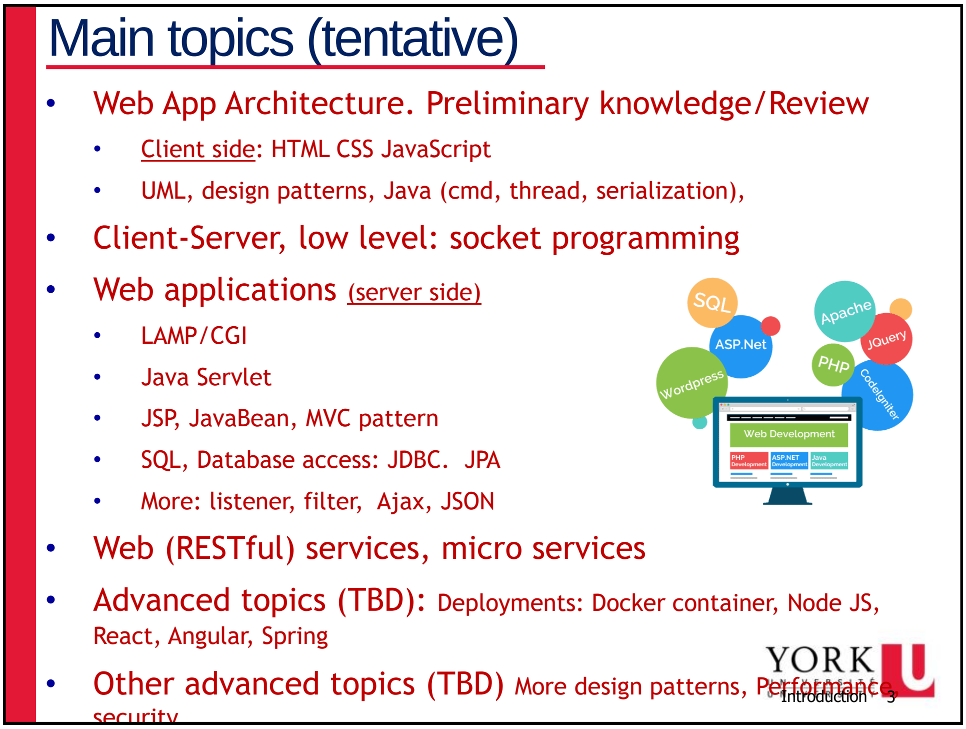
LE/EECS4413

Building E-commerce Systems

F 2023

Sep 15, 2023 Lecture 2.

1



Main topics (tentative)

- Web App Architecture. Preliminary knowledge/Review
  - Client side: HTML CSS JavaScript
  - UML, design patterns, Java (cmd, thread, serialization),
- Client-Server, low level: socket programming
- Web applications (server side)
  - LAMP/CGI
  - Java Servlet
  - JSP, JavaBean, MVC pattern
  - SQL, Database access: JDBC. JPA
  - More: listener, filter, Ajax, JSON
- Web (RESTful) services, micro services
- Advanced topics (TBD): Deployments: Docker container, Node JS, React, Angular, Spring
- Other advanced topics (TBD) More design patterns, Performance, security

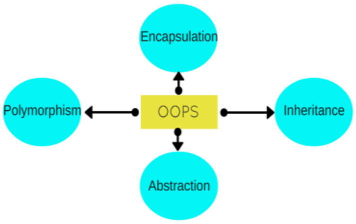
3

Preliminary knowledge

- Client side: HTML CSS JS

- Java review
  - OOP, UML, design pattern
  - Command line, class files, jar files
  - Multithreading in JAVA
  - Serialization

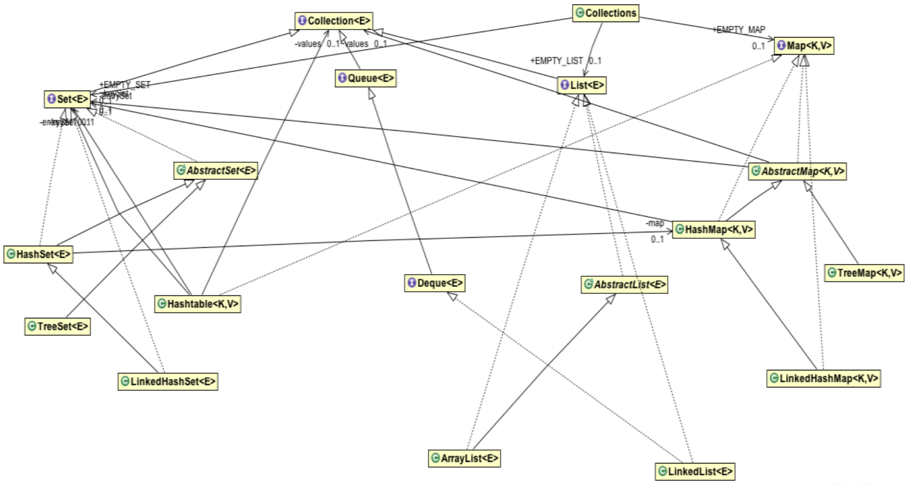
- Relational database and SQL



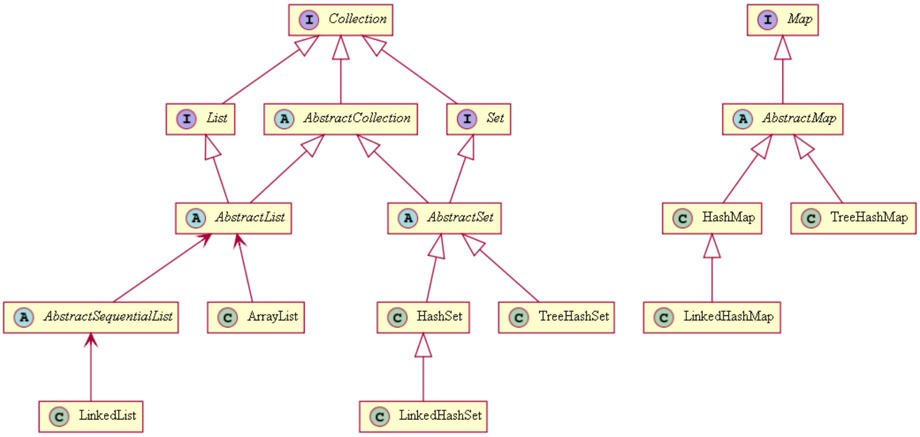
Java review: Java Collection Framework

- made up of:
  - interfaces
    - these define what methods the various types of collections support
  - abstract classes. concrete classes
    - these implement the interfaces
  - algorithms
    - these are the methods that operate on collections (such as sorting a collection and searching a collection)

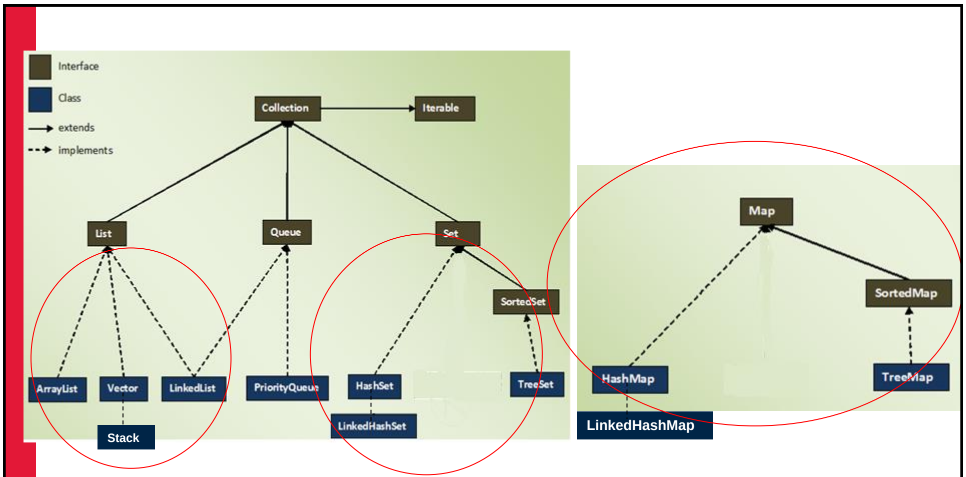
Java Collection Framework



Java Collection Framework



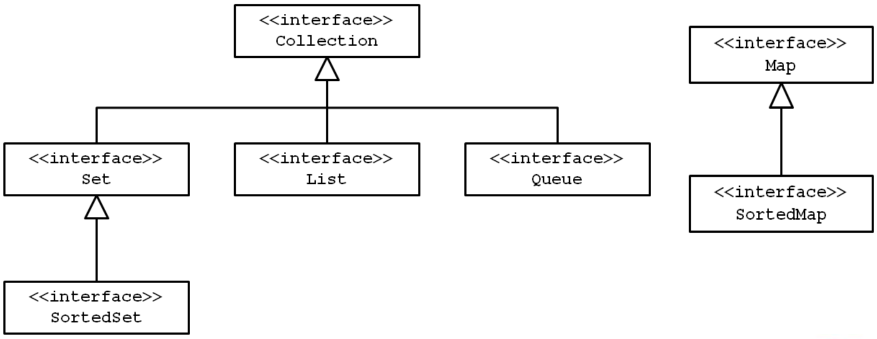
Simplified view



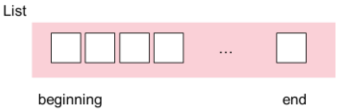
Very simplified view (no abstract classes)

Collection interface hierarchy

defines the most general operations that a client can ask a collection to perform

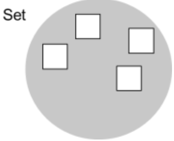


Summary

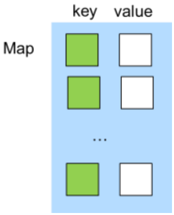


- `List`: ordered list
  - `add(o), add(i,o), remove(o), remove(i), get(o), get(i)`
  - `set(i)` to replace
  - shift for `add(), remove()` change indexes of all subsequent elements
  - ArrayList, LinkedList

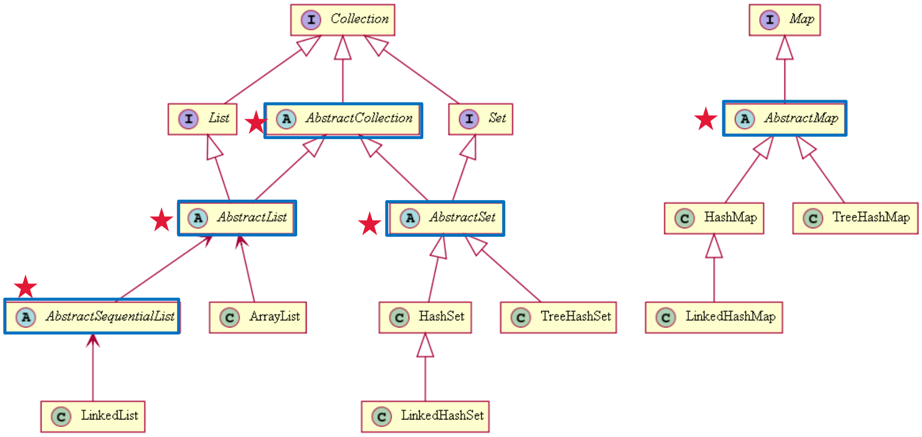
- `Set`: unordered, no duplicate element `add` reject
  - No get. Use iterator or for-each loop to traverse
  - Union: `allAll()`,
  - Intersection: `retainAll()`
  - HashSet, LinkedhashSet, TreeSet



- `Map`: “dictionary”.
  - `put(k, v)` add entry. replace if k exist.
  - `get(k)` return value. return null if k not exist
  - Query: `containsKey(k) containsValues(v)`
  - Traverse: `keySet(), values(), entrySet()`
  - HashMap, LinkedhashMap, TreeMap



Abstract classes, Interfaces



```
public abstract class AbstractList<E> extends AbstractCollection<E> implements List<E> {
    /**
     * Sole constructor.  (For invocation by subclass constructors, typically
     * implicit.)
     */
```

Abstract Classes

If the base class has methods that only subclasses can define,

- We would like to postpone the definition and give it to derived class to implement
- We would like to add a ‘note’ that: “*there will be a method bark() for each Dog but I don’t yet know how it is defined”*

- In Java, to leave the node, make the method abstract.
  - methods that have no implementation (empty body)
  - ; in place of the missing body

```
public abstract void bark();
```

Higher lever view

- As long as there is one abstract method, the class must be declared as abstract class*

** syntactically, abstract class can have 0 or all abstract methods

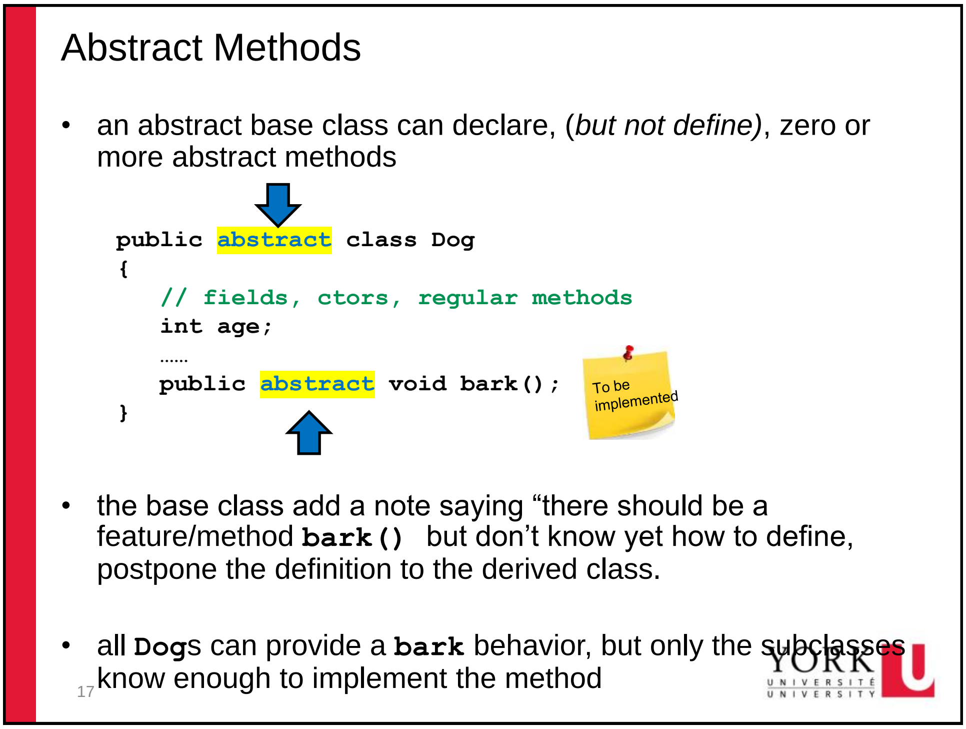
- An abstract class provides a partial definition of a class
  - the "partial definition" contains everything that is common to all of the subclasses.
  - the subclasses complete the definition

- An abstract class can define fields and normal methods
  - subclasses *inherit* these

- An abstract class can declare abstract methods
  - methods that have no implementation (empty body)
  - subclasses *implement* these
  - cannot be final -- subclasses *must define* these (unless the subclass is also abstract)

* abstract method also cannot be private, static

- An abstract class can define constructors
  - not for public (cannot create instance).
  - For subclasses to call (explicitly or implicitly)



Abstract Methods

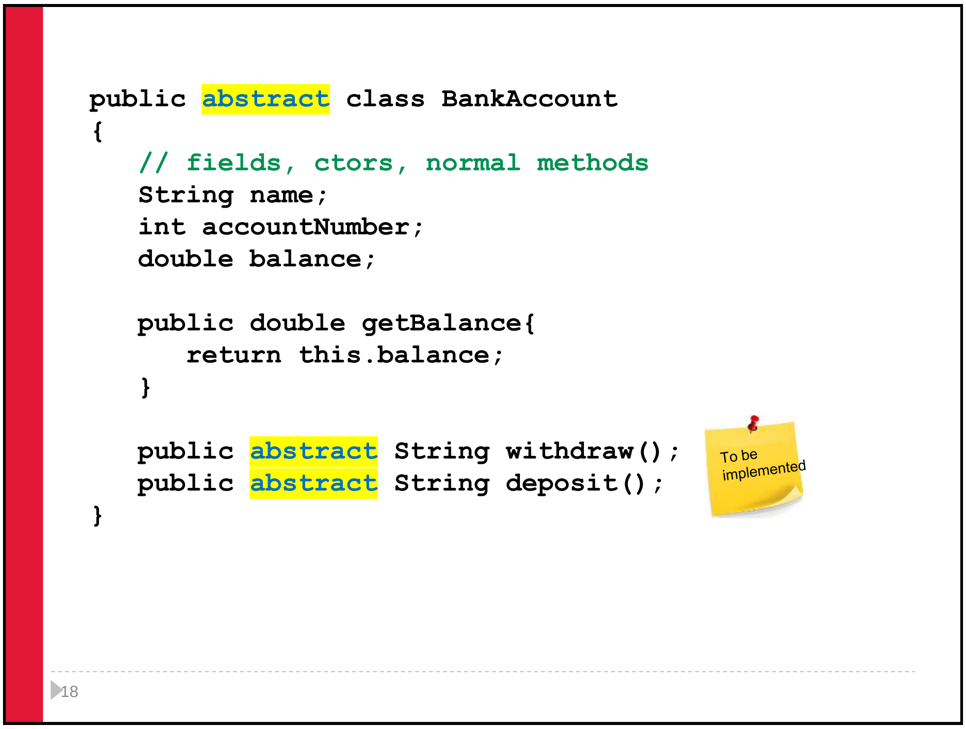
- an abstract base class can declare, (*but not define)*, zero or more abstract methods

```
public abstract class Dog
{
    // fields, ctors, regular methods
    int age;
    ……
    public abstract void bark();
}
```

To be implemented

- the base class add a note saying “there should be a feature/method **bark()** but don’t know yet how to define, postpone the definition to the derived class.
- all **Dog**s can provide a **bark** behavior, but only the subclasses know enough to implement the method

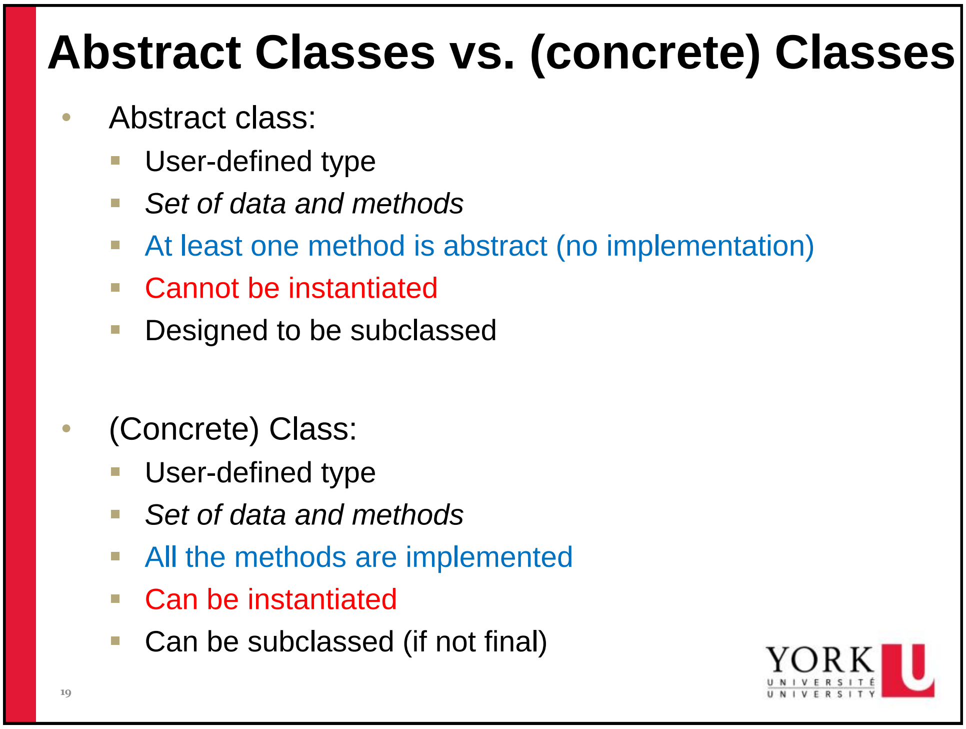
17



```
public abstract class BankAccount
{
    // fields, ctors, normal methods
    String name;
    int accountNumber;
    double balance;

    public double getBalance{
        return this.balance;
    }

    public abstract String withdraw();
    public abstract String deposit();
}
```

To be implemented

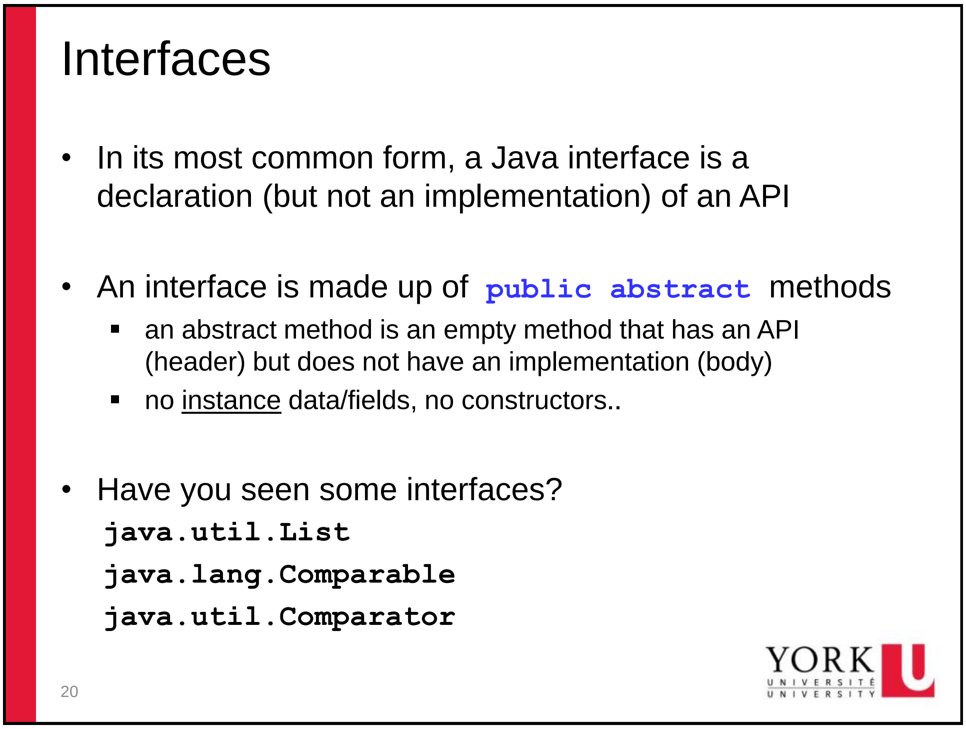
18



Abstract Classes vs. (concrete) Classes

- Abstract class:
  - User-defined type
  - *Set of data and methods*
  - At least one method is abstract (no implementation)
  - Cannot be instantiated
  - Designed to be subclassed

- (Concrete) Class:
  - User-defined type
  - *Set of data and methods*
  - All the methods are implemented
  - Can be instantiated
  - Can be subclassed (if not final)



Interfaces

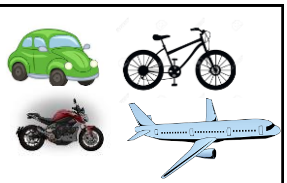
- In its most common form, a Java interface is a declaration (but not an implementation) of an API

- An interface is made up of `public abstract` methods
  - an abstract method is an empty method that has an API (header) but does not have an implementation (body)
  - no <u>instance</u> data/fields, no constructors..

- Have you seen some interfaces?

```
java.util.List
java.lang.Comparable
java.util.Comparator
```

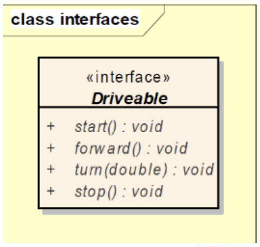
Example- Drivable Interface define behavior



```
public interface Drivable {
     public void start();
     public void forward();
     public void turn(double angle);
     public void stop();
}
```

semicolon, and no method body

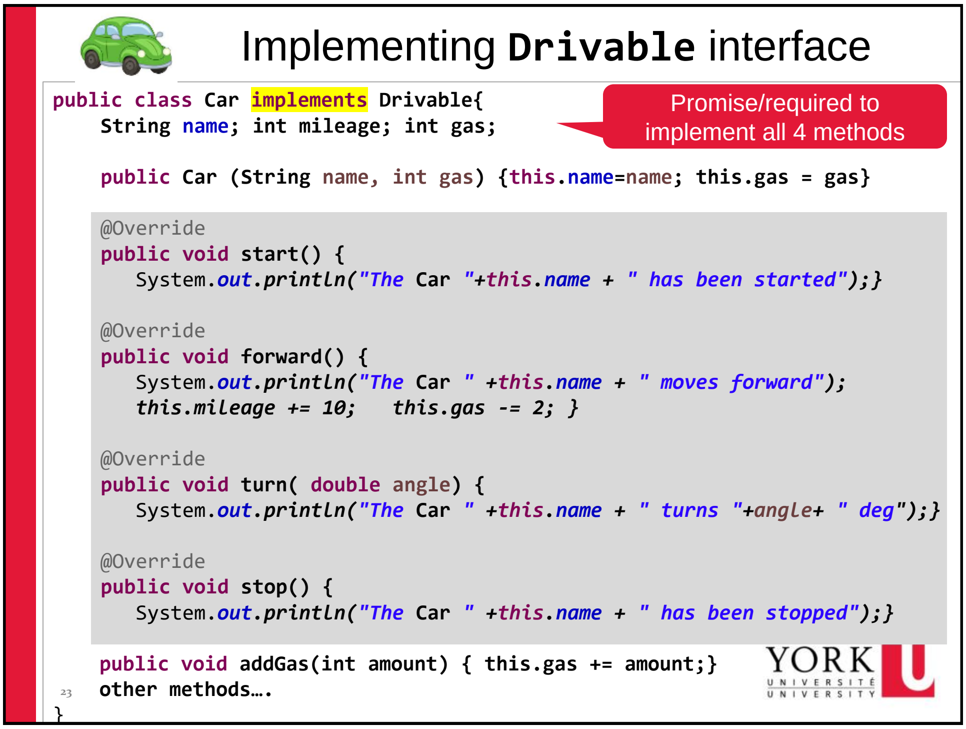
- notice that the interface declares which methods exist and specifies the contract of the methods
  - but it does not specify how the methods are implemented
- the method implementations are defined by classes that implement the interface



Implementing **Drivable** interface

```
public class Bicycle implements Drivable{
    String name; int mileage;

    public Bicycle (String name) {this.name=name;}

    @Override
    public void start() {
       System.out.println("The Bicycle "+this.name + " has been started");}

    @Override
    public void forward() {
       System.out.println("The Bicycle " +this.name + " moves forward");
       this.mileage += 1; }

    @Override
    public void turn( double angle) {
       System.out.println("The Bicycle " +this.name + " turns "+angle);}

    @Override
    public void stop() {
       System.out.println("The Bicycle " +this.name + " has been stopped");}

    public void fixPedal() {…}
    other methods……
}
```

Promise/required to implement all 4 methods declared in Drivable



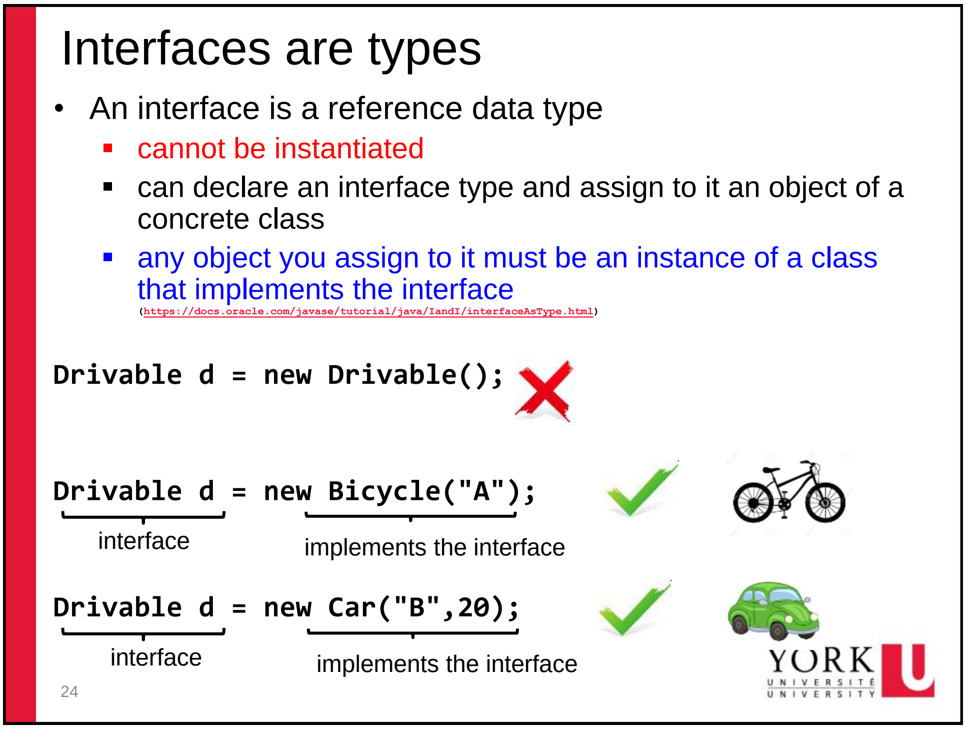
Implementing `Drivable` interface

```
public class Car implements Drivable{
    String name; int mileage; int gas;

    public Car (String name, int gas) {this.name=name; this.gas = gas}

    @Override
    public void start() {
       System.out.println("The Car "+this.name + " has been started");}

    @Override
    public void forward() {
       System.out.println("The Car " +this.name + " moves forward");
       this.mileage += 10;   this.gas -= 2; }

    @Override
    public void turn( double angle) {
       System.out.println("The Car " +this.name + " turns "+angle+ " deg");}

    @Override
    public void stop() {
       System.out.println("The Car " +this.name + " has been stopped");}

    public void addGas(int amount) { this.gas += amount;}
    other methods….
}
```

