Interfaces in Java 8

- *Functional interface*: a special type of interface that has a single *abstract* (or empty) method.
  - *Non-static* and *non-default* method
- Before Java 8, all methods defined in an interface were abstract.
  - `public interface Foo{ public void Boo() }`
  - `public interface Comparator<T>{ public int compare(T o1, T o2) }`
  - No methods could have their bodies (ipmls) in an interface.
- Java 8
  - Introduces 2 extra types of methods to interfaces: *static methods* and *default methods*.
  - Calls traditional abstract/empty methods as *abstract methods*.
- `Comparator<T>` in Java 8 has…
  - one abstract method (`compare()`)
  - many *static* and *default* methods.

Abstract Interface Methods

- Java 8 introduces the keyword `abstract`.
  - `public interface Foo{ public abstract void Boo() }`
  - `abstract` can be omitted.
    - `public interface Comparator<T>{ public int compare(T o1, T o2) }`
    - `public interface Comparator<T>{ public abstract int compare(T o1, T o2) }`

Static Interface Methods

```
public interface I1{
    public static int getValue(){ return 123; } }
I1.getValue();      // Returns 123.

public interface I2 extends I1{}
I2.getValue();      // Cannot inherit it. Compilation error.

public interface I2 extends I1{}
    public static int getValue(){ return 987; } }
I2.getValue();      // Can override it. Returns 987.

public class C1 implements I1{}
C1.getValue();      // Results in a compilation error.
```

- Can call a static method of an interface without a class that implements the interface.
  - Classes never implement/have static interface methods.
- There are some restrictions.

Default Interface Methods

```
public interface I1{
    public default int getValue(){ return 123; } }
I1.getValue();  // Cannot call it like a static method. Compilation error.

public class C1 implements I1{}
C1 c = new C1();
c.getvalue();   // Returns 123.

public interface I2 extends I1{}
public class C2 implements I2{}
C2 c = new C2();
c.getvalue();   // Can inherit it. Returns 123.

public interface I2 extends I1{
    public default int getValue(){ return 987; } }
public class C2 implements I2{
C2 c = new C2();
c.getvalue();   // Can override it. Returns 987.

public class C1 implements I1{
    public int getValue(){ return 987; } }
C1 c = new C1();
c.getvalue();   // Can override it. Returns 987.
```

```
public interface I1{
    public default int getValue(){ return 123; } }
public class C1{
    public int getValue(){ return 987; } }
public class C2 extends C1 implements I1{}
```

```
C2 c = new C2();
c.getvalue();     // Returns 987.
```

– A super class's method precedes an interface's default method.

```
public class C2 extends C1 implements I1{
    public int getValue(){
        I1.super.getValue(); } }
```

```
C2 c = new C2();
c.getValue(); // Returns 123.
```



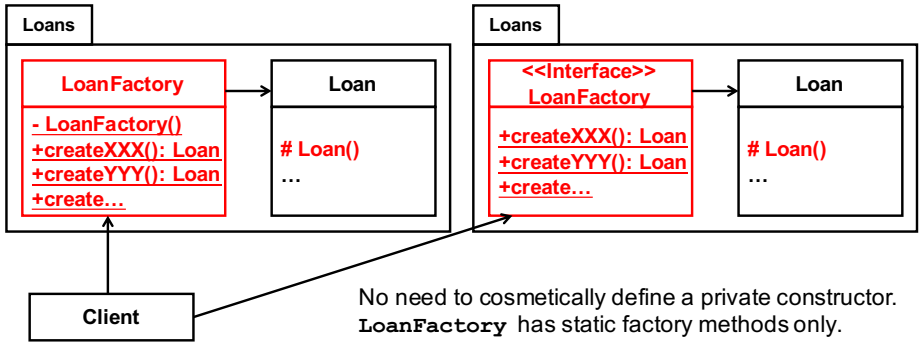
```
public interface I1{
    public default int getValue(){ return 123; } }
public interface I2 {
    public default int getValue(){ return 987; } }
public class C1 implements I1, I2{}     // Compilation error.
```

– Default methods from different interfaces conflict.

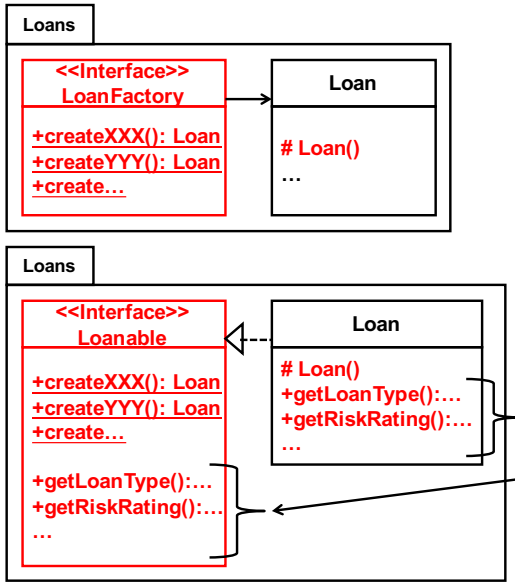
```
public class C1 implements I1, I2{
    public int getValue(){
        return I1.super.getValue(); } }    // Returns 123.
```


Examples: Static Interface Methods

- Factory methods to create an object that implements an interface
  - A variant of the Static Factory Method design pattern
    - C.f. CS680 lecture note



No need to cosmetically define a private constructor.
`LoanFactory` has static factory methods only.



LoanFactory has static factory methods only.

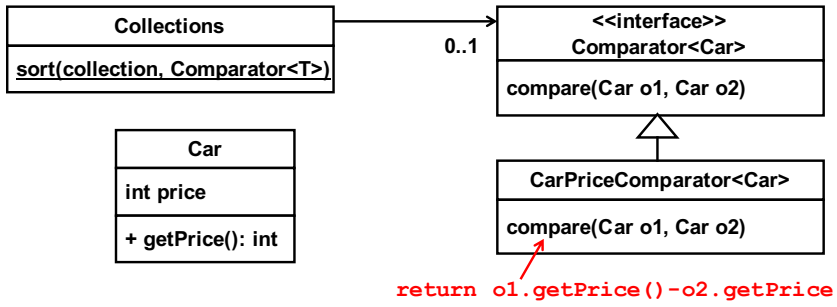
Loanable can define loan-specific method signatures, if you want. Then, Loan implements them.

