Object-Oriented Programming

Designer Mode

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

Outline

Outline

- 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 lecture, we used classes provided by **API** as a **user**.
- Today, we become **designers** to *create* our own classes!

Outline

World View of a Programming Language

- 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/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

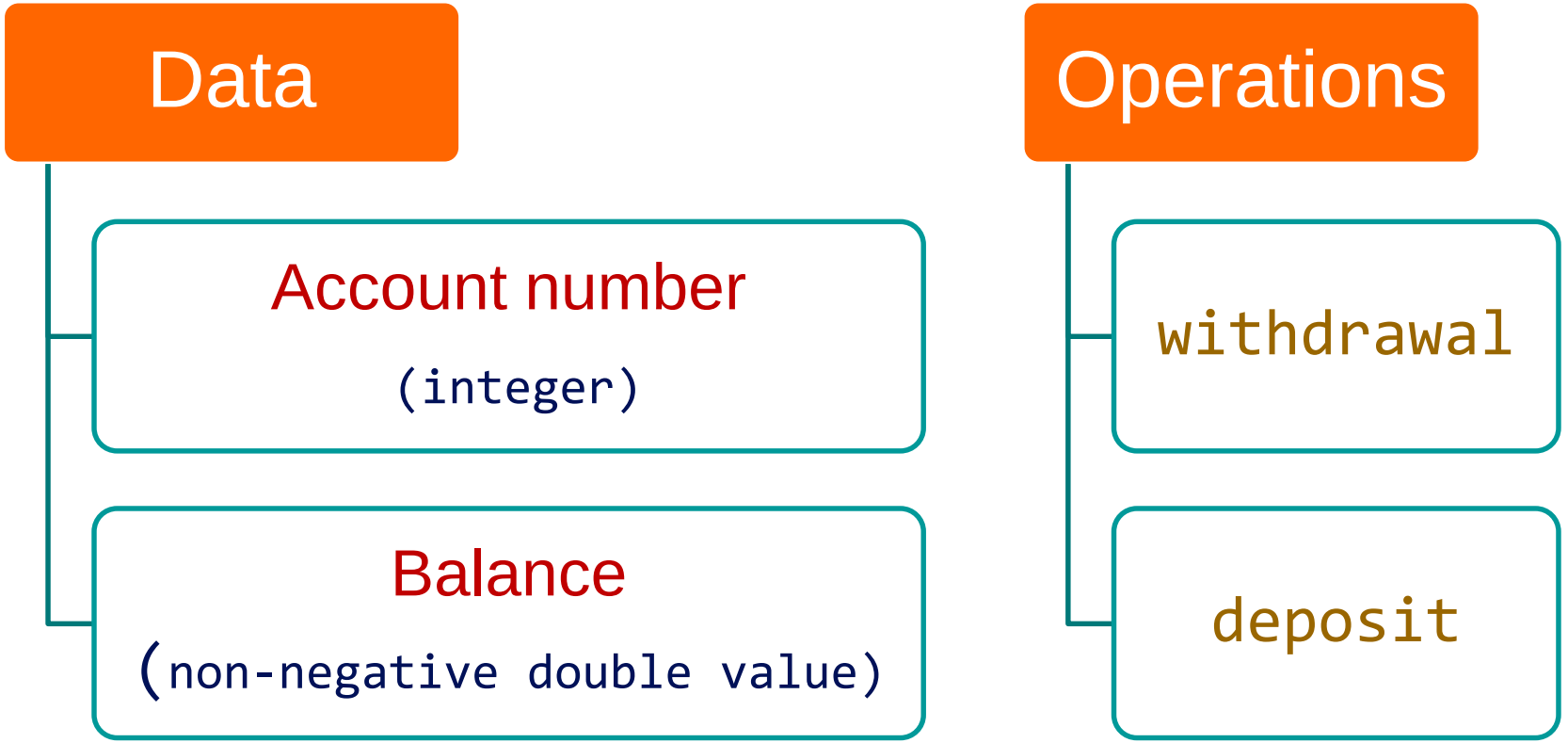
4 fundamental OOP concepts

- 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

- *(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)

Structure to hold data

```
typedef struct {
    int accountNumber;
    double balance;
} BankAccount;
```

