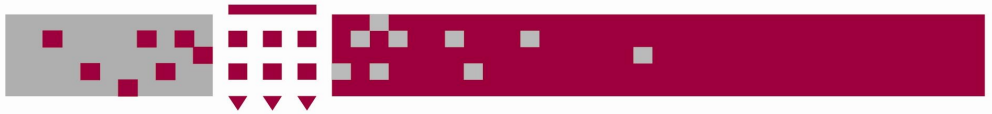
5COSC019W – Object Oriented Programming
Week 2

Dr. Barbara Villarini

b.villarini@westminster.ac.uk

Outline

- Static methods and variables
- Static, non-static context
- Differences: instance, class and local variable
- Using Java Predefined Classes
- String class and Packages
- UML diagrams

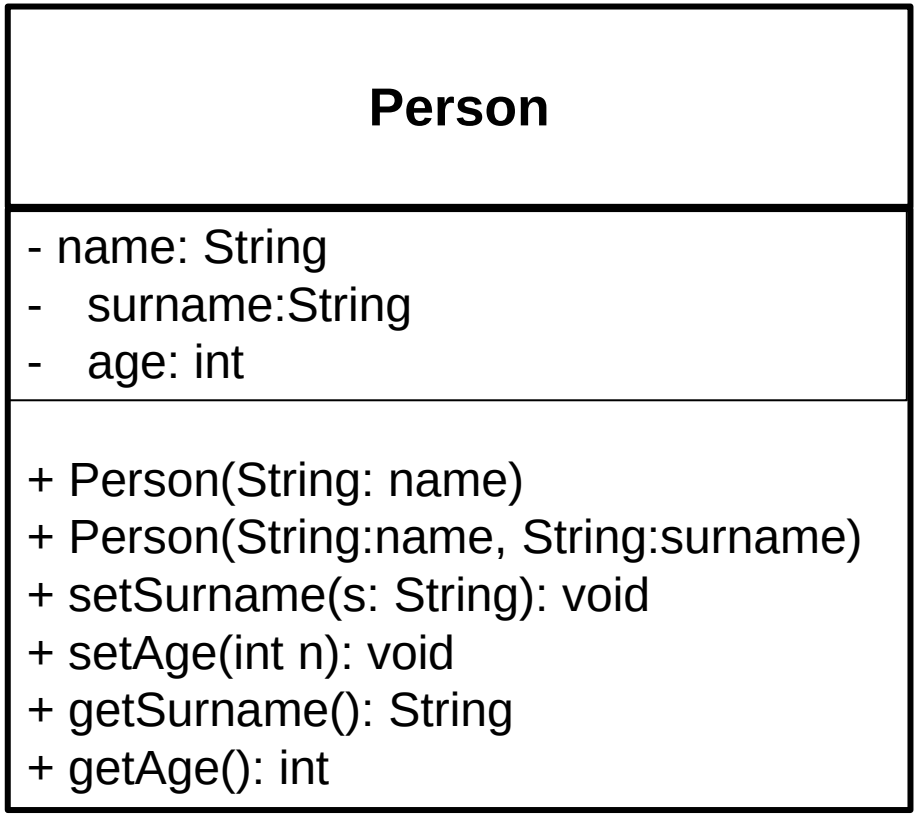


Last week…

- Objects and Classes
- Class declaration
- Object as instance of a class
- Constructors
- Object Assignment
- Overloading methods

RECAP – A Class

- A class is the collection of related data and functions under a single name
- When related data and functions are kept under a class, it helps to visualize the complex problem efficiently and effectively.



Defining the class

```
public class Person{

  attributes/data/variables for Person

  Constructors: to instantiate objects

  Methods: to interact with the object

}
```

Example

```
class Person {

private String name;
private String surname;
private int age;

public Person(String n) {
name= n;
}

public Person(String n, String s) {
name= n;
surname = s;
}

public String getSurname() {
return surname;
}

public int getAge() {
Return age;
}

… }
```

Attributes of the class Person

Constructors

Methods

Keywords: private and public

- Private variables and methods are accessible inside that class only
- Public variables and methods are accessible both inside and outside the class.
- This feature in OOP is known as data hiding. If programmer mistakenly tries to access private data outside the class, compiler shows error which prevents the misuse of data.

