PBCST304:
OBJECT ORIENTED PROGRAMMING

Learning Outcomes

- Understand the basic concepts and important terminologies of Object-Oriented Programming

Object-Oriented Programming

- Basic Concepts:
  - Objects
  - Classes
  - Abstraction
  - Encapsulation
  - Inheritance
  - Polymorphism

Objects

- Objects are the **basic runtime entities** in an object oriented system
- They may represent a **tangible real-world entity** such as a library member, a book, an issue register, etc.
- The working of a **software system** can be imagined in terms of **a set of interacting objects**.
- When a program is executed, the **objects interact** by sending messages to one another.

Objects

- Each **object** essentially consists of **some data** that is private to the object and **a set of functions** (termed as **operations**/ **methods**) that operate on those data.
- The data of the object can **only be accessed** by the methods of the object.
- The only access point to the data for the external objects is **through the invocation of the methods of the object**.
- i.e., **No** object can **directly** access the data of any other object

Objects

- This **mechanism of hiding data** from other objects is known as the principle of ***data hiding*** or ***data abstraction***
- The data stored internally in an object are called its ***attributes***, and the operations supported by an object are called its ***methods***.
- Example: **Employee –** data, functions.

Class

- **Similar objects** constitute a **class**
- All the objects constituting a class possess similar attributes and methods

Class
Dog

Properties:	**Behavior:**
breed	eat()
size	run()
age	bark()
color	sleep()

We will get different dog objects based on different values of data members

Class

- A ***Class*** is a **user-defined data-type** which has ***data members*** and ***member functions***.
- **Data members** are the data variables and **member functions** are the functions used to manipulate these variables
- Together these data members and member functions define the **properties** and **behaviour** of the objects in a Class.

Class Relationships

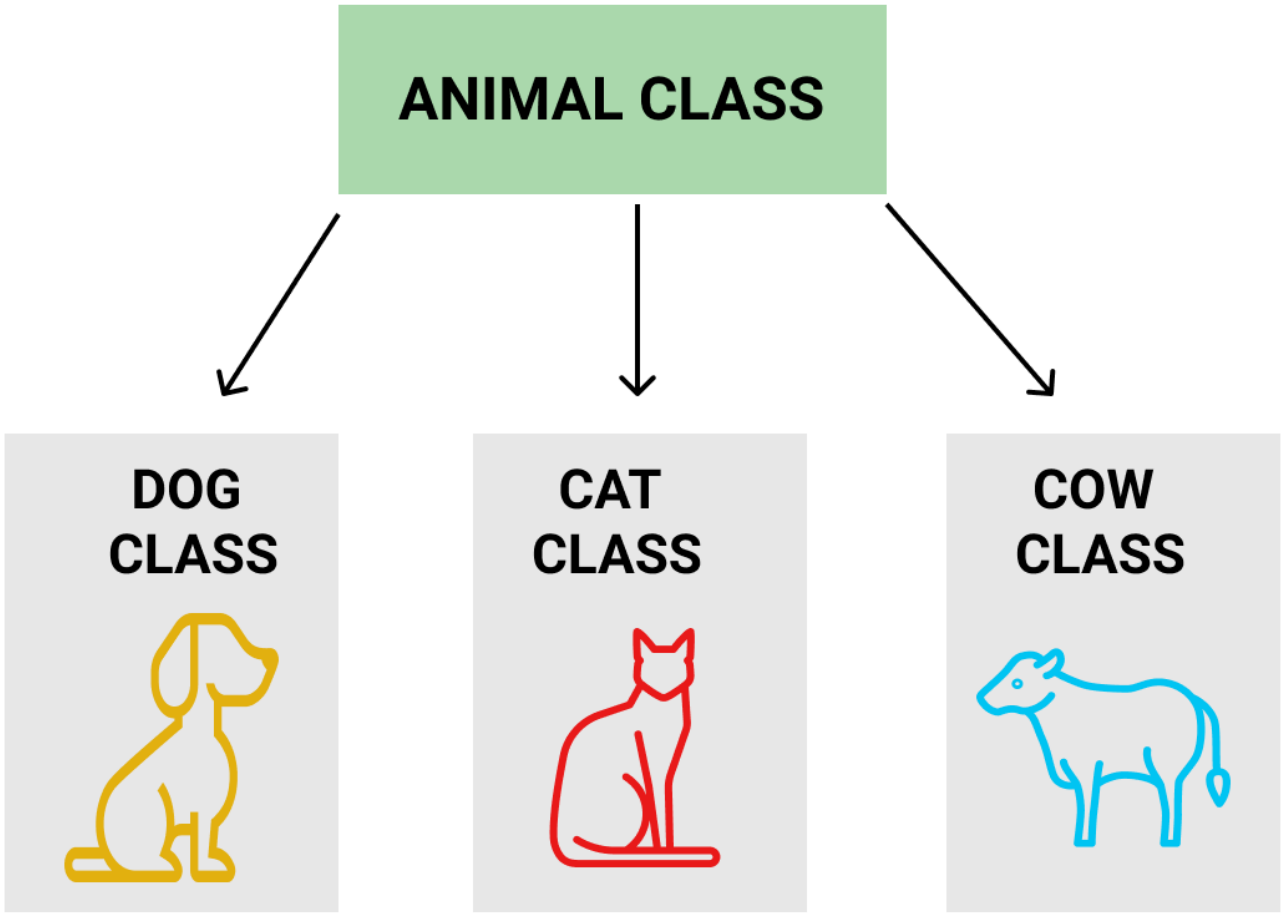
- Classes in a program can be related to each other in 4 ways:
  - Inheritance
  - Association and Link
  - Aggregation and Composition
  - Dependency