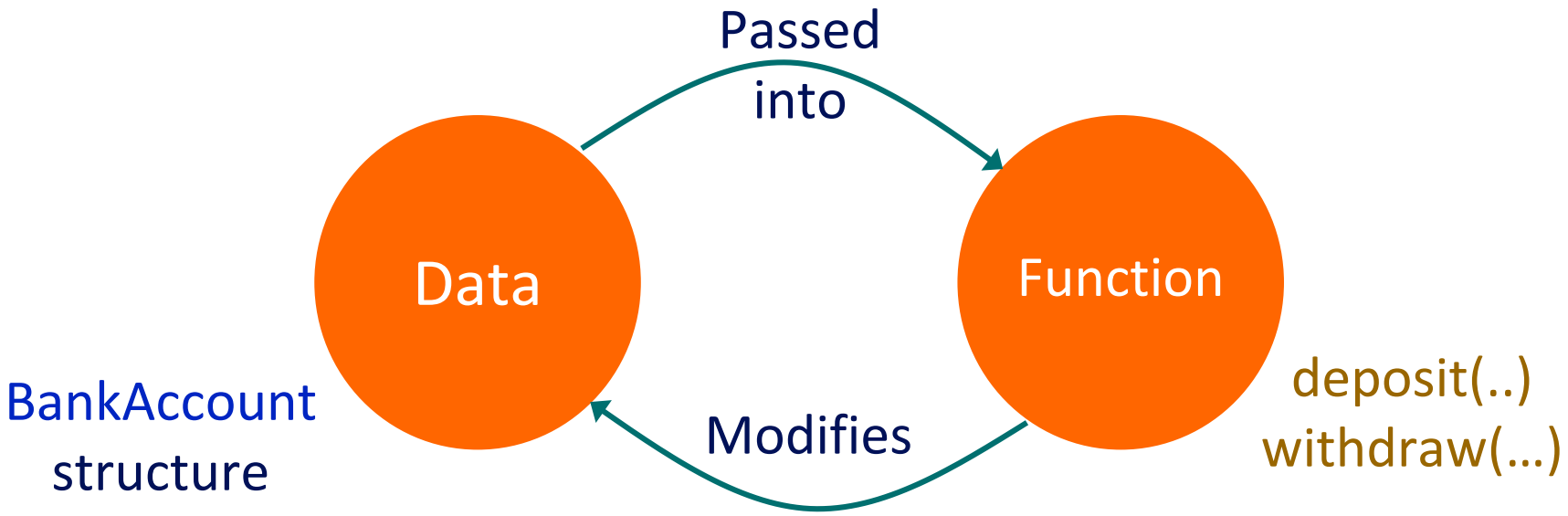
Functions to provide basic operations

```
void initialize(BankAccount *bankAccountPtr, int aNumber) {
    bankAccountPtr->accountNumber = aNumber;
    bankAccountPtr->balance = 0;
}

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

void deposit(BankAccount *bankAccountPtr, double amount) {
    ... Code not shown ...
}
```

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



Bank Account (C implementation) (3/4)

Correct use of BankAccount and its operations

```
BankAccount bankAccount;
initialize(&bankAccount, 12345);
deposit(&bankAccount, 1000.50);
withdraw(&bankAccount, 500.00);
withdraw(&bankAccount, 600.00);
    ...
```

Wrong and malicious exploits of BankAccount

```
BankAccount bankAccount;

deposit(&bankAccount, 1000.50);
initialize(&bankAccount, 12345);

bankAccount.accountNumber = 54321;

bankAccount.balance = 10000000.00;
       ...
```

Forgot to initialize

Account Number should not change!

