

---

CS1020 Lecture Note #4:

# **Object Oriented Programming (OOP) Part 2 – Designer Mode**

---

*Creating our own classes*

# Objectives

## Programming model and OOP

- Using object-oriented modeling to formulate solution

## Creating our own classes

- Determining what services to provide for a class

## Unified Modeling Language (UML)

- Graphic representation of OOP components

# References



## Textbook

- Chapter 2: Section 2.2 (pg 119 – 130), Section 2.3 (pg 131 – 150)



CS1020 website →  
Resources → Lectures

- [http://www.comp.nus.edu.sg/~cs1020/2\\_resources/lectures.html](http://www.comp.nus.edu.sg/~cs1020/2_resources/lectures.html)

# Outline (1/2)

1. Recapitulation
2. Programming Model and OOP
  - 2.1 Procedural vs OOP
  - 2.2 Illustration: Bank Account
3. OOP Design
  - 3.1 Designing Own Classes
  - 3.2 Bank Account: BankAcct class
  - 3.3 Accessors and Mutators
  - 3.4 Writing Client Class

# Outline (2/2)

## 4. More OOP Concepts

4.1 Class and Instance members

4.2 MyBall class: Draft

4.3 “this” reference

4.4 Using “this” in Constructors

4.5 Overriding Methods: toString() and equals()

4.6 MyBall class: Improved

## 5. Unified Modeling Language (UML)

# 1. Recapitulation

- We revisited a few classes ([Scanner](#), [String](#), [Math](#)) and learnt a few new ones ([DecimalFormat](#), [Random](#), wrapper classes, [Point](#))
- We discussed some basic OOP features/concepts such as **modifiers**, **class and instance methods**, **constructors** and **overloading**.
- Last week, we used classes provided by API as a user.
- Today, we become designers to *create* our own classes!

---

## **2. Programming Model and OOP**

---

World View of a Programming Language

# Programming Model

- All programming languages like C, C++, Java, etc. have an underlying **programming model** (or **programming paradigm**):
  - ❑ How to organize the information and processes needed for a solution (program)
  - ❑ Allows/facilitates a certain way of thinking about the solution
  - ❑ Analogy: it is the “**world view**” of the language
- Various programming paradigms:
  - ❑ **Procedural/Imperative**: C, Pascal
  - ❑ **Object Oriented**: Java, C++
  - ❑ **Functional**: Scheme, LISP
  - ❑ **Logic programming**: PROLOG
  - ❑ others



# Hello World!

Pascal

```
Program HelloWorld;  
Begin  
  WriteLn('Hello World!');  
End.
```

Java

```
public class HelloWorld {  
  public static void main(String[] args) {  
    System.out.println("Hello World!");  
  }  
}
```

LISP

```
(defun Hello-World ()  
  (print (list 'Hello 'World!)))
```

Prolog

```
go :-  
  writeln('Hello World!').
```

# Procedural (eg: C) versus OOP (eg: Java)

## Procedural/Imperative

- View program as a process of transforming data
- Data and associated functions are separated
- Data is publicly accessible to everyone

## OOP

- Encapsulation
- Inheritance
- Abstraction
- Polymorphism

### **Advantages**

- Resembles execution model of computer
- Less overhead when designing

### **Disadvantages**

- Harder to understand as logical relation between data and functions is unclear
- Hard to maintain
- Hard to extend/expand

# OOP

## 4 fundamental OOP concepts

- **Encapsulation**

- Bundling data and associated functionalities
- Hide internal details and restricting access

Today's  
focus

- **Inheritance**

- Deriving a class from another, affording code reuse

- **Abstraction**

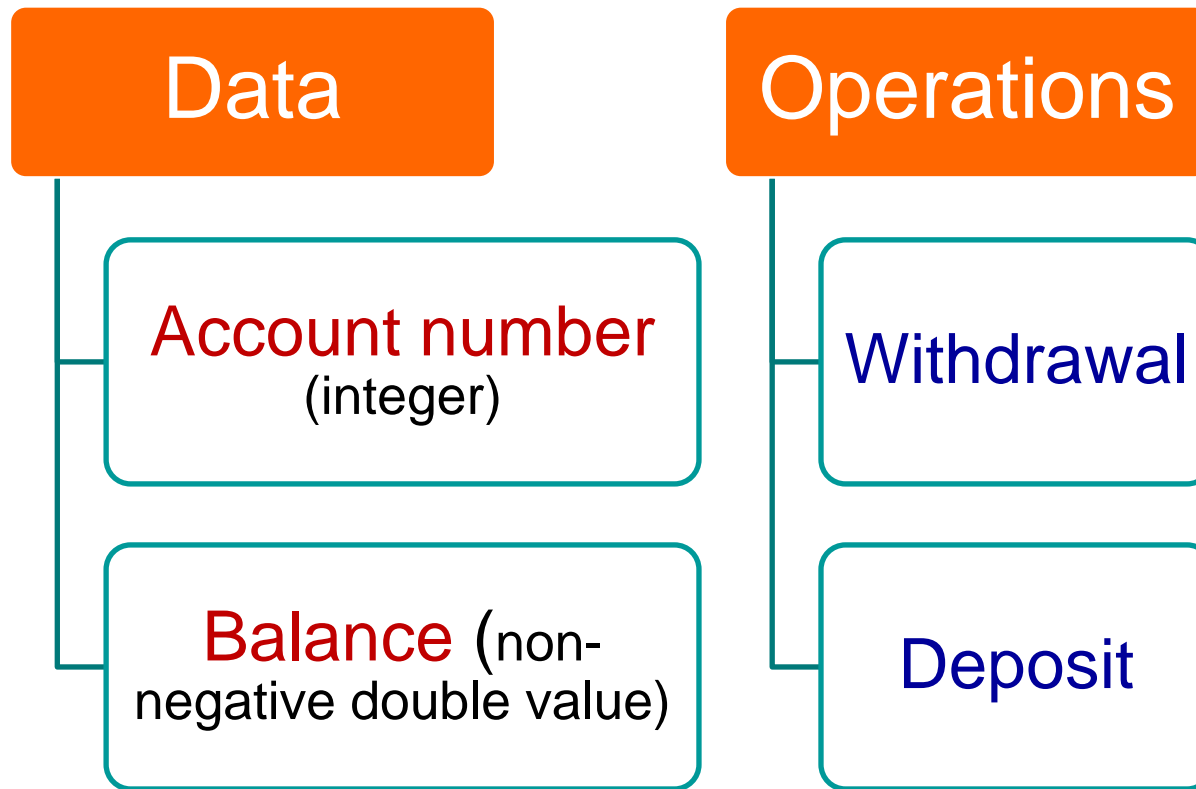
- Hiding the complexity of the implementation
- Focusing on the specifications and not the implementation details

- **Polymorphism**

- Behavior of functionality changes according to the actual type of data

# Illustration: Bank Account

- *(Note: This illustration serves as a quick comparison between a procedural language and an object-oriented language; it is not meant to be comprehensive.)*



