



# Chapter 5: Reusability-Oriented Software Construction Approaches

## 5.2 Construction for Reuse

Ming Liu

March 27, 2019

# Outline

- **Designing reusable classes**
  - Inheritance and overriding
  - Behavioral subtyping and overloading
  - Parametric polymorphism and generic programming
  - Composition and delegation
  
- **Designing system-level reusable libraries and frameworks**
  - API and Library
  - Framework
  - Java Collections Framework (an example)



## (e) Behavioral subtyping and Liskov Substitution Principle (LSP)



# Behavioral subtyping

- 기반 타입은 서브 타입으로 대체할 수 있어야 한다.
- 자식 타입들은 부모 타입들이 사용되는 곳에 대체될 수 있어야 한다.

- 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
- **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

**Liskov  
Substitution  
Principle  
(LSP)**

# Behavioral subtyping (LSP)

- Subclass fulfills the same invariants (and additional ones)
- Overridden method has the same pre and postconditions

```
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() { ... }  
}
```

# Behavioral subtyping (LSP)

- Subclass fulfills the same invariants (and additional ones)
- Overridden method start has weaker precondition
- Overridden method brake has stronger postcondition

```
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() { ... }
}
```

# Behavioral subtyping (LSP)

- How about these two classes? Is LSP satisfied?

```
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);
    }
}
```

