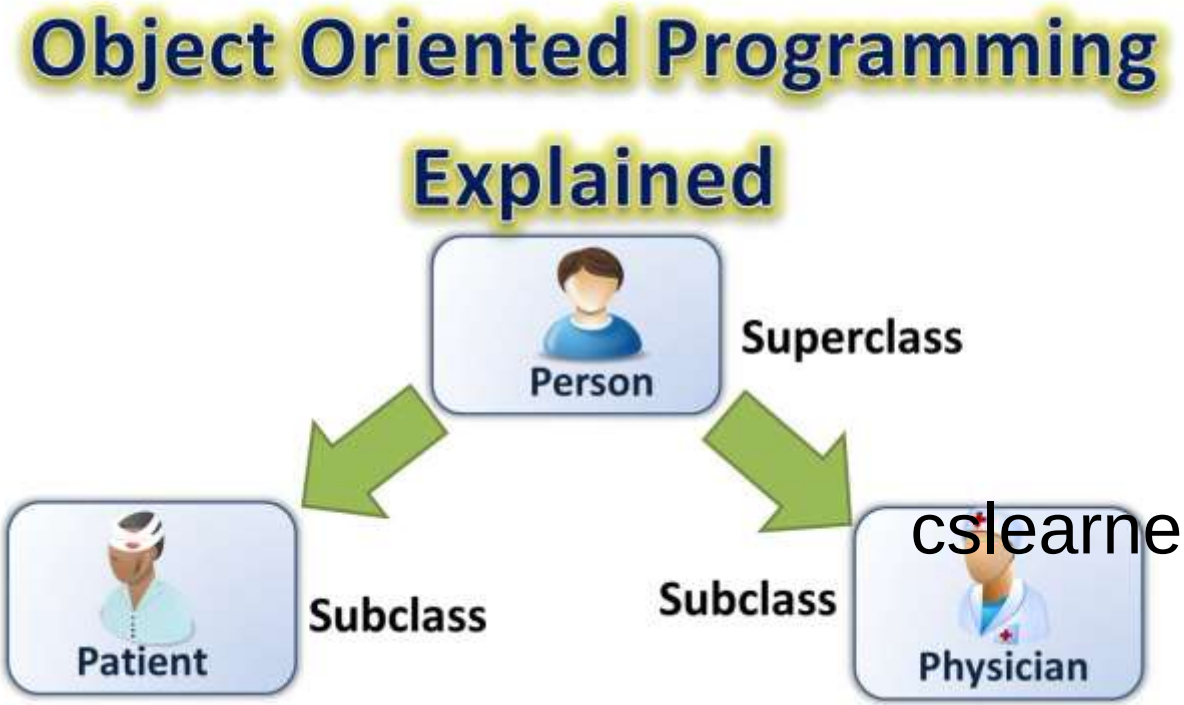
Yasir Ahmad
Visiting Faculty Member
From
ICS & IT Department
The University Of Agriculture Peshawar

