JAVA FULL STACK DEVELOPMENT PROGRAM

Session 2: Java OOP

OUTLINE

- Object
- Object-oriented programing(OOP)
    - Inheritance
    - Polymorphism
    - Abstract
    - Encapsulation

OBJECT

- An object is a combination of data and procedures working on the available data
- An object has a state and behavior
- The state of an object is stored in fields (variables), while methods (functions) display the object's behavior

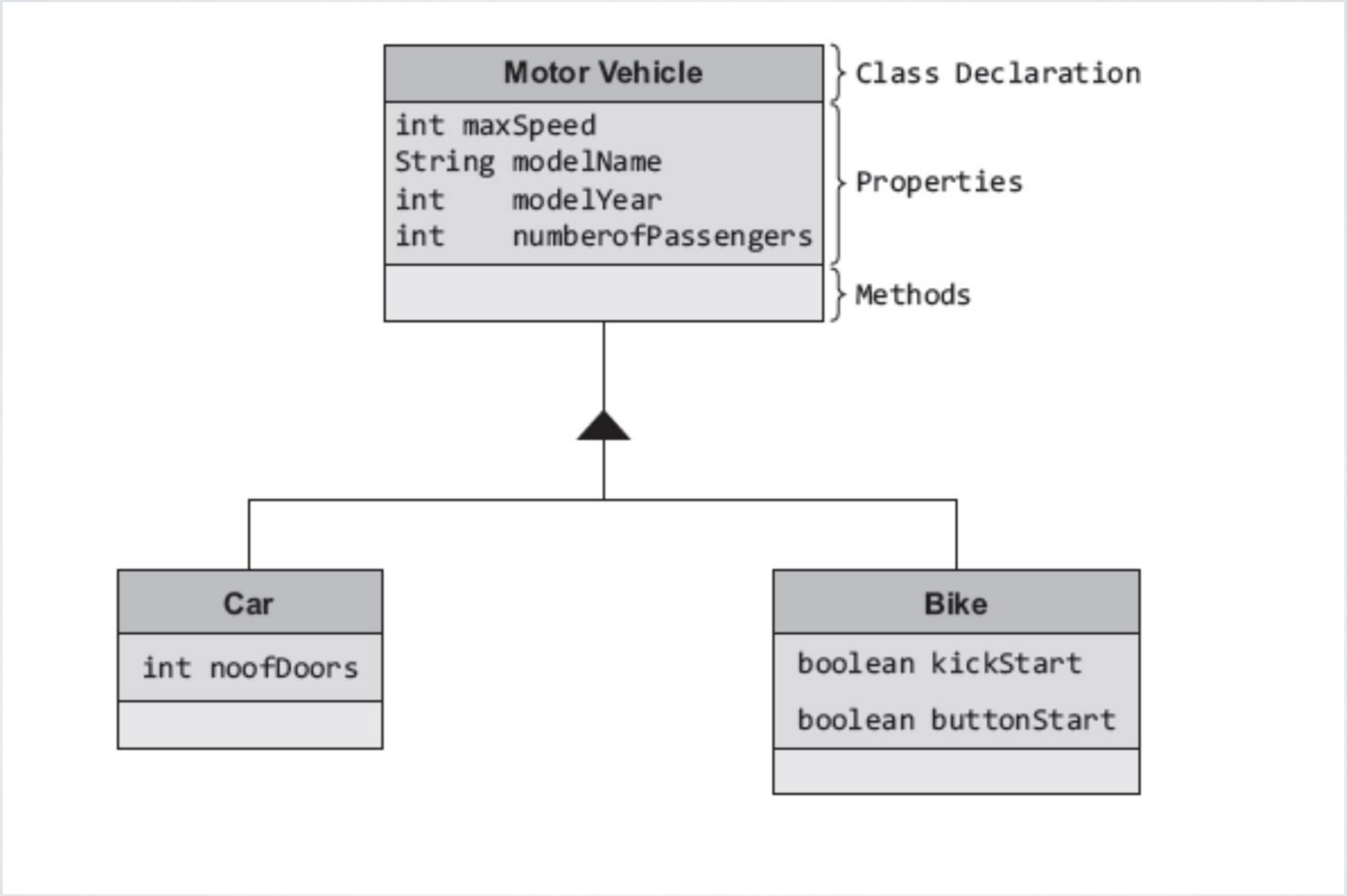
OBJECT-ORIENTED PROGRAMMING

- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

INHERITANCE

- Is the ability to derive something specific from something generic.
- Aids in the reuse of code.
- A class can inherit the features of another class and add its own modification.
- The parent class is the *super class* and the child class is known as the *subclass*.
- A subclass inherits all the properties and methods of the super class.