Accessing variables and methods

- Variables and methods can be accessed in similar way using member operator (.)
- Example: **method1()** can be called using code:

```
object_name.method1();
```

- Similarly a variable can be accessed:

```
Object_name.nameVariable
```

They can be accessed outside the class only if they are not private!

Example

```
public class TestPerson{

    public static void main(String[] args) {
        Person p = new Person(”Tom");
        p.surname = “Smith”;
        p.setSurname(“Smith”);
        System.out.printl(The surname is +
                p.getSurname());
    }
}
```

NO! Surname is private

We access surname through the set method

Dynamic Variables and Methods

- All instance variables and methods we have created so far have been *dynamic*
  - Note: There is no "dynamic" keyword in Java
  - Dynamic by default
- In general, *dynamic* refers to things created at "run time" i.e. when the program is running
- Every object gets its own (dynamic) instance variables.
- *The absence* of the keyword *static* before non-local variables and methods means *dynamic* (one per object/instance)

Static Variables

- *Static* means "pertaining to the class in general", *not* to an individual object

- A variable may be declared (outside of a method) with the *static* keyword:
  - **E.g. `static int numTicketsSold;`**

- A static variable is *shared* by all instances (if any).
  - All instances may be able to read/write it

- A static variable that is public may be accessed by:
  - **`ClassName.variableName`**
  - Example: `Math.PI`

Static methods

- A method may be declared with the *static* keyword
- Static methods live at *class level*, not at *object level*
- Static methods *may access* static variables and methods, but not dynamic ones

  - Example:

```
public static int getNumSold(){
    return numTicketsSold;
    }
```

Static variable

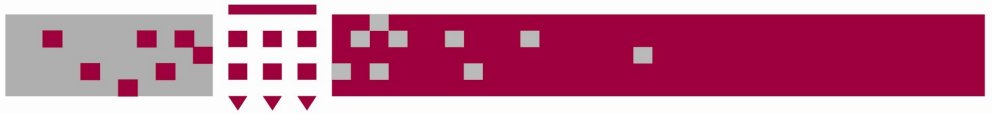
Static Methods

- A static method that is public can be accessed

`ClassName.methodName(args)`

- Examples:

```
double result = Math.sqrt(25.0);

int numSold = Ticket.getNumberSold();
```



Example

```
public class Ticket{
  private static int numTicketsSold = 0; // shared
  private int ticketNum; // one per object

  public Ticket(){
    numTicketsSold++;
    ticketNum = numTicketsSold;
  }

  public static int getNumberSold() {
    return numTicketsSold;
  }

  public int getTicketNumber() {
     return ticketNum;
  }

  public String getInfo(){
    return "ticket # " + ticketNum + "; " +
             numTicketsSold + " ticket(s) sold.";
  }
}
```

Example - output

```
> Ticket.getNumberSold()
0
> Ticket t1 = new Ticket();
> t1.getTicketNum()
1
> t1.getInfo()
"ticket # 1; 1 ticket(s) sold."
> t1.getNumberSold()
1
> Ticket t2 = new Ticket();
> t2.getTicketNum()
2
> t2.getInfo()
"ticket # 2; 2 ticket(s) sold."
> t2.getInfo()
"ticket # 2; 2 ticket(s) sold."
> t1.getInfo()
"ticket # 1; 2 ticket(s) sold."
> Ticket.getNumberSold()
2
```

Static context

- To have standalone Java Application we need a `public static void main(String args[])` method
  - The main method belongs to the class in which it is written
- It must be static because, before your program starts, there *aren't any objects* to send messages to
- This is a static context (a class method)
  - You can send messages to objects, *if* you have some objects: d1.bark();
  - You *cannot* send a message to yourself, or use any instance variables - this is a static context, not an object
- Non-static variable cannot be referenced from a static context

A wrong example

```
public class JustAdd {
    int x;
    int y;
    int z;

    public static void main(String args[]) {
        x = 5;
        y = 10;
        z = x + y;
    }
}
```

Solution

```
public class JustAdd {
    int x;
    int y;
    int z;

     public static void main(String args[]) {
           JustAdd myAdd = new JustAdd();
           myAdd.doItAll();
      }

     void doItAll() {
         x = 5;
         y = 10;
         z = x + y;
     }
}
```