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

**Invalidates stronger invariant  
( $w==h$ ) in subclass**

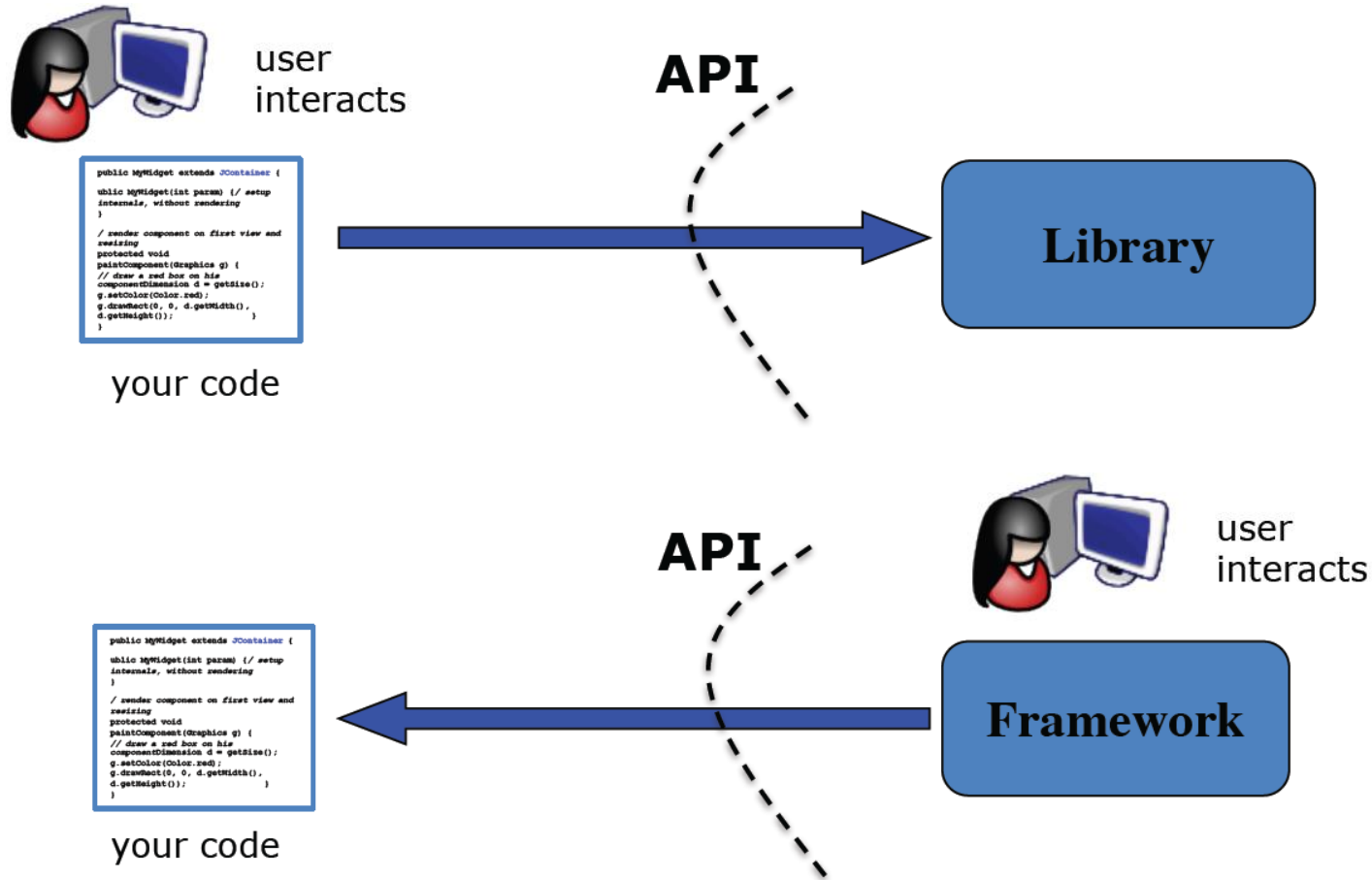


## 2 Designing system-level reusable libraries and frameworks



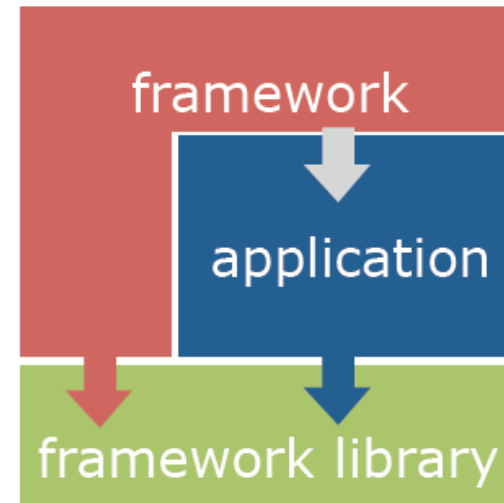


# General distinction: Library vs. framework



# Libraries and frameworks in practice

- Defines key abstractions and their interfaces
- Defines object interactions & invariants
- Defines flow of control
- Provides architectural guidance
- Provides defaults



# More terms

- **API: Application Programming Interface**, the interface of a library or framework
- **Client:** The code that uses an API
- **Plugin:** Client code that customizes a framework
- **Extension point:** A place where a framework supports extension with a plugin
- **Protocol:** The expected sequence of interactions between the API and the client
- **Callback:** A plugin method that the framework will call to access customized functionality
- **Lifecycle method:** A callback method that gets called in a sequence according to the protocol and the state of the plugin




# (1) API design



# Why is API design important?

- **If you program, you are an API designer, and APIs can be among your greatest assets**
  - Good code is modular – each module has an API
  - Users invest heavily: acquiring, writing, learning
  - Thinking in terms of APIs improves code quality
  - Successful public APIs capture users
- **Can also be among your greatest liabilities**
  - Bad API can cause unending stream of support calls
  - Can inhibit ability to move forward
- **Public APIs are forever – one chance to get it right**
  - Once module has users, can't change API at will

# Characteristics of a good API

- 
- Easy to learn
  - Easy to use, even without documentation
  - Hard to misuse
  - Easy to read and maintain code that uses it
  - Sufficiently powerful to satisfy requirements
  - Easy to evolve  
발달하다
  - Appropriate to audience

# (1) API should do one thing and do it well

- **Functionality should be easy to explain**
  - If it's hard to name, that's generally a bad sign
  - Good names drive development
  - Be amenable to splitting and merging modules
- **Good:** Font, Set, PrivateKey, Lock, ThreadFactory, TimeUnit, Future<T>
- **Bad:**
  - DynAnyFactoryOperations
  - \_BindingIteratorImplBase
  - ENCODING\_CDR\_ENCAPS
  - OMGVMCID

## (2) API should be as small as possible but no smaller

- **API should satisfy its requirements**
- **When in doubt leave it out**
  - Functionality, classes, methods, parameters, etc.
  - You can always add, but you can never remove
- **Conceptual weight more important than bulk**
- **Look for a good power-to-weight ratio**  
비율



### (3) Implementation should not impact API

- **Implementation details in APIs are harmful**
  - Confuse users
  - Inhibit freedom to change implementation
- **Be aware of what is an implementation detail**
  - Do not overspecify the behavior of methods
- **For example: do not specify hash functions**
  - All tuning parameters are suspect
- **Don't let implementation details “leak” into API**
  - Serialized forms, exceptions thrown
- **Minimize accessibility of everything (information hiding)**
  - Make classes, members as private as possible
  - Public classes should have no public fields

## (4) Documentation matters

- **Document every class, interface, method, constructor, parameter, and exception**
  - Class: what an instance represents
  - Method: contract between method and its client
- **Preconditions, postconditions, side-effects**
  - Parameter: indicate units, form, ownership
- **Document thread safety**
- **If class is mutable, document state space**

Recall Chapter 4 for Understandability

Reuse is something that is far easier to say than to do. Doing it requires both good design and very good documentation. Even when we see good design, which is still infrequently, we won't see the components reused without good documentation.

