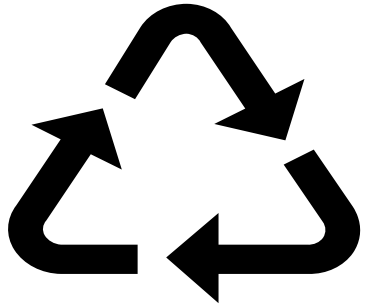
OBJECT ORIENTED PROGRAMMING (OOP)

Inheritance and Polymorphism

OOP Principles

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Concepts and terminology



Reuse

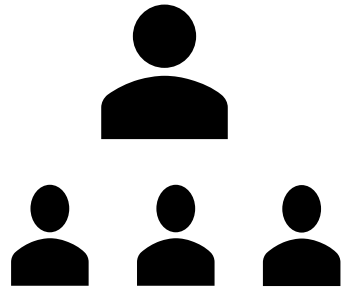
Inheritance enables efficiency as it allows the **reuse** of existing code, whilst allowing additional code to be added

Parent

The Parent class is the existing class from which the new class will be derived through inheritance (**Super** or **Base** class)

Child

The Child class is new class, derived from the Parent class and child class header uses the **extends** keyword (**Derived** or **Sub** class)

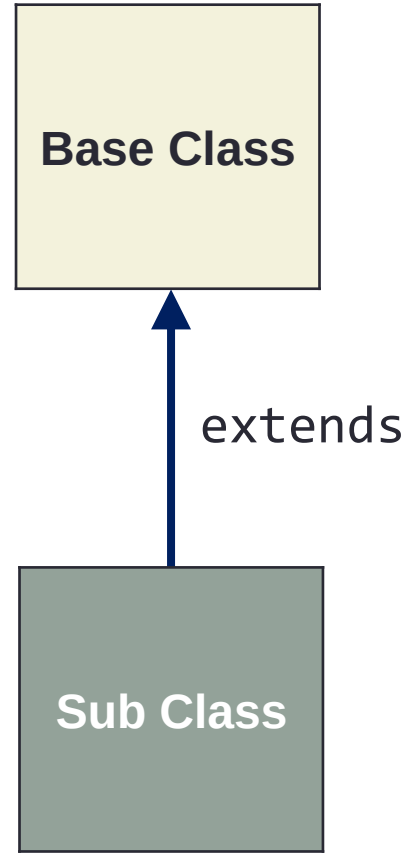


Inherits

The Child class inherits all the fields and methods from the Parent class (**inheritance**)