- `java.util.Comparator<T>` has…
  - one abstract method (`compare()`) and
  - many *static* and default methods.
  - `static Comparator<T> comparing(Function<T, R> keyExtractor)`
    - Accepts a function that extracts a `Comparable` sort key from T
      - Sort key: data/value to be used in ordering/sorting
      - `Function<T, R>`
        - » Represents a function that accepts a parameter (T) and returns a result (R).
        - » A functional interface whose abstract method: `R apply(T t)`.
    - Returns a `Comparator<T>`

```
class Car{ int getPrice(); }
Collections.sort(carList, Comparator.comparing(
                          (Car car)-> car.getPrice() );
                         //comparing() returns a Comparator<Car>
```

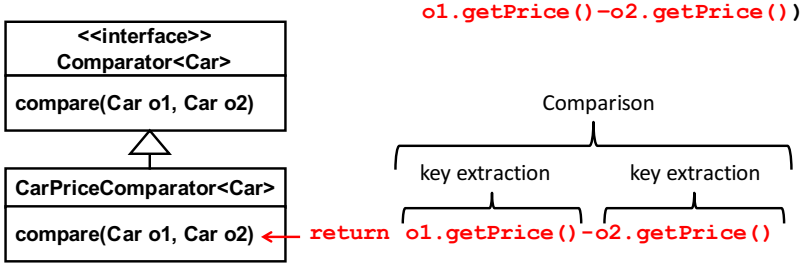


```
class Car{
 int getPrice(); }
Collections.sort(carList,
                 Comparator.comparing(
                     (Car car)-> car.getPrice() );
Collections.sort(carList,
                  (Car o1, Car o2)->
                     o1.getPrice()-o2.getPrice())
```



- What `Comparator.comparing()` does is to
  - Transform a key extraction function to a comparison function
- *Higher-order function*
  - Accepts a function as a parameter and produces another function as a result

```
- class Car{ int price;
             int getPrice();}
  Collections.sort(carList,
                   Comparator.comparing(
                       (Car car)-> car.getPrice() );
- Collections.sort(carList,
                   (Car o1, Car o2)->
                       o1.getPrice()-o2.getPrice());
```



Benefits of Using Lambda Expressions

- More concise code
  - This may or may not mean "easier to understand" depending on how much you are used to lambda expressions.
- Power of functional programming
  - e.g., higher-order functions

A Bit More about Comparator

```
class Car{ int getPrice();}
Collections.sort(carList,
                 Comparator.comparing(
                     (Car car)-> car.getPrice() );
Collections.sort(carList,
                 Comparator.comparing( Car::getPrice ) );
```

- Method references in lambda expressions
  - *object::method*
    - `System.out::println`
      - `System.out` contains an instance of PrintStream.
    - `(int x) -> System.out.println(x)`
  - *Class::staticMethod*
    - `Math::max`
    - `(double x, double y) -> Math.max(x, y)`
  - *Class::method*
    - `Car::getPrice`
    - `(Car car)-> car.getPrice()`
    - `Car::setPrice`
    - `(Car car, int price)-> car.setPrice(price)`


```
class Car{ int getPrice();}
Collections.sort(carList,
                 Comparator.comparing(
                     (Car car)-> car.getPrice() );
Collections.sort(carList,
                 Comparator.comparing( Car::getPrice ) );
Collections.sort(carList,
                 (Car o1, Car o2)->
                     o1.getPrice()-o2.getPrice());
```

  - Ascending order (natural order) by default

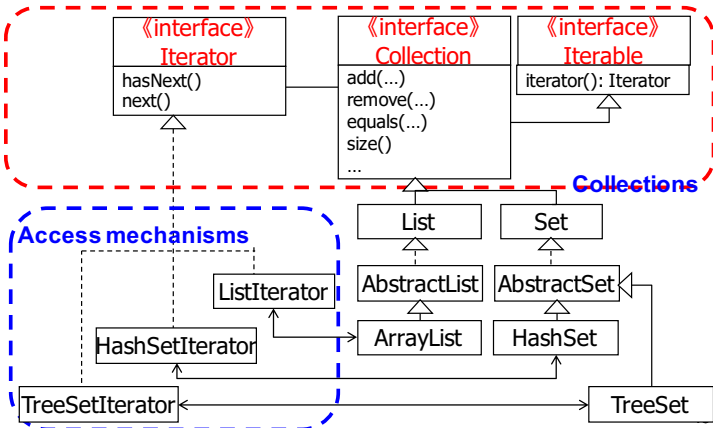
- What if you want descending ordering with comparing()?
  -
```
Collections.sort(carList,
                 Comparator.comparing(Car::getPrice,
                                      Comparator.reverseOrder() );
```
  -
```
Collections.sort(carList,
                 Comparator.comparing(Car::getPrice,
                                      Comparator.naturalOrder() );
```
  -
```
Collections.sort(carList,
                 Comparator.comparing(Car::getPrice).reversed() );
```


Lambda Expressions for Collections

- `java.lang.Iterable<T>`
  - Used to have only one (abstract) method before Java 8
    - `Iterator<T> iterator()`
  - C.f. CS680 lecture note

Users know these three interfaces.

Users do not have to know these access mechanisms. They are hidden from users. API documentation is not available for these classes.

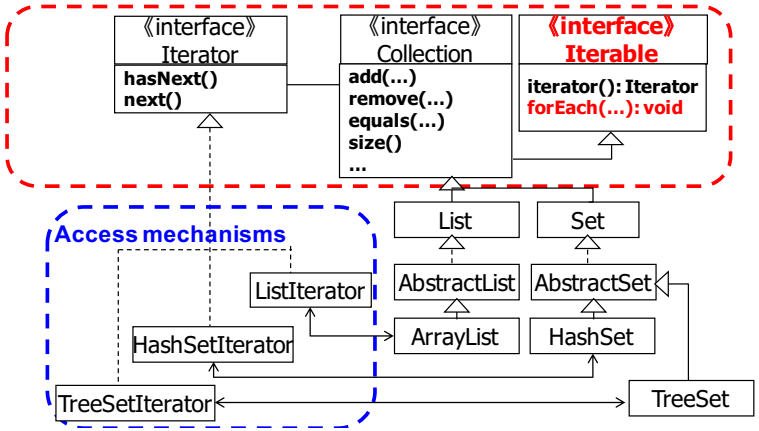



```
ArrayList<String> strList = new ArrayList<>();
strList.add("a"); strList.add("b");
Iterator<ArrayList> iterator = strList.iterator();
while( iterator.hasNext() ) {
    System.out.print(iterator.next()); }
```

```
ArrayList<String> strList = new ArrayList<>();
strList.add("a"); strList.add("b");
for(String str: strList){
    System.out.println(str) }
```

- Note: "for-each" is a syntactic sugar for iterator-based code.

