());
}
public Date end() {
    return new Date(end.getTime());
}

Now Period class is robust!
Summary

• Don’t incorporate mutable parameters into object; make defensive copies
• Return defensive copies of mutable fields...
• Or return unmodifiable view of mutable fields
• **Real lesson – use immutable components**
  – Eliminates the need for defensive copying
IMMUTABILITY
Immutable classes

• **Class whose instances cannot be modified**
• Examples: String, Integer, BigInteger
• How, why, and when to use them
How to write an immutable class

- Don’t provide any mutators
- Ensure that no methods may be overridden
- Make all fields final
- Make all fields private
- Ensure security of any mutable components
public final class Complex {
    private final double re, im;

    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    // Getters without corresponding setters
    public double realPart() { return re; }
    public double imaginaryPart() { return im; }

    // subtract, multiply, divide similar to add
    public Complex add(Complex c) {
        return new Complex(re + c.re, im + c.im);
    }
}
Distinguishing characteristic

• Return new instance instead of modifying
• *Functional programming*
• May seem unnatural at first
• Many advantages
Advantages

- Simplicity
- Inherently Thread-Safe
- Can be shared freely
- No need for defensive copies
- Excellent building blocks
Major disadvantage

- **Separate instance for each distinct value**
- **Creating these instances can be costly**
  - `BigInteger moby = ...; // A million bits long`
  - `moby = moby.flipBit(0); // Ouch!`
- **Problem magnified for multistep operations**
  - Well-designed immutable classes provide common multistep operations as primitives
  - Alternative: mutable companion class
    - e.g., `StringBuilder` for `String`
When to make classes immutable

• **Always, unless there's a good reason not to**
• Always make small “value classes” immutable!
  – Examples: Color, PhoneNumber, Unit
  – Date and Point were mistakes!
  – Experts often use long instead of Date
When to make classes mutable

• Class represents entity whose state changes  
  – Real-world - BankAccount, TrafficLight  
  – Abstract - Iterator, Matcher, Collection  
  – Process classes - Thread, Timer  

• If class must be mutable, *minimize mutability*  
  – Constructors should fully initialize instance  
  – Avoid reinitialize methods
PARAMETRIC POLYMORPHISM (GENERICS)
Recall the Java Collection API (excerpt)
Consider the `java.util.Stack`

```java
public class Stack {
    public void push(Object obj) { ... }
    public Object pop() { ... }
}
```

• Some possible client code:
  ```java
  Stack stack = new Stack();
  String s = "Hello!";
  stack.push(s);
  String t = stack.pop();
  ```
Consider the `java.util.Stack`

```java
public class Stack {
    public void push(Object obj) { ... }
    public Object pop() { ... }
}
```

• Some possible client code:
  ```java
  Stack stack = new Stack();
  String s = "Hello!";
  stack.push(s);
  String t = (String) stack.pop();
  ```

  To fix the type error with a downcast *(urgs!)*
Parametric polymorphism via Java Generics

- **Parametric polymorphism** is the ability to define a type generically to allow static type-checking without fully specifying types
- The `java.util.Stack` instead
  - A stack of some type `T`:
    ```java
    public class Stack<T> {
        public void push(T obj) { ... }
        public T pop() { ... }
    }
    ```
- Improves typechecking, simplifies(?) client code:
  ```java
  Stack<String> stack = new Stack<String>();
  String s = "Hello!";
  stack.push(s);
  String t = stack.pop();
  ```
Many Java Generics details

• Can have multiple type parameters
  – e.g., `Map<Integer, String>`

• Wildcards
  – e.g., `ArrayList< ?>` or `ArrayList< ? extends Animal >`

• Subtyping
  – `ArrayList<String>` is a subtype of `List<String>`
  – `ArrayList<String>` is not a subtype of `ArrayList< Object >`

• Cannot create Generic arrays
  
  ```java
  List<String>[] foo = new List<String>[42]; // won't compile
  ```

• Type erasure
  – Generic type info is compile-time only
    • Cannot use `instanceof` to check generic type
      – But shouldn’t use `instanceof` anyway
ITERATOR PATTERN
Traversing a collection

