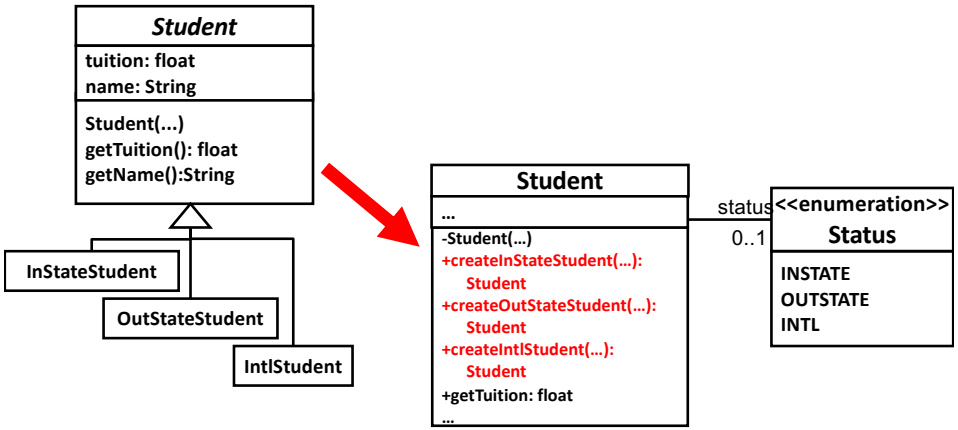
State Design Pattern

State

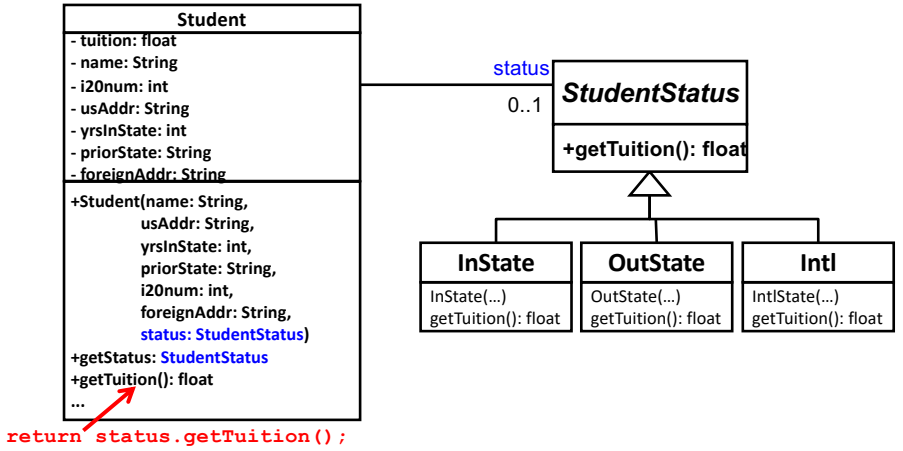
- Intent
  - Allow an object to change its behavior according to its state.

Eliminating Class Inheritance



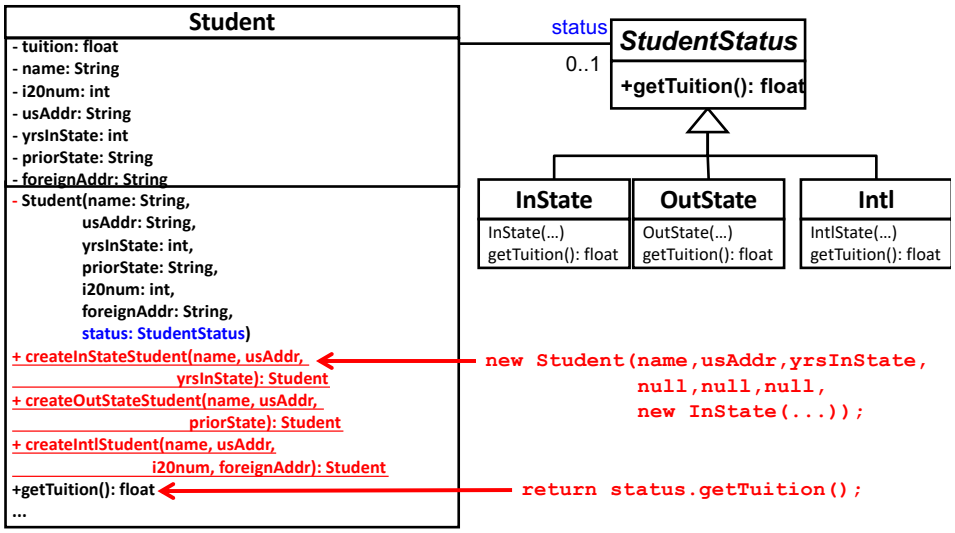
- c.f. previous lecture note
- Allows each student to change his/her status dynamically
- Needs a conditional in getTuition()
  - Can remove the conditional with *State*.

Design Improvement with *State*




```
Student s1 = new Student( ..., new InState(...) );
s1.getTuition();
```

Adding *Static Factory Methods*




```
Student s1 = Student.createInStateStudent( "John Smith", ...);
s1.getTuition();
```


Another Example: DVD Player

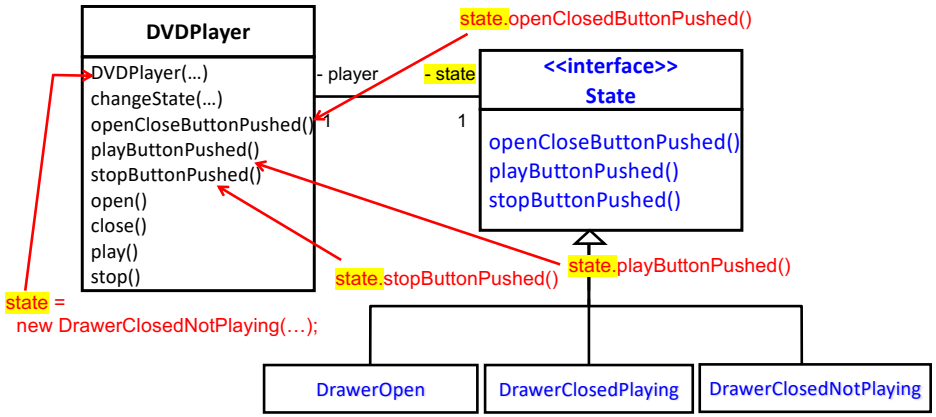
- Suppose you are implementing a firmware of DVD players
  - Focus on a player's *behaviors* upon *events*.
  - Events
    - The "Open/Close" button is pushed.
    - The "Play" button is pushed.
    - The "Stop" button is pushed.
  - The player differently behaves upon an event depending on its current state.
    - State-dependent behaviors