Promise/required to implement all 4 methods

23

23



Interfaces are types

- An interface is a reference data type
  - cannot be instantiated
  - can declare an interface type and assign to it an object of a concrete class
  - any object you assign to it must be an instance of a class that implements the interface (https://docs.oracle.com/javase/tutorial/java/IandI/interfaceAsType.html)

```
Drivable d = new Drivable();
```

✗

```
Drivable d = new Bicycle("A");
```

✓

$\underbrace{\texttt{Drivable d}}_{\text{interface}} = \texttt{new}\ \underbrace{\texttt{Bicycle("A")}}_{\text{implements the interface}};$

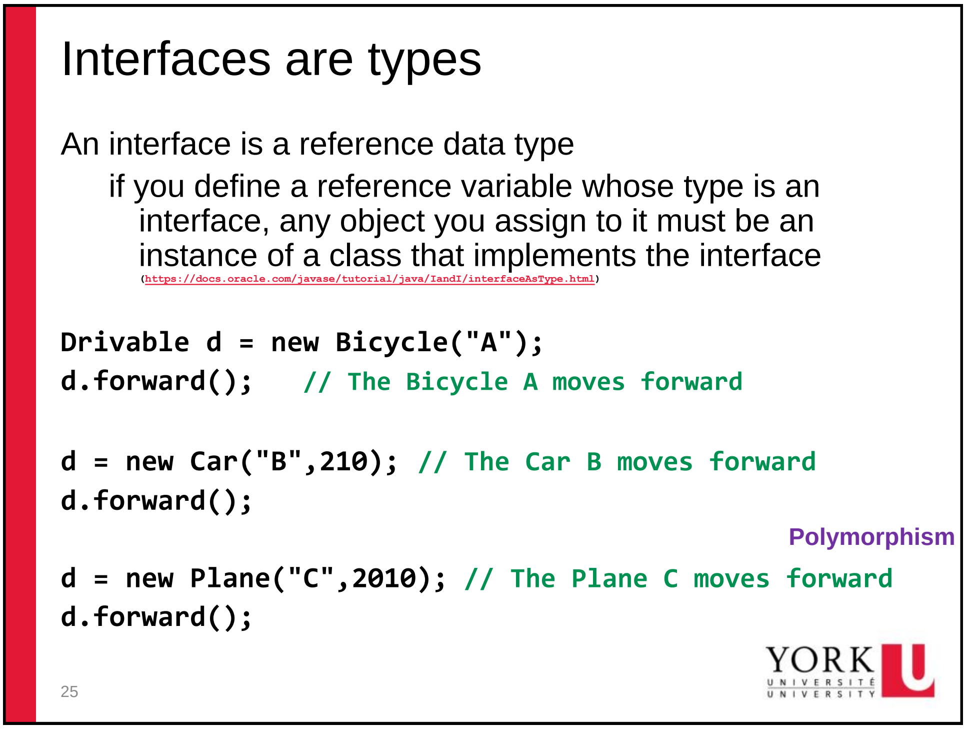
```
Drivable d = new Car("B",20);
```

✓

$\underbrace{\texttt{Drivable d}}_{\text{interface}} = \texttt{new}\ \underbrace{\texttt{Car("B",20)}}_{\text{implements the interface}};$

24

24



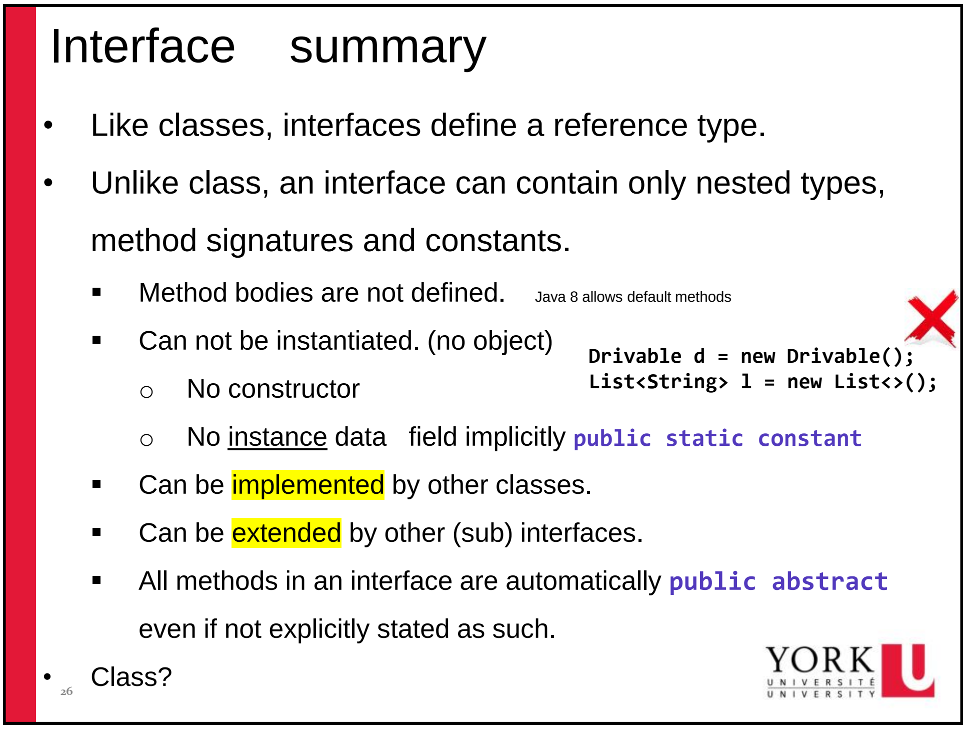
Interfaces are types

An interface is a reference data type

if you define a reference variable whose type is an interface, any object you assign to it must be an instance of a class that implements the interface (https://docs.oracle.com/javase/tutorial/java/IandI/interfaceAsType.html)

```
Drivable d = new Bicycle("A");
d.forward();    // The Bicycle A moves forward

d = new Car("B",210); // The Car B moves forward
d.forward();

d = new Plane("C",2010); // The Plane C moves forward
d.forward();
```

Polymorphism



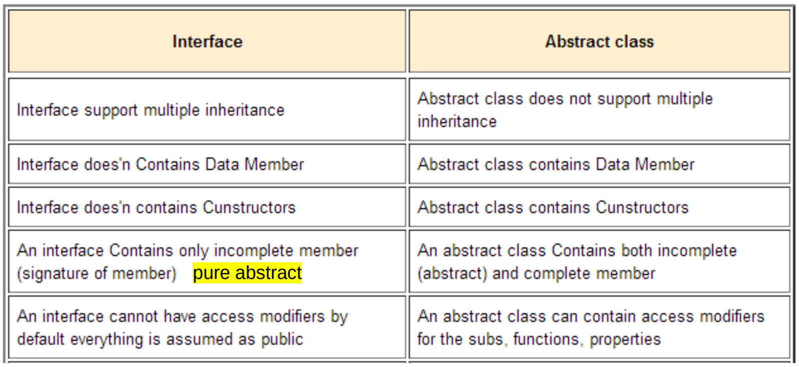
Interface summary

- Like classes, interfaces define a reference type.
- Unlike class, an interface can contain only nested types, method signatures and constants.
  - Method bodies are not defined. Java 8 allows default methods
  - Can not be instantiated. (no object)
    - No constructor
    - No instance data field implicitly `public static constant`
  - Can be implemented by other classes.
  - Can be extended by other (sub) interfaces.
  - All methods in an interface are automatically `public abstract` even if not explicitly stated as such.
- Class?

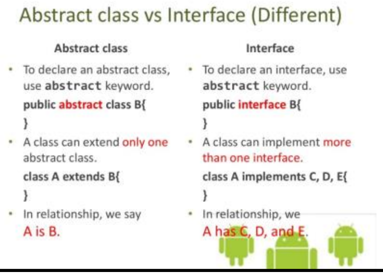
✗

```
Drivable d = new Drivable();
List<String> l = new List<>();
```

Abstract class vs interface



Interface	Abstract class
Interface support multiple inheritance	Abstract class does not support multiple inheritance
Interface does'n Contains Data Member	Abstract class contains Data Member
Interface does'n contains Cunstructors	Abstract class contains Cunstructors
An interface Contains only incomplete member (signature of member) pure abstract	An abstract class Contains both incomplete (abstract) and complete member
An interface cannot have access modifiers by default everything is assumed as public	An abstract class can contain access modifiers for the subs, functions, properties



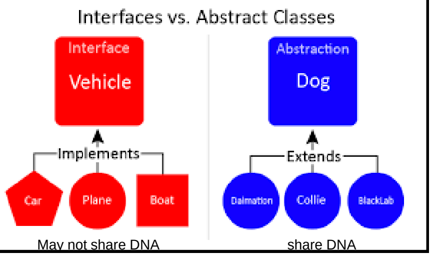
Abstract class vs Interface (Different)

Abstract class

- To declare an abstract class, use **abstract** keyword.
  public abstract class B{
  }
- A class can extend only one abstract class.
  class A extends B{
  }
- In relationship, we say
  A is B.

Interface

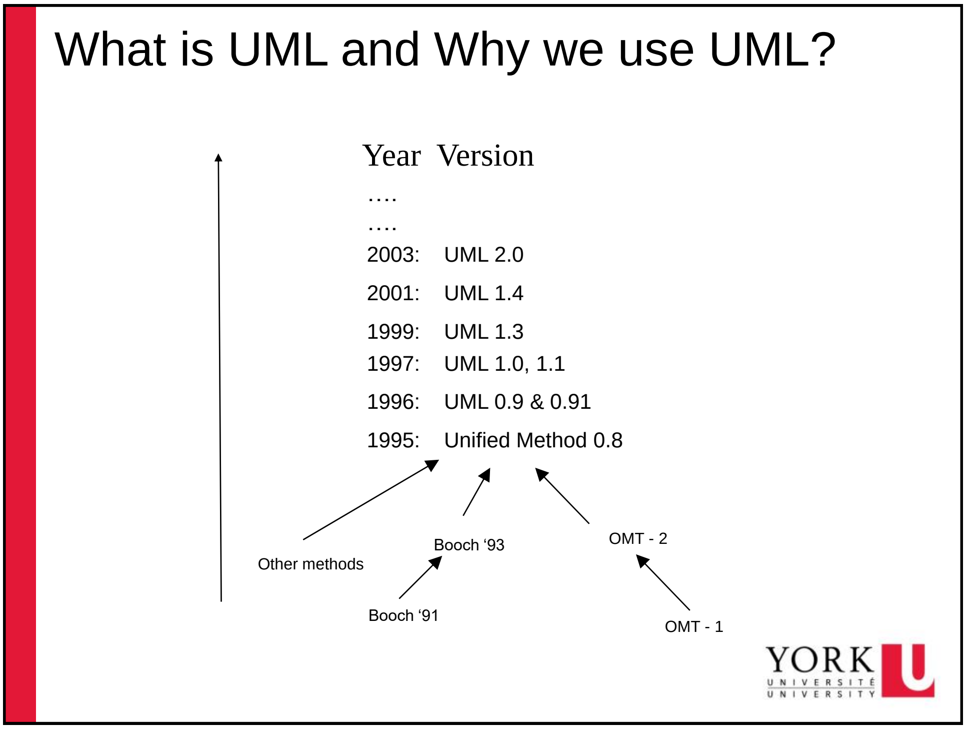
- To declare an interface, use **abstract** keyword.
  public interface B{
  }
- A class can implement more than one interface.
  class A implements C, D, E{
  }
- In relationship, we
  A has C, D, and E.



What is UML and Why we use UML?

- UML stands for "**Unified Modeling Language**"
- It is a industry-standard graphical language .
- UML is a pictorial language used to make software blue prints
- It is used for specifying, visualizing, constructing, and documenting the artifacts of software systems
- UML is different from the other common programming languages
- It uses mostly graphical notations.
- Simplifies the complex process of software design

- UML is *not* dependent on any one language or technology.
- UML can be defined as a simple modeling mechanism to model all possible practical systems in today's complex environment.
- *"A picture is worth than thousand words"*



What is UML and Why we use UML?

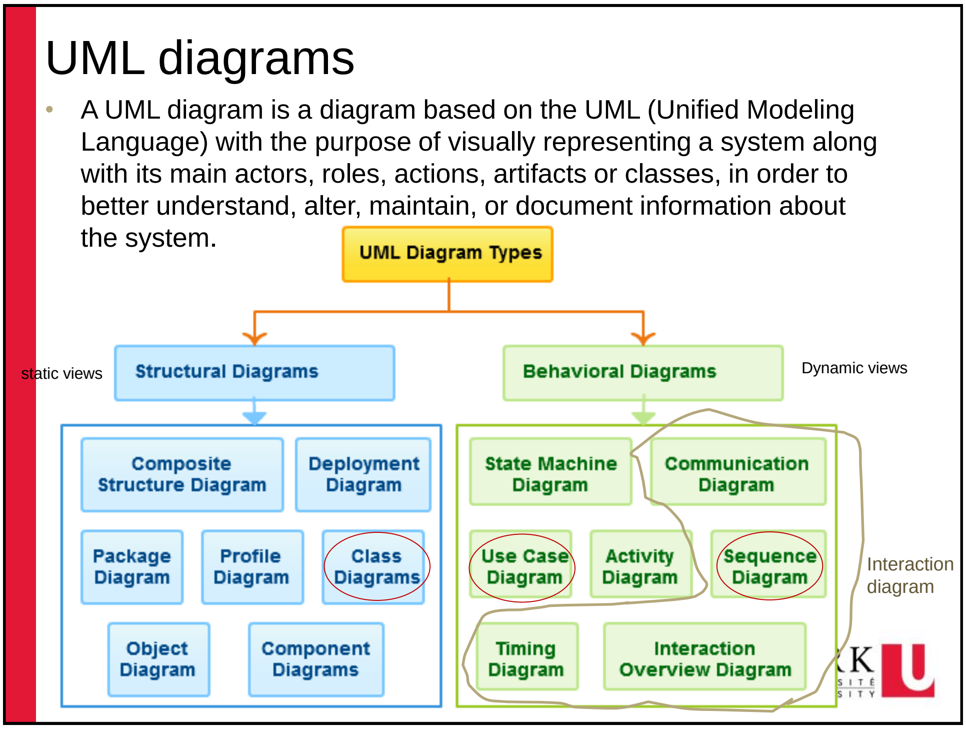
Year Version

….

….

2003:	UML 2.0
2001:	UML 1.4
1999:	UML 1.3
1997:	UML 1.0, 1.1
1996:	UML 0.9 & 0.91
1995:	Unified Method 0.8

29



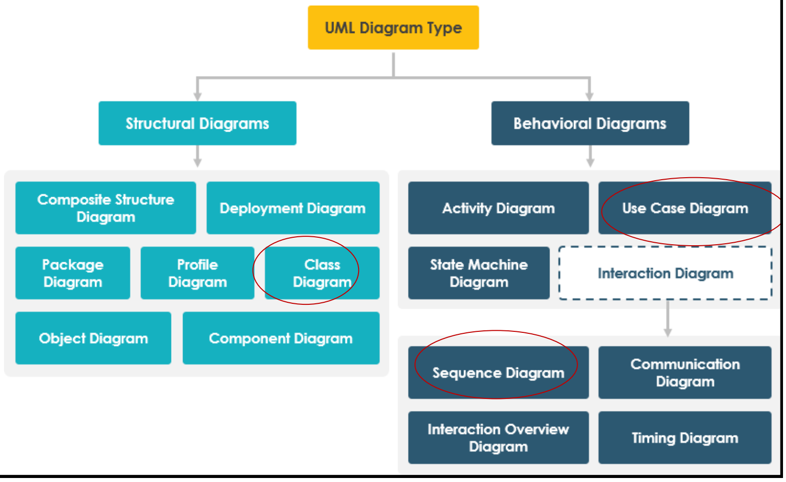
UML diagrams

- A UML diagram is a diagram based on the UML (Unified Modeling Language) with the purpose of visually representing a system along with its main actors, roles, actions, artifacts or classes, in order to better understand, alter, maintain, or document information about the system.

31

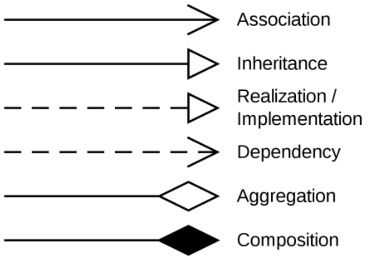
UML diagrams

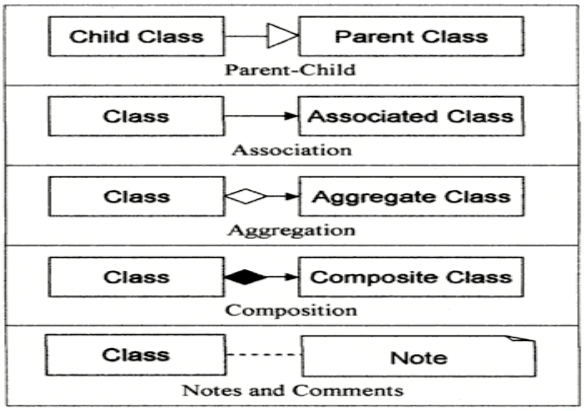
- A UML diagram is a diagram based on the UML (Unified Modeling Language) with the purpose of visually representing a system along with its main actors, roles, actions, artifacts or classes, in order to better understand, alter, maintain, or document information about the system.



32

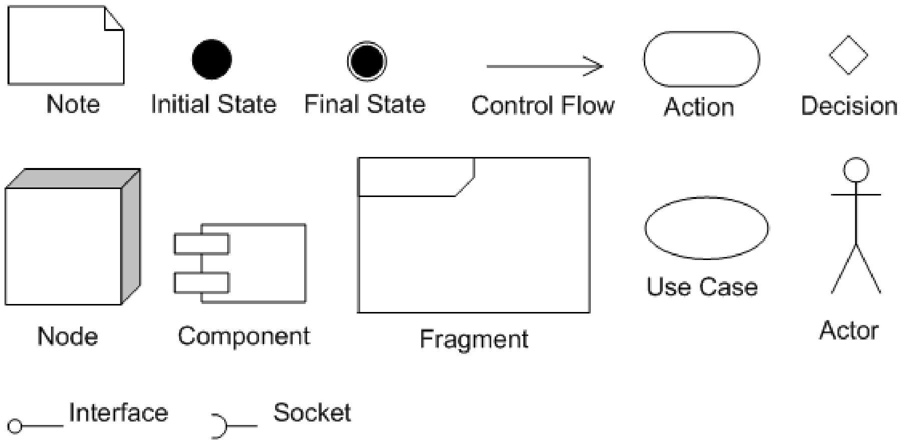
32





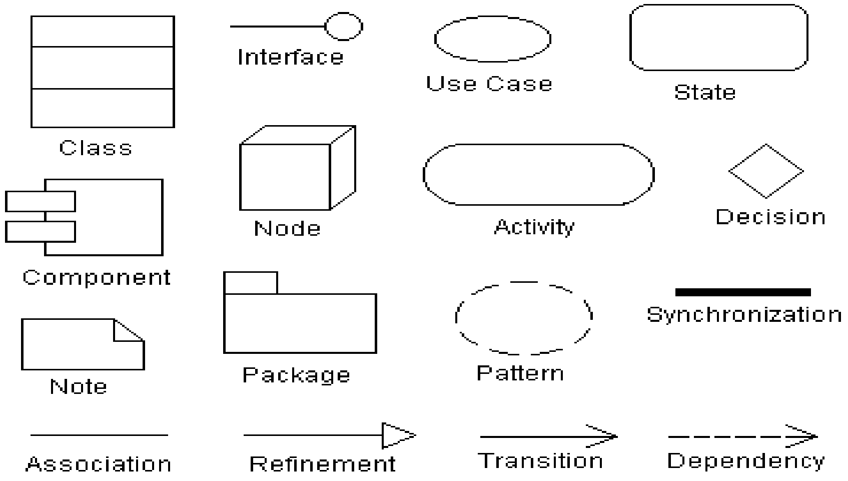
33

Notations



34

Notations

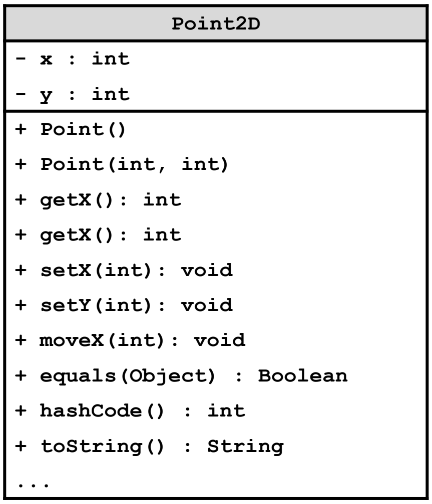


35

Class relations and UML class diagram

- **Public** members are shown by +
- **Private** members are shown by -
- **Protected** members are shown by #
- **Package** members are shown by ~

Class Diagram

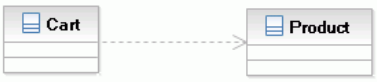


```
Point2D
- x : int
- y : int
+ Point()
+ Point(int, int)
+ getX(): int
+ getX(): int
+ setX(int): void
+ setY(int): void
+ moveX(int): void
+ equals(Object) : Boolean
+ hashCode() : int
+ toString() : String
...
```

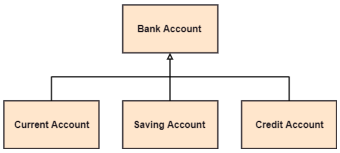
(Unified Modeling Language) is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems

Class diagram, relationships

- **Dependency:** A dependency is a semantic relationship between two or more classes where a change in one class cause changes in another class. It forms a weaker relationship. One class depends on another if the independent class is **a parameter variable or local variable** of a method of the dependent class.

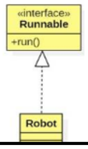


- **Generalization (extends):** A generalization is a relationship between a parent class (superclass) and a child class (subclass). In this, the child class is inherited from the parent class.



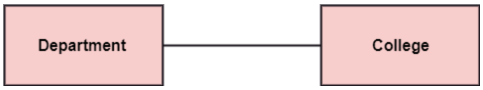
- **realization (implementation):**

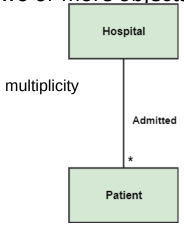
In a realization relationship of UML, one entity denotes some responsibility which is not implemented by itself and the other entity that implements them. This relationship is mostly found in the case of *interfaces*.



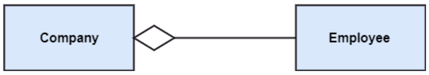
Class diagram, relationships (cont')

•**Association:** It describes a static or physical connection between two or more objects. It depicts how many objects are there in the relationship.
For example, a department is associated with the college.



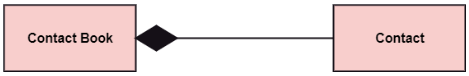


Aggregation: An aggregation is a subset of **association**, which represents has a relationship. It is more specific than association. It defines a part-whole or part-of relationship. In this kind of relationship, the child class can exist independently of its parent class.



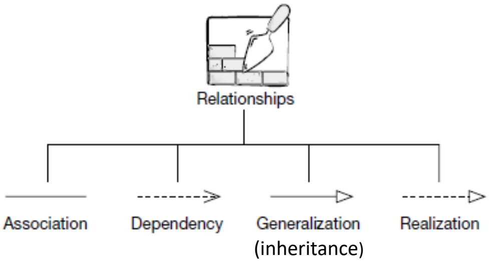
"is part of"

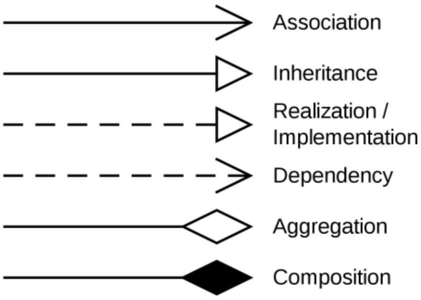
Composition: The composition is a subset (stronger version) of **aggregation**. It portrays the dependency between the parent and its child, which means if one part is deleted, then the other part also gets discarded. It represents a whole-part relationship.

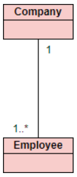


"is entirely made of"

Relations summary

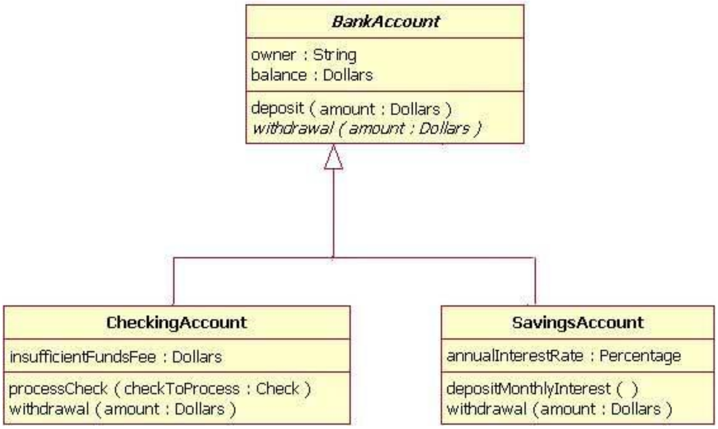






Multiplicities examples:

1	Exactly one, no more and no less
0..1	Zero or one
*	Many
0..*	Zero or many
1..*	One or many



BankAccount

owner : String
balance : Dollars

deposit (amount : Dollars)
withdrawal (amount : Dollars)

CheckingAccount

insufficientFundsFee : Dollars

processCheck (checkToProcess : Check)
withdrawal (amount : Dollars)

SavingsAccount

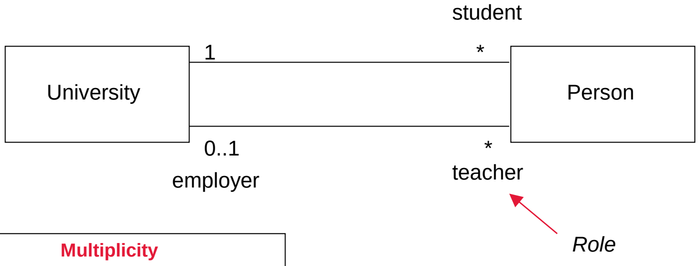
annualInterestRate : Percentage

depositMonthlyInterest ( )
withdrawal (amount : Dollars)

40

40

Association: Multiplicity and Roles



student

1

*

University

Person

0..1

*

employer

teacher

Role

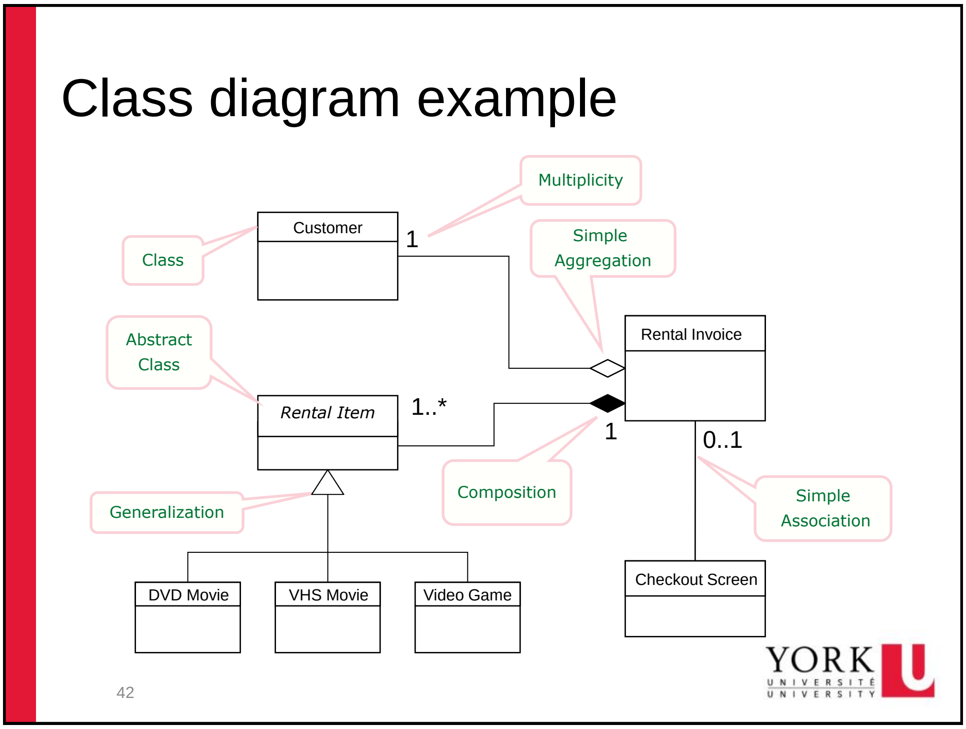
Multiplicity

Symbol	Meaning
1	One and only one
0..1	Zero or one
M..N	From M to N (natural language)
*	From zero to any positive integer
0..*	From zero to any positive integer
1..*	From one to any positive integer

Role

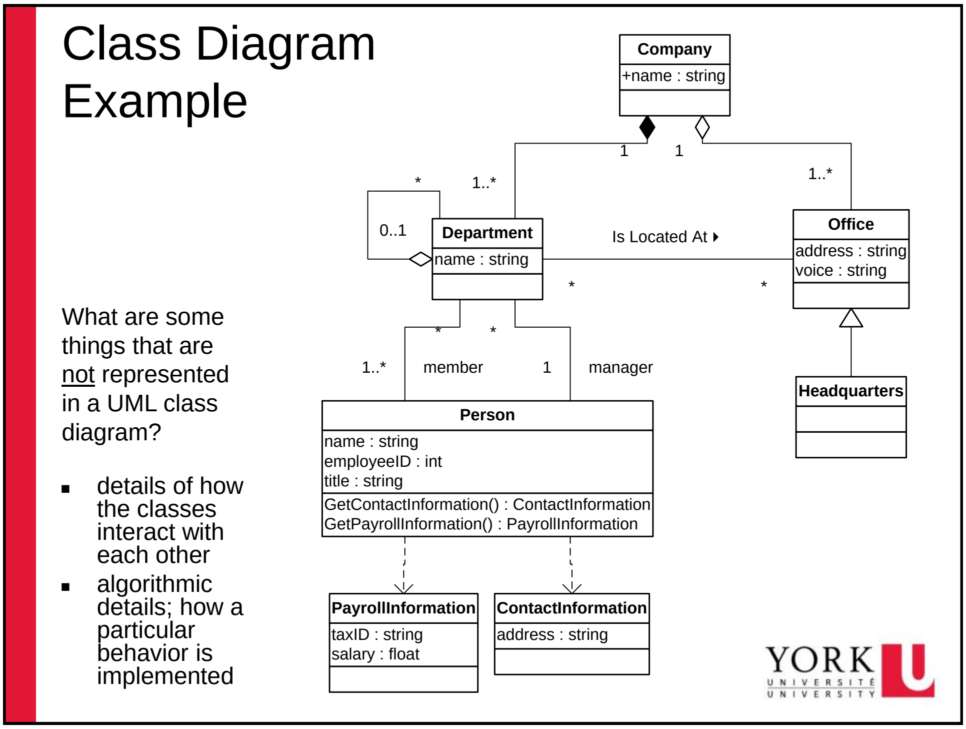
“A given university groups many people; some act as students, others as teachers. A given student belongs to a single university; a given teacher may or may not be working for the university at a particular time.”