– D. L. Parnas  
Software Aging, on ICSE 1994

## (5) Consider performance consequences

- **Bad decisions can limit performance**
  - Making type mutable
  - Providing constructor instead of static factory
  - Using implementation type instead of interface
- **Do not warp API to gain performance**
  - Underlying performance issue will get fixed, but headaches will be with you forever
- **Good design usually coincides with good performance**
- **Performance effects of a bad API decisions can be real and permanent**
  - `Component.getSize()` returns `Dimension`, but `Dimension` is mutable, thus each `getSize` call must allocate `Dimension`, causing millions of needless object allocations

## (6) API must coexist peacefully with platform

공존하다

- **Do what is customary**
  - Obey standard naming conventions
  - Avoid obsolete parameter and return types
  - Mimic patterns in core APIs and language
- **Take advantage of API-friendly features**
  - Generics, varargs, enums, functional interfaces
- **Know and avoid API traps and pitfalls**
  - Finalizers, public static final arrays, etc.
- **Don't transliterate APIs**

## (7) Class design

- **Minimize mutability:** Classes should be immutable unless there's a good reason to do otherwise
  - **Advantages:** simple, thread-safe, reusable
  - **Disadvantage:** separate object for each value
  - If mutable, **keep state-space small**, well-defined.
- **Subclass only where it makes sense:** Subclassing implies substitutability (LSP)
  - Don't subclass unless an is-a relationship exists. Otherwise, use delegation or composition.
  - Don't subclass just to reuse implementation.
  - Inheritance violates encapsulation, and subclasses are sensitive to implementation details of superclass

## (8) Method design

- **Don't make the client do anything the module could do**

- Clients generally do via cut-and-paste, which is ugly, annoying, and error-prone.

```
import org.w3c.dom.*;
import java.io.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;

/** DOM code to write an XML document to a specified output stream. */
static final void writeDoc(Document doc, OutputStream out)throws IOException{
    try {
        Transformer t = TransformerFactory.newInstance().newTransformer();
        t.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM, doc.getDoctype().getSystemId());
        t.transform(new DOMSource(doc), new StreamResult(out)); // Does actual writing
    } catch(TransformerException e) {
        throw new AssertionError(e); // Can't happen!
    }
}
```

- **APIs should fail fast: report errors as soon as possible. Compile time is best – static typing, generics.**

- At runtime, first bad method invocation is best
- Method should be failure-atomic

## (8) Method design

- **Provide programmatic access to all data available in string form.** Otherwise, clients will parse strings, which is painful for clients
- **Overload with care.** Often better to use a different name.
- **Use appropriate parameter & return types.**
  - Favor interface types over classes for input for flexibility, performance
  - Use most specific possible input parameter type, thus moves error from runtime to compile time.
- **Avoid long parameter lists.** Three or fewer parameters is ideal.
  - What if you have to use many parameters?
- **Avoid return values that demand exceptional processing.** Return zero-length array or empty collection, not null.