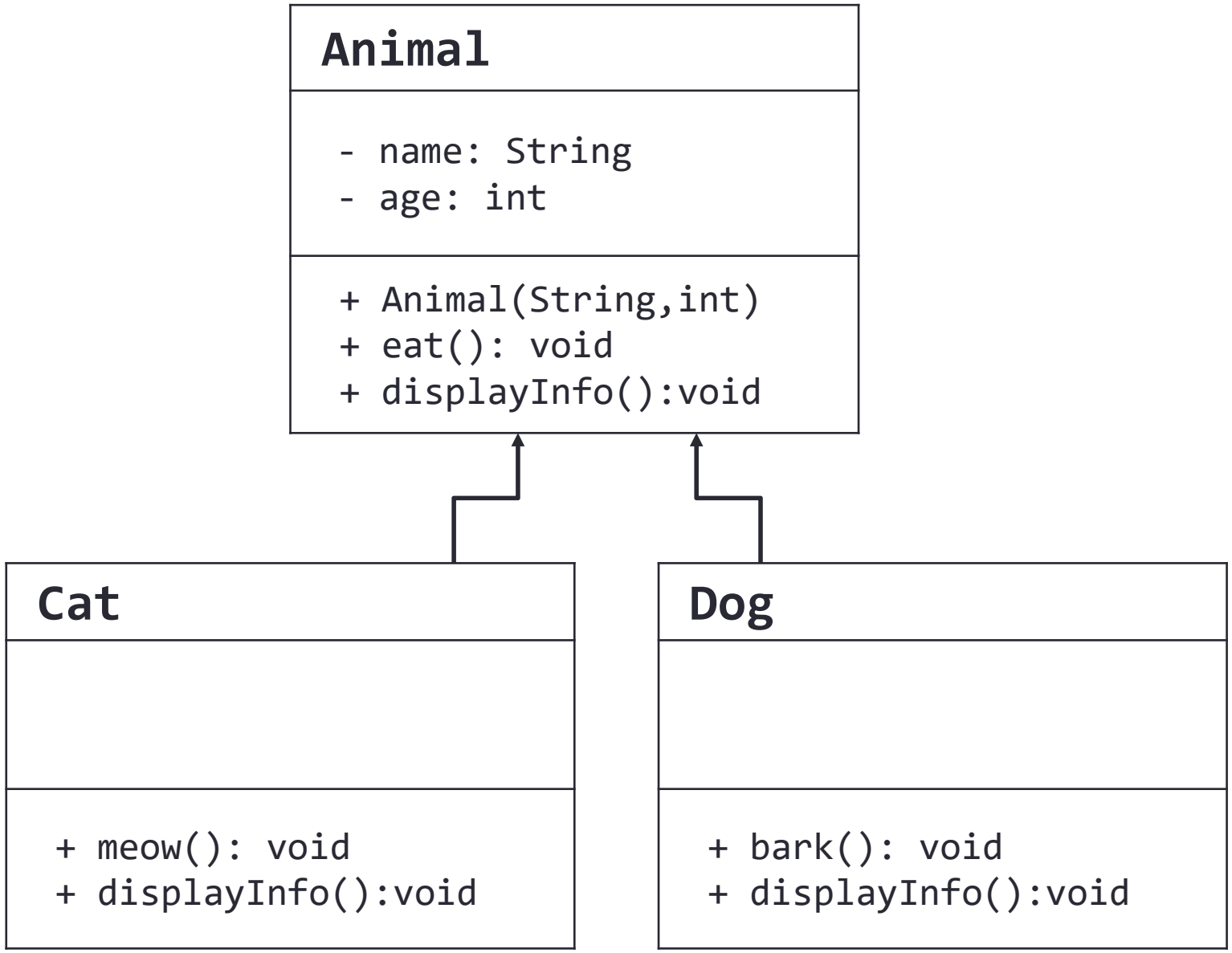
Terminology

- **Base class** will be a term used for parent class
- **Sub class** will be a term used for child class
- Inheritance is invoked by use of the `extends` keyword
- Relationship arrow points away from Sub class and towards Base class
- '`this`' keyword is reference to the current class
- '`super`' is a reference used by Sub class to refer to Base class



Animal Example

- The base class will be `Animal`
- The class will be kept simple
  - Two fields
  - One constructor
  - One method
- The sub class will be Cat & Dog
- displayInfo() is an inherited method and will be rewritten (i.e., overridden)