State-Dependent Behaviors

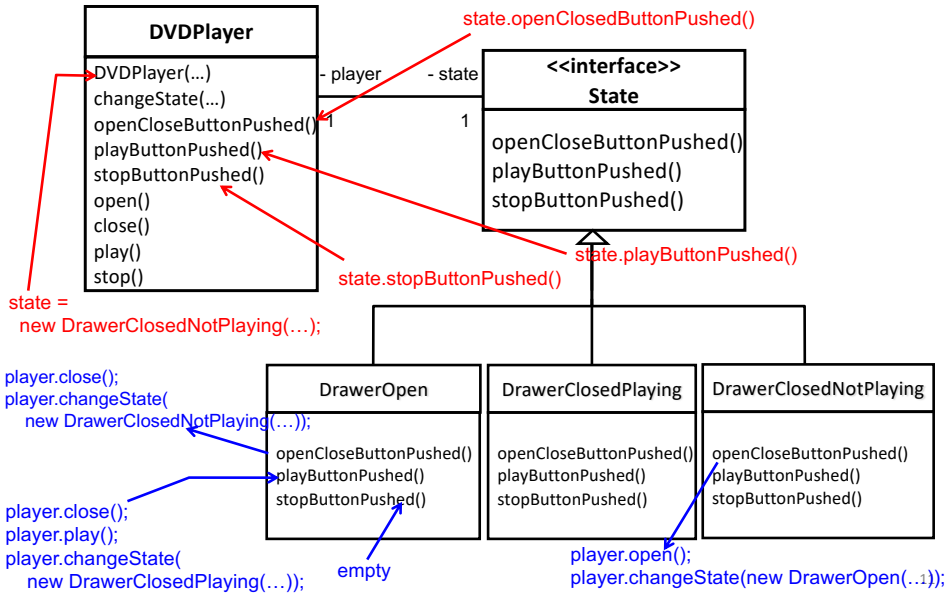
- When the "open/close" button pushed,
  - Opens the drawer
    - If the drawer is closed and the player is not playing a DVD.
  - Stops playing a DVD and opens the drawer
    - if the drawer is closed and the player is playing a DVD.
  - Closes the drawer
    - if the drawer is open.
- When the "play" button pushed,
  - Plays a DVD
    - if the drawer is closed.
  - Displays an error message
    - if the drawer is empty.
  - Closes the drawer and plays a DVD
    - if the drawer is open.
- When the "stop" button pushed
  - Stops playing a DVD
    - If the drawer is closed and the player is playing a DVD
  - Does nothing.
    - If the drawer is closed and the player is not playing a DVD.
  - Does nothing
    - If the drawer is open.

How to Implement State-dependent Behaviors?

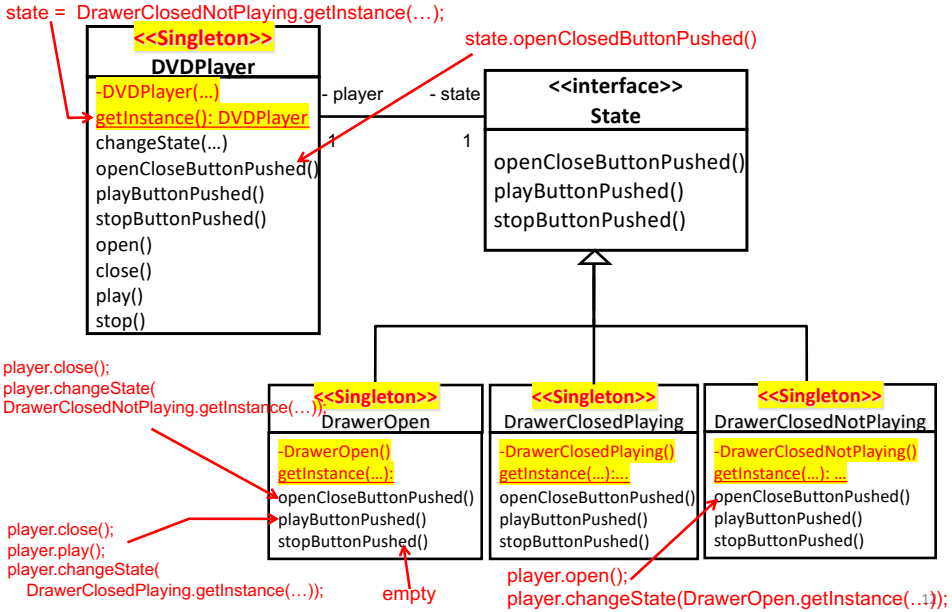
Defining States as Classes



Defining States as Classes



State Classes as *Singleton*



HW 14

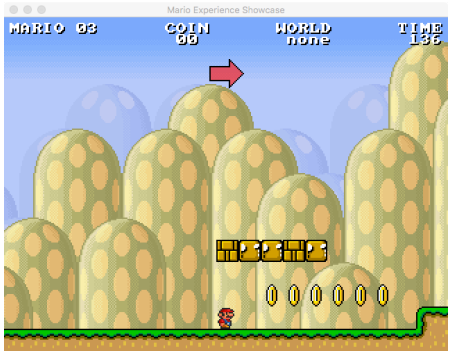
- Implement the DVD example with
  - *State*
  - *Singleton*

If-based and State-based Designs

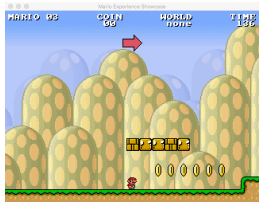
- If-based
  - Easy/straightforward to implement at first
  - Hard to maintain a long sequence of conditional branches
- *State*-based
  - May not be that easy/straightforward to implement at first
  - Easy to maintain
    - If new buttons/events are added, just add extra methods in state classes.
      - No need to modify existing methods.
  - Initial cost may be higher, but maintenance cost (or total cost) should be lower
    - as changes are made in the future.

One More Example: Game Characters

- Game characters often have state-dependent behaviors.
- Think of a simple 2D game like Super Mario



Handling User Inputs



Game loop

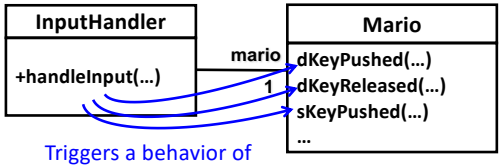
Detect an input from the user
→ Call InputHandler.handleInput()

Update the scene (game world)

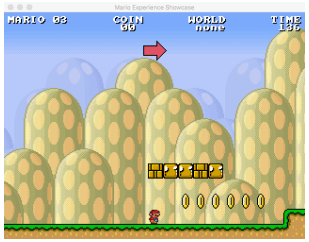
Render the game world

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

- 5 types of inputs
  - The user can push the right arrow, left arrow, down arrow and "s" keys.
    - R arrow to move right
    - L arrow to move left
    - D arrow to duck
    - "s" to jump
  - The user releases the D arrow to stand up.
- **`InputHandler`**
  - **`handleInput()`**
    - identifies a keyboard input since the last game loop iteration (i.e. since the last frame).
    - 60 frames/s (FPS): One input per frame (i.e. during 1.6 msec)



Triggers a behavior of Mario



For simplicity, let's focus on 3 inputs only here:
D arrow pushed, D arrow released, and "s" key pushed