Programming Languages II -- Java

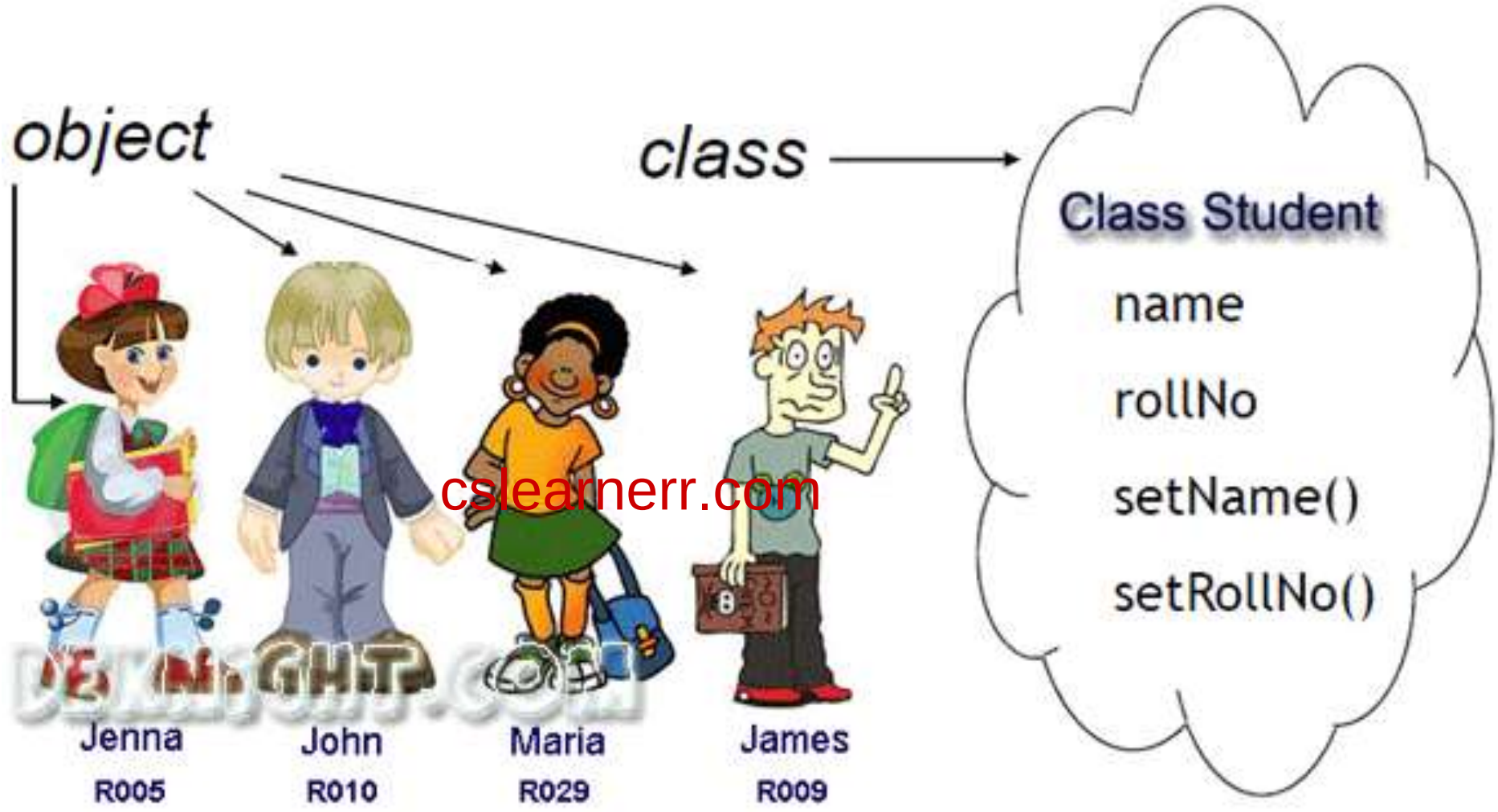
Object Oriented Programming in Java-II

Programming Languages II -- Java

Object Oriented Programming in Java-II



Object Oriented Concepts



Object Oriented Concepts

- Data Abstraction and Encapsulation
- Inheritance
- Polymorphism

cslearnerr.com

Today's Lecture

- Data Abstraction and Encapsulation

Today's Lecture

- Data Abstraction and Encapsulation

```
public class Car{
    private string _color;
    private string _model;
    private string _makeYear;
    private string _fuelType;

    public void Start(){
        ..
    }

    public void Stop(){
        ..
    }

    public void Accelerate(){
        ..
    }
}
```

Data Abstraction(خیال)

- Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of car or applying brakes will stop the car, **but he does not know about how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes etc in the car.** This is what abstraction is.

Encapsulation (1/2)

- **Realtime Example :**
  Suppose you have an account in the bank. If your balance variable is declared as a public variable in the bank software, your account balance will be known as **public**, In this case, anyone can know your account balance. So, would you like it? Obviously No.
- So, they declare balance variable as private for making your account safe, so that anyone cannot see your account balance.
- The person who has to see his account balance, will have to access only **private members** through methods defined inside that class and this method will ask your account holder name or user Id, and password for authentication.
- Thus, we can achieve **security** by utilizing the concept of data hiding. This is called Encapsulation in Java.