Base Class Code: Animal

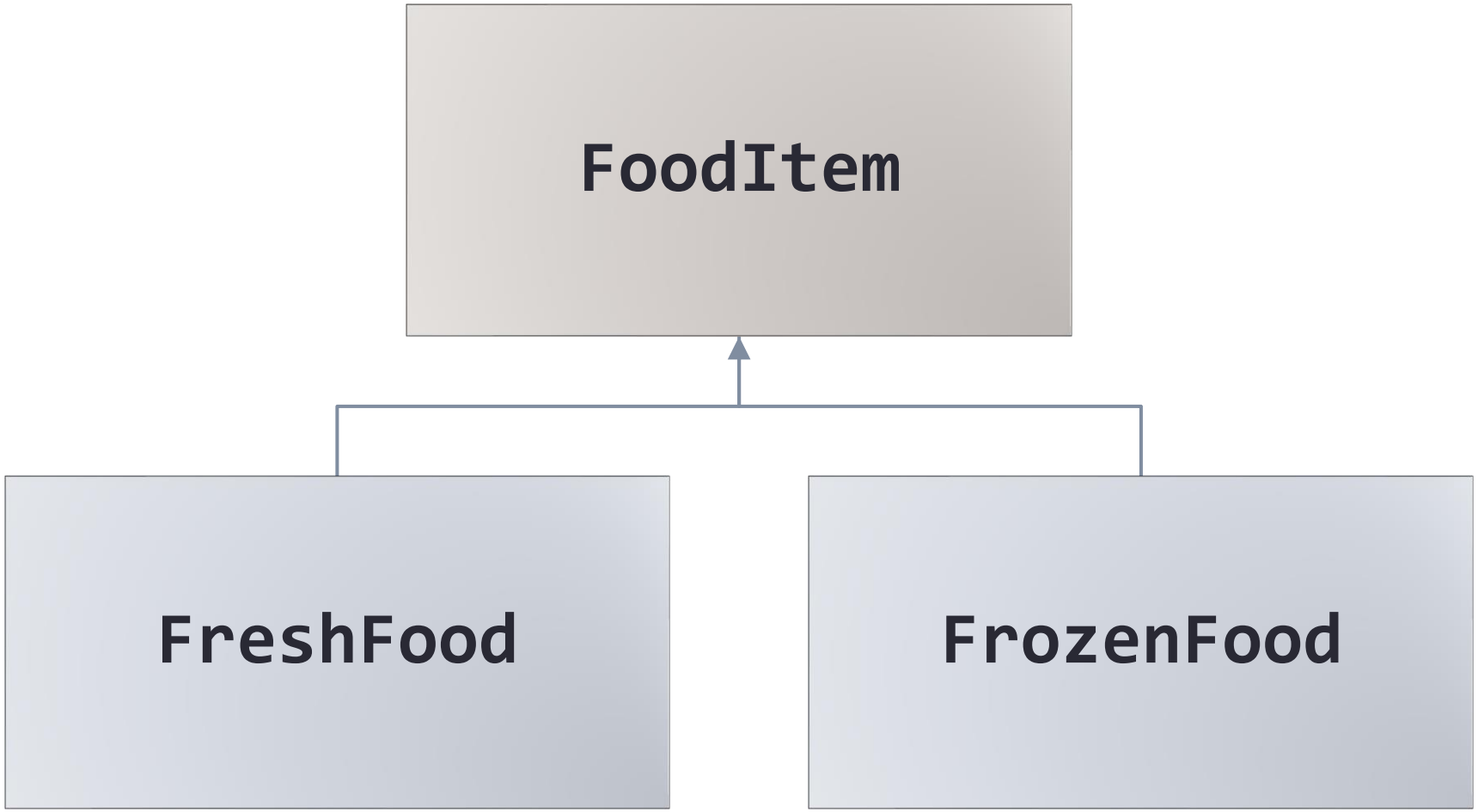
```
public class Animal {
    String name;
    int age;
    public Animal(String name, int age) {
      this.name = name;
      this.age = age;
    }
    public void eat() {
      System.out.println(name + " is eating.");
    }
    public void displayInfo() {
      System.out.println("Name: " + name + ", Age: " + age);
     }
}
```

Derived Class Code: Dog and Cat

```
class Dog extends Animal {
  public Dog(String name, int age) {
    super(name, age);
  }
  // Specific method for Dog
  public void bark() {
    System.out.println(name + " is barking.");
  }
  // Overriding displayInfo method
  @Override
  public void displayInfo() {
    System.out.println("Dog's Name: " + name + ", Age: " + age);
  }
}
```

```
class Cat extends Animal {
  public Cat(String name, int age) {
    super(name, age);
  }
  // Specific method for Cat
  public void meow() {
    System.out.println(name + " is meowing.");
  }
  // Overriding displayInfo method
  @Override
  public void displayInfo() {
    System.out.println("Cat's Name: " + name + ", Age: " + age);
  }
}
```