Inheritance

Inheritance

- It is the process by which ***objects of one class acquire the properties of objects of another class***.
- Each **derived class** shares **common characteristics** with the class from which it is derived
- Each derived class can be considered as a specialisation of its base class because it modifies or extends the basic properties of the base class in certain ways.
- Therefore, the **inheritance** relationship can be viewed as a ***generalisation-specialisation*** relationship.

Inheritance



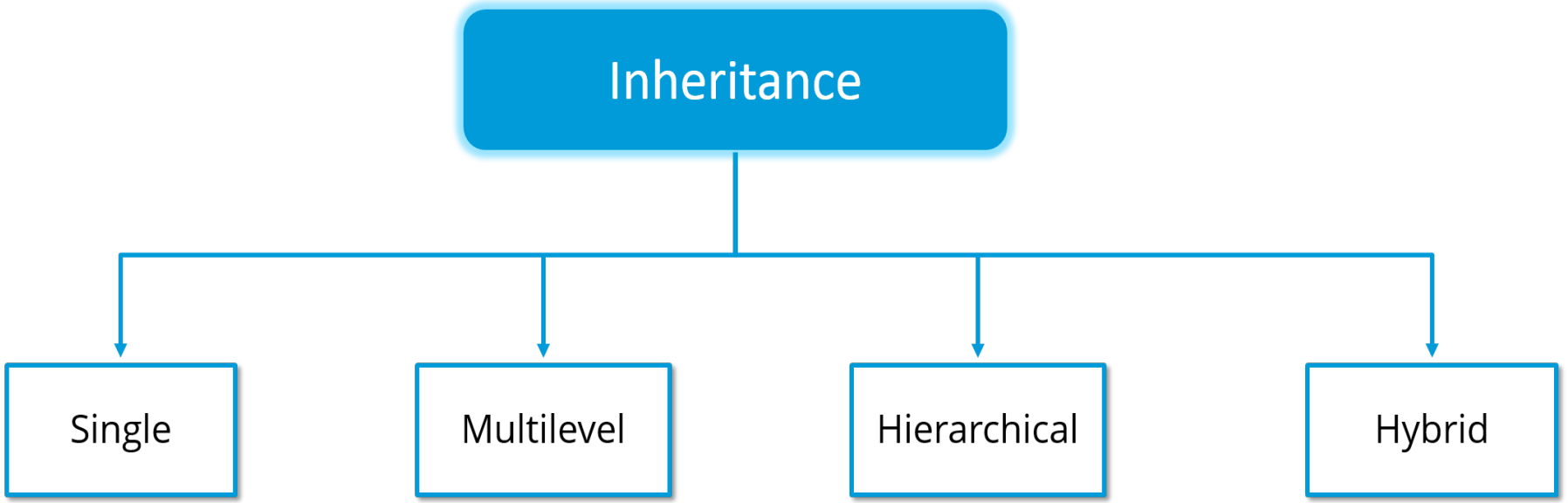
Inheritance

- In OOP, the concept of **inheritance** provides the idea of **reusability** and **simplicity**.

- Without the use of inheritance, each class would have to explicitly include all of its features.

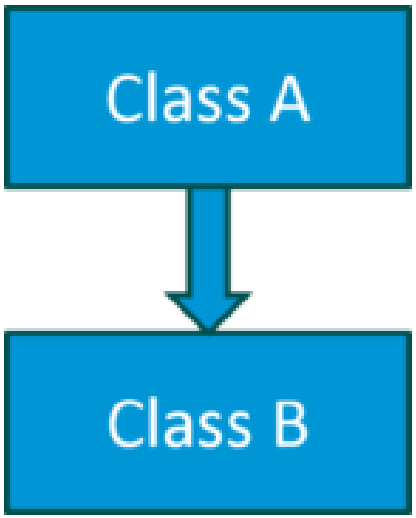
Inheritance

- Types of Inheritance (Java):