# Bank Account (C implementation) (1/4)

```
typedef struct {  
    int acctNum;  
    double balance;  
} BankAcct;
```

Structure to  
hold data

```
void initialize(BankAcct *baPtr, int anum) {  
    baPtr->acctNum = anum;  
    baPtr->balance = 0;  
}
```

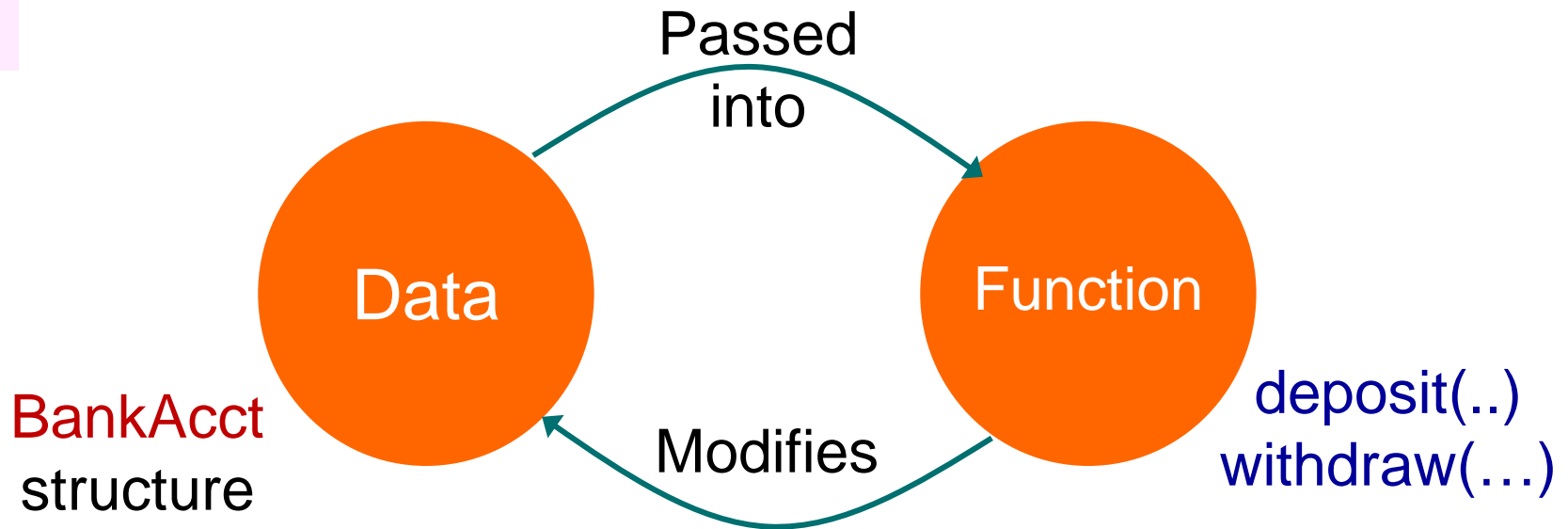
Functions to  
provide basic  
operations

```
int withdraw(BankAcct *baPtr, double amount) {  
    if (baPtr->balance < amount)  
        return 0; // indicate failure  
    baPtr->balance -= amount;  
    return 1; // indicate success  
}
```

```
void deposit(BankAcct *baPtr, double amount)  
{ ... Code not shown ... }
```

## Bank Account (C implementation) (2/4)

- In C, the data (structure) and operations (functions) are treated as **separate entities**:



# Bank Account (C implementation) (3/4)

Correct use  
of `BankAcct`  
and its  
operations

```
BankAcct ba1;  
  
initialize(&ba1, 12345);  
deposit(&ba1, 1000.50);  
withdraw(&ba1, 500.00);  
withdraw(&ba1, 600.00);  
...
```

Wrong and  
malicious  
exploits of  
`BankAcct`

```
BankAcct ba1;  
  
deposit(&ba1, 1000.50);  
  
initialize(&ba1, 12345);  
ba1.acctNum = 54321;  
  
ba1.balance = 10000000.00;  
...
```

Forgot to initialize

Account Number  
should not change!

Balance should be  
changed by  
authorized  
operations only

## Bank Account (C implementation) (4/4)

- Characteristics of a procedural language
  - View program as a process of transforming data
  - Data and associated functions are separated
    - Requires good programming discipline to ensure good organization in a program
  - Data is publicly accessible to everyone (!)
    - Potentially vulnerable to unauthorised or uncontrolled access/modification

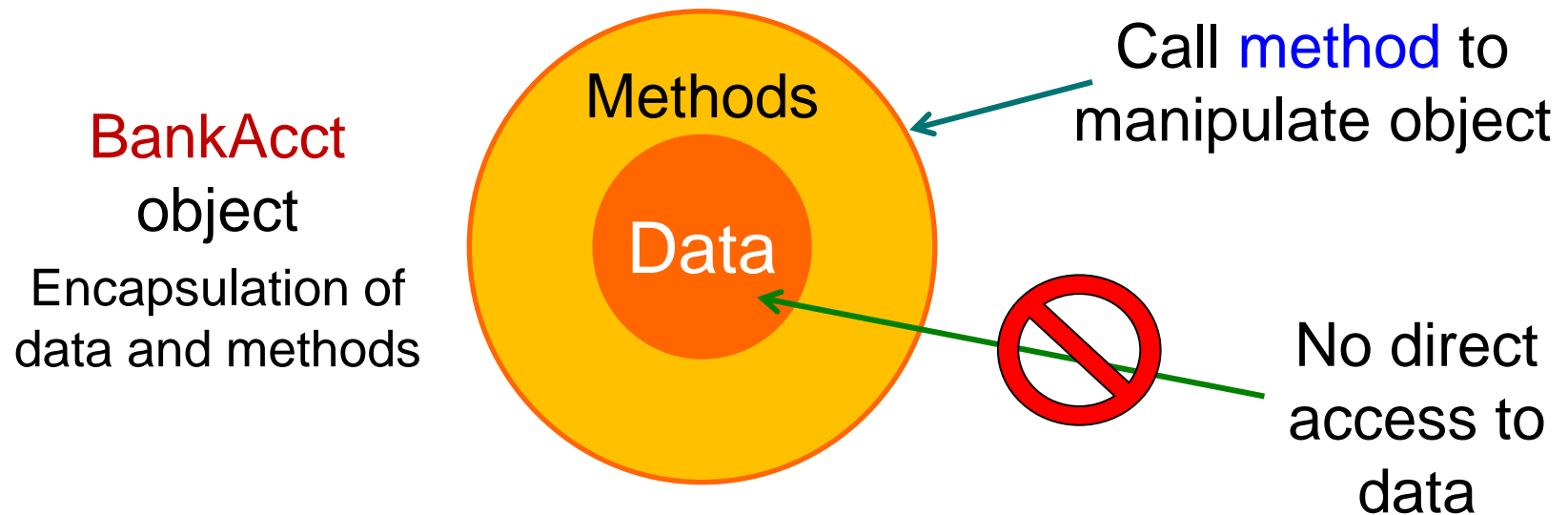


# Bank Account (OO implementation) (1/2)

- Characteristics of an OOP language
  - View program as a collection of objects
    - Computation is performed through interaction with the objects
  - Each object has data attributes and a set of functionalities (behaviours)
    - Functionalities are generally exposed to the public...
    - While data attributes are generally kept within the object, hidden from and inaccessible to the public