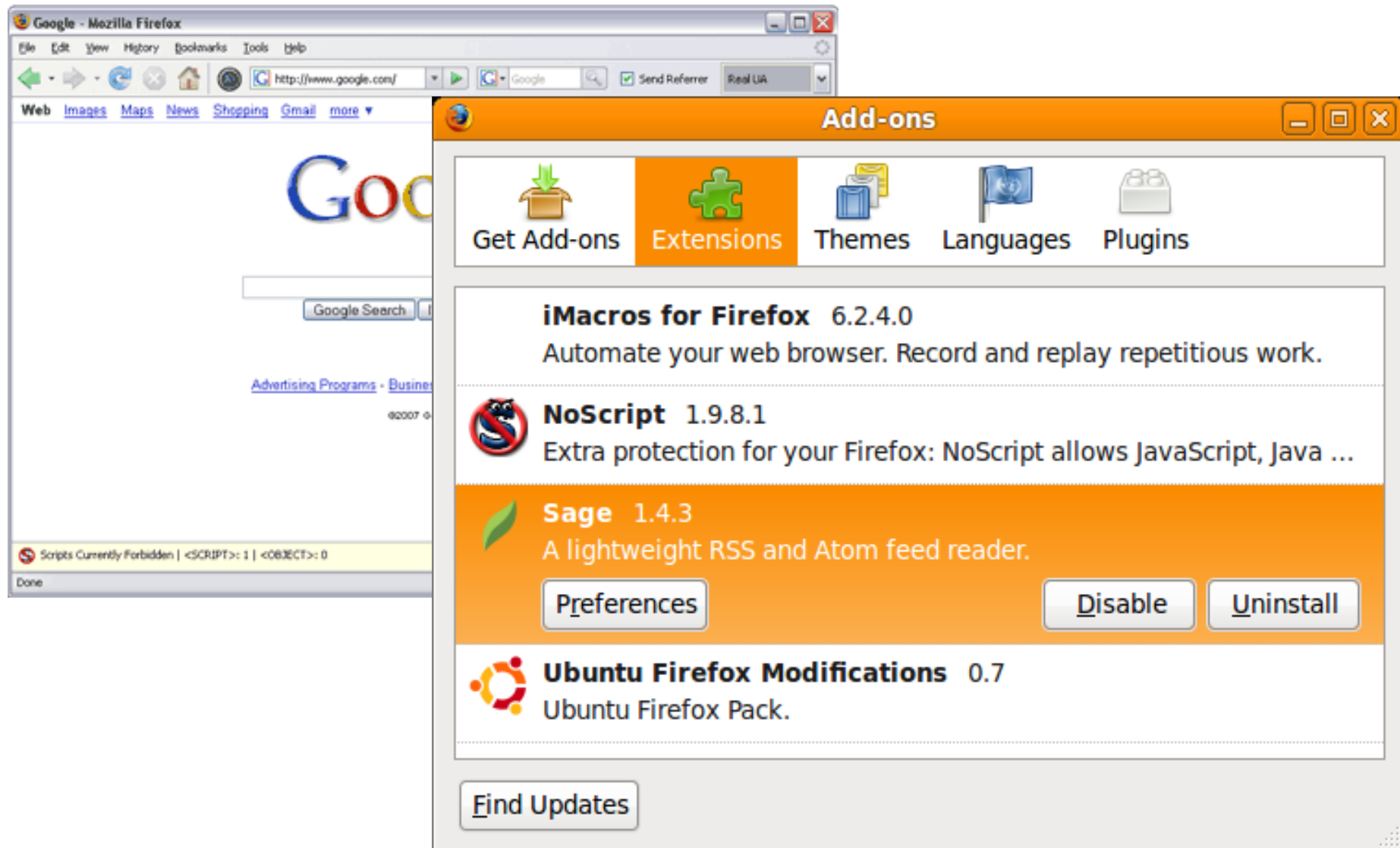


## (2) Framework design





# An example: web browser plugin



# Whitebox and Blackbox frameworks

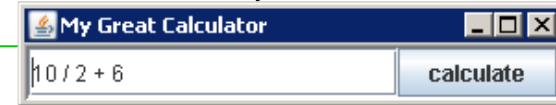
## ■ Whitebox frameworks

- Extension via **subclassing** and **overriding** methods
- Common design pattern(s): Template Method
- Subclass has main method but gives control to framework

## ■ Blackbox frameworks

- Extension via implementing a **plugin interface**
- Common design pattern(s): Strategy, Observer
- Plugin-loading mechanism loads plugins and gives control to the framework

# A calculator example (without a framework)



```
public class Calc extends JFrame {
    private JTextField textField;
    public Calc() {
        JPanel contentPane = new JPanel(new BorderLayout());
        contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));
        JButton button = new JButton();
        button.setText("calculate");
        contentPane.add(button, BorderLayout.EAST);
        textField = new JTextField("");
        textField.setText("10 / 2 + 6");
        textField.setPreferredSize(new Dimension(200, 20));
        contentPane.add(textfield, BorderLayout.WEST);
        button.addActionListener(/* calculation code */);
        this.setContentPane(contentPane);
        this.pack();
        this.setLocation(100, 100);
        this.setTitle("My Great Calculator");
        ...
    }
}
```