41



Class diagram example

42



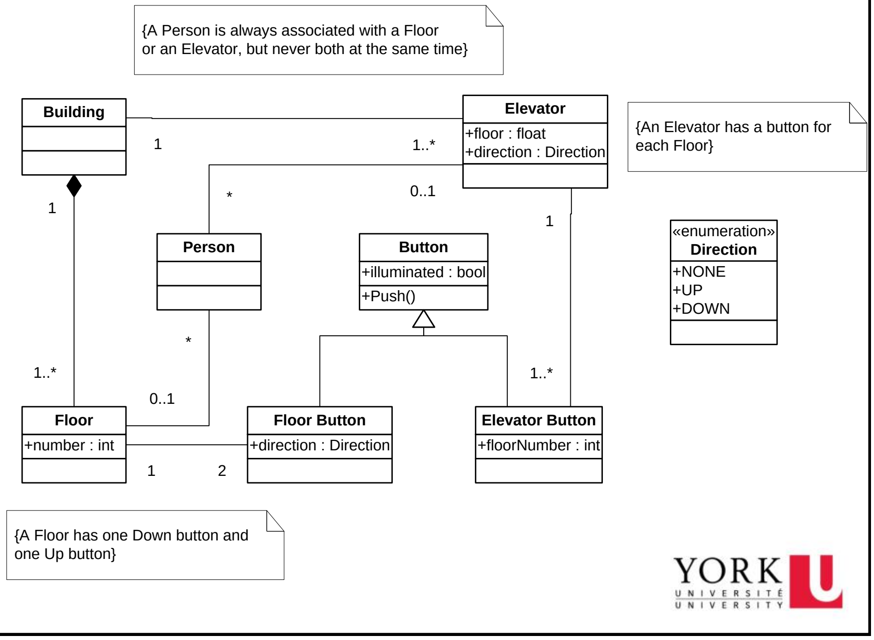
Class Diagram Example

What are some things that are <u>not</u> represented in a UML class diagram?

- details of how the classes interact with each other
- algorithmic details; how a particular behavior is implemented

43

Class Diagram Example



{A Person is always associated with a Floor or an Elevator, but never both at the same time}

{An Elevator has a button for each Floor}

{A Floor has one Down button and one Up button}

Building

Elevator
+floor : float
+direction : Direction

Person

Button
+illuminated : bool
+Push()

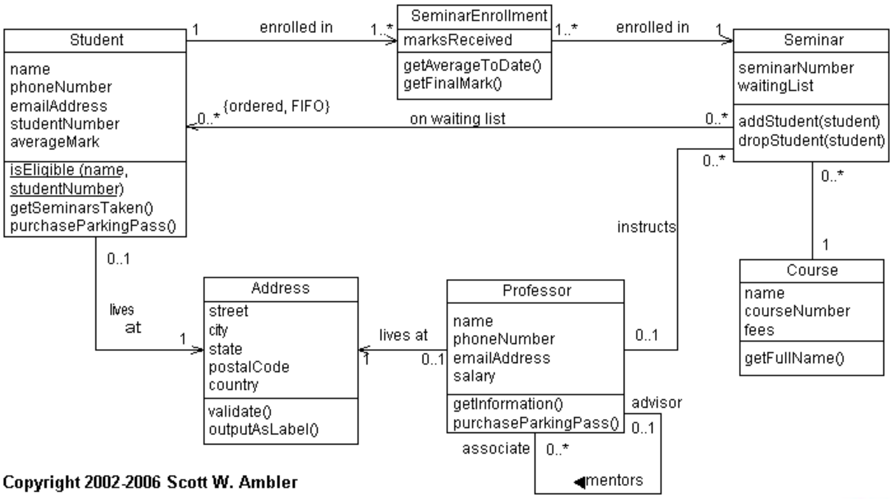
«enumeration»
Direction
+NONE
+UP
+DOWN

Floor
+number : int

Floor Button
+direction : Direction

Elevator Button
+floorNumber : int

Example UML Class Diagram



Student
name
phoneNumber
emailAddress
studentNumber
averageMark
isEligible (name, studentNumber)
getSeminarsTaken()
purchaseParkingPass()

SeminarEnrollment
marksReceived
getAverageToDate()
getFinalMark()

Seminar
seminarNumber
waitingList
addStudent(student)
dropStudent(student)

Address
street
city
state
postalCode
country
validate()
outputAsLabel()

Professor
name
phoneNumber
emailAddress
salary
getInformation()
purchaseParkingPass()

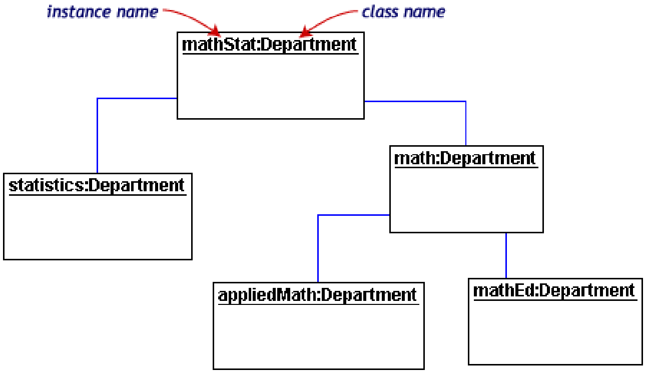
Course
name
courseNumber
fees
getFullName()

http://www.agiledata.org/images/oo101ClassDiagram.gif

45

Object diagram

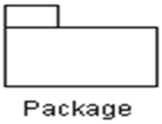
- UML 2 Object diagrams (instance diagrams), are useful for exploring real world examples of objects and the relationships between them. It shows instances instead of classes. They are useful for explaining small pieces with complicated relationships, especially recursive relationships.

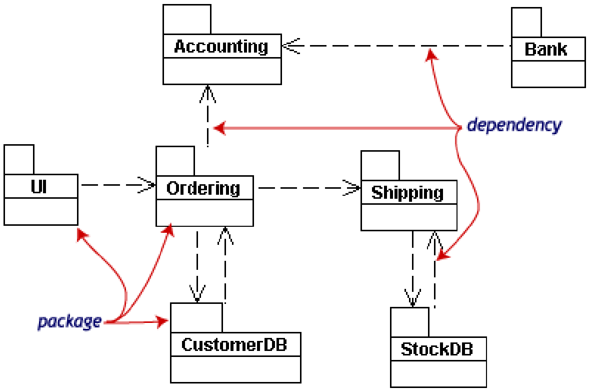


46

Package diagram

- UML 2 Package diagrams simplify complex class diagrams, it can group classes into **packages**. A package is a collection of logically related UML elements. Packages are depicted as file folders and can be used on any of the UML diagrams.





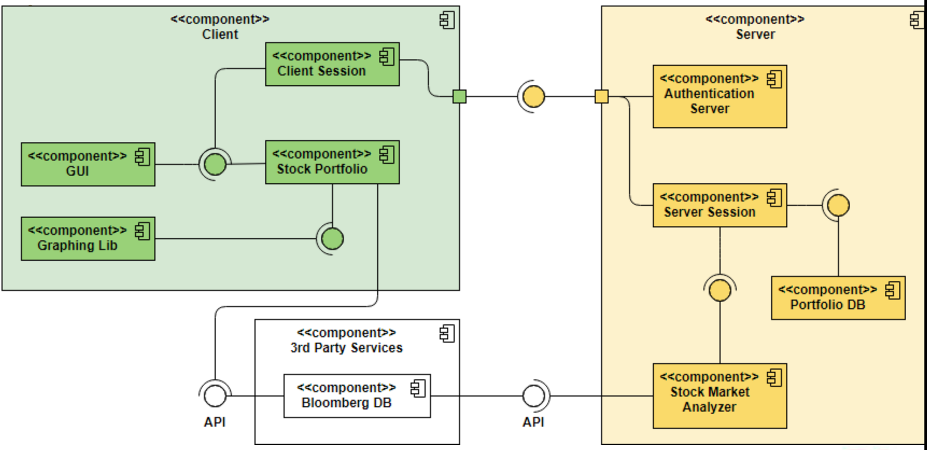
47

Component diagram

- Displays the structural relationship of components of a software system
- In UML, Components are made up of software objects that have been classified to serve a similar purpose.
- Components are considered autonomous, encapsulated units within a system or subsystem that provide one or more interfaces.
- By classifying a group of classes as a component the entire system becomes more modular as components may be interchanged and reused.

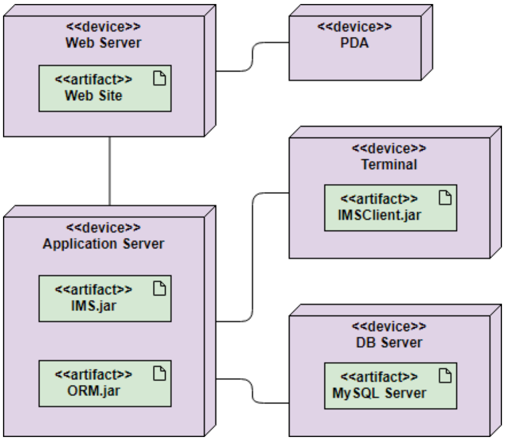
Component diagram

- Displays the structural relationship of components of a software system



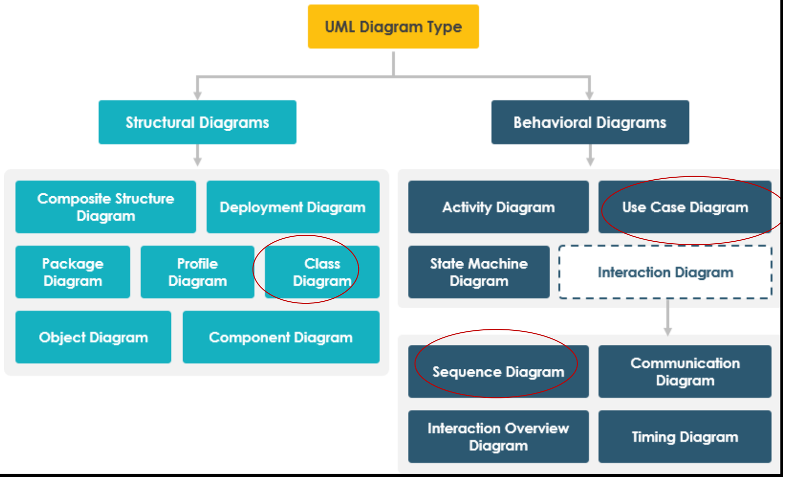
Deployment diagram

- Shows the hardware of your system and the software in those hardware.



UML diagrams

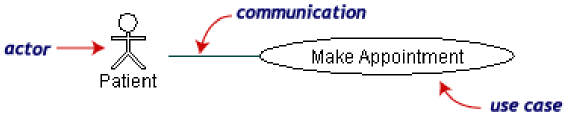
- A UML diagram is a diagram based on the UML (Unified Modeling Language) with the purpose of visually representing a system along with its main actors, roles, actions, artifacts or classes, in order to better understand, alter, maintain, or document information about the system.



Use cases diagram

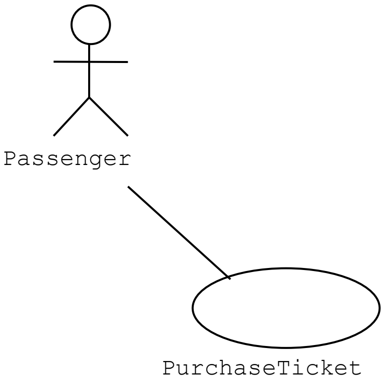
UML 2 Use cases diagrams describes the behavior of the target system from an external point of view. A use-case diagram is a set of use cases. A use case is a model of the interaction between External users of a software product (actors) and the software product itself

- **Use cases**. A use case describes a sequence of actions that provide something of measurable value to an actor and is drawn as a horizontal ellipse.
- **Actors**. An actor is a person, organization, or external system that plays a role in one or more interactions with your system. Actors are drawn as stick figures.
- **Associations**. Associations between actors and use cases are indicated by solid lines. An association exists whenever an actor is involved with an interaction described by a use case.



53

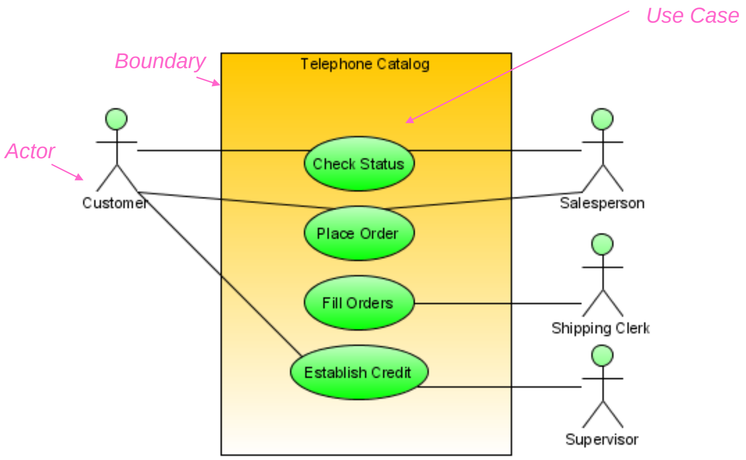
Use Case Diagrams



- Used during requirements elicitation to represent external behavior
- ***Actors*** represent roles, that is, a type of user of the system
- ***Use cases*** represent a sequence of interaction for a type of functionality
- The use case model is the set of all use cases. It is a complete description of the functionality of the system and its environment

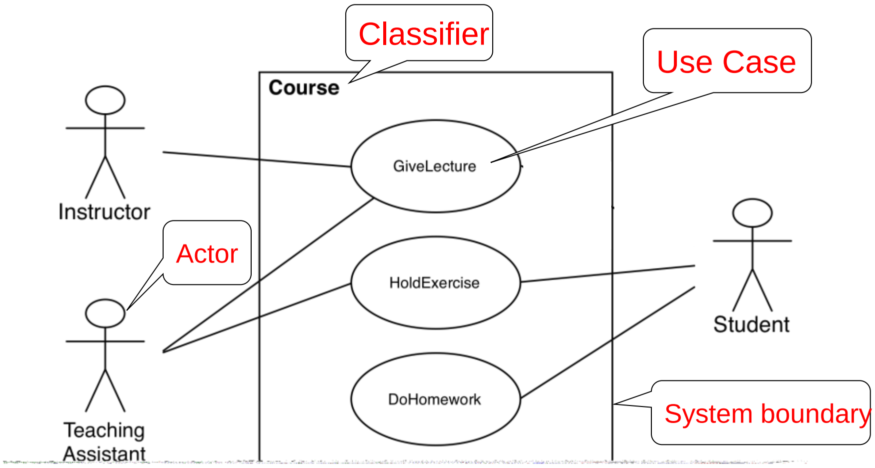
54

Use cases diagram



Use case diagrams represent the functionality of the system from user's point of view

Use case diagrams



Use case diagrams represent the functionality of the system from user's point of view

Use-Case Diagrams

Include: a dotted line labeled <<include>> beginning at base use case and ending with an arrows pointing to the include use case. The include relationship occurs when a chunk of behavior is similar across more than one use case. Use "include" in stead of copying the description of that behavior.

<<include>>

------------->

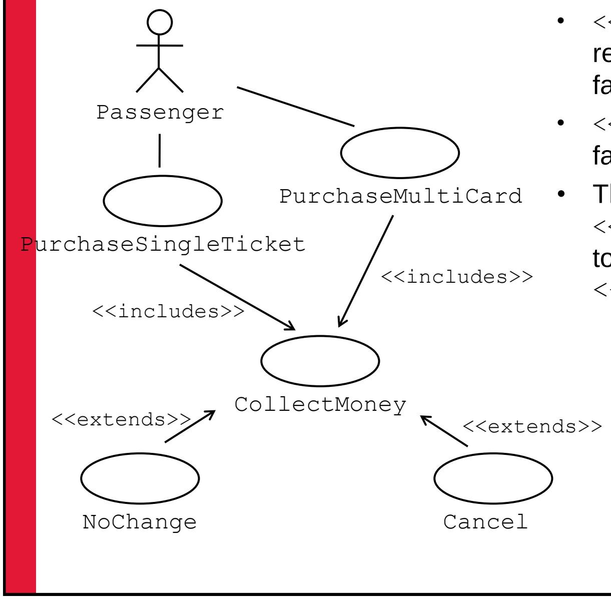
Extend: a dotted line labeled <<extend>> with an arrow toward the base case. The extending use case may add behavior to the base use case. The base class declares "extension points".

<<extend>>

--------------------->

58

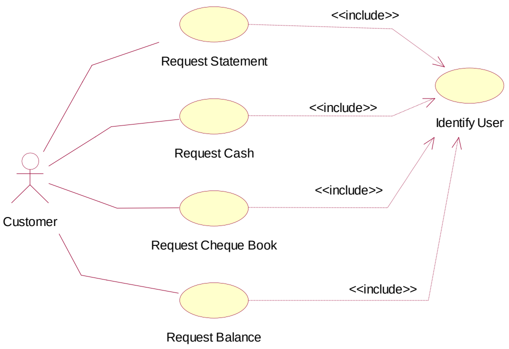
The <<includes>> Relationship



- <<includes>> relationship represents behavior that is factored out of the use case.
- <<includes>> behavior is factored out for reuse, .
- The direction of a <<includes>> relationship is to the using use case (unlike <<extends>> relationships).

59

The <<includes>> Relationship



- When designing use-cases it is sometimes apparent that there exists some commonality or replication between the steps involved in the execution of one or more use cases.

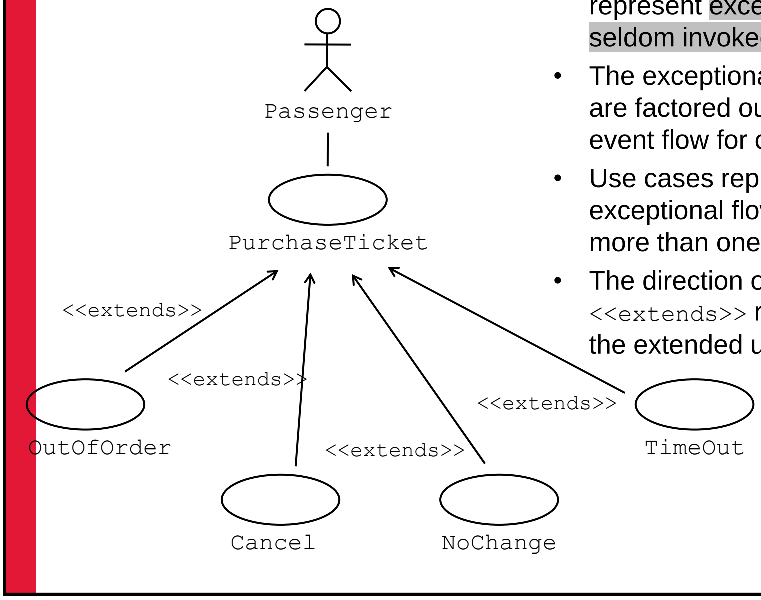
- In each one of the four use cases
  - *Request Cash*
  - *Request Balance*
  - *Request Statement*
  - *Request Cheque Book*

  the user is required to insert their ID card and enter their PIN, which is then verified by the bank central computer.

- Rather than duplicate this common user interaction within each of the above four use-case descriptions, we might extract it and chose to represent it with a mini-use-case called *'identify user'* whose functionality is included as part of the other four use-cases.

61

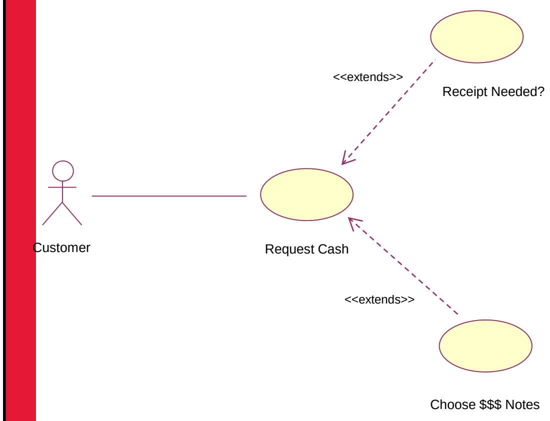
The <<extends>> Relationship



- <<extends>> relationships represent exceptional or seldom invoked cases.
- The exceptional event flows are factored out of the main event flow for clarity.
- Use cases representing exceptional flows can extend more than one use case.
- The direction of a <<extends>> relationship is to the extended use case

62

The <<extends>> Relationship

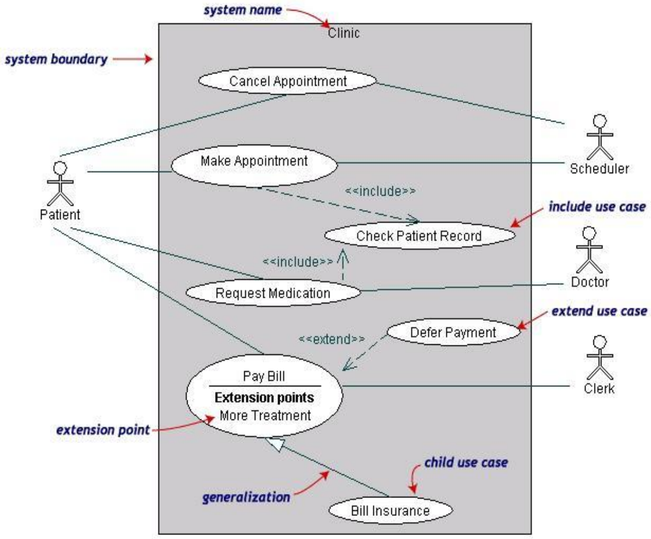


- <<extends>> relationships represent exceptional or seldom invoked cases.
- The exceptional event flows are factored out of the main event flow for clarity.
- Use cases representing exceptional flows can extend more than one use case.
- The direction of a <<extends>> relationship is to the extended use case

63

Use-Case Diagrams

- Both **Make Appointment** and **Request Medication** include **Check Patient Record** as a subtask (include)
- The **extension point** is written inside the base case **Pay bill**; the extending class **Defer payment** adds the behavior of this extension point. (extend)
- **Pay Bill** is a parent use case and **Bill Insurance** is the child use case. (generalization)



64

Sequence diagram

- UML 2 Sequence diagrams models the collaboration of objects based on a time sequence. It shows how the objects interact with others in a particular scenario of a use case.
- Sequence diagrams represent the behavior of a system as messages ("interactions") between different objects
- Depict object interactions in a given scenario identified for a given Use Case
- Specify the messages passed (method call) between objects using horizontal arrows including messages to/from external actors
- Show time sequences that are not easily depicted in other diagrams. Emphasis on time ordering
- Time increases from Top to bottom

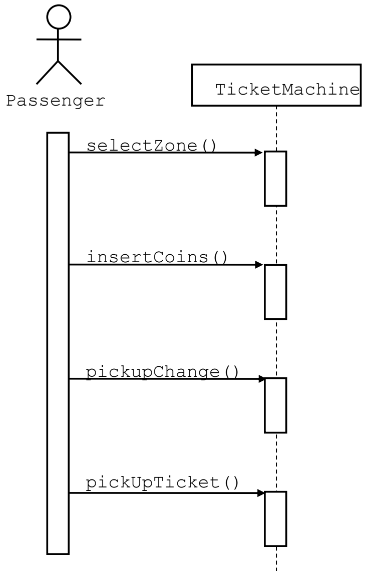
65

Purpose of Sequence Diagram

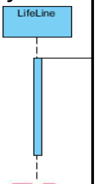
- Model high-level interaction between active objects in a system
- Model the interaction between object instances within a collaboration that realizes a use case
- Model the interaction between objects within a collaboration that realizes an operation
- Either model generic interactions (showing all possible paths through the interaction) or specific instances of a interaction (showing just one path through the interaction)

66

UML sequence diagrams

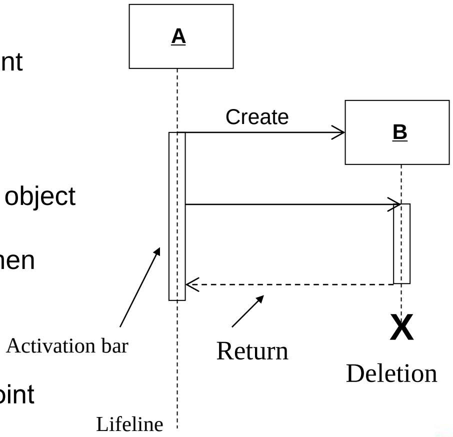


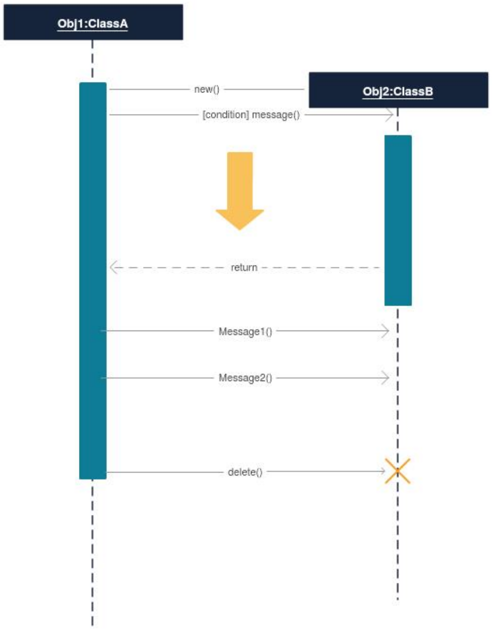
- Used during requirements analysis
  - to refine use case descriptions
  - to find additional objects ("participating objects")
- Used during system design
  - to refine subsystem interfaces
- ***Classes*** are represented by columns
- ***Messages*** are represented by arrows
- ***Activations*** are represented by narrow rectangles
- ***Lifelines*** are represented by dashed lines



Sequence Diagrams – Object Life Spans

- Creation
  - Create message
  - Object life starts at that point
- Activation
  - Symbolized by rectangular stripes
  - Place on the lifeline where object is activated.
  - Rectangle also denotes when object is deactivated.
- Deletion
  - Placing an 'X' on lifeline
  - Object's life ends at that point





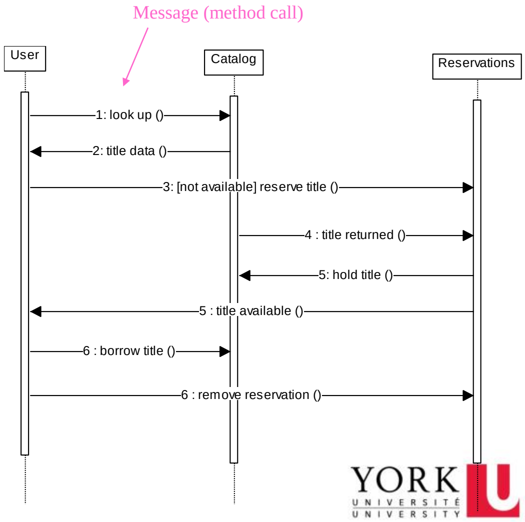
Sequence Diagram

•Sequence diagrams demonstrate the behavior of objects in a use case by describing the objects and the messages they pass.

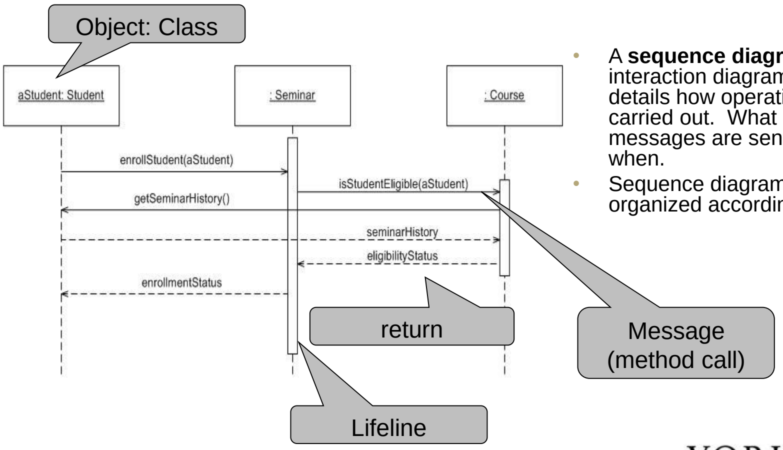
•The horizontal dimension shows the objects participating in the interaction.

•The vertical arrangement of messages indicates their order.

•The labels may contain the seq. # to indicate concurrency.

