Inheritance

Extending a class

Recall...Object-Oriented Programming Pillars

Encapsulation

- Hide complexity of an object from outside. This is possible because the methods that can manipulate the object's data are bound together, INSIDE the class => the manipulation is hidden from outside of the class and safe from outside interference

Abstraction

- Reduce Complexity. Example: when representing a "person" object, no need to represent EVERYTHING about a person. Just the info needed for our program to work correctly
- Information hiding (no details about implementation; communicate only pre/post conditions). Isolate the impact of change

Inheritance

Polymorphism

Dealing with Redundancy… in OOP

- Consider the following situations:
  - You need define classes to represent some geometric shapes (model circles, rectangles, and triangles).
  - You need to model different types of “persons” that are related to a University (student, faculty, administrators, donors)
  - You need to represent different types of pets (cat, dog, bird)

- In each of these cases, the objects we are trying to model have many common features. What is the best way to design these classes so to avoid redundancy?

Recall...Object-Oriented Programming Pillars

Encapsulation

- Hide complexity of an object from outside. This is possible because the methods that can manipulate the object's data are bound together, INSIDE the class => the manipulation is hidden from outside of the class and safe from outside interference

Abstraction

- Reduce Complexity. Example: when representing a "person" object, no need to represent EVERYTHING about a person. Just the info needed for our program to work correctly
- Information hiding (no details about implementation; communicate only pre/post conditions). Isolate the impact of change

Inheritance

- Allows a class to derive its properties and methods from another class
- Eliminate code redundancy (child class can inherit parent class methods, behaviors, states) example: employee, students are also a person

Polymorphism

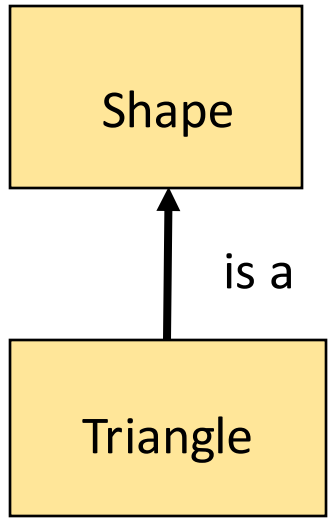
- Code can take on multiple forms. Each class can implement an inherited method in its own way

Introduction to Inheritance

- Inheritance is a form of software reuse in which a new class is created by absorbing an existing class's members.

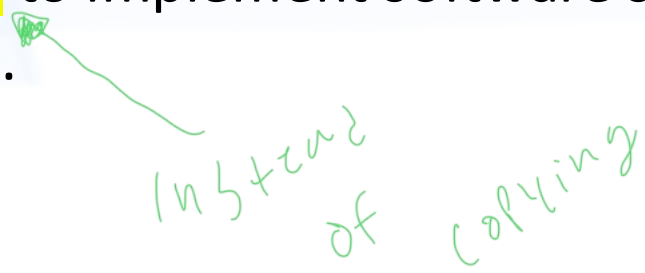
- The new class **extend**s the original class by adding new or modified capabilities to the original class.

- Inheritance is called **is-a** relationship between classes



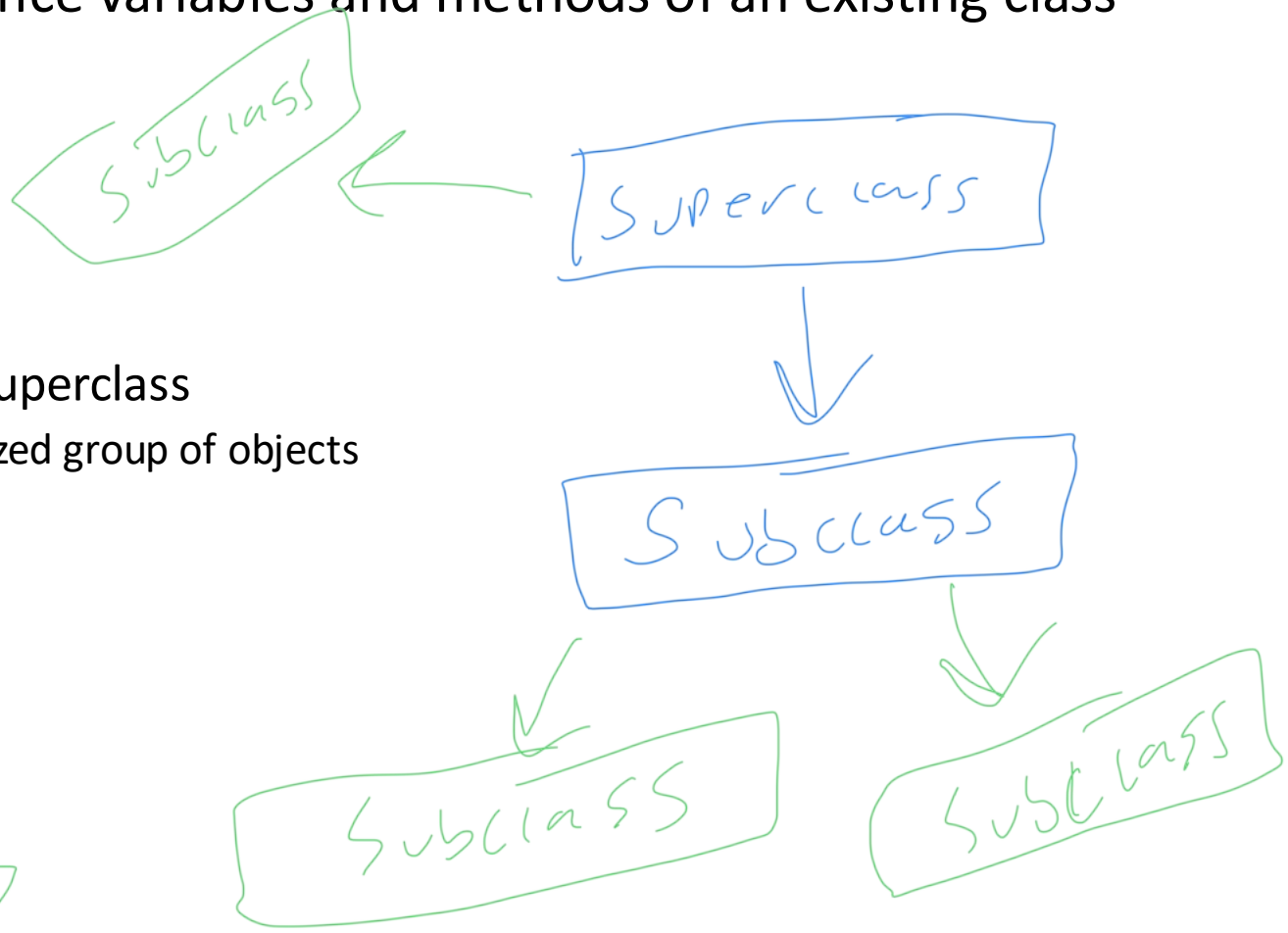
Why Inheritance?

- Assume you have a complicated class that basically does what you want but you need a small modification
- If you edit the source code, even to make a small change, you risk breaking something.
  - You must study the original code to be sure that your changes are correct. This may not be easy
- Inheritance allows you to save time during program development by basing new classes on existing proven and debugged high-quality classes.
- Code is re-used to implement software system with the minimum amount of code repetition.



Superclass and Subclass

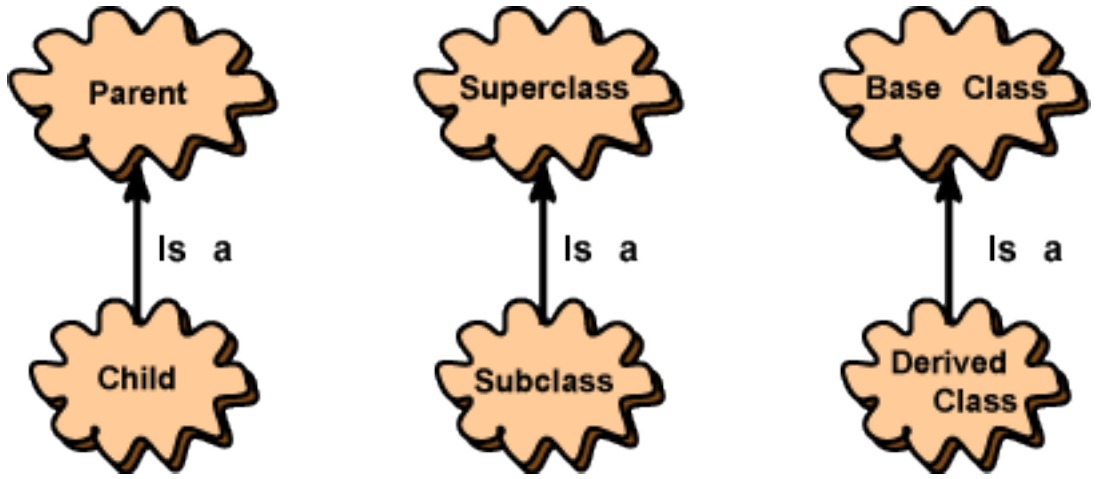
- A new class (subclass) inherits instance variables and methods of an existing class (superclass)



- The concept of specialization:
  - A superclass is more general
  - A subclass is more specific than its superclass
    - A subclass represents a more specialized group of objects

- A subclass can:
  - be a superclass of future subclasses
  - add its own fields and methods

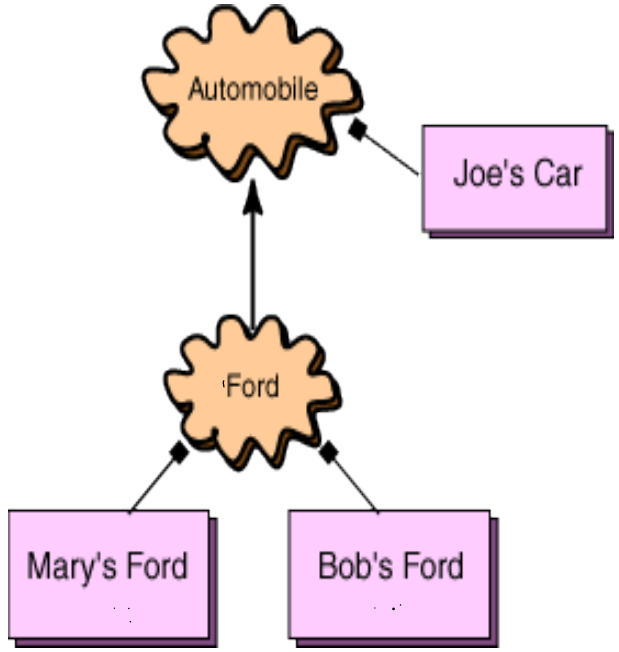
Interchangeable Phrases



- There are three sets of phrases for describing inheritance relationships:
  - parent / child
  - base class / derived class
  - superclass / subclass

Important Note

- **Inheritance is between classes, not between objects.**
- A parent class is a blueprint that is used when an object is constructed.
- A child class is another blueprint (that looks much like the parent), but with added features.