Balance should be changed by authorized operations only

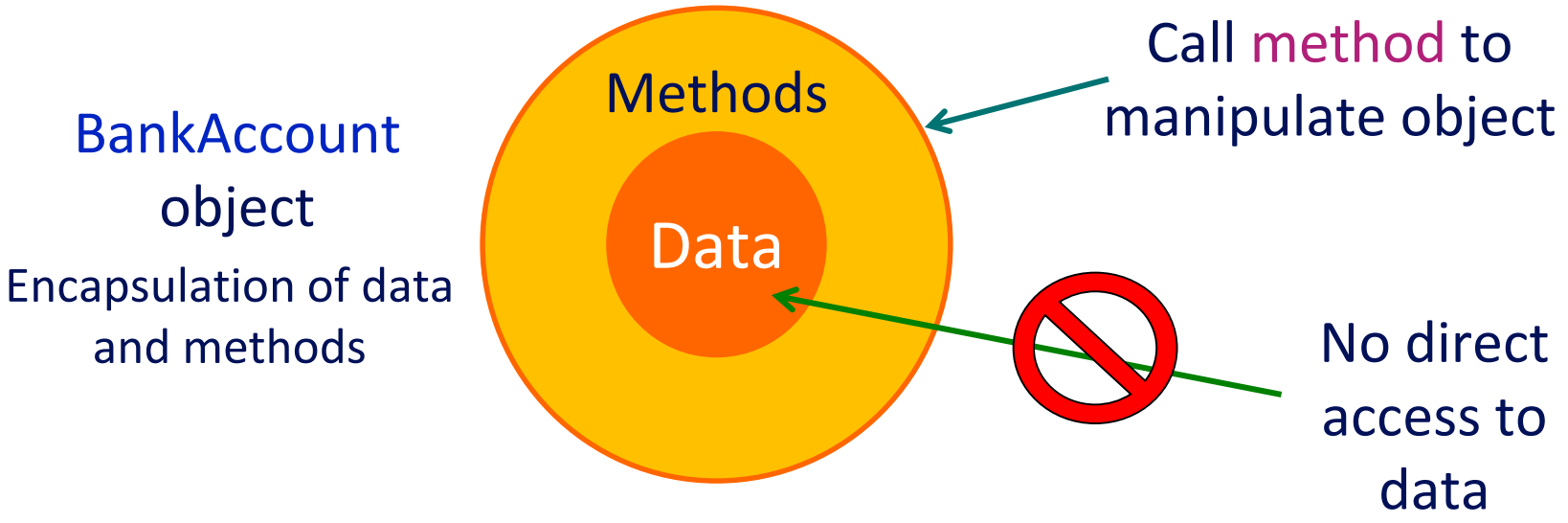
Bank Account (C implementation) (4/4)

- 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

- A conceptual view of an OO implementation for Bank Account



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

Outline

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.

- 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.

- 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.

- 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

`public`	• Anyone can access • Usually intended for methods only
`private`	• Can be accessed by the same class • Recommended for all attributes
`protected`	• Can be accessed of the same class or its child classes can access it AND • Can be accessed 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**

- 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: BankAccount Class (1/2)

BankAccount.java

```
1  class BankAccount {
2
3    private int accountNumber;
4    private double balance;
5
6    // Default constructor
7    public BankAccount() {
8      // By default, numeric attributes
9      // are initialised to 0
10   }
11
12   public BankAccount(int aNumber, double aBalance) {
13     // Initilize attributes with user
14     // provided values
15     accountNumber = aNumber;
16     balance = aBalance;
17   }
18
19   // Other methods on next slide
```

Attributes of BankAccount

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

Can be overloaded.

Bank Account: BankAccount Class (2/2)

BankAccount.java

```
20  public int getAccountNumber() {
21    return accountNumber;
22  }
23
24  public double getBalance() {
25    return balance;
26  }
27
28  public boolean withdraw(double amount) {
29    if (balance < amount) {
30      return false;
31    }
32
33    balance -= amount;
34    return true;
35  }
```

BankAccount.java

```
36    public void deposit(double amount) {
37      if (amount <= 0) {
38        return;
39      }
40
41      balance += amount;
42    }
43
44    public void print() {
45      System.out.println("Account number: "
46                         + getAccountNumber());
47      System.out.printf("Balance: $%.2f\n",
48                        getBalance());
49    }
50  }
```

- 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: getAccountNumber(), 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

- 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 BankAccount 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.

- Note that there is **no** main() method in BankAccount class because it is a service class, not a client class (application program). You cannot execute BankAccount.
- So how do we write a client class to make use of BankAccount?
- 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: TestBankAccount

TestBankAccount.java

```
1  public class TestBankAccount {
2    public static void main(String[] args) {
3      BankAccount bankAccount1 = new BankAccount();
4      BankAccount bankAccount2 = new BankAccount(1234, 321.70);
5
6      System.out.println("Before transactions:");
7      bankAccount1.print();
8      bankAccount2.print();
9
10     bankAccount1.deposit(1000);
11     bankAccount1.withdraw(200.50);
12     bankAccount2.withdraw(500.25);
13
14     System.out.println();
15     System.out.println("After transactions:");
16     bankAccount1.print();
17     bankAccount2.print();
18   }
19 }
```

Which constructor is used?

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

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

What happens if...

```
1  public class TestBankAccount {
2    public static void main(String[] args) {
3
4      BankAccount bankAccount1 = new BankAccount();
5
6      /* Instead of
7      * bankAccount1.deposit(1000);
8      */
9
10     bankAccount1.balance += 1000;
11   }
12 }
```

Compilation error!
balance has private access in BankAccount

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

- BankAccount.java and TestBankAccount.java can be compiled independently.
- Only TestBackAccount class can be executed.

```
javac BankAccount.java
javac TestBankAccount.java
java TestBankAccount
```

- We say TestBankAccount uses or depends on BankAccount.
- 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 previous lecture depends on both Scanner and Math service classes.)