When to use static

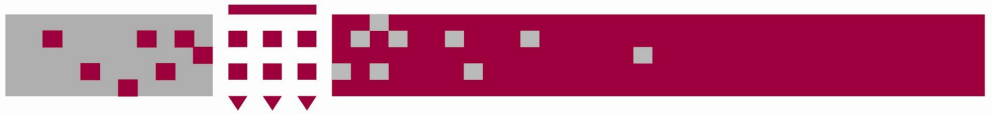
- A variable should be static if:
  - It logically describes the class as a whole
  - There should be only one copy of it
- A method should be static if:
  - It does not use or affect the objects

Static – Non-static

- We can access static variables without creating an instance of the class
- As they are already available at class loading time, we can use them in any of our non static methods.
- We cannot use non static methods and variables without creating an instance of the class as they are bound to the instance of the class.
- They are initialized by the constructor when we create the object using new operator.

Example

```
Class staticDemo{
   public static int a = 100; // All instances of staticDemo have this
                       // variable as a common variable
   public int b =2 ;

   public static void showA(){
         System.out.println("A is "+a);
   }
}

Class exerciseClass{
   public static void main(String args[]){
         staticDemo.a = 35; // when we use the class name, the class is
                     // loaded, direct access to a without any instance

         staticDemo.b = 22; // ERROR this is not valid for non static variable

         staticDemo demo = new staticDemo();

         demo.b = 200;  // valid to set a value for a non static variable after
                        // creating an instance.

         staticDemo.showA();  //prints 35
   }
}
```



Why do we need this?

- Static methods are identified to be mostly used when we are writing any utility methods.
- We can also use static variables when sharing data.
- When sharing data do keep in mind about multithreading can cause inconsistency in the value. (synchronize the variable) *WE WILL SEE LATER DURING THE MODULE*

Recap: Class variables and methods

We saw that:

- Instance variables belong to a specific instance.
- Instance methods are invoked by an instance of the class.

We can also define:

- Class variables are shared by all the instances of the class.
- Class methods are not tied to a specific object.
- *To declare class variables and methods, use the static modifier.*

USING JAVA PREDEFINED CLASSES

Using Java Predefined Classes

- Java Packages
- The *String* Class
- Using *System.out* and *System.in*
- The *Math* Class

Java Predefined Classes

- Included in the Java SDK are more than 2,000 classes that can be used to add functionality to our programs
- APIs for Java classes are published on the Oracle web site:

  https://docs.oracle.com/javase/7/docs/api/

Java Packages

- Classes are grouped in **packages** according to functionality

Package	Categories of Classes
java.lang	Basic functionality common to many programs, such as the String class and Math class
java.awt	Graphics classes for drawing and using colors
javax.swing	User-interface components
java.text	Classes for formatting numeric output
java.util	The Scanner class and other miscellaneous classes

Using a Class From a Package

- Classes in *java.lang* are automatically available to use
- Classes in other packages need to be "imported" using this syntax:

```
import package.ClassName;
```

or

```
import package.*;
```

- Example

```
import java.text.DecimalFormat;
```

or

```
import java.text.*;
```

The String Class

- Represents a sequence of characters
- *String* constructors:

`String( String str )` allocates a *String* object with the value of *str*, which can be *String* object or a *String* literal
`String( )` allocates an empty *String*

Constructing Strings

```
String newString = new String(stringLiteral);
```

```
String message = new String("Welcome to Java");
```

Since strings are used frequently, Java provides a shorthand initializer for creating a string:

```
String message = "Welcome to Java";
```

The *length* Method

Return type	Method name and argument list
`int`	`length( )` returns the number of characters in the *String*

Example:

```
String hello = "Hello";
int len = hello.length( );
```

The value of len is 5

The *Math* Class Constants

- Two *static* constants

  PI - the value of pi

  E - the base of the natural logarithm

- Example:

```
System.out.println( Math.PI );
System.out.println( Math.E );
```

output is:

```
3.141592653589793
2.718281828459045
```

Methods of the *Math* Class

All methods are *static*

Return type	Method name and argument list
`dataTypeOfArg`	`abs( dataType arg )` returns the absolute value of the argument *arg*, which can be a *double, float, int* or *long*.
`double`	`log( double a )` returns the natural logarithm (in base e) of its argument.
`double`	`sqrt( double a )` returns the positive square root of a
`double`	`pow( double base, double exp )` returns the value of *base* raised to the power of *exp*