Class relationships: `is-a` vs. `has-a`

- ***is-a*** represents inheritance
  - In an *is-a* relationship, an object of a subclass can also be treated as an object of its superclass

→ Very common

- ***has-a*** represents composition
  - In a *has-a* relationship, an object contains as members references to other objects

– An array of objects?

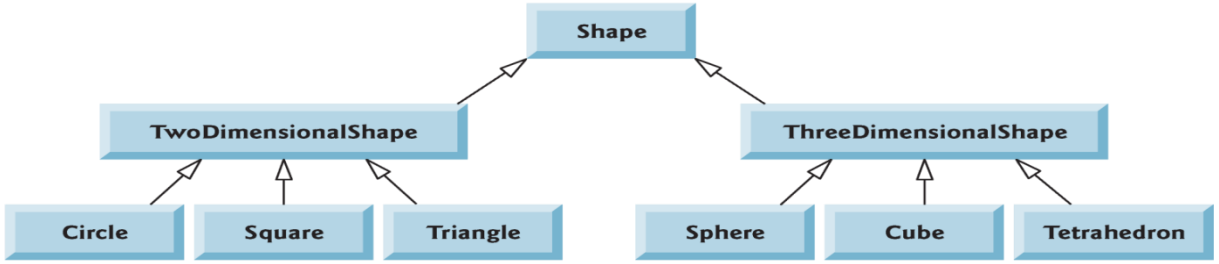
Examples on Superclass and Subclass

Superclass	Subclasses
Student	GraduateStudent, UndergraduateStudent
Shape	Circle, Triangle, Rectangle, Sphere, Cube
Loan	CarLoan, HomeImprovementLoan, MortgageLoan
Employee	Faculty, Staff
BankAccount	CheckingAccount, SavingsAccount

Class Hierarchy

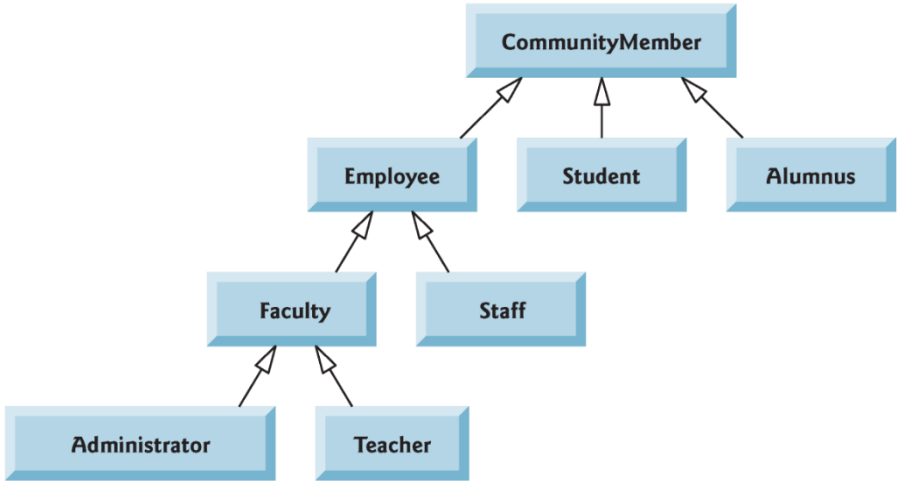
- The direct superclass is the superclass from which the subclass explicitly inherits (parent)
- An indirect superclass is any class above the direct superclass in the class hierarchy (grandparent or great-grandparent or …)
- The Java class hierarchy begins with class `Object` (in package `java.lang`)
  - Every class in Java directly or indirectly extends (or "inherits from") `Object`
- Java supports only single inheritance, in which each class is derived from exactly one direct superclass (parent)

Class Inheritance Hierarchy: Example 1

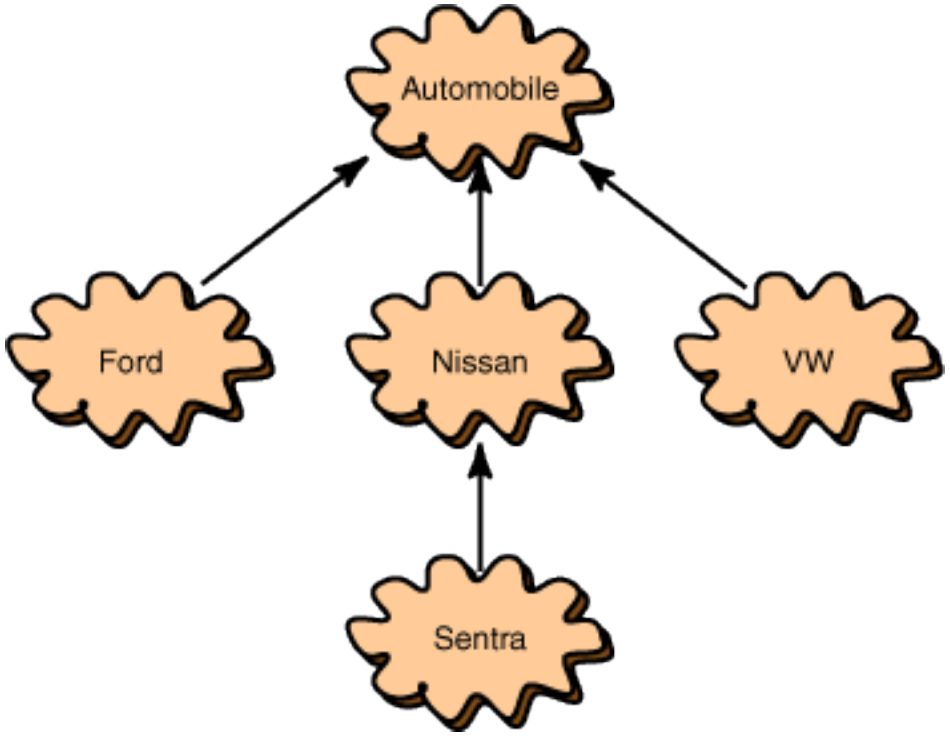


- A Triangle *is-a* TwoDimensionalShape and *is-a* Shape
- A Sphere *is-a* ThreeDimensionalShape and *is-a* Shape

Class Inheritance Hierarchy: Example 2



Class Inheritance Hierarchy: Example 3



`extends` Keyword

- The syntax for deriving a child class from a parent class is:

```
class childClass extends parentClass {
        // new members and constructors of the child class are addded here
}
```

- Members (instance variables and methods) of the parent class are included in the child by inheritance.

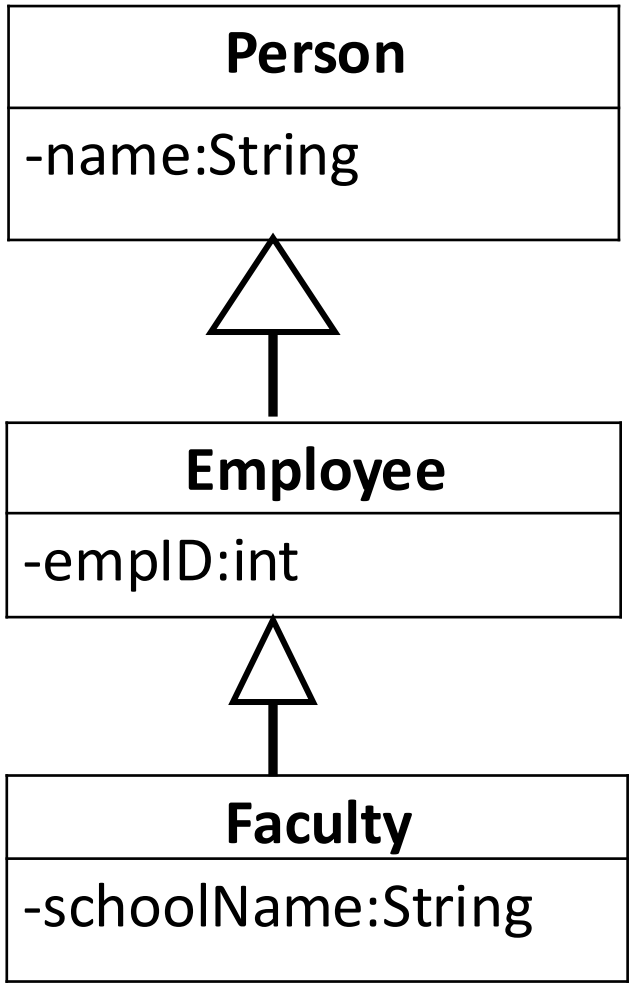
- Additional members are added to the child in its class definition.

Example: PersonApplication

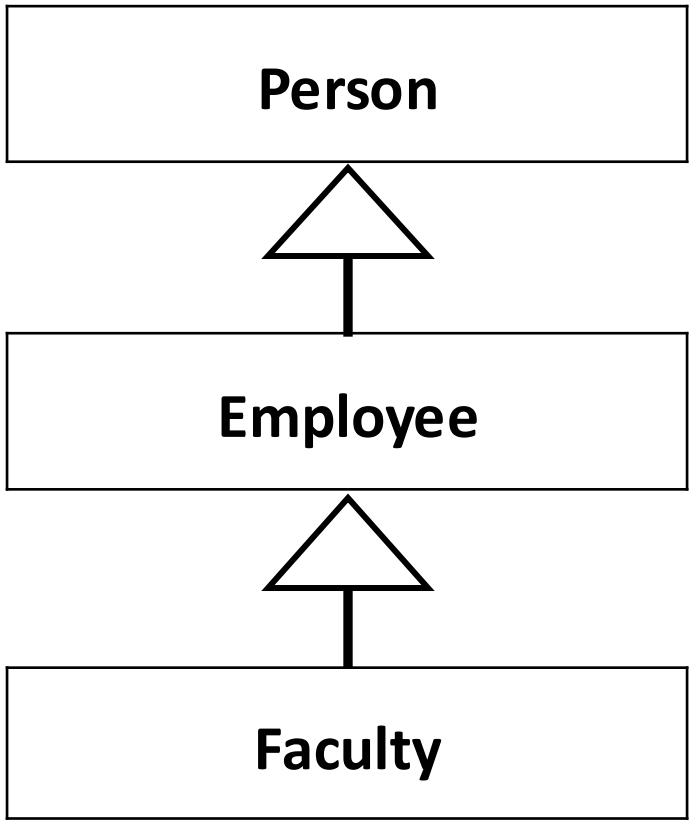
```
public class Person {
    private String name;
}
```

```
public class Employee extends Person {
  private int empID;
}
```