Inheritance

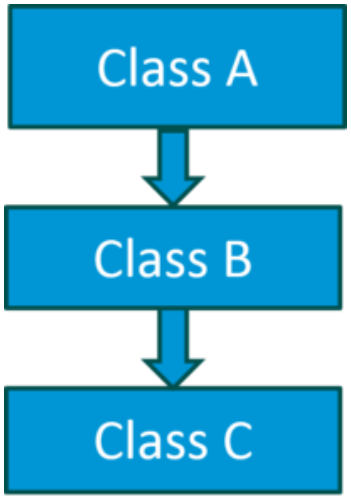
- Single Inheritance:
  - It enables a derived class to inherit the properties and behavior from a single parent class
  - Here, Class A is your parent class and Class B is your child class which inherits the properties and behavior of the parent class.




```
Class A
{
---
}
Class B extends A
{
---
}
```

Inheritance

- Multilevel Inheritance:

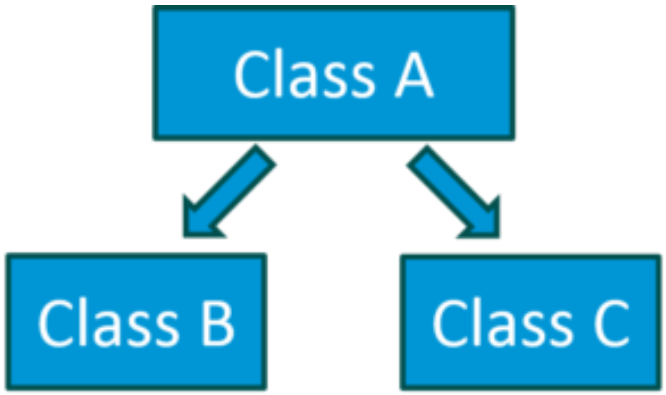


- When a class is derived from a class which is also derived from another class, i.e.
- A class having more than one parent class but at different levels, such type of inheritance is called Multilevel Inheritance.
- Here, A is the parent class for B and class B is the parent class for C.
- So in this case **class C implicitly inherits the properties and methods of class A along with Class B**

```
Class A{
---
}
Class B extends A{
---
}
Class C extends B{
---
}
```

Inheritance

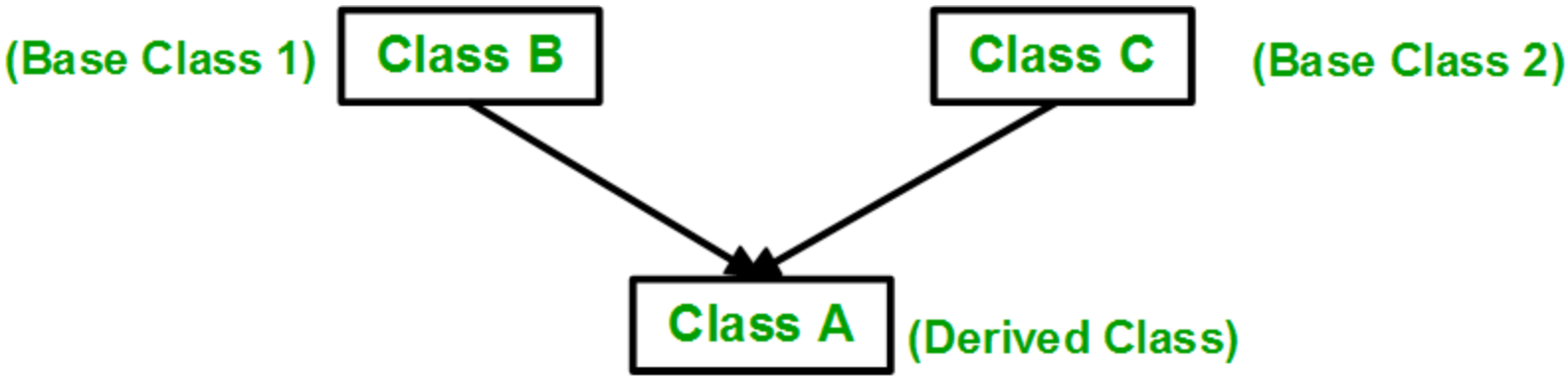
- Hierarchical Inheritance:
  - When a class has more than one child classes (sub classes) or in other words, more than one child classes have the same parent class
  - Here, Class B and C are the child classes which are inheriting from the parent class i.e Class A.




```
Class A{
---
}
Class B extends A{
---
}
Class C extends A{
---
}
```

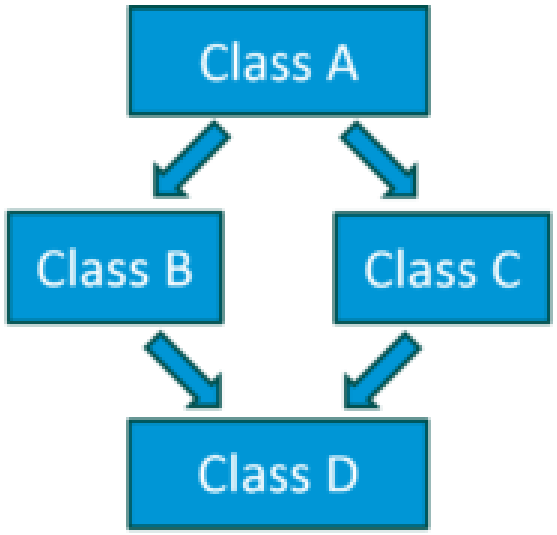
Inheritance

- **Multiple** Inheritance:
  - **A class can inherit from more than one classes**. i.e one derived class is inherited from more than one base classes.
  - Small Talk, Java, C# **do not** support Multiple inheritance.
  - Multiple Inheritance is supported in C++.