UML - THE UNIFIED MODELING LANGUAGE

UML

- De-facto standard notation
  – A small subset introduced in COMP1008.
  – Maintained by the OMG (Object Management Group), see www.uml.org

- UML provides a complete language for describing object-oriented models (like a programming language).
- Also provides a *visual notation* for displaying models.
  – This is what we are interested in here.

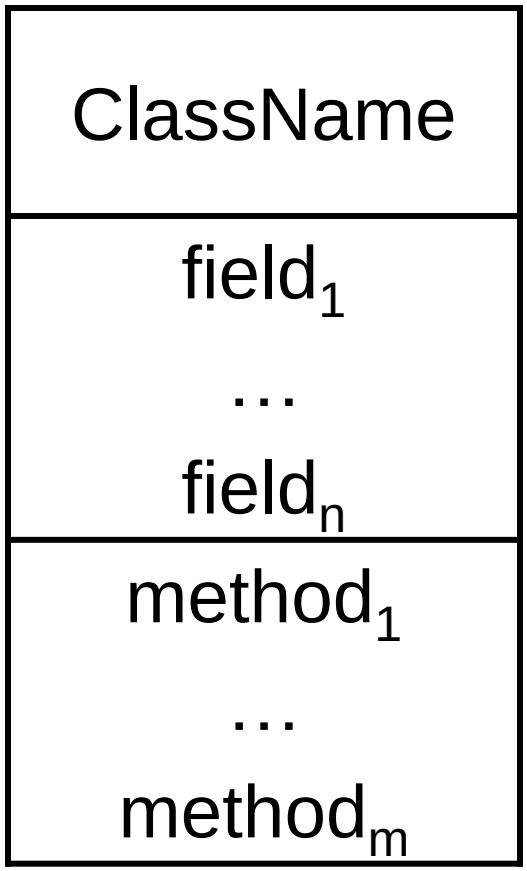
Building Blocks of UML

- Things -- abstraction
- Relations -- tie things together
- Diagrams -- group interesting collections of things

Objects and Classes

	Interpretation in the Real World	Interpretation in the Model
Object	An object is a thing that can be distinctly identified.	An object has an identity, a state, and a behavior.
Class	A class represents a set of objects with similar characteristics and behavior. This objects are called the instances of the class.	A class characterizes the structure of states and behaviors that are shared by all instances.

UML Notation for Classes



The top compartment shows the class name.

The middle compartment contains the declarations of the fields of the class.

The bottom compartment contains the declarations of the methods

Field Declaration

- The name of the field is required in the field declaration.
- Field declaration may include:

```
[Visibility]Name[[Multiplicity]][:Type][=InitialValue]
```

- Visibility or accessibility defines the scope:
  - Public -- the feature is accessible to any class
  - Protected -- the feature is accessible to the class itself, all the classes in the same package, and all its subclasses.
  - Package -- the feature is accessible to the class itself and all classes in the same package.
  - Private -- the feature is only accessible within the class itself.

Visibility syntax in Java and UML

Visibilty	Java Syntax	UML Syntax
public	public	+
protected	protected	#
package		~
private	private	-

Examples

Java Syntax	UML Syntax
Date birthday	Birthday:Date
Public int duration = 100	+duration:int = 100
Private Student students[0..MAX_Size]	-students[0..MAX_Size]:Student

Method Declaration

- The name of the method is required in the method declaration.
- Method declaration may include:

```
[Visibility]Name([Parameter, ...])[:Type]
```

Examples

Java Syntax	UML Syntax
void move(int dx, int dy)	~move(dx:int, dy:int): void
public int getSize()	+ getSize():int

Example

Class name: Point

Fields: x, y

Method: move

```
class Point {
  int x, y;
  public void move
   (int dx, int dy){
   // implementation
  }
```