INHERITANCE

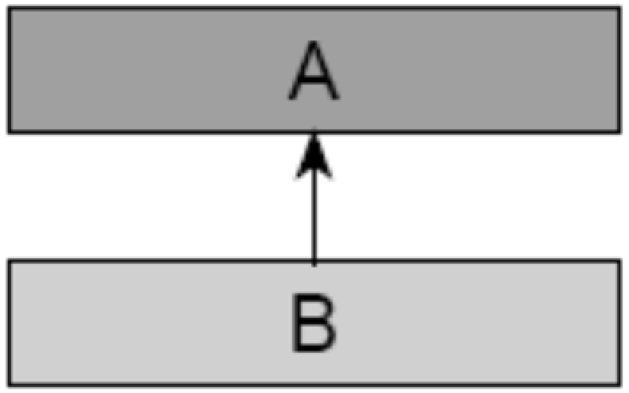


TYPE OF INHERITANCE

- Single
- Multiple
- Multilevel
- Hierarchy
- Hybrid

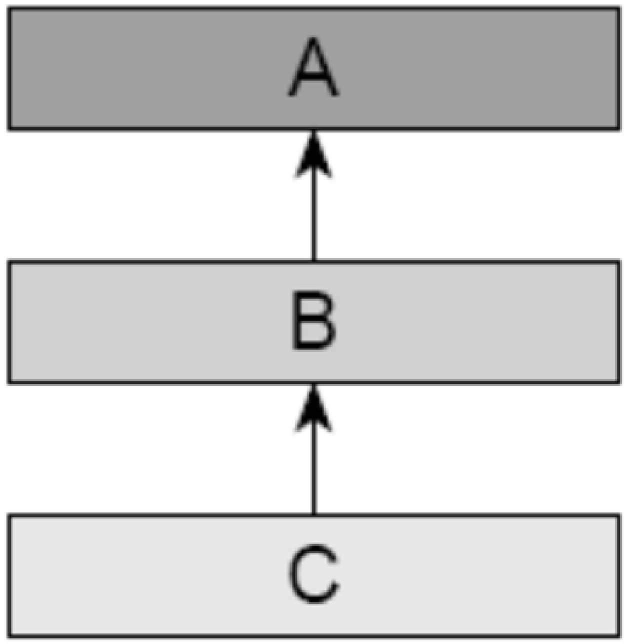
SINGLE INHERITANCE

Classes have only base class



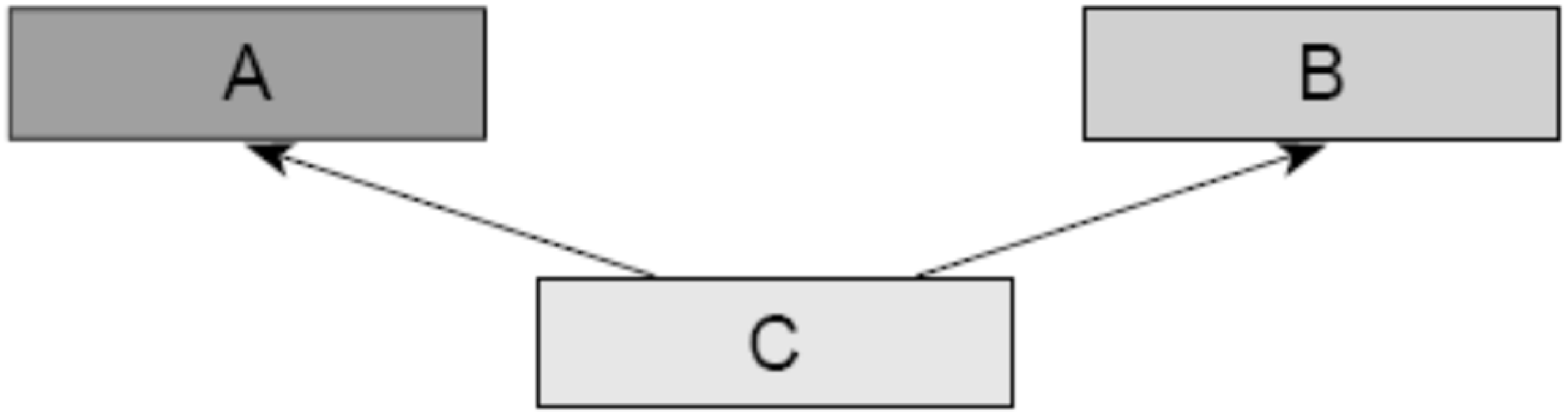
MULTI LEVEL

There is no limit to this chain of inheritance (as shown below) but getting down deeper to four or five levels makes code excessively complex.