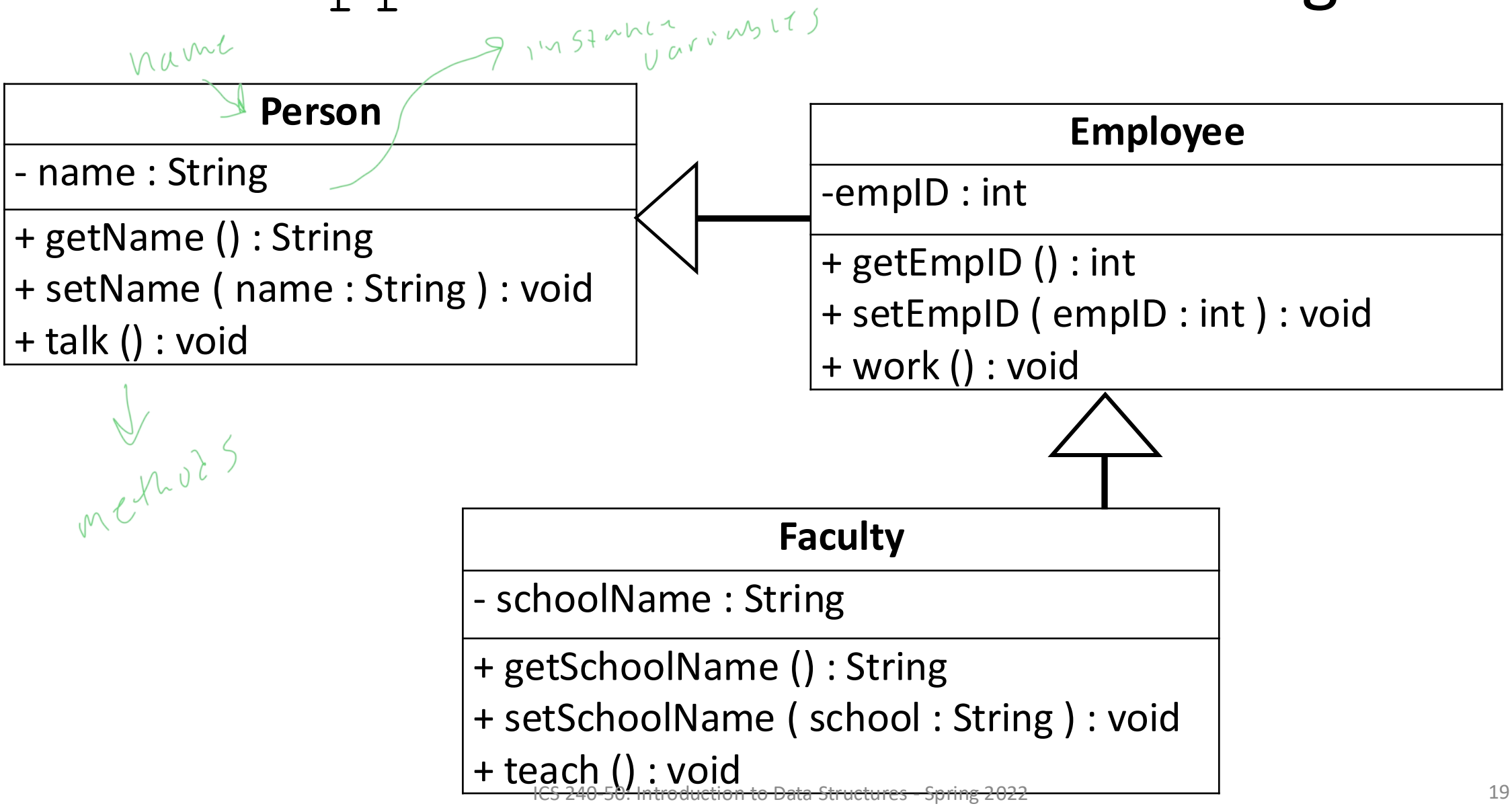
```
public class Faculty extends Employee {
  private String schoolName;
}
```



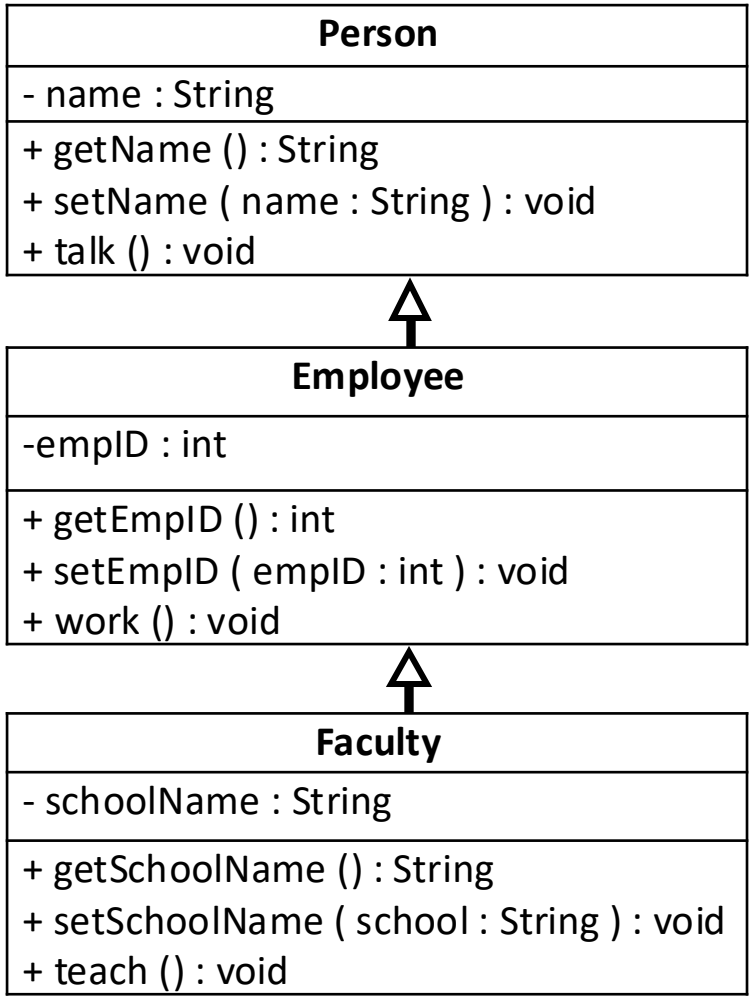
`PersonApplication` UML Structure Diagram



PersonApplication UML Class Diagrams



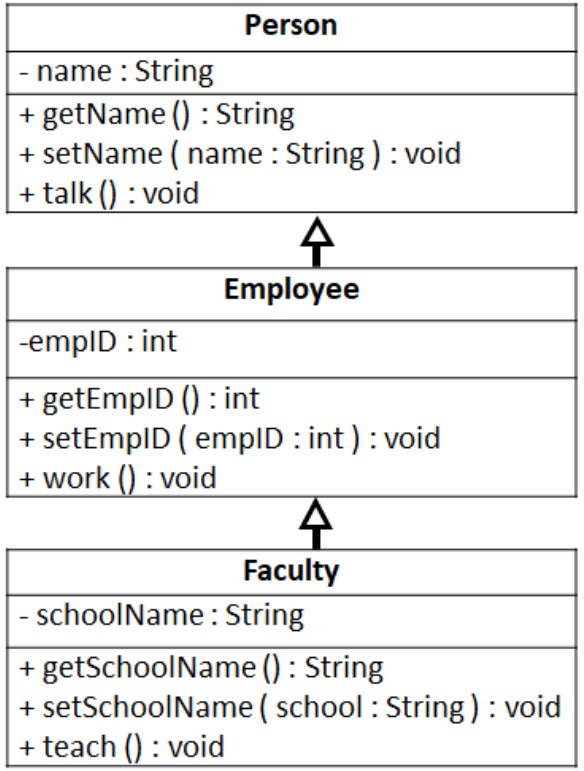
PersonApplication UML Class Diagrams



```
public class Person {
   private String name;

    public void talk(){
        System.out.println ("Hi.  I am a Person");
    }
}
```

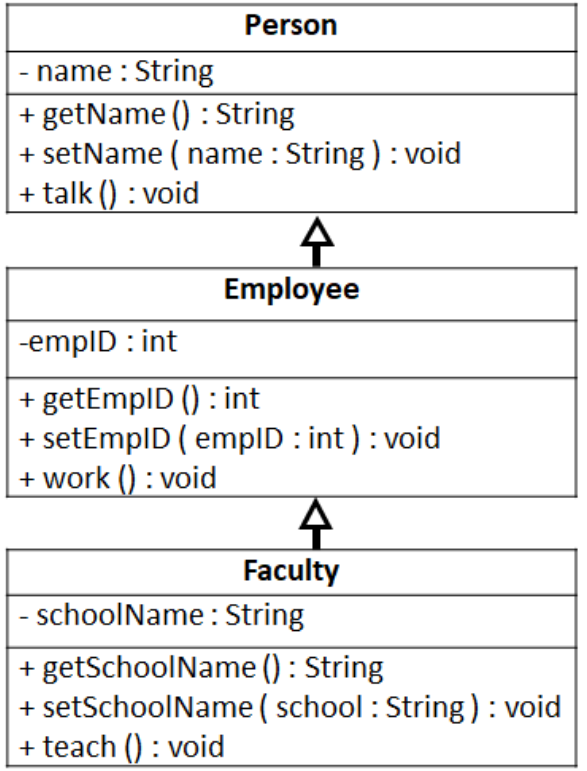
```
public class Employee extends Person {
  private int empID;

    public void work(){
        System.out.println("I am working");
    }
}
```

```
public class Faculty extends Employee {
  private String schoolName;

    public void teach()              {
        System.out.println("I am teaching");
    }
}
```



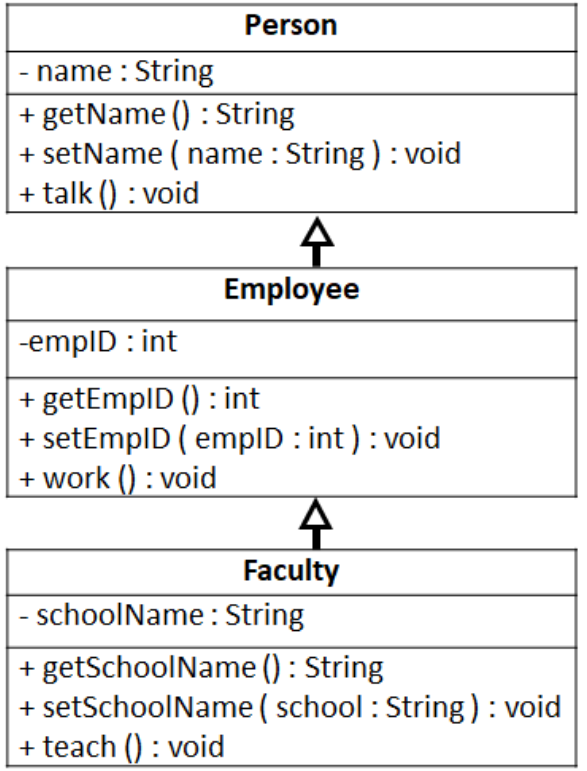
Ordinary Use of Classes

```
Person pers = new Person();
Employee emp = new Employee();
Faculty fac = new Faculty();

pers.talk();
emp.work();
fac.teach();
```