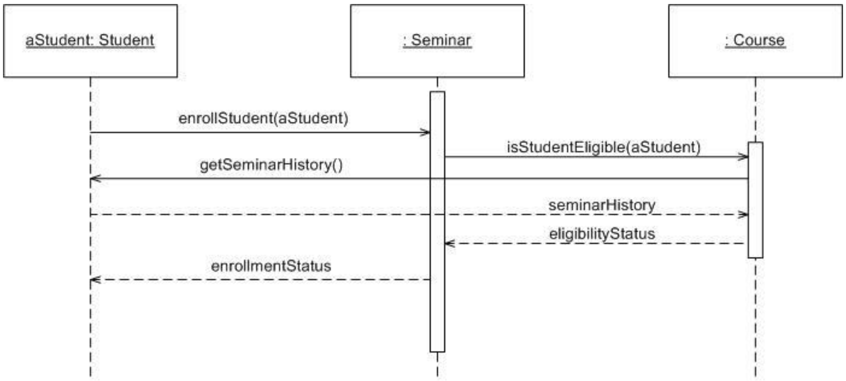


Sequence Diagram



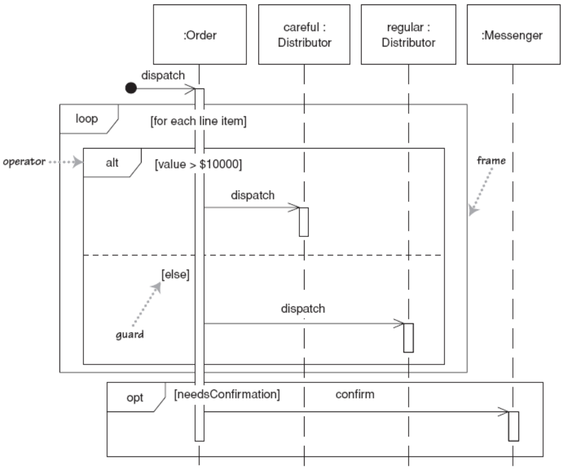
- A **sequence diagram** is an interaction diagram that details how operations are carried out. What messages are sent and when.
- Sequence diagrams are organized according to time

Sequence diagram

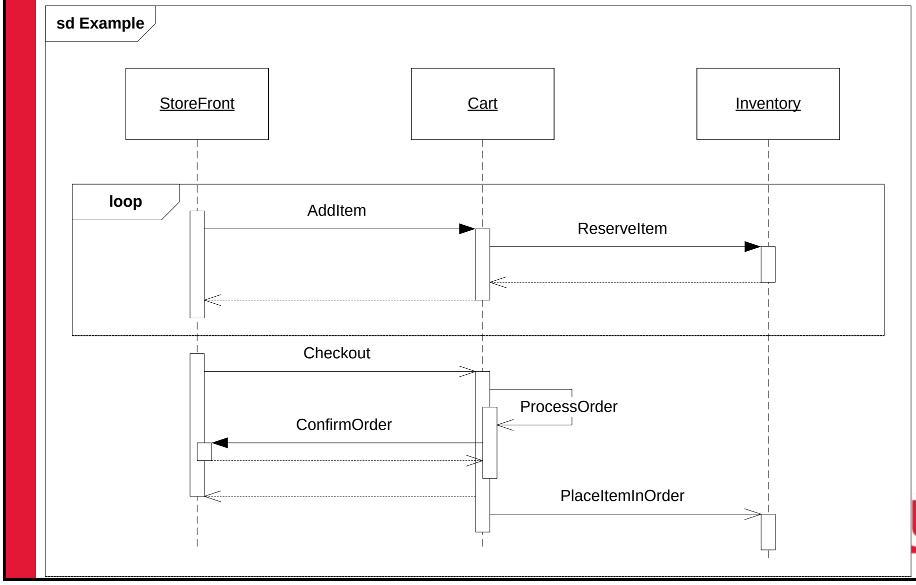


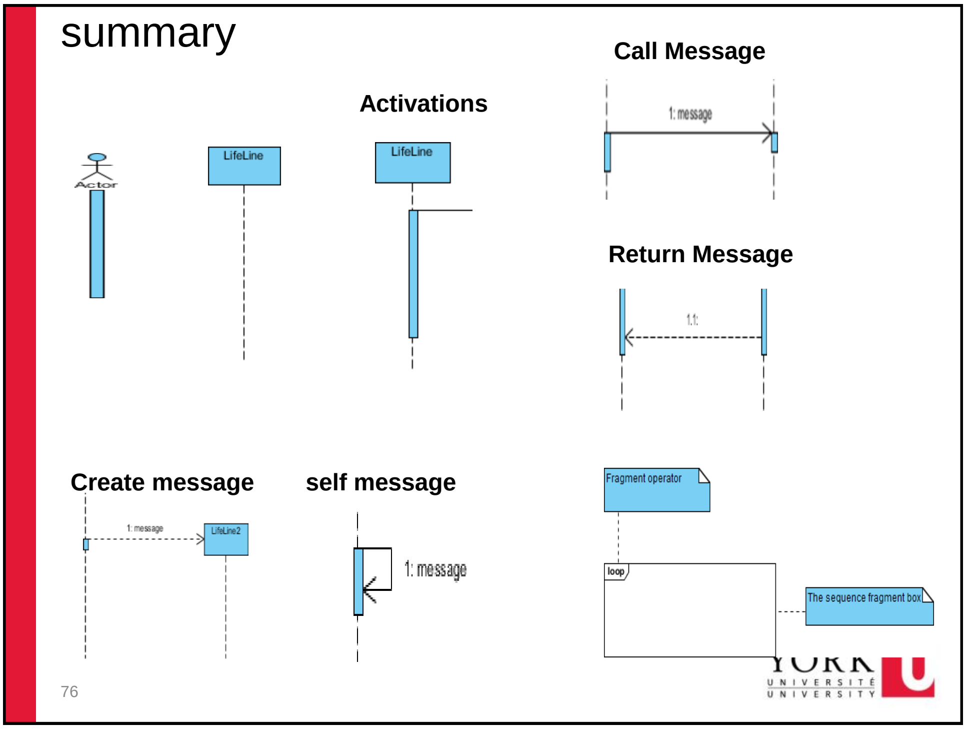
Indicating selection and loops

- frame: box around part of a sequence diagram to indicate selection or loop
  - `if` → (opt) [condition]
  - `if/else` → (alt) [condition], separated by horizontal dashed line
  - loop → (loop) [condition or items to loop over]



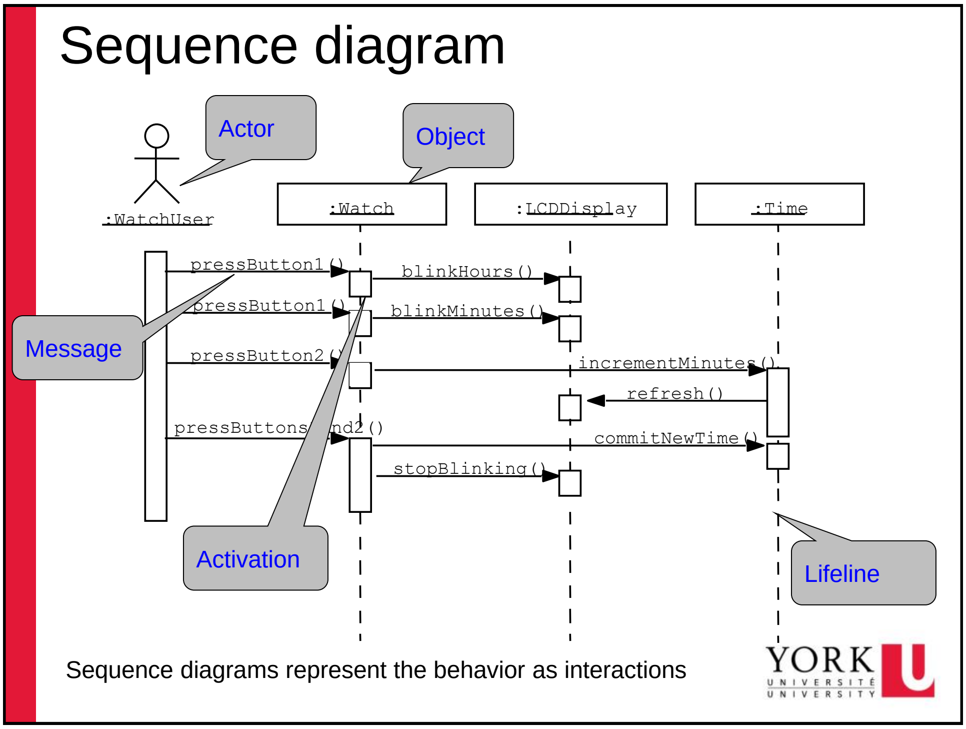
Example sequence diagram





summary

76



Sequence diagram

Sequence diagrams represent the behavior as interactions

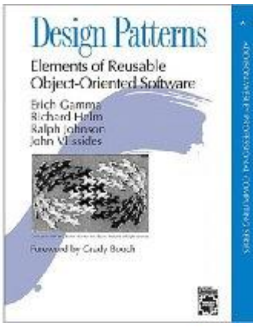
77

“Gang of Four” (GoF) Book

- Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Publishing Company, 1994

- Written by this "gang of four"
  - Dr. Erich Gamma, then Software Engineer, Taligent, Inc.
  - Dr. Richard Helm, then Senior Technology Consultant, DMR Group
  - Dr. Ralph Johnson, then and now at University of Illinois, Computer Science Department
  - Dr. John Vlissides, then a researcher at IBM
    Thomas J. Watson Research Center

    See John's WikiWiki tribute page
    http://c2.com/cgi/wiki?JohnVlissides



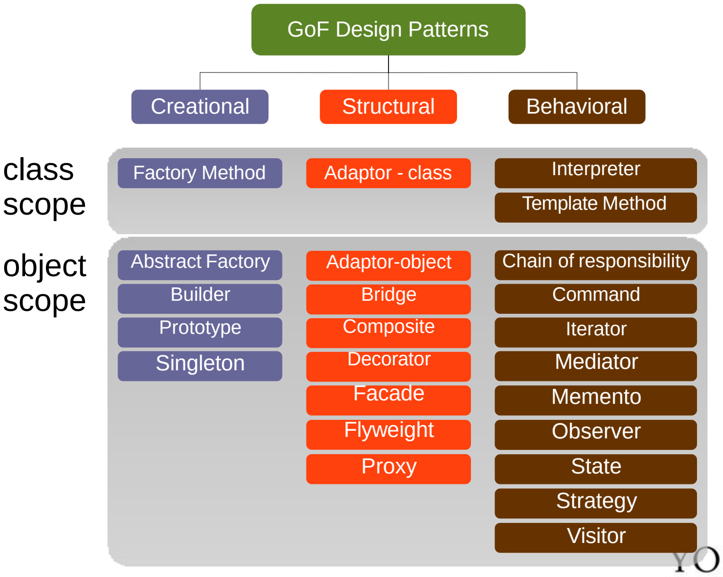
What is design patten and Why

- Design pattern is a general reusable solution to a commonly occurring problem within a given context in software design.

- Christopher Alexander says each pattern is a three-part rule which expresses a relation between a certain context, a problem, and a solution.
- Design patterns represent solutions to problems that arise when developing software within a particular context.
  - i.e Patterns = problems~solution pairs in a context

- A design pattern is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations. Patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

Design Patterns Classification



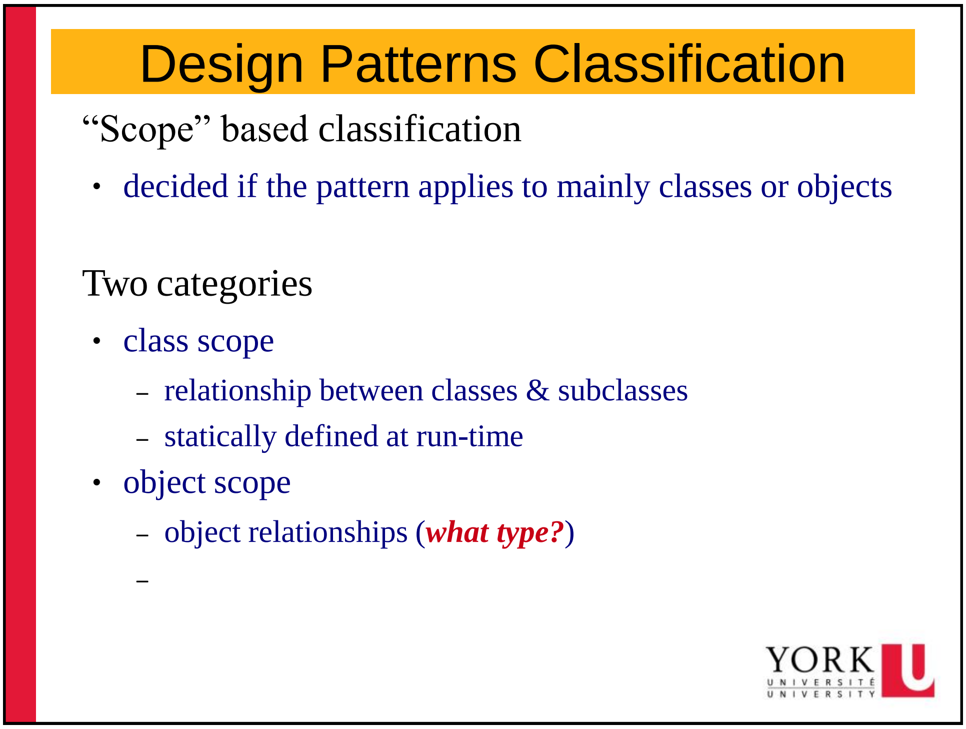
GoF Design Patterns

	Creational	Structural	Behavioral
class scope	Factory Method	Adaptor - class	Interpreter Template Method
object scope	Abstract Factory Builder Prototype Singleton	Adaptor-object Bridge Composite Decorator Facade Flyweight Proxy	Chain of responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Design Patterns Classification

"Purpose" based classification

- creational:
  - concerns with **creation process** of objects & classes
- structural
  - **composition** of classes & objects
- behavioral
  - characterizes **interaction & responsibility** of objects & classes



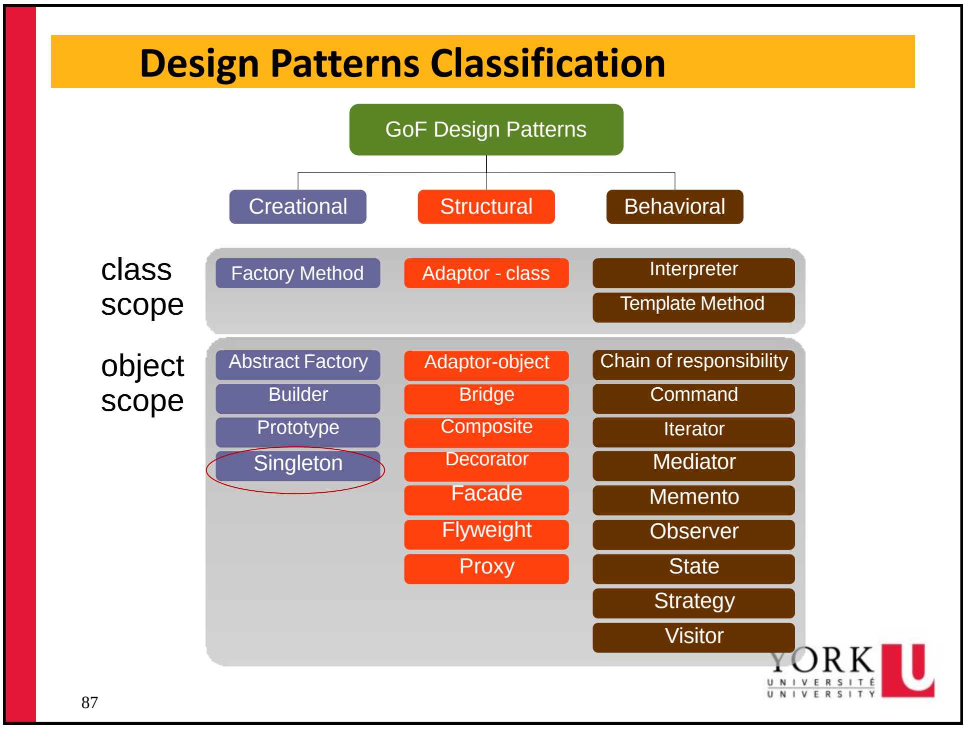
Design Patterns Classification

"Scope" based classification

- decided if the pattern applies to mainly classes or objects

Two categories

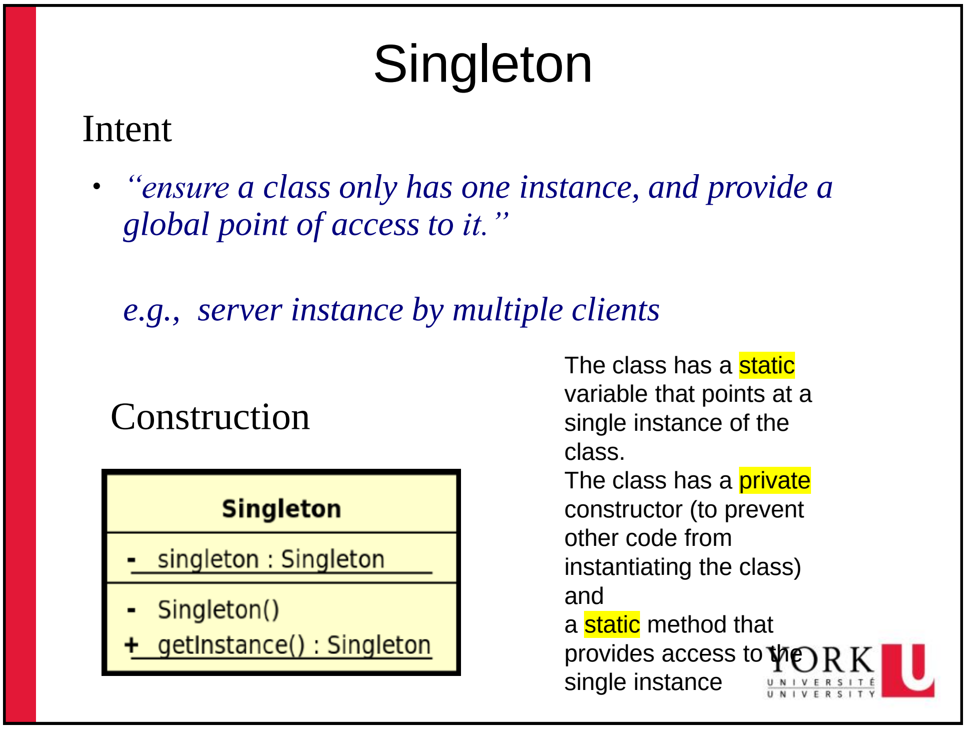
- class scope
  - relationship between classes & subclasses
  - statically defined at run-time
- object scope
  - object relationships (***what type?***)
  -



Design Patterns Classification

GoF Design Patterns

	Creational	Structural	Behavioral
class scope	Factory Method	Adaptor - class	Interpreter Template Method
object scope	Abstract Factory Builder Prototype Singleton	Adaptor-object Bridge Composite Decorator Facade Flyweight Proxy	Chain of responsibility Command Iterator Mediator Memento Observer State Strategy Visitor



Singleton

Intent

- *"ensure a class only has one instance, and provide a global point of access to it."*

  e.g., server instance by multiple clients

Construction

Singleton

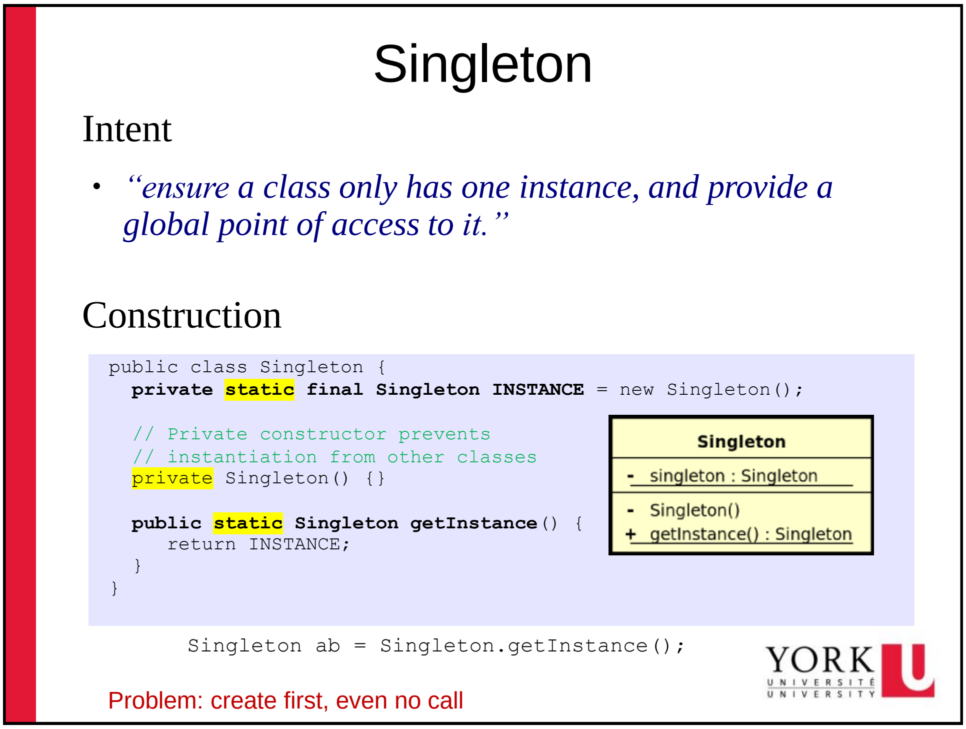
- singleton : Singleton

- Singleton()

+ getInstance() : Singleton

The class has a static variable that points at a single instance of the class.
The class has a private constructor (to prevent other code from instantiating the class) and
a static method that provides access to the single instance

88



Singleton

Intent

- *"ensure a class only has one instance, and provide a global point of access to it."*

Construction

```
public class Singleton {
  private static final Singleton INSTANCE = new Singleton();

  // Private constructor prevents
  // instantiation from other classes
  private Singleton() {}

  public static Singleton getInstance() {
     return INSTANCE;
  }
}
```

Singleton

- singleton : Singleton

- Singleton()

+ getInstance() : Singleton

```
Singleton ab = Singleton.getInstance();
```

Problem: create first, even no call

89

Singleton

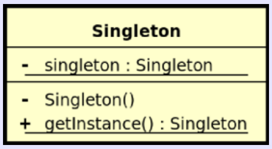
Intent

- *“ensure a class only has one instance, and provide a global point of access to it.”*

Construction

“Lazy initialization”

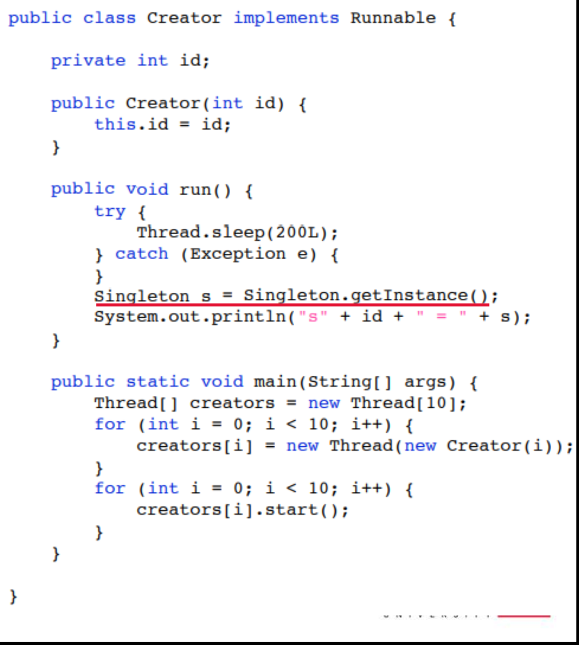
```
public class Singleton {
  private static final Singleton INSTANCE;

  // Private constructor prevents
  // instantiation from other classes
  private Singleton() {}

  public static Singleton getInstance() {
     if (INSTANCE == null)
        INSTANCE = new Singleton();
     return INSTANCE;
  }
}
```



```
Singleton ab = Singleton.getInstance();
```

Problem: multiple client may request at the same time, creating more than one instance

- The Java code just shown is not thread safe
- This means that it is possible for two threads to attempt to create the singleton for the first time simultaneously
- If both threads check to see if the static variable is empty at the same time, they will both proceed to creating an instance and you will end up with two instances of the singleton object (not good!)



```
public class Creator implements Runnable {

    private int id;

    public Creator(int id) {
        this.id = id;
    }

    public void run() {
        try {
            Thread.sleep(200L);
        } catch (Exception e) {
        }
        Singleton s = Singleton.getInstance();
        System.out.println("s" + id + " = " + s);
    }

    public static void main(String[] args) {
        Thread[] creators = new Thread[10];
        for (int i = 0; i < 10; i++) {
            creators[i] = new Thread(new Creator(i));
        }
        for (int i = 0; i < 10; i++) {
            creators[i].start();
        }
    }

}
```

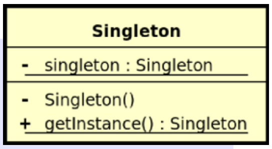
Singleton

Intent

- *“ensure a class only has one instance, and provide a global point of access to it.”*

Construction

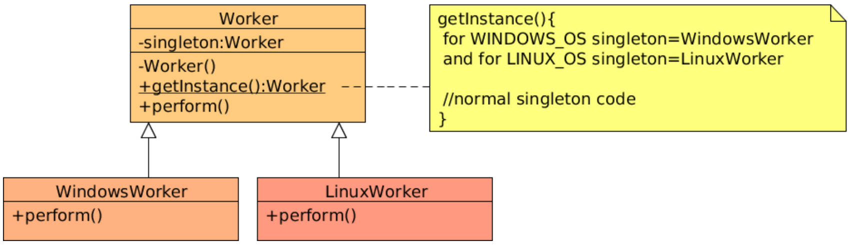
```
public class Singleton {
  private static final Singleton INSTANCE

  // Private constructor prevents
  // instantiation from other classes
  private Singleton() {}

  public static synchronized Singleton getInstance() {
     if (INSTANCE == null)
        INSTANCE = new Singleton();
     return INSTANCE;
  }
}
      Singleton ab = Singleton.getInstance();
```



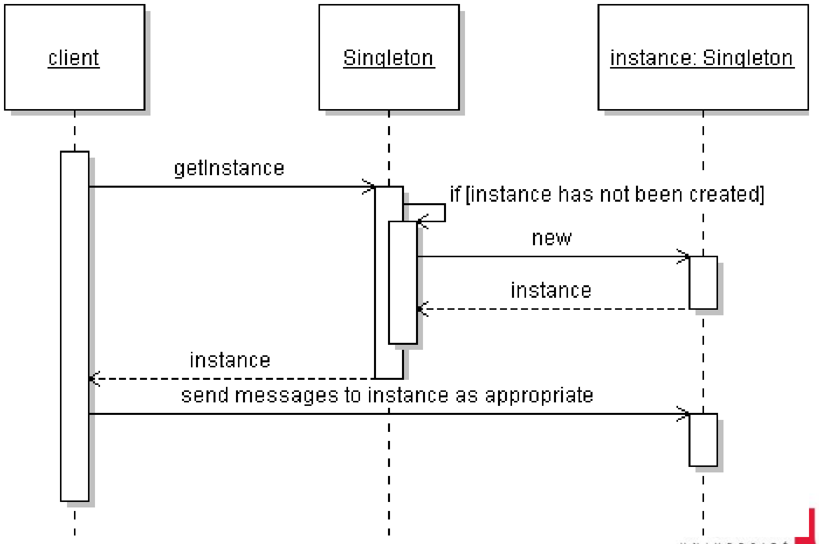
Singleton

Advantages

- controlled access to the class instance(s)
  - can dictate who, and when a client can access
- refinement of functionality
  - via inheritance/subclass



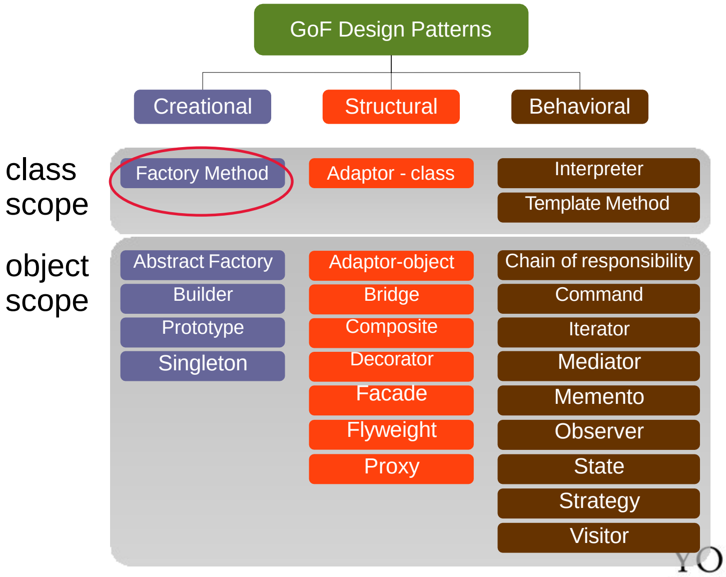
Singleton sequence diagram


client
Singleton
instance: Singleton
getInstance
if [instance has not been created]
new
instance
instance
send messages to instance as appropriate

94

94

Design Patterns Classification



	Creational	Structural	Behavioral
class scope	Factory Method	Adaptor - class	Interpreter Template Method
object scope	Abstract Factory Builder Prototype Singleton	Adaptor-object Bridge Composite Decorator Facade Flyweight Proxy	Chain of responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

95

95

Factory Method Design Pattern

- Intent: “encapsulate the instantiation of concrete types.”
- When constructing objects from classes, we use the “constructor" of the corresponding class. However, there are cases where we do not want the client code to know what kind of objects will be built, or, don’t want them to have the burden of the varying class selection criteria.
- The design pattern is designed to allow us to define an interface (in this example the interface is the *FactoryMethod* method), in a class (in the example is the Creator class) that can be used to construct objects. (However, what kind of objects will ultimately be constructed is defined by the type of classes that will be applied to the *FactoryMethod* Interface.)
- By encapsulating the functionality required to select and instantiate an appropriate class, application objects can make use of the factory method to get access to the appropriate class instance,. When there are several sub-classes, this eliminates the need for an application to deal with the varying class selection criteria

Structural Elements of the Factory Method Design Pattern

The classes that are used in this Design Pattern are:

The Class **Product**

Specifies the abstract class or the interface of the objects that can be manufactured by FactoryMethod

The Class **ConcreteProduct**

Implements the interface defined by the class Product

The Class **Creator** // factory

Defines the *FactoryMethod* Interface, which constructs and returns a Product item. The Creator class can define a default implementation that returns a particular object type (eg ConcreteProduct), and invokes this default implementation of the FactoryMethod