# A simple whitebox framework

```
public abstract class Application extends JFrame {
    protected String getApplicationTitle() { return ""; }
    protected String getButtonText() { return ""; }
    protected String getInitialText() { return ""; }
    protected void buttonClicked() { }
    private JTextField textField;
    public Application() {
        JPanel contentPane = new JPanel(new BorderLayout());
        contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));
        JButton button = new JButton();
        button.setText(getButtonText());
        contentPane.add(button, BorderLayout.EAST);
        textField = new JTextField("");
        textField.setText(getInitialText());
        textField.setPreferredSize(new Dimension(200, 20));
        contentPane.add(textField, BorderLayout.WEST);
        button.addActionListener((e) -> { buttonClicked(); });
        this.setContentPane(contentPane);
        this.pack();
        this.setLocation(100, 100);
        this.setTitle(getApplicationTitle());
        ...
    }
}
```

# Using the whitebox framework

```
public class Calculator extends Application {
    protected String getApplicationTitle() { return "My Great Calculator"; }
    protected String getButtonText() { return "calculate"; }
    protected String getInititalText() { return "(10 - 3) * 6"; }
    protected void buttonClicked() {
        JOptionPane.showMessageDialog(this, "The result of " + getInput() +
            " is " + calculate(getInput()));
    }
    private String calculate(String text) { ... }
}
```

Extension via subclassing and overriding methods  
Subclass has main method but gives control to framework

```
public class Ping extends Application {
    protected String getApplicationTitle() { return "Ping"; }
    protected String getButtonText() { return "ping"; }
    protected String getInititalText() { return "127.0.0.1"; }
    protected void buttonClicked() { ... }
}
```

# An example blackbox framework

```

public class Application extends JFrame {
    private JTextField textField;
    private Plugin plugin;
    public Application() { }
    protected void init(Plugin p) {
        p.setApplication(this);
        this.plugin = p;
        JPanel contentPane = new JPanel(new BorderLayout());
        contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));
        JButton button = new JButton();
        button.setText(plugin != null ? plugin.getButtonText() : "ok");
        contentPane.add(button, BorderLayout.EAST);
        textField = new JTextField("");
        if (plugin != null)
            textField.setText(plugin.getInititalText());
        textField.setPreferredSize(new Dimension(200, 20));
        contentPane.add(textField, BorderLayout.WEST);
        if (plugin != null)
            button.addActionListener((e) -> { plugin.buttonClicked(); } );
        this.setContentPane(contentPane);
        ...
    }
    public String getInput() { return textField.getText(); }
}

```

```

public interface Plugin {
    String getApplicationTitle();
    String getButtonText();
    String getInititalText();
    void buttonClicked() ;
    void setApplication(Application app);
}

```

# Using the blackbox framework

```
public class CalcPlugin implements Plugin {
    private Application app;
    public void setApplication(Application app) { this.app = app; }
    public String getButtonText() { return "calculate"; }
    public String getInititalText() { return "10 / 2 + 6"; }
    public void buttonClicked() {
        JOptionPane.showMessageDialog(null, "The result of "
            + application.getInput() + " is "
            + calculate(application.getInput()));
    }
    public String getApplicationTitle() { return "My Great Calculator"; }
}
```

Extension via implementing a plugin interface  
Plugin-loading mechanism loads plugins and gives  
control to the framework

# Whitebox vs. Blackbox Frameworks

- **Whitebox frameworks use subclassing**

- Allows extension of every nonprivate method
- Need to understand implementation of superclass
- Only one extension at a time
- Compiled together
- Often so-called developer frameworks

- **Blackbox frameworks use composition**

- Allows extension of functionality exposed in interface
- Only need to understand the interface
- Multiple plugins
- Often provides more modularity
- Separate deployment possible (.jar, .dll, ...)
- Often so-called end-user frameworks, platforms



# Typical framework design and implementation

- **Define your domain**
  - Identify potential common parts and variable parts
  - Design and write sample plugins/applications
- **Factor out & implement common parts as framework**
- **Provide plugin interface & callback mechanisms for variable parts**
  - Use well-known design principles and patterns where appropriate...
- **Get lots of feedback, and iterate**
  
- **This is usually called “Domain Engineering”.**

# Evolutionary design: Extract Commonalities

- **Extracting interfaces is a new step in evolutionary design:**
  - Abstract classes are discovered from concrete classes
  - Interfaces are distilled from abstract classes
- **Start once the architecture is stable**
  - Remove non-public methods from class
  - Move default implementations into an abstract class which implements the interface

# Running a framework

- **Some frameworks are runnable by themselves**
  - e.g. Eclipse
- **Other frameworks must be extended to be run**
  - Swing, JUnit, MapReduce, Servlets
- **Methods to load plugins:**
  - Client writes main(), creates a plugin and passes it to framework
  - Framework writes main(), client passes name of plugin as a command line argument or environment variable
  - Framework looks in a magic location, and then config files or .jar files are automatically loaded and processed.