- **`Iterable<T>`**
  - Java 8 introduced new *default* methods.
    - **`default void forEach(Consumer<T> action)`**
      - Performs a given action for each element of a collection, which implements **`Iterable`**.



- **`Iterable<T>`**
  - **`default void forEach(Consumer<T> action)`**
- **`Consumer<T>`**
  - Functional interface
    - Can pass a lambda expression to **`forEach()`**
  - Represents an operation that accepts a parameter (T) and returns no result.
  - Abstract method: **`void accept(T t)`**
    - Performs this operation (i.e., an operation that Consumer<T> represents) on a given parameter (T).

```
ArrayList<String> strList = new ArrayList<>();
strList.add("a"); strList.add("b");
strList.forEach( (String s)->System.out.println(s) );
strList.forEach( System.out::println );
```


- Without a lambda expression
  - ```
    Iterator<ArrayList> iterator = strList.iterator();
    while( iterator.hasNext() ) {
     System.out.print(iterator.next()); }
    ```
  - ```
    for(String str: strList){
        System.out.println(str) }
    ```
- With a lambda expression
  - ```
    strList.forEach( (Integer i)->System.out.println(i) )
    ```
    - ```
      strList.forEach( System.out::println )
      ```


Method References in LEs

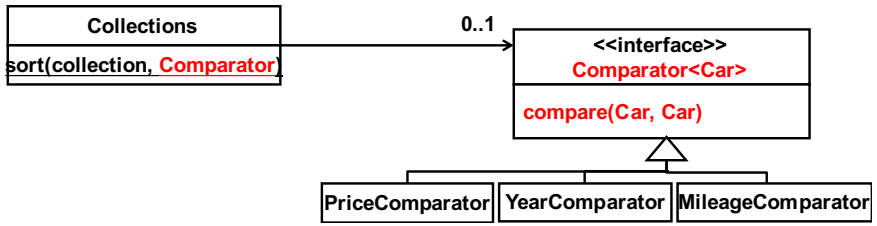
- *object::method*
  - **`System.out::println`**
  - **`(int x)-> System.out.println(x)`**
- *Class::staticMethod*
  - **`Math::max`**
  - **`(double x, double y)-> Math.pow(x, y)`**
- *Class::method*
  - **`Car::setPrice`**
  - **`(Car car, int price)-> car.setPrice(price)`**
  - **`Car::getPrice`**
  - **`(Car car)-> car.getPrice()`**
- *this::method, super::method*
  - **`this::equals`**
  - **`(String s)-> this.equals(s)`**
  - **`super::foo`**
  - **`(String s)-> super.foo(s)`**

HW1

- Refer to HW11-2 of CS680 and work on it with lambda expressions.
  - Implement the Car class

```
class Car{
    private int getPrice();
    private int getYear();
    private float getMileage(); }
```

  - Instead of defining Comparator<Car> and its 3 implementation classes, represent the body of each compare() method with a lambda expression and pass it to Collections.sort().
    - Pass 3 different lambda expressions to Collections.sort()



- [Optional] Implement a lambda expression that performs *Pareto comparison*.
  - C.f. CS680's HW 11-2

- Create several cars (Car instances) and sort them with each of three (or four) comparators.
  - Minimum requirement: ascending order (natural order)
  - [Optional] Do descending ordering as well with `reverseOrder()` or `reserved()` of `Comparator`.

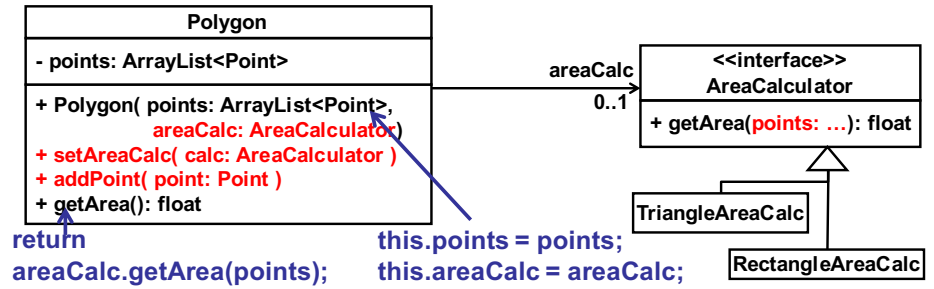
- Submit Java source code and an Ant script (build.xml)
  - No binary code (no libraries, no byte code)
  - No need to submit test cases in CS681.

Implementing *Strategy* with LEs

- Comparator and its implementation classes form the Strategy design pattern
  - C.f. CS680 lecture note #11

- In general, LEs make it possible to implement *Strategy* in a concise way
  - By avoiding to define an implementation class for an interface (or a subclass of a super class) explicitly.

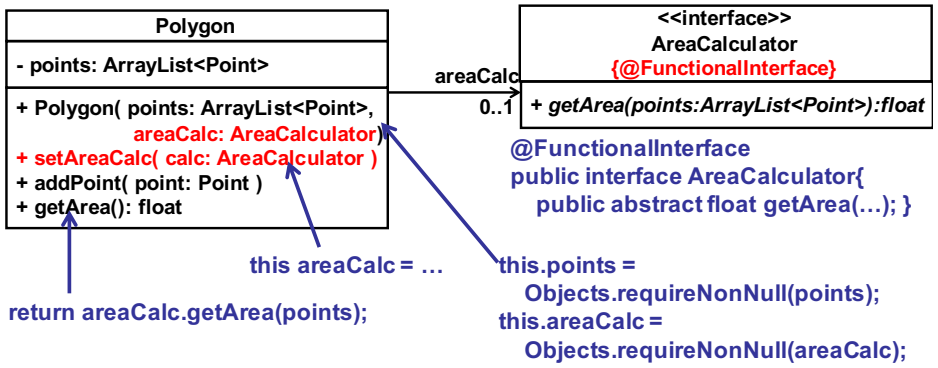
Recap: CS680's HW11-1



User/client of Polygon:

```
ArrayList<Point> al = new ArrayList<Point>();
al.add( new Point(…) ); al.add( new Point(…) ); al.add( new Point(…) );
Polygon p = new Polygon( al, new TriangleAreaCalc() );
p.getArea(); // triangle's area
p.addPoint( new Point(…) );
p.setAreaCalc( new RectangleAreaCalc() );
p.getArea(); // rectangle's area
```

HW2: Use Lambda Expressions



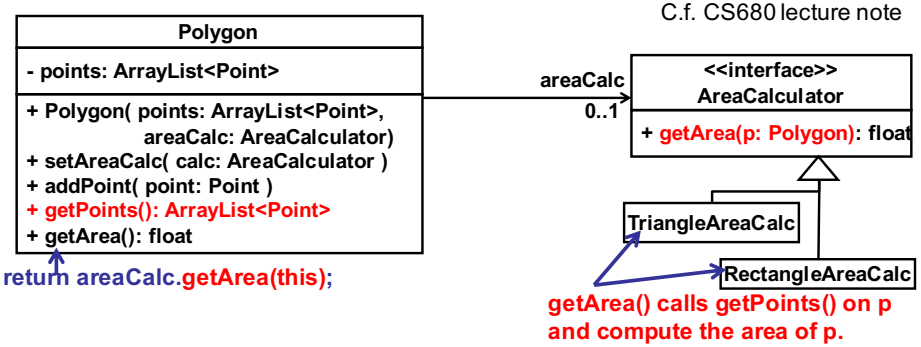
User/client of Polygon:

```
ArrayList<Point> al = …          // add 3 points
Polygon p = new Polygon( al, (ArrayList<Point> points)->{…});
p.getArea(); // triangle's area
p.addPoint( new Point(…) );    // add the 4th point
p.setAreaCalc( (ArrayList<Point> points)->{…} );
p.getArea(); // rectangle's area
```


- Modify your code for CS680's HW11-1 to implement *Strategy* with lambda expressions.
  - C.f. Heron's formula in CS680's HW 2-4
- Use `Objects.requireNonNull()` in Polygon's constructor and `setAreaCalc()`,
  - as shown in the previous slide.
- Transform a triangle to a rectangle.
  - [optional] Transform the rectangle to a triangle later on.

Let's Consider a Variant

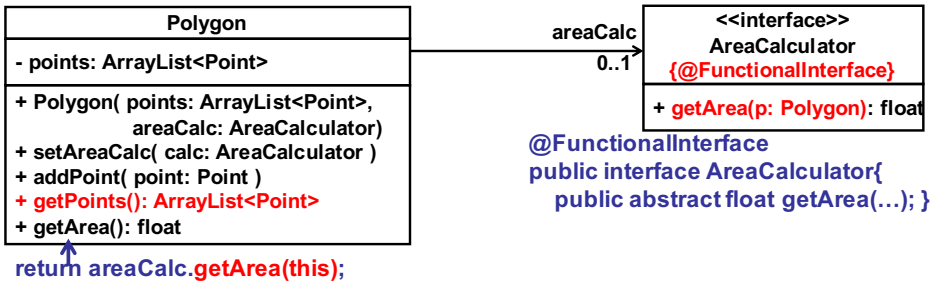
C.f. CS680 lecture note



User/client of Polygon:

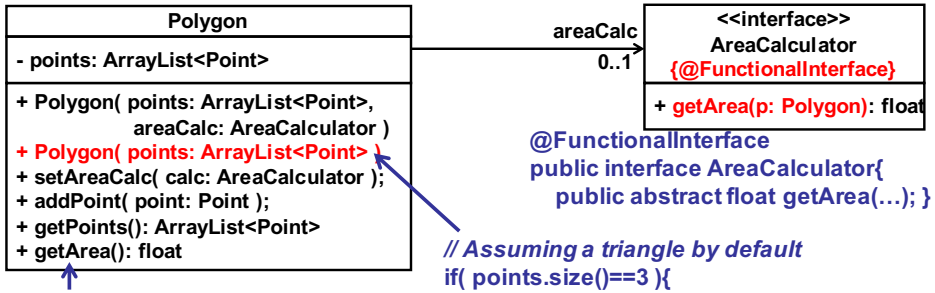
```
ArrayList<Point> al = new ArrayList<Point>();
al.add( new Point(…) ); al.add( new Point(…) ); al.add( new Point(…) );
Polygon p = new Polygon( al, new TriangleAreaCalc() );
p.getArea(); // triangle's area
p.addPoint( new Point(…) );
p.setAreaCalc( new RectangleAreaCalc() );
p.getArea(); // rectangle's area
```


Use LEs to implement *Strategy*



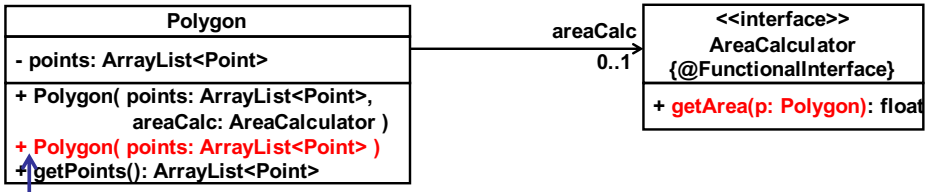
User/client of Polygon:

```
ArrayList<Point> al = …          // add 3 points
Polygon p = new Polygon( al, (Polygon p)->{…});
p.getArea(); // triangle's area
p.addPoint( new Point(…) );    // add the 4th point
p.setAreaCalc( (Polygon p)->{…} );
p.getArea(); // rectangle's area
```



User/client of Polygon:

```
ArrayList<Point> al = …          // add 3 points
Polygon p = new Polygon(al);
p.getArea(); // triangle's area
p.addPoint( new Point(…) );    // add the 4th point
p.setAreaCalc( (Polygon p)->{…} );
p.getArea(); // rectangle's area
```

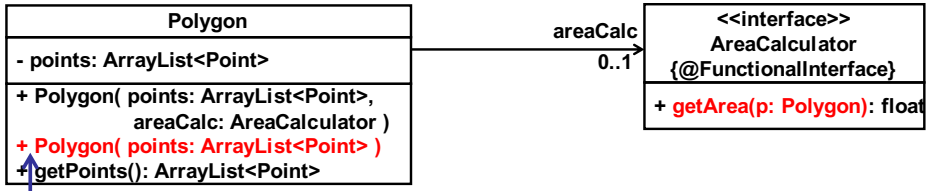



3 Alternatives