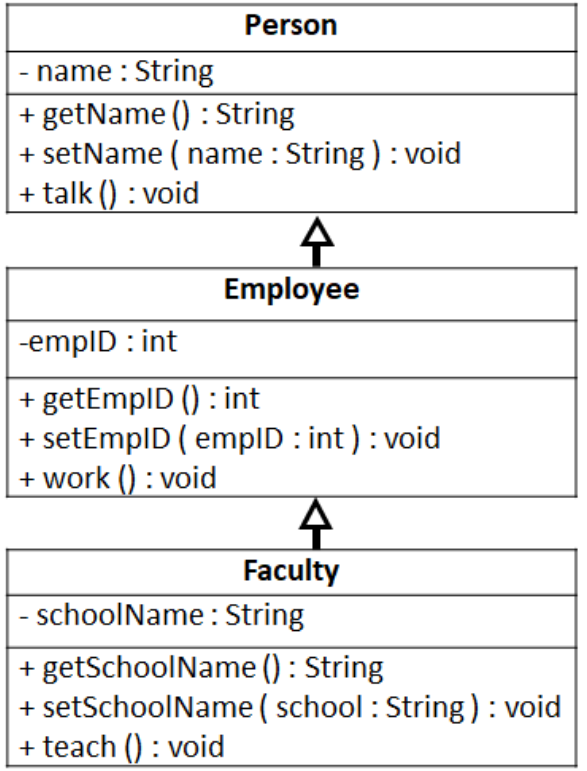
Calling an Inherited Method

```
Person pers = new Person();
Employee emp = new Employee();
Faculty fac = new Faculty();

emp.talk();// calling talk from People class
fac.work();// calling work from Employee class
fac.teach();
```

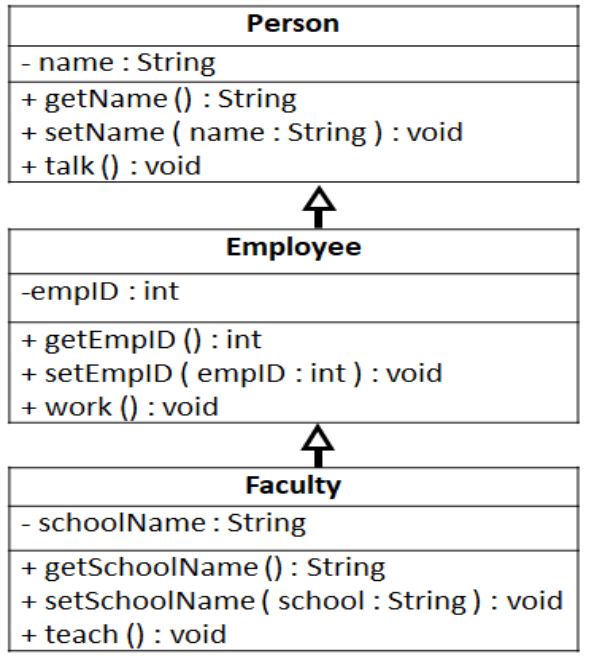


Inheritance Goes in One Direction

```
X emp.teach();   // An Employee is NOT a Faculty

X pers.work();   // A Person is NOT an Employee
```



Accessing Inherited Instance Variables



Legal

```
Person pers = new Person();
Employee emp = new Employee();
Faculty fac = new Faculty();

pers.setName("Ahmed");
emp.setName("Fred");
fac.setName("Sarah");

emp.setEmpID(202003);
fac.setEmpID(202015);

fac.setSchoolName("Metro");
```

Illegal

```
Xpers.setEmpID(201520);

Xemp.setSchoolName("Hamline");
Xpers.setSchoolName("Augsburg");
```

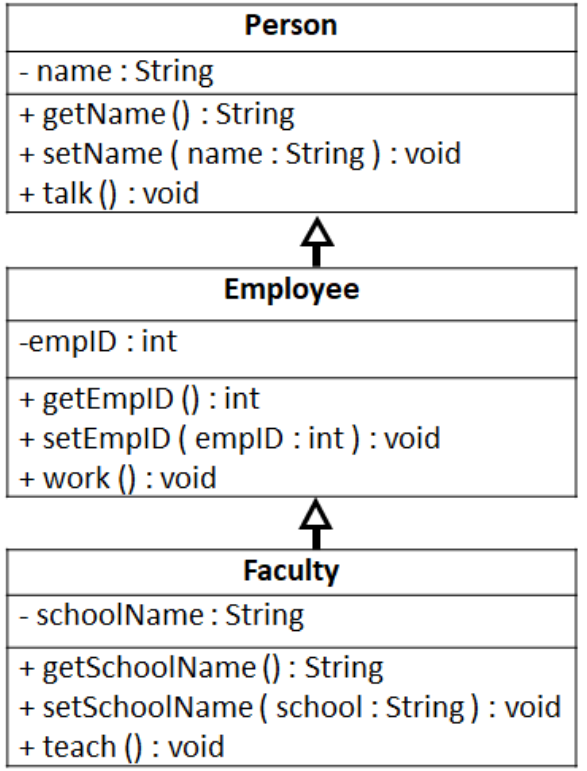
But Wait! There's More!

Legal

```
Person pers2 = new Employee();// Employee IS-A Person
Employee emp2 = new Faculty();// Faculty IS-A Employee
Person pers3 = new Faculty(); // Faculty IS-A Person
```

Illegal

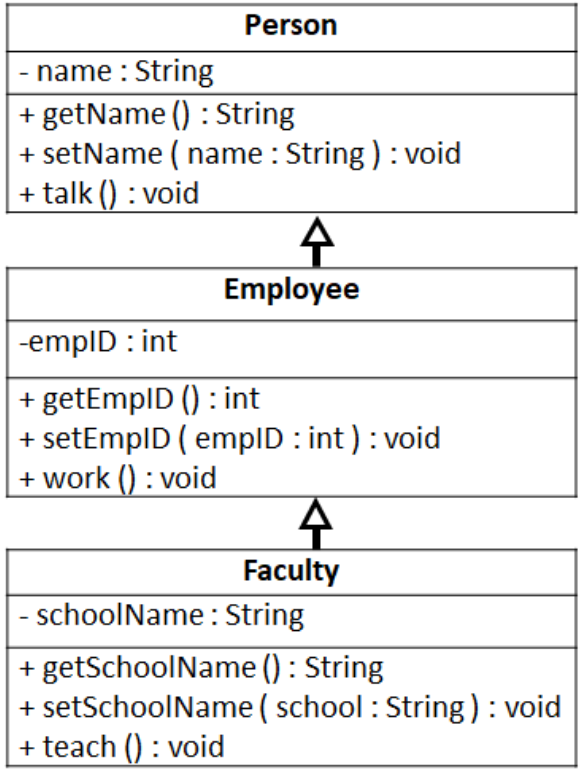
```
Faculty fac2 = new Employee();//Employee IS NOT A Faculty
Employee emp3 = new Person();  //Person is NOT A Employee
```



`instanceof` & casting

- Use `instanceof` operator to determine the type of object stored in the reference variable
- TIP: Test for the most specific cases first because the most general case will always be true

```
Person pers2 = new Employee();

if (pers2 instanceof Faculty)
   ((Faculty) pers2).teach();
else if (pers2 instanceof Employee)
   ((Employee) pers2).work();
else if (pers2 instanceof Person)
   pers2.talk();
```



Output:

```
I am working
```

The `Object` class in Java

- The class at the top of the Java class hierarchy is called `Object`
- All classes have a parent (i.e., superclass) class
  - If a class definition does not explicitly `extend` another class, then it automatically has `Object` as its parent class.
  - For example, `class Video` is equivalent to `class Video extends Object`
- All classes in Java share some common characteristics and those characteristics are defined in `Object`
  - For example, all classes have a `toString()` method because the class `Object` defines that method, so all classes get it by inheritance
  - When you write a class you `override` the `toString()` method

Important concepts

- Constructors are treated different than other methods in inheritance.
- Method overriding (`@override`)
- `protected` access modifier

How inheritance deals with Constructors?

- Constructors are **NOT** inherited.

- The first task of a subclass constructor is to call its direct superclass's constructor explicitly or implicitly
  - This is done to ensure that the instance variables inherited from the superclass are initialized properly.

- If the code does not include an explicit call to the superclass constructor, then Java implicitly calls the superclass's default or no-argument constructor.

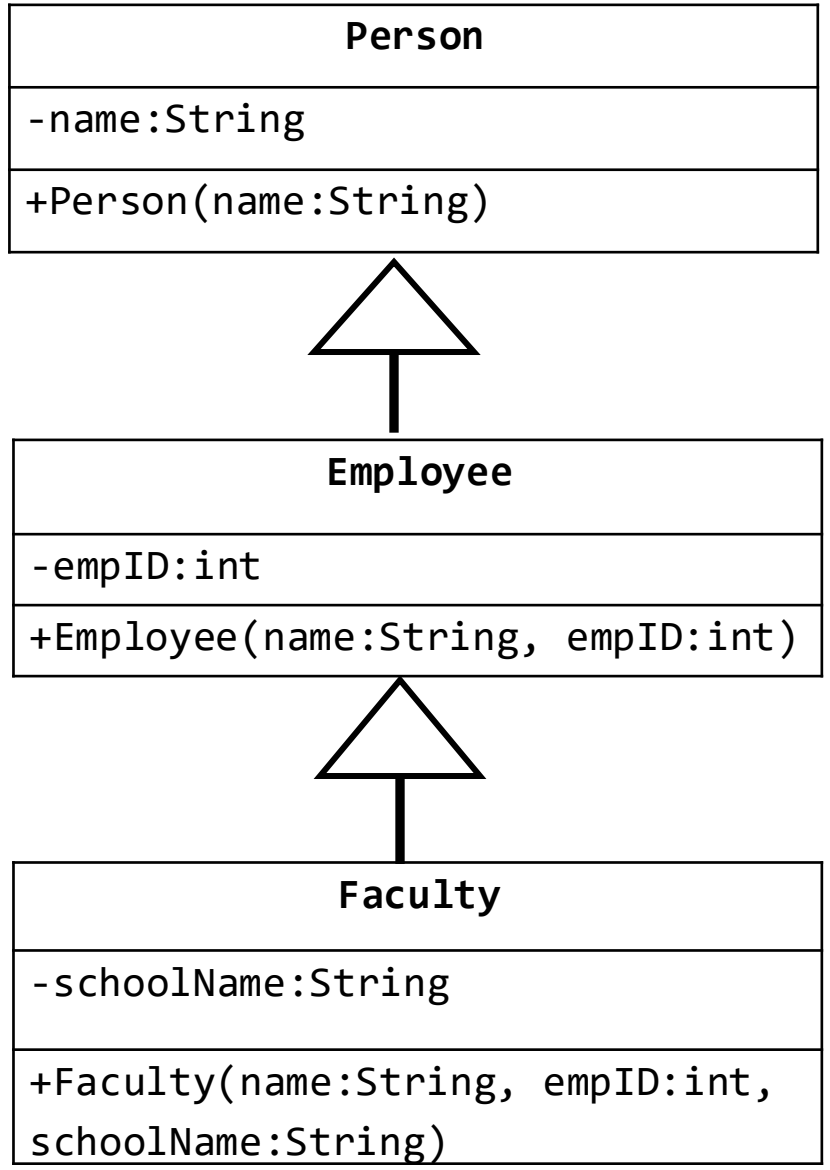
if there is one, if not exception.

