Principles of Software Construction

Designing classes for reuse

Principles, patterns, and parametric polymorphism

Josh Bloch  Charlie Garrod
Administrivia

• Homework 3 due Sunday, February 7\textsuperscript{th}
  SEND
  + MORE
  --------
  MONEY

• Midterm exam next Thursday, February 12\textsuperscript{th}
  – Review session Wednesday, Feb 11\textsuperscript{th}, 7-9 p.m. DH 1212
  – Practice exam will be released this weekend
Java puzzlers: “Animal Farm” (2005)

class AnimalFarm {
    public static void main(String[] args) {
        final String pig = "length: 10";
        final String dog = "length: " + pig.length();
        System.out.println("Animals are equal: "
                          + pig == dog);
    }
}
What does it print?

public class AnimalFarm {
    public static void main(String[] args) {
        final String pig = "length: 10";
        final String dog = "length: " + pig.length();
        System.out.println("Animals are equal: " + pig == dog);
    }
}

(a) Animals are equal: true
(b) Animals are equal: false
(c) It varies
(d) None of the above
What does it print?

(a) Animals are equal: true
(b) Animals are equal: false
(c) It varies
(d) None of the above: false

The + operator binds tighter than ==
public class AnimalFarm {
    public static void main(String[] args) {
        final String pig = "length: 10";
        final String dog = "length: " + pig.length();
        System.out.println("Animals are equal: "+ pig == dog);
    }
}
You could try to fix it like this...

```java
public class AnimalFarm {
    public static void main(String[] args) {
        final String pig = "length: 10";
        final String dog = "length: " + pig.length();
        System.out.println("Animals are equal: " + (pig == dog));
    }
}
```

**Prints** Animals are equal: false
public class AnimalFarm {
    public static void main(String[] args) {
        final String pig = "length: 10";
        final String dog = "length: " + pig.length();
        System.out.println("Animals are equal: "
            + pig.equals(dog));
    }
}
The moral

• Use parens, not spacing, to express intent
  – The compiler ignores whitespace
  – Spacing can be deceptive; parentheses never lie

• Use parens whenever there is any doubt
  – They clarify your intent and cost nothing
  – They make your code easier to read and maintain
  – They may save your bacon

• Don’t depend on interning of string constants

• Use `.equals`, not `==` for object references
Key concepts from Thursday...
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.
Avoiding `instanceof` with the template method pattern

```java
public void doSomething(Account acct) {
    float adj = 0.0;
    if (acct instanceof CheckingAccount) {
        checkingAcct = (CheckingAccount) acct;
        adj = checkingAcct.getFee();
    } else if (acct instanceof SavingsAccount) {
        savingsAcct = (SavingsAccount) acct;
        adj = savingsAcct.getInterest();
    }
    ...
}

Instead:
    public void doSomething(Account acct) {
        long adj = acct.getMonthlyAdjustment();
        ...
    }
```
The decorator design pattern
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.
Learning goals for today

• Be able to write reusable data structures using Java Generics.
• Be able to use and write Iterators.
• Be able to use and write Exceptions, including client code depending on try, catch, and finally.
Today: More class-level reuse

- Puzzlers...
- Parametric polymorphism (a.k.a. generics)
- The Iterator design pattern
- An important aside: Exceptions
An implementation of pairs, a la 2003

public class Pair {
    private final Object first, second;
    public Pair(Object first, Object second) {
        this.first = first;
        this.second = second;
    }
    public Object first() { return first; }
    public Object second() { return second; }
}
An implementation of pairs, a la 2003

public class Pair {
    private final Object first, second;
    public Pair(Object first, Object second) {
        this.first = first;
        this.second = second;
    }
    public Object first() { return first; }
    public Object second() { return second; }
}

• Some possible client code?
  Pair p = new Pair("Hello", "world");
  String result = p.first();
An implementation of pairs, a la 2003

public class Pair {
    private final Object first, second;
    public Pair(Object first, Object second) {
        this.first = first;
        this.second = second;
    }
    public Object first() { return first; }
    public Object second() { return second; }
}