Inheritance

- Hybrid Inheritance:
  - Hybrid inheritance is a combination of **multiple** inheritance and **multilevel** inheritance.
  - Since multiple inheritance is **not supported** in Java as it leads to ambiguity, so ***this type of inheritance can only be achieved through*** the use of the ***interfaces***.
  - Class A is a parent class for class B and C, whereas Class B and C are the parent class of D which is the only child class of B and C.

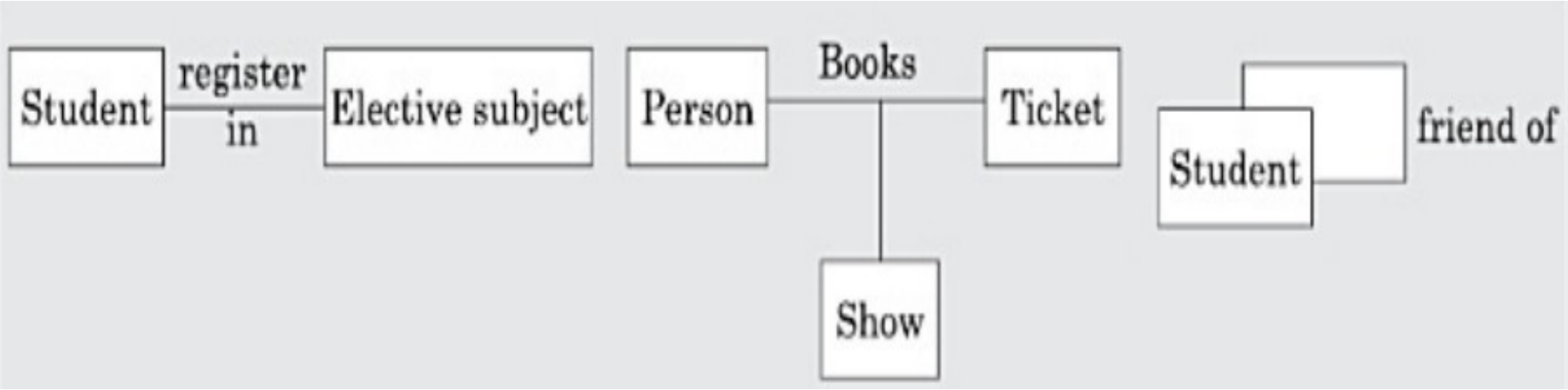


Association and Link

- **Association** is a common type of **relation among classes**.
- When two classes are associated, they can take each others help (i.e. invoke each others methods) to serve user requests
- It becomes possible for the **object of one class to invoke the methods of the corresponding object of the other class**.
- When two classes are associated, the **relationship between two objects of the corresponding classes** is called a ***link***.

Association and Link

- An ***association*** describes a *group of similar links*, **or**
- A ***link*** can be considered as an *instance of an association* relation.
- **Unary**, **Binary**, **Ternary**, and **n-ary** associations
- In ***unary*** association, two (or more) different objects of the same class are linked by the association relationship.



Association and Link

- Eg: "**A person works for a company**. Ram works for Infosys. Hari works for TCS."
- A **person** works for a **company** – association relationship
- **Ram** works for **Infosys – link** '*work for*' exists
- **Hari** works for **TCS – link** '*work for*' exists
- If **two classes are associated**, then the association relationship exists at all points of time (***static*** in nature).
- In contrast, **links between objects are *dynamic*** in nature.
- **Links** between the objects of the associated classes can get ***formed*** and ***dissolved*** as the program executes.

Composition and Aggregation

- Objects which contain other objects are called ***composite*** objects.

- ***Book*** object is said to be composed of upto ten ***Chapter*** objects, ***or,*** A ***Book*** *has* upto ten ***Chapter*** objects
- Composition/aggregation relationship is also known as ***has a*** relationship
- Aggregation/composition can occur in a **hierarchy** of levels
- An object contained in another object may itself contain some other object

Dependency

- A class is said to be ***dependent*** on another class, if any changes to the latter class necessitates a change to be made to the dependent class.
- Dependencies among classes may arise due to:
  - A method of a class takes an object of another class as an argument.
  - A class implements an interface class. If some properties of the interface class are changed, then a change becomes necessary to the class implementing the interface class as well.

How to Identify Classes & their Relationships?

