Java OOP

SDET Program

Outline

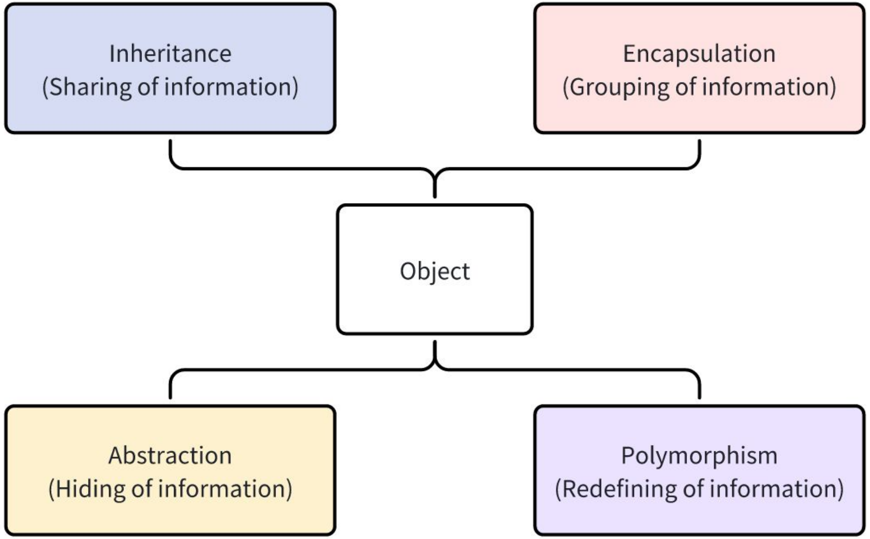
- Object in Java
- Object-Oriented Programming (OOP)
    - Inheritance
        - `java.lang.Object`
        - Different kinds of inheritance
    - Polymorphism
        - Overriding, Overloading
    - Abstraction
        - Abstract class
        - Interface
    - Encapsulation
        - Abstraction vs. Encapsulation

Object (recap)

- An object is a combination of **data** and **operations** working on the available data
  - Example: a car
- In Java:
  - Data of an object is stored in **fields**
  - **Methods** display the object's behavior
- Class is the blueprint, instance is the specific occurrence

Object-Oriented Programming (OOP)

- What is OOP
  - A programming paradigm
  - Treating everything as an object
- 4 Pillars
  - Inheritance
  - Polymorphism
  - Abstraction
  - Encapsulation



Inheritance
(Sharing of information)

Encapsulation
(Grouping of information)

Object

Abstraction
(Hiding of information)

Polymorphism
(Redefining of information)

To Memorize The Four Pillars

- A PIE
    - Abstraction
    - Polymorphism
    - Inheritance
    - Encapsulation