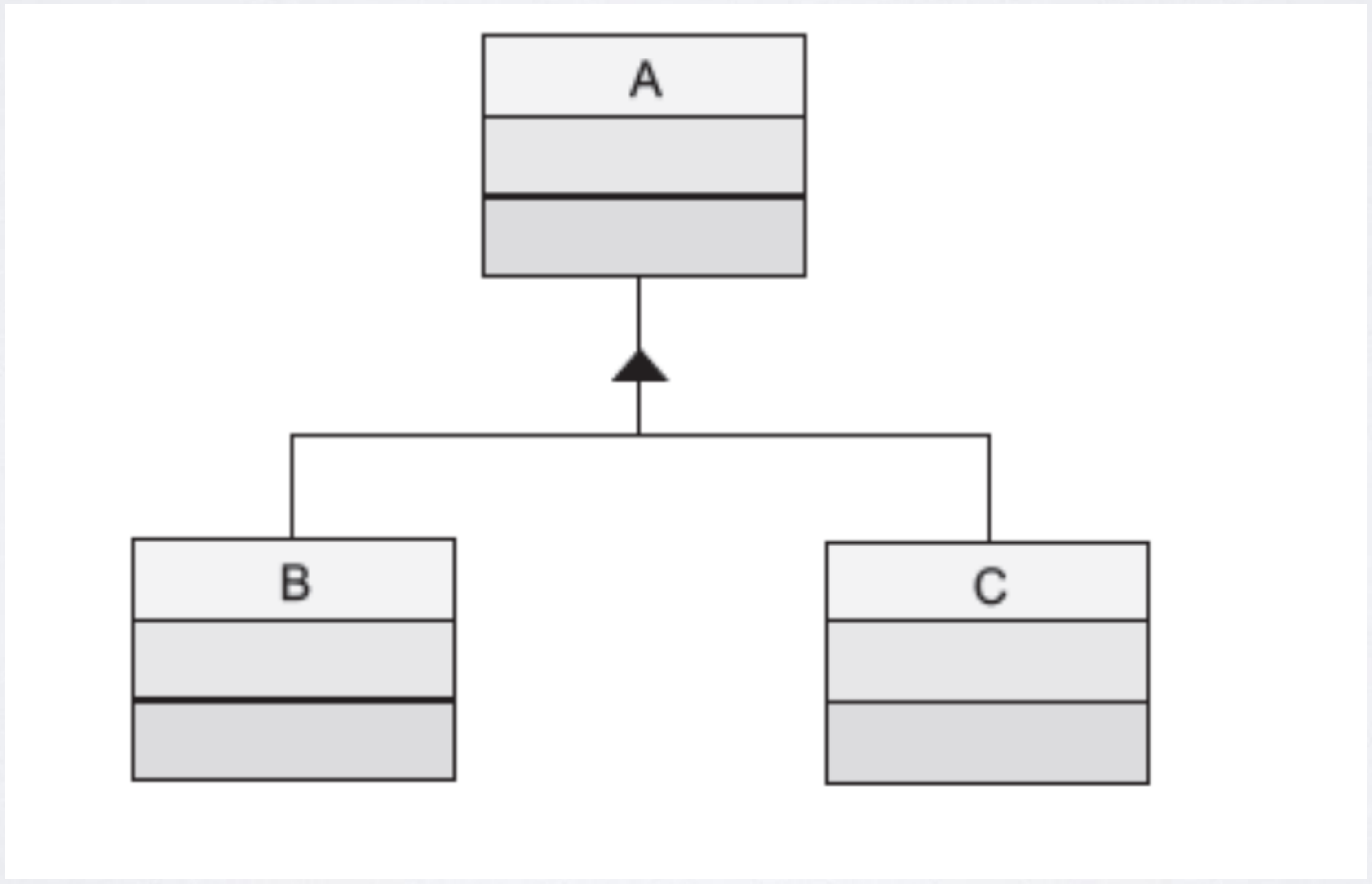
MULTIPLE INHERITANCE

A class can inherit from more than one unrelated class



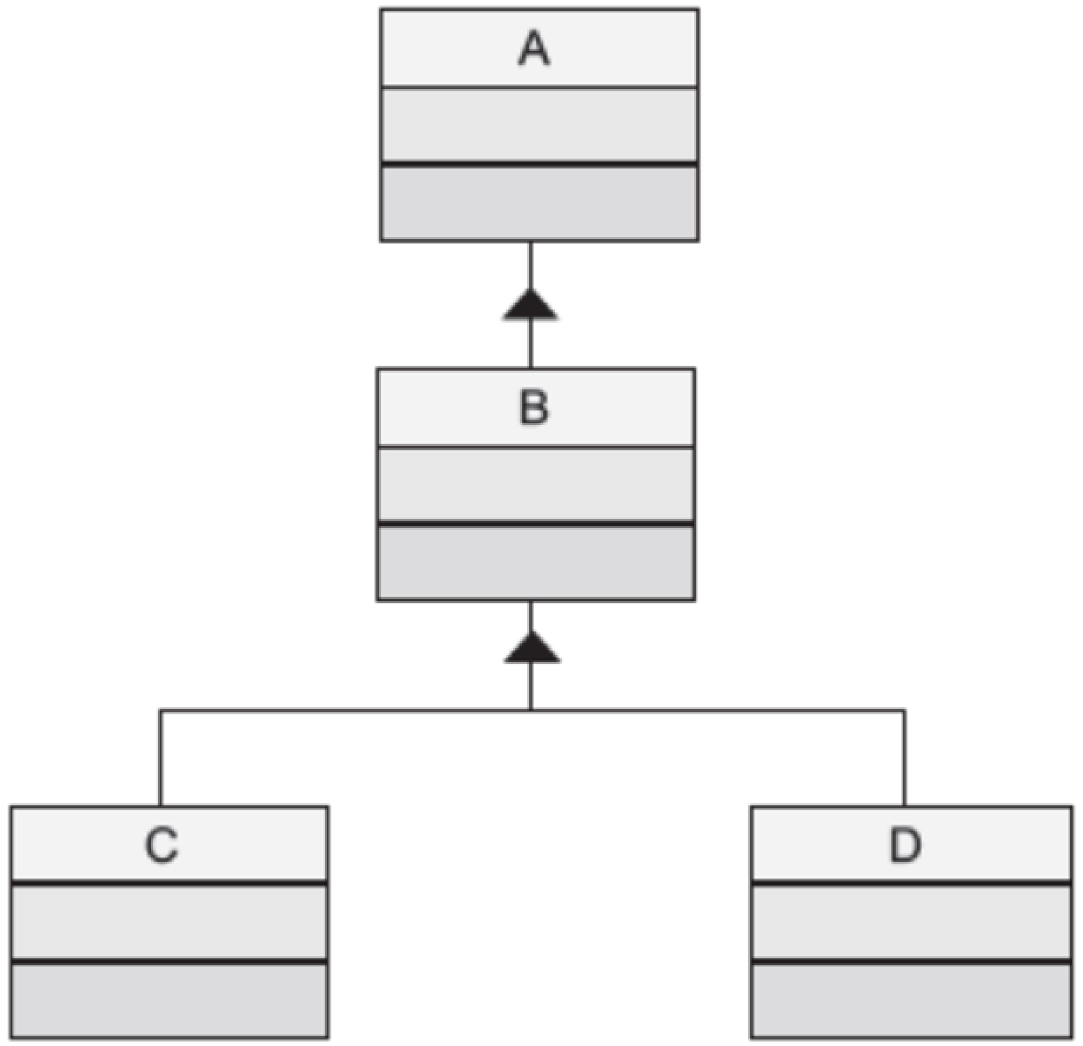
HIERARCHY INHERITANCE

- In hierarchical inheritance, more than one class can inherit from a single class