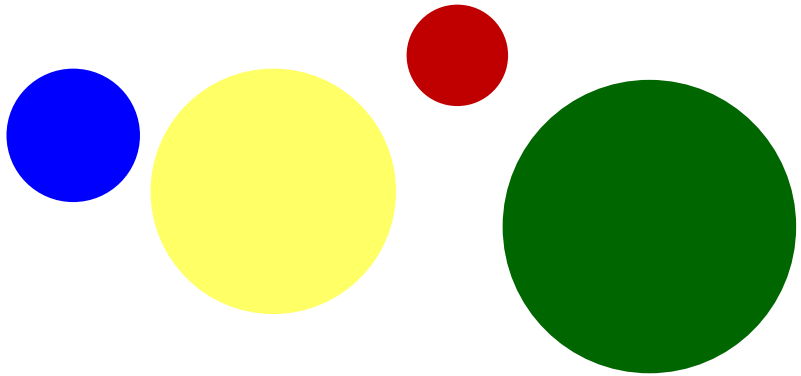
Outline

- 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

```
1  public class MyBall {
2    /************** Data members **********************/
3    private static int quantity = 0;
4    private String colour;
5    private double radius;
6
7    /************** Constructors **********************/
8    // Default constructor creates a yellow, radius 10.0 ball
9    public MyBall() {
10     setColour("yellow");
11     setRadius(10.0);
12     quantity++;
13   }
14
15   public MyBall(String newColour, double newRadius) {
16     setColour(newColour);
17     setRadius(newRadius);
18     quantity++;
19   }
```

MyBall Class: Draft (2/2)

MyBall_draft/MyBall.java

```
20    /*************** Accessors *************/
21    public static int getQuantity() {
22      return quantity;
23    }
24
25    public String getColour() {
26      return colour;
27    }
28
29    public double getRadius() {
30      return radius;
31    }
32
33    /*************** Mutators **************/
34    public void setColour(String newColour) {
35      colour = newColour;
36    }
37
38    public void setRadius(double newRadius) {
39      radius = newRadius;
40    }
41  }
```

Class method

The rest are all instance methods

Testing MyBall: TestBallV1 (1/2)

MyBall_draft/TestBallV1.java

```
1  public class TestBallV1 {
2    public static void main(String[] args) {
3      Scanner sc = new Scanner(System.in);
4      // Read ball's input and create a ball object
5      System.out.print("Enter colour: ");
6      String inputColour = sc.next();
7      System.out.print("Enter radius: ");
8      double inputRadius = sc.nextDouble();
9      MyBall myBall1 = new MyBall(inputColour, inputRadius);
10     System.out.println();
11
12     // Read another ball's input and create another ball object
13     System.out.print("Enter colour: ");
14     inputColour = sc.next();
15     System.out.print("Enter radius: ");
16     inputRadius = sc.nextDouble();
17     sc.close();
18
19     MyBall myBall2 = new MyBall(inputColour, inputRadius);
20     System.out.println();
21     System.out.println(MyBall.getQuantity() + " balls are created.");
22     System.out.println("1st ball's colour and radius: " + myBall1.getColour() + ", " + myBall1.getRadius());
23     System.out.println("2nd ball's colour and radius: " + myBall2.getColour() + ", " + myBall2.getRadius());
24   }
25 }
```

constructor

Calling a class method

Calling instance methods

Testing MyBall: TestBallV1 (2/2)

MyBall_draft/TestBallV1.java

```
1  public class TestBallV1 {
2    public static void main(String[] args) {
3      Scanner sc = new Scanner(System.in);
4      // Read ball's input and create a ball object
5      System.out.print("Enter colour: ");
6      String inputColour = sc.next();
7      System.out.print("Enter radius: ");
8      double inputRadius = sc.nextDouble();
9      MyBall myBall1 = new MyBall(inputColour, inputRadius);
10     System.out.println();
11
12     // Read another ball's input and create another ball object
13     System.out.print("Enter colour: ");
14     inputColour = sc.next();
15     System.out.print("Enter radius: ");
16     inputRadius = sc.nextDouble();
17     sc.close();
18     MyBall myBall2 = new MyBall(inputColour, inputRadius);
19     System.out.println();
20
21     System.out.println(MyBall.getQuantity() + " balls are created.");
22     System.out.println("1st ball's colour and radius: " + myBall1.getColour() + ", " + myBall1.getRadius());
23     System.out.println("2nd ball's colour and radius: " + myBall2.getColour() + ", " + myBall2.getRadius());
24   }
}
```

```
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
```