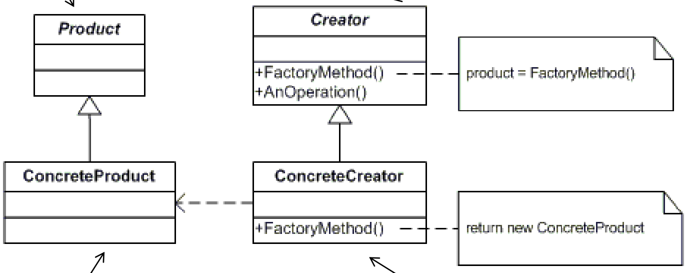
The Class **ConcreteCreator** // concrete factory

It is a sub-class of the Creator class and overrides the *FactoryMethod* method in order for FactoryMethod to construct and return an object (eg, ConcreteProduct) for which the client code does not know its type (simply knows that the object which was manufactured is Product type)

Factory Method – Class Diagram

The top class for Product(s)

The top class for Creator(s)



Specific ConcreteProduct(s) which can be created by the corresponding ConcreteCreator(s) using the FactoryMethod() method of the corresponding ConcreteCreator

Specific ConcreteCreator(s) e.g. ConcreteCreatorA, ConcreteCreatorB classes

Factory Method - Example

Or interface

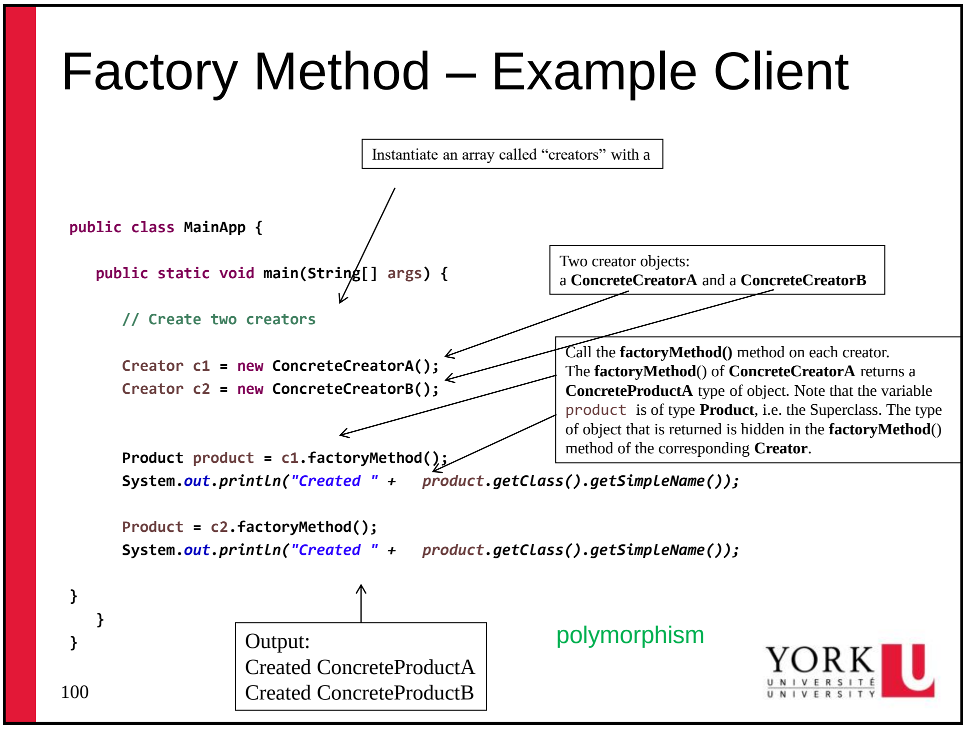
```
public abstract class Product {
  …
}

// "Concrete Product A"
public class ConcreteProductA extends Product
{
  …
}
//"Concrete Product B"
public class ConcreteProductB extends Product
{
  …
}
```

```
// “Factory”
public abstract class Creator {
  public abstract Product factoryMethod();
}
```

ConcreteCreatorA/B which creates a ConcreteProductA/B through its **factoryMethod()** method

```
// "Concrete Creator A"
public class ConcreteCreatorA
          extends Creator {
    @Override
    public Product factoryMethod() {
        return new ConcreteProductA();
    }
}
```

```
//"Concrete Creator B"
public class ConcreteCreatorB
          extends Creator {
    @Override
    public Product factoryMethod() {
        return new ConcreteProductB();
    }
}
```



Factory Method – Example Client

Instantiate an array called “creators” with a

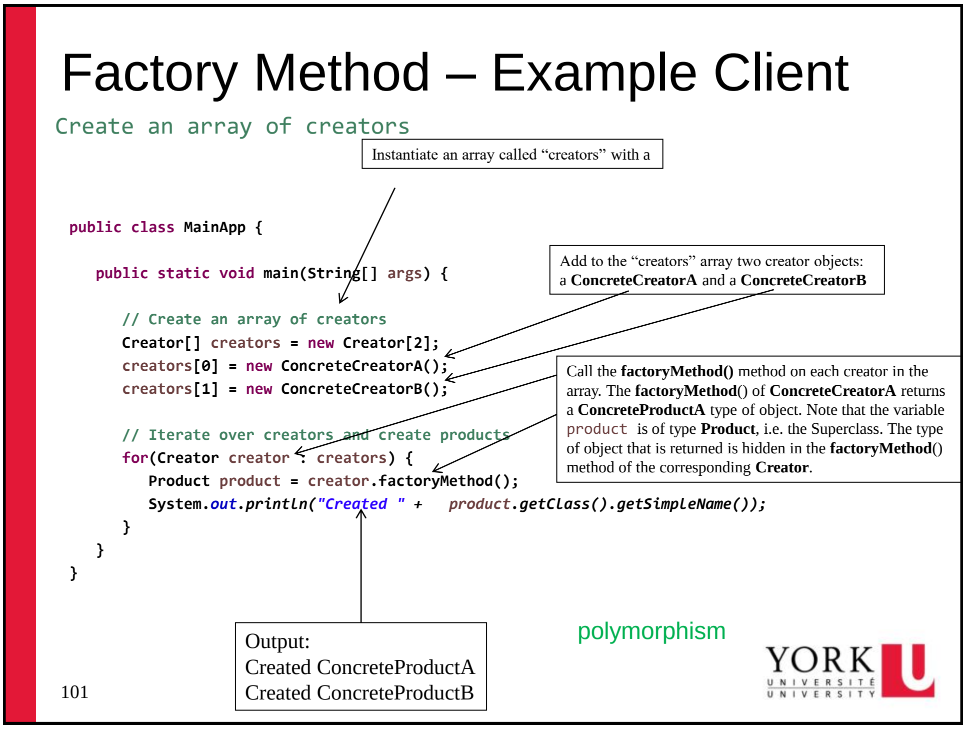
```
public class MainApp {

   public static void main(String[] args) {

      // Create two creators

      Creator c1 = new ConcreteCreatorA();
      Creator c2 = new ConcreteCreatorB();


      Product product = c1.factoryMethod();
      System.out.println("Created " +   product.getClass().getSimpleName());

      Product = c2.factoryMethod();
      System.out.println("Created " +   product.getClass().getSimpleName());

}
   }
}
```

Two creator objects:
a **ConcreteCreatorA** and a **ConcreteCreatorB**

Call the **factoryMethod()** method on each creator. The **factoryMethod**() of **ConcreteCreatorA** returns a **ConcreteProductA** type of object. Note that the variable `product` is of type **Product**, i.e. the Superclass. The type of object that is returned is hidden in the **factoryMethod**() method of the corresponding **Creator**.

Output:
Created ConcreteProductA
Created ConcreteProductB

polymorphism



Factory Method – Example Client

Create an array of creators

Instantiate an array called “creators” with a

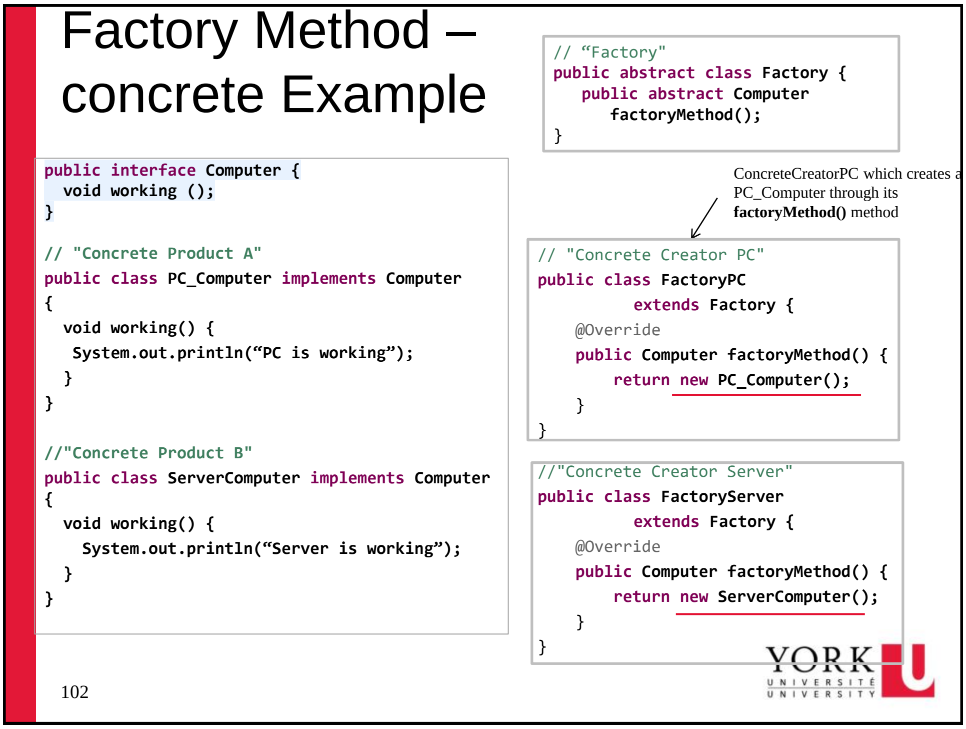
```
public class MainApp {

   public static void main(String[] args) {

      // Create an array of creators
      Creator[] creators = new Creator[2];
      creators[0] = new ConcreteCreatorA();
      creators[1] = new ConcreteCreatorB();

      // Iterate over creators and create products
      for(Creator creator : creators) {
         Product product = creator.factoryMethod();
         System.out.println("Created " +   product.getClass().getSimpleName());
      }
   }
}
```

Add to the “creators” array two creator objects:
a **ConcreteCreatorA** and a **ConcreteCreatorB**

Call the **factoryMethod()** method on each creator in the array. The **factoryMethod**() of **ConcreteCreatorA** returns a **ConcreteProductA** type of object. Note that the variable `product` is of type **Product**, i.e. the Superclass. The type of object that is returned is hidden in the **factoryMethod**() method of the corresponding **Creator**.

Output:
Created ConcreteProductA
Created ConcreteProductB

polymorphism



Factory Method – concrete Example

```
public interface Computer {
  void working ();
}

// "Concrete Product A"
public class PC_Computer implements Computer
{
  void working() {
   System.out.println(“PC is working”);
  }
}

//"Concrete Product B"
public class ServerComputer implements Computer
{
  void working() {
    System.out.println(“Server is working”);
  }
}
```

```
// “Factory”
public abstract class Factory {
   public abstract Computer
      factoryMethod();
}
```

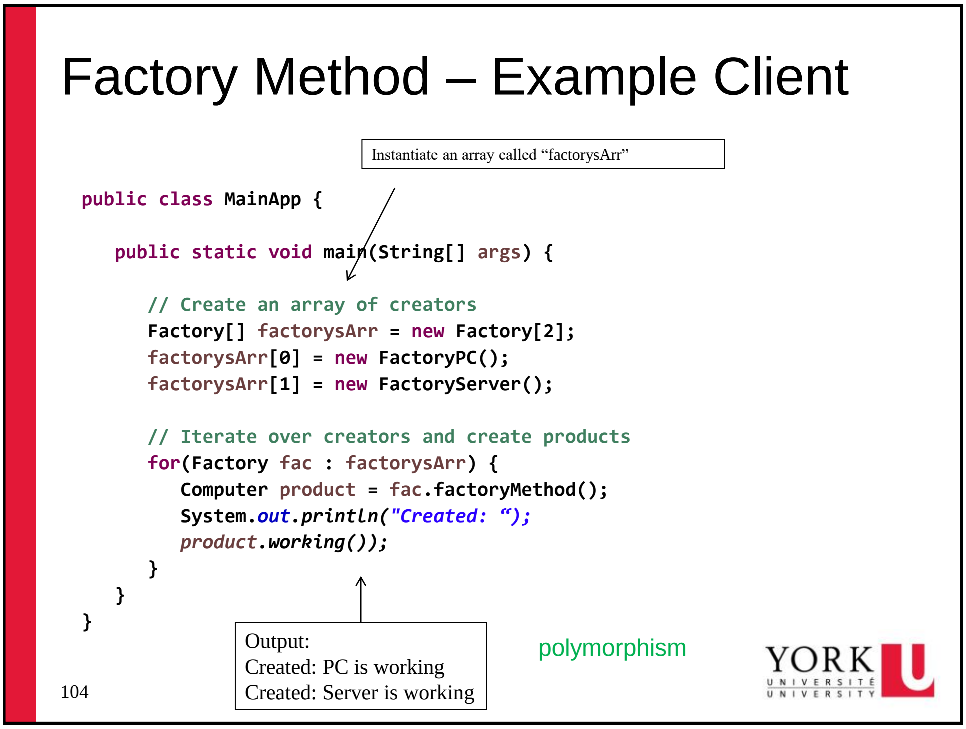
ConcreteCreatorPC which creates a PC_Computer through its **factoryMethod()** method

```
// "Concrete Creator PC"
public class FactoryPC
          extends Factory {
    @Override
    public Computer factoryMethod() {
        return new PC_Computer();
    }
}
```

```
//"Concrete Creator Server"
public class FactoryServer
          extends Factory {
    @Override
    public Computer factoryMethod() {
        return new ServerComputer();
    }
}
```

102

102



Factory Method – Example Client

Instantiate an array called “factorysArr”

```
public class MainApp {

   public static void main(String[] args) {

      // Create an array of creators
      Factory[] factorysArr = new Factory[2];
      factorysArr[0] = new FactoryPC();
      factorysArr[1] = new FactoryServer();

      // Iterate over creators and create products
      for(Factory fac : factorysArr) {
         Computer product = fac.factoryMethod();
         System.out.println("Created: “);
         product.working());
      }
   }
}
```

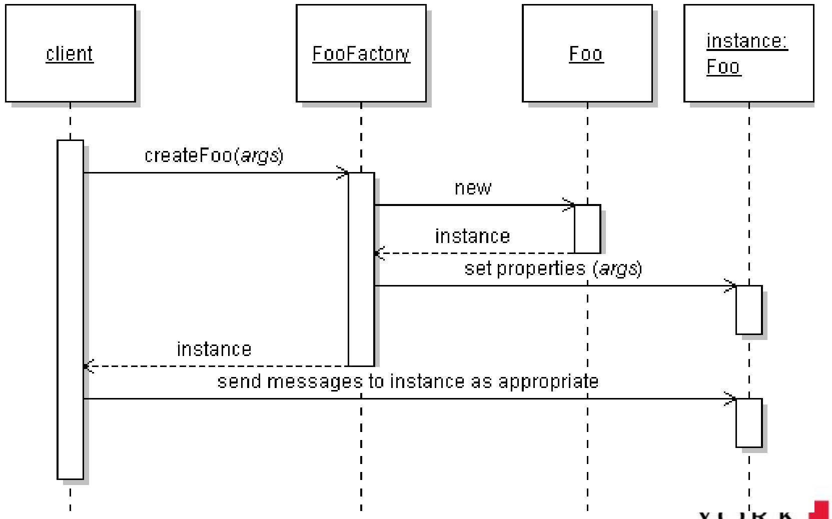
```
Output:
Created: PC is working
Created: Server is working
```

polymorphism

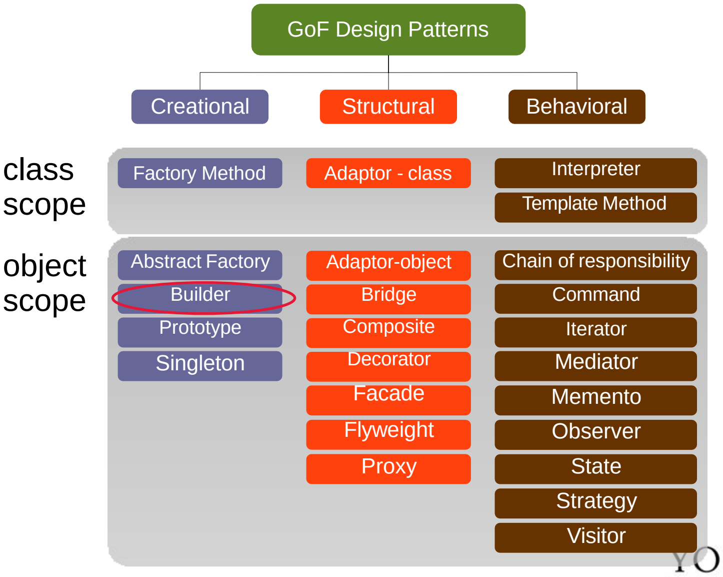
104

104

Factory sequence diagram



Design Patterns Classification



GoF Design Patterns

Creational

Structural

Behavioral

class scope

Factory Method

Adaptor - class

Interpreter

Template Method

object scope

Abstract Factory

Builder

Prototype

Singleton

Adaptor-object

Bridge

Composite

Decorator

Facade

Flyweight

Proxy

Chain of responsibility

Command

Iterator

Mediator

Memento

Observer

State

Strategy

Visitor

Builder pattern

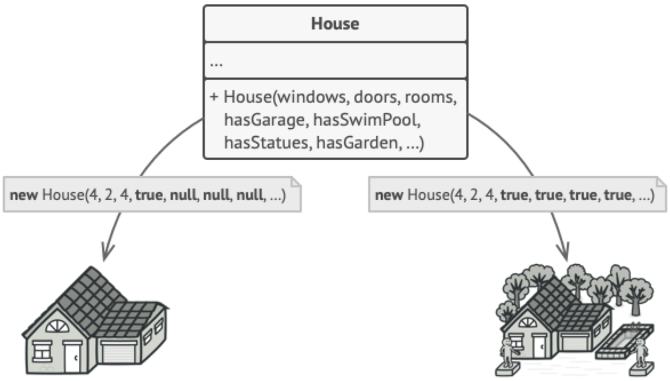
Intent

Separate the construction of a complex object from its representation so that the same construction process can create different representations

Think of building a house (complicated)

There are different components, and different types of house – simple vs luxurious

108


House
…
+ House(windows, doors, rooms, hasGarage, hasSwimPool, hasStatues, hasGarden, ...)
new House(4, 2, 4, true, null, null, null, ...)
new House(4, 2, 4, true, true, true, true, ...)

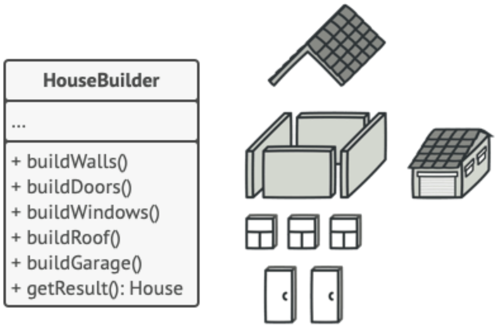
Other solution: basic constructor, other as setters
House (window, door, room);

```
h = new House (window, door, roof);  // basic
h.setGarden()
h.setSwimmingPool()
h.setGartaghe
…
```

109 Problem: ***h*** Maybe in inconsistent states

109

The Builder pattern suggests that you extract the object construction code out of its own class and move it to separate objects called *builders*.

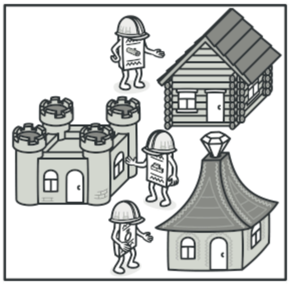


HouseBuilder

...

+ buildWalls()
+ buildDoors()
+ buildWindows()
+ buildRoof()
+ buildGarage()
+ getResult(): House

Different builders

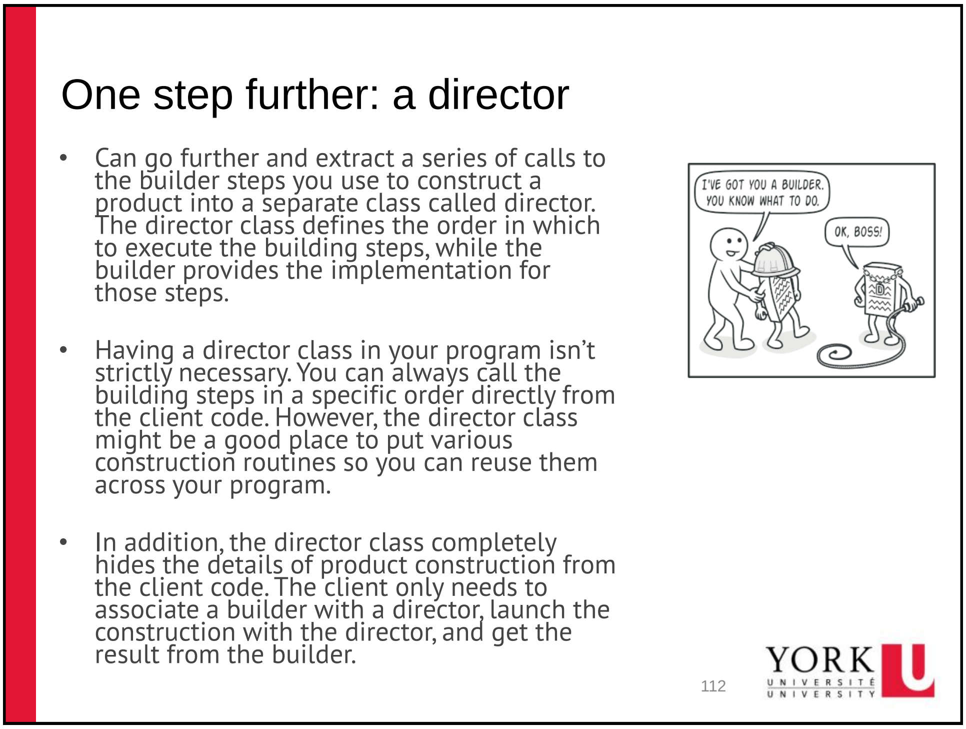


Builders can build differently, and also different components

The important part is that you don't need to call all of the steps. **You can call only those steps that are necessary for producing a particular configuration of an object.**

Some of the construction steps might require different implementation when you need to build various representations of the product. For example, walls of a cabin may be built of wood, but the castle walls must be built with stone.

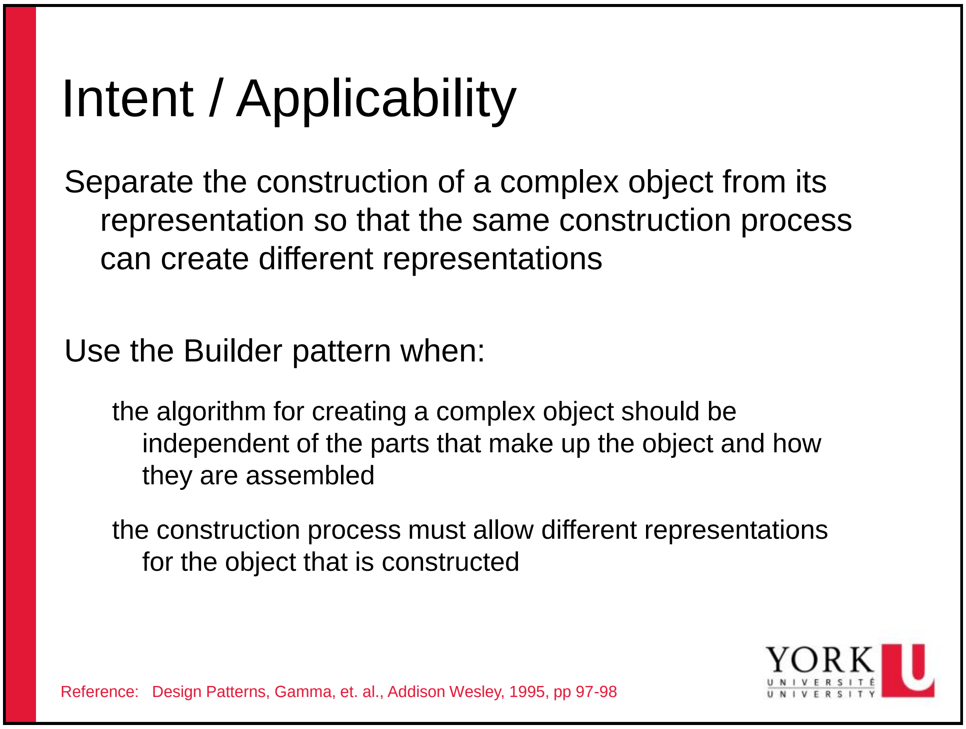
In this case, you can create several different builder classes that implement the same set of building steps, but in a different manner. Then you can use these builders in the construction process (i.e., an ordered set of calls to the building steps) to produce different kinds of objects.



I'VE GOT YOU A BUILDER. YOU KNOW WHAT TO DO.
OK, BOSS!
D

One step further: a director

- Can go further and extract a series of calls to the builder steps you use to construct a product into a separate class called director. The director class defines the order in which to execute the building steps, while the builder provides the implementation for those steps.
- Having a director class in your program isn't strictly necessary. You can always call the building steps in a specific order directly from the client code. However, the director class might be a good place to put various construction routines so you can reuse them across your program.
- In addition, the director class completely hides the details of product construction from the client code. The client only needs to associate a builder with a director, launch the construction with the director, and get the result from the builder.