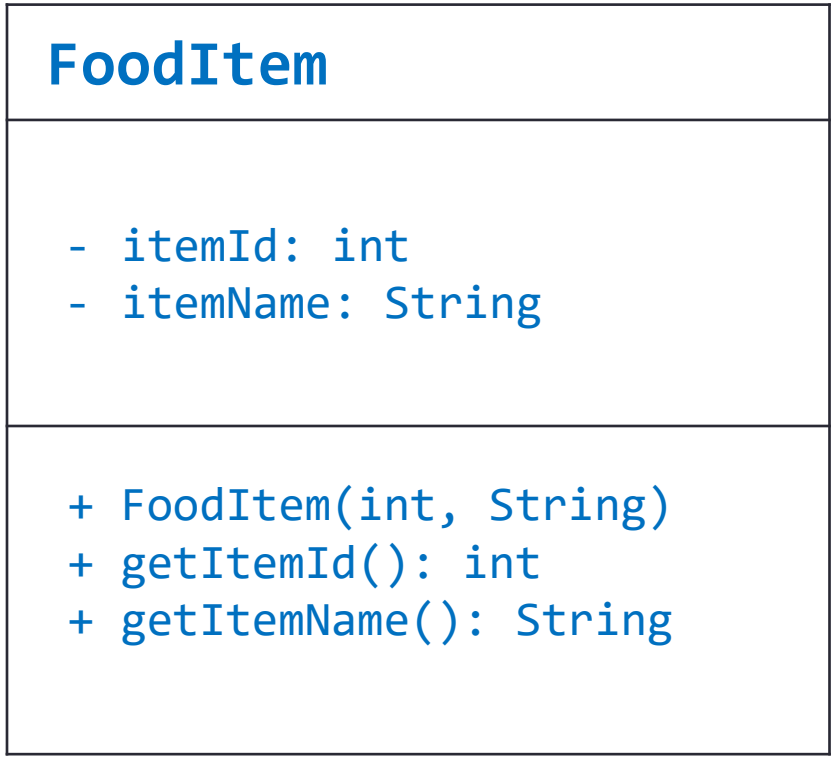
Food Item Example



`FreshFood` and `FrozenFood` are sub classes of `FoodItem`

Base Class Design: Food Item

- The base class will be `FoodItem`
- The class will be kept simple
  - Two fields
  - One constructor
  - Two accessors
- We will not include price or stock
  - Remember the abstraction concept

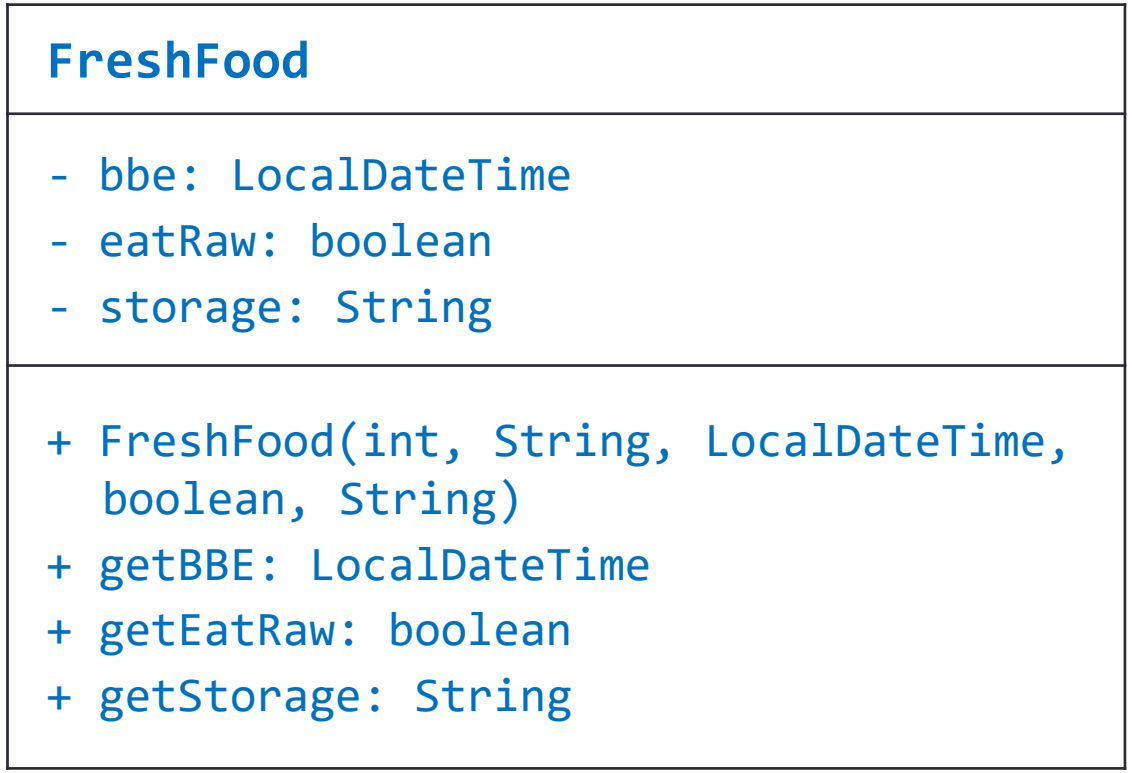


Base Class Code: Food Item

```
import java.time.LocalDateTime;
 class FoodItem {
    private int itemId;
    private String itemName;
    public FoodItem(int itemId, String itemName) {
        this.itemId = itemId;
        this.itemName = itemName;
    }
     public int getItemId() {
        return itemId;
    }
     public String getItemName() {
        return itemName;
    }
}
```