- 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

```
1   import java.util.*;
2   public class TestBallV2 {
3     // This method reads ball's input data from user, creates a ball object, and returns it to the caller.
4     public static MyBall readBall(Scanner sc) {
5       System.out.print("Enter colour: ");
6       String inputColour = sc.next();
7       System.out.print("Enter radius: ");
8       double inputRadius = sc.nextDouble();
9       return new MyBall(inputColour, inputRadius);
10    }
11
12    public static void main(String[] args) {
13      Scanner sc = new Scanner(System.in);
14      MyBall myBall1 = readBall(sc);  // Read input and create ball object
15      System.out.println();
16
17      MyBall myBall2 = readBall(sc);  // Read input and create another ball object
18      sc.close();
19      System.out.println();
20
21      System.out.println(MyBall.getQuantity() + " balls are created.");
22      System.out.println("1st ball's colour and radius: " + myBall1.getColour() + ", " + myBall1.getRadius());
23      System.out.println("2nd ball's colour and radius: " + myBall2.getColour() + ", " + myBall2.getRadius());
24    }
25  }
```

"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;
}
```

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

The original code:

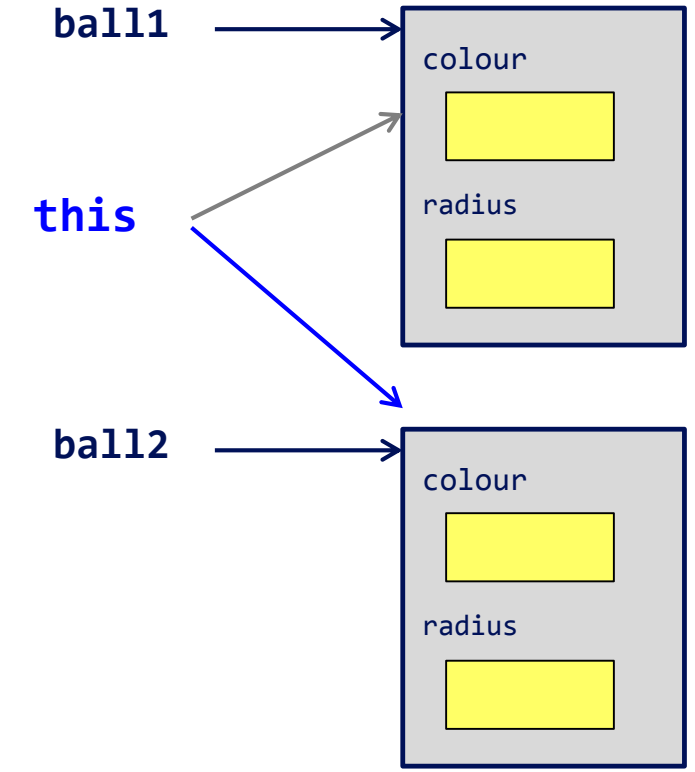
```
public void setColour(String newColour) {
  colour = newColour;
}

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

"this" reference (2/4)

- A common confusion:
  - How does the method "know" which is the "object" it is currently communicating with? (Since there could be many objects created from that class.)

```
// ball1 and ball2 are MyBall objects
ball1.setColour("purple");
ball2.setColour("brown");
```