## Bank Account (OO implementation) (2/2)

- A conceptual view of an OO implementation for Bank Account



# Procedural (eg: C) versus OOP (eg: Java)

## Procedural/Imperative

- View program as a process of transforming data
- Data and associated functions are separated
- Data is publicly accessible to everyone

### Advantages

- Resembles execution model of computer
- Less overhead when designing

### Disadvantages

- Harder to understand as logical relation between data and functions is unclear
- Hard to maintain
- Hard to extend/expand

## OOP

- Encapsulation
- Inheritance
- Abstraction
- Polymorphism

### Advantages

- Easier to design as it resembles real world
- Easier to maintain as modularity is enforced
- Extensible

### Disadvantages

- Less efficient in execution
- Longer code with higher design overhead

## **3. OOP Design**

---

Designing Your Own Class

## Designing Own Classes (1/7)

- Previously, we studied classes provided by Java API (Scanner, String, Math, Point, etc.)
- These are **service classes**, where each class provides its own functionalities through its methods.
- We then wrote application programs (such as TestMath.java, TestPoint.java) to use the services of one or more of these classes. Such application programs are **client classes** or **driver classes** and they must contain a main() method.

## Designing Own Classes (2/7)

- We were in user mode.
- Now, we are in designer mode to **create our own (service) classes**, so that we (or other users) may write client classes to use these service classes.
- We will see some of the OOP concepts covered before (eg: class and instance methods, constructors, overloading, attributes) and also learn new concepts.

## Designing Own Classes (3/7)

- What is the purpose of a (service) class?

A template to create instances (objects) out of it.

- What does a (service) class comprise?



All instances (objects) of the same class are independent entities that possess the same set of attributes and behaviours.

## Designing Own Classes (4/7)

- **Attributes** are also called **Member Data**, or **Fields** (in Java API documentation)
- **Behaviours** (or **Member Behaviours**) are also called **Methods** (in Java API documentation)
- Attributes and members can have different level of **accessibilities/visibilities** (next slide)
- Each class has one or more **constructors**
  - ❑ To create an instance of the class
  - ❑ **Default constructor** has no parameter and is automatically generated by compiler if class designer does not provide any constructor.
  - ❑ Non-default constructors are added by class designer
  - ❑ Constructors can be overloaded



## Designing Own Classes (5/7)

**public**

- Anyone can access
- Usually intended for methods only

**private**

- Can be assessed by the same class
- Recommended for all attributes

**protected**

- Can be assessed of the same class or its child classes can access it AND
- Can be assessed by the classes in the same **Java package** (not covered)
- Recommended for attributes/methods that are common in a “family”

**[None]  
(default)**

- Only accessible to classes in the same **Java package** (not covered)
- Known as the **package private visibility**

# Designing Own Classes (6/7)

- Some general guidelines...
- **Attributes** are usually **private**
  - Information hiding, to shield data of an object from outside view
  - Instead, we provide public methods for user to access the attributes through the public methods
  - There are exceptions. Example: **Point** class has public attributes **x** and **y**, most likely due to legacy reason.
- **Methods** are usually **public**
  - So that they are available for users
    - Imagine that the methods in **String** class and **Math** class are private instead, then we cannot even use them!
  - If the methods are to be used internally in the service class itself and not for users, then the methods should be declared private instead

# Bank Account: BankAcct Class (1/2)

BankAcct.java

```
class BankAcct {
```

```
    private int acctNum;  
    private double balance;
```

Attributes of BankAcct

```
    // Default constructor
```

```
    public BankAcct() {
```

```
        // By default, numeric attributes  
        // are initialised to 0
```

```
    }
```

```
    public BankAcct(int aNum, double bal) {
```

```
        // Initialize attributes with user  
        // provided values
```

```
        acctNum = aNum;  
        balance = bal;
```

```
    }
```

```
    // Other methods on next slide
```

**Constructors:**  
Name must be identical to class name.  
No return type.

Can be **overloaded**.

# Bank Account: BankAcct Class (2/2)

BankAcct.java

```
public int getAcctNum() { return acctNum; }

public double getBalance() { return balance; }

public boolean withdraw(double amount) {
    if (balance < amount) return false;
    balance -= amount;
    return true;
}

public void deposit(double amount) {
    if (amount <= 0) return;
    balance += amount;
}

public void print() {
    System.out.println("Account number: " + getAcctNum());
    System.out.printf("Balance: $%.2f\n", getBalance());
}
}
```

# Accessors and Mutators

- Note that for service class, we use the **default** visibility for the class (i.e. no modifier before the class name)
- Besides constructors, there are two other types of special methods that can be referred to as **accessors** and **mutators**.
- An **accessor** is a method that accesses (retrieves) the value of an object's attribute
  - Eg: `getAcctNum()`, `getBalance()`
  - Its return type must match the type of the attribute it retrieves
- A **mutator** is a method that mutates (modifies) the value of an object's attribute
  - Eg: `withdraw()`, `deposit()`
  - Its return type is usually **void**, and it usually takes in some argument to modify the value of an attribute

# Designing Own Classes (7/7)

- As a (service) class designer, you decide the following:
  - What attributes you want the class to have
  - What methods you want to provide for the class so that users may find them useful
  - For example, the `print()` method is provided for `BankAcct` as the designer feels that it might be useful. Or, add a `transfer()` method to transfer money between 2 accounts?
- As in any design undertaking, there are no hard and fast rules. One approach is to study the classes in the API documentation to learn how others designed the classes, and google to explore.
- You need to practise a lot and ask questions.

## Writing Client Class – User Mode

- Note that there is no `main()` method in `BankAcct` class because it is a service class, not a client class (application program). You cannot execute `BankAcct`.
- So how do we write a client class to make use of `BankAcct`?
- You have written a number of client classes in the past weeks. These classes contain the `main()` method.
- In general, the service class and the client class may be put into a single `.java` program, mostly for quick testing. (However, there can only be 1 public class in such a program, and the public class name must be identical to the program name.)
- We will write 1 class per `.java` program here (most of the time) to avoid confusion.

# Client Class: TestBankAcct

TestBankAcct.java

```
public class TestBankAcct {  
  
    public static void main(String[] args) {  
  
        BankAcct ba1 = new BankAcct();  
        BankAcct ba2 = new BankAcct(1234, 321.70);  
  
        System.out.println("Before transactions:");  
        ba1.print();  
        ba2.print();  
  
        ba1.deposit(1000);  
        ba1.withdraw(200.50);  
        ba2.withdraw(500.25);  
  
        System.out.println();  
        System.out.println("After transactions:");  
        ba1.print();  
        ba2.print();  
    }  
}
```

Which **constructor** is used?