• Since Java 1.0:
  
  ```java
  List<String> arguments = ...;
  for (int i = 0; i < arguments.size(); ++i) {
    System.out.println(arguments.get(i));
  }
  ```

• Java 1.5: for-each loop
  
  ```java
  List<String> arguments = ...;
  for (String s : arguments) {
    System.out.println(s);
  }
  ```

• For-each loop works for every implementation of Iterable
  
  ```java
  public interface Iterable<E> {
    public Iterator<E> iterator();
  }
  ```
The Iterator interface

```java
public interface java.util.Iterator<E> {
    boolean hasNext();
    E next();
    void remove();  // removes previous returned item
}                 // from the underlying collection

• To use explicitly, e.g.:
  List<String> arguments = ...
  for (Iterator<String> it = arguments.iterator();
       it.hasNext(); ) {
    String s = it.next();
    System.out.println(s);
  }
```
Getting an Iterator

```
public interface Collection<E> extends Iterable<E> {
  boolean add(E e);
  boolean addAll(Collection<? extends E> c);
  boolean remove(Object e);
  boolean removeAll(Collection<?> c);
  boolean contains(Object e);
  boolean containsAll(Collection<?> c);
  void clear();
  int size();
  boolean isEmpty();
  Iterator<E> iterator();
  Object[] toArray();
  <T> T[] toArray(T[] a);
  ...
}
```

Defines an interface for creating an Iterator, but allows Collection implementation to decide which Iterator to create.
public class Pair<E> {
    private final E first, second;
    public Pair(E f, E s) { first = f; second = s; }
}

Pair<String> pair = new Pair<String>("foo", "bar");
for (String s : pair) { ... }
An Iterator implementation for Pairs

```java
public class Pair<E> implements Iterable<E> {
    private final E first, second;
    public Pair(E f, E s) { first = f; second=s;}
    public Iterator<E> iterator() {
        return new PairIterator();
    }
    private class PairIterator implements Iterator<E> {
        private boolean seenFirst = false, seenSecond = false;
        public boolean hasNext() { return !seenSecond; }
        public E next() {
            if (!seenFirst) { seenFirst = true; return first; }  
            if (!seenSecond) { seenSecond = true; return second; }
            throw new NoSuchElementException();
        }
        public void remove() {
            throw new UnsupportedOperationException();
        }
    }
    Pair<String> pair = new Pair<String>("foo", "bar");
    for (String s : pair) { ... }
}
```
Iterator design pattern

• Problem: Clients need uniform strategy to access all elements in a container, independent of the container type
  – Order is unspecified, but access every element once
• Solution: A strategy pattern for iteration
• Consequences:
  – Hides internal implementation of underlying container
  – Easy to change container type
  – Facilitates communication between parts of the program
Using a `java.util.Iterator<E>`: A warning

- The default Collections implementations are mutable...
- ...but their Iterator implementations assume the collection does not change while the Iterator is being used
  - You will get a `ConcurrentModificationException`
Using a java.util.Iterator<E>: A warning

- The default Collections implementations are mutable...
- ...but their Iterator implementations assume the collection does not change while the Iterator is being used
  - You will get a ConcurrentModificationException
  - If you simply want to remove an item:
    ```java
    List<String> arguments = ...;
    for (Iterator<String> it = arguments.iterator();
         it.hasNext(); ) {
      String s = it.next();
      if (s.equals("Charlie"))
        arguments.remove("Charlie"); // runtime error
    }
    ```
Using a `java.util.Iterator<E>`: A warning

- The default Collections implementations are mutable...
- ...but their Iterator implementations assume the collection does not change while the Iterator is being used
  - You will get a ConcurrentModificationException
  - If you simply want to remove an item:
    ```java
    List<String> arguments = ...;
    for (Iterator<String> it = arguments.iterator();
         it.hasNext(); ) {
      String s = it.next();
      if (s.equals("Charlie"))
        it.remove();
    }
    ```