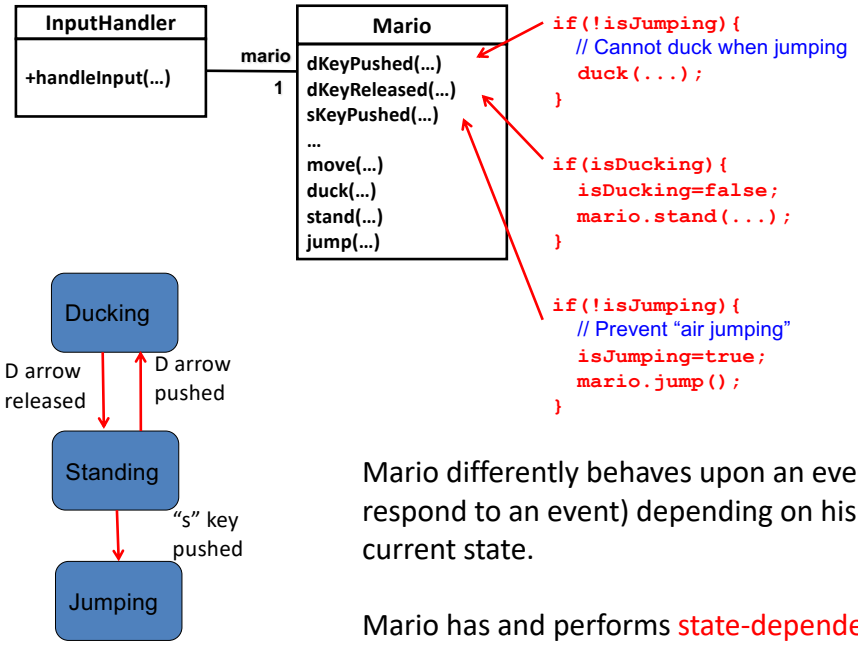
Game loop

Detect an input from the user
→ Call InputHandler.handleInput()

Update the scene (game world)

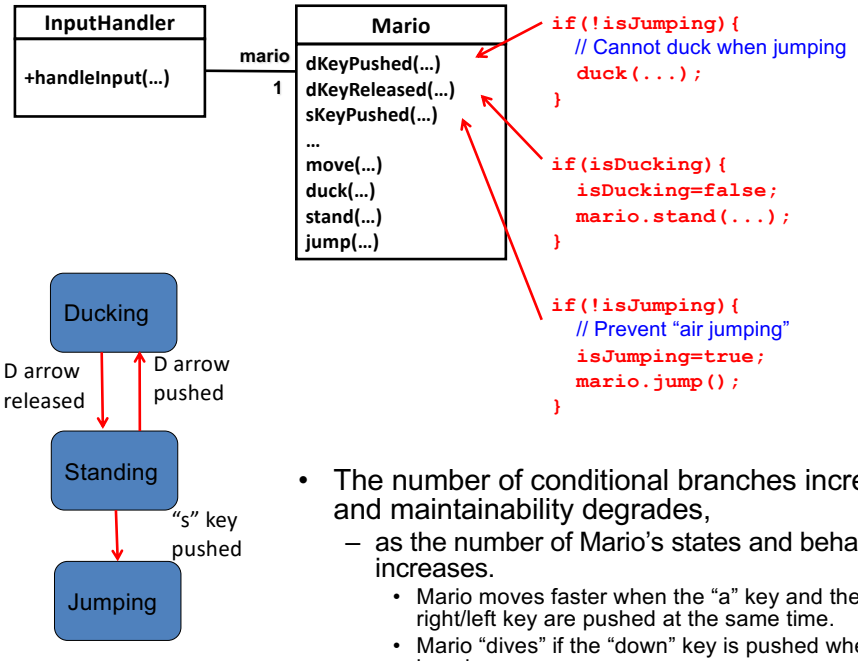
Render the game world

```
InputHandler ih = new InputHandler(...);
while(true){
  ih.handleInput(...);
    // If D arrow is pushed,
    //     call dKeyPushed() on Mario
    // If D arrow is released,
    //     ...
  ...
}
```

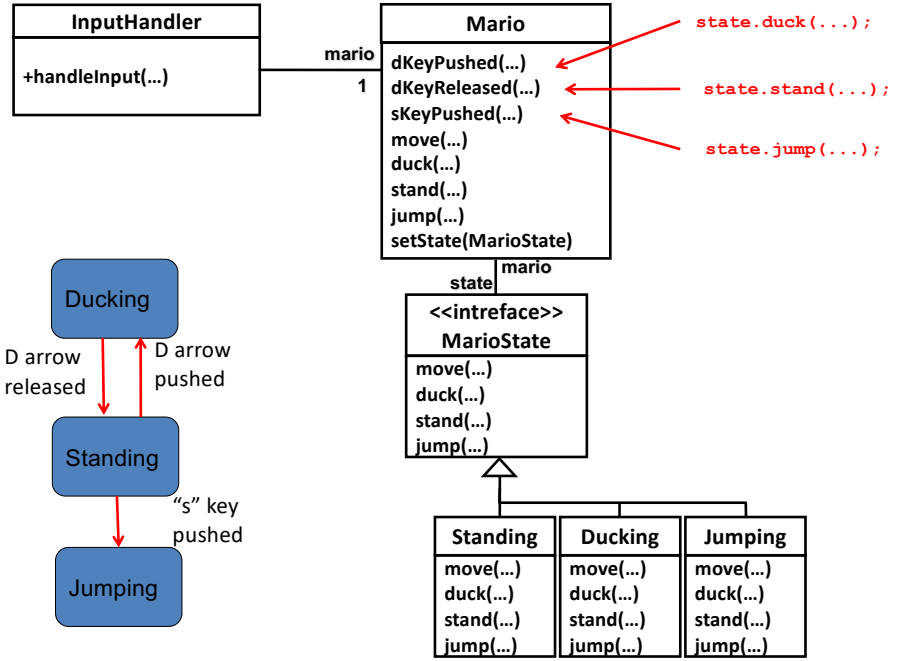


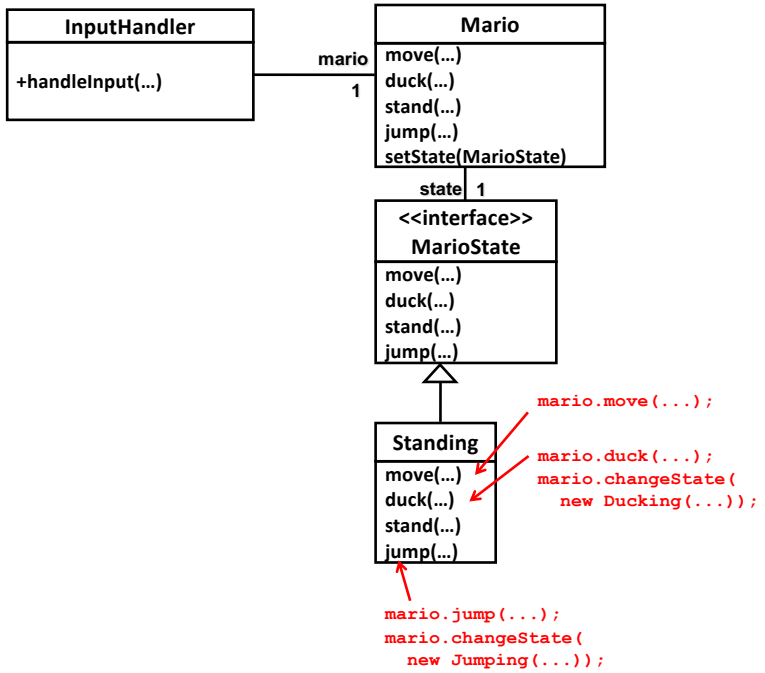
Mario differently behaves upon an event (or respond to an event) depending on his current state.

Mario has and performs state-dependent behaviors



- The number of conditional branches increases and maintainability degrades,
  - as the number of Mario's states and behaviors increases.
    - Mario moves faster when the "a" key and the right/left key are pushed at the same time.
    - Mario "dives" if the "down" key is pushed when jumping.