- This can be done by a **careful analysis of the sentences given** in the problem description
- The **nouns** in a sentence often denote the ***classes***.
- The ***relationships*** among classes are usually indicated by the presence of certain **keywords**
- Example: Consider two classes A and B.
- **Composition:**
  - B is a permanent part of A
  - A is made up of Bs
  - A is a permanent collection of Bs

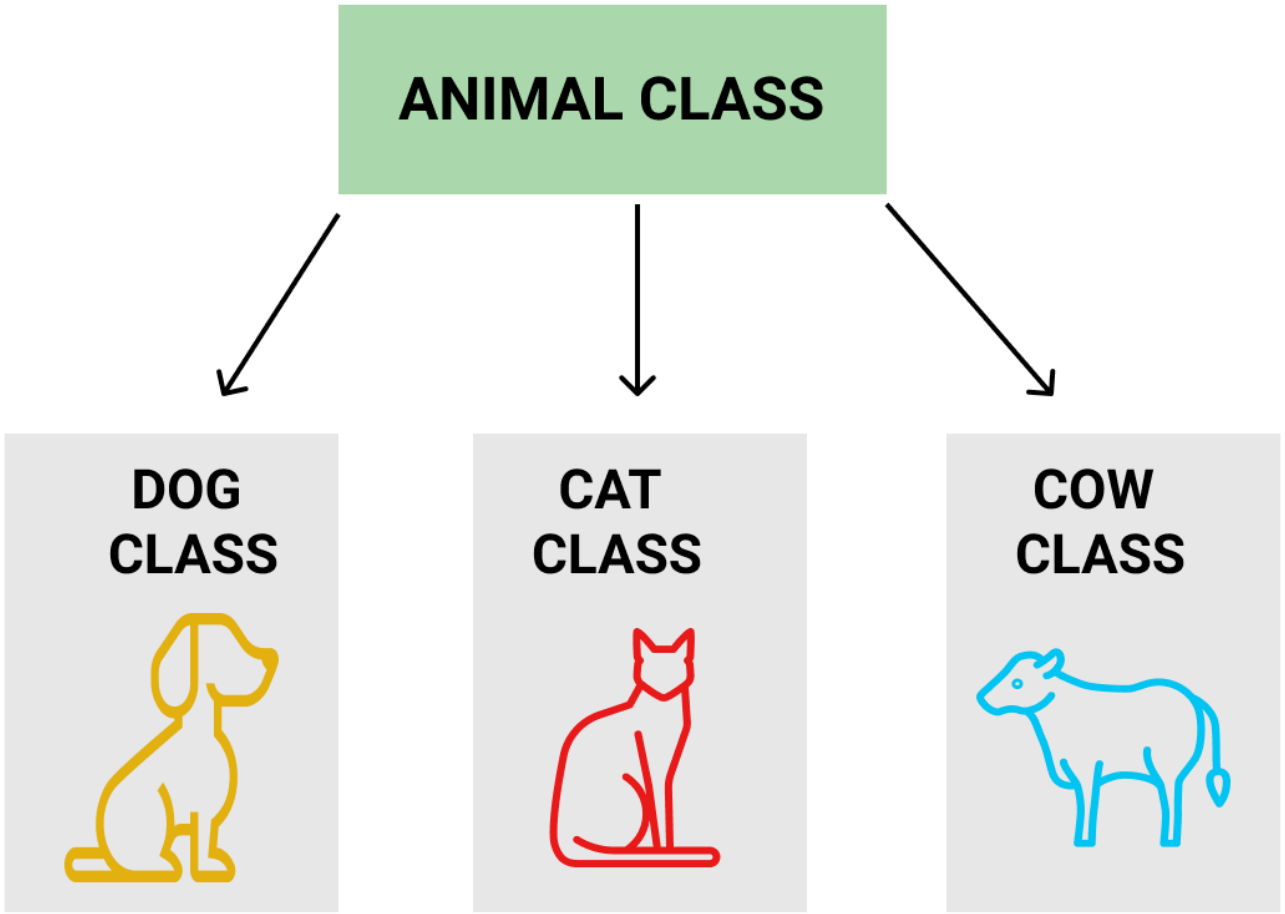
How to Identify Classes & their Relationships?

- **Inheritance:**
  - A is a kind of B
  - A is a specialisation of B
  - A behaves like B
- **Association:**
  - A delegates to B
  - A needs help from B
  - A collaborates with B

Abstraction

- Abstraction refers to the act of representing **essential** features **without including the background details** or **implementations**.
- Uses **only essential information** to describe the properties of an object
- Abstraction is supported in two ways in an OOD.
  - Feature Abstraction
  - Data Abstraction

Feature Abstraction - Inheritance



Feature Abstraction

- A **class hierarchy** can be viewed as defining several levels (hierarchy) of abstraction, where **each class is an abstraction of its subclasses.**
- Every class is a simplified (abstract) representation of its derived classes and retains only those features that are common to all its children classes and ignores the rest of the features.
- Thus, the **inheritance** mechanism can be thought of as providing ***feature abstraction***.