Before transactions:  
Account number:  
Balance:  
Account number:  
Balance:

After transactions:  
Account number:  
Balance:  
Account number:  
Balance:



# What happens if...

```
public class TestBankAcct {  
    public static void main(String[] args) {  
        BankAcct bal = new BankAcct();  
  
        /* Instead of  
        bal.deposit(1000);  
        */  
  
        bal.balance += 1000;  
    }  
}
```

Compilation error!  
**balance has private access in BankAcct**

The above code works only if **balance** is declared as a **public** attribute in **BankAcct**. (But we don't want that.)

# Compiling Classes

- `BankAcct.java` and `TestBankAcct.java` can be compiled independently.
- Only `TestBackAcct` class can be executed.

```
javac BankAcct.java  
javac TestBankAcct.java  
java TestBankAcct
```

- We say `TestBankAcct` **uses** or **depends on** `BankAcct`.
- We can write many clients that depend on the same service class. (Eg: Many client programs you have seen depend on the `Scanner` service class.)
- Likewise, a client may also depend on more than one service class. (Eg: `TestMath` in lecture #1 depends on both `Scanner` and `Math` service classes.)

---

## **4. More OOP Concepts**

---

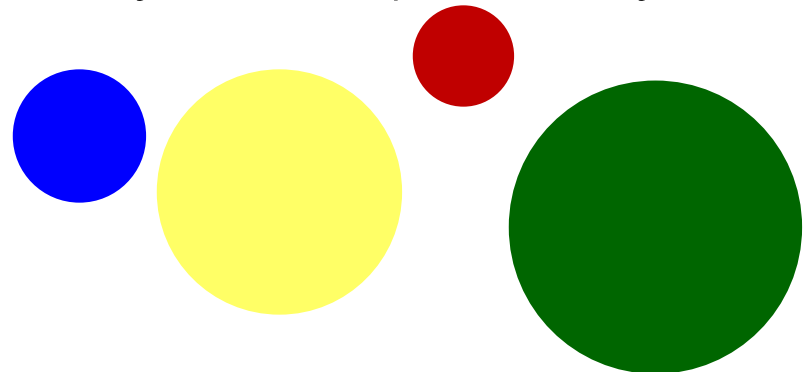
# Class and Instance members

- A class comprises 2 types of members: **attributes** (data members) and **methods** (behaviour members)
- Java provides the modifier **static** to indicate if the member is a **class member** or an **instance member**

	Attribute	Method
static	Class attribute	Class method
default	Instance attribute	Instance method

# Designing MyBall Class (1/2)

- Let's create a new class called **MyBall**
  - Obviously, we want to create ball objects out of it
- Let's start with something simple, and add more complexity gradually.
- We may start with 2 **instance attributes**:
  - **Colour** of the ball, which is a string (e.g.: "blue", "yellow")
  - **Radius** of the ball, which is of type double (e.g.: 6.5, 12.8)
  - These are instance attributes because each **MyBall** object created has its own attribute values (i.e. colour and radius)
- Some **MyBall** instances we may create (well, they look like circles on the screen):



## Designing MyBall Class (2/2)

- Sometimes, we want to have some **class attributes** in a class, shared by all instances (objects) of that class
- Let's have one **class attribute** for illustration purpose
  - The number of **Myball** objects created in a program run
- Next, for behaviours, a class in general consists of at least these 3 types of methods
  - **Constructors**: to create an instance. Usually there are overloaded constructors. **Default constructor** has no parameter, and is automatically provided by the compiler if there is no constructor present in the class, and all numeric attributes are initialised to 0 and object attributes initialised to NULL.
  - **Accessors**: to access (retrieve) values of the attributes
  - **Mutators**: to mutate (modify) values of the attributes

# MyBall Class: Draft (1/2)

MyBall\_draft/MyBall.java

```
class MyBall {
    /***** Data members *****/
    private static int quantity = 0;
    private String colour;
    private double radius;

    /***** Constructors *****/
    // Default constructor creates a yellow, radius 10.0 ball
    public MyBall() {
        setColour("yellow");
        setRadius(10.0);
        quantity++;
    }

    public MyBall(String newColour, double newRadius) {
        setColour(newColour);
        setRadius(newRadius);
        quantity++;
    }
}
```

# MyBall Class: Draft (2/2)

MyBall\_draft/MyBall.java

```
/****** Accessors *****/
public static int getQuantity() {
    return quantity;
}

public String getColour() {
    return colour;
}

public double getRadius() {
    return radius;
}

/****** Mutators *****/
public void setColour(String newColour) {
    colour = newColour;
}

public void setRadius(double newRadius) {
    radius = newRadius;
}
}
```

Class method

The rest are all  
instance methods.



# Testing MyBall: TestBallV1 (1/2)

MyBall\_draft/TestBallV1.java

```
import java.util.*;
public class TestBallV1 {
    public static void main(String[] args) {
        String inputColour;
        double inputRadius;
        Scanner sc = new Scanner(System.in);

        // Read ball's input and create a ball object
        System.out.print("Enter colour: "); inputColour = sc.next();
        System.out.print("Enter radius: "); inputRadius = sc.nextDouble();
        MyBall myBall1 = new MyBall(inputColour, inputRadius);
        System.out.println();

        // Read another ball's input and create another ball object
        System.out.print("Enter colour: "); inputColour = sc.next();
        System.out.print("Enter radius: "); inputRadius = sc.nextDouble();
        MyBall myBall2 = new MyBall(inputColour, inputRadius);
        System.out.println();

        System.out.println(MyBall.getQuantity() + " balls are created.");
        System.out.println("1st ball's colour and radius: "
            + myBall1.getColour() + ", " + myBall1.getRadius());
        System.out.println("2nd ball's colour and radius: "
            + myBall2.getColour() + ", " + myBall2.getRadius());
    }
}
```

constructor

Calling a class method

Calling instance methods

# Testing MyBall: TestBallV1 (2/2)

```
import java.util.*;
public class TestBallV1 {
    public static void main(String[] args) {
        String inputColour;
        double inputRadius;
        Scanner sc = new Scanner(System.in);

        // Read ball's input and create a ball object
        System.out.print("Enter colour: "); inputColour = sc.next();
        System.out.print("Enter radius: "); inputRadius = sc.nextDouble();
        MyBall myBall1 = new MyBall(inputColour, inputRadius);
        System.out.println();

        // Read another ball's input and create another ball object
        System.out.print("Enter colour: "); inputColour = sc.next();
        System.out.print("Enter radius: "); inputRadius = sc.nextDouble();
        MyBall myBall2 = new MyBall(inputColour, inputRadius);
        System.out.println();

        System.out.println(MyBall.getQuantity() + " balls are created.");
        System.out.println("1st ball's colour and radius: "
            + myBall1.getColour() + ", " + myBall1.getRadius());
        System.out.println("2nd ball's colour and radius: "
            + myBall2.getColour() + ", " + myBall2.getRadius());
    }
}
```

Enter colour: red

Enter radius: 1.2

Enter colour: blue

Enter radius: 3.5

2 balls are created.