• Some possible client code?
    Pair p = new Pair("Hello", "world");
    String result = p.first(); // Won't compile--type error
An implementation of pairs, a la 2003

```java
public class Pair {
    private final Object first, second;
    public Pair(Object first, Object second) {
        this.first = first;
        this.second = second;
    }
    public Object first() { return first; }
    public Object second() { return second; }
}

• Some possible client code:
    Pair p = new Pair("Hello", "world");
    assert p.first() instanceof String;
    String result = (String) p.first();
```
Parametric polymorphism (a.k.a. generics)

- *Parametric polymorphism* is the ability to define a type generically, allowing static type-checking without fully specifying the type
  - e.g.:

```java
public class Frequency {
    public static void main(String[] args) {
        Map<String, Integer> m = new TreeMap<>();
        for (String word : args) {
            Integer freq = m.get(word);
            m.put(word, (freq == null ? 1 : freq + 1));
        }
        System.out.println(m);
    }
}
```
A generic implementation of pairs

```java
public class Pair<E> {
    private final E first, second;
    public Pair(E first, E second) {
        this.first = first;
        this.second = second;
    }
    public E first() { return first; }
    public E second() { return second; }
}

• Better client code:
    Pair<String> p = new Pair<>("Hello", "world");
    String result = p.first();
```
Some Java Generics details

- Can have multiple type parameters
  - e.g., `Map<String, Integer>`
- Wildcards
  - e.g. `List<?>` or `List<?> extends Animal` or `List<?> super Animal`
- Generics are type invariant
  - `ArrayList<String>` is a subtype of `List<String>`
  - `List<String>` is not a subtype of `List<Object>`
- Generic type info is erased (i.e. compile-time only)
  - Cannot use `instanceof` to check generic type
- Cannot create Generic arrays
  ```java
  Pair<String>[] foo = new Pair<String>[42]; // won't compile
  ```
Generic array creation is illegal

// won't compile
List<String>[] stringLists = new List<String>[1];
List<Integer> intList = Arrays.asList(42);
Object[] objects = stringLists;
objects[0] = intList;
String s = stringLists[0].get(0); // Would be type-safe
Generic design advice: Prefer lists to arrays

// Fails at runtime
Object[] oArray = new Long[42];
oArray[0] = "I don't fit in"; // Throws ArrayStoreException

// Won't compile
List<Object> ol = new ArrayList<Long>(); // Incompatible type
ol.add("I don't fit in");
Wildcard types provide API flexibility

- List<String> is not a subtype of List<Object>
- List<String> is a subtype of List<? extends Object>
- List<Object> is a subtype of List<? super String>
An inflexible API without wildcards

- Suppose you want to add bulk methods to Stack<E>:
  ```java
  void pushAll(Collection<E> src);
  void popAll(Collection<E> dst);
  ```

- Problem:
  - It should be fine to push a Long onto a Stack<Number>:
    ```java
    Collection<Long> numbers = …;
    Stack<Number> numberStack = …;
    for (Long n : numbers) {
        numberStack.push(n);
    }
    ```
  - This API prevents `pushAll(Collection<Long>)` onto a Stack<Number>
Wildcards in the java.util.Collection API

public interface Collection<E> ... {
    boolean add(E e);
    boolean addAll(Collection<? extends E> c);
    boolean remove(Object e);
    boolean removeAll(Collection<?> c);
    boolean retainAll(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);
    ...
}
Generic design advice: Use your PECS

- **PECS:** Producer extends, Consumer super
  - For a T producer, use Foo<? extends T>
  - For a T consumer, use Foo<? super T>
  - Mnemonic only works for input parameters
Use your PECS

• Suppose you want to add bulk methods to Stack\(<E>\):
  
  ```java
  void pushAll(Collection\(<E>\) src);
  
  void popAll(Collection\(<E>\) dst);
  ```
Use your PECS

- Suppose you want to add bulk methods to Stack<E>:
  void pushAll(Collection<? extends E> src);
    - src is an E producer

  void popAll(Collection<? super E> dst);
    - dst is an E consumer
Today: More class-level reuse

- Puzzlers...
- Parametric polymorphism (a.k.a. generics)
- The Iterator design pattern
- An important aside: Exceptions
Traversing a collection

- Old-school Java for loop for ordered types
  ```java
  List<String> arguments = ...;
  for (int i = 0; i < arguments.size(); i++) {
    System.out.println(arguments.get(i));
  }
  ```