Intent / Applicability

Separate the construction of a complex object from its representation so that the same construction process can create different representations

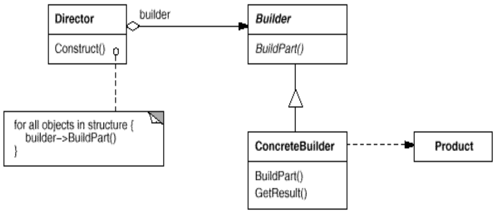
Use the Builder pattern when:

the algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled

the construction process must allow different representations for the object that is constructed

Reference: Design Patterns, Gamma, et. al., Addison Wesley, 1995, pp 97-98

Builder: Participants



Product

Represents the complex object under construction

Includes classes that define the constituent parts

Gives interfaces for assembling the parts

Builder

Specifies an abstract interface for creating parts of a Product object

ConcreteBuilder

Constructs and assembles parts of the product by implementing the Builder interface

Director

Constructs an object using the Builder interface

115

Builder: Collaborations

Client creates **Director** object and configures it with a **Builder**

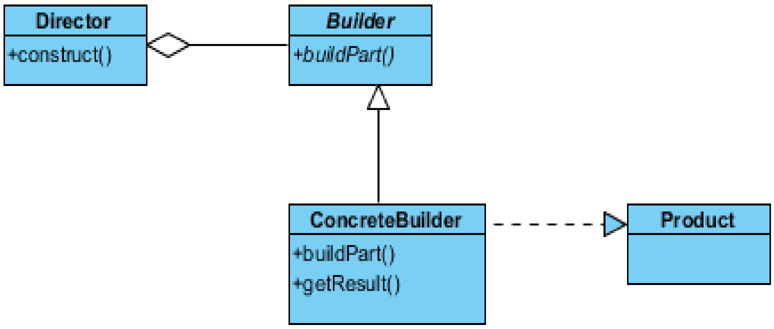
Director notifies **Builder** to build each part of the product

Builder handles requests from **Director** and adds parts to the product

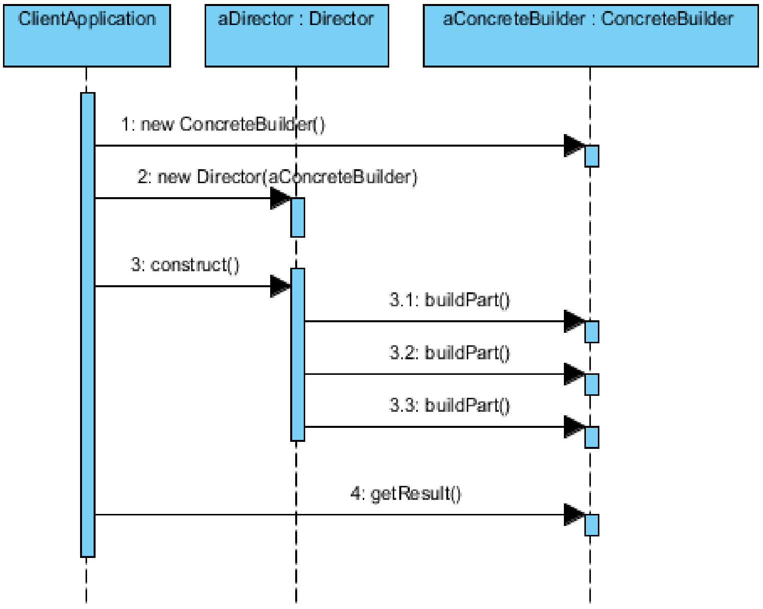
Client retrieves product from the **Director**/**Builder**

116

UML Structure



Collaborations



Example: building different types of airplanes

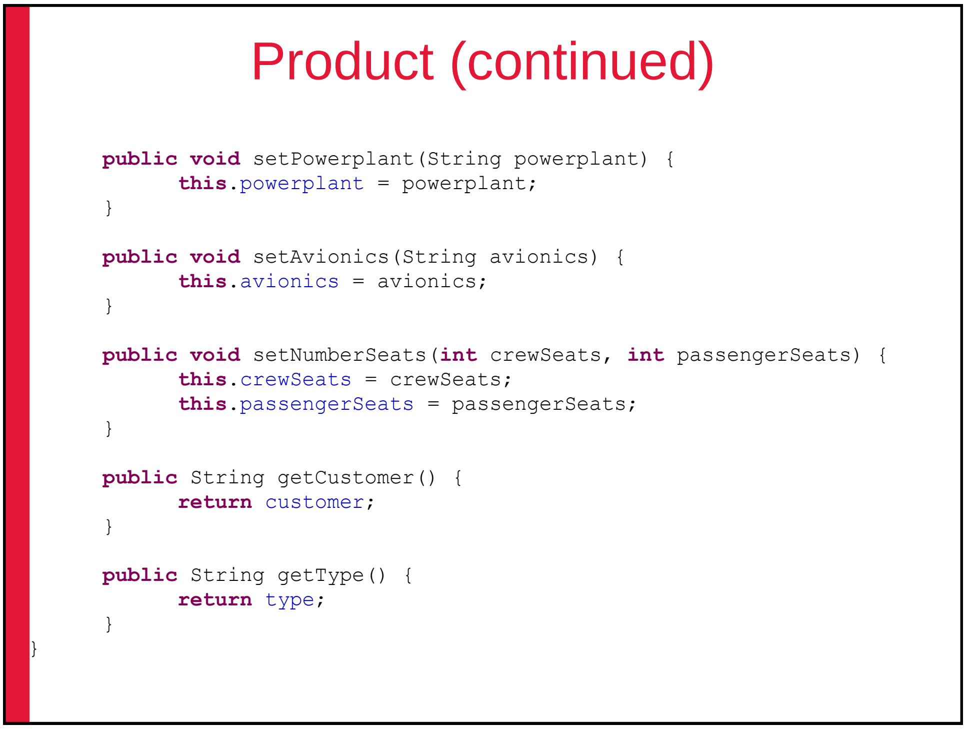
- **Airplane**: product
- **AirplaneBuilder**: abstract builder
- Some concrete builders:
  - **CropDuster**
  - **FighterJet**
  - **Glider**
  - **Airliner**
- **AerospaceEngineer**: director

119

Product

```
package builder;
/** "Product" */
public class Airplane {

      private String type;
      private float wingspan;
      private String powerplant;
      private int crewSeats;
      private int passengerSeats;
      private String avionics;
      private String customer;

      Airplane (String customer, String type){
            this.customer = customer;
            this.type = type;
      }

      public void setWingspan(float wingspan) {
            this.wingspan = wingspan;
      }
```

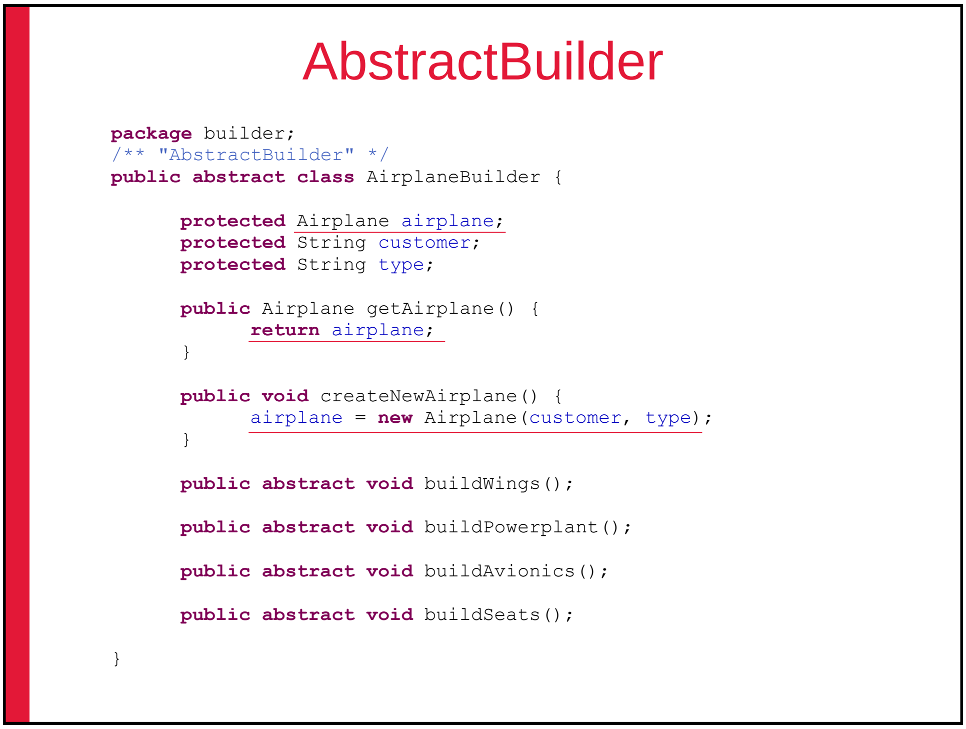
120



Product (continued)

```java
    public void setPowerplant(String powerplant) {
        this.powerplant = powerplant;
    }

    public void setAvionics(String avionics) {
        this.avionics = avionics;
    }

    public void setNumberSeats(int crewSeats, int passengerSeats) {
        this.crewSeats = crewSeats;
        this.passengerSeats = passengerSeats;
    }

    public String getCustomer() {
        return customer;
    }

    public String getType() {
        return type;
    }
}
```

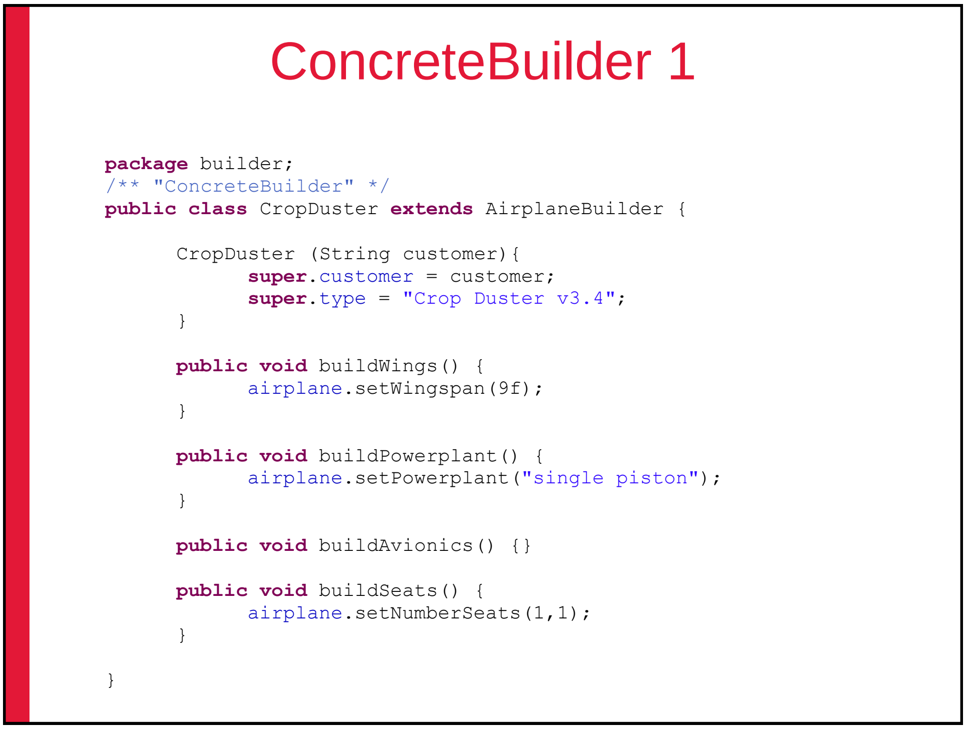
121



AbstractBuilder

```java
package builder;
/** "AbstractBuilder" */
public abstract class AirplaneBuilder {

    protected Airplane airplane;
    protected String customer;
    protected String type;

    public Airplane getAirplane() {
        return airplane;
    }

    public void createNewAirplane() {
        airplane = new Airplane(customer, type);
    }

    public abstract void buildWings();

    public abstract void buildPowerplant();

    public abstract void buildAvionics();

    public abstract void buildSeats();

}
```

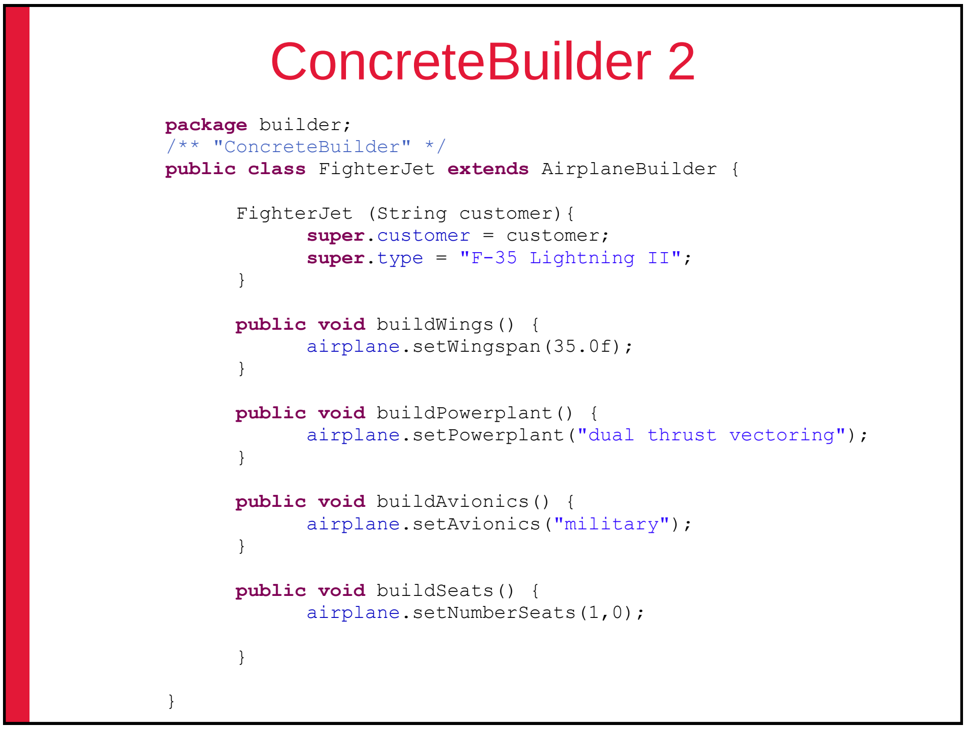
122



ConcreteBuilder 1

```
package builder;
/** "ConcreteBuilder" */
public class CropDuster extends AirplaneBuilder {

      CropDuster (String customer){
            super.customer = customer;
            super.type = "Crop Duster v3.4";
      }

      public void buildWings() {
            airplane.setWingspan(9f);
      }

      public void buildPowerplant() {
            airplane.setPowerplant("single piston");
      }

      public void buildAvionics() {}

      public void buildSeats() {
            airplane.setNumberSeats(1,1);
      }

}
```

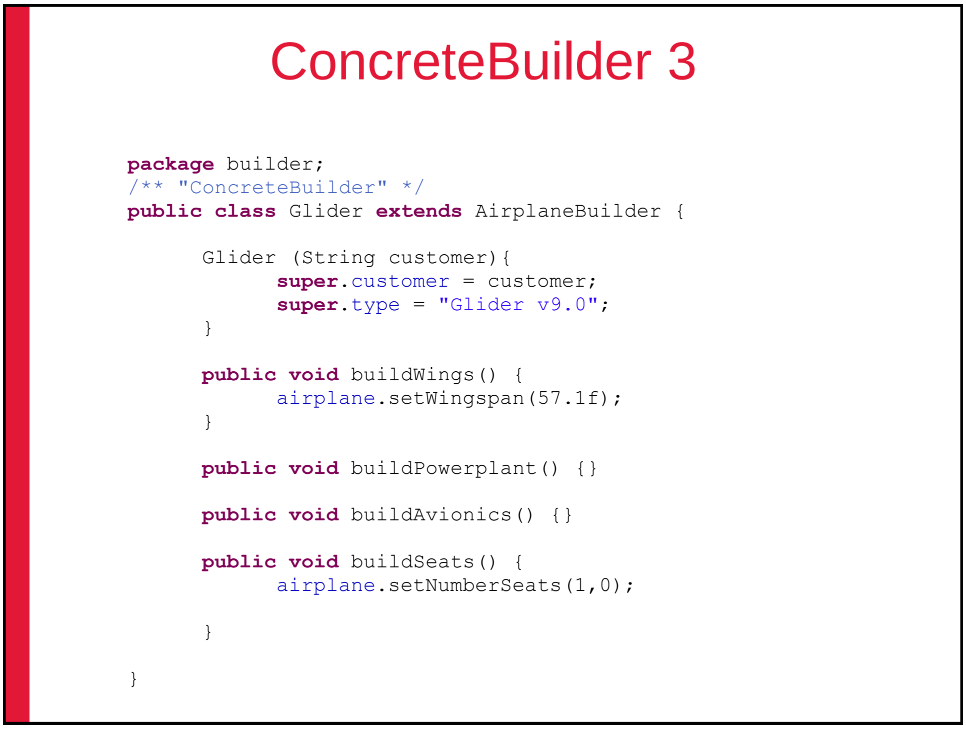
123



ConcreteBuilder 2

```
package builder;
/** "ConcreteBuilder" */
public class FighterJet extends AirplaneBuilder {

      FighterJet (String customer){
            super.customer = customer;
            super.type = "F-35 Lightning II";
      }

      public void buildWings() {
            airplane.setWingspan(35.0f);
      }

      public void buildPowerplant() {
            airplane.setPowerplant("dual thrust vectoring");
      }

      public void buildAvionics() {
            airplane.setAvionics("military");
      }

      public void buildSeats() {
            airplane.setNumberSeats(1,0);

      }

}
```

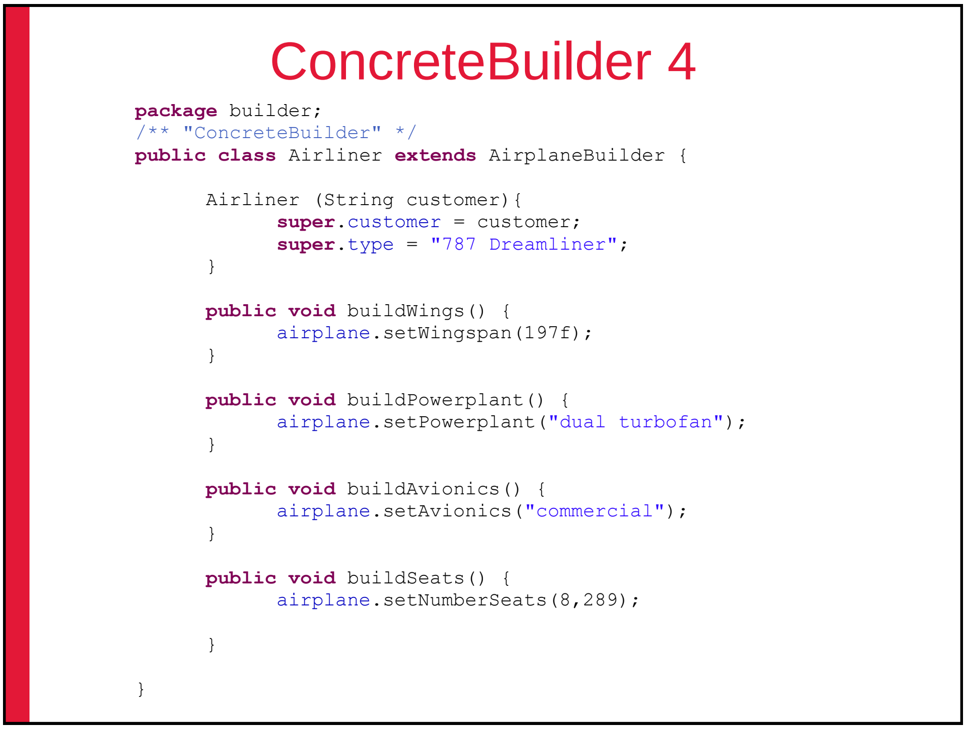
124



ConcreteBuilder 3

```
package builder;
/** "ConcreteBuilder" */
public class Glider extends AirplaneBuilder {

        Glider (String customer){
                super.customer = customer;
                super.type = "Glider v9.0";
        }

        public void buildWings() {
                airplane.setWingspan(57.1f);
        }

        public void buildPowerplant() {}

        public void buildAvionics() {}

        public void buildSeats() {
                airplane.setNumberSeats(1,0);

        }

}
```

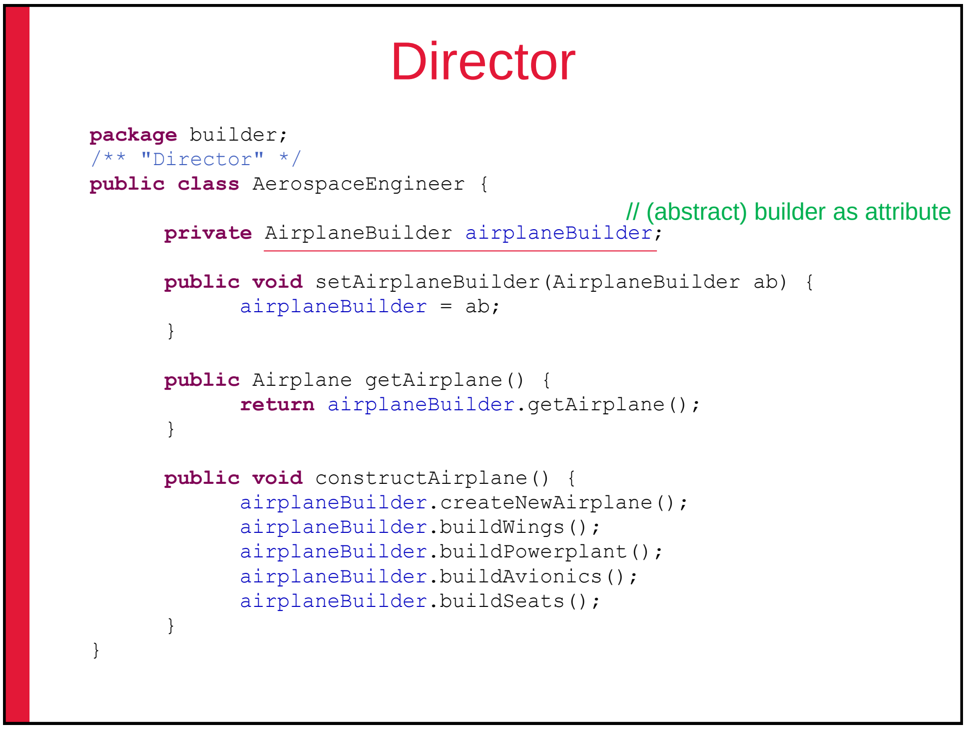
125



ConcreteBuilder 4

```
package builder;
/** "ConcreteBuilder" */
public class Airliner extends AirplaneBuilder {

        Airliner (String customer){
                super.customer = customer;
                super.type = "787 Dreamliner";
        }

        public void buildWings() {
                airplane.setWingspan(197f);
        }

        public void buildPowerplant() {
                airplane.setPowerplant("dual turbofan");
        }

        public void buildAvionics() {
                airplane.setAvionics("commercial");
        }

        public void buildSeats() {
                airplane.setNumberSeats(8,289);

        }

}
```

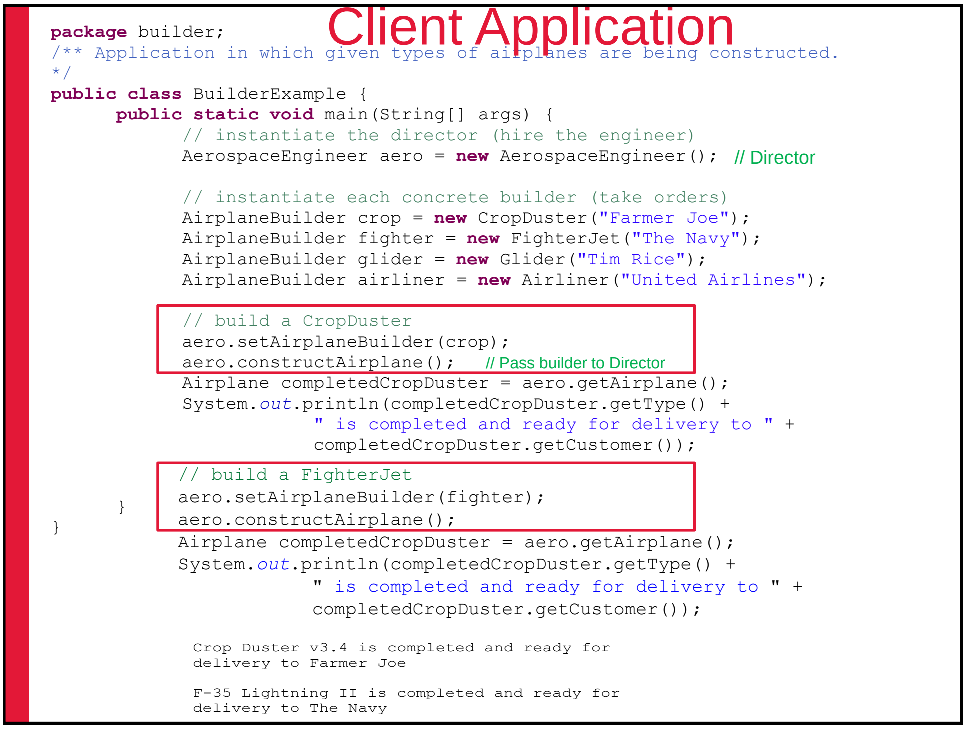
126



Director

```
package builder;
/** "Director" */
public class AerospaceEngineer {
                                                    // (abstract) builder as attribute
      private AirplaneBuilder airplaneBuilder;

      public void setAirplaneBuilder(AirplaneBuilder ab) {
            airplaneBuilder = ab;
      }

      public Airplane getAirplane() {
            return airplaneBuilder.getAirplane();
      }

      public void constructAirplane() {
            airplaneBuilder.createNewAirplane();
            airplaneBuilder.buildWings();
            airplaneBuilder.buildPowerplant();
            airplaneBuilder.buildAvionics();
            airplaneBuilder.buildSeats();
      }
}
```

127



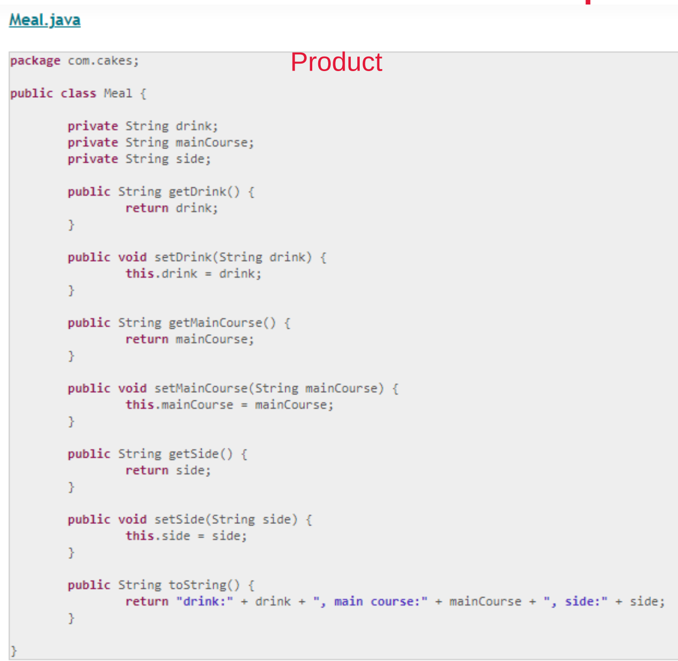
Client Application

```
package builder;
/** Application in which given types of airplanes are being constructed.
*/
public class BuilderExample {
      public static void main(String[] args) {
            // instantiate the director (hire the engineer)
            AerospaceEngineer aero = new AerospaceEngineer();  // Director

            // instantiate each concrete builder (take orders)
            AirplaneBuilder crop = new CropDuster("Farmer Joe");
            AirplaneBuilder fighter = new FighterJet("The Navy");
            AirplaneBuilder glider = new Glider("Tim Rice");
            AirplaneBuilder airliner = new Airliner("United Airlines");

            // build a CropDuster
            aero.setAirplaneBuilder(crop);
            aero.constructAirplane();    // Pass builder to Director
            Airplane completedCropDuster = aero.getAirplane();
            System.out.println(completedCropDuster.getType() +
                        " is completed and ready for delivery to " +
                        completedCropDuster.getCustomer());

            // build a FighterJet
            aero.setAirplaneBuilder(fighter);
      }
            aero.constructAirplane();
}
            Airplane completedCropDuster = aero.getAirplane();
            System.out.println(completedCropDuster.getType() +
                        " is completed and ready for delivery to " +
                        completedCropDuster.getCustomer());
```

```
Crop Duster v3.4 is completed and ready for
delivery to Farmer Joe

F-35 Lightning II is completed and ready for
delivery to The Navy
```

129

Another example

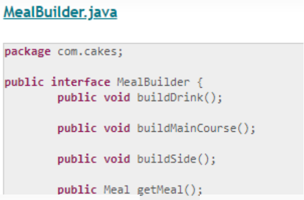


Meal.java

Product

```
package com.cakes;

public class Meal {

        private String drink;
        private String mainCourse;
        private String side;

        public String getDrink() {
                return drink;
        }

        public void setDrink(String drink) {
                this.drink = drink;
        }

        public String getMainCourse() {
                return mainCourse;
        }

        public void setMainCourse(String mainCourse) {
                this.mainCourse = mainCourse;
        }

        public String getSide() {
                return side;
        }

        public void setSide(String side) {
                this.side = side;
        }

        public String toString() {
                return "drink:" + drink + ", main course:" + mainCourse + ", side:" + side;
        }
}
```

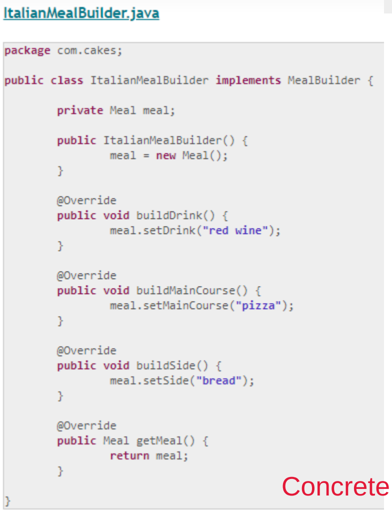
130



MealBuilder.java

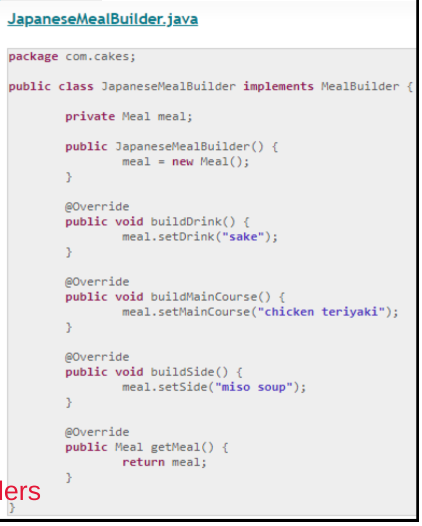
AbstractBuilder

```
package com.cakes;

public interface MealBuilder {
        public void buildDrink();

        public void buildMainCourse();

        public void buildSide();

        public Meal getMeal();
```



ItalianMealBuilder.java

```
package com.cakes;

public class ItalianMealBuilder implements MealBuilder {

        private Meal meal;

        public ItalianMealBuilder() {
                meal = new Meal();
        }

        @Override
        public void buildDrink() {
                meal.setDrink("red wine");
        }

        @Override
        public void buildMainCourse() {
                meal.setMainCourse("pizza");
        }

        @Override
        public void buildSide() {
                meal.setSide("bread");
        }

        @Override
        public Meal getMeal() {
                return meal;
        }
}
```



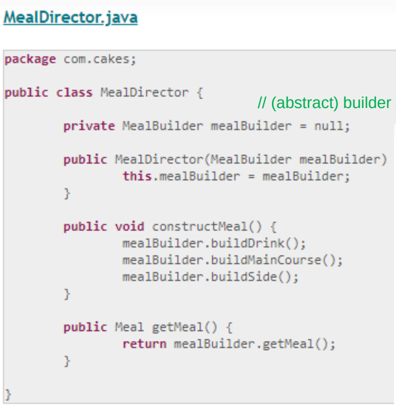
JapaneseMealBuilder.java

```
package com.cakes;

public class JapaneseMealBuilder implements MealBuilder {

        private Meal meal;

        public JapaneseMealBuilder() {
                meal = new Meal();
        }

        @Override
        public void buildDrink() {
                meal.setDrink("sake");
        }

        @Override
        public void buildMainCourse() {
                meal.setMainCourse("chicken teriyaki");
        }

        @Override
        public void buildSide() {
                meal.setSide("miso soup");
        }

        @Override
        public Meal getMeal() {
                return meal;
        }
}
```

131

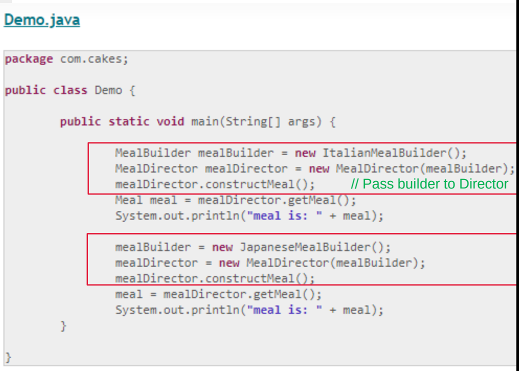
ConcreteBuilders

131



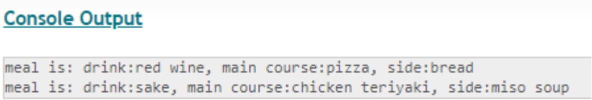
MealDirector.java

```java
package com.cakes;

public class MealDirector {                    // (abstract) builder as attribute

        private MealBuilder mealBuilder = null;

        public MealDirector(MealBuilder mealBuilder)
                this.mealBuilder = mealBuilder;
        }

        public void constructMeal() {
                mealBuilder.buildDrink();
                mealBuilder.buildMainCourse();
                mealBuilder.buildSide();
        }

        public Meal getMeal() {
                return mealBuilder.getMeal();
        }

}
```