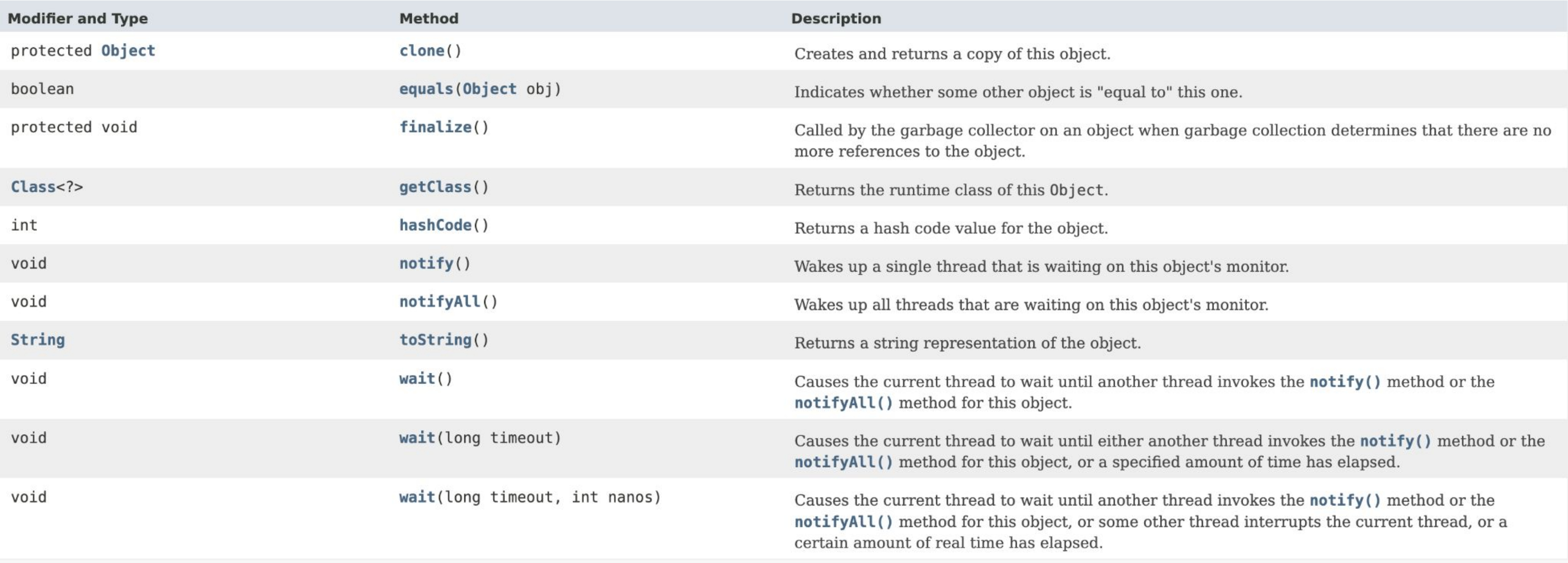
Inheritance

- Is the ability to derive something specific from something generic
    - Animal -> Dog
- Enhances reusability
- A class can inherit the features of another class and add its own modifications
    - Dog: **Subclass** or **Child class**
    - Animal: **Superclass** or **Parent class**
- A subclass inherits **ALL** the properties and behaviors of the superclass
    - Eat, Drink, etc.

Object Class in Java

- The Object class is the parent class of all the classes in Java by default.
- It sits at the top of the class hierarchy
  - `java.lang.Object`
  - Contains some basic methods describing the common behavior of all Java objects:
    - `equals()`
    - `hashCode()`
    - `toString()`

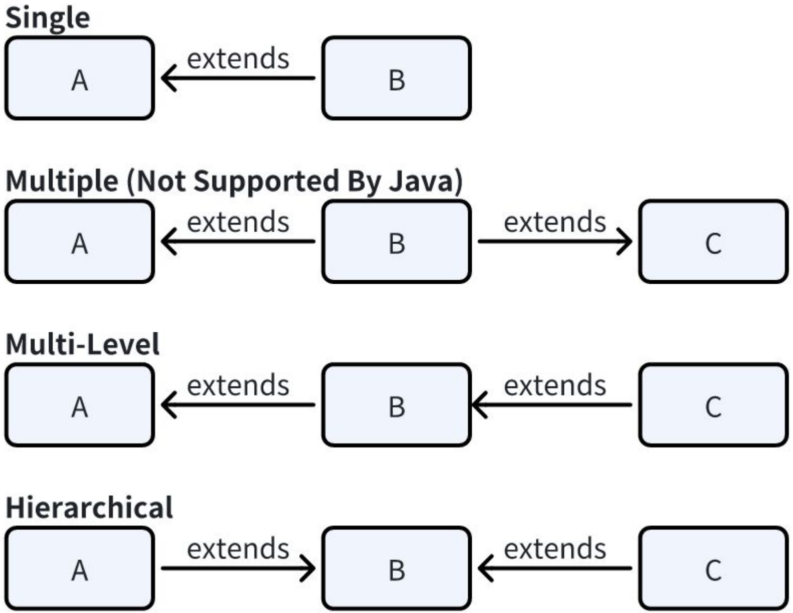
Methods of Object Class



Modifier and Type	Method	Description
protected Object	clone()	Creates and returns a copy of this object.
boolean	equals(Object obj)	Indicates whether some other object is "equal to" this one.
protected void	finalize()	Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?>	getClass()	Returns the runtime class of this Object.
int	hashCode()	Returns a hash code value for the object.
void	notify()	Wakes up a single thread that is waiting on this object's monitor.
void	notifyAll()	Wakes up all threads that are waiting on this object's monitor.
String	toString()	Returns a string representation of the object.
void	wait()	Causes the current thread to wait until another thread invokes the **notify()** method or the **notifyAll()** method for this object.
void	wait(long timeout)	Causes the current thread to wait until either another thread invokes the **notify()** method or the **notifyAll()** method for this object, or a specified amount of time has elapsed.
void	wait(long timeout, int nanos)	Causes the current thread to wait until another thread invokes the **notify()** method or the **notifyAll()** method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

Types of Inheritance

- Single
  - Inherit from only one class
- Multiple*
  - Not supported by Java
  - Inherit from more than one unrelated class
- Multi-Level
  - No limit to the chain of inheritance
  - Makes code excessively complex if too deep
- Hierarchical
  - More than one class can inherit from a single class
- Hybrid
  - Any combination of the above



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 superclass
  - A dog **is** an Animal
- How can we refer to current object and parent/super object in Java?