1st ball's colour and radius: red, 1.2

2nd ball's colour and radius: blue, 3.5

# Modularising TestBallV1

- You may have noticed that the codes for reading and construction a **MyBall** object are duplicated in **TestBallV1.java**
- We can modularise the program by creating a method **readBall()** to perform this task, which can then be called as many times as necessary
- We name this modified program **TestBallV2.java**, shown in the next slide
- Changes in the client program do not affect the services defined in the service class **MyBall**

# Testing MyBall: TestBallV2

MyBall\_draft/TestBallV2.java

```
import java.util.*;
public class TestBallV2 {
    // This method reads ball's input data from user, creates
    // a ball object, and returns it to the caller.
    public static MyBall readBall(Scanner sc) {
        System.out.print("Enter colour: ");
        String inputColour = sc.next();
        System.out.print("Enter radius: ");
        double inputRadius = sc.nextDouble();
        return new MyBall(inputColour, inputRadius);
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        MyBall myBall1 = readBall(sc); // Read input and create ball object
        System.out.println();

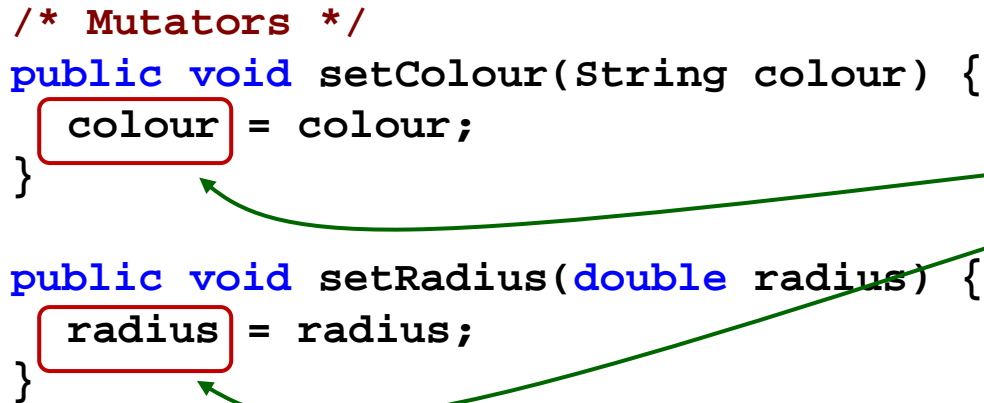
        MyBall myBall2 = readBall(sc); // Read input and create another ball object
        System.out.println();

        System.out.println(MyBall.getQuantity() + " balls are created.");
        System.out.println("1st ball's colour and radius: "
            + myBall1.getColour() + ", " + myBall1.getRadius());
        System.out.println("2nd ball's colour and radius: "
            + myBall2.getColour() + ", " + myBall2.getRadius());
    }
}
```

# “this” reference (1/4)

- What if the parameter of a method (or a local variable) has the same name as the data attribute?

```
/* Mutators */  
public void setColour(String colour) {  
    colour = colour;  
}  
  
public void setRadius(double radius) {  
    radius = radius;  
}
```

The diagram shows the original code with the parameter names 'colour' and 'radius' boxed in red. Two green arrows originate from these boxes: one points to the 'colour' parameter in the callout box, and the other points to the 'radius' parameter in the callout box.

These methods will **not** work, because **colour** and **radius** here refer to the parameters, not the data attributes.

The original code:

```
public void setColour(String newColour) {  
    colour = newColour;  
}  
public void setRadius(double newRadius) {  
    radius = newRadius;  
}
```

# “this” reference (2/4)

- A common confusion:



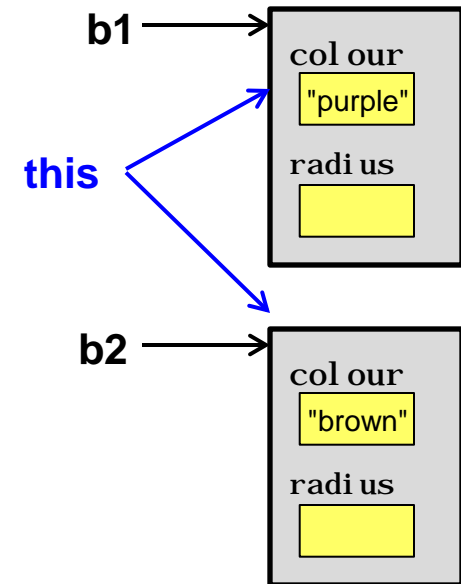
```
// b1 and b2 are MyBall objects  
b1.setColour("purple");  
b2.setColour("brown");
```

- How does the method “know” which is the “object” it is currently communicating with? (Since there could be many objects created from that class.)

- Whenever a method is called,

- a **reference to the calling object** is set automatically
- Given the name “**this**” in Java, meaning “*this particular object*”


- All attributes/methods are then accessed implicitly through this reference




## “this” reference (3/4)

- The “**this**” reference can also be used to solve the ambiguity in the preceding example where the parameter is identical to the attribute name

```
/* Mutators */  
public void setColour(String colour) {  
    colour = colour;  
}  
  
public void setRadius(double radius) {  
    radius = radius;  
}
```



```
/* Mutators */  
public void setColour(String colour) {  
    this.colour = colour;  
}  
  
public void setRadius(double radius) {  
    this.radius = radius;  
}
```



attributes

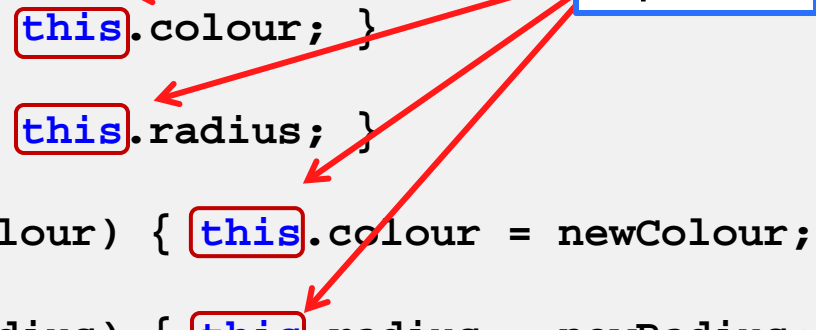


parameters



# “this” reference (4/4)

- The “this” is optional for unambiguous case



```
public String getColour() { return this.colour; }  
public double getRadius() { return this.radius; }  
public void setColour(String newColour) { this.colour = newColour; }  
public void setRadius(double newRadius) { this.radius = newRadius; }
```

- The use of “this” reference below is wrong. Why?

```
public static int getQuantity() { return this.quantity; }
```





# Naming Convention for Attributes

- Some suggested that object's attributes be named with a prefix “\_” (or “m\_”) or a suffix “\_” to distinguish them from other variables/parameters.
- This would avoid the need of using “**this**” as there would be no ambiguity

```
class MyBall {  
    /***** Data members *****/  
    private static int _quantity = 0;  
    private String _colour;  
    private double _radius;  
    . . .  
}
```

- Some also proposed that “**this**” should be always written even for unambiguous cases
- We will leave this to your decision. Important thing is that you should be consistent.