- Modern standard Java for-each loop
  ```java
  List<String> arguments = ...;
  for (String s : arguments) {
    System.out.println(s);
  }
  ```
Traversing a collection

• Old-school Java for loop for ordered types
  List<String> arguments = ...;
  for (int i = 0; i < arguments.size(); ++i) {
      System.out.println(arguments.get(i));
  }

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

Works for every implementation of Iterable:
  public interface Iterable<E> {
      public Iterator<E> iterator();
  }
The Iterator interface

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);
    }
The Iterator design pattern

• Strategy to uniformly access all elements in a sequence
  – Independent of the container implementation
  – Ordering is unspecified, but every element visited once

• Design for change, information hiding
  – Hides internal implementation of underlying container

• Design for reuse, division of labor
  – Hides complex data structure behind simple interface
  – Facilitates communication between parts of the program
A design principle for reuse: *low coupling*

- Each component should depend on as few other components as possible
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 retainAll(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.
An Iterator implementation for Pairs

```java
public class Pair<E> {
    private final E first, second;
    public Pair(E f, E s) { first = f; second = s; }
}
```

```java
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) { ... }
}
```
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:
    List<String> arguments = ...;
    for (Iterator<String> it = arguments.iterator();
        it.hasNext(); ) {
        String s = it.next();
        if (s.equals("Charlie"))
          it.remove();
    }
Today: More class-level reuse

- Puzzlers...
- Parametric polymorphism (a.k.a. generics)
- The Iterator design pattern
- An important aside: Exceptions
What does this code do?

FileInputStream fIn = new FileInputStream(filename);
if (fIN == null) {
    switch (errno) {
    case _ENOFILE:
        System.err.println("File not found: " + …);
        return -1;
    default:
        System.err.println("Something else bad happened: " + …);
        return -1;
    }
}
DataInput dataInput = new DataInputStream(fIn);
if (dataInput == null) {
    System.err.println("Unknown internal error.");
    return -1; // errno > 0 set by new DataInputStream
}
int i = dataInput.readInt();
if (errno > 0) {
    System.err.println("Error reading binary data from file");
    return -1;
} // The slide lacks space to close the file. Oh well.
return i;
What does this code do?

```java
FileInputStream fIn = new FileInputStream(filename);
if (fIn == null) {
    switch (errno) {
    case _ENOFILE:
        System.err.println("File not found: " + ...);
        return -1;
    default:
        System.err.println("Something else bad happened: " + ...);
        return -1;
    }
}
DataInputStream dataInput = new DataInputStream(fIn);
if (dataInput == null)
    System.err.println("Unknown internal error.");
    return -1;  // errno > 0 set by new DataInputStream
}
int i = dataInput.readInt();
if (errno > 0) {
    System.err.println("Error reading binary data from file");
    return -1;
}
    // The slide lacks space to close the file. Oh well.
return i;
```
Compare to:

```java
FileInputStream fileInput;
try {
    FileInputStream fileInput = new FileInputStream(filename);
    DataInputStream dataInput = new DataInputStream(fileInput);
    return dataInput.readInt();
} catch (FileNotFoundException e) {
    System.out.println("Could not open file " + filename);
} catch (IOException e) {
    System.out.println("Couldn’t read file: " + e);
} finally {
    if (fileInput != null) fileInput.close();
}
```
Exceptions

- Notify caller of an exceptional condition by automatic transfer of control
- Semantics:
  - Propagates up stack until main method is reached (terminates program), or exception is caught
- Sources:
  - Program – e.g., IllegalArgumentException
  - JVM – e.g., NullPointerException
Exceptional control-flow in Java

```java
public static void test() {
    try {
        System.out.println("Top");
        int[] a = new int[10];
        a[42] = 42;
        System.out.println("Bottom");
    } catch (NegativeArraySizeException e) {
        System.out.println("Caught negative array size");
    }
}

public static void main(String[] args) {
    try {
        test();
    } catch (IndexOutOfBoundsException e) {
        System.out.println("Caught index out of bounds");
    }
}
```
The exception hierarchy in Java

```
Object
  
Throwables
    
Exception
      
RuntimeException
  
IOException
    EOFException
  
NullPointerException
  
IndexOutOfBoundsException
  
ClassNotFoundException
  
...
```
Checked vs. unchecked exceptions