```
if( points.size()==3 ){
  Polygon(points,
  (Polygon p)->{…} );
}
```

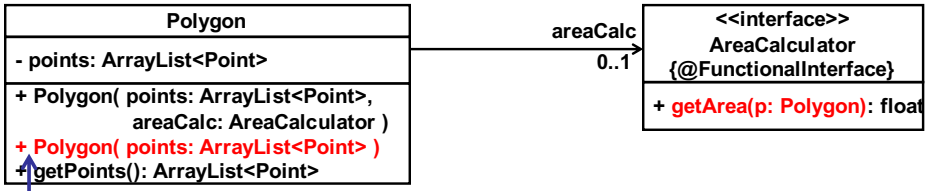
```
(Polygon p)->{
  points.get(0).getX()…; // A parameter of Polygon()
  …
  return …; }

(Polygon p)->{
  this.points.get(0).getX()…; // A data field of Polygon
  …
  return …; }

(Polygon p)->{
  this.getPoints().get(0).getX()…; //Polygon's getPoints()
  …
  return …; }
```


Free Variables




```
…
(Polygon p)->{
  points.get(0).getX()…;
  // A parameter of Polygon()
  …
  return …; }

…
(Polygon p)->{
  this.points.get(0).getX()…;
  // A data field of Polygon
  …
  return …; }
```

- *Free* variables
  - Variables that are not defined in (or bound to) a lambda expression.
    - Not parameters of a LE.
    - Not local variables in a LE.
  - Parameters of an *enclosing* method or data fields of an *enclosing* class




```
(Polygon p)->{
  points.get(0).getX()…;
  // A parameter of Polygon()
  …
  return …; }

(Polygon p)->{
  this.points.get(0).getX()…;
  // A data field of Polygon
  …
  return …; }
```

When a free variable is evaluated, its enclosing lambda expression *captures* its value.

User/client of Polygon:

```
ArrayList<Point> al = …;  // add 3 points
Polygon p = new Polygon(al);
p.getArea(); // triangle's area
```

HW 3

- Implement a variant of your code for HW 2.
  - Use free variables in Polygon().

Closures

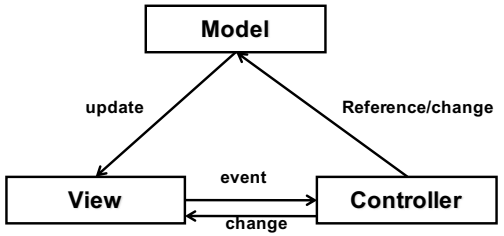
- Closure
  - Code block that can capture a free variable(s)
- Lambda expression in Java
  - Code block that has a parameter(s) and can capture a free variable(s)
- Lambda expressions in Java are closures.

Benefits of Using Lambda Expressions

- Benefits
  - More concise code
    - This may or may not mean "easier to understand" depending on how much you are used to lambda expressions.
  - Power of functional programming
    - e.g., higher-order functions
- Key areas where lambda expressions are beneficial.
  - Callback
    - Comparator, GUI, event handling (e.g., Observer)
  - Collections
    - Newly-added default methods that accept lambda expressions
    - Collection streams
  - Concurrency/parallelism
    - Repeated execution of code block (lambda expression)

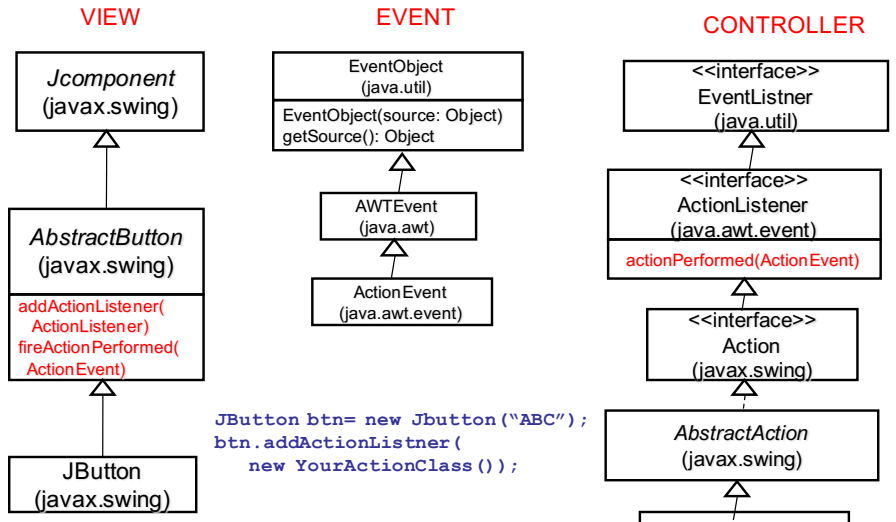
Model-View-Controller (MVC) Architecture

- An architectural design strategy (or architectural pattern) to separate…
  - Model
    - Maintains data or primary business logic
  - View
    - Deals with UI for a model.
    - Visualization/representation of a model to the user
  - Controller
    - Processes inputs/changes from the user to a model.
    - Update the model with the inputs/changes and update its corresponding view.

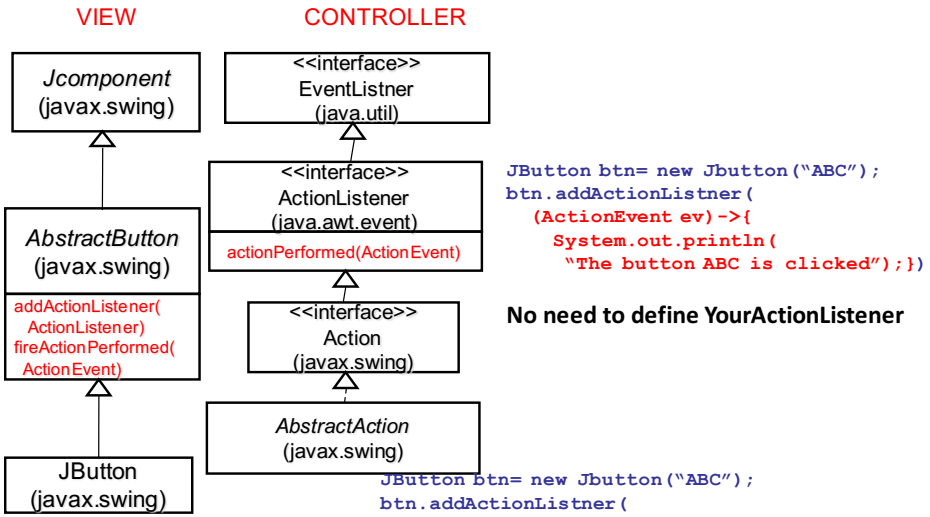


- MVC uses several design patterns internally.
  - Observer, Multicast (typed Observer), etc.
- Many (too many) examples/applications
  - Swing, Struts, Android, Flex, etc. etc., etc.

View and Controller in Swing

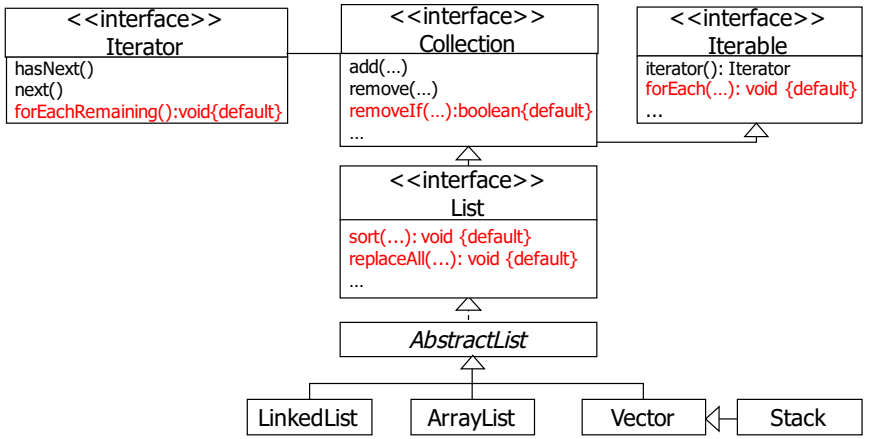


Lambda Expressions in Swing



New Default Methods for Lists

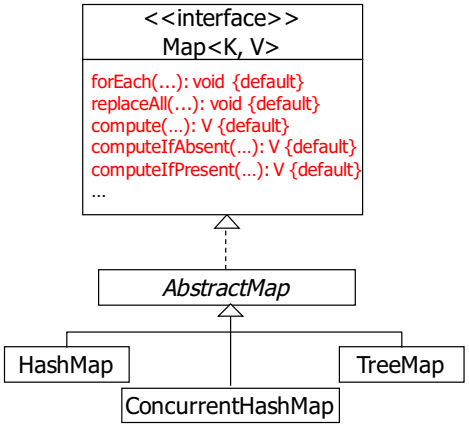
- Default methods have been added to various interfaces for lists.
  - Accept lambda expressions