Sub Class Design: Fresh Food

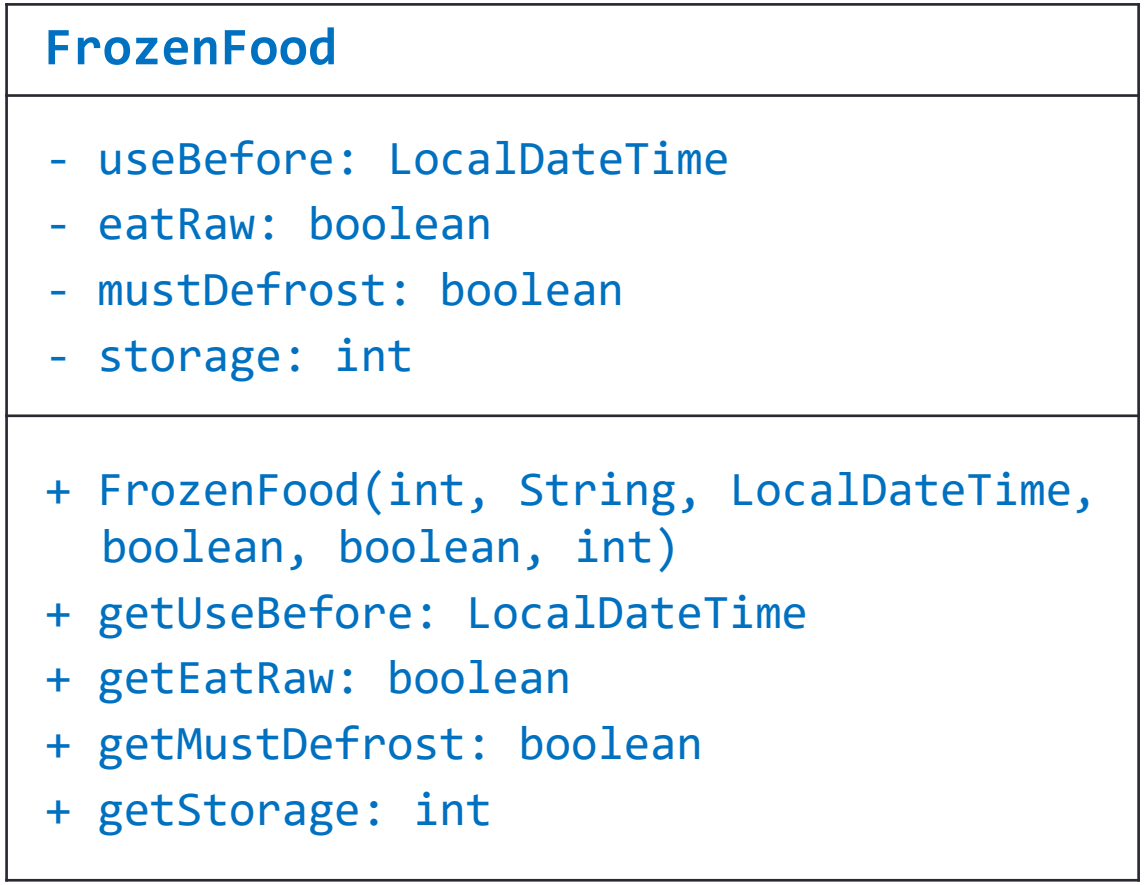
- Top row of class diagram specifies name of Sub Class only
- Only specify new fields in middle row
  - I.e. Do not specify inherited fields
- Constructor method design must specify parameters for inherited and new fields
- New accessors and methods are defined
- If any inherited service method is to be rewritten (i.e., overridden) then it is included