Encapsulation (2/2) Public ,private, protected

Visibility	Public	Protected	Default	Private
From the same class	Yes	Yes	Yes	Yes
From any class in the same package	Yes	Yes	Yes	No
From a subclass in the same package	Yes	Yes *(Package, Inheritance)*	Yes *(Package)*	No
From a subclass outside the same package	Yes	Yes *(Inheritance)*	No	No
From any non-subclass class outside the package	Yes	No	No	No

- **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
- **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
- **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
- **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

Real World Objects

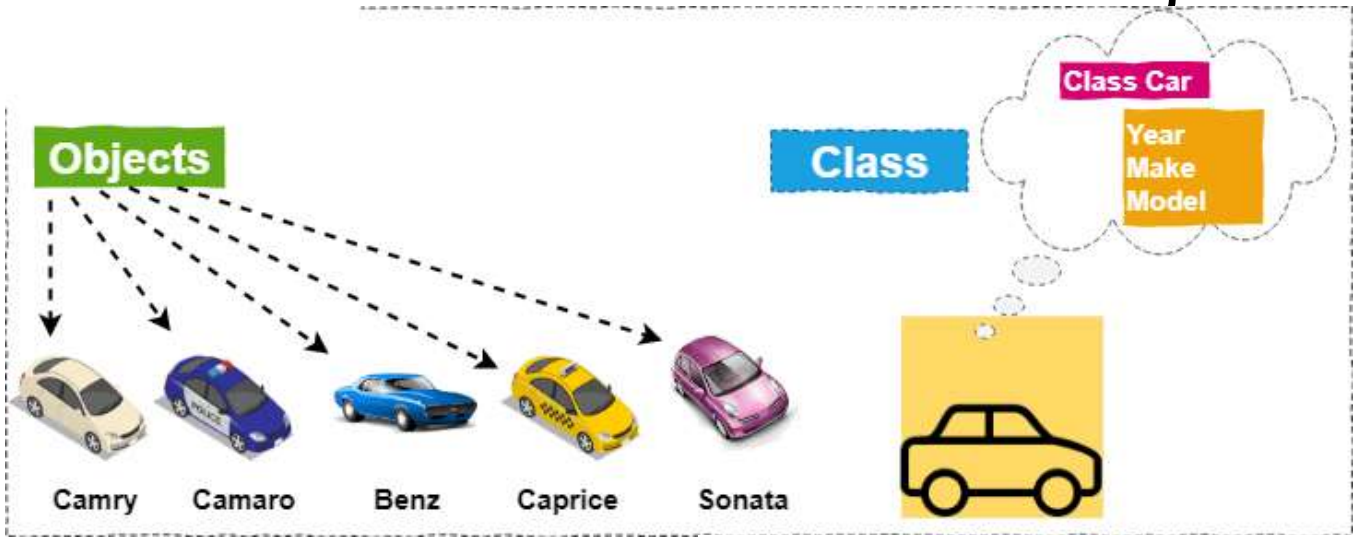
- We all interact with real world objects (i.e. things)
  - A chair
  - A sweet
  - A pen

cslearnerr.com

- All objects can be described by:
  - Attributes (combination of which define the object state)
  - Behaviours (actions performed using attributes)

- Simple attributes are measurable quantities
  - E.g. height, length, weight, calories, ink-level, etc
- Some behaviours are easy to describe
  - Adjust Height – increases / decrease chair height
  - Write – decreases ink in pen

Real World Objects




```
public class Car{
    private string _color;
    private string _model;
    private string _makeYear;
    private string _fuelType;

    public void Start(){
        ..
    }

    public void Stop(){
        ..
    }

    public void Accelerate(){
        ..
    }
}
```

Classes of Real Objects

- When two objects can be described by the exact *same set of* attributes(filelds) and behaviours(functions)
  - Then the objects belong to same the class
  - Not necessarily the same attribute values!

- If two objects can be described by a similar set of attributes and actions
  - Then the objects could be related

- If two objects are of the same class and have the same value for their attributes
  - Then at that point in time they are identical

Java Class

- We design computer programs to solve problems in the real world
  - The real world is composed of objects
  - Thus we can use software "objects" in programs

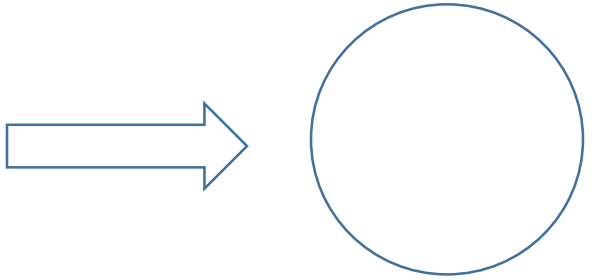
- In Java, we can define a Class, which specifies
  - Attributes as a set of *fields* (variables, constants, etc.)
  - Behaviours as *methods*