An Interesting Bug

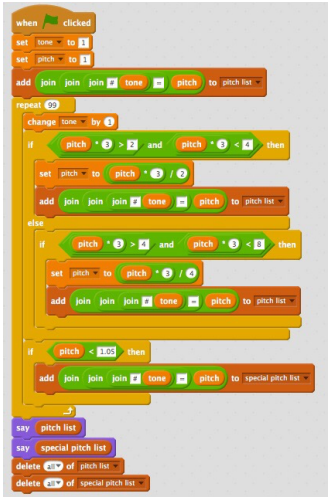
- The source of a few bugs should be related to state-dependent behaviors.
  - At least one of them has not been fixed for a year.
- Many states exist.
  - Regions, primary phone language, extra phone languages, primary device language, extra device languages, etc.

- Many state-dependent behaviors exist.
  - In the US, the device can play radios, uploaded music and custom playlists on G Play Music.
    - Regardless of the language settings
  - In another certain country, it can play radios only.
  - In the US with English as the primary phone language, it allows the user to define custom voice commands (called routines).
  - In the US with another language as the primary phone language, it does not allow the user to define custom voice commands.

Preliminaries: Road to Object-Oriented Design (OOD)

Brief History

- In good, old days… programs had no structures.
  - One dimensional code.
    - From the first line to the last line on a line-by-line basis.
    - "Go to" statements to control program flows.
      - Produced a lot of "spaghetti" code
        - » "Go to" statements considered harmful.



- In good, old days… programs had no structures.
  - No notion of structures (or modularity)
    - Modularity: Making a chunk of code (module) self-contained and independent from the other code
      - Improve reusability and maintainability
        - » Higher reusability → higher productivity, less production costs
        - » Higher maintainability → higher productivity and quality, less maintenance costs

Modules in SD and OOD

- Modules in Structured Design (SD)
  - Structure = a set of variables (data fields)
  - Function = a block of code
- Modules in OOD
  - Class = a set of data fields and functions (methods)
  - Interface = a set of functions (methods)
- Key design questions/challenges:
  - how to define modules?
  - how to separate a module from others?
  - how to let modules interact with each other?

SD v.s. OOD

- OOD
  - Intends coarse-grained modularity
    - The size of each module is often bigger.
  - Extensibility in mind in addition to reusability and maintainability
    - How easy (cost effective) to add and revise existing modules (classes and interfaces) to implement new/modified requirements.
    - How to make software more flexible/robust against changes in the future.
  - How to gain reusability, maintainability and extensibility?

How to Gain Reusability, Maintainability and Extensibility?

- Design patterns can answer this question to some extent.
  - You can learn how you can/should organize your code to gain these properties.
- Recall, for example,
  - *Strategy*
    - How to make algorithms interchangeable and extensible?
    - How to make algorithm users maintainable?
  - *State*
    - How to make state-dependent behaviors (operations) maintainable?
  - *Visitor*
    - How to make visitors (i.e. operations to be applied on a set of data structures) extensible?
    - How to make the set of data structures maintainable?

- You can learn about code organization for reusability, maintainability and extensibility only through writing and running your own code.
  - Through DOING, not listening to someone, reading something or drawing mental pictures.

Recap: Looking Ahead - AOP, Functional Programming, etc.

- OOD does a pretty good job in terms of modularity, but it is not perfect.
- OOD still has some modularity issues
  - Aspect Oriented Programming (AOP)
    - Dependency injection
    - Handles cross-cutting concerns well.
      - e.g. logging, security, DB access, transactional access to a DB
- Highly modular code sometimes look ***redundant***.
  - Functional programming
    - Makes code less redundant.
  - Lambda expressions in Java
    - Intended to make modular (OOD-backed) code less redundant.

Functional Programming with Java

Notable Enhancements in Java 8

- Lambda expressions
  - Allow you to do *functional programming* in Java
- Static and default methods in interfaces

Lambda Expressions in Java

- Lambda expression
  - A block of code (or a function) that you can pass to a method.
- Before Java 8, methods could receive primitive type values and objects only.
  - `public void example(int i, String s, ArrayList<String> list)`
  - Methods could receive nothing else.
    - You couldn't do like this:
      - `foo.example( [ if(Math.random()>0.5){ // Do something } else{ // Do something else } ] )`

How to Define a Lambda Expression?

- A lambda expression consists of
  - A code block
  - A set of parameters to be passed to the code block

```
– (String str) -> str.toUpperCase()

– (StringBuffer first, StringBuffer second)
                -> first.append(second)

– (int first, int second) -> second – first
```


- No need to give a name to a function.
  - Lambda expression ~ *anonymous* function/method that is not bound to a class/interface

```
– (int first, int second)-> second – first

– public int subtract(int first, int second){
      return second – first; }
```


- No need to explicitly specify the type of a returned value.
  - Your Java compiler automatically infers that.
- Single-expression code block does not even require the `return` keyword.

```
– (int first, int second)-> second – first

– public int subtract(int first, int second){
      return second – first; }
```


- Multi-expression code block
  - Surrounds expressions with { and }. Use ; in the end of each expression.

```
– (double threshold)-> {
      if(Math.random() > threshold) return true;
      else return false; }

– () -> {
      if(Math.random) > 0.5) return true;
      else return false; }
```

- Multi-expression code block
  - Requires a `return` statement in each control flow.
    - Every conditional branch must return a value.

```
() -> {
    if(Math.random) > 0.5) return true;
    else return false; }
```

```
() -> {
    if(Math.random) > 0.5) return true;
//    else return false; ← A compilation error occurs
                          here if this line is
                          commented out.
}
```


How to Pass a Lambda Expression to a Method?

- A method can receive a lambda expression *as a method parameter*.

```
foo.example( (int first, int second) -> second-first )
```

  - What is the type of that parameter?
    - ***Functional interface*!**

Functional Interface

- A special type of interfaces
  - An interface that has a single abstract (or empty) method.

Functional Interface

- An interface that has a single abstract (or empty) method.
- Example: `java.util.Comparator`
  - Has `compare()`, which is the only abstract method.
    - A new annotation introduced in Java 8:
      -
```
@FunctionalInterface
public interface Comparator<T>
```
      - All functional interfaces in Java API have this annotation.
        - » The API documentation says "This is a functional interface and can therefore be used as the assignment target for a lambda expression…"

An Example Use Case

- Example functional interface: `java.util.Comparator`
  - Has `compare()`, which is the only abstract method.