Data Abstraction

- Data abstraction implies that each object hides (abstracts away) from other objects the exact way in which it stores its internal information
- Each object only provides a set of methods, which other objects can use for accessing and manipulating this private information of the object. Eg. Stack
- **Advantage:** It reduces coupling among various objects, and leads to a reduction of the overall complexity of a design, and helps in easy maintenance and code reuse.

Abstraction

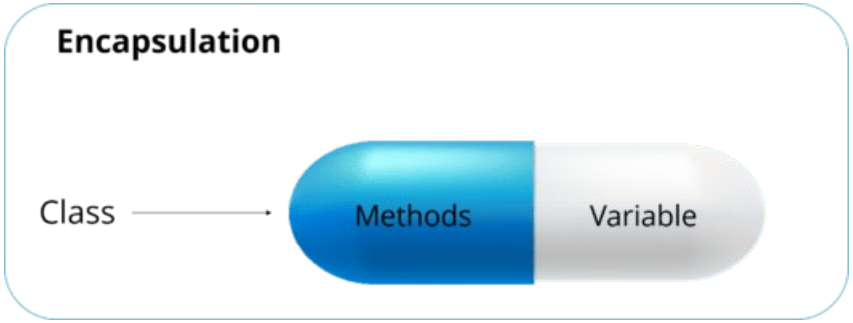
- A real-life example: **a man driving a car**.
- The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car
- but he does not know about how on pressing accelerator the speed is actually increasing,
- he does not know about the inner mechanism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

Abstraction

- Ways to achieve abstraction:
  - Abstract Class
  - Interface
- ***abstract methods***
  - Only has function declaration
  - Function is defined somewhere else

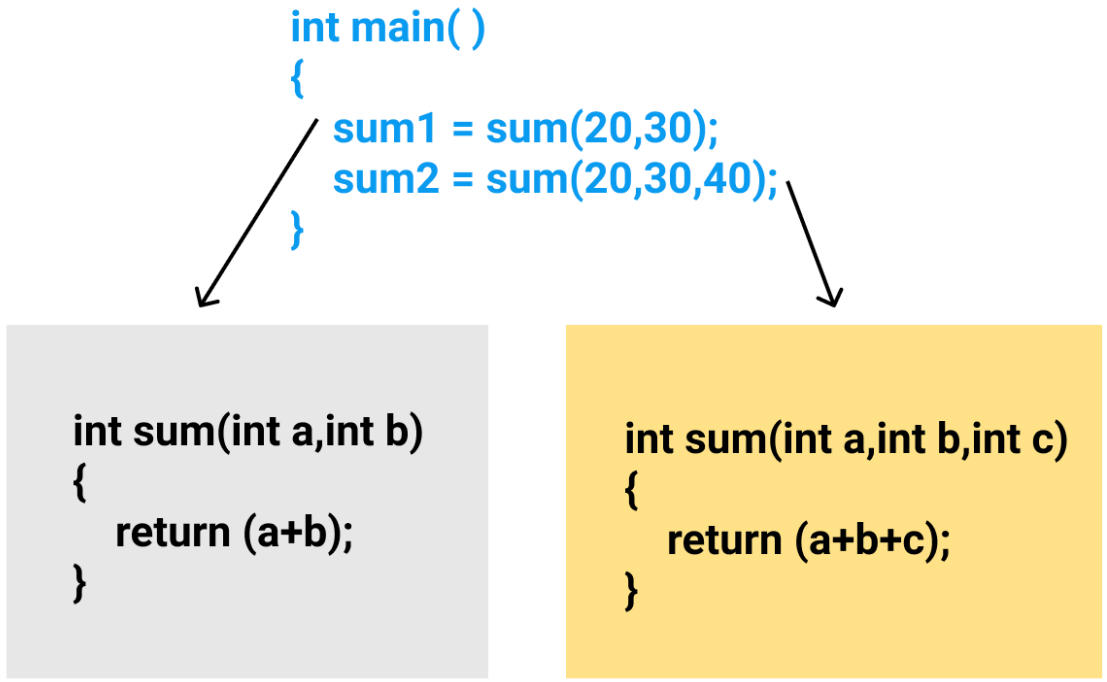
Data Encapsulation

- The ***wrapping up of data and methods into a single unit*** (called ***class***) is known as **encapsulation**.
- The data is **not accessible** to the outside world and only those methods, which are wrapped in the class can access it
- These methods provide the interface between the objects's data and the program.
- **This *insulation of the data from direct access by the program*** is called **data hiding**.