PersonApplication

```
public class Person {
        private String name;
        public Person(String name) {
                this.name = name;
        }
}
```

```
public class Employee extends Person {
        private int empID;
        public Employee(String name,int empID) {
                super(name);
                this.empID = empID;
        }
}
```

```
public class Faculty extends Employee{
        private String schoolName;
        public Faculty(String name,int empID, String schoolName) {
                super(name,empID);
                this.schoolName = schoolName;
        }
}
```



Superclass Constructor is Always Invoked

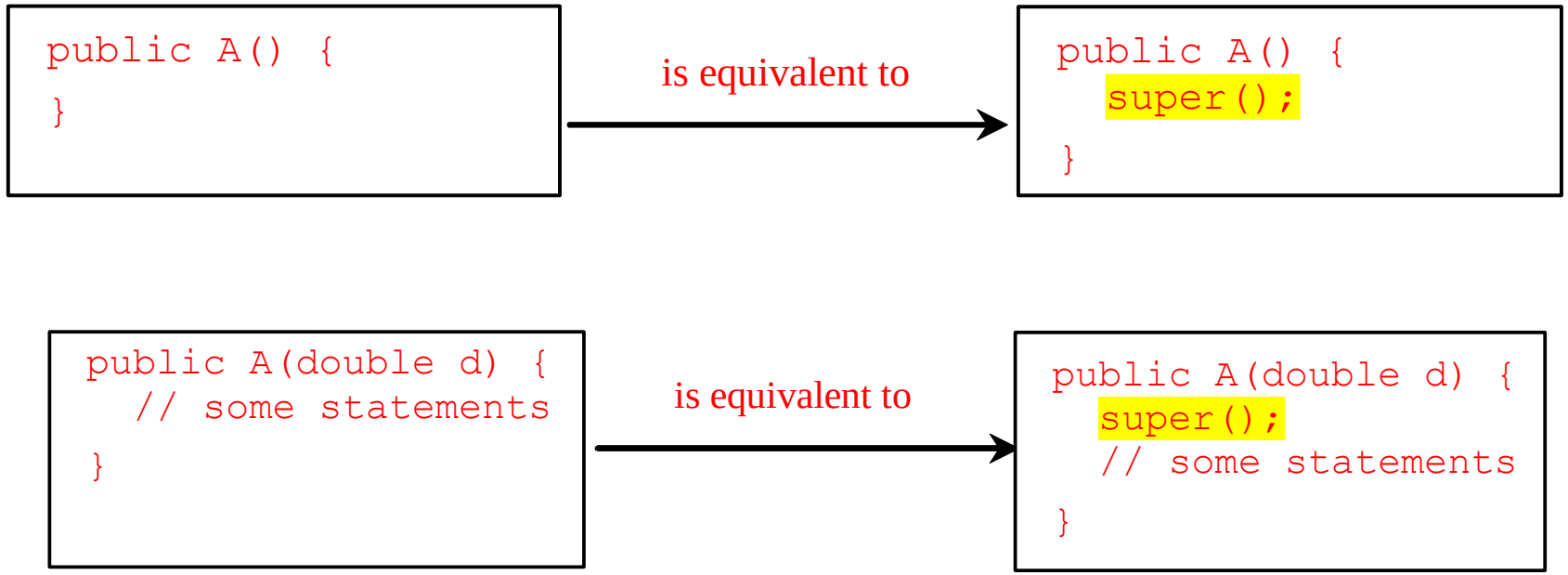
- A constructor may explicitly invoke an overloaded superclass constructor or the superclass default (no arguments) constructor

```
public A() {
  super("Hello", 52);
}
```

```
public A() {
  super();
}
```

- If not invoked explicitly, the compiler adds `super()` as the first statement in the constructor. This invokes the no arguments constructor of the superclass
- If the superclass does not have a default (no arguments) constructor, then you will get an error if you do not explicitly call a constructor with arguments from the superclass

Superclass Constructor is Always Invoked



Important notes about the default constructor

- You can explicitly write a no-argument constructor for a class.
- If you do not write any constructors for a class, then a no-argument constructor (called the **default constructor**) is automatically supplied.
- If you write even one constructor for a class, then the default constructor is **not** automatically supplied.
- Conclusion: if you write a constructor for a class, then you must also write a no-argument constructor if one is expected.

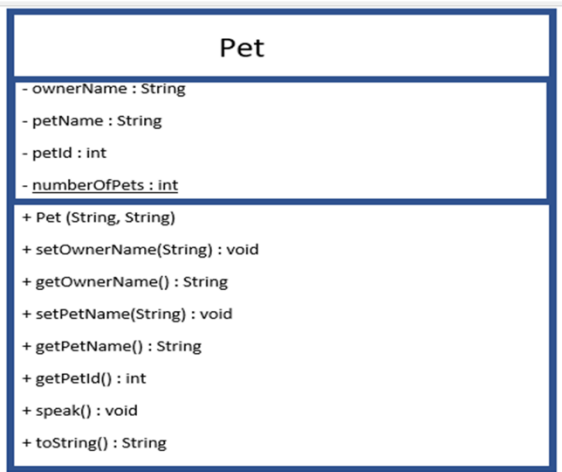
Another Example:

*Pet.java Cat.java Dog.java PetDriver.java

```java
public class Pet {
    private String ownerName;
    private String petName;
    private int petId;
    private static int numberOfPets = 0;

    public Pet(String oName, String pName) {
        ownerName = oName;
        petName = pName;
        petId = ++numberOfPets;
    }

    public void setOwnerName(String oName) {
        ownerName = oName;
    }

    public String getOwnerName() {
        return ownerName;
    }

    public void setPetName(String pName) {
        petName = pName;
    }

    public String getPetName() {
        return petName;
    }

    public int getPetId() {
        return petId;
    }

    public void speak() {
        System.out.println("Pet noises");
    }

    public String toString() {
        String output = new String(petName + " owned by " + ownerName + " has pet id number " + petId);
        return output;
```

*Pet.java Cat.java Dog.java PetDriver.java

```java
public class Pet {
    private String ownerName;
    private String petName;
    private int petId;
    private static int numberOfPets = 0;

    public Pet(String oName, String pName) {
        ownerName = oName;
        petName = pName;
        petId = ++numberOfPets;
    }

    public void setOwnerName(String oName) {
        ownerName = oName;
    }

    public String getOwnerName() {
        return ownerName;
    }

    public void setPetName(String pName) {
        petName = pName;
    }

    public String getPetName() {
        return petName;
    }

    public int getPetId() {
        return petId;
    }

    public void speak() {
        System.out.println("Pet noises");
    }

    public String toString() {
        String output = new String(petName + " owned by " + ownerName + " has pet id number " + petId);
        return output;
```



*Pet.java Cat.java Dog.java PetDriver.java

```java
public class Pet {
    private String ownerName;
    private String petName;
    private int petId;
    private static int numberOfPets = 0;

    public Pet(String oName, String pName) {
        ownerName = oName;
        petName = pName;
        petId = ++numberOfPets;
    }

    public void setOwnerName(String oName) {
        ownerName = oName;
    }

    public String getOwnerName() {
        return ownerName;
    }

    public void setPetName(String pName) {
        petName = pName;
    }

    public String getPetName() {
        return petName;
    }

    public int getPetId() {
        return petId;
    }

    public void speak() {
        System.out.println("Pet noises");
    }

    public String toString() {
        String output = new String(petName + " owned by " + ownerName + " has pet id number " + petId);
        return output;
```

Pet
- ownerName : String - petName : String - petId : int - numberOfPets : int
+ Pet (String, String) + setOwnerName(String) : void + getOwnerName() : String + setPetName(String) : void + getPetName() : String + getPetId() : int + speak() : void + toString() : String

*Pet.java *Cat.java *Dog.java PetDriver.java

```java
public class Dog extends Pet{
    private int numBones;

    public Dog(String oName, String pName, int bones){
        super(oName, pName);
        numBones = bones;
    }

    public void setNumBones(int bones){
        numBones = bones;
    }

    public int getNumBones(){
        return numBones;
    }

    public void speak(){
        System.out.println("Woof");
    }

    public String toString(){
        String output = super.toString();
        output += new String(" and this dog has " +
                numBones + " bones.");
        return output;
    }
}
```

Dog
- numBones : int
+ Dog (String, String, int) + setNumBones(int) : void + getNumBones() : int + speak() : void + toString() : String

*Pet.java Cat.java Dog.java PetDriver.java

```java
public class Pet {
    private String ownerName;
    private String petName;
    private int petId;
    private static int numberOfPets = 0;

    public Pet(String oName, String pName) {
        ownerName = oName;
        petName = pName;
        petId = ++numberOfPets;
    }

    public void setOwnerName(String oName) {
        ownerName = oName;
    }

    public String getOwnerName() {
        return ownerName;
    }

    public void setPetName(String pName) {
        petName = pName;
    }

    public String getPetName() {
        return petName;
    }

    public int getPetId() {
        return petId;
    }

    public void speak() {
        System.out.println("Pet noises");
    }

    public String toString() {
        String output = new String(petName + " owned by " + ownerName + " has pet id number " + petId);
        return output;
```

Pet
- ownerName : String - petName : String - petId : int - numberOfPets : int
+ Pet (String, String) + setOwnerName(String) : void + getOwnerName() : String + setPetName(String) : void + getPetName() : String + getPetId() : int + speak() : void + toString() : String

*Pet.java *Cat.java *Dog.java PetDriver.java

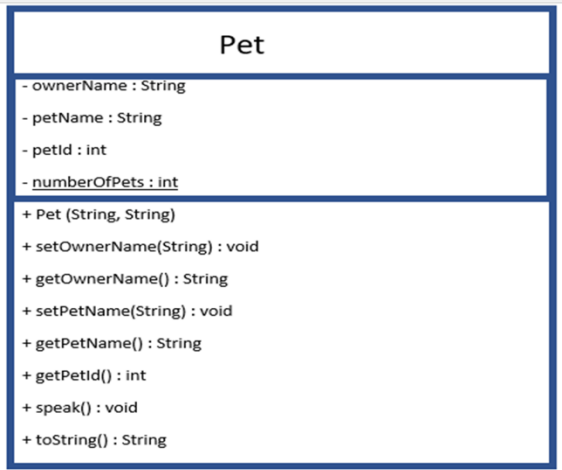
```java

public class Dog extends Pet{
    private int numBones;

    public Dog(String oName, String pName, int bones){
        super(oName, pName);
        numBones = bones;
    }

    public void setNumBones(int bones){
        numBones = bones;
    }

    public int getNumBones(){
        return numBones;
    }

    public void speak(){
        System.out.println("Woof");
    }

    public String toString(){
        String output = super.toString();
        output += new String(" and this dog has " +
                numBones + " bones.");
        return output;
    }
}
```

←This refers to the parametrized constructor

Dog
- numBones : int
+ Dog (String, String, int) + setNumBones(int) : void + getNumBones() : int + speak() : void + toString() : String