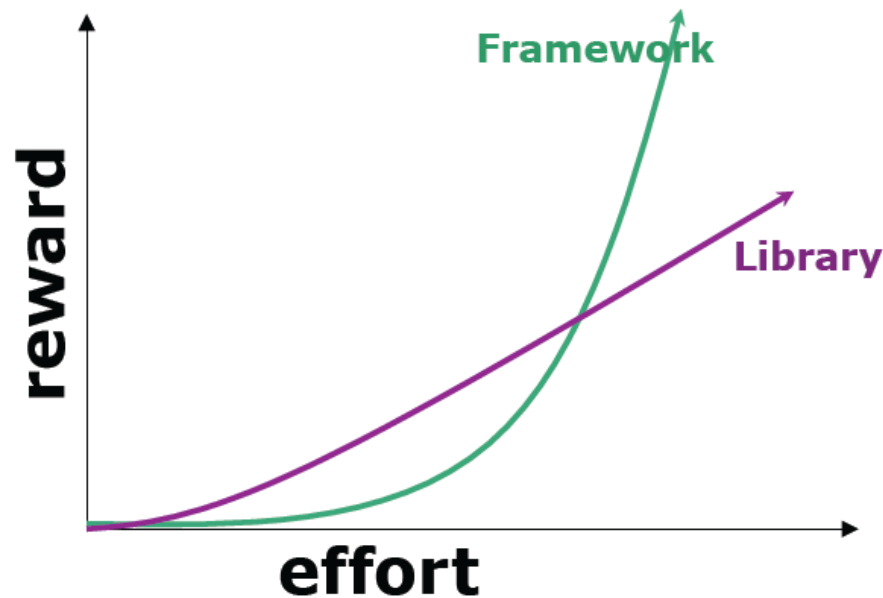
# Example: A JUnit Plugin

```
public class SampleTest {  
    private List<String> emptyList;  
  
    @Before  
    public void setUp() {  
        emptyList = new ArrayList<String>();  
    }  
  
    @After  
    public void tearDown() {  
        emptyList = null;  
    }  
  
    @Test  
    public void testEmptyList() {  
        assertEquals("Empty list should have 0 elements",  
            0, emptyList.size());  
    }  
}
```

In JUnit the plugin mechanism is Java annotations

# Learning a framework

- Documentation
- Tutorials, wizards, and examples
- Other client applications and plugins
- Communities, email lists and forums





## (3) Java Collections Framework

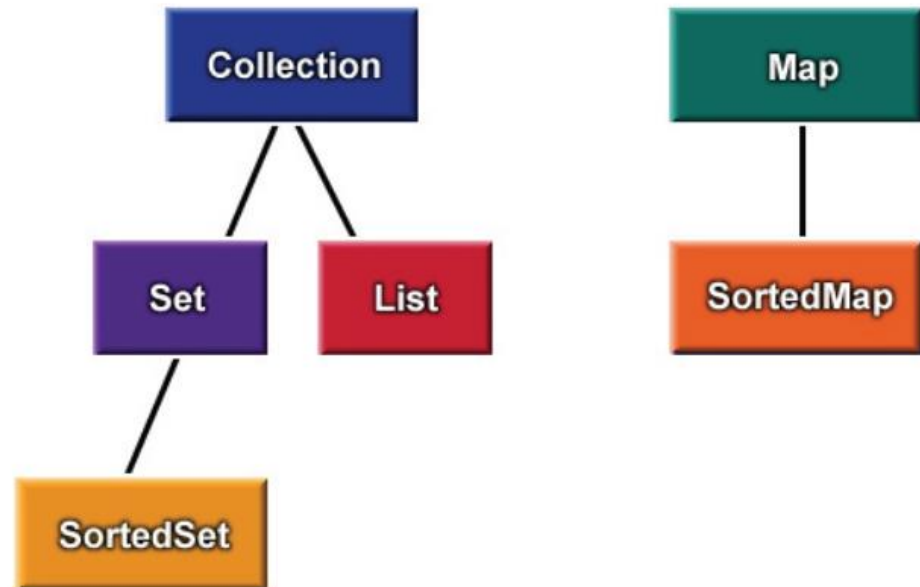


# What is a Collection and Collection Framework?

- **Collection:** an object that groups elements
- **Main Uses:** data storage and retrieval, and data transmission
- **Familiar Examples**
  - `java.util.Vector`
  - `java.util.Hashtable`
  - `Array`
- **Collection Framework:** a Unified Architecture for
  - Interfaces - implementation-independence
  - Implementations - reusable data structures
  - Algorithms - reusable functionality
- **Best-known examples**
  - C++ Standard Template Library (STL)