Sub Class Code: Fresh Food

```
// Subclass FreshFood extending FoodItem
class FreshFood extends FoodItem {
    private LocalDateTime bbe; // Best Before End
    private boolean eatRaw;
    private String storage; // How to store it?
    // Constructor for FreshFood
    public FreshFood(int itemId, String itemName,
LocalDateTime bbe, boolean eatRaw, String storage)
{
        super(itemId, itemName);
        this.bbe = bbe;
        this.eatRaw = eatRaw;
        this.storage = storage;
    }
// Getter for bbe
    public LocalDateTime getBBE() {
        return bbe;
    }
     // Getter for eatRaw
    public boolean getEatRaw() {
        return eatRaw;
    }
     // Getter for storage
    public String getStorage() {
        return storage;
    }
}
```

Sub Class Design: Frozen Food



Sub Class Code: Frozen Food

```
// Subclass FrozenFood extending FoodItem
class FrozenFood extends FoodItem {
    private LocalDateTime useBefore; // Use Before date
    private boolean eatRaw;
    private boolean mustDefrost;
    private int storage; // Storage temp
    // Constructor for FrozenFood
    public FrozenFood(int itemId, String itemName,
LocalDateTime useBefore, boolean eatRaw, boolean
mustDefrost, int storage) {
        super(itemId, itemName);
        this.useBefore = useBefore;
        this.eatRaw = eatRaw;
        this.mustDefrost = mustDefrost;
        this.storage = storage;
    }
    // Getter for useBefore
public LocalDateTime getUseBefore() {
        return useBefore;
    }
     // Getter for eatRaw
    public boolean getEatRaw() {
        return eatRaw;
    }
     // Getter for mustDefrost
    public boolean getMustDefrost() {
        return mustDefrost;
    }
     // Getter for storage (could represent
temperature in degrees)
    public int getStorage() {
        return storage;
    }
}
```

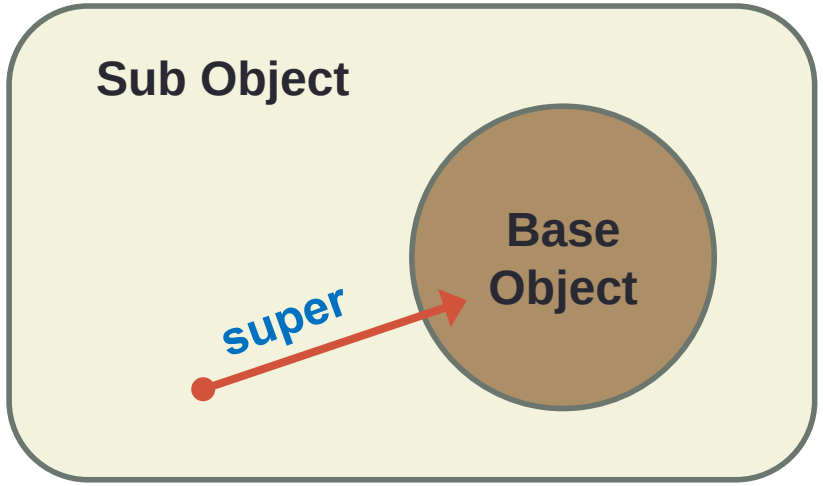
Main Class Code

```
// Main class to test the implementation
public class Main {
    public static void main(String[] args) {
        // Example of creating a FreshFood object
        FreshFood apple = new FreshFood(1, "Apple", LocalDateTime.now().plusDays(10), true, "Refrigerate");
        System.out.println("FreshFood: " + apple.getItemName() + ", BBE: " + apple.getBBE());

        // Example of creating a FrozenFood object
        FrozenFood pizza = new FrozenFood(2, "Pizza", LocalDateTime.now().plusMonths(6), false, true, -18);
        System.out.println("FrozenFood: " + pizza.getItemName() + ", Use Before: " + pizza.getUseBefore());
    }
}
```

Super?

- When a sub class object is created
  - Java will create two objects
- Firstly, the base class object
- Secondly, the sub class object
  - Wrapped around the base class object
- The `super` keyword:
  - Can be used by code in the sub class object wrapper
  - To access the inner base class object
  - Usually, the base class constructor and accessors



Base Class field visibility

Fields of the base class are kept `private`

- The code in sub classes could not directly interact with inherited fields
- Instead, will have to make use of inherited accessor methods

If sub classes need to interact with inherited fields

- Then set the visibility of the fields in the Base class to `protected`.