HYBRID INHERITANCE

any combination of the above defined inheritances



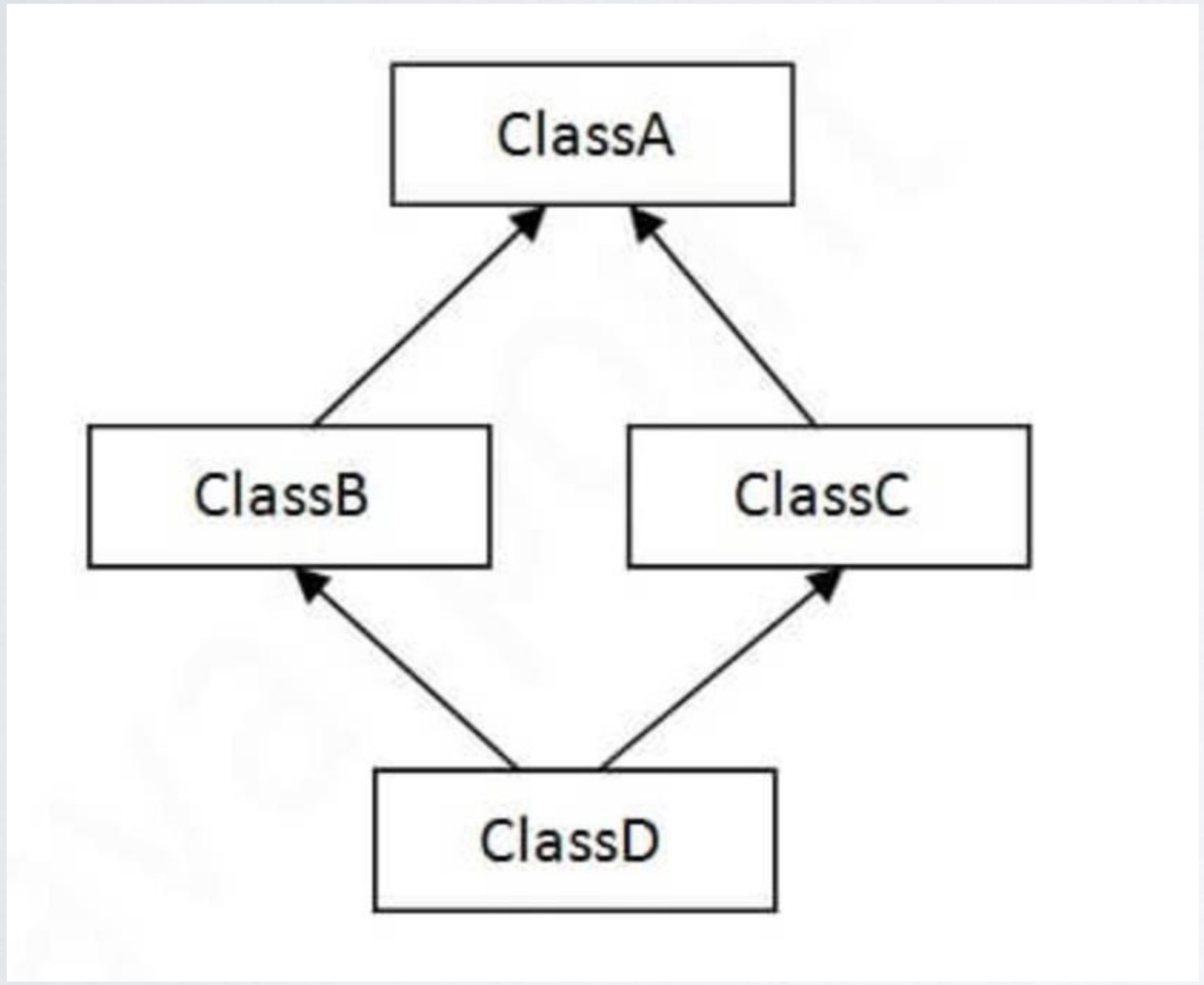
INHERITANCE IN JAVA

- Syntax
  - class MySubClass **extends** MySuperClass
  - *extends* keyword declares that *MySubClass* inherits the parent class *MySuperClass*.
- IS-A relationship
  - Inheritance in Java implies that there is an IS-A relationship between subclass and super class
  - eg. Car is a Vehicle; Dog is an animal; Triangle is a shape

INHERITANCE IN JAVA

- Java supports following inheritance:
  - Single
  - Multi-level
  - Hierarchy
- How about Multiple and Hybrid?

DIAMOND PROBLEM



- An ambiguity that can arise as a consequence of allowing multiple inheritance

INTERFACE

- Like a class, an interface can have methods and variables, but the methods declared in interface are by default abstract (only method signature, no body)
  - Interfaces specify what a class must do and not how. It is the blueprint of the class.
  - It specifies a set of methods that the class has to implement
  - If a class implements an interface and does not provide method bodies for all functions specified in the interface, then class must be declared abstract.

INTERFACE

- Syntax
  - public *interface* Shape{}
  - public Triangle *implements* Shape {}
- Java allows **multiple** inheritance in Interface
  - Why?

AGGREGATION / COMPOSITION

- If a class have an entity reference, it is known as Aggregation
- Aggregation represents HAS-A relationship
  - Consider a situation, Employee object contains many informations such as id, name, emailId etc. It contains one more ***object*** named ***address***, which contains its own informations such as city, state, country, zipcode etc.
  - In such case, Employee has an entity reference address, so relationship is Employee HAS-A address.

WHY AGGREGATION

- Code reuse is also best achieved by aggregation when there is no is-a relationship.
  - If we don’t have the Address object
    - Have to add all street, state information in employee
    - If we introduce an object called Company later which has the address too, we will have to add same attributes to Company object again

AGGREGATION

- How does Aggregation solve the diamond problem?
    - Instead of extends class B and C, introducing the B and C as the instance variables in class D
    - Thus, we can use b.doSomething() or c.doSomething() to eliminate the ambiguity

INHERITANCE VS. AGGREGATION

- Inheritance should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.
  - Flexibility of aggregation
    - eg. the method return type changed in super class: if inheritance, we have no choice but to change our implementation; if aggregation, simply change the type of variable which stores the return value of super class.
  - Unit testing (Mocking)

INHERITANCE

- How can we refer to current object and parent/super object in Java?