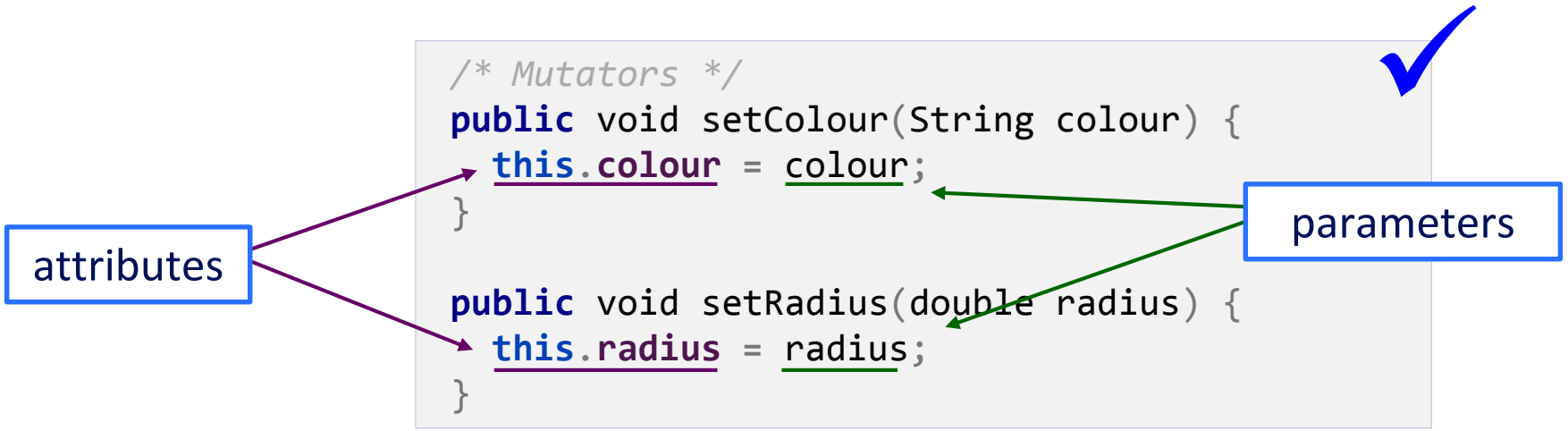


- Whenever a method is called,
  - a reference to the calling object is set automatically
  - Given the name **"this"** in Java, meaning "*this particular object*"
- All attributes/methods are then accessed implicitly through this reference

- 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;
}
```



- 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;
}
```

Optional

- 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 suffice "_" 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 this work?

```
public MyBall() {
  colour = "yellow";
  radius = 10.0;
  quantity++;
}

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

- Both work, but the left version follows the principle of code reuse which minimises code duplication, but is slightly less efficient.
- The left 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:

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

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

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

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

```
MyBall ball = 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.

- 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() (1/2)

- 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

```
1  import java.util.*;
2  public class TestBallV3 {
3
4    // readBall() method omitted
5    public static void main(String[] args) {
6      Scanner sc = new Scanner(System.in);
7
8      MyBall myBall1 = readBall(sc); // Read input and create ball object
9      System.out.println();
10
11     MyBall myBall2 = readBall(sc); // Read input and create another ball object
12     System.out.println();
13
14     System.out.println("1st ball: " + myBall1);
15     System.out.println("2nd ball: " + myBall2);
16   }
17 }
```

```
Enter colour: red
Enter radius: 1.2

Enter colour: blue
Enter radius: 3.5
```

```
1st ball: Ball@471e30
2nd ball: Ball@10ef90c
```

Object identifiers (OIDs)

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

- 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)

MyBall_draft/MyBall.java

```
1  class MyBall {
2    // original code omitted
3
4    public String toString() {
5      return "[" + getColour() + ", " + getRadius()
6                 + "]";
7    }
8  }
```

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

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

```
System.out.println(myBall.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

- 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)

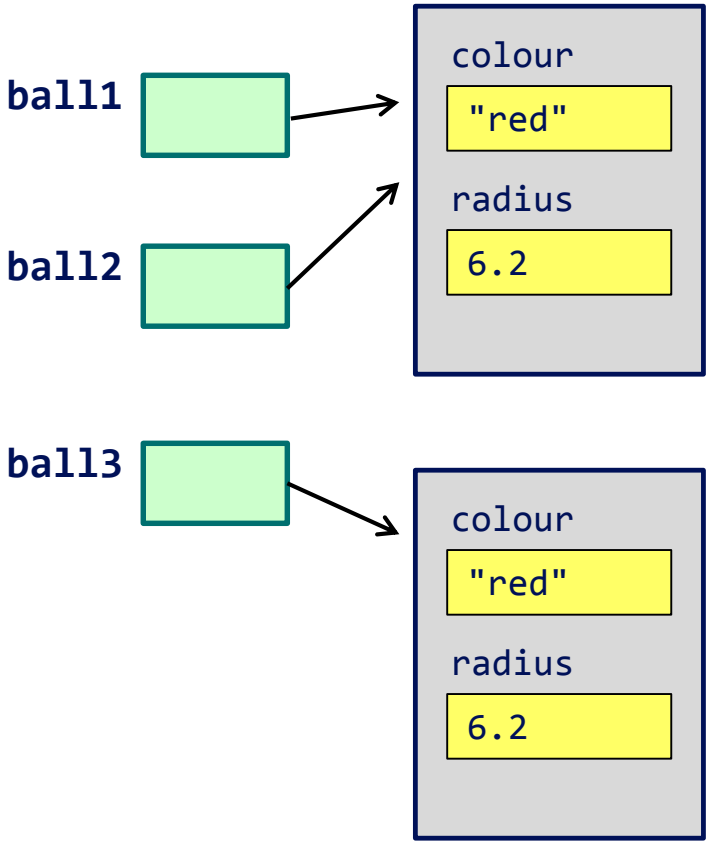
- To compare if two objects have the same data values, we should use equals() instead of ==
- == compares the references of the objects instead

MyBall/TestEquals.java