```
ArrayList<String> list = new ArrayList(
                                Arrays.asList(
                                  "Yahoo","Yahooo","Yahoooo"));
// Print out each element
list.forEach( (String s)-> System.out.println(s) );
list.forEach(System.out::println);

// Sort elements based on the length of each element (descending order)
// Result: Yahoooo, Yahooo, Yahoo
list.sort( (String s1, String s2)-> s2.length()-s1.length() );
list.sort( Comparator.comparing( (String s)-> s.length(),
                                 Comparator.reverseOrder() );
list.sort( Comparator.comparing(String::length).reversed());

// Replace each element with the one returned by a given lambda expression
// Result: YAHOOOO, YAHOOO, YAHOO
list.replaceAll( (String s)-> s.toUpperCase() );
list.replaceAll(String::toUpperCase);

// Remove every element that matches a criterion defined in a given lambda expression
// Result: YAHOOO, YAHOO
list.removeIf( (String s)-> s.endsWith("OOOO") );
```


New Default Methods for Maps

- Default methods have been added to **`java.util.Map<K, V>`**
  - Accept lambda expressions

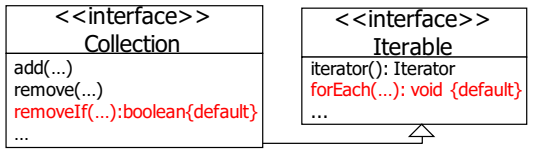


- **`forEach(LE)`**
  - Perform an action, which is defined as a given lambda expression, on each element
- **`replaceAll(LE)`**
  - Replace each element with the one returned by a given lambda expression
- **`compute(key, LE)`**
  - Pair a given key and a value that a given lambda expression returns and add the key-value pair.
- **`computeIfAbsent(key, LE)`**
  - Pair a given key and a value that a given lambda expression returns, ONLY IF the key does not exist, and add the key-value pair.
- **`computeIfPresent(key, LE)`**
  - Pair a given key and a value that a given lambda expression returns, ONLY IF the key does exist, and replace an existing key-value pair with the new pair.

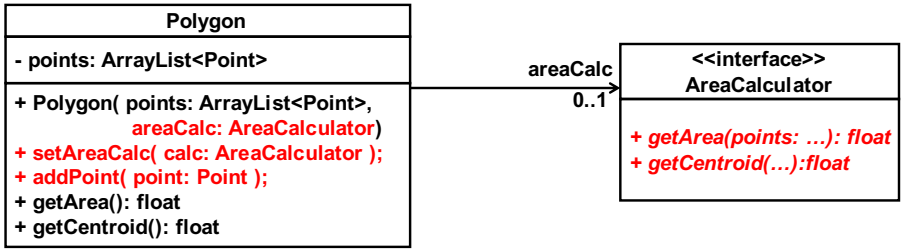

```
HashMap<String,Integer> map = new HashMap<>();
map.put("A",1); map.put("B",2); map.put("C",3);

// Print out each element
map.forEach((String key,Integer val)->
                System.out.println(key + "=" + val));

// Result: A=10, B=20, C=30
map.replaceAll((String key,Integer val)-> val*10);

// Result: A=1, B=20, C=30
map.compute("A", (String key,Integer val)->{
                        if(val==null){returns 0;}
                        else{returns val/10;} });
// Result: A=1, B=20, C=30, D=4
map.computeIfAbsent("D", (String key)-> 4);