# Code Reuse

- In our draft `MyBall` class, the following is done:

```
public MyBall() {  
    setColour("yellow");  
    setRadius(10.0);  
    quantity++;  
}  
public MyBall(String newColour, double newRadius) {  
    setColour(newColour);  
    setRadius(newRadius);  
    quantity++;  
}
```

- What about this? Does

```
public MyBall() {  
    colour = "yellow";  
    radius = 10.0;  
    quantity++;  
}  
public MyBall(String newColour, double newRadius) {  
    colour = newColour;  
    radius = newRadius;  
    quantity++;  
}
```

- Both work, but the top version follows the principle of **code reuse** which minimises code duplication, but is slightly less efficient.
- The top version would be superior if the methods `setColour()` and `setRadius()` are long and complex. In this case, the two versions make little difference.

# Using “this” in Constructors (1/2)

- Still on code reusability, and another use of “this”.
- Our draft `MyBall` class contains these two constructors:

```
public MyBall() {  
    setColour("yellow");  
    setRadius(10.0);  
    quantity++;  
}
```

```
public MyBall(String newColour, double newRadius) {  
    setColour(newColour);  
    setRadius(newRadius);  
    quantity++;  
}
```

Recall that this is called **overloading**

- Note that the logic in both constructors are essentially the same (i.e. change the `colour` and `radius`, and increment the quantity)

## Using “this” in Constructors (2/2)

- To reuse code, we can use “**this**” in a constructor to call another constructor:

```
public MyBall() {  
    this("yellow", 10.0);  
}
```

```
public MyBall(String newColour, double newRadius) {  
    setColour(newColour);  
    setRadius(newRadius);  
    quantity++;  
}
```

*Restriction:* Call to “**this**” must be the **first** statement in a constructor.

- When we instantiate a **MyBall** object in a client program using the default constructor:

```
MyBall b1 = new MyBall();
```

- It calls the default constructor, which in turn calls the second constructor to create a **MyBall** object with **colour** “yellow” and **radius** 10.0, and increment the quantity.

# Overriding Methods

- We will examine two common services (methods) expected of every class in general
  - To **display** the values of an object's attributes
  - To **compare** two objects to determine if they have identical attribute values
- This brings on the issue of **overriding methods**

# Printing an Object: toString() (1/3)

- In `TestBallV2.java`, we display individual attributes (`colour` and `radius`) of a `MyBall` object.
- Suppose we print a `MyBall` object as a whole unit in `TestBallV3.java`:

`MyBall_draft/TestBallV3.java`

```
import java.util.*;
public class TestBallV3 {
    // readBall() method omitted

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        MyBall myBall1 = readBall(sc); // Read input
        System.out.println();

        MyBall myBall2 = readBall(sc); // Read input
        System.out.println();

        System.out.println("1st ball: " + myBall1);
        System.out.println("2nd ball: " + myBall2);
    }
}
```

Enter colour: `red`

Enter radius: `1.2`

Enter colour: `blue`

Enter radius: `3.5` ject

1st ball: Ball@471e30

2nd ball: Ball@10ef90c

Object identifiers  
(OIDs)

# Printing an Object: toString() (2/3)

- How do you get a custom-made output like this?

```
1st ball: [red, 1.2]  
2nd ball: [blue, 3.5]
```

- To do that, you need to add a `toString()` method in the `MyBall` class
  - ▣ The `toString()` method returns a string, which is a string representation of the data in an object (up to you to format the string to your desired liking)

```
class MyBall {  
    // original code omitted  
  
    public String toString() {  
        return "[" + getColour() + ", " + getRadius() + "];"  
    }  
}
```

MyBall\_draft/MyBall.java

# Printing an Object: toString() (3/3)

- After `toString()` method is added in `MyBall.java`, a client program can use it in either of these ways:

```
System.out.println(myBall1);
```

```
System.out.println(myBall1.toString());
```



## Object class and inherited methods (1/2)

- Why did we call the preceding method `toString()` and not by other name?
- All Java classes are implicitly *subclasses* of the class `Object`
- `Object` class specifies some basic behaviours common to all kinds of objects, and hence these behaviours are inherited by its subclasses
- Some inherited methods from the `Object` class are:
  - `toString()` method: to provide a string representation of the object's data
  - `equals()` method: to compare two objects to see if they contain identical data
- However, these inherited methods usually don't work (!) as they are not customised

## Object class and inherited methods (2/2)

- Hence, we often (almost always) need to customise these inherited methods for our own class
- This is called **overriding**
- We have earlier written an overriding method **toString()** for **MyBall** class
- We shall now write an overriding method **equals()** for **MyBall** class
- The **equals()** method in **Object** class has the following header, hence our overriding method must follow the same header: (if we don't then it is not overriding)

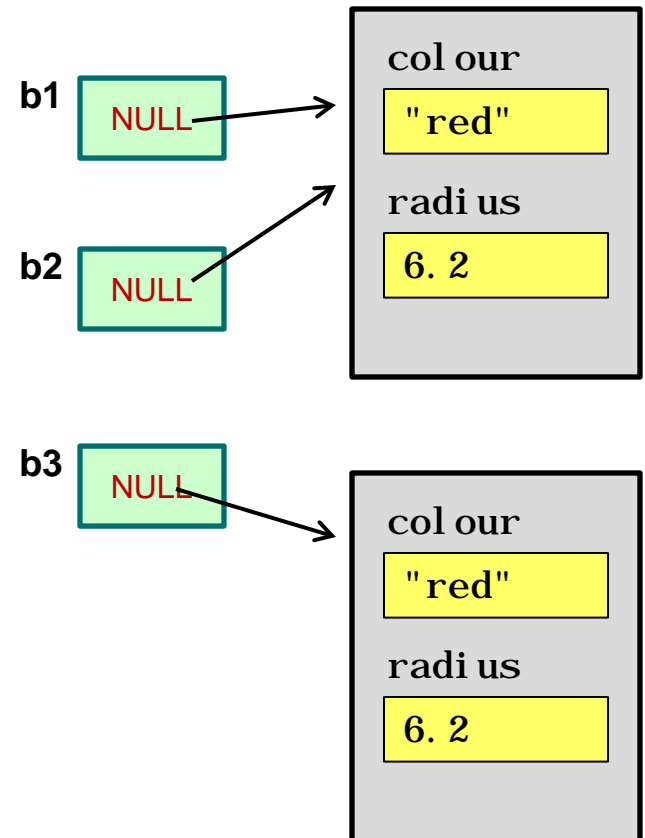
```
public boolean equals(Object obj)
```