```
MyBall ball1 = new MyBall("red", 6.2);
MyBall ball2 = ball1;
MyBall ball3 = new MyBall("red", 6.2);
```

```
(ball1 == ball2) →
(ball1 == ball3) →
ball1.equals(ball2) →
ball1.equals(ball3) →
```

True or false?



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 object)

MyBall/MyBall.java

```
1  class MyBall {
2    // Other parts omitted
3
4    // Overriding equals() method
5    public boolean equals(Object obj) {
6      if (obj instanceof MyBall) {
7        MyBall ball = (MyBall)obj;
8        return this.getColour().equals(ball.getColour())
9          && this.getRadius() == ball.getRadius();
10     }
11
12     return false;
13   }
14 }
```

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

```
1  class MyBall {
2    /************ Data members **************/
3    private static int quantity = 0;
4    private String colour;
5    private double radius;
6
7    /************ Constructors *************/
8    public MyBall() {
9      this("yellow", 10.0);
10   }
11
12   public MyBall(String colour, double radius) {
13     setColour(colour);
14     setRadius(radius);
15     quantity++;
16   }
```

MyBall/MyBall.java

```
17   /************ Accessors **************/
18   public static int getQuantity() {
19     return quantity;
20   }
21
22   public String getColour() {
23     return this.colour;
24   }
25
26   public double getRadius() {
27     return this.radius;
28   }
```

"this" is optional here.

MyBall Class: Improved (2/2)

MyBall/MyBall.java

```
29  /************ Mutators *************/
30  public void setColour(String colour) {
31    this.colour = colour;
32  }
33
34  public void setRadius(double radius) {
35    this.radius = radius;
36  }
37
38  /******** Overriding methods ********/
39  // Overriding toString() method
40  public String toString() {
41    return "[" + getColour() + ", "
42               + getRadius() + "]";
43  }
```

"this" is required here.

MyBall/MyBall.java

```
44    // Overriding equals() method
45    public boolean equals(Object obj) {
46      if (obj instanceof MyBall) {
47        MyBall ball = (MyBall)obj;
48        return this.getColour().equals(ball.getColour())
49          && this.getRadius() == ball.getRadius();
50      }
51
52      return false;
53    }
54  }
```

Final client program: TestBallV4 (1/2)

MyBall/TestBallV4.java

```
1  import java.util.*;
2  public class TestBallV4 {
3    // readBall() method omitted for brevity
4    public static void main(String[] args) {
5      Scanner sc = new Scanner(System.in);
6      MyBall myBall1 = readBall(sc); // Read input and create ball object
7      MyBall myBall2 = readBall(sc); // Read input and create another ball object
8
9      // Testing toString() method
10     // You may also write: System.out.println("1st ball: " + myBall1.toString());
11     //                     System.out.println("2nd ball: " + myBall2.toString());
12     System.out.println("1st ball: " + myBall1);
13     System.out.println("2nd ball: " + myBall2);
14     // Testing ==
15     System.out.println("(myBall1 == myBall2) is " + (myBall1 == myBall2));
16     // Testing equals() method
17     System.out.println("myBall1.equals(myBall2) is " + myBall1.equals(myBall2));
18   }
19 }
```

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)

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
```

- 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

Outline

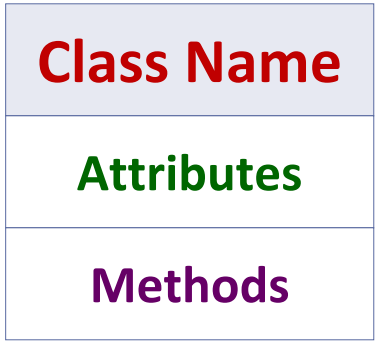
Abstraction in graphical form

- **U**nified **M**odeling **L**anguage 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

UML: Class Icon (1/2)

- A class icon summarizes:
  - Attributes and methods



SYNTAX

```
For attributes:
  [visibility] attribute: dataType

For methods:
  [visibility] method(para: dataType): returnType
```

Visibility Symbol	Meaning
+	public
-	private
#	protected

- Example: MyBall class

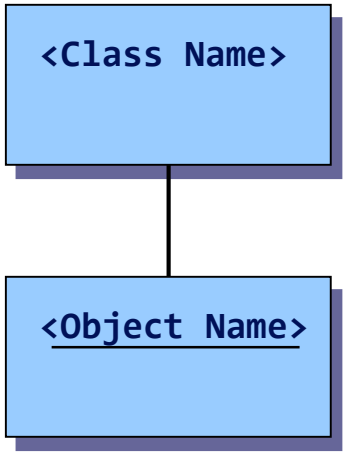
- Underlined attributes/methods indicate class attributes/methods
- Otherwise, they are instance attributes/methods