# Architecture Overview

- Core Collection Interfaces
- General-Purpose Implementations
- Wrapper Implementations
- Abstract Implementations
- Algorithms





# Collection Interface

```
public interface Collection<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element); // Optional
    boolean remove(Object element); // Optional
    Iterator<E> iterator();
    Object[] toArray();
    T[] toArray(T a[]);

    // Bulk Operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? Extends E> c); // Optional
    boolean removeAll(Collection<?> c); // Optional
    boolean retainAll(Collection<?> c); // Optional
    void clear(); // Optional
}
```

# Iterator Interface

반복하다

## ■ Replacement for Enumeration interface

- Adds remove method
- Improves method names

```
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove(); // Optional  
}
```

# Set Interface

- Adds no methods to Collection!
- Adds stipulation: no duplicate elements
- Mandates equals and hashCode calculation

```
public interface Set<E> extends Collection<E> {  
}
```

- Set Idioms:  
Set<Type> s1, s2;  
boolean **isSubset** = s1.containsAll(s2);  
Set<Type> **union** = new HashSet<>(s1);  
union = union.addAll(s2);  
Set<Type> **intersection** = new HashSet<>(s1);  
intersection.retainAll(s2);  
Set<Type> **difference** = new HashSet<>(s1);  
difference.removeAll(s2);  
Collection<Type> c;  
Collection<Type> **noDups** = new HashSet<>(c);

# List Interface: a sequence of objects

```
public interface List<E> extends Collection<E> {  
  
    E get(int index);  
    E set(int index, E element);           // Optional  
    void add(int index, E element);        // Optional  
    Object remove(int index);              // Optional  
    boolean addAll(int index, Collection<? extends E> c);  
                                           // Optional  
  
    int indexOf(Object o);  
    int lastIndexOf(Object o);  
  
    List<E> subList(int from, int to);  
  
    ListIterator<E> listIterator();  
    ListIterator<E> listIterator(int index);  
}
```

# List Example

- **Reusable algorithms to swap and randomize:**

```
public static <E> void swap(List<E> a, int i, int j) {  
    E tmp = a.get(i);  
    a.set(i, a.get(j));  
    a.set(j, tmp);  
}  
private static Random r = new Random();  
public static void shuffle(List<?> a) {  
    for (int i = a.size(); i > 1; i--)  
        swap(a, i - 1, r.nextInt(i));  
}
```

- **List Idioms:**  
List<Type> a, b;  
a.addAll(b); // Concatenate two lists  
a.subList(from, to).clear(); // Range-remove  
// Range-extract  
List<Type> partView = a.subList(from, to);  
List<Type> part = new ArrayList<>(partView);  
partView.clear();

# Map Interface: a Key-Value mapping

```
public interface Map<K,V> {  
  
    int size();  
    boolean isEmpty();  
    boolean containsKey(Object key);  
    boolean containsValue(Object value);  
  
    Object get(Object key);  
    Object put(K key, V value);           // Optional  
    Object remove(Object key);           // Optional  
    void putAll(Map<? Extends K, ? Extends V> t); // Optional  
    void clear();                         // Optional  
  
    // Collection Views  
    public Set<K> keySet();  
    public Collection<V> values();  
    public Set<Map.Entry<K,V>> entrySet();  
}
```

# Map Idioms

```
// Iterate over all keys in Map m
Map<Key, Val> m;
for (iterator<Key> i = m.keySet().iterator(); i.hasNext(); )
    System.out.println(i.next());

// As of Java 5 (2004)
for (Key k : m.keySet())
    System.out.println(i.next());

// "Map algebra"
Map<Key, Val> a, b;
boolean isSubMap = a.entrySet().containsAll(b.entrySet());
Set<Key> commonKeys =
    new HashSet<>(a.keySet()).retainAll(b.keySet()); [sic!]

//Remove keys from a that have mappings in b
a.keySet().removeAll(b.keySet());
```