Demo.java

```java
package com.cakes;

public class Demo {

        public static void main(String[] args) {

                MealBuilder mealBuilder = new ItalianMealBuilder();
                MealDirector mealDirector = new MealDirector(mealBuilder);
                mealDirector.constructMeal();        // Pass builder to Director
                Meal meal = mealDirector.getMeal();
                System.out.println("meal is: " + meal);

                mealBuilder = new JapaneseMealBuilder();
                mealDirector = new MealDirector(mealBuilder);
                mealDirector.constructMeal();
                meal = mealDirector.getMeal();
                System.out.println("meal is: " + meal);
        }

}
```



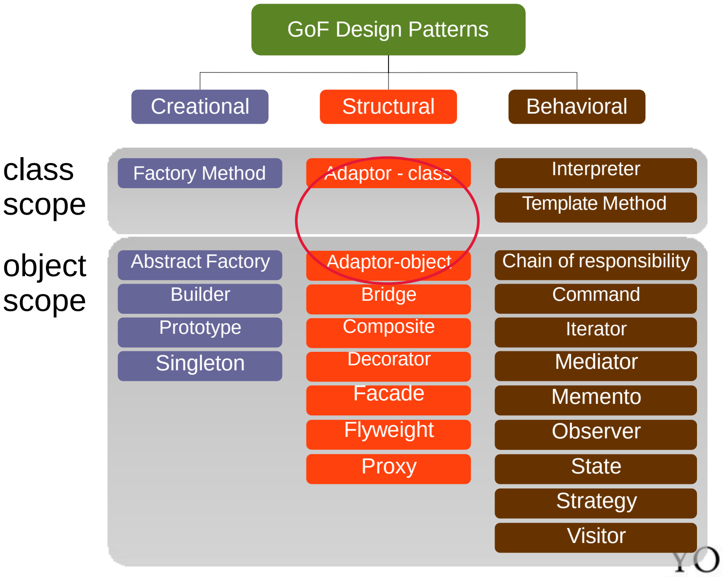
Console Output

```
meal is: drink:red wine, main course:pizza, side:bread
meal is: drink:sake, main course:chicken teriyaki, side:miso soup
```

132

132

Design Patterns Classification



133

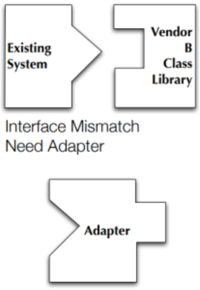
133

Adapter

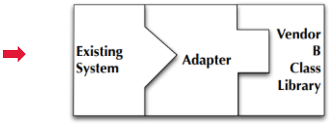
Intent

- *“**convert** the **interface of a class** into anothe . Adapter lets classes work together that couldn't otherwise because of **incompatible interface**”*

- also known as “wrapper”



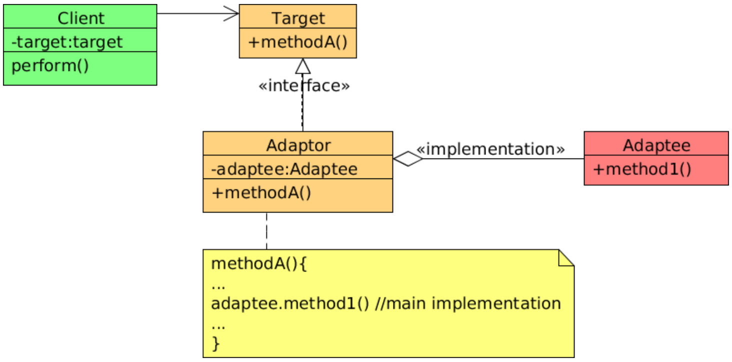
Interface Mismatch
Need Adapter



Adapter – Object

Requirement

- via object composition

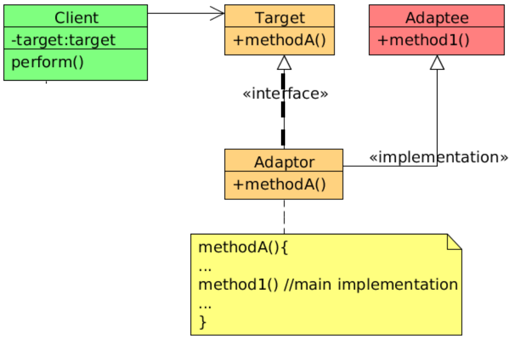


```
methodA(){
...
adaptee.method1() //main implementation
...
}
```

Adapter – Class

Requirement

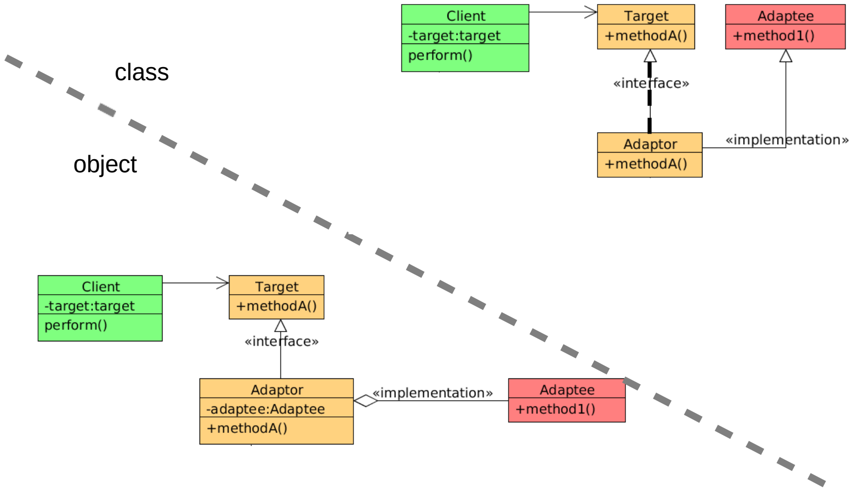
- requires multiple inheritance



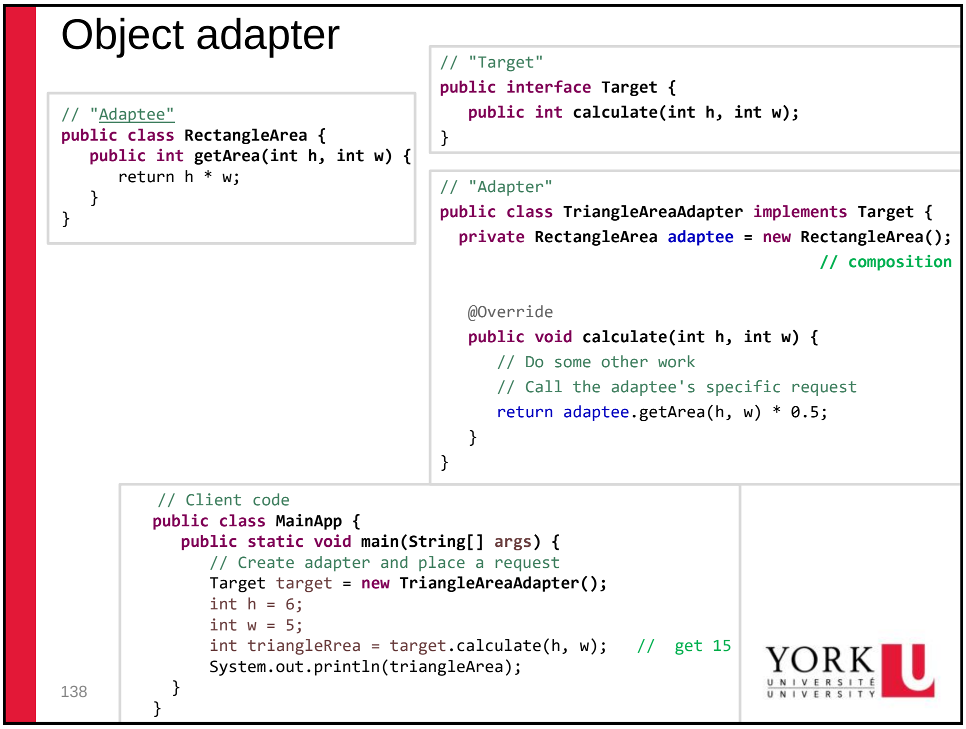
what about implementations that do not support multiple inheritance (Java)?

136

Adapter – Class vs. Object



137



Object adapter

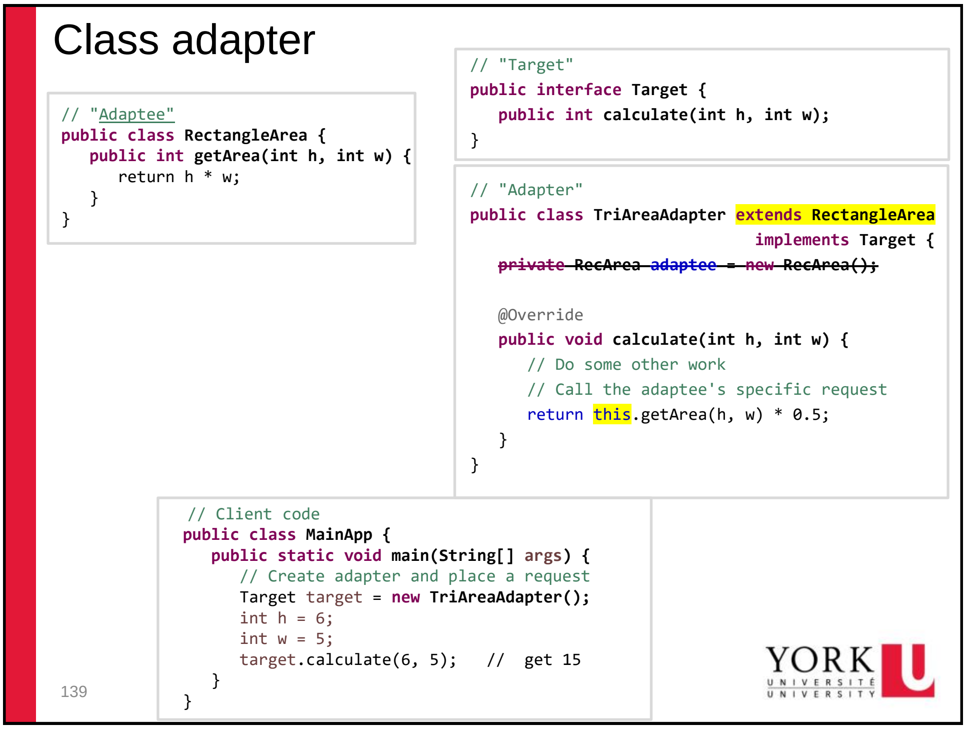
```
// "Adaptee"
public class RectangleArea {
   public int getArea(int h, int w) {
      return h * w;
   }
}
```

```
// "Target"
public interface Target {
   public int calculate(int h, int w);
}
```

```
// "Adapter"
public class TriangleAreaAdapter implements Target {
  private RectangleArea adaptee = new RectangleArea();
                                         // composition

   @Override
   public void calculate(int h, int w) {
      // Do some other work
      // Call the adaptee's specific request
      return adaptee.getArea(h, w) * 0.5;
   }
}
```

```
 // Client code
public class MainApp {
   public static void main(String[] args) {
      // Create adapter and place a request
      Target target = new TriangleAreaAdapter();
      int h = 6;
      int w = 5;
      int triangleRrea = target.calculate(h, w);   //  get 15
      System.out.println(triangleArea);
   }
}
```

138



Class adapter

```
// "Adaptee"
public class RectangleArea {
   public int getArea(int h, int w) {
      return h * w;
   }
}
```

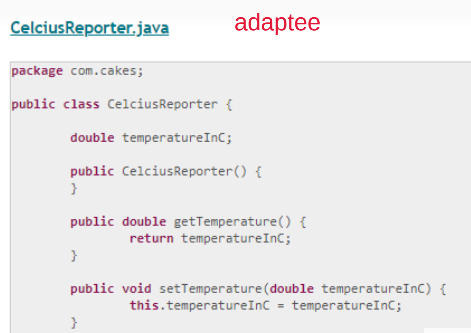
```
// "Target"
public interface Target {
   public int calculate(int h, int w);
}
```

```
// "Adapter"
public class TriAreaAdapter extends RectangleArea
                                  implements Target {
   ~~private RecArea adaptee = new RecArea();~~

   @Override
   public void calculate(int h, int w) {
      // Do some other work
      // Call the adaptee's specific request
      return this.getArea(h, w) * 0.5;
   }
}
```

```
 // Client code
public class MainApp {
   public static void main(String[] args) {
      // Create adapter and place a request
      Target target = new TriAreaAdapter();
      int h = 6;
      int w = 5;
      target.calculate(6, 5);   //  get 15
   }
}
```

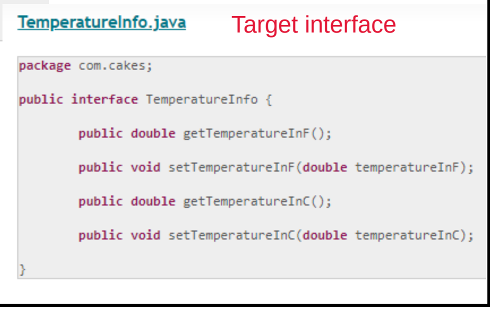
139

Another example



CelciusReporter.java adaptee

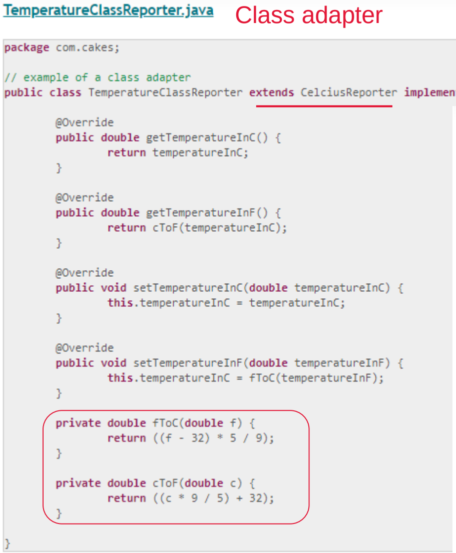
```
package com.cakes;

public class CelciusReporter {

        double temperatureInC;

        public CelciusReporter() {
        }

        public double getTemperature() {
                return temperatureInC;
        }

        public void setTemperature(double temperatureInC) {
                this.temperatureInC = temperatureInC;
        }

}
```



TemperatureInfo.java Target interface

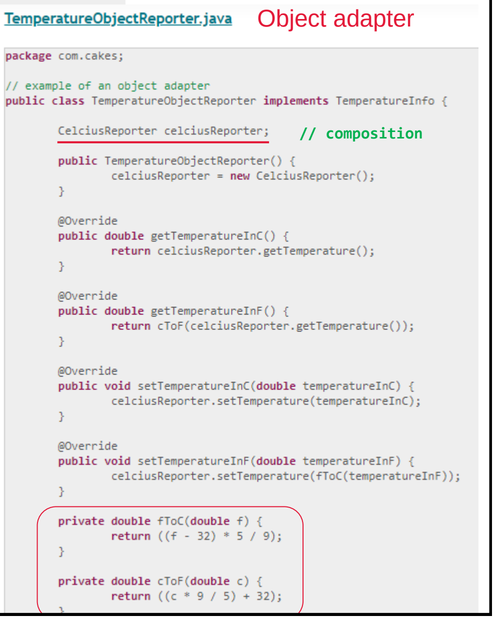
```
package com.cakes;

public interface TemperatureInfo {

        public double getTemperatureInF();

        public void setTemperatureInF(double temperatureInF);

        public double getTemperatureInC();

        public void setTemperatureInC(double temperatureInC);

}
```

140



TemperatureClassReporter.java Class adapter

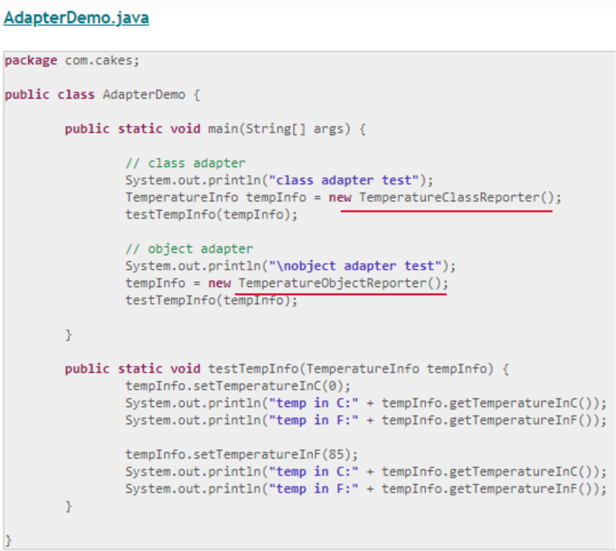
```
package com.cakes;

// example of a class adapter
public class TemperatureClassReporter extends CelciusReporter implements TemperatureInfo {

        @Override
        public double getTemperatureInC() {
                return temperatureInC;
        }

        @Override
        public double getTemperatureInF() {
                return cToF(temperatureInC);
        }

        @Override
        public void setTemperatureInC(double temperatureInC) {
                this.temperatureInC = temperatureInC;
        }

        @Override
        public void setTemperatureInF(double temperatureInF) {
                this.temperatureInC = fToC(temperatureInF);
        }

        private double fToC(double f) {
                return ((f - 32) * 5 / 9);
        }

        private double cToF(double c) {
                return ((c * 9 / 5) + 32);
        }

}
```



TemperatureObjectReporter.java Object adapter

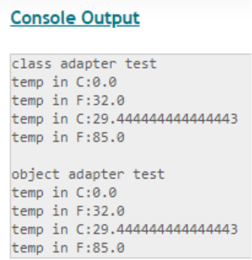
```
package com.cakes;

// example of an object adapter
public class TemperatureObjectReporter implements TemperatureInfo {

        CelciusReporter celciusReporter;   // composition

        public TemperatureObjectReporter() {
                celciusReporter = new CelciusReporter();
        }

        @Override
        public double getTemperatureInC() {
                return celciusReporter.getTemperature();
        }

        @Override
        public double getTemperatureInF() {
                return cToF(celciusReporter.getTemperature());
        }

        @Override
        public void setTemperatureInC(double temperatureInC) {
                celciusReporter.setTemperature(temperatureInC);
        }

        @Override
        public void setTemperatureInF(double temperatureInF) {
                celciusReporter.setTemperature(fToC(temperatureInF));
        }

        private double fToC(double f) {
                return ((f - 32) * 5 / 9);
        }

        private double cToF(double c) {
                return ((c * 9 / 5) + 32);
        }
```

141



AdapterDemo.java

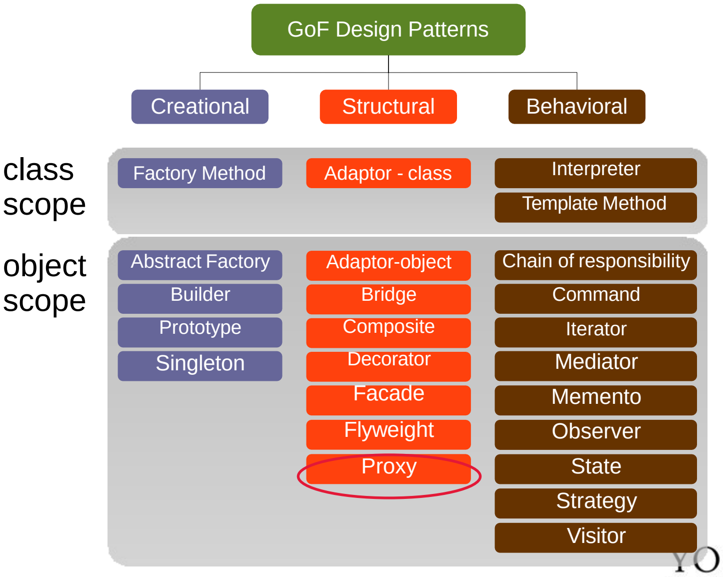
```
package com.cakes;

public class AdapterDemo {

        public static void main(String[] args) {

                // class adapter
                System.out.println("class adapter test");
                TemperatureInfo tempInfo = new TemperatureClassReporter();
                testTempInfo(tempInfo);

                // object adapter
                System.out.println("\nobject adapter test");
                tempInfo = new TemperatureObjectReporter();
                testTempInfo(tempInfo);

        }

        public static void testTempInfo(TemperatureInfo tempInfo) {
                tempInfo.setTemperatureInC(0);
                System.out.println("temp in C:" + tempInfo.getTemperatureInC());
                System.out.println("temp in F:" + tempInfo.getTemperatureInF());

                tempInfo.setTemperatureInF(85);
                System.out.println("temp in C:" + tempInfo.getTemperatureInC());
                System.out.println("temp in F:" + tempInfo.getTemperatureInF());
        }

}
```

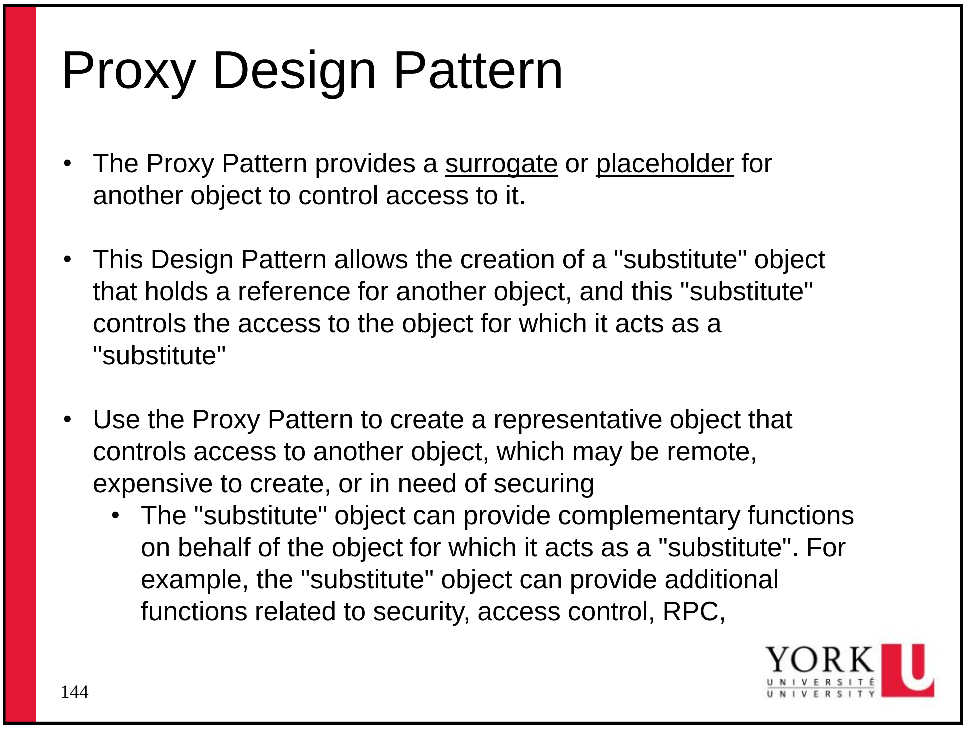


Console Output

```
class adapter test
temp in C:0.0
temp in F:32.0
temp in C:29.444444444444443
temp in F:85.0

object adapter test
temp in C:0.0
temp in F:32.0
temp in C:29.444444444444443
temp in F:85.0
```

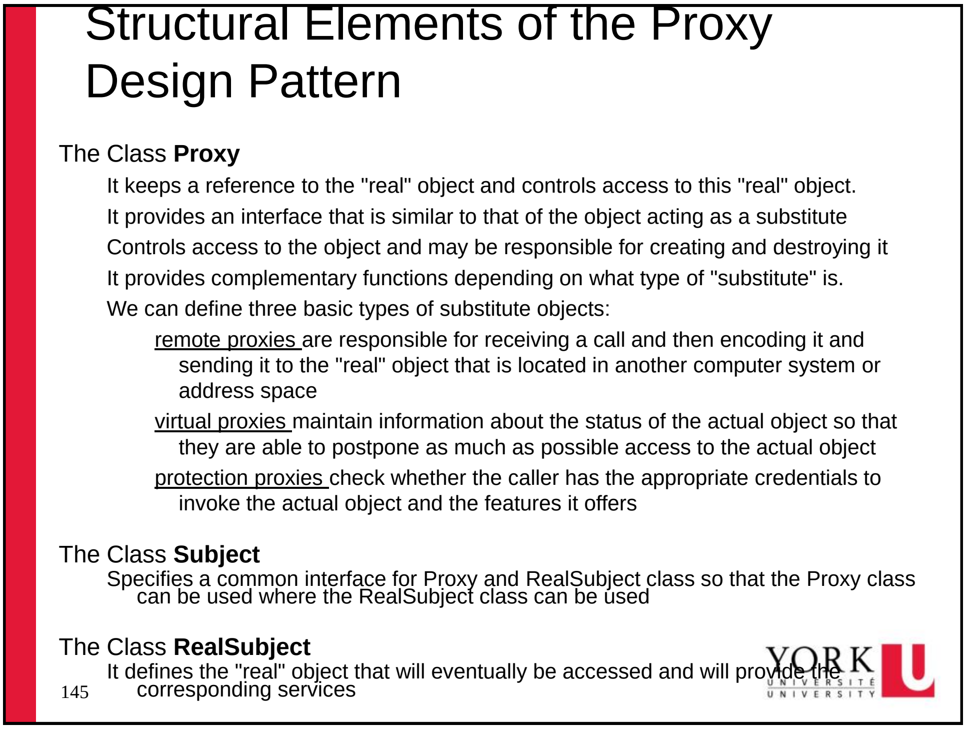
Design Patterns Classification





Proxy Design Pattern

- The Proxy Pattern provides a surrogate or placeholder for another object to control access to it.
- This Design Pattern allows the creation of a "substitute" object that holds a reference for another object, and this "substitute" controls the access to the object for which it acts as a "substitute"
- Use the Proxy Pattern to create a representative object that controls access to another object, which may be remote, expensive to create, or in need of securing
    - The "substitute" object can provide complementary functions on behalf of the object for which it acts as a "substitute". For example, the "substitute" object can provide additional functions related to security, access control, RPC,



Structural Elements of the Proxy Design Pattern

The Class **Proxy**

It keeps a reference to the "real" object and controls access to this "real" object.

It provides an interface that is similar to that of the object acting as a substitute

Controls access to the object and may be responsible for creating and destroying it

It provides complementary functions depending on what type of "substitute" is.