// Result: A=100, B=20, C=30, D=4
map.computeIfPresent("A", (String key,Integer val)-> val*100);
```

Benefits of Default Methods

- Useful for API designers to add extra methods to existing interfaces
  - Without worrying about backward compatibility.
  - What if Oracle defined `forEach()` as an abstract method rather than a default method?

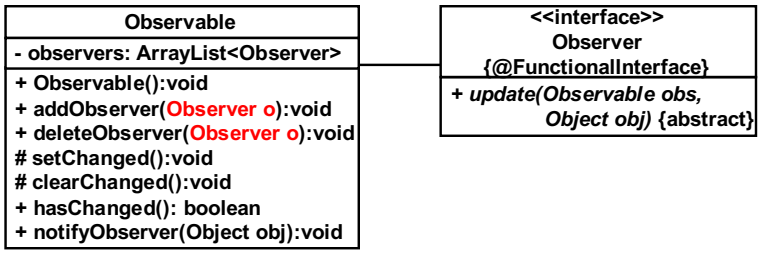


- Note: If an interface defines more than one abstract methods, it is no longer a functional interface. You cannot use lambda expressions for it.
  - Divide an interface so that each of divided interfaces has a single abstract method.



HW 4: Implement *Observer* with LEs

- Define your own Observable (class) and Observer (interface)
  - DO NOT reuse java.util.Observable and java.util.Observer
  - Define Observer as a functional interface.
    - update() as an abstract method.
  - Implement the listed methods for Observable
    - C.f. Java API doc for expected behaviors/responsibilities for the methods
- Use a lambda expression rather than defining a class that implements Observer



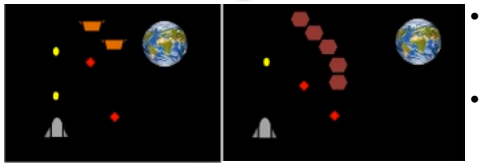
- Example client code

```
Observable observable = new Observable();
observable.setChanged();
observable.addObserver( (Observable o, Object obj)->
                          {System.out.println(obj);} );
observable.notifyObservers("Hello World!");
```

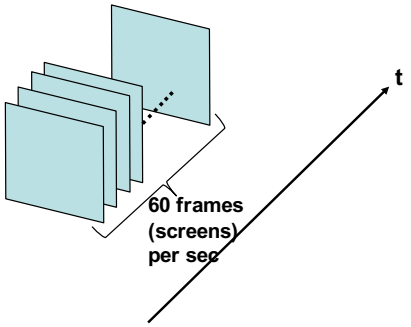
HW 5: Implement *Command* with LEs

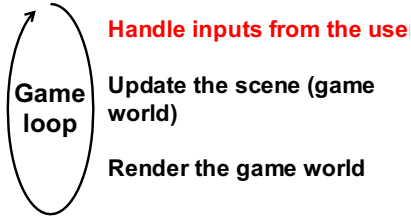
- Refresh your memory about Quiz #3, which is about how to use *Command* to design a 2D shooting game.
- Define the Command interface as a functional interface
- Explain how to implement InputHandler with lambda expressions
  - Pseudo code (required) for InputHandler and its client code
  - UML class diagram (if necessary)
  - Use lambda expressions rather than defining the classes that implement Command.

Q3 in CS680: Imagine a Simple 2D Game

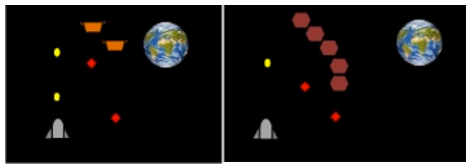


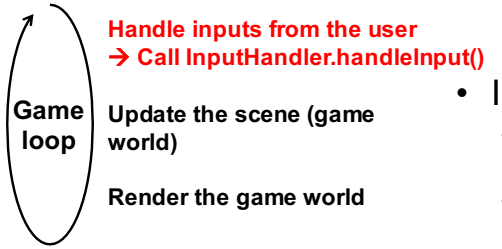
- The game loop is iterated repeatedly.
- Each iteration takes care of rendering one frame (or screen).
- Typical frame rate (FPS)
  - 60 iterations per second
  - 1.6 msec (1/60 sec) per frame





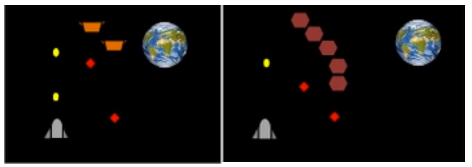
Handling User Inputs

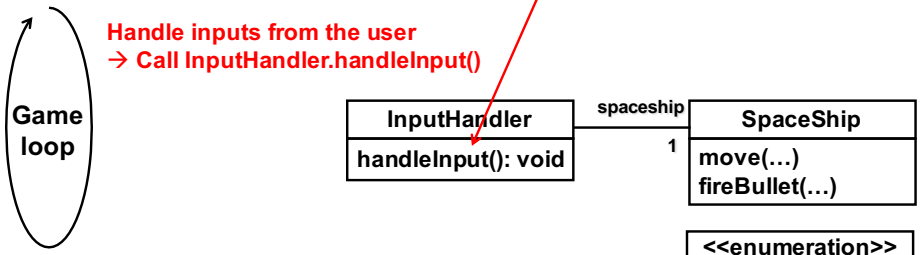




- 3 types of inputs
  - The user can push the right arrow, left arrow and space keys.
    - R arrow to move right
    - L arrow to move left
    - Space to fire a bullet
- InputHandler
  - Collects user inputs and respond to them.
  - handleInput()
    - identifies a keyboard input since the last game loop iteration (i.e. since the last frame).
    - One input per frame (i.e. during 1.6 msec)

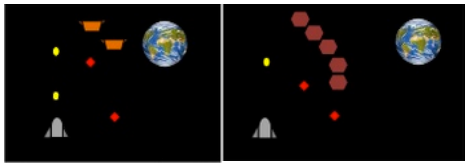
```
InputHandler ih = new InputHandler(...);
while(true){
  ih.handleInput();
  ...
}
```




```
if( isPressed(Key.RIGHT) )
  spaceship.move(…);
else if( isPressed(Key.LEFT) )
  spaceship.move(…);
else if( isPressed(Key.SPACE) )
  spaceship.fireBullet(…);
```

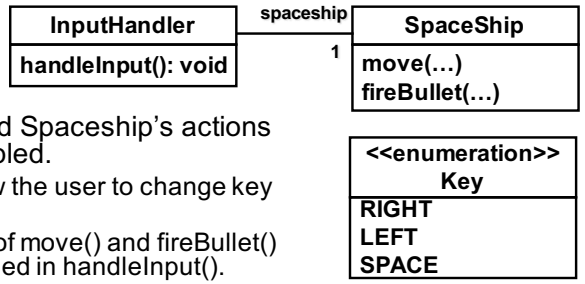
Not That Good… Why?