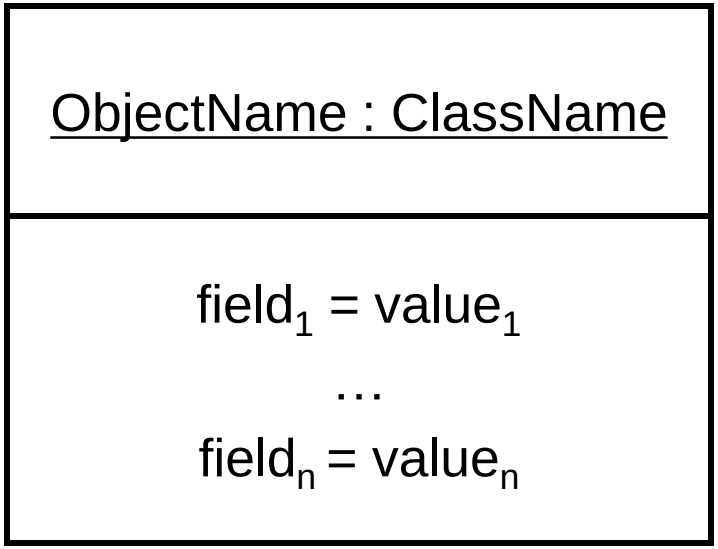
Example

Point
private int x private int y
public void move(int dx,int dy)

UML Standard

Point
-x:int -y:int
+move(dx:int,dy:int)

UML Notation for Object



The top compartment shows the object name and its class.

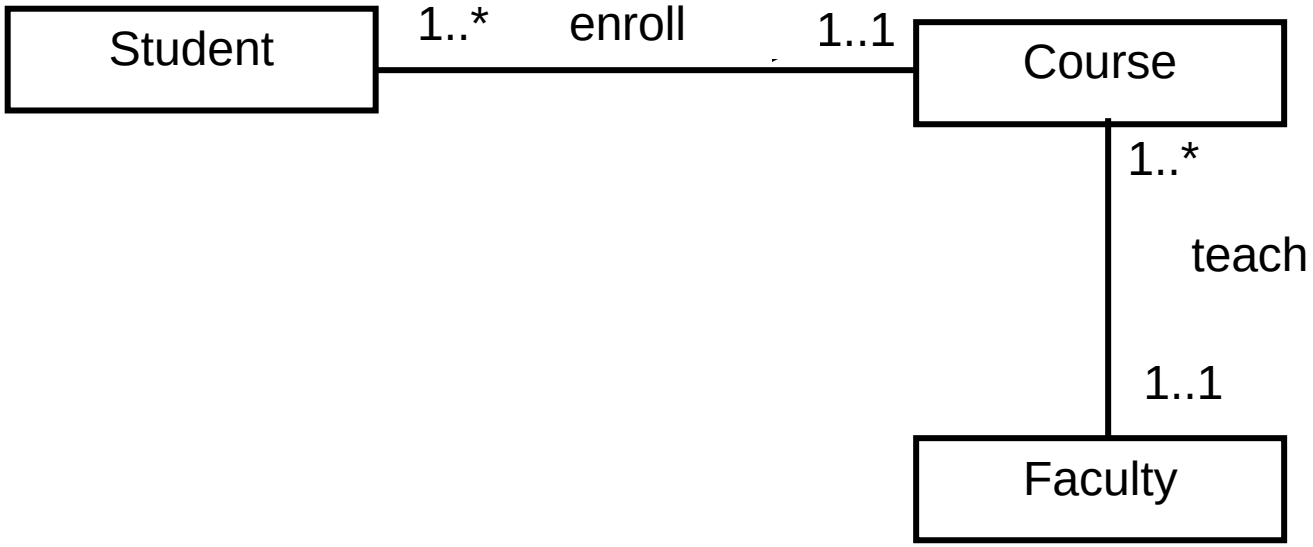
The bottom compartment contains a list of the fields and their values.