We can define three basic types of substitute objects:

remote proxies are responsible for receiving a call and then encoding it and sending it to the "real" object that is located in another computer system or address space

virtual proxies maintain information about the status of the actual object so that they are able to postpone as much as possible access to the actual object

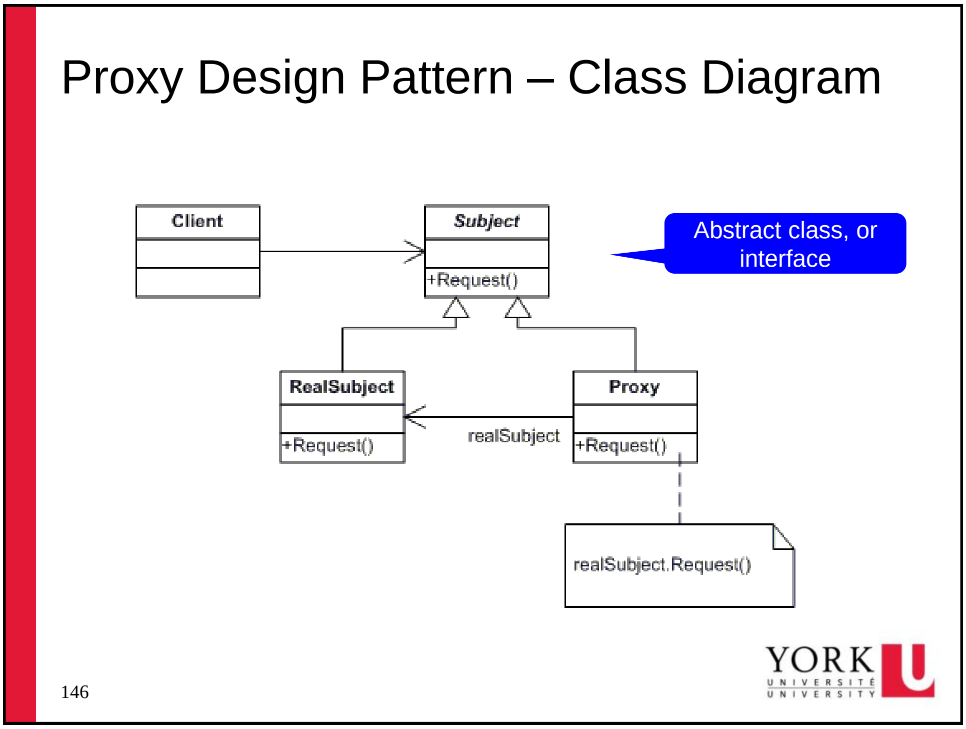
protection proxies check whether the caller has the appropriate credentials to invoke the actual object and the features it offers

The Class **Subject**

Specifies a common interface for Proxy and RealSubject class so that the Proxy class can be used where the RealSubject class can be used

The Class **RealSubject**

It defines the "real" object that will eventually be accessed and will provide the corresponding services

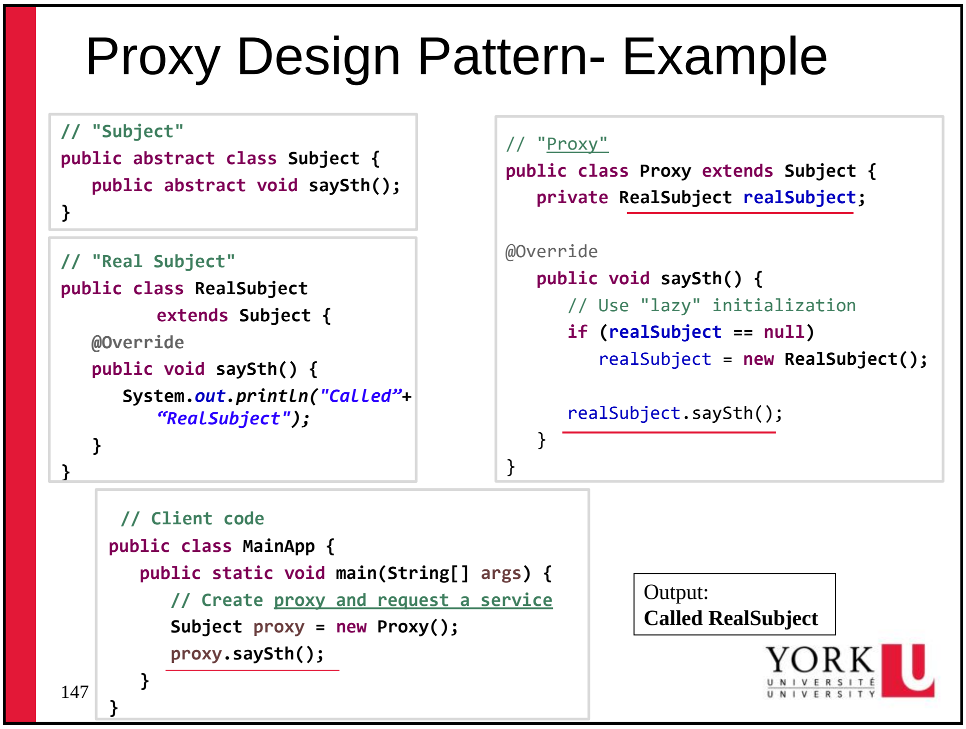


Proxy Design Pattern – Class Diagram

Abstract class, or interface

146

146



Proxy Design Pattern- Example

```
// "Subject"
public abstract class Subject {
   public abstract void saySth();
}
```

```
// "Real Subject"
public class RealSubject
         extends Subject {
   @Override
   public void saySth() {
      System.out.println("Called"+
         "RealSubject");
   }
}
```

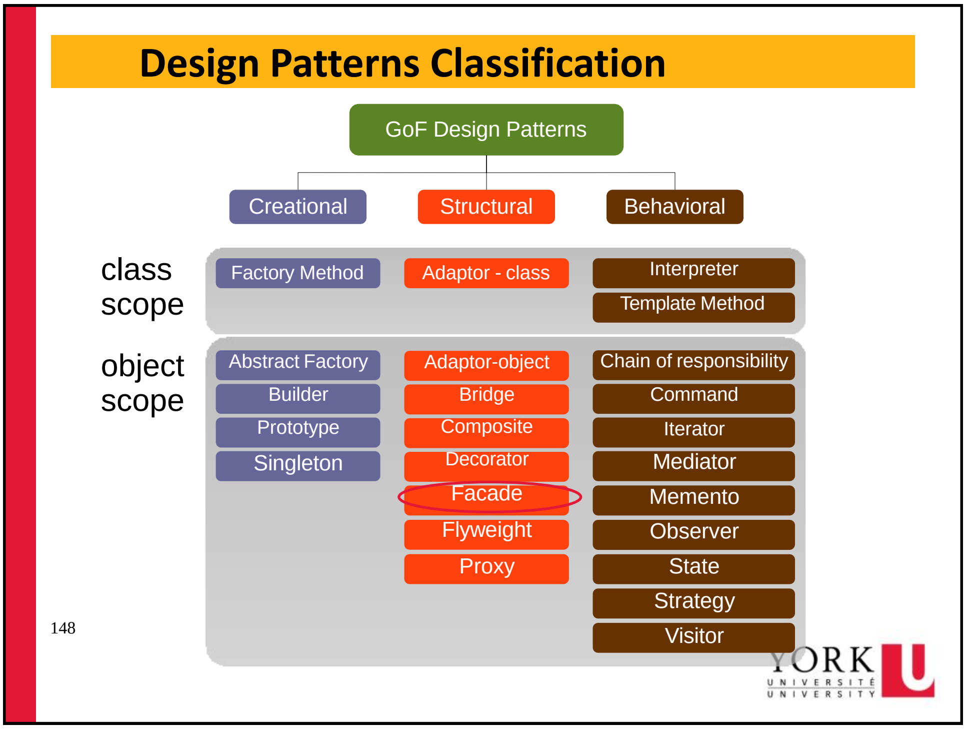
```
// "Proxy"
public class Proxy extends Subject {
   private RealSubject realSubject;

@Override
   public void saySth() {
      // Use "lazy" initialization
      if (realSubject == null)
         realSubject = new RealSubject();

      realSubject.saySth();
   }
}
```

```
 // Client code
public class MainApp {
   public static void main(String[] args) {
      // Create proxy and request a service
      Subject proxy = new Proxy();
      proxy.saySth();
   }
}
```

```
Output:
Called RealSubject
```

147

147



Design Patterns Classification

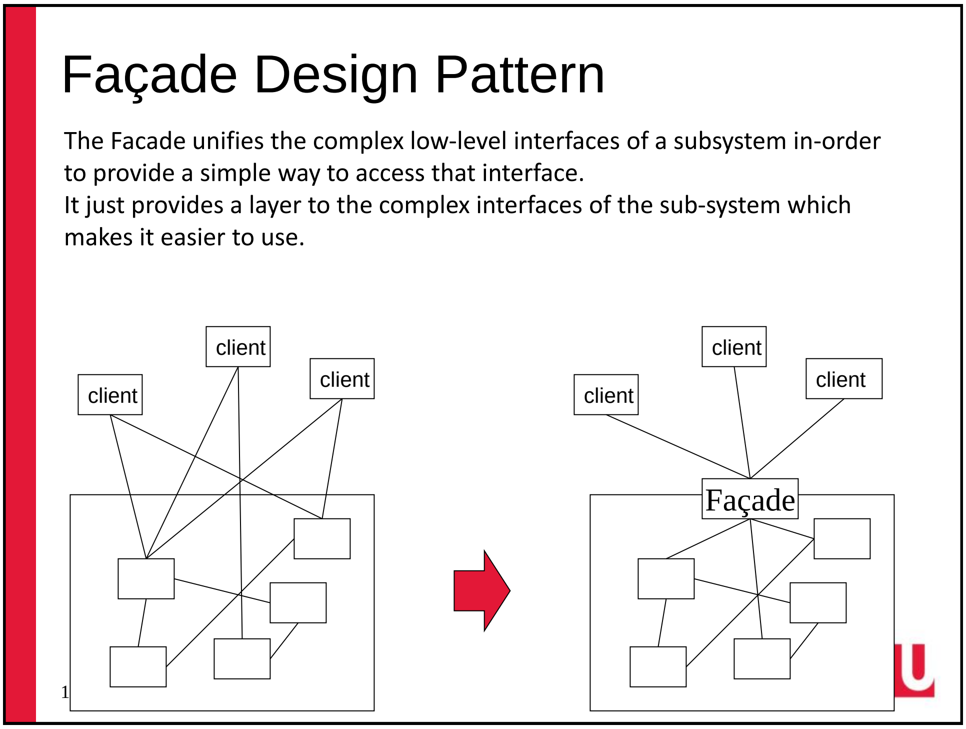
GoF Design Patterns

	Creational	Structural	Behavioral
class scope	Factory Method	Adaptor - class	Interpreter Template Method
object scope	Abstract Factory Builder Prototype Singleton	Adaptor-object Bridge Composite Decorator Facade Flyweight Proxy	Chain of responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

148

YORK UNIVERSITÉ UNIVERSITY

148

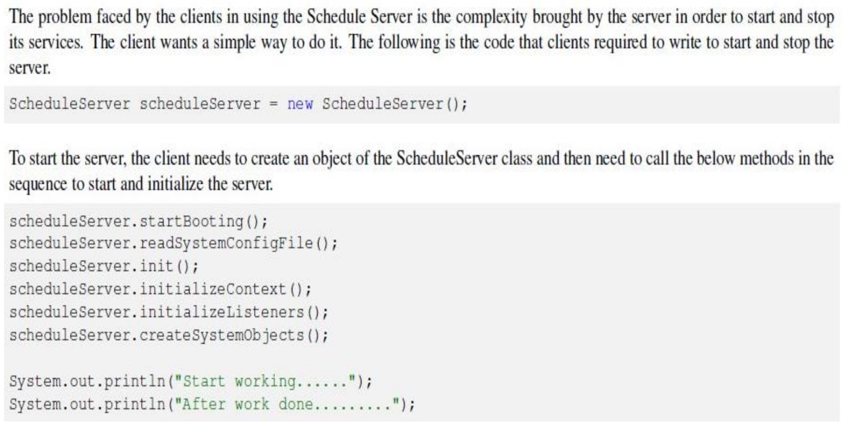


Façade Design Pattern

The Facade unifies the complex low-level interfaces of a subsystem in-order to provide a simple way to access that interface.
It just provides a layer to the complex interfaces of the sub-system which makes it easier to use.

1

149

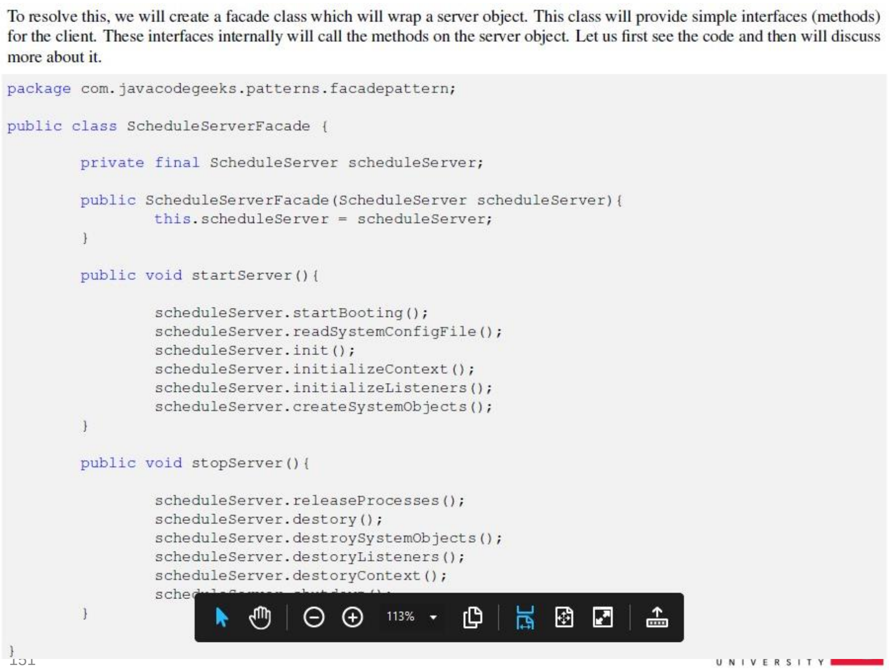


The problem faced by the clients in using the Schedule Server is the complexity brought by the server in order to start and stop its services. The client wants a simple way to do it. The following is the code that clients required to write to start and stop the server.

```
ScheduleServer scheduleServer = new ScheduleServer();
```

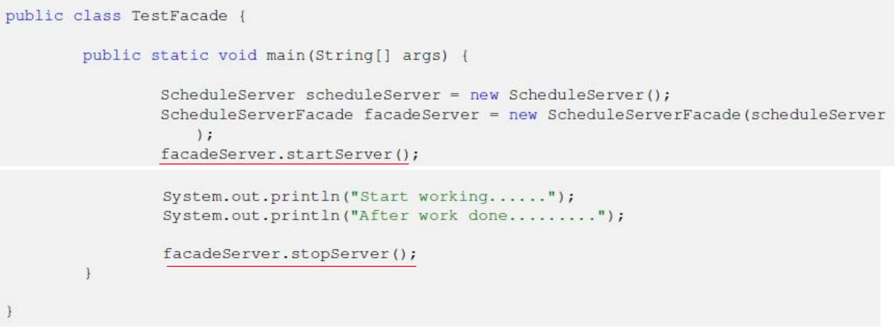
To start the server, the client needs to create an object of the ScheduleServer class and then need to call the below methods in the sequence to start and initialize the server.

```
scheduleServer.startBooting();
scheduleServer.readSystemConfigFile();
scheduleServer.init();
scheduleServer.initializeContext();
scheduleServer.initializeListeners();
scheduleServer.createSystemObjects();

System.out.println("Start working......");
System.out.println("After work done.........");
```



To resolve this, we will create a facade class which will wrap a server object. This class will provide simple interfaces (methods) for the client. These interfaces internally will call the methods on the server object. Let us first see the code and then will discuss more about it.

```
package com.javacodegeeks.patterns.facadepattern;

public class ScheduleServerFacade {

        private final ScheduleServer scheduleServer;

        public ScheduleServerFacade(ScheduleServer scheduleServer){
                this.scheduleServer = scheduleServer;
        }

        public void startServer(){

                scheduleServer.startBooting();
                scheduleServer.readSystemConfigFile();
                scheduleServer.init();
                scheduleServer.initializeContext();
                scheduleServer.initializeListeners();
                scheduleServer.createSystemObjects();
        }

        public void stopServer(){

                scheduleServer.releaseProcesses();
                scheduleServer.destory();
                scheduleServer.destroySystemObjects();
                scheduleServer.destoryListeners();
                scheduleServer.destoryContext();
                scheduleServer.shutdown();
        }
}
```



```
public class TestFacade {

        public static void main(String[] args) {

                ScheduleServer scheduleServer = new ScheduleServer();
                ScheduleServerFacade facadeServer = new ScheduleServerFacade(scheduleServer
                   );
                facadeServer.startServer();

                System.out.println("Start working......");
                System.out.println("After work done.........");

                facadeServer.stopServer();
        }

}
```

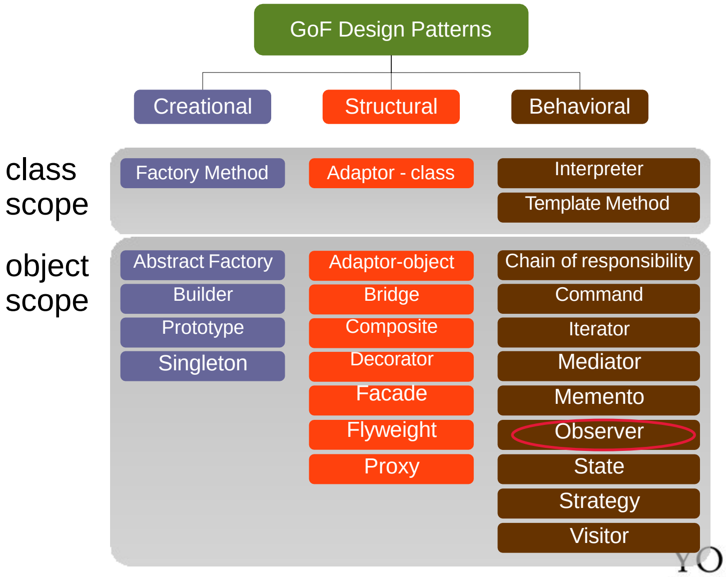
Façade – Another Example(2)

```
// "Car Engine Facade"
public class CarEngineFacade {
   private static int DEFAULT_COOLING_TEMP = 90;
   private static int MAX_ALLOWED_TEMP = 50;
   private FuelInjector fuelInjector =
                          new FuelInjector();
   private AirFlowController airFlowController =
                          new AirFlowController();
   private Starter starter = new Starter();
   private CoolingController coolingController =
                          new CoolingController();
   private CatalyticConverter catalyticConverter =
                          new CatalyticConverter();
```

```
public void startEngine() {
      airFlowController.on();
      airFlowController.takeAir();
      fuelInjector.on();
      fuelInjector.inject();
      starter.start();
      coolingController
           .setTemperatureUpperLimit(
                DEFAULT_COOLING_TEMP
            );
      coolingController.run();
      catalyticConverter.on();
   }

   public void stopEngine() {
      fuelInjector.off();
      catalyticConverter.off();
      coolingController
           .cool(MAX_ALLOWED_TEMP);
      coolingController.stop();
      airFlowController.off();
   }
}
```

```
// client code
main(){
  CarEngineFacade cef = new carEnginefacade();

   // To start the engine
  cef.startEngine();

  // To stop the engine
  cef.stopEngine();
}
```

Design Patterns Classification



GoF Design Patterns

	Creational	Structural	Behavioral
class scope	Factory Method	Adaptor - class	Interpreter Template Method
object scope	Abstract Factory Builder Prototype Singleton	Adaptor-object Bridge Composite Decorator Facade Flyweight Proxy	Chain of responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

154

154

Observer Design Pattern

Intent

Define a one-to-many dependency between objects so that when one object (i.e. the *subject*) changes state, all its dependents (i.e. *observers*) are notified and updated (or perform an operation) automatically.

Applicability

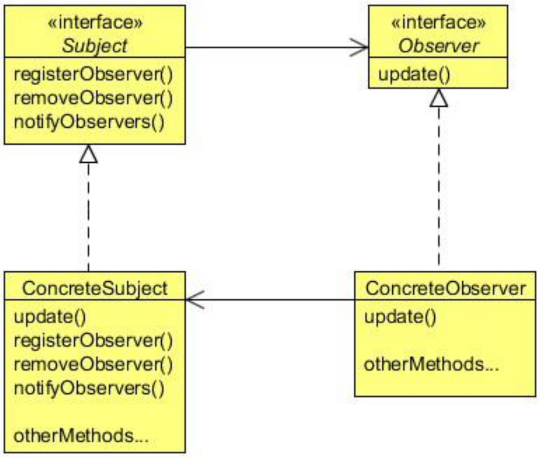
When an abstraction has two aspects, one dependent on the other.

When a change to one object requires changing others, and you don't know how many objects need to be changed.

When an object should notify other objects without making assumptions about who these objects are.

155

155



Observer - Example (1)

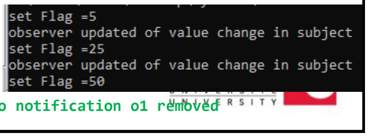
```
// "Subject"
public interface Subject {
  public void register(Observer observer);
  public void unregister(Observer observer);
  public void notifyObservers();
  public void setFlag(int fg){
}
```

```
// "Observer"
interface Observer {
  public void update();
}
```

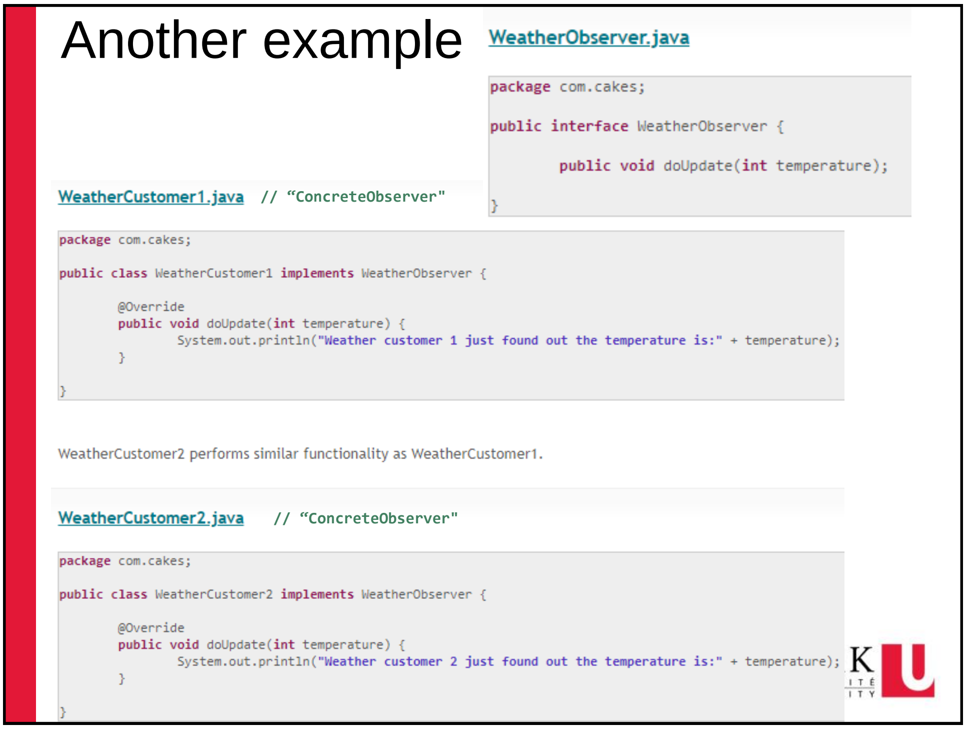
```
// “ConcreteObserver"
class ConcreteObserver implements Observer{
  public void update(){
    System.out.println("observer updated of
                   value change in subject");
  }
}
```

```
// " concrete subject"
public class ConcreteSubject implements Subject {
  List<Observer> observerList = new ArrayList<>();
  int flag;

  public void setFlag(int fg){
     this.flag = fg;
     notifyObservers();
  }

  ~~@override~~
  public void register(Observer o) {
    observerList.add(o);
  }

  ~~@override~~
  public void unregister(Observer o) {
    observerList.remove(o);
  }

  ~~@override~~
  public void notifyObservers() {
    for(Observer o: observerList)
       o.update();
  }
}
```

```
public static void main(String[] args) {

   Observer o1 = new ConcreteObserver();
   Subject sub1 = new ConcreteSubject();
   sub1.register(o1);
   System.out.println("set Flag =5");  sub1.setFlag(5);
   System.out.println("set Flag =25"); sub1.setFlag(25);
   sub1.unregister(o1);
   System.out.println("set Flag =50"); sub1.setFlag(50); // no notification o1 removed
}
```



```
set Flag =5
observer updated of value change in subject
set Flag =25
observer updated of value change in subject
set Flag =50
```



Another example

WeatherObserver.java

```
package com.cakes;

public interface WeatherObserver {

        public void doUpdate(int temperature);

}
```

WeatherCustomer1.java   // “ConcreteObserver"

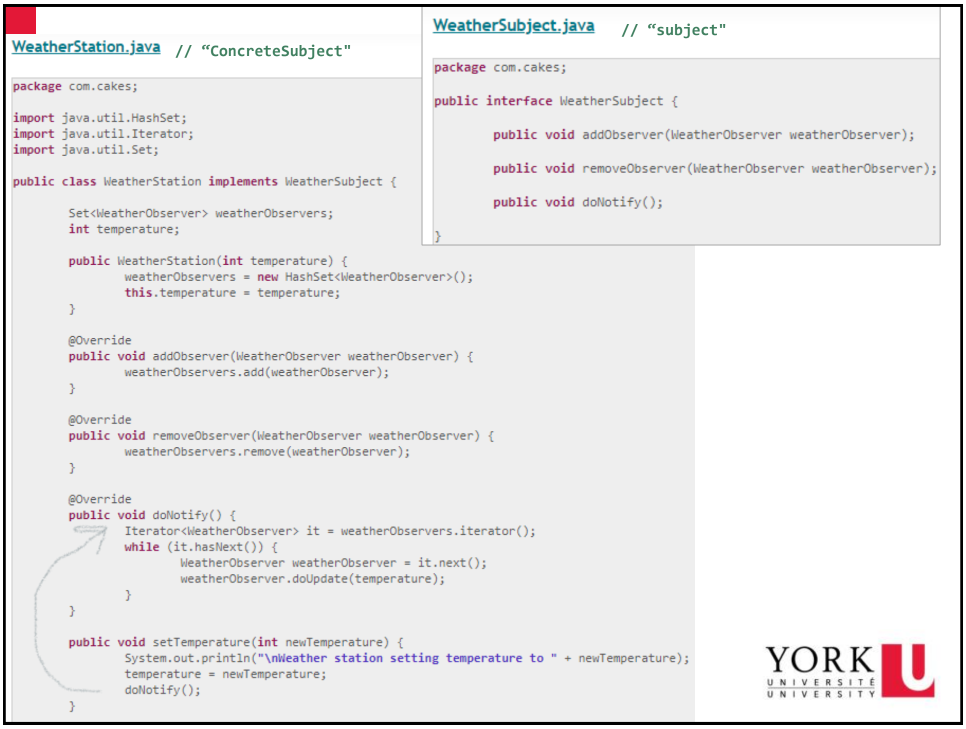
```
package com.cakes;

public class WeatherCustomer1 implements WeatherObserver {

        @Override
        public void doUpdate(int temperature) {
                System.out.println("Weather customer 1 just found out the temperature is:" + temperature);
        }

}
```

WeatherCustomer2 performs similar functionality as WeatherCustomer1.

WeatherCustomer2.java   // “ConcreteObserver"

```
package com.cakes;

public class WeatherCustomer2 implements WeatherObserver {

        @Override
        public void doUpdate(int temperature) {
                System.out.println("Weather customer 2 just found out the temperature is:" + temperature);
        }

}
```

161



WeatherStation.java   // “ConcreteSubject"

```
package com.cakes;

import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

public class WeatherStation implements WeatherSubject {

        Set<WeatherObserver> weatherObservers;
        int temperature;

        public WeatherStation(int temperature) {
                weatherObservers = new HashSet<WeatherObserver>();
                this.temperature = temperature;
        }

        @Override
        public void addObserver(WeatherObserver weatherObserver) {
                weatherObservers.add(weatherObserver);
        }

        @Override
        public void removeObserver(WeatherObserver weatherObserver) {
                weatherObservers.remove(weatherObserver);
        }

        @Override
        public void doNotify() {
                Iterator<WeatherObserver> it = weatherObservers.iterator();
                while (it.hasNext()) {
                        WeatherObserver weatherObserver = it.next();
                        weatherObserver.doUpdate(temperature);
                }
        }

        public void setTemperature(int newTemperature) {
                System.out.println("\nWeather station setting temperature to " + newTemperature);
                temperature = newTemperature;
                doNotify();
        }
```

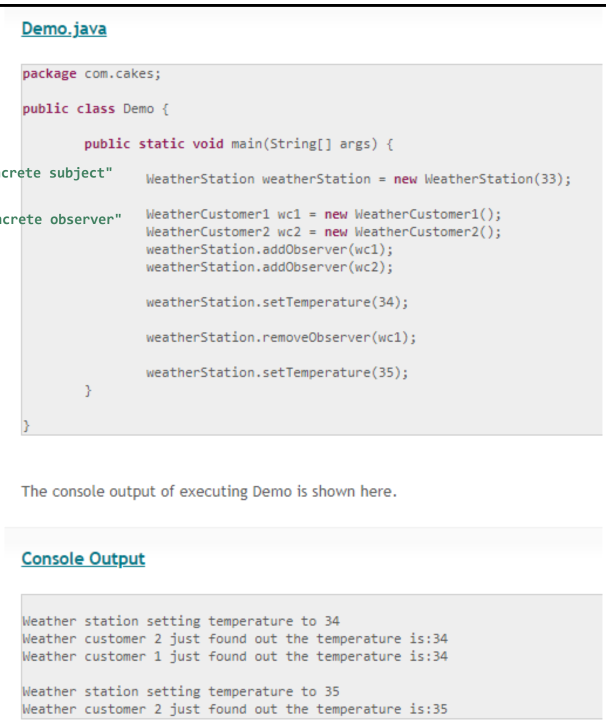
WeatherSubject.java   // “subject"

```
package com.cakes;

public interface WeatherSubject {

        public void addObserver(WeatherObserver weatherObserver);

        public void removeObserver(WeatherObserver weatherObserver);

        public void doNotify();

}
```

162

// "Concrete subject"

// "Concrete observer"



Demo.java

```
package com.cakes;

public class Demo {

        public static void main(String[] args) {

                WeatherStation weatherStation = new WeatherStation(33);

                WeatherCustomer1 wc1 = new WeatherCustomer1();
                WeatherCustomer2 wc2 = new WeatherCustomer2();
                weatherStation.addObserver(wc1);
                weatherStation.addObserver(wc2);

                weatherStation.setTemperature(34);

                weatherStation.removeObserver(wc1);

                weatherStation.setTemperature(35);
        }

}
```

The console output of executing Demo is shown here.

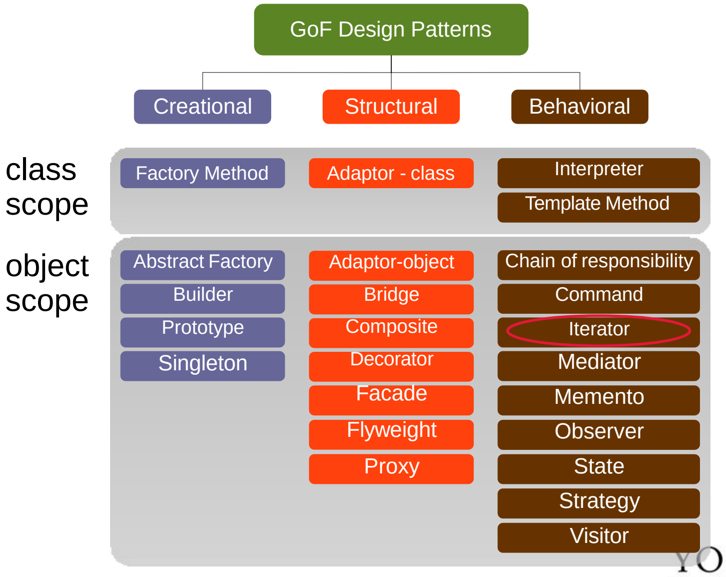
Console Output

```
Weather station setting temperature to 34
Weather customer 2 just found out the temperature is:34
Weather customer 1 just found out the temperature is:34

Weather station setting temperature to 35
Weather customer 2 just found out the temperature is:35
```

163

163

Design Patterns Classification



166

166

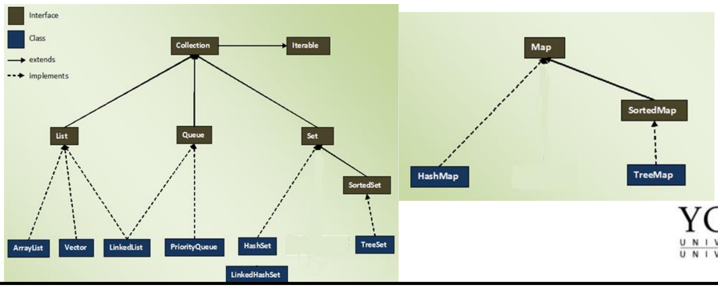
Iterator pattern

iterator: an object that provides a standard way to examine all elements of any collection

- uniform interface for traversing many different data structures
- supports concurrent iteration and element removal

```
  Iterator<Account> itr = list.iterator();
  while (itr.hasNext()) {
    Account a = itr.next();
    System.out.println(a);
}
```

```
set.iterator()
map.keySet().iterator()
map.values().iterator()
```


Interface
Class
extends
implements
Collection
Iterable
List
Queue
Set
SortedSet
ArrayList
Vector
LinkedList
PriorityQueue
HashSet
TreeSet
LinkedHashSet
Map
SortedMap
HashMap
TreeMap