```
if( isPressed(Key.RIGHT) )
  spaceship.move(…);
else if( isPressed(Key.LEFT) )
  spaceship.move(…);
else if( isPressed(Key.SPACE) )
  spaceship.fireBullet(…);
```

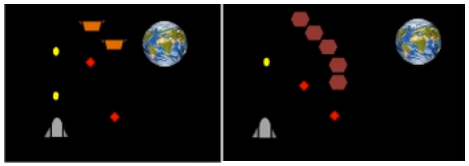
Game loop

Handle inputs from the user
→ Call InputHandler.handleInput()



- User inputs and Spaceship's actions are tightly coupled.
  - Hard to allow the user to change key bindings.
  - Invocations of move() and fireBullet() are hard-coded in handleInput().

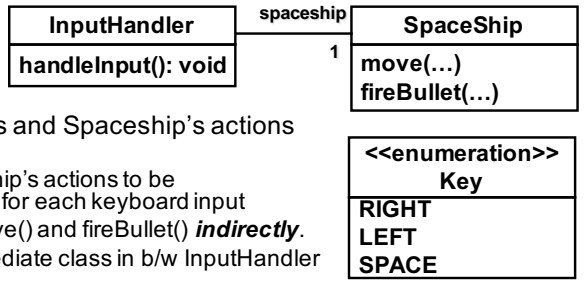
A Design Revision Needed



```
if( isPressed(Key.RIGHT) )
  spaceship.move(…);
else if( isPressed(Key.LEFT) )
  spaceship.move(…);
else if( isPressed(Key.SPACE) )
  spaceship.fireBullet(…);
```

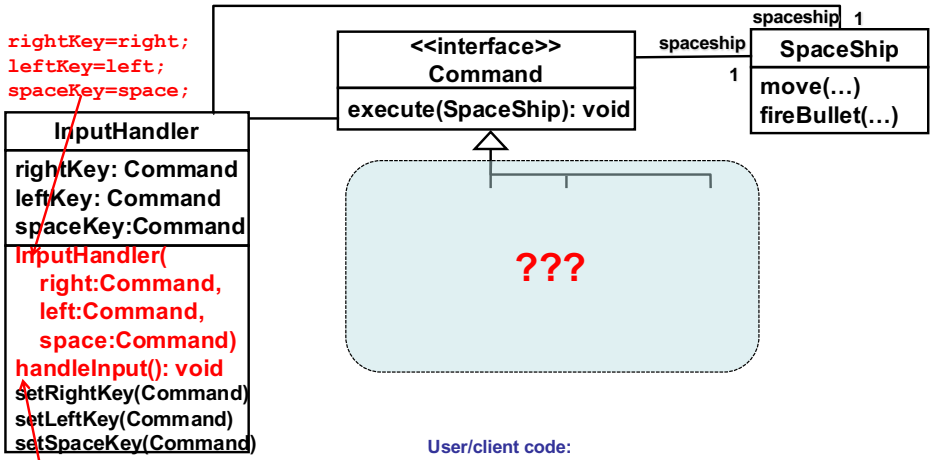
Game loop

Handle inputs from the user
→ Call InputHandler.handleInput()



- Making user inputs and Spaceship's actions loosely coupled.
  - Making Spaceship's actions to be interchangeable for each keyboard input
  - Need to call move() and fireBullet() ***indirectly***.
  - Need an intermediate class in b/w InputHandler and SpaceShip

SpaceShip's Actions as Command Classes




```
if(isPressed(Key.RIGHT))
  ???
else if(isPressed(Key.LEFT))
  ???
else if(isPressed(Key.SPACE))
  ???
```

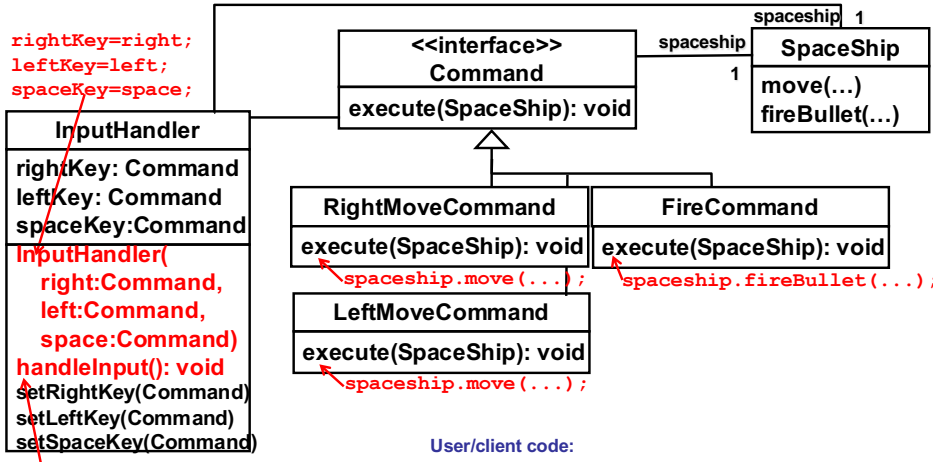
User/client code:

```
InputHandler ih =
  new InputHandler(new RightMoveCommand(...),
                   new LeftMoveCommand(...),
                   new FireCommand(...));
while(true){
  ih.handleInput();
  ... }
```


Quiz

- Complete InputHandler's handleInput() by replacing ??? with pseudo code.
- Define command classes that implements Command.
  - Complete the UML diagram by adding classes in the blue region.
  - Show how execute() should look like in each command class.

SpaceShip's Actions as Command Classes




```
if(isPressed(Key.RIGHT))
  rightKey.execute(spaceship);
else if(isPressed(Key.LEFT))
  leftKey.execute(spaceship);
else if(isPressed(Key.SPACE))
  spaceKey.execute(spaceship);
```

User/client code:

```
InputHandler ih =
  new InputHandler(new RightMoveCommand(...),
                   new LeftMoveCommand(...),
                   new FireCommand(...));
while(true){
  ih.handleInput();
  ... }
```