Example

P1:Point	Point p1 = new Point();
x = 0 y = 0	p1.x = 0; P1.y = 0;

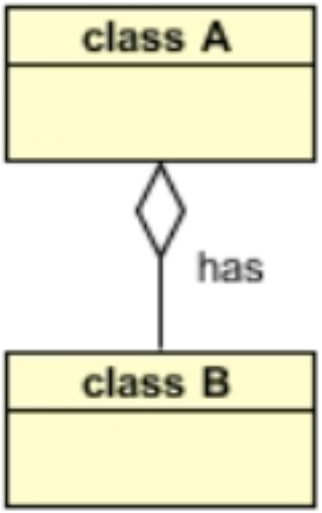
Association

- Solid line that connects two classes represents an association
  - numbers near end of each line are multiplicity values

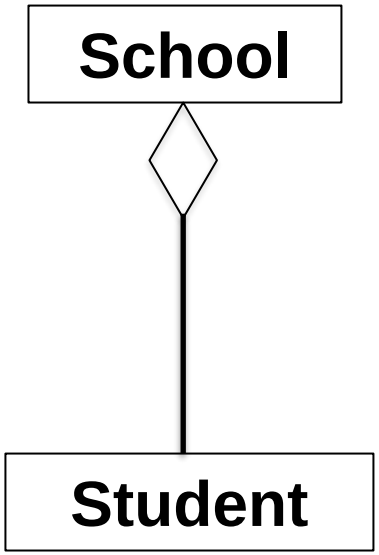


Aggregation

- It is a special form of association. It represents **Has-A** relationship.
- It is a unidirectional association i.e. a one way relationship.
- Both the entries can survive individually which means ending one entity will not effect the other entity

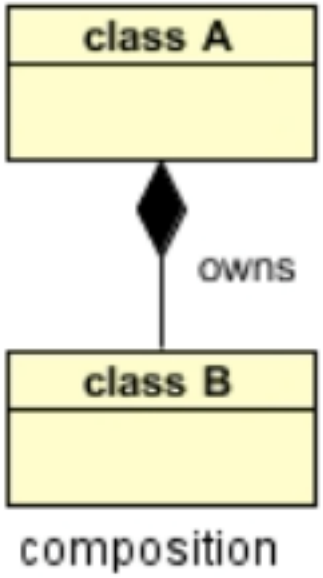


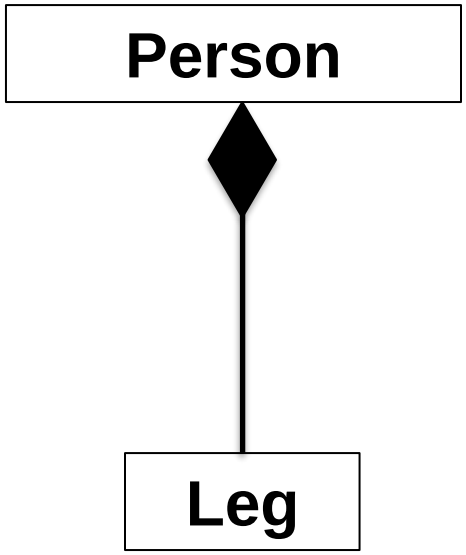
aggregation



Composition

- The composition link shows that a class (container, whole) has exclusive ownership over other class/s (parts).
- It represents **owns** relationship
- It implies a relationship where the child cannot exist independent of the parent.





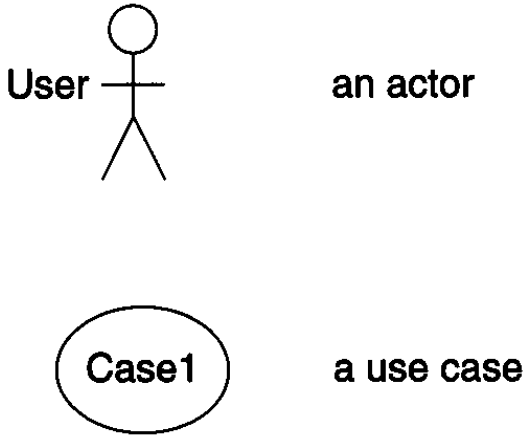
When the Person object is destroyed the Leg object get destroyed with it.