- `Collections.sort(List, Comparator<T>)`
  - The second parameter can accept a lambda expression.

```
Collections.sort( aList,
                  (Integer first, Integer second)->
                      second.intValue()-first.intValue() );
```


Recap: `Collections.sort()`

- Sorting collection elements:

```
ArrayList<Integer> years2 = new ArrayList<Integer>();
years2.add( Integer.valueOf(2010) );
years2.add( Integer.valueOf(2000) );
years2.add( Integer.valueOf(1997) );
years2.add( Integer.valueOf(2006) );
Collections.sort(years2);
for(Integer y: years2)
    System.out.println(y);
```

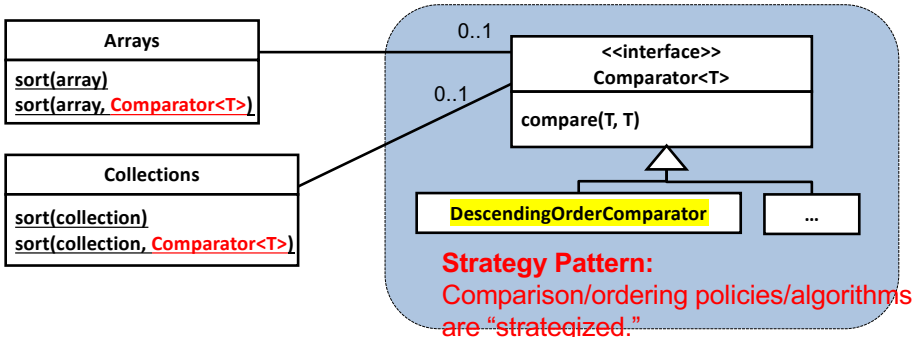
  - `java.util.Collections`: a utility class (i.e., a set of static methods) to process collections and collection elements

  - `sort()` orders collection elements in an ascending order.
    - 1997 -> 2000 -> 2006 -> 2010

Comparison/Ordering Policies

- What if you want a custom (non-default) comparator?
  - `Collections.sort()` implement ascending ordering only.
    - They do not implement any other policies.

- Define a custom comparator by implementing `java.util.Comparator`



- Example custom comparator

```
public class DescendingOrderComparator<Integer>{
  implements Comparator<Integer>{
    public int compare(Integer o1, Integer o2){
      return o2.intValue()-o1.intValue();
    }
}
```

Sorting Collection Elements with a Custom Comparator

```
ArrayList<Integer> years = new ArrayList<Integer>();
years.add(new Integer(2010)); years.add(new Integer(2000));
years.add(new Integer(1997)); years.add(new Integer(2006));

Collections.sort(years);
for(Integer y: years)
    System.out.println(y);

Collections.sort(years, new DescendingOrderComparator());
for(Integer y: years)
    System.out.println(y);
```

– 1997 -> 2000 -> 2006 -> 2010

– 2010 -> 2006 -> 2000 -> 1997

Okay, so What's the Point?

- Now, you have 2 different ways to implement custom comparators:
  - Without a lambda expression (LE)

```
public class DescendingOrderComparator<Integer>{
  implements Comparator<Integer>{
    public int compare(Integer o1, Integer o2){
      return o2.intValue()-o1.intValue(); } }

Collections.sort(years, new DescendingOrderComparator());
```

  - With a lambda expression (LE)

```
Collections.sort(years,(Integer o1, Integer o2)->
                       o2.intValue()-o1.intValue());
```


- Without a LE

```
public class DescendingOrderComparator<Integer>{
  implements Comparator<Integer>{
    public int compare(Integer o1, Integer o2){
      return o2.intValue()-o1.intValue(); } }

Collections.sort(years, new DescendingOrderComparator());
```

- With a LE

```
Collections.sort(years,(Integer o1, Integer o2)->
                       o2.intValue()-o1.intValue());
```

- Code gets more concise (less redundant/repetitive).
  - The LE defines `DescendingOrderComparator`'s `compare()` in a concise way.
- The LE version is a *syntactic sugar* for the non-LE version.
  - Your compiler does program transformation at compilation time.

- Without a LE

```
public class DescendingOrderComparator<Integer>{
  implements Comparator<Integer>{
    public int compare(Integer o1, Integer o2){
      return o2.intValue()-o1.intValue(); } }

Collections.sort(years, new DescendingOrderComparator());
```

- With a LE

```
Collections.sort(years,(Integer o1, Integer o2)->
                       o2.intValue()-o1.intValue());

Collections.sort(years,(o1, o2)->
                       o2.intValue()-o1.intValue());
```

- You can omit parameter types in a LE.
  - Parameter values follow the method signature of `DescendingOrderComparator`'s `compare()`.
  - Code gets a bit more concise (less redundant/repetitive).

FYI: Anonymous Class

- The most expressive (default) version

```
– public class DescendingOrderComparator<Integer>{
    implements Comparator<Integer>{
      public int compare(Integer o1, Integer o2){
        return o2.intValue()-o1.intValue();
      }
  }
  Collections.sort(years, new DescendingOrderComparator());
```

- With an anonymous class

```
– Collections.sort(years,
                   new Comparator<Integer>(){
                     @Override
                     public int compare(Integer o1, Integer o2){
                       return o2.intValue()-o1.intValue();
                     }
                   } );
```

- With a LE (more concise and less ugly)

```
– Collections.sort(years,(Integer o1, Integer o2)->
                            o2.intValue()-o1.intValue());
```


How Do You Know Where You can Use a Lambda Expression?

- `Collections.sort(List, Comparator<T>)`
- Check out `Comparator` in the API doc.
- Notice that `Comparator` is a functional interface.
  - `@FunctionalInterface`
    `public interface Comparator<T>`
    - The API doc says "This is a functional interface and can therefore be used as the assignment target for a lambda expression…"
  - This means you can pass a lambda expression to `sort()`.

- Find out the abstract (or empty) method in `Comparator`.
  - `public int compare(T o1, T o2)`
- Define a lambda expression that represents the method body of `compare()` and pass it to `sort()`.

```
– ArrayList<Integer> aList = new ArrayList<Integer>();
  Collections.sort(aList,
                   (Integer first, Integer second)->
                      second.intValue()-first.intValue());
```


Assignment of a LE to a Functional Interface

- A lambda expression can be assigned to a variable that is typed with a functional interface.

```
– Comparator<Integer> comparator =
       (Integer o1, Integer o2)-> o2.intValue()-o1.intValue();
  Collections.sort(years, comparator);
```

- Parameter types can be omitted through type inference.

```
– Comparator<Integer> comparator =
       (o1, o2)-> o2.intValue()-o1.intValue()

– ArrayList<Integer> aList = new ArrayList<Integer>();
  Collections.sort(aList,
                   (first, second)->
                      second.intValue()-first.intValue());
```

What does `Collections.sort()` do?

```
class Collections
    static ... sort(List<T> list, Comparator<T> c){
        for each pair (o1 and o2) of elements in list{
            int result = c.compare(o1, o2);
            if(result < 0){
                ...
            }else if(result > 0){
                ...
            }else if(result==0){
                ...
        } } }
```

- c.f. Run this two-line code.
  - ```
    Comparator<Integer> comparator =
                          (o1, o2)-> o2.intValue()-o1.intValue();
    comparator.compare(1, 10);
    // compare() returns 9 (10-9)
    ```


Some Notes

- A lambda expression can be assigned to a functional interface.
  - ```
    public interface Comparator<T>{
        public int compare(T o1, T o2)
    }
    Comparator<Integer> comparator =
        (Integer o1, Integer o2)-> o2.intValue()-o1.intValue()
    ```
  - `Collections.sort(years, comparator);`
- It CANNNOT be assigned to `Object`.
  - ```
    Object comparator =
        (Integer o1, Integer o2)-> o2.intValue()-o1.intValue()
    ```


- Without a lambda expression
  - ```
    public class DescendingOrderComparator<Integer>{
      implements Comparator<Integer>{
        public int compare(Integer o1, Integer o2){
          return o2.intValue()-o1.intValue();
        }
    }
    Collections.sort(years, new DescendingOrderComparator());
    ```
- With a lambda expression
  - ```
    Collections.sort(years,(Integer o1, Integer o2)->
                           o2.intValue()-o1.intValue());
    ```
- A type mismatch results in a compilation error.
  - ```
    Collections.sort(years,(Integer o1, Integer o2)->
                          o2.floatValue()-o1.floatValue());
    ```
  - The return value type must be int, not float.