*Pet.java Cat.java Dog.java PetDriver.java

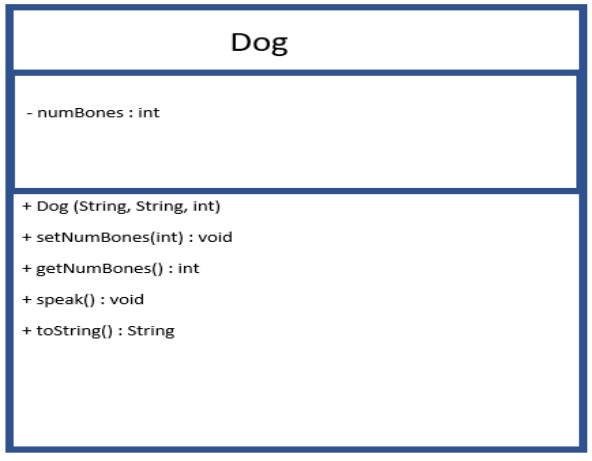
```java
public class Pet {
    private String ownerName;
    private String petName;
    private int petId;
    private static int numberOfPets = 0;

    public Pet(String oName, String pName) {
        ownerName = oName;
        petName = pName;
        petId = ++numberOfPets;
    }

    public void setOwnerName(String oName) {
        ownerName = oName;
    }

    public String getOwnerName() {
        return ownerName;
    }

    public void setPetName(String pName) {
        petName = pName;
    }

    public String getPetName() {
        return petName;
    }

    public int getPetId() {
        return petId;
    }

    public void speak() {
        System.out.println("Pet noises");
    }

    public String toString() {
        String output = new String(petName + " owned by " + ownerName + " has pet id number " + petId);
        return output;
```

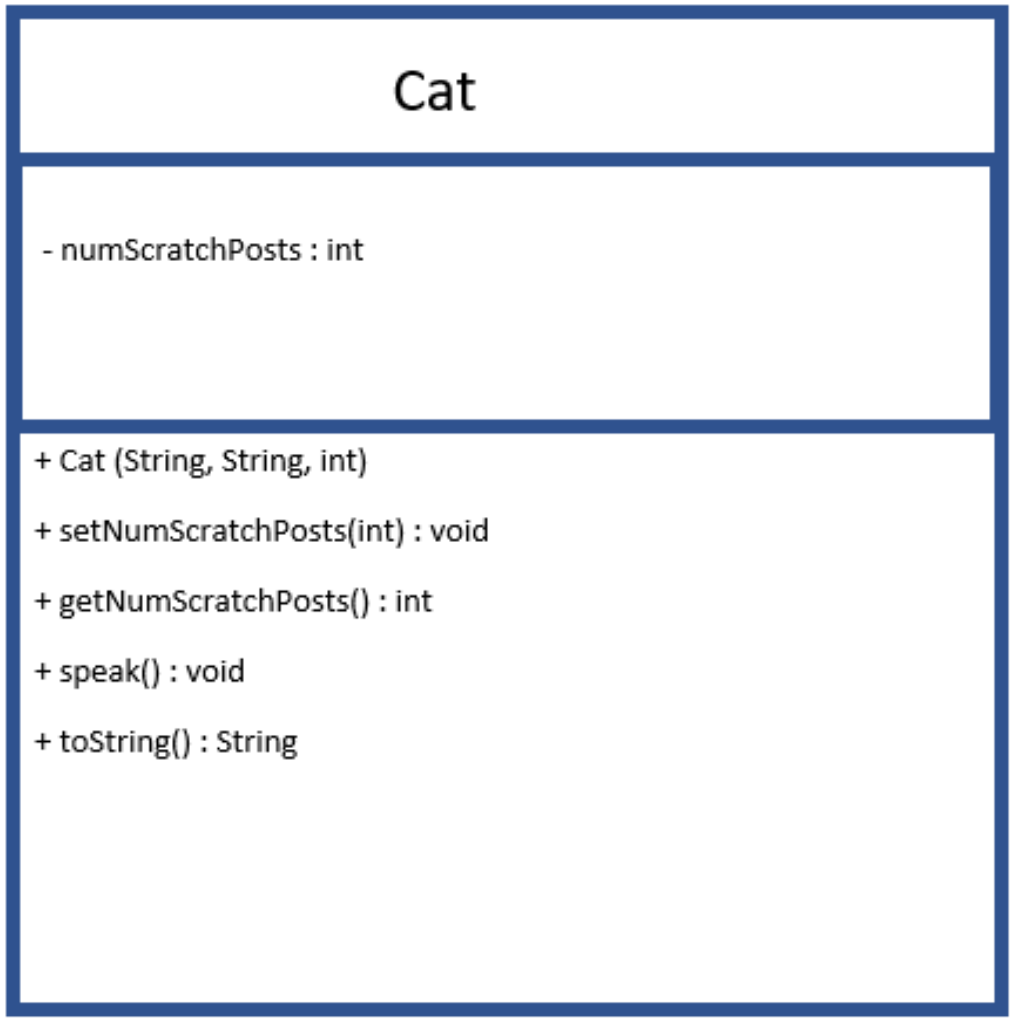


*Pet.java *Cat.java *Dog.java PetDriver.java

```java
public class Dog extends Pet{
    private int numBones;

    public Dog(String oName, String pName, int bones){
        super(oName, pName);
        numBones = bones;
    }

    public void setNumBones(int bones){
        numBones = bones;
    }

    public int getNumBones(){
        return numBones;
    }

    public void speak(){
        System.out.println("Woof");
    }

    public String toString(){
        String output = super.toString();
        output += new String(" and this dog has " +
                numBones + " bones.");
        return output;
    }
}
```

←This refers to the parametrized constructor (wouldn't be necessary if no-argument constructor)