Modeling Requirements with Use Cases

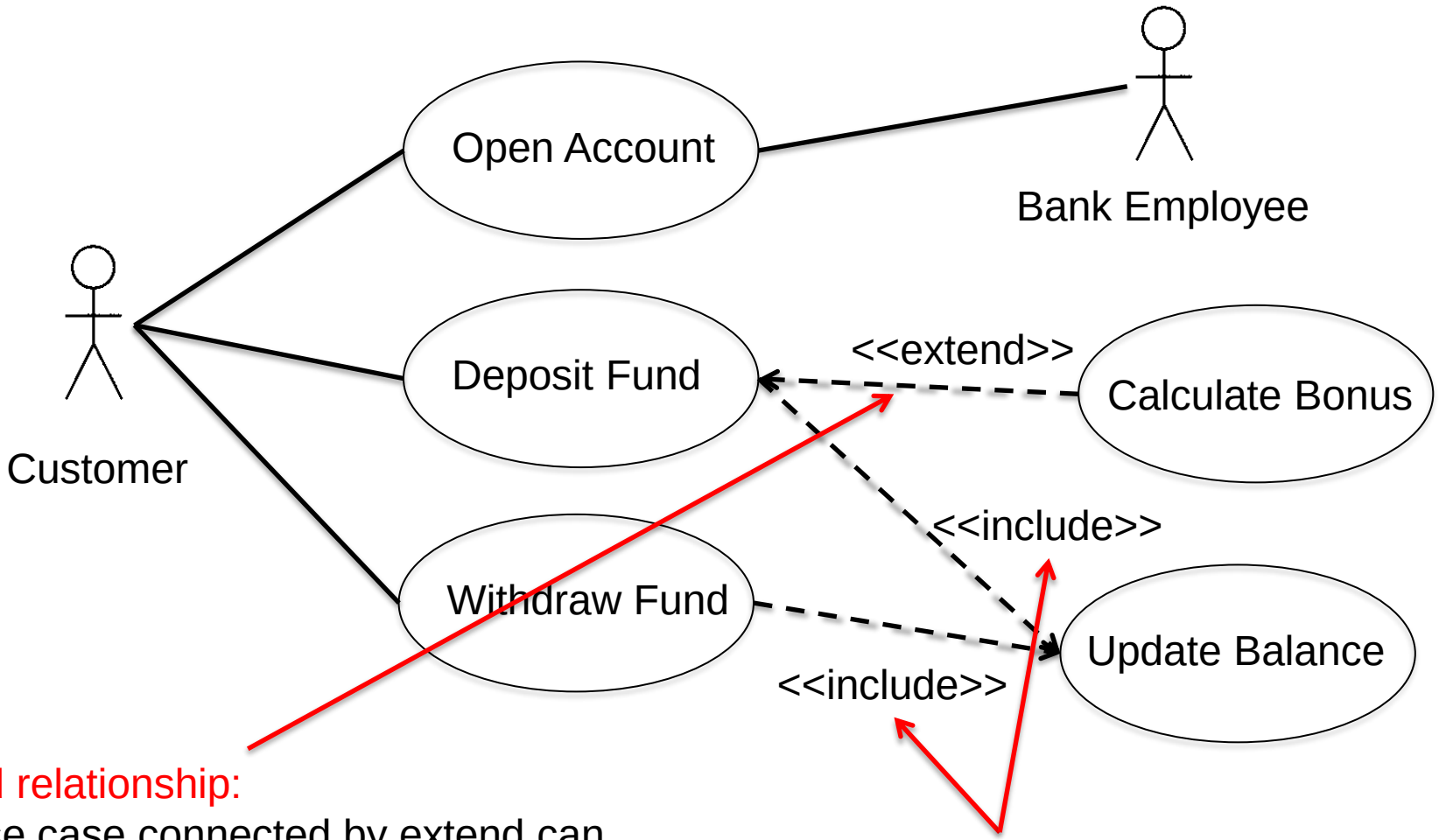
- Use cases describes the externally observable behavior of system functions in the form of interactions between the system to be developed and the external entities -- *Actors*.
- Actors may represent roles played by users of the system or other systems.

Use Case Diagram

- Consist of:
  - Use cases
  - Actors
  - Associations between actors and use cases are indicated in use case diagrams by solid lines



Example



Extend relationship:
The use case connected by extend can supplement the base use case

Include relationship:
It indicates that the use case to which the arrow points is included in the use case on the other side of the arrow

Summary

- Recap on Objects and Classes
- Static and non-static
- Packages and APIs
- Introduction to UML