- A lambda expression cannot throw an exception
  - if its corresponding functional interface does not specify that for the abstract/empty method.
- Not good (Compilation fails.)
  - ```
    public interface Comparator<T>{
        public int compare(T o1, T o2)
    }
    ```
  - ```
    Collections.sort(years,(Integer o1, Integer o2)->{
                           if(...) throw new XYZException;
                           else return ... );
    ```
- Good
  - ```
    public interface Comparator<T>{
        public int compare(T o1, T o2) throws ZYZException
    }
    ```
  - ```
    Collections.sort(years,(Integer o1, Integer o2)->{
                           if(...) throw new XYZException;
                           else return ... );
    ```

LEs make Your Code Concise, but…

- You still need to clearly understand
  - the Strategy design pattern
    - `Comparator` and its implementation classes
    - What `compare()` is expected to do
- Using or not using LEs just impact how to *express* your code.
  - This does not impact how to *design* your code.

A Benefit of Using Lambda Expressions

- Your code gets more concise (less redundant/repetitive).
  - This may or may not mean "easier to understand" depending on how much you are familiar with lambda expressions.

Interfaces in Java 8

- *Functional interface*: a special type of interface that has a single ***abstract*** (or empty) 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

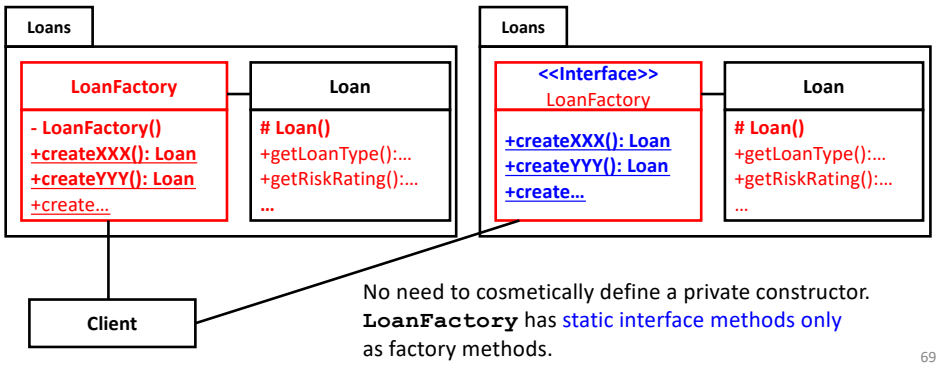
- Java 8 introduces the keyword `abstract`.
  - ```
    public interface Foo{
        public abstract void Boo()
    }
    ```
  - `abstract` can be omitted.
    - ```
      public interface Comparator<T>{
          public int compare(T o1, T o2)
      }
      ```
    - ```
      public interface Comparator<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();      // I2 does not inherit getValue(). Compilation error.
  ```

- ```
  public interface I2 extends I1{}
      public static int getValue(){ return 987; } }
  I2.getValue();      // I2 can override getValue(). 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.

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();    // I2 inherits getValue(). Returns 123.
  ```

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

- ```
  public class C1 implements I1{
      public int getValue(){ return 987; } }
  ```
- ```
  C1 c = new C1();
  c.getValue();    // C1 can override getValue(). 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.
  ```

- **Precedence rule:** The super class's method precedes an interface's default method.

- You can call an interface's default method, if you want.
  - ```
    public class C2 extends C1 implements I1{
        public int getValue(){
            return 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; } }

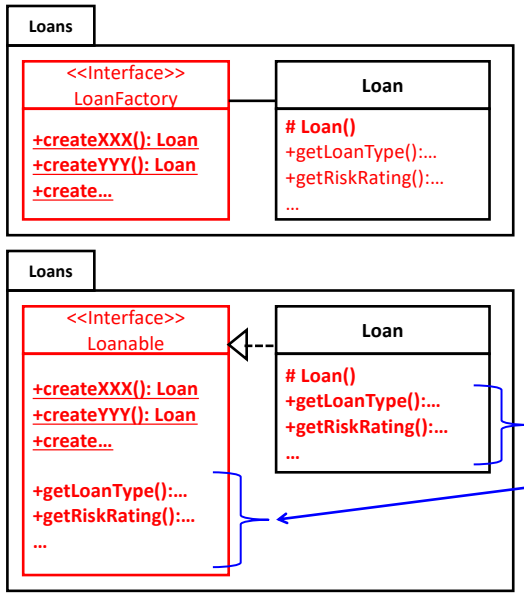
  public class C1 implements I1, I2{}      // Compilation error.
  ```
  - Default methods from different interfaces conflict.

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

Example Static Interface Methods

- Static factory methods to create an object that implements an interface.
- They can be implemented as static interface methods.



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



`LoanFactory` has static interface methods only as factory methods.

`Loanable` can define loan-specific methods as abstract methods, if you want.

Then, **`Loan`** implements them.

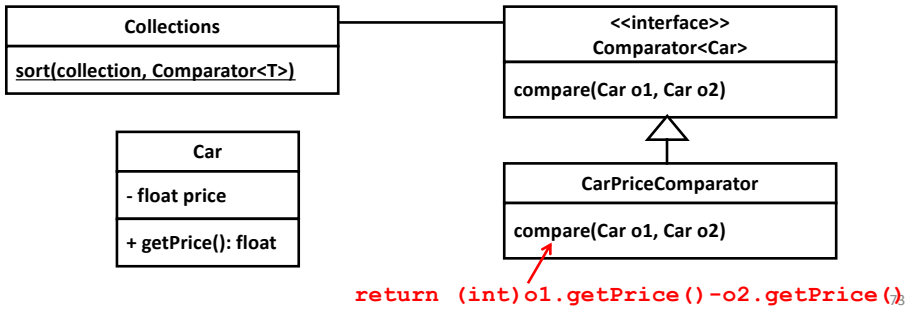
Static Methods in `Comparator`

- **`java.util.Comparator<T>`** has…
  - one abstract method (**`compare()`**) and
  - many *static* and *default* methods.
    - `static Comparator<T> comparing(Function<T, R> keyExtractor)`

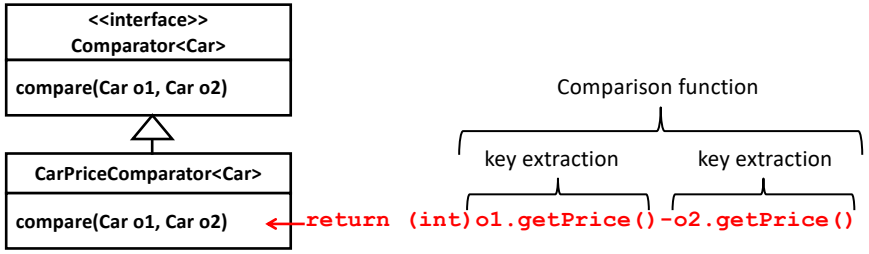
- **`java.util.Comparator<T>`** has…
  - `static Comparator<T> comparing(Function<T, R> keyExtractor)`
    - Accepts a LE that extracts a **`Comparable`** sort key from **`T`**
      - Sort key (**`R`**): data/value to be used in ordering
      - **`Function<T, R>`**
        - » Represents a function (lambda expression) that accepts a parameter (**`T`**) and returns a result (R).
    - Returns a **`Comparator<T>`**

```
class Car{ private float getPrice(); }
ArrayList<Car> carList = new ...
...
Collections.sort(carList, Comparator.comparing(
                                (Car car)-> car.getPrice());
                          //comparing() returns a Comparator<Car>
```