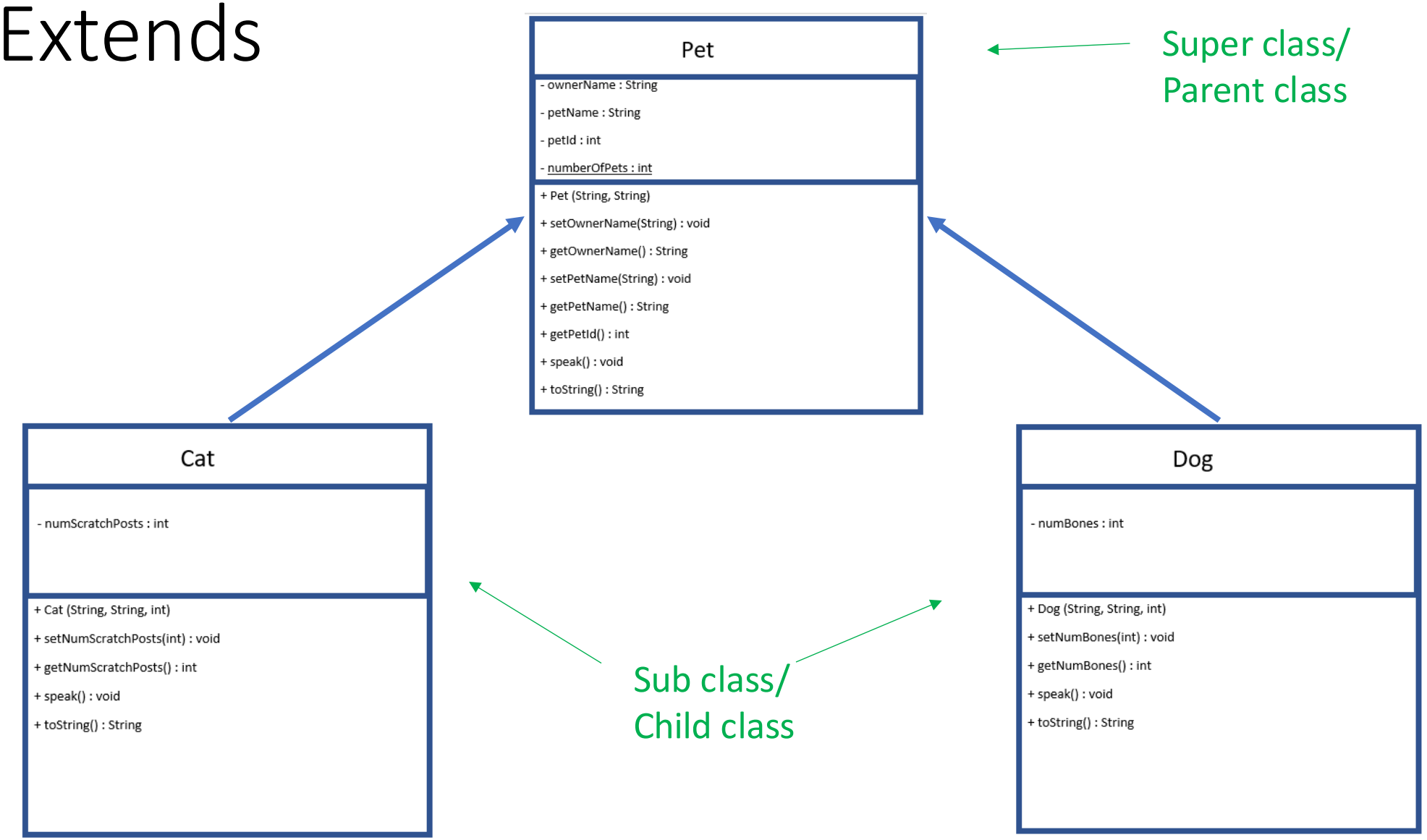




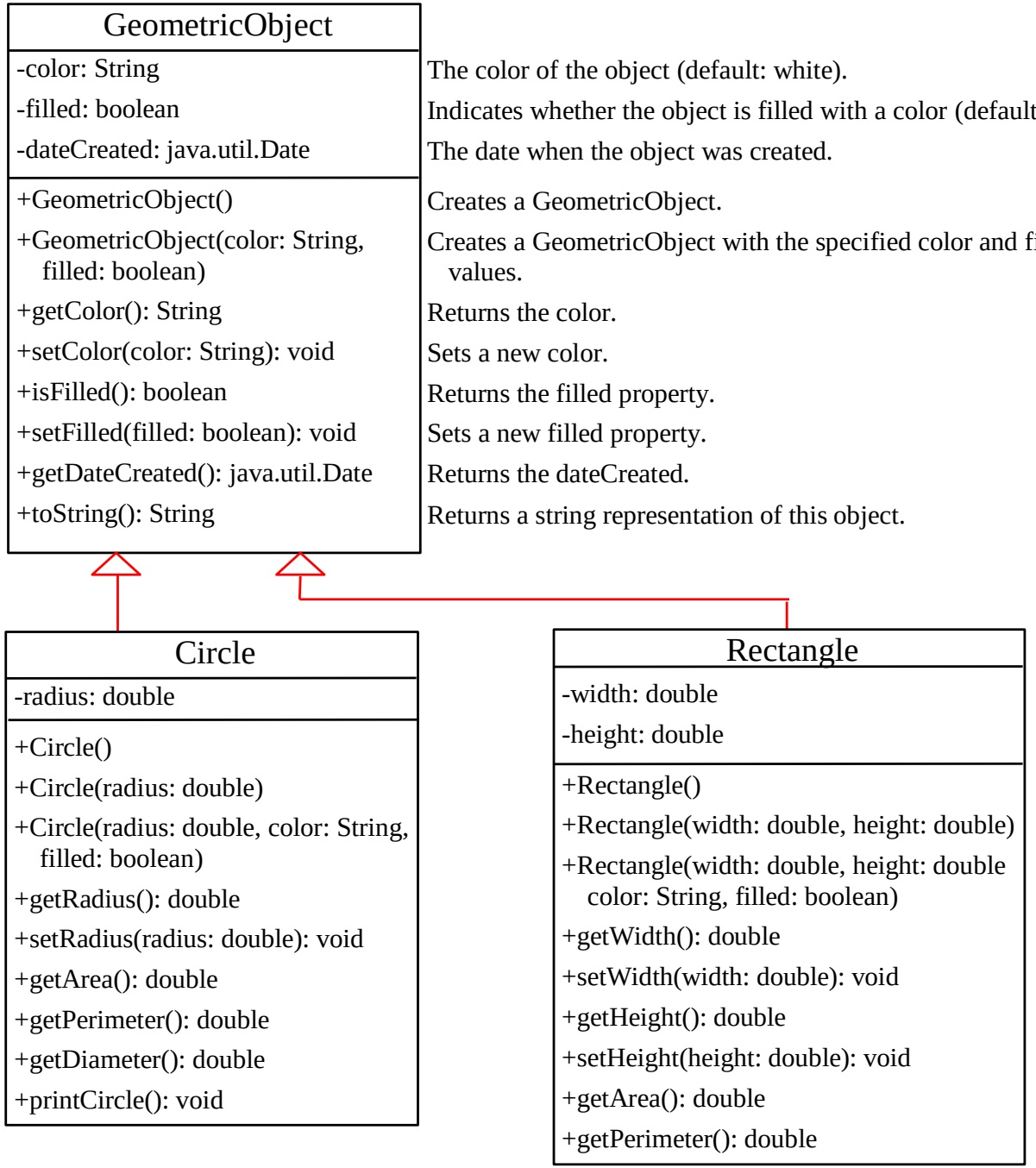
*Pet.java | *Cat.java | *Dog.java | PetDriver.java

```java

public class Cat extends Pet{
    private int numScratchPosts;

    public Cat(String oName, String pName, int posts){
        super(oName, pName);
        numScratchPosts = posts;
    }

    public void setNumScratchPosts(int posts){
        numScratchPosts = posts;
    }

    public int getNumScratchPosts(){
        return numScratchPosts;
    }

    public void speak(){
        System.out.println("Meow");
    }

    public String toString(){
        String output = super.toString();
        output += new String(" and this cat has " +
                numScratchPosts + " scratching posts.");
        return output;
    }

}
```

Extends


Pet
- ownerName : String
- petName : String
- petId : int
- numberOfPets : int
+ Pet (String, String)
+ setOwnerName(String) : void
+ getOwnerName() : String
+ setPetName(String) : void
+ getPetName() : String
+ getPetId() : int
+ speak() : void
+ toString() : String
Super class/
Parent class
Cat
- numScratchPosts : int
+ Cat (String, String, int)
+ setNumScratchPosts(int) : void
+ getNumScratchPosts() : int
+ speak() : void
+ toString() : String
Dog
- numBones : int
+ Dog (String, String, int)
+ setNumBones(int) : void
+ getNumBones() : int
+ speak() : void
+ toString() : String
Sub class/
Child class

Another Example: Geometric Objects



Useful Notes: Using the Keyword `super`

The keyword super refers to the superclass of the class in which super appears. This keyword can be used in two ways:

- To call a superclass constructor
- To call a superclass method

- Caution: You must use the keyword super to call the superclass constructor.
- Invoking a superclass constructor's name in a subclass causes a syntax error.
- Java requires that the statement that uses the keyword super appear **first** in the constructor.

Constructor Chaining

- Constructing an instance of a class invokes all the superclasses' constructors along the inheritance chain.
- This is called *constructor chaining*.

```
public class FacultyDriver{
  public static void main(String[] args) {
    Faculty fac = new Faculty();
  }
}
```

```
public class Faculty extends Employee {
  public Faculty() {
    System.out.println("(3) Faculty's no-arg constructor is invoked");
  }
}
```

```
class Employee extends Person {
  public Employee() {
    System.out.println("(2) Employee's no-arg constructor is invoked");
  }
  public Employee(String s) {
    System.out.println(s);
  }
}
```

```
class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

OUTPUT

Constructor Chaining

- Constructing an instance of a class invokes all the superclasses' constructors along the inheritance chain.
- This is called *constructor chaining*.

```
public class FacultyDriver{
  public static void main(String[] args) {
    Faculty fac = new Faculty();
  }
}
```

← A Faculty object is instantiated

```
public class Faculty extends Employee {
  public Faculty() {
    System.out.println("(3) Faculty's no-arg constructor is invoked");
  }
}
```

```
class Employee extends Person {
  public Employee() {
    System.out.println("(2) Employee's no-arg constructor is invoked");
  }
  public Employee(String s) {
    System.out.println(s);
  }
}
```

```
class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

OUTPUT

Constructor Chaining

- Constructing an instance of a class invokes all the superclasses' constructors along the inheritance chain.
- This is called *constructor chaining*.

```
public class FacultyDriver{
  public static void main(String[] args) {
    Faculty fac = new Faculty();
  }
}
```

← A Faculty object is instantiated

```
public class Faculty extends Employee {
  public Faculty() {
    System.out.println("(3) Faculty's no-arg constructor is invoked");
  }
}
```

← (1) The no-argument constructor of Employee is called first

```
class Employee extends Person {
  public Employee() {
    System.out.println("(2) Employee's no-arg constructor is invoked");
  }
  public Employee(String s) {
    System.out.println(s);
  }
}
```

```
class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

OUTPUT

Constructor Chaining

- Constructing an instance of a class invokes all the superclasses' constructors along the inheritance chain.
- This is called *constructor chaining*.

```
public class FacultyDriver{
  public static void main(String[] args) {
    Faculty fac = new Faculty();
  }
}
```

← A Faculty object is instantiated

```
public class Faculty extends Employee {
  public Faculty() {
    System.out.println("(3) Faculty's no-arg constructor is invoked");
  }
}
```

← (1) The no-argument constructor of Employee is called first

```
class Employee extends Person {
  public Employee() {
    System.out.println("(2) Employee's no-arg constructor is invoked");
  }
  public Employee(String s) {
    System.out.println(s);
  }
}
```

← (2) The no-argument constructor of Person is then called

```
class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

OUTPUT

Constructor Chaining

- Constructing an instance of a class invokes all the superclasses' constructors along the inheritance chain.
- This is called *constructor chaining*.

```
public class FacultyDriver{
  public static void main(String[] args) {
    Faculty fac = new Faculty();
  }
}
```

← A Faculty object is instantiated

```
public class Faculty extends Employee {
  public Faculty() {
    System.out.println("(3) Faculty's no-arg constructor is invoked");
  }
}
```

← (1) The no-argument constructor of Employee is called first

```
class Employee extends Person {
  public Employee() {
    System.out.println("(2) Employee's no-arg constructor is invoked");
  }
  public Employee(String s) {
    System.out.println(s);
  }
}
```

← (2) The no-argument constructor of Person is then called

```
class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

← (3) The no-argument constructor of Person is executed

OUTPUT

```
(1) Person's no-arg
    constructor is invoked
```

Constructor Chaining

- Constructing an instance of a class invokes all the superclasses' constructors along the inheritance chain.
- This is called *constructor chaining*.

```
public class FacultyDriver{
  public static void main(String[] args) {
    Faculty fac = new Faculty();
  }
}
```

← A Faculty object is instantiated

```
public class Faculty extends Employee {
  public Faculty() {
    System.out.println("(3) Faculty's no-arg constructor is invoked");
  }
}
```

← (1) The no-argument constructor of Employee is called first

```
class Employee extends Person {
  public Employee() {
    System.out.println("(2) Employee's no-arg constructor is invoked");
  }
  public Employee(String s) {
    System.out.println(s);
  }
}
```

← (2) The no-argument constructor of Person is then called

← (4) The no-argument constructor of Employee is executed

```
class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

← (3) The no-argument constructor of Person is executed

OUTPUT

```
(1) Person's no-arg
   constructor is invoked
(2) Employee's no-arg
constructor is invoked
```

Constructor Chaining

- Constructing an instance of a class invokes all the superclasses' constructors along the inheritance chain.
- This is called *constructor chaining*.

```
public class FacultyDriver{
  public static void main(String[] args) {
    Faculty fac = new Faculty();
  }
}
```

← A Faculty object is instantiated

```
public class Faculty extends Employee {
  public Faculty() {
    System.out.println("(3) Faculty's no-arg constructor is invoked");
  }
}
```

← (1) The no-argument constructor of Employee is called first

← (5) The no-argument constructor of Faculty is executed

```
class Employee extends Person {
  public Employee() {
    System.out.println("(2) Employee's no-arg constructor is invoked");
  }
  public Employee(String s) {
    System.out.println(s);
  }
}
```

← (2) The no-argument constructor of Person is then called

← (4) The no-argument constructor of Employee is executed

```
class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

← (3) The no-argument constructor of Person is executed

OUTPUT