- Checked exception
  - Must be caught or propagated, or program won’t compile

- Unchecked exception
  - No action is required for program to compile
  - But uncaught exception will cause program to fail!
Design choice: Checked and unchecked exceptions and return values

• Unchecked exception
  – Programming error, other unrecoverable failure

• Checked exception
  – An error that every caller should be aware of and handle

• Special return value (e.g., null from Map.get)
  – Common but atypical result

• Do NOT use return codes

• Do NOT return null to indicate a zero-length result
  – Use a zero-length list or array
Creating and throwing your own exceptions

public class SpanishInquisitionException extends RuntimeException {
    ...
}

public class HolyGrail {
    public void seek() {
        ...
        if (heresyByWord() || heresyByDeed())
            throw new SpanishInquisitionException();
        ...
    }
}

Benefits of exceptions

• You can’t forget to handle common failure modes
  – Compare: using a flag or special return value

• Provide high-level summary of error and stack trace
  – Compare: core dump in C

• Improve code structure
  – Separates normal code path from exceptional
  – Eases task of recovering from failure

• In sum: ease task of writing robust, maintainable code
Guidelines for using exceptions (1)

- Avoid unnecessary checked exceptions (EJ Item 59)
- Favor standard exceptions (EJ Item 60)
  - IllegalArgumentException – invalid parameter value
  - IllegalStateException – invalid object state
  - NullPointerException – null param where prohibited
  - IndexOutOfBoundsException – invalid index param
- Throw exceptions appropriate to abstraction (EJ Item 61)
Guidelines for using exceptions (2)

• Document all exceptions thrown by each method
  – Checked and unchecked (EJ Item 62)

• Include failure-capture info in detail message (EJ Item 63)
  – throw new IllegalArgumentException(
    "Modulus must be prime: " + modulus);

• Don’t ignore exceptions (EJ Item 65)
  // Empty catch block ignores exception – Bad smell in code!
  try {
    ...
  } catch (SomeException e) {
  }
import org.junit.*;
import static org.junit.Assert.fail;

public class Tests {

    @Test
    public void testSanityTest(){
        try {
            openNonexistingFile();
            fail("Expected exception");
        } catch(IOException e) { }
    }

    @Test(expected = IOException.class)
    public void testSanityTestAlternative(){
        openNonexistingFile();
    }
}
Remember this slide?
You can do much better!

```java
FileInputStream fileInput;
try {
    FileInputStream fileInput = new FileInputStream(filename);
    DataInput dataInput = new DataInputStream(fileInput);
    return dataInput.readInt();
} catch (FileNotFoundException e) {
    System.out.println("Could not open file " + filename);
} catch (IOException e) {
    System.out.println("Couldn’t read file: " + e);
} finally {
    if (fileInput != null) fileInput.close();
}
```
Manual resource termination is ugly and error prone

• Even good programmers usually get it wrong
  – Sun’s guide to Persistent Connections got it wrong in code that claimed to be exemplary
  – Solution on page 88 of Bloch and Gafter’s *Java Puzzlers* is badly broken; no one noticed for years

• 70% of the uses of the `close` method in the JDK itself were wrong in 2008(!)

• Even “correct” idioms for manual resource management are deficient
The solution: try-with-resources

Automatically closes resources

```java
try (DataInput dataInput = new DataInputStream(  
    new FileInputStream(filename))) {
    return dataInput.readInt();
} catch (FileNotFoundException e) {
    System.out.println("Could not open file " + filename);
} catch (IOException e) {
    System.out.println("Couldn’t read file: " + e);
}
```
static void copy(String src, String dest) throws IOException {
    InputStream in = new FileInputStream(src);
    try {
        OutputStream out = new FileOutputStream(dest);
        try {
            byte[] buf = new byte[8 * 1024];
            int n;
            while ((n = in.read(buf)) >= 0)
                out.write(buf, 0, n);
        } finally {
            out.close();
        }
    } finally {
        in.close();
    }
}
static void copy(String src, String dest) throws IOException {
    try (InputStream in = new FileInputStream(src);
         OutputStream out = new FileOutputStream(dest)) {
        byte[] buf = new byte[8 * 1024];
        int n;
        while ((n = in.read(buf)) >= 0)
            out.write(buf, 0, n);
    }
}