THIS

- ***this*** is a keyword that refers to the current object
- Usage
  - this can be used to refer current class instance variable.
  - this can be used to invoke current class method (implicitly)
  - this() can be used to invoke current class constructor.
  - this can be passed as an argument in the method call.
  - this can be used to return the current class instance from the method.

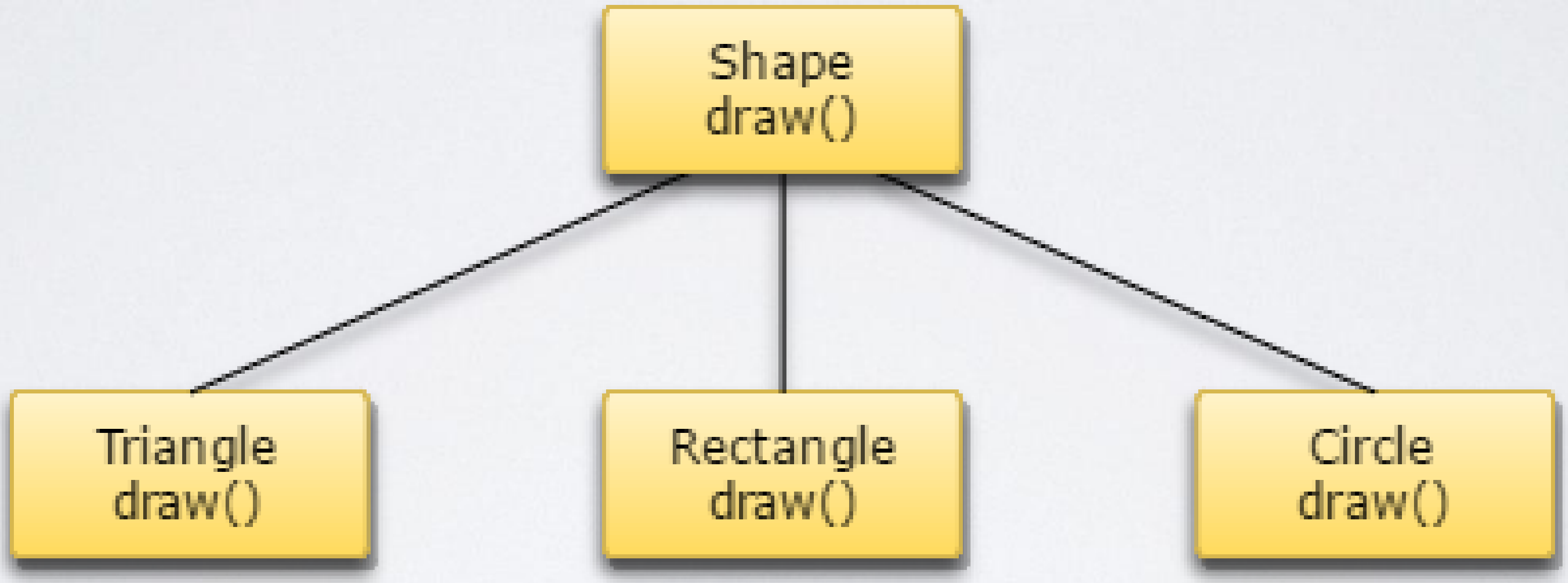
SUPER

- A reference variable which is used to refer immediate parent class object.
  - Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.
- Usage
  - super can be used to refer immediate parent class instance variable.
  - super can be used to invoke immediate parent class method.
  - super() can be used to invoke immediate parent class constructor.

POLYMORPHISM

- **Polymorphism in java** is a concept by which we can perform a *single action by different ways*.
  - The word "poly" means many and "morphs" means forms. So polymorphism means many forms.
- There are two types of polymorphism in java
  - Compile time polymorphism
  - Runtime polymorphism

POLYMORPHISM



Polymorphism

COMPILE TIME VS RUNTIME

- Compile time — the instance where the code you entered is converted to executable
    - eg. from .java to .class
- Runtime — the instance where the executable is running
    - eg. when the for loop get executed