# General Purpose Implementations

- Consistent Naming and Behavior

		Implementations			
		Hash Table	Resizable Array	Balanced Tree	Linked List
Interfaces	Set	HashSet		TreeSet	
	List		ArrayList		Linked List
	Map	HashMap		TreeMap	



# Choosing an Implementation

## ■ Set

- HashSet --  $O(1)$  access, no order guarantee
- TreeSet --  $O(\log n)$  access, sorted

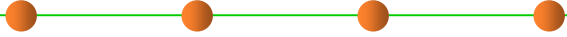
## ■ Map

- HashMap -- (See HashSet)
- TreeMap -- (See TreeSet)

## ■ List

- ArrayList --  $O(1)$  random access,  $O(n)$  insert/remove
- LinkedList --  $O(n)$  random access,  $O(1)$  insert/remove;
  - Use for queues and dequeues (no longer a good idea!)

# Unmodifiable Wrappers

- 
- Anonymous implementations
  - Static factory methods
  - One for each core interface
  - Provide read-only access – Immutable!
- 
- **We have discussed these wrapper in Chapter 3**

# Synchronization Wrappers

- **Not thread-safe!**
- **Synchronization Wrappers: a new approach to thread safety**
  - Anonymous implementations, one per core interface
  - Static factories take collection of appropriate type
  - Thread-safety assured if all access through wrapper
  - Must manually synchronize iteration
- **It was new then; it's old now!**
  - Synch wrappers are largely obsolete
  - Made obsolete by concurrent collections
- **To be discussed in Chapter 10**

# Synchronization Wrapper Example


```
Set<String> s = Collections.synchronizedSet(new
HashSet<>());
...
s.add("wombat"); // Thread-safe
...
synchronized(s) {
    Iterator<String> i = s.iterator(); // In synch block!
    while (i.hasNext())
        System.out.println(i.next());
}

// In Java 5 (post-2004)
synchronized(s) {
    for (String t : s)
        System.out.println(i.next());
}
```

# Convenience Implementations

- `Arrays.asList(E[] a)`
  - Allows array to be "viewed" as `List`
  - Bridge to Collection-based APIs
- `EMPTY_SET`, `EMPTY_LIST`, `EMPTY_MAP`
  - Immutable constants
- `singleton(E o)`
  - Immutable set with specified object
- `nCopies(E o)`
  - Immutable list with `n` copies of object

# Reusable Algorithms



```
static <T extends Comparable<? super T>> void sort(List<T> list);
static int binarySearch(List list, Object key);
static <T extends Comparable<? super T>> T min(Collection<T> coll);
static <T extends Comparable<? super T>> T max(Collection<T> coll);
static <E> void fill(List<E> list, E e);
static <E> void copy(List<E> dest, List<? Extends E> src);
static void reverse(List<?> list);
static void shuffle(List<?> list);
```

# Example 1: Sorting lists of comparable elements

```
List<String> strings; // Elements type: String
...
Collections.sort(strings); // Alphabetical order

LinkedList<Date> dates; // Elements type: Date
...
Collections.sort(dates); // Chronological order

// Comparable interface (Infrastructure)
public interface Comparable<E> {
    int compareTo(Object o);
}
```

## Algorithm Example 2: Sorting with a comparator

```
List<String> strings; // Element type: String
Collections.sort(strings, Collections.ReverseOrder());

// Case-independent alphabetical order
static Comparator<String> cia = new Comparator<>() {
    public int compare(String c1, String c2) {
        return c1.toLowerCase().compareTo(c2.toLowerCase());
    }
};

Collections.sort(strings, cia);

public interface Comparator<T> {
    public int compare(T o1, T o2);
}
```



# Compatibility

- **Old and new collections interoperate freely**
- **Upward Compatibility**
  - `Vector<E>` implements `List<E>`
  - `Arrays.asList(myArray)`
- **Backward Compatibility**
  - `myCollection.toArray()`
  - `new Vector<>(myCollection)`
  - `new Hashtable<>(myMap)`
  - `Hashtable<K,V>` implements `Map<K,V>`

# API Design Guidelines

- **Avoid ad hoc collections**
  - Input parameter type:
    - Any collection interface (Collection, Map best)
    - Array may sometimes be preferable
  - Output value type:
    - Any collection interface or class
    - Array
  
- **Provide adapters for your legacy collections**



The end

March 27, 2019