

# 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.
  - 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

1

2

## 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
  - Interface = a set of abstract functions
- Key design questions/challenges:
  - how to define modules
  - how to separate a module from others
  - how to let modules interact with each other

3

## SD v.s. OOD

- OOD
  - Intends coarse-grained modularity
    - The size of each code chunk 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 accommodate new/modified requirements.
    - How to make software more flexible/robust against changes in the future.
  - How to gain reusability, maintainability and extensibility?

4

## How to gain reusability, maintainability and extensibility?

- Many design patterns are identified and documented to answer this question.
- Recall, for example,
  - Iterator
    - Collection implementation and its access mechanism
  - Command
    - Command senders, receivers and commands
  - Visitor
    - Visitors and data structures that they are applied to

5

## Functional Programming with Java

## Looking Ahead: AOP, 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.

6

## Notable Enhancements in Java 8

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

7

8

## 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 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){  
    System.out.println(...);}  
    else{  
    System.out.println(...);} ] )`

9

- No need to specify the name of 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 return value's type.
  - Your Java compiler automatically infers that.
- Single-expression code block
  - Does not use the "return" keyword.

11

## 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
```

10

- A lambda expression consists of
    - A code block
    - A set of parameters to be passed to the code block
- ```
- (double threshold)-> {  
    if(Math.random() > threshold) return true;  
    else return false;  
}  
- () -> {  
    if(Math.random() > 0.5) return true;  
    else return false;  
}
```

12

- Single-expression code block
  - Does not use the “return” keyword.
- Multi-expression code block
  - Surrounds expressions with { and }. Use ; in the end of each expression.
  - Needs the “return” keyword in the end of each control flow.

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

13

## Functional Interface

- A special type of interface
  - An interface that has a single abstract (or empty) method.
- An example functional interface: `java.util.Comparator`
  - Defines `compare()`, which is the only abstract method.
  - A new annotation is available:
    - `@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...”
- `Collections.sort(List, Comparator<T>)`
  - The second parameter can accept a lambda expression.
  - ```
Collections.sort( aList,
    (Integer first, Integer second)->
      second.intValue()-first.intValue() );
```

15

## How to Pass a Lambda Expression?

- A method can receive a lambda expression(s).
  - ```
foo.example( (int first, int second) -> second-first )
```
  - The method receives a lambda expression as a parameter.
  - What is the type of that parameter?
    - *Functional interface!*

14

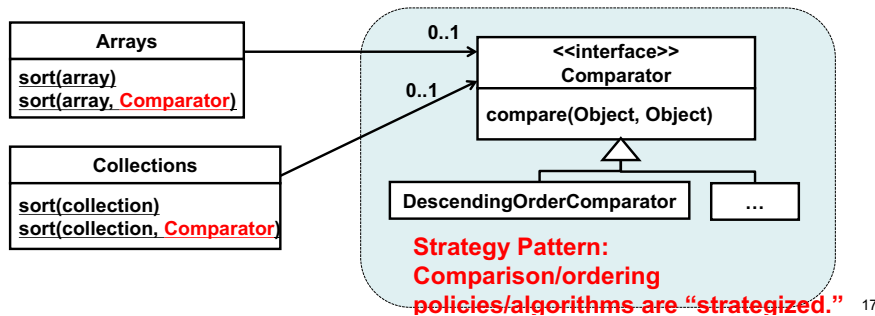
## Recap: Comparators

- Sorting collection elements:
  - ```
ArrayList<Integer> years2 = new ArrayList<Integer>();
years2.add(new Integer(2010));
years2.add(new Integer(2000));
years2.add(new Integer(1997));
years2.add(new Integer(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

16

# Comparison/Ordering Policies

- What if you want to sort array/collection elements in a descending order or any specialized (user-defined) order?
  - `Arrays.sort()` and `Collections.sort()` implement ascending ordering only.
    - They do not implement any other policies.
- Define a custom comparator by implementing `java.util.Comparator`



17

- `Arrays.sort()` and `Collections.sort()` are defined to sort array/collection elements from “smaller” to “bigger” elements.
  - By default, “smaller” elements mean the elements that have lower numbers.
- A descending ordering can be implemented by treating “smaller” elements as the elements that have higher numbers.
- `compare()` in comparator classes can define (or re-define) what “small” means and what’s “big” means.
  - Returns a negative integer, zero, or a positive integer as the first argument is “smaller” than, “equal to,” or “bigger” than the second.
- ```
public class DescendingOrderComparator implements Comparator{
    public int compare(Object o1, Object o2){
        return ((Integer)o2).intValue()-((Integer) o1).intValue();
    }
}
```

18

## 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

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

- A more type-safe option is available/recommended:

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

## Okay, so What's the Point?

- 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());
```
- Code gets more concise (shorter and simpler).
  - The lambda expression defines `DescendingOrderComparator`'s `compare()` in a concise way.
  - More readable and less ugly than the code based on an anonymous class.
- The LE version is a *syntactic sugar* for the non-LE version.
  - Your compiler does program transformation at compilation time.

21

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

- You are trying to use `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 which method is the only abstract/empty (i.e., non-static, non-default) method.
  - `public int compare(T o1, T o2)`
- Define a lambda expression to represent the method body of `compare()` and pass it to `sort()`.
  - `Collections.sort( aList,`  
 `(Integer first, Integer second)->`  
 `second.intValue()-first.intValue())`

23

## 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 lambda expression

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

22

## Assignment of a LE to a Functional Interface

- `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..."
- 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 thru type inference.
  - `Comparator<Integer> comparator =`  
 `(o1, o2)-> o2.intValue()-o1.intValue()`
    - C.f. Type inference with the diamond operator (introduced in Java 7).

24

## 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 =
(Integer o1, Integer o2)-> o2.intValue()-o1.intValue();
comparator.compare(1, 10);
```
  - compare() returns 9 (10 - 1).

25

## 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 cannot be assigned to object.
  - ```
Object comparator =
(Integer o1, Integer o2)-> o2.intValue()-o1.intValue()
```

26

- 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.
    - compare() is expected to return an int value.

27

- 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 XYZException
}
```
  - ```
Collections.sort(years, (Integer o1, Integer o2)->{
    if(...) throw new XYZException;
    else return ... });
```

28

## LEs make Your Code Concise, but...

- You still need to clearly understand
  - the Strategy design pattern
    - Comparator and its implementation classes
    - What `compare()` is meant to do
  - How `Collection.sort()` calls `compare()`.
- 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.
  - This may or may not mean “easier to understand” depending on how much you are familiar with lambda expressions.