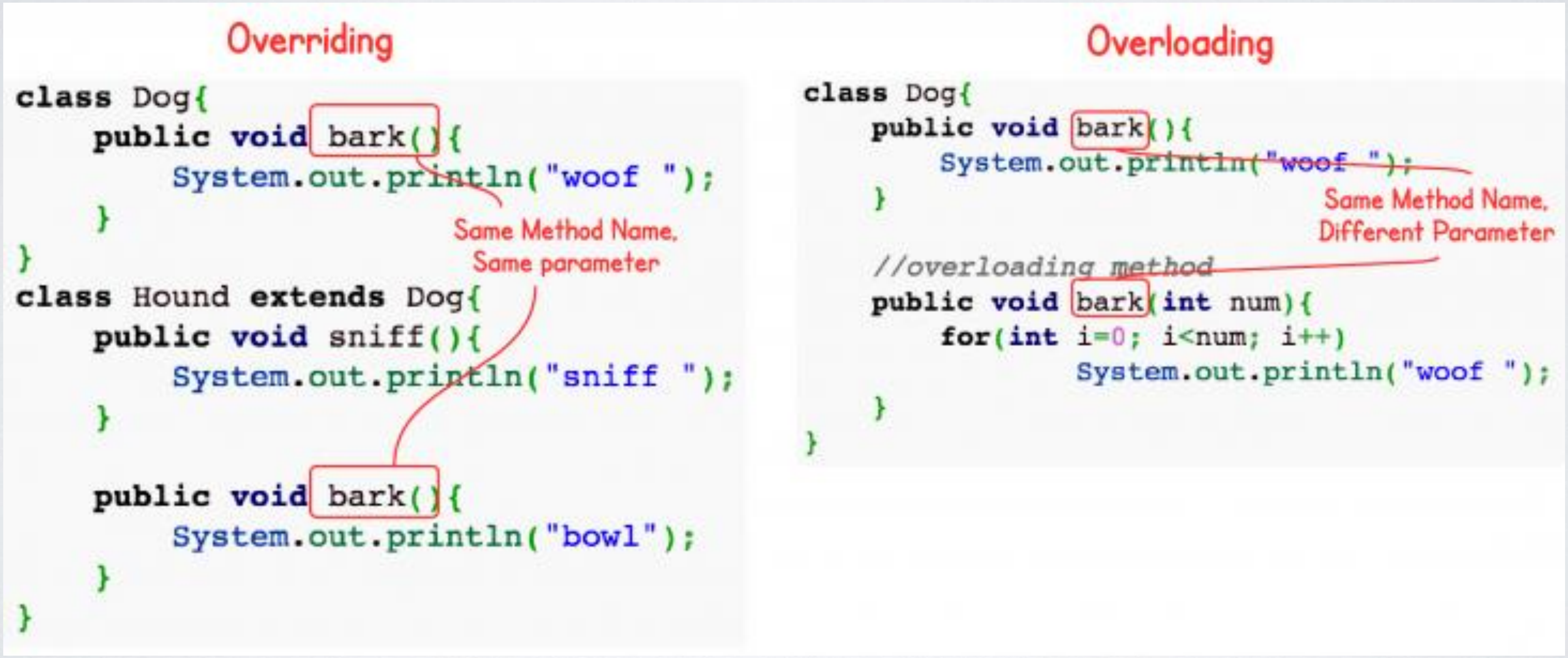
COMPILE TIME POLYMORPHISM

- The form is determined at compile time
- Method ***overloading***
  - a class has multiple methods having same name but different in parameters
    - Different number of parameters
    - Different data type of parameters
  - Method overloading increases the readability of the program

RUNTIME POLYMORPHISM

- The form is determined at runtime
- Method ***overriding***
  - a method in a *subclass* has the ***same name*** and ***return type*** as a method in its *superclass*, then the method in the subclass is said to *override* the method in the superclass
  - When an overridden method is called through the subclass object, it will always refer to the version of the method defined by the subclass. The superclass version of the method is hidden.

POLYMORPHISM



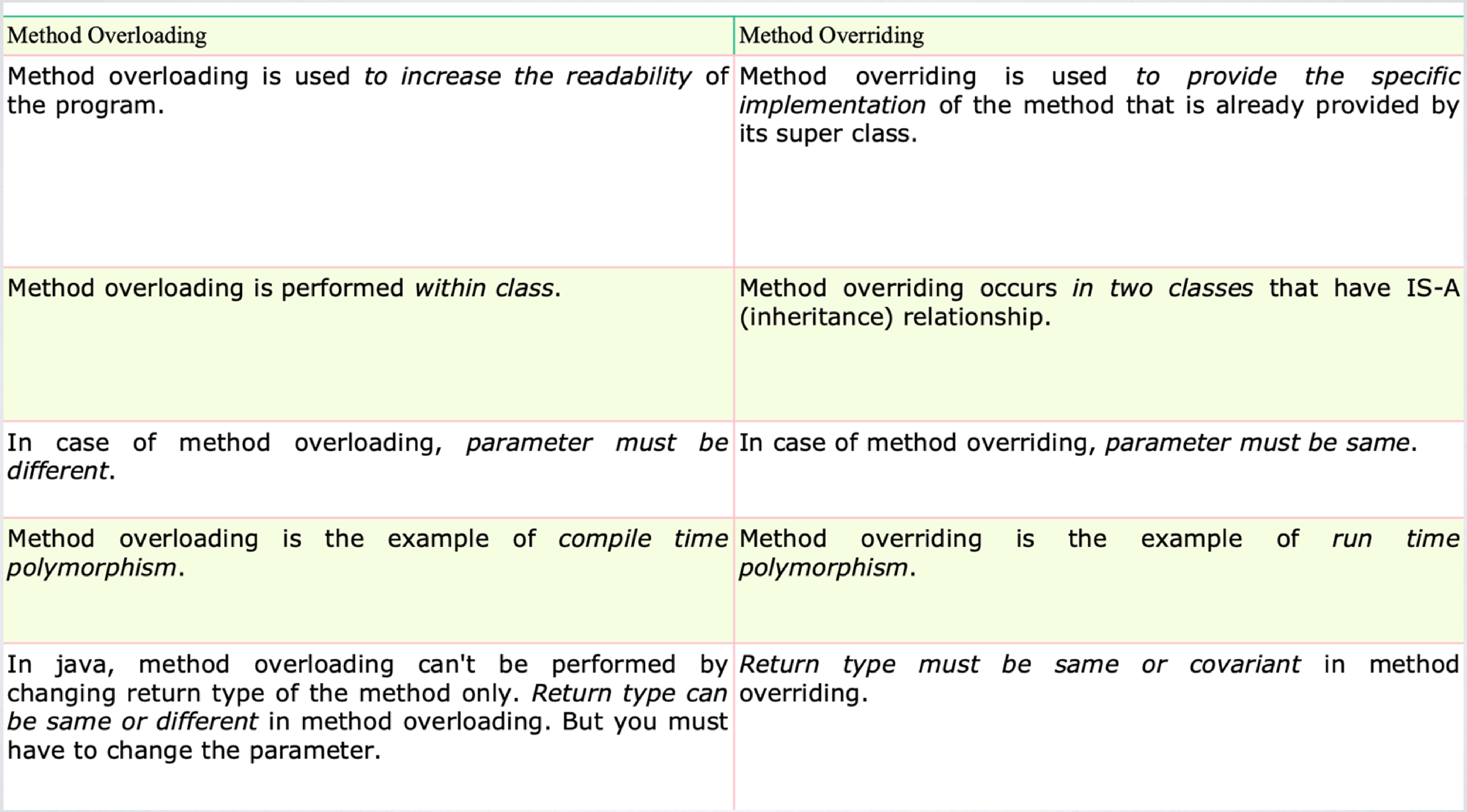
```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }
}
class Hound extends Dog{
    public void sniff(){
        System.out.println("sniff ");
    }

    public void bark(){
        System.out.println("bowl");
    }
}
```

```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }

    //overloading method
    public void bark(int num){
        for(int i=0; i<num; i++)
            System.out.println("woof ");
    }
}
```