This

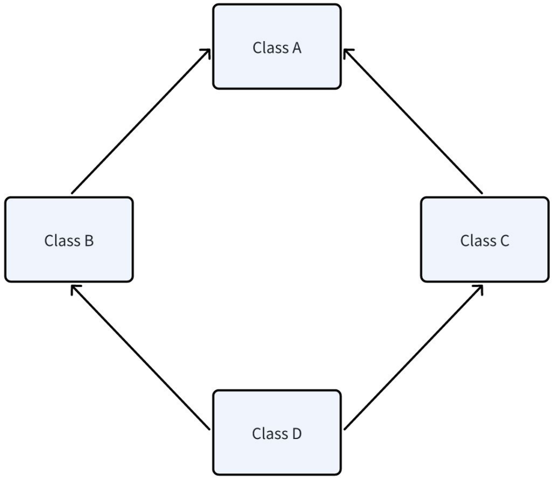
- `this` is a keyword that refers to the current object
- `this` Usage
    - Refer current class instance variable
    - Invoke current class method (implicitly)
    - Invoke current class constructor
    - Can be passed as argument in the method call
    - Return the current class instance from the method

Super

- A reference variable which is used to refer immediate parent class object
  - Whenever you create the instance of a subclass, an instance of the superclass is created implicitly
  - Referred by super reference variable
- super Usage
  - Refer immediate parent class instance variable
  - Invoke immediate parent class method
  - Invoke immediate parent class constructor

Why Not Multiple Inheritance?

- The 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
  - Methods declared in interface are by default abstract
    - Only method signature, no body
- It’s the blueprint of a class
  - Specifies what a class must do but not how to do it
- Specifies a set of methods that the class has to implement
- If class implements interface but doesn’t 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

Aggregation / Composition

- If a class have an entity reference, it is known as **Aggregation** or **Composition**
- Aggregation represents **HAS-A** relationship
    - Employee object contains many information such as:
        - `String id`
        - `String name`
        - `String email`
        - **`Address address`**
            - `String city`
            - `String zipcode`
    - In such case, Employee has an entity reference address, so relationship is Employee **HAS-A** address

Why Aggregation

- Code reusability is best achieved by aggregation when there is no IS-A relationship
    - If we don’t have the Address object
        - Have to add all related information such as city and zipcode in employee
        - If we introduce another object called Company later which has the address too, we will have to add the same attributes to Company object again

Aggregation

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

Inheritance vs. Aggregation

- Inheritance should be used only if the IS-A relationship is maintained throughout the lifetime of the objects involved; Otherwise, use aggregation
    - Flexibility of aggregation
        - For example, 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) - will introduce in the future

Polymorphism

- To perform a **single action in different ways** in Java
  - The word “poly” means many and “morphs” means forms
    - Hence, polymorphism means “many forms”
- There are two types of polymorphism in Java:
  - Compile time polymorphism (Overloading)
  - Runtime polymorphism (Overriding)

Compile Time vs. Runtime

- Compile time: the instance where the code you entered is converted to executable
    - For example: from file.java to file.class
- Runtime: the instance where the executable is running
    - For example: when the for loop gets executed