- All classes are unique, but can be related
  - Thus encouraging re-use of code
- To put the class into action

cslearnerr.com

  - We declare *instances* of the class called objects

Circle.java



```
public class Circle {

private int radius;
  public Circle(int rad){radius = rad;}//constructor


  public int diameter(){return 2 * radius;}


  public double area(){
      return radius * radius * Math.PI;
  }

  public double circumference(){
      return 2 * radius * Math.PI;
  }
}
```

Fields

- Only one field is specified:

```
private int radius;
```

- Data Encapsulation:
  - The field is declared **`private`** meaning only code inside the class can directly access it
- Data Abstraction:
  - Other potential state attributes (diameter, area and circumference) can be calculated from the radius
  - Thus fields are not defined for these potential attributes
  - Instead we define methods to calculate them

Constructor

- Purpose is to initialise some/all the fields of a class when initializing an object
  - Always has same name as class
  - A class can have zero, one or more constructors
  - If no constructor is defined then JVM generates a default constructor which initialises all fields to default values

- Circle Constructor is:

```
public Circle(int rad){radius = rad;}
```

  - The constructor will be used by outside code, so is declared `public`
  - This constructor accepts one parameter and initialises the field to that parameter

Methods

- The purpose of methods within a class is to simulate behaviour of real world equivalent
  - Calculate derivable attributes
- Method Types:
  - Constructors: Used to initialize the fields of a class when creating an instance of the class (discussed on previous slide)
  - Accessors: Read the value of a field
  - Mutators: Change the value of a field
- Many methods will be marked as `public`
  - Some may be also specified to be `static`, meaning they can be used without an instance
- In the circle class we have three methods, all of which calculate a derived attribute:
  - `+ diameter(): int`
  - `+ circumference(): double`
  - `+ area(): double`

Main Class

```
public class CircleDemo {
  public static void main(String[] args) {
    Circle circle = new Circle(10);
    System.out.format("%nCircle object
    created with radius of 5");
    System.out.format("%nDiameter is %d",
    circle.diameter());
    System.out.format("%nCircumference is
    %.2f", circle.circumference());
    System.out.format("%nArea is %.2f",
    circle.area());
  }
}
```

Object Instance

- The class, by itself, is a template(سانچے)
- Will remain dormant(غیر فعال) unless we create an instance

- Syntax:
  - *ClassName identifier* **=** **new** *ClassName***(***args***)**
- Example
  - `Circle circle = new Circle(10);`

- The left hand side is declaring the object variable
  - **The right hand side is instantiating the object by using the class constructor**

Reference and Instantiation (1)

- When an object instance is declared,

```
Circle circle3;
```

  - A reference variable is declared
  - The reference has nowhere to point i.e. no object data
- When an object instance is instantiated using the constructor,

```
circle3 = new Circle();
```

  - An object is created on the heap, with sufficient memory for each field in the object
  - A link is created back, such that the reference variable will point to the newly created object
- Setting a reference to `null` destroys the link

Reference and Instantiation (2)

- We do not have to instantiate every object reference
  - Instead we can assign an instantiated object to a reference

```
circle3 = circle2;
```

- In doing so the object reference and instantiated object link to the exact same object on the heap
  - I.e. `circle2` and `circle3` point to same memory locations
  - Thus any changes made by one object to its field will be reflect by the other object
- If we set `circle2` to `null`, i.e: `circle2 = null;`
  - Link between `circle2` reference and object data is broken
  - Leaving only `circle3` pointing to the object data on heap

Object Methods

- To make use of an object methods, we apply dot notation to the object instance
- Format

  identifier`.`*method*`(`*args*`);`
- Example

```
circle1.area();
```

Static Fields

- Static fields are class variables
  - Each object instance shares these fields
- If one object changes the value of a static field
  - Then change is visible to every object instance
- Common use for static variables is to maintain an auto-number count for generating `ID` field values

Static Methods

- Static methods are methods which are used via the Class rather than through an object instance.
  - E.g. `Integer.parseInt()` and `String.format()`
- In the example, static versions of the `diameter`, `area` and `circumference` methods can be defined

```
//non static sevice method
public int diameter(){return 2 * radius;}
//static – class method
public static int diameter(int r){return 2 * r;}
```

- Static methods can be used externally, instead of re-coding relevant calculations
  - All is needed is for the radius to be provided

```
Circle.diameter(radius)
```