OVERLOADING VS OVERRIDING



Method Overloading	Method Overriding
Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

ABSTRACTION

- Hiding internal details and showing functionality
  - eg. phone call, we don't know the internal processing
- Abstraction is achieved by creating either Abstract Classes or Interfaces on top of your class
- Why
  - Hide the unnecessary things from user so providing easiness.
  - Hiding the internal implementation of software so providing security.

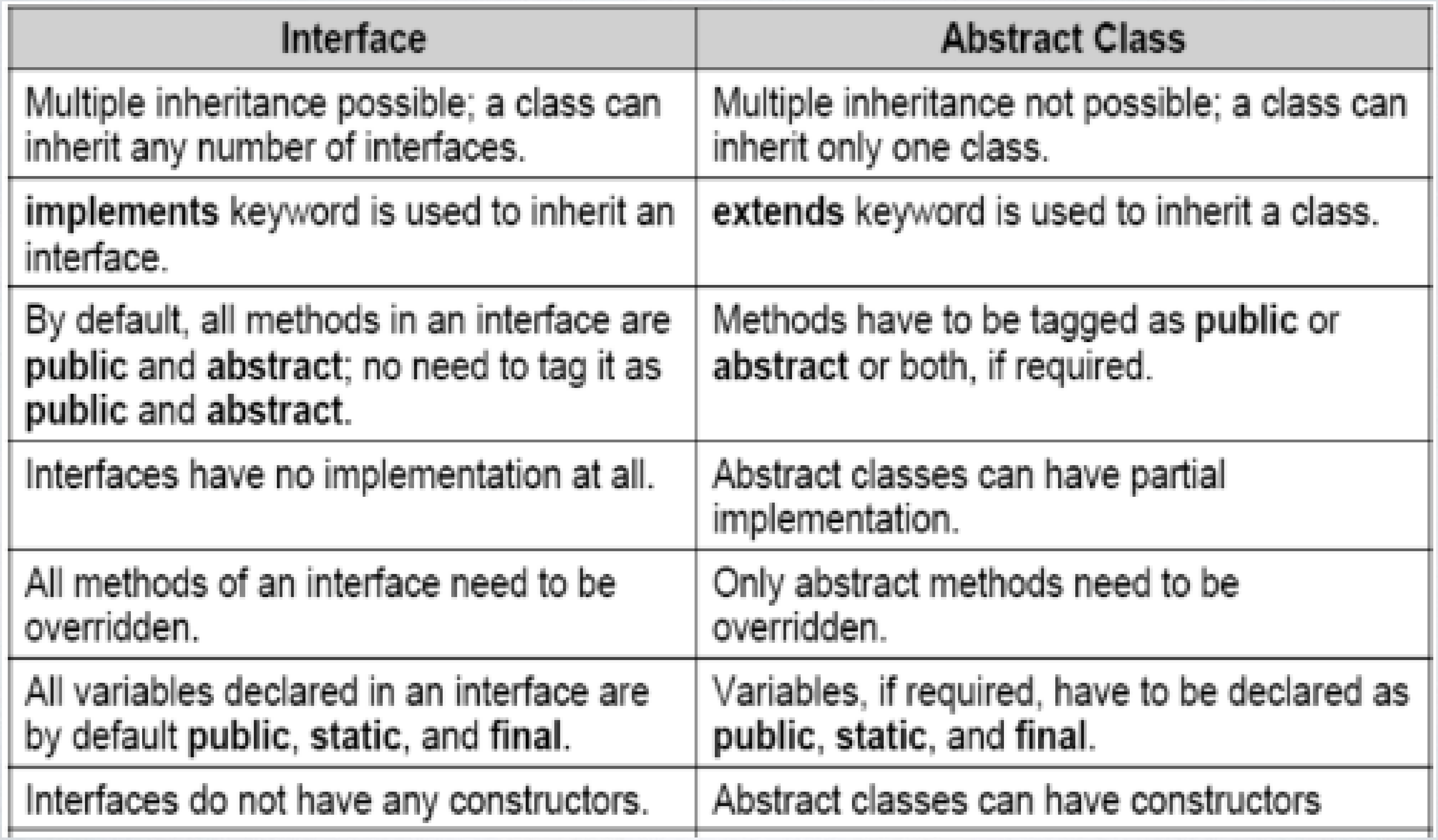
ABSTRACT CLASS

- Abstract classes are classes with a generic concept, not related to a specific class.
- Abstract classes define partial behavior and leave the rest for the subclasses to provide
- Contain one or more abstract methods.
- Abstract method contains no implementation (like method in Interface)
- Abstract classes cannot be instantiated, but they can have reference variable
- If the subclasses does not override the abstract methods of the abstract class, then it is mandatory for the subclasses to tag itself as *abstract.*

WHY ABSTRACT CLASS

- To force same name and signature pattern in all the subclasses
- To have the flexibility to code these methods with their own specific requirements
- To prevent accidentally initialization
- To define common attributes or methods