- `class Car{ private float getPrice(); }`
- ```
  Collections.sort(carList,
                   Comparator.comparing(
                               (Car car)-> car.getPrice() );
  ```
- ```
  Collections.sort(carList,
                   (Car o1, Car o2)->
                               (int)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/returns another function as a result
    - `class Car{ public int getPrice();}`
    - ```
      Collections.sort(carList,
                       Comparator.comparing(
                           (Car car)-> car.getPrice() );
      ```
    - ```
      Collections.sort(carList,
                       (Car o1, Car o2)->
                           (int)o1.getPrice()-o2.getPrice());
      ```



- `Comparator.comparing()` uses/follows ascending ordering (natural ordering) by default.
  - `class Car{ public float getPrice();}`
  - ```
    Collections.sort(carList,
                     Comparator.comparing(
                                 (Car car)-> car.getPrice() );
    ```
  - ```
    Collections.sort(carList,
                     (Car o1, Car o2)->
                                 (int)o1.getPrice()-o2.getPrice());
    ```
- What if you want *descending ordering*?
  - ```
    Collections.sort(carList,
                     Comparator.comparing((Car car)-> car.getPrice(),
                                          Comparator.reverseOrder());
    ```
  - ```
    Collections.sort(carList,
                     Comparator.comparing((Car car)-> car.getPrice(),
                                          Comparator.naturalOrder());
    ```


Benefits of Using Lambda Expressions

- Can make your code more concise (less repetitive)
- Can enjoy the power of functional programming
  - e.g., higher-order functions

A Bit More about Comparator

```
class Car{ public float 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)`

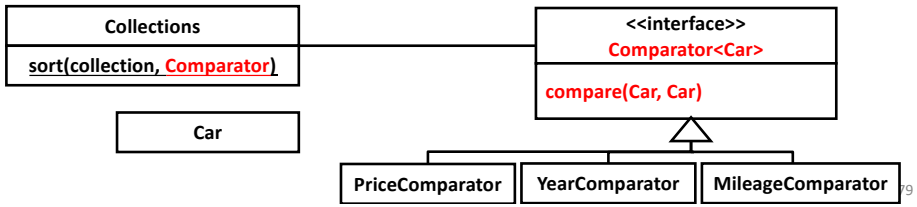

```
class Car{ public float getPrice();}
Collections.sort(carList,
                 Comparator.comparing((Car car)-> car.getPrice() );
Collections.sort(carList,
                 Comparator.comparing( Car::getPrice ) );
```

  - Ascending order (natural order) by default

- What if you want *descending ordering*?
  -
```
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() );
```


HW 15

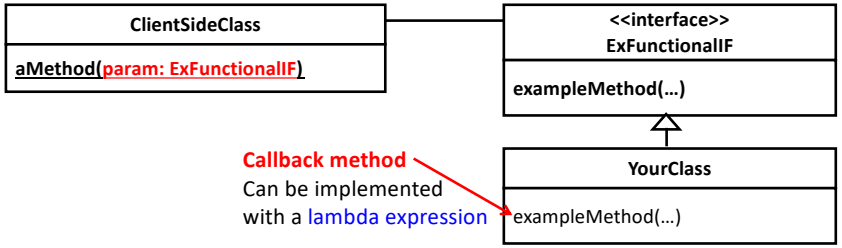
- Revise your HW 10 solution with lambda expressions.
  - Instead of defining 4 classes that implement `Comparator<Car>`, define the body of each `compare()` method as a lambda expression and pass it to `Collections.sort()`.



- Pass 4 different lambda expressions to `Collections.sort()`
  -
```
ArrayList<Car> carList = ...
Collections.sort(carList,
                 (Car car1, Car car2)->{ ... } );
...
```
  - Use `Comparator.comparing()`, if you like. You will get some extra point.
- Create several `Car` instances and sort them with each lambda expression.
  - Minimum requirement: ascending ordering (natural ordering)
  - [Optional] Do descending ordering as well with `reverseOrder()` or `reserved()` of `Comparator`.

Where/When to Use Lambda Expressions

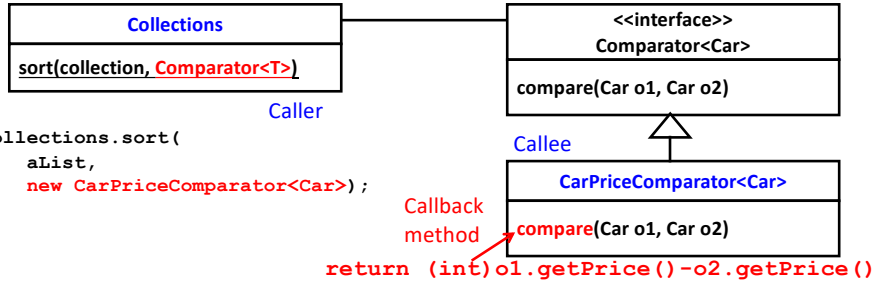
- There is a functional interface:
  - An interface that has an abstract method
  - You are expected to define a class that implements the interface with the body of the abstract method.
    - The method is generally called callback method.
- There is a method that accepts a parameter that is typed with that functional interface.
  - It will call the callback method.



- Many design patterns follow this structure.
  - You can use lambda expressions to implement those design patterns.
  - HW 14
    - Use lambda expressions to implement *Strategy*
  - What else?
    - *Observer*, *Command*, etc. etc.

Callback Method in Comparator

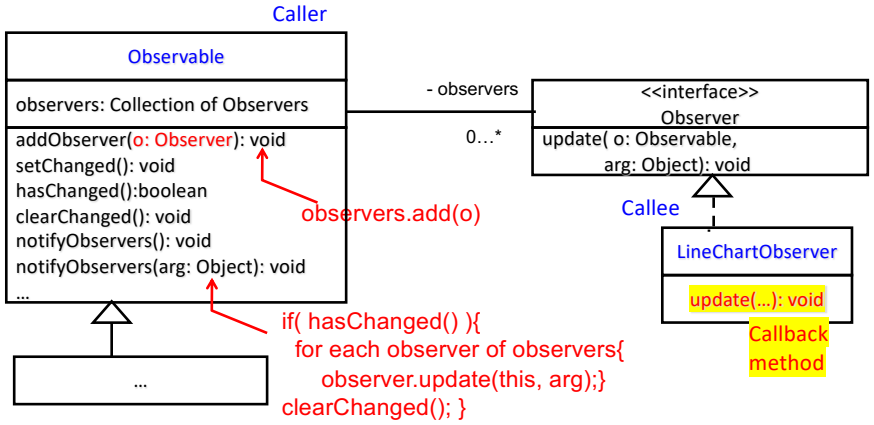
- Two key players
  - A "callee" class that implements a callback method
    - You implement it, but you do not call the method directly.
  - A "caller" class that will call the callback method in the future
    - Someone else has implemented it.
- Interaction between a callee and a caller
  - You make a callee class instance and set it to a caller class instance, so the caller can call the callback method in the future.