Which to use

- If the problem requires only **read** access to inherited fields, then keep them **private**
- If **write** access is required, then make fields **protected** in base class

Base class fields are private, meaning no direct access by sub class code

```
public class FoodItem {

   private int itemId;
   private String itemName;
```

Base class fields are protected, meaning sub class code can directly interact with them.

```
public class FoodItem {

   protected int itemId;
   protected String itemName;
```

Read & Write Access

```
// Base class (Vehicle)
class Vehicle {
  private String model; // Private field
(read-only access via getter)
  protected int speed; // Protected
field (read and write access for
subclasses)
    // Constructor to initialize fields
    …
    // Getter for model (read-only
access)
    public String getModel() {
        return model;
    }
   // Getter for speed (read access)
    public int getSpeed() {
        return speed;
    }
    // Protected method to set speed
(write access for subclasses)
    protected void setSpeed(int speed) {
        this.speed = speed;
    }
}
```

```
// Derived class (Car)
class Car extends Vehicle {
  public Car(String model, int speed) {
    super(model, speed);
  }
  // Method to accelerate the car (write
access to speed)
  public void accelerate(int increment) {
    setSpeed(getSpeed() + increment); //
Access protected method from base class
  }
}

// Test Class
public static void main(String[] args) {
    Car myCar = new Car("Toyota Camry", 50);
    System.out.println("Initial speed of " +
myCar.getModel() + ": " + myCar.getSpeed() + "
km/h");
    myCar.accelerate(20);
    System.out.println("Speed after acceleration:
" + myCar.getSpeed() + " km/h");
}
```

Multiple Inheritance

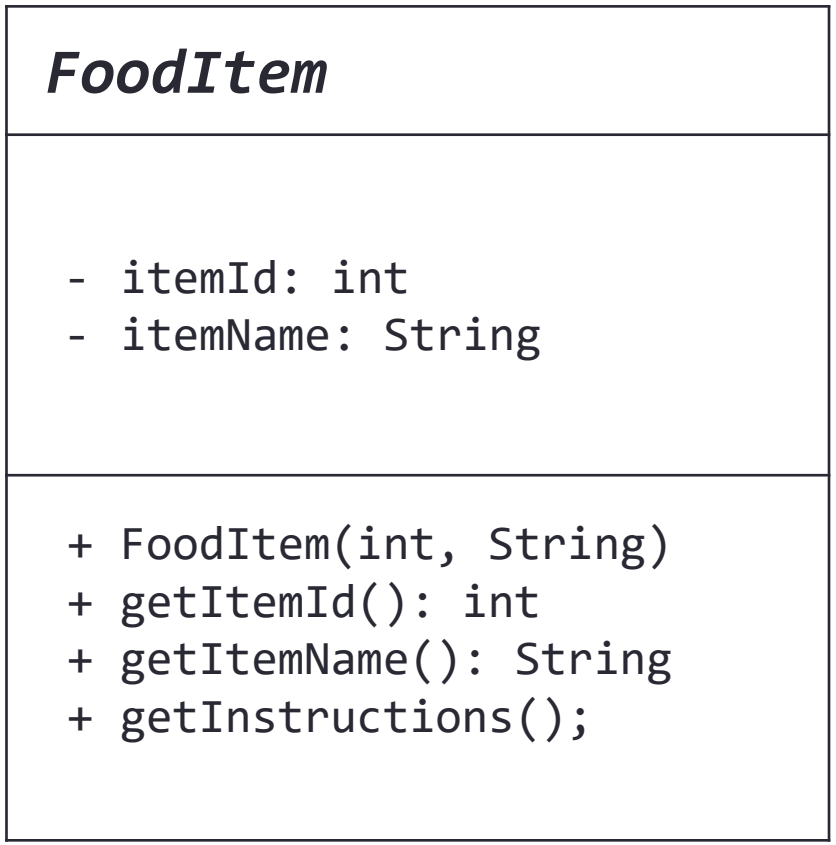
- In programming multiple inheritance allows a sub class to be derived from two or more classes, inheriting the members from all parents
  - However, such an approach invokes the Diamond Problem
  - I.e., Collisions of fields and methods which have the same name in each parent
- Java only supports single inheritance from a class
  - Meaning that a sub class can have only one (concrete or abstract) base class
- Java does offer a form of multiple inheritance
  - By allowing a class to implement multiple interfaces
  - See: https://ioflood.com/blog/java-multiple-inheritance/