Polymorphism

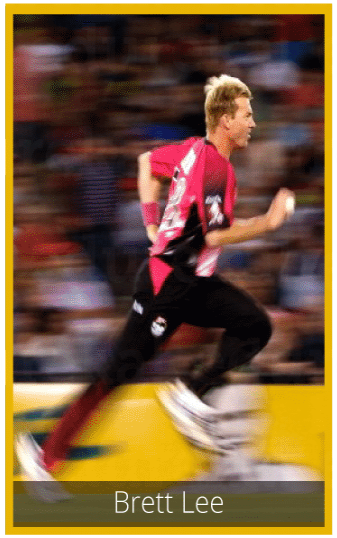
- It is the property of an object which allows it to take multiple forms.
- Examples:·



Polymorphism

Bowler Class : (Parent Class)
-bowlingMethod()

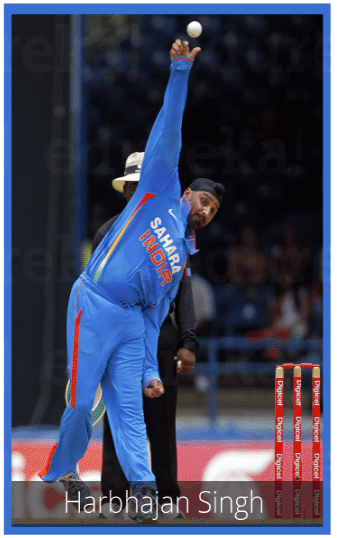
FastPacer :(Child Class)
-Bowling Method()


Brett Lee

MediumPacer :(Child Class)
-Bowling Method()


Irfan Pathan

Spinner :(Child Class)
-Bowling Method()



Harbhajan Singh

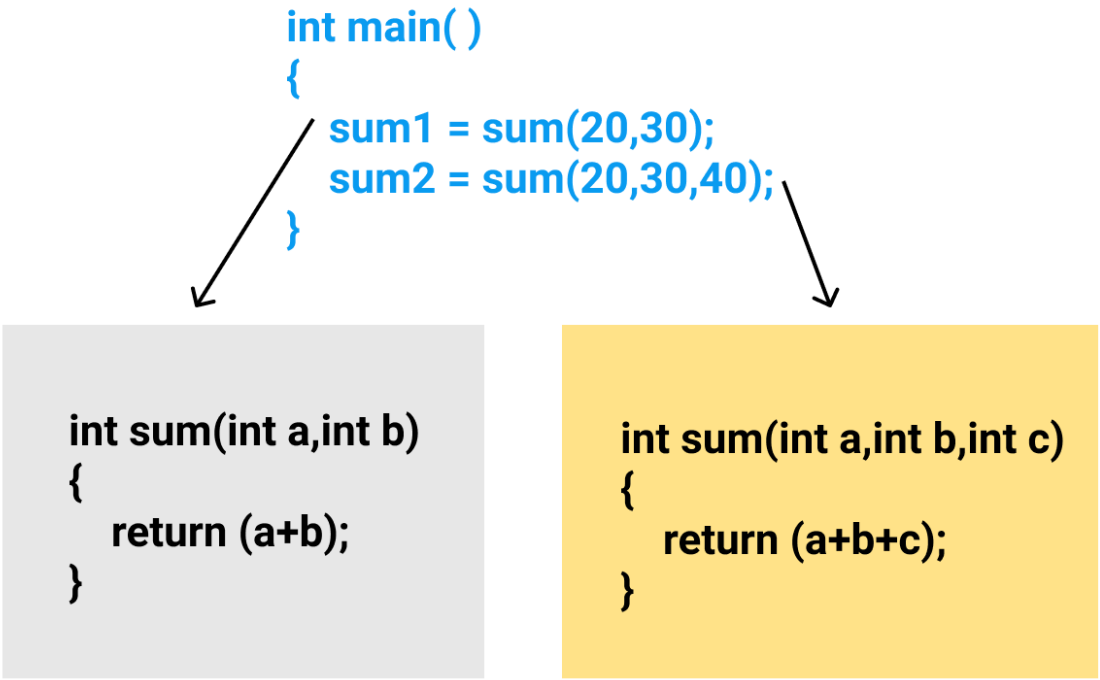
Polymorphism

- Types: Compile time / Static and Run time / Dynamic

  - Compile Time Polymorphism – eg. function **overloading**
  - **Rules** for Overloading:
    - Overload methods must have **different argument list**
    - It can have **different return types** if argument list is different
    - It can **throw different exceptions**
    - It can have **different access modifiers**