What does `Collections.sort()` do?

```
class Collections
    static ... sort(List<T> list, Comparator<T> c){
        for each pair (o1 and o2) of elements in list{
            int result = c.compare(o1, o2);
            if(result < 0){
                ...
            }else if(result > 0){
                ...
            }else if(result==0){
                ...
    } } }
```

Another Example: *Observer*

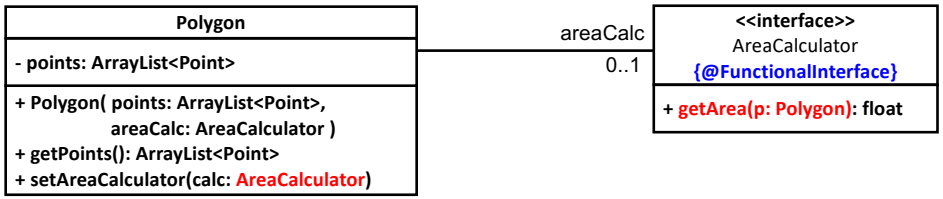



```
anObservable.addObserver( new LineChartObserver() );
anObservable.notifyObservers( ... );
```

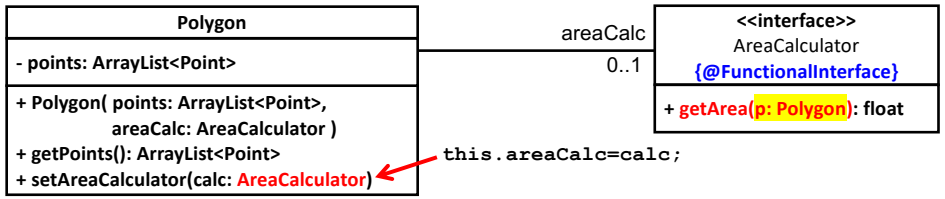

Free Variables

- Variables that….
  - a lambda expressions can access, but
  - are not defined in that lambda expression.
    - Not parameters of a lambda expression (LE).
    - Not local variables defined in a LE's code block.
  - Local variables of an *enclosing* method
    - Incl. Method parameters of the *enclosing* method
    - e.g., `Collections.sort(List<T> list, Comparator<T> c)`
  - Data fields of an *enclosing* class
- A lambda expression can access those free variables.

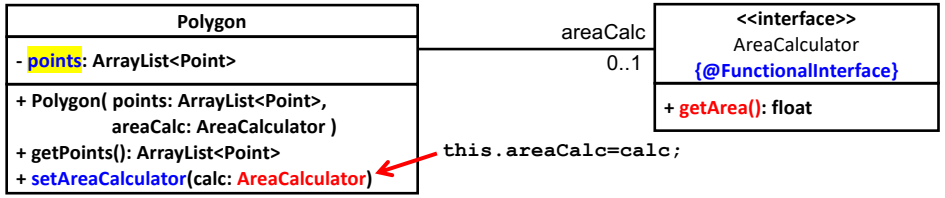
An Example: *Strategy*




```
Polygon polygon = new Polygon(...);
polygon.setAreaCalculator( (Polygon p)->{ p.getPoints()...;
                                          ...; } );
```

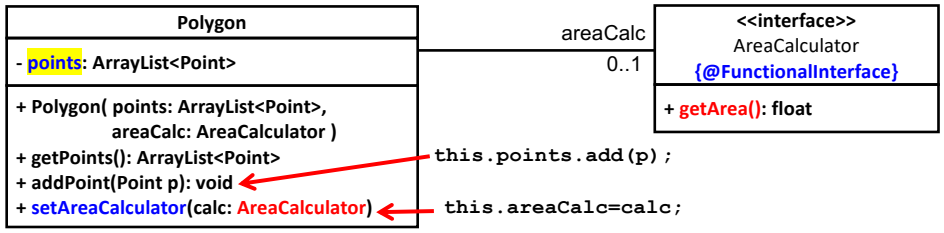

```
Polygon polygon = ...
polygon.setAreaCalculator( (Polygon p)->{ p.getPoints()...;
                                          ...; } );
```




```
Polygon polygon = ...
polygon.setAreaCalculator( ()->{ if(points.size()==3){...}
                                 ...; } );
```

A Note on Free Variables

- The value of a free variable must be fixed (or immutable).
  - Once a value is assigned to the variable, no re-assignments (value changes) are allowed.
- Traditionally, immutable variables are defined as `final`; free variables are often defined as `final`.
- In fact, a LE can access the variables that are not `final`, but they still have to be *effectively* `final`.
  - Even if they are not `final`, they need to be used as `final` if they are to be used in lambda expressions.

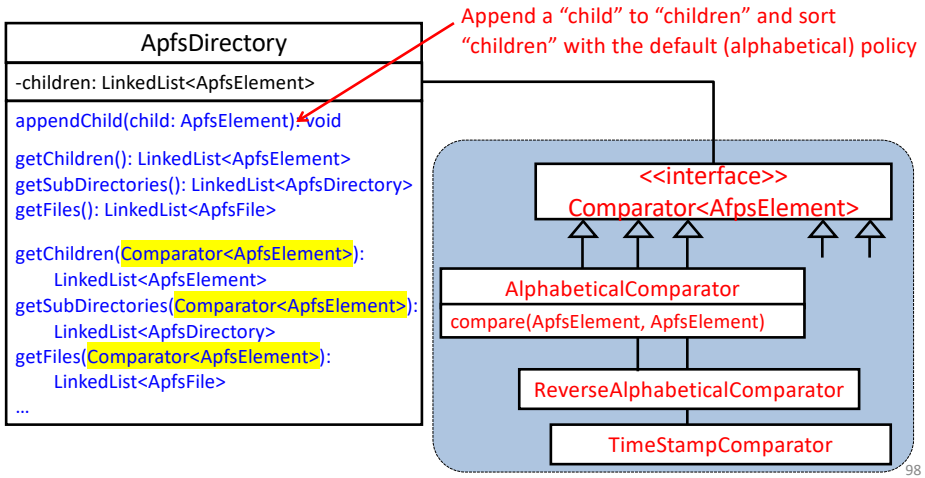



```
Polygon polygon = new Polygon( points,
                               ()->{ if(points.size()==3){...}
                                     ...; } );
                               // Passing the 3 points of a triangle
                               // and a triangle area calculator
polygon.getArea();
polygon.add( new Point(...) ); // Adding an extra point to transform
                               // the current triangle to a rectangle

polygon.setAreaCalculator( ()->{ if(points.size()==4){...}
                                 ...; } );
                               // Replacing the current triangle area
                               // calculator with a rectangle area
                               // calculator
polygon.getArea();
```

This code is perfectly fine; `points` is still effectively final.

HW 16

- Revise your HW 13 solution with lambda expressions.



- Instead of defining classes that implement `Comparator<ApfsElement>`, define the body of each `compare()` method as a lambda expression and pass it to `getChildren()`, `getSubDirectories()` and `getFiles()` of `ApfsDirectory`.
  - Re-sorts FS elements based on a custom (non-default) sorting policy, which is indicated by the method parameter, and returns re-sorted FS elements.
  - No need to change the bodies of `getChildren()`, `getSubDirectories()` and `getFiles()`
    - Just change their client code.

HW Submission Due

- December 22 (Sun) midnight
  - No extensions can be granted.

- Questions/inquiries: jxs@cs.umb.edu
- HW submissions: umasscs680@gmail.com