Abstract Base Class

- A Sub class is a specialised version of the base class
  - Additional fields provides more state information
  - Additional Methods provide more behaviours
- Inheritance also allows for classification to be modelled
  - Each sub class is related but represents a different category of food
- The question arises, *"Do we need an instance object of the base class?"*
  - If not, we make specify the base class to be an abstract class
  - This prevents any object instance of the base class being created
  - Ensuring that (concrete) instances are of the sub classes

Abstract Base Class Example

- In class diagram any Class, field or method which is abstract is shown in italics



- For this example, only the base class will be abstract.
- Fields and methods will remain unchanged
- Abstract keyword need to be added to the signature

  abstract class FoodItem { }
- A new **abstract** method need to be added, for example getInstructions() is added in signature form

  public abstract String getInstructions();

Abstract Base Class Example

Abstract Class FoodItem:

- We define FoodItem as abstract because it's a generic concept for different types of food items, but it's not useful to instantiate a FoodItem object directly
- The FoodItem class contains both concrete methods (like getItemId() and getItemName()) and an abstract method getInstructions(), which must be implemented by subclasses

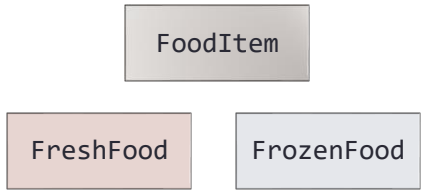
Abstract Method getInstructions():

- This method is defined as abstract in FoodItem because the way expiration information is handled is different for different types of food

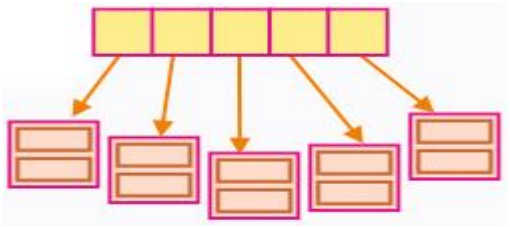
Polymorphism

- Working with related Classes
- Simple Polymorphism
- Learning
- True Polymorphism

Working with Related Classes



Related Classes



Collection of objects

Polymorphism

Polymorphism

- Polymorphism means "*many forms*" and is an automatic side benefit of inheritance
  - Which allows the use of a base class to mimic sub classes
  - But the mimicry is restricted to the inherited methods
- Let's look back to our Animal example
- Base class reference can refer to an object of subclass Dog:

```
Animal myDog = new Dog();
myDog.makeSound(); // Outputs: The dog barks
```

- Base class reference can refer to an object of subclass Cat:

```
Animal myCat = new Cat();
myCat.makeSound(); // Outputs: The cat meows
```

- An Animal object itself:

```
Animal genericAnimal = new Animal();
genericAnimal.makeSound(); // Outputs: The animal makes a sound
```

Task 1: Practice the session examples

- Test the Animal example by adding a Main class to the code, and create instances of each subclass and call their methods with printing statements
- Test the Food Item example and add two other subclasses:
  - 1) Canned Food: Food that comes in cans and typically has a longer shelf life
  - 2) Dry Food: This includes items like rice, pasta, and grains that can be stored for long periods without refrigeration
- Test the new subclasses by creating instances and printing results

Task 2: Inheritance Basics

- Practice creating a base class and derived classes using inheritance
- Create a base class Vehicle
- Create subclasses Car, Bike, and Truck that inherit from Vehicle
- Each subclass should override at least one method
- In the main() method, create instances of each subclass and call their methods with printing statements

Task 3: Exploring Polymorphism

- Add a method makeSound() to the base class Animal

  makeSound() should print a generic message like "Animal makes a sound.“
- Add a subclass Bird in addition to the existing two subclasses Dog and Cat that inherit from Animal
- Override makeSound() in each subclass to print specific sounds like "Dog barks", "Cat meows", etc
- In the main() method, do the following:
  - Create an array of Animal objects
  - Store a Dog, Cat, and Bird object in the array
  - Iterate through the array and call the makeSound() method for each object
  - Observe polymorphism in action.