Polymorphism

- Compile Time Polymorphism – eg. function **overloading**

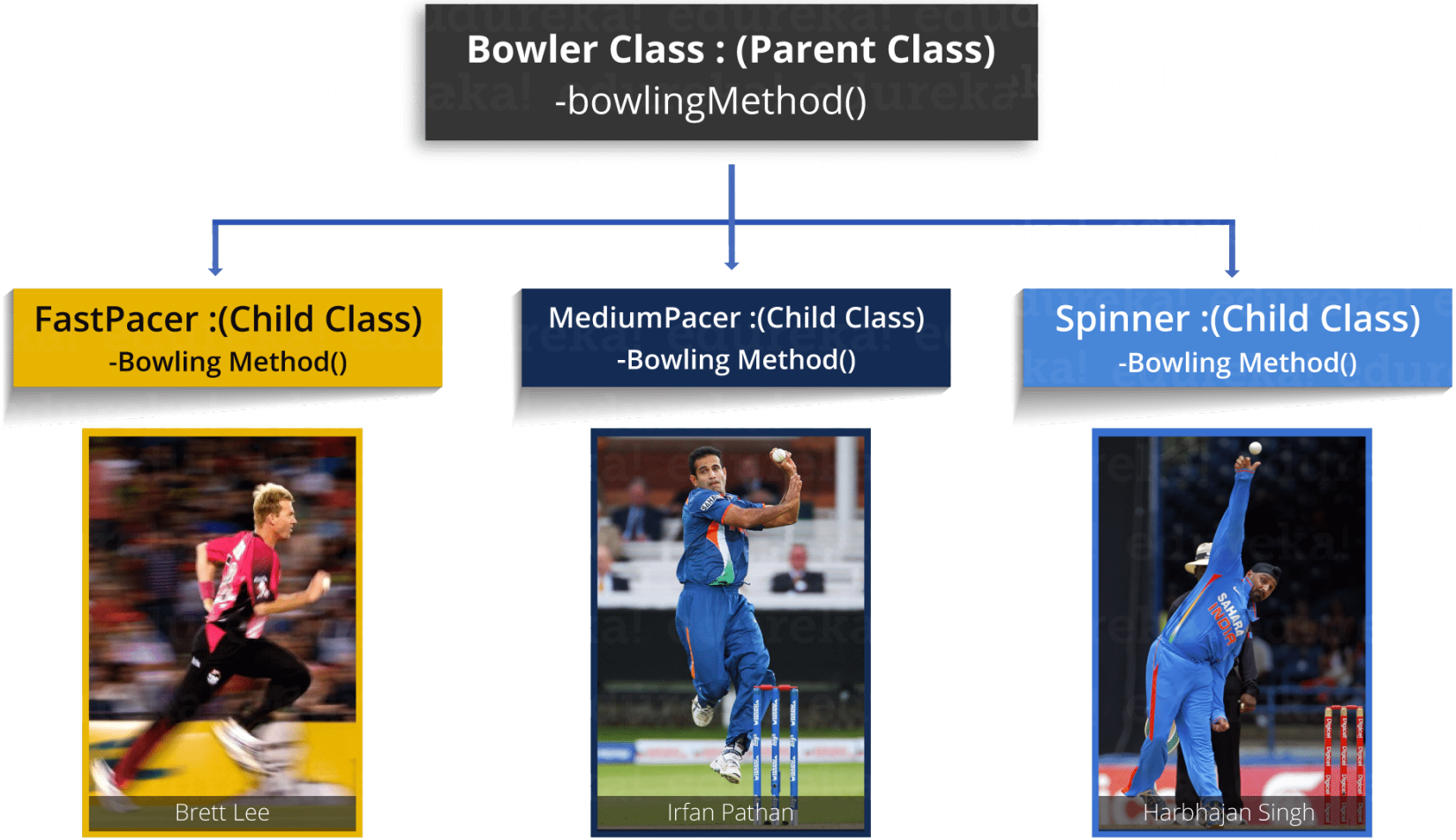


Polymorphism

- **Dynamic Polymorphism** is resolved during **run time**
- Eg: function **overriding –** function body is different
- An overridden method is called through the reference variable of a superclass
- **Rules** for overriding:
  - Overriding method **argument list must match** the overridden method
  - The **return type must be the same** or **subtype** of overridden method
  - **Access level cannot be more restrictive** than overridden method

Polymorphism

- Dynamic Polymorphism - Eg: function **overriding –** function body is different



Genericity

- ***Genericity is the ability to parameterise class definitions.***
- Eg: while defining a **class stack** of different types of elements such as **integer stack**, **character stack**, **floating-point stack**, etc.,
- ***Genericity*** permits us to define a ***generic class*** **of type stack** and later **instantiate** it either as an integer stack, a character stack, or a floating-point stack as may be required.
- This can be achieved by assigning a **suitable value to a parameter** used in the ***generic class definition***.

Advantages of OOD

- Simplicity - Software complexity can be easily managed
- Reusability
- Extendability - Object oriented system can be easily upgrade from small system to large systems
- It is easy to partition the works in a project based on objects
- Message passing technique make communication easier
- Maintenance cost is less
- Security

Disadvantages of OOD

- Incurs ***run-time overhead*** due to the additional code that gets generated on account of features like abstraction, data hiding, inheritance, etc.
- **Data gets scattered across various objects** in an object-oriented implementation
  - **Spatial locality of data** becomes **weak** and this leads to **higher cache miss ratios** and consequently to **larger memory access times**.
  - This finally shows up as **increased program run time**.