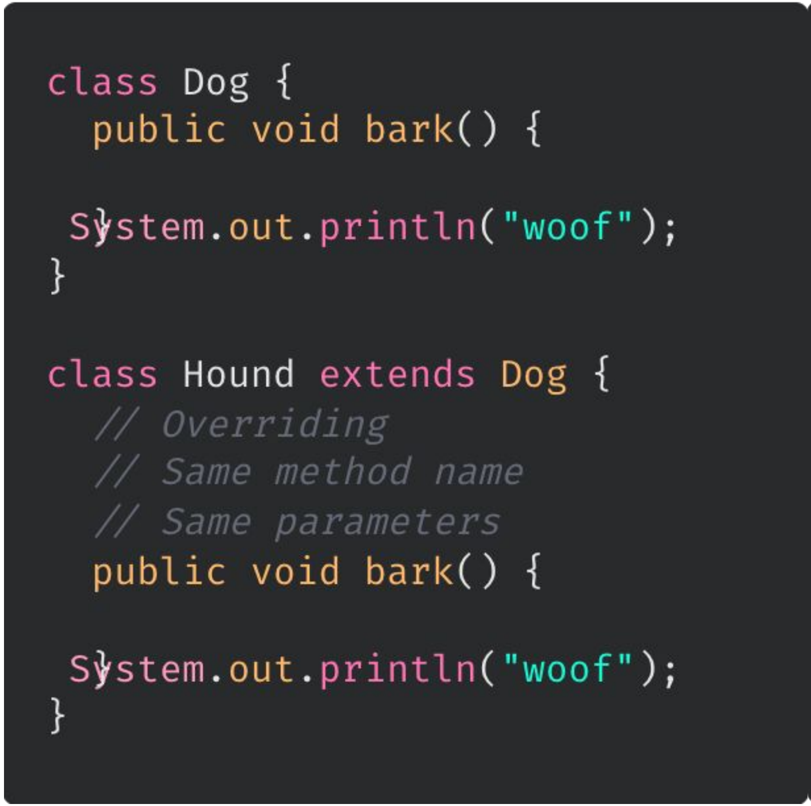
Compile Time Polymorphism

- 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 overridden

Polymorphism

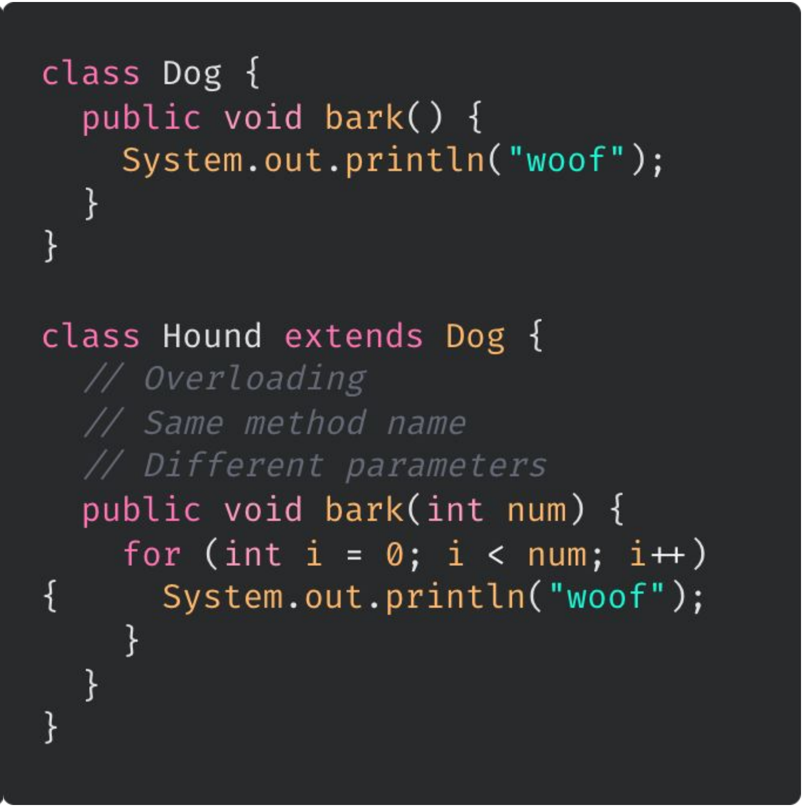


```
class Dog {
 public void bark() {

 S}ystem.out.println("woof");
}

class Hound extends Dog {
 // Overriding
 // Same method name
 // Same parameters
 public void bark() {

 S}ystem.out.println("woof");
}
```



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

class Hound extends Dog {
 // Overloading
 // Same method name
 // Different parameters
 public void bark(int num) {
  for (int i = 0; i < num; i++)
{  System.out.println("woof");
  }
 }
}
```

Overloading vs. Overriding

Method Overloading	Method Overriding
Increases the readability of the code	Provides specific implementation of the method already provided by its superclass
Performed within a class	Occurs in two classes that have an IS-A relationship (inheritance)
Parameters must be different	Parameters must be the same
Compile time polymorphism	Runtime polymorphism
Parameters must be changed, return type can be either the same or different	Return type must be the same or covariant

Abstraction

- What:
  - Hiding internal details and showing functionality
    - For example, making a phone call, we don’t know the internal processing
- How:
  - By creating either **Abstract Classes** or **Interfaces**
- Why:
  - Hide unnecessary things from user providing easiness
  - Hiding the internal implementation of software 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 zero 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 doesn’t override the abstract methods of the abstract class
    - 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 accidental 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 and interface	`extends` keyword is used to inherit a class
By default, all methods in an interface are public and abstract, no need to tag them as public and abstract	Methods can be public, protected, or package-private and can be abstract or concrete
Interfaces have no implementation at all (but from Java 8, interfaces can have default and static methods)	Abstract classes can have partial or full implementation of some methods
All methods of an interface need to be overridden unless they are default or static	Only abstract methods need to be overridden
All variables in an interface are implicitly public, static, and final	Variables can have any access modifier and do not have to be static or final
Interfaces do not have constructors	Abstract classes can have constructors

Encapsulation

- What:
  - Encapsulation is hiding information
    - We have access modifiers to do that
- How:
  - Making the fields in a class private and providing access to the fields via public getter 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 other 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
        - Encapsulation is the house

Thank You!