ABSTRACT CLASS VS INTERFACE



Interface	Abstract Class
Multiple inheritance possible; a class can inherit any number of interfaces.	Multiple inheritance not possible; a class can inherit only one class.
implements keyword is used to inherit an interface.	**extends** keyword is used to inherit a class.
By default, all methods in an interface are **public** and **abstract**; no need to tag it as **public** and **abstract**.	Methods have to be tagged as **public** or **abstract** or both, if required.
Interfaces have no implementation at all.	Abstract classes can have partial implementation.
All methods of an interface need to be overridden.	Only abstract methods need to be overridden.
All variables declared in an interface are by default **public**, **static**, and **final**.	Variables, if required, have to be declared as **public**, **static**, and **final**.
Interfaces do not have any constructors.	Abstract classes can have constructors

ENCAPSULATION

- Encapsulation is hiding information.
  - An Wrapper class(Integer) is an example of Encapsulation
- How
  - Making the fields in a class private and providing access to the fields via public methods
- Why
  - Flexibility — Internal logic changes won’t affect the caller of the method
  - Reusability — Encapsulated code can be used by different callers
  - Maintainability — Operations on encapsulated unit won’t affect others parts

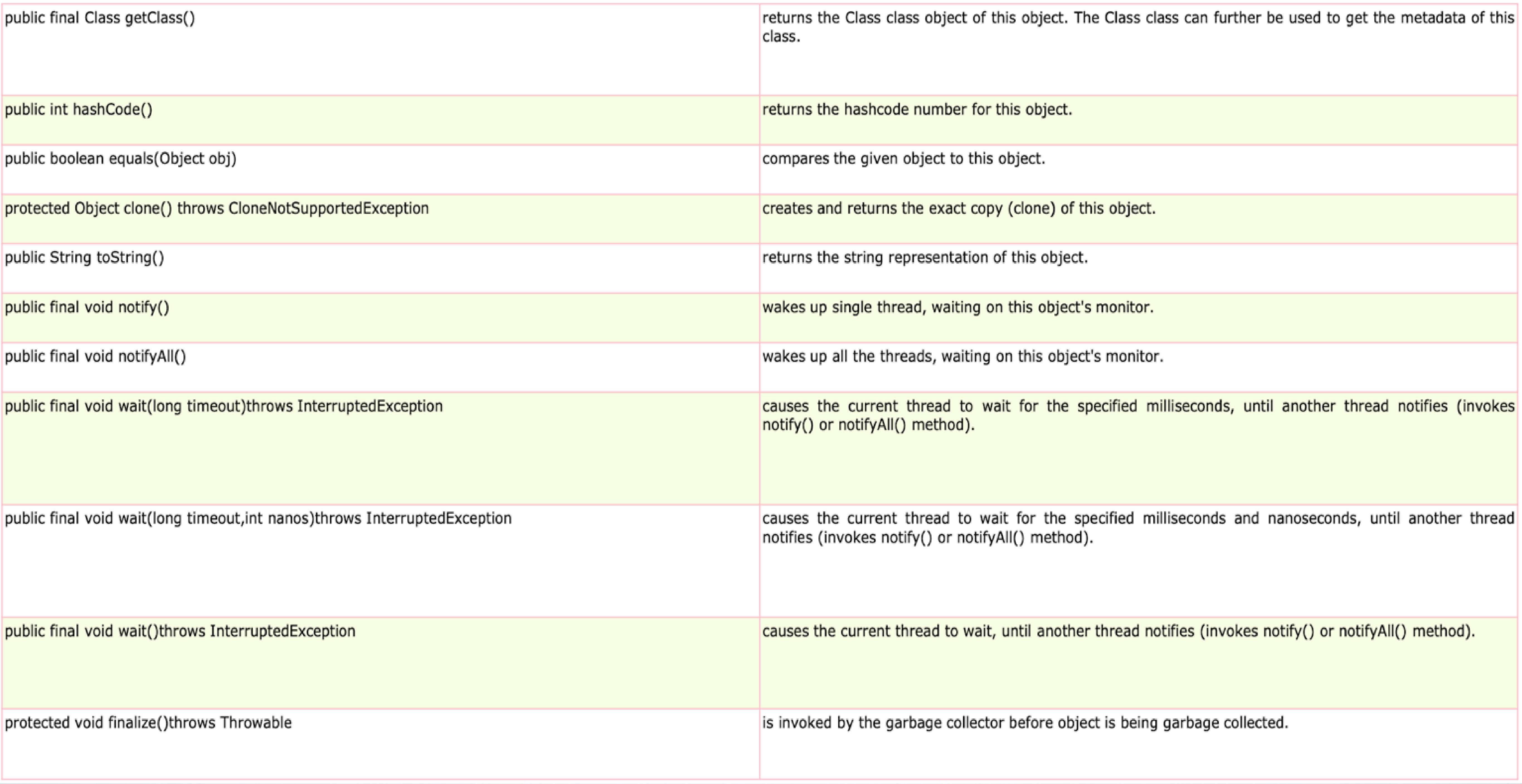
ABSTRACTION VS ENCAPSULATION

- Encapsulation is hiding WHAT THE PHONE USES to achieve whatever it does
- Abstraction is hiding HOW IT DOES it
- Encapsulation = Data Hiding + Abstraction.
    - Imagine you are building a house. The house is made by bricks. Abstraction is the bricks and encapsulation is the house

OBJECT CLASS IN JAVA

- The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java
  - The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, know as ***upcasting***
  - Let's take an example, there is getObject() method that returns an object but it can be of any type like Employee,Student etc, we can use Object class reference to refer that object. For example:
    - Object obj=getObject();
    - We don't know what object will be returned from this method

METHODS OF OBJECT CLASS



public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this object.
protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public String toString()	returns the string representation of this object.
public final void notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void wait(long timeout)throws InterruptedException	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait(long timeout,int nanos)throws InterruptedException	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait()throws InterruptedException	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
protected void finalize()throws Throwable	is invoked by the garbage collector before object is being garbage collected.

ANY QUESTIONS?