# Comparing objects: equals() (1/2)

- To compare if two objects have the same data values, we should use `equals()` instead of `==`
- `==` compares the references of the objects instead

MyBall/TestEquals.java

```
MyBall b1, b2, b3;  
b1 = new MyBall("red", 6.2);  
b2 = b1;  
b3 = new MyBall("red", 6.2);
```



*True or false?*

`(b1 == b2) →`

`(b1 == b3) →`

`b1.equals(b2) →`

`b1.equals(b3) →`

# Comparing objects: equals() (2/2)

- Code for `equals()` method
  - It compares the `colour` and `radius` of both objects ("`this`" and `ball`, which is the 'equivalent' of the parameter `obj`)

MyBall/MyBall.java

```
class MyBall {  
    // Other parts omitted  
  
    // Overriding equals() method  
    public boolean equals(Object obj) {  
        if (obj instanceof MyBall) {  
            MyBall ball = (MyBall) obj;  
            return this.getColour().equals(ball.getColour())  
                && this.getRadius() == ball.getRadius();  
        }  
        else  
            return false;  
    }  
}
```

`instanceof`: To check that the parameter `obj` is indeed a `MyBall` object

Made a local reference `ball` of class `MyBall` so that `getColour()` and `getRadius()` can be applied on it, because `obj` is an `Object` instance, not a `MyBall` instance.

# MyBall Class: Improved (1/2)

- We apply more OOP concepts to our draft `MyBall` class: “`this`” reference, “`this`” in constructor, overriding methods `toString()` and `equals()`

MyBall/MyBall.java

```
class MyBall {
    /***** Data members *****/
    private static int quantity = 0;
    private String colour;
    private double radius;

    /***** Constructors *****/
    public MyBall() {
        this("yellow", 10.0);
    }
    public MyBall(String colour, double radius) {
        setColour(colour);
        setRadius(radius);
        quantity++;
    }

    /***** Accessors *****/
    public static int getQuantity() { return quantity; }

    public String getColour() { return this.colour; }
    public double getRadius() { return this.radius; }
```

“`this`” is  
optional here.

# MyBall Class: Improved (2/2)

MyBall/MyBall.java

```
/****** Mutators *****/
public void setColour(String colour) {
    this.colour = colour;
}

public void setRadius(double radius) {
    this.radius = radius;
}

/****** Overriding methods *****/
// Overriding toString() method
public String toString() {
    return "[" + getColour() + ", " + getRadius() + "]";
}

// Overriding equals() method
public boolean equals(Object obj) {
    if (obj instanceof MyBall) {
        MyBall ball = (MyBall) obj;
        return this.getColour().equals(ball.getColour()) &&
            this.getRadius() == ball.getRadius();
    }
    else
        return false;
}
}
```

"this" is required here.

# Final client program: TestBallV4 (1/2)

- With the overriding methods `toString()` and `equals()` added to the `MyBall` class, the final client program `TestBallV4.java` is shown here (some part of the code not shown here due to space constraint)

`MyBall/TestBallV4.java`

```
import java.util.*;
public class TestBallV4 {
    // readBall() method omitted for brevity

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        MyBall myBall1 = readBall(sc); // Read input and create ball object
        MyBall myBall2 = readBall(sc); // Read input and create another ball object

        // Testing toString() method
        // You may also write: System.out.println("1st ball: " + myBall1.toString());
        //                      System.out.println("2nd ball: " + myBall2.toString());
        System.out.println("1st ball: " + myBall1);
        System.out.println("2nd ball: " + myBall2);
        // Testing ==
        System.out.println("(myBall1 == myBall2) is " + (myBall1 == myBall2));
        // Testing equals() method
        System.out.println("myBall1.equals(myBall2) is "
                           + myBall1.equals(myBall2));
    }
}
```

## Final client program: TestBallV4 (2/2)

- Sample run

```
Enter colour: red
Enter radius: 1.2

Enter colour: red
Enter radius: 1.2
```

```
2 balls are created.
1st ball:
2nd ball:
(myBall1 == myBall2) is
myBall1.equals(myBall2) is
```



---

## **5. Unified Modeling Language (UML)**

---

Abstraction in graphical form

# Introduction to UML

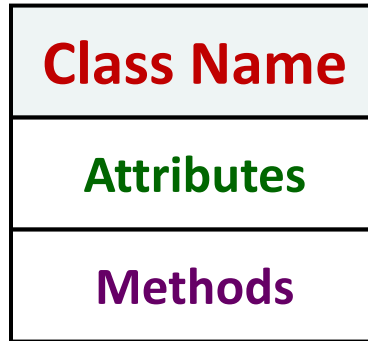


- **Unified Modeling Language is a:**
  - Graphical language
    - A set of diagrams with specific syntax
    - A total of 14 different types of diagram (as of UML2.2)
  - Used to represent object oriented program components in a succinct way
  - Commonly used in software industry
- **In this module:**
  - The diagrams are used loosely
    - We won't be overly strict on the syntax 😊
    - We will only use few diagrams such as class diagram
  - You will learn more in CS2103 Software Engineering or equivalent module

# UML: Class Icon (1/2)



- A **class icon** summarizes:
  - Attributes and methods



## SYNTAX

For attributes:

*[visibility]* attribute: **data\_type**

For methods:

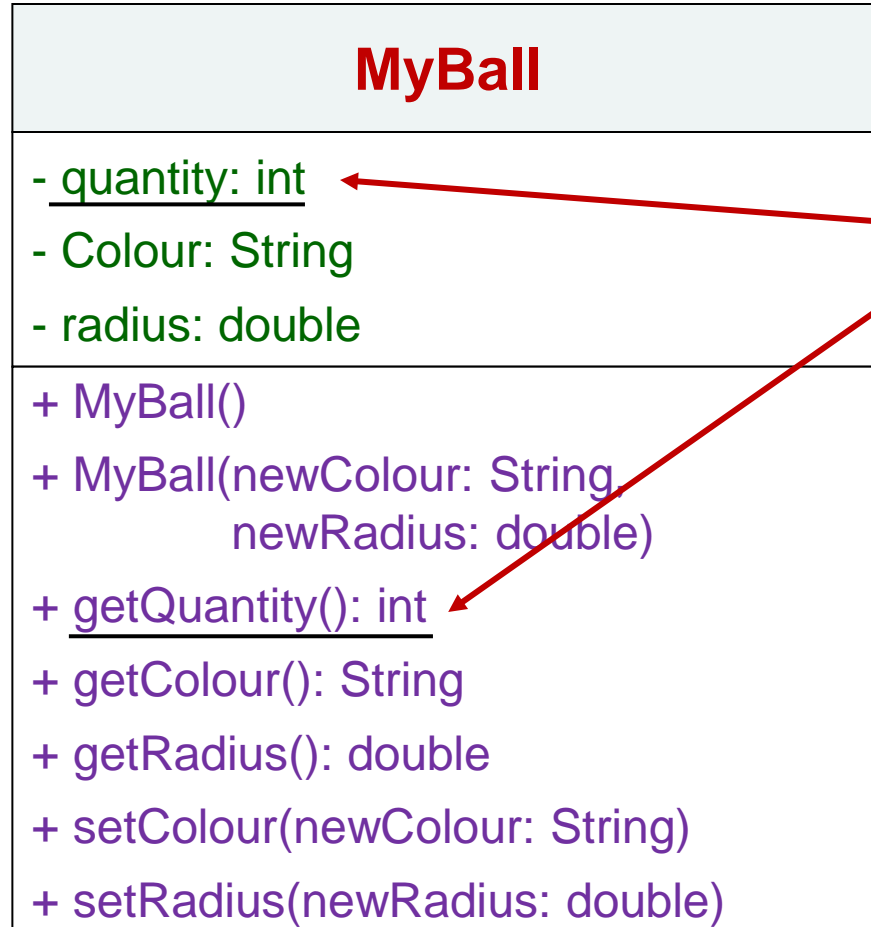
*[visibility]* method(para: **data\_type**): return\_type