UML Diagrams (1/3)

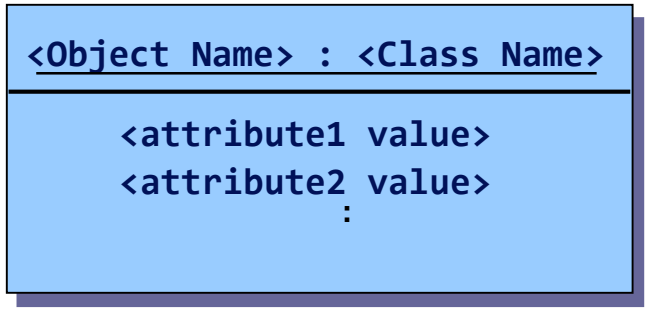
		Examples
A class	<Class Name>	MyBall
An object	<Object Name>	myBall1 myBall2
An object with class name	<Object Name>: <Class Name>	myBall1: MyBall myBall2: MyBall

UML Diagrams (2/3)

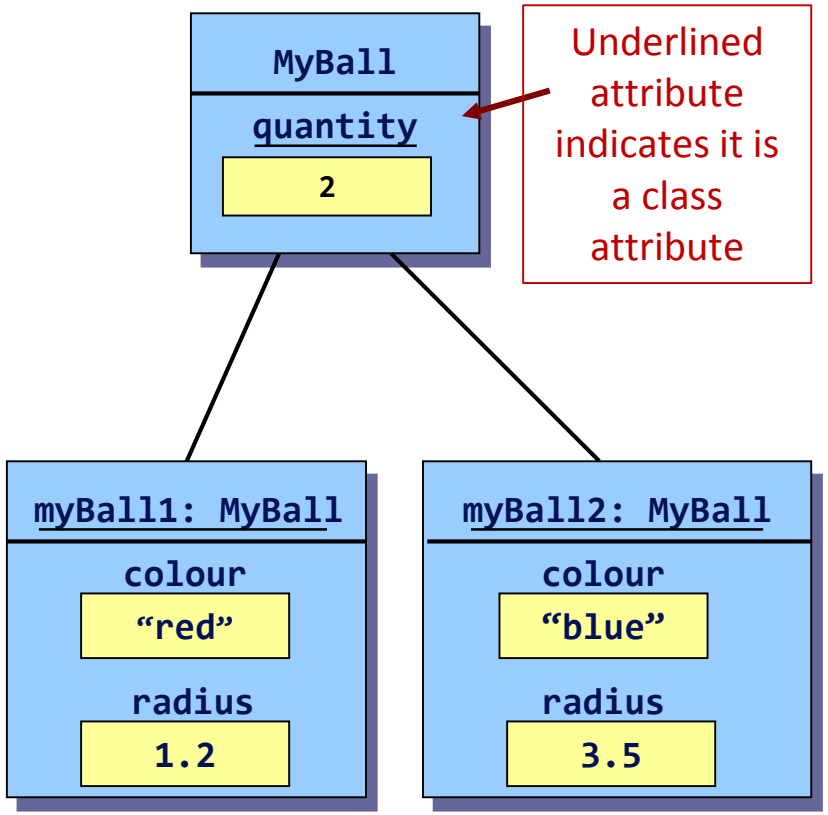
Line showing ***instance-of*** *relationship*



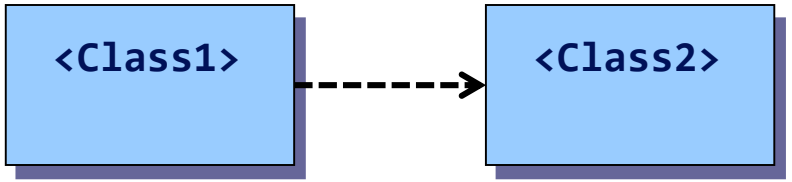
An object with data values



Example

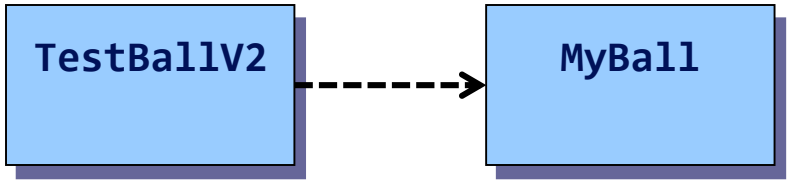


Dotted arrow shows ***dependency relationship***



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

Example



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

- 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

Thank you!