Visibility Symbol	Meaning
+	public
-	private
#	protected

# UML: Class Icon (2/2)



- Example: **MyBall** class



- Underlined attributes/methods indicate **class attributes/methods**
- Otherwise, they are **instance attributes/methods**

# UML Diagrams (1/3)



## Examples

*A class*

<Class Name>

MyBall

*An object*

<Object Name>

myBall1

myBall2

*An object  
with class  
name*

<Object Name>: <Class Name>

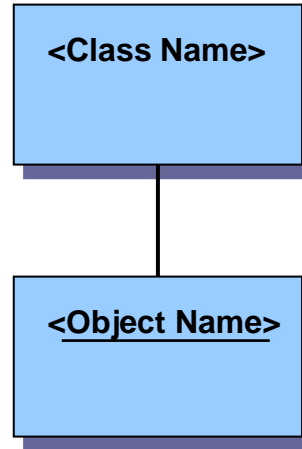
myBall1: MyBall

myBall2: MyBall

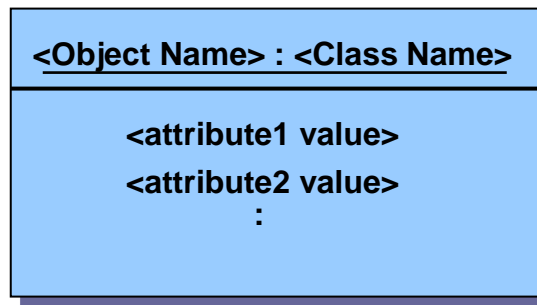
# UML Diagrams (2/3)



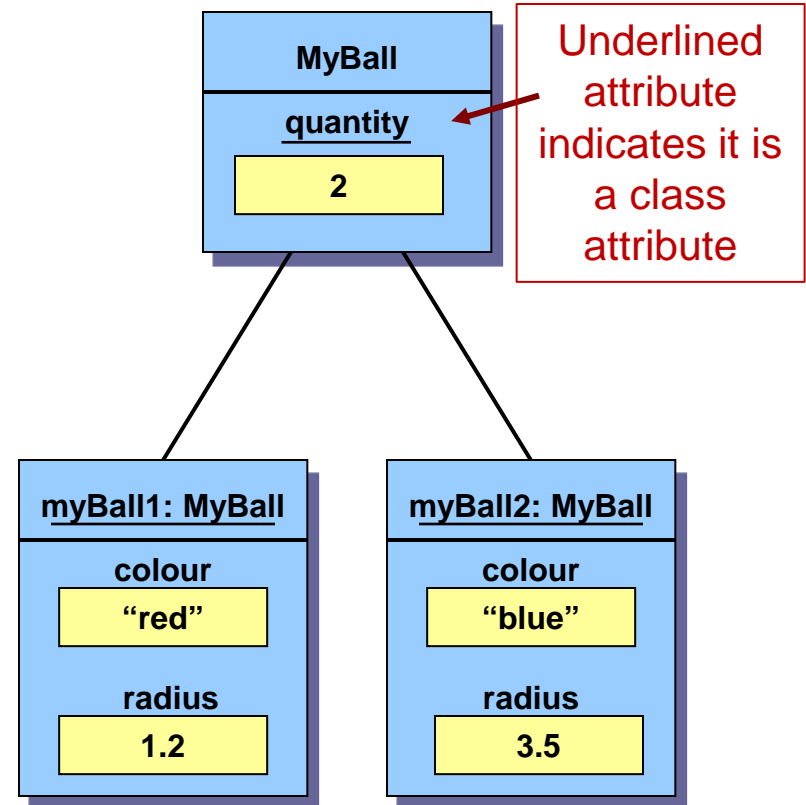
Line showing  
***instance-of***  
relationship



An object  
with data  
values



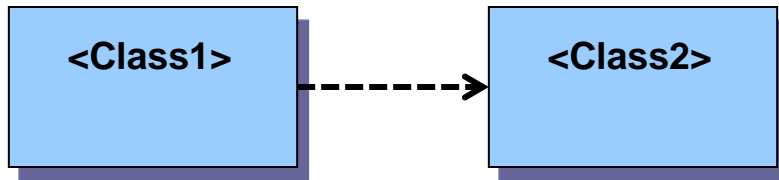
Example



# UML Diagrams (3/3)

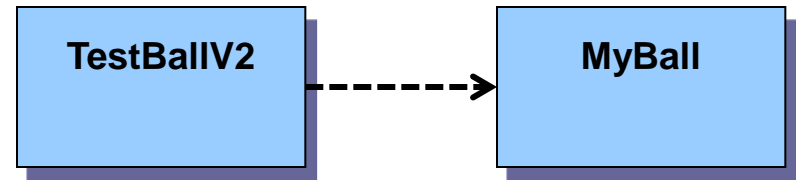


Dotted arrow shows  
**dependency relationship**



**Class1** "depends" on the  
services provided by **Class2**

Example



**TestBallV2** "depends" on the  
services provided by **MyBall**

# Summary

- OOP concepts discussed :
  - ❑ Encapsulation and information hiding
  - ❑ Constructors, accessors, mutators
  - ❑ Overloading methods
  - ❑ Class and instance members
  - ❑ Using “this” reference and “this” in constructors
  - ❑ Overriding methods
- UML
  - ❑ Representing OO components using diagrams



# Practice Exercises

- Practice Exercise #13: Ball with centre
  - Submit both [CentredBall.java](#) and [TestCentredBall.java](#)
- Practice Exercise #14: Fraction
  - Given [TestFraction.java](#), submit only [Fraction.java](#)

End of file