```
(1) Person's no-arg
   constructor is invoked
(2) Employee's no-arg
constructor is invoked
(3) Faculty's no-arg
constructor is invoked
```

Visibility Modifiers

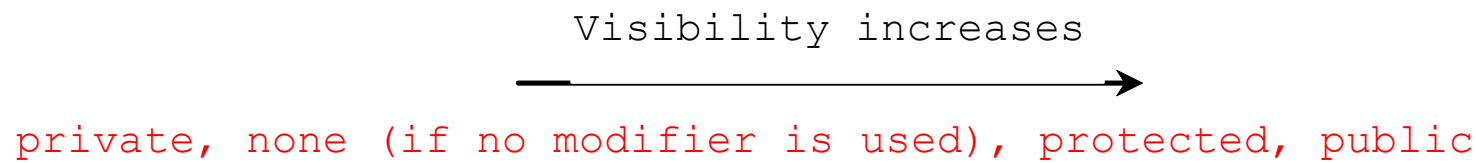
Accessing `private` members

- Methods of a subclass cannot directly access `private` members of their superclass.

- `private` variables can be accessed only through the `public` setters and getters methods.

A Subclass Cannot Weaken the Accessibility

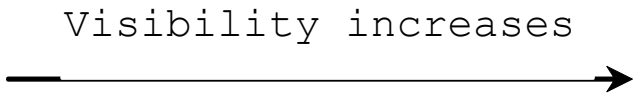
- The `protected` modifier can be applied on data and methods in a class.
- A protected data or a protected method in a public class can be accessed by any class in the same package or its subclasses, even if the subclasses are in a different package.
- private, default, protected, public



The `protected` Modifier

- A subclass may override a protected method in its superclass and change its visibility to public.
- However, a subclass cannot weaken the accessibility of a method defined in the superclass.
- For example, if a method is defined as public in the superclass, it must be defined as public in the subclass.

Almost always, make fields private and methods public.



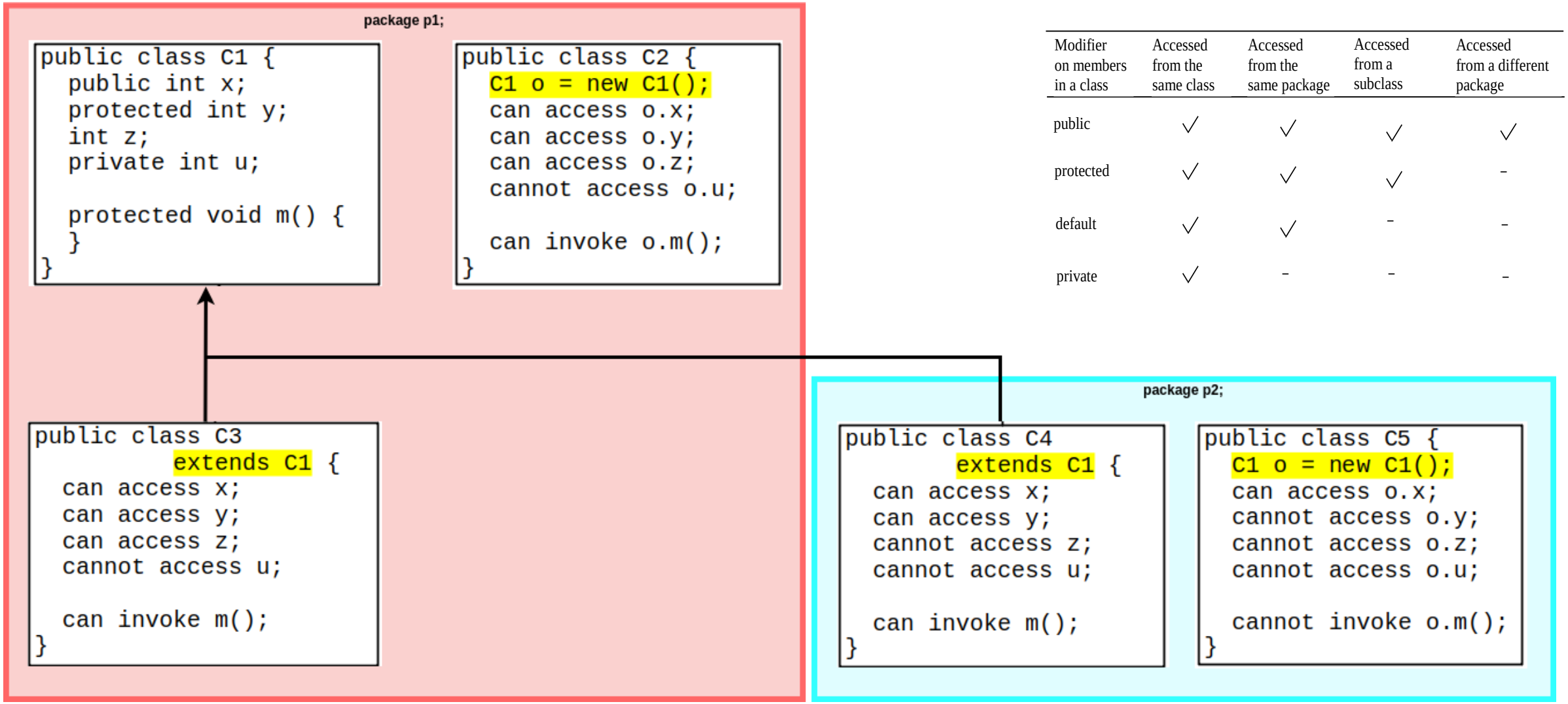
`private, none (if no modifier is used), protected, public`

Visibility Modifiers - summary

Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default	✓	✓	–	–
private	✓	–	–	–

Beware private variables of parent class!

Visibility Modifiers – a better way to look at it…



Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	-
default	✓	✓	-	-
private	✓	-	-	-

Overriding vs Overloading

The Object Class and Its Methods

Every class in Java is descended from the java.lang.Object class.

If no inheritance is specified when a class is defined, the superclass of the class is Object.

```
public class Circle {
  ...
}
```

Equivalent

```
public class Circle extends Object {
  ...
}
```

The toString() method in Object

The toString() method returns a string representation of the object. The default implementation returns a string consisting of a class name of which the object is an instance, the at sign (@), and a number representing this object.

```
Loan loan = new Loan();
System.out.println(loan.toString());
```

The code displays something like Loan@15037e5 . This message is not very helpful or informative. Usually you should override the toString method so that it returns a digestible string representation of the object.

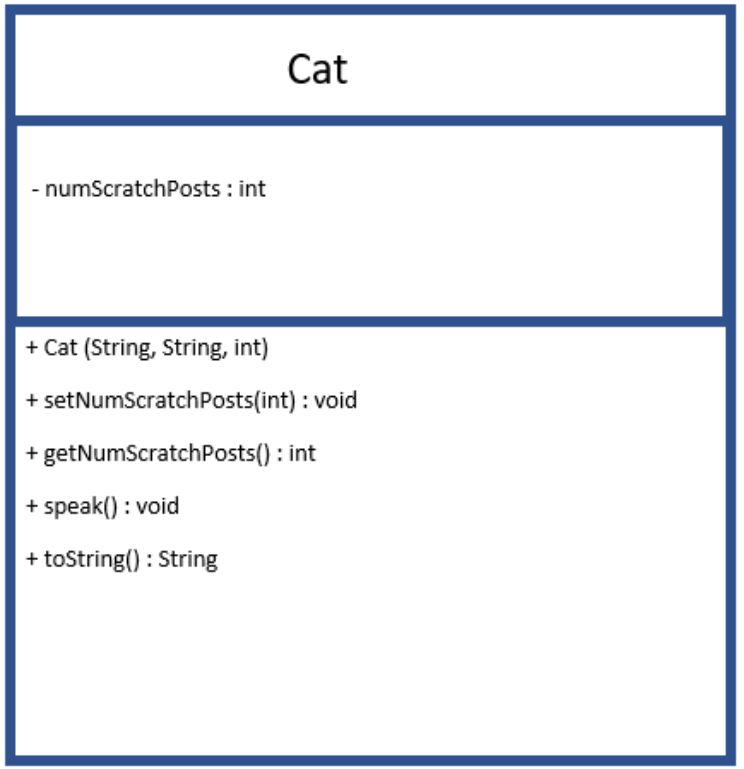
Overriding Methods

- A subclass can override any of the super class methods
- For example, `class Faculty` may override the inherited `toString()` method from `class Person` to add the `schoolName` to the output string.

```
@Override
public String toString(){
	String output = super.toString();
	output += this.schoolName;
	return output;
}
```

*Pet.java | *Cat.java | *Dog.java | PetDriver.java

```
1
2  public class Cat extends Pet{
3      private int numScratchPosts;
4
5      public Cat(String oName, String pName, int posts){
6          super(oName, pName);
7          numScratchPosts = posts;
8      }
9
10     public void setNumScratchPosts(int posts){
11         numScratchPosts = posts;
12     }
13
14     public int getNumScratchPosts(){
15         return numScratchPosts;
16     }
17
18     public void speak(){
19         System.out.println("Meow");
20     }
21
22     public String toString(){
23         String output = super.toString();
24         output += new String(" and this cat has " +
25                 numScratchPosts + " scratching posts.");
26         return output;
27     }
28
29 }
```



Overloading

- Overloading allows you to have more than one method with the same name (often within one class) but with different method signature where the method signature consists of
  - The name of the method
  - The number of parameters
  - Their types

Return type is not considered even though it is part of the signature.

- The compiler chooses the correct version of the method based on the number and type of the parameters used when the method is called

Overriding vs. Overloading

The following methods uses `@Overriding` from `Object`

- `String toString ()`
- `boolean equals (Object)`

Overloading is application-dependent

- `Movie (String title, int length)`
- `Movie (String title, int length, String director)`
- `Movie (String title, int length, String director, String rating)`

Overriding vs. Overloading

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overrides the method in B
  public void p(double i) {
    System.out.println(i);
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overloads the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

Abstract Classes

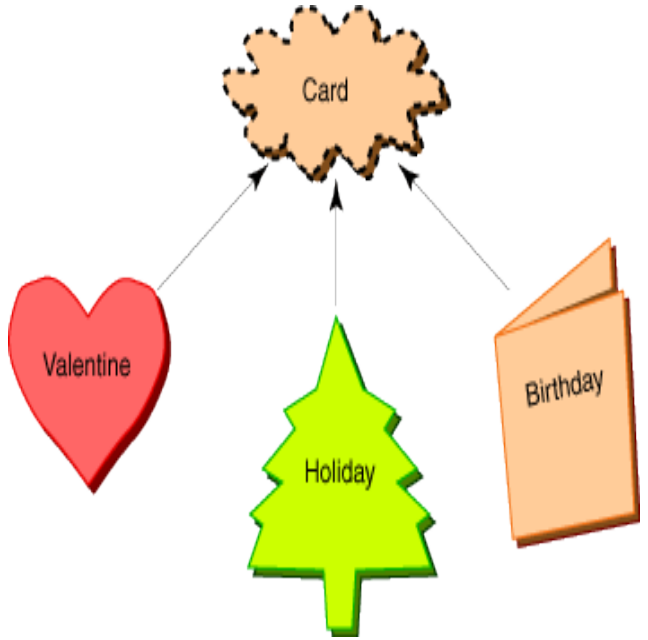
`abstract` classes

- An `abstract` class is a class that can**NOT** be instantiated, but it can be the parent of other classes.
- This is useful when you have a broad concept (like `Animal`) but actual objects must be specific types (like `Bird`).
- Even though it can not be instantiated, an abstract class defines methods and variables that children classes inherit.
- The advantage of using an abstract class is that you can group several related classes together as **siblings.**

`GreetingCard` Hierarchy

- In this example, an object must be an instance of one of the three children types: `Valentine`, `Holiday`, and `Birthday`.

There will never be an object that is merely a "Card".



`abstract` Methods

- In the `GreetingCard` application example, each card class has its own version of the `greeting()` method but each one is implemented differently.

- Hence, `greeting()` should be defines as an `abstract` method in the parent class.

- This means that each child inherits the "idea" of `greeting()`, but each implementation is different.

```
public abstract class Card{
  private String recipient;      // name of who gets the card
  public abstract void greeting();  // abstract greeting() method
}
```

Example of an Abstract Class

```
public abstract class Card           {
   private String recipient;       // name of who gets the card
   public abstract void greeting();  // abstract greeting() method

   public Card(String r){
      recipient = r;
   }
   public String getRecipient() {
      return recipient;
   }
   public void setRecipient(String recipient) {
      this.recipient = recipient;
   }
}
```

constructor

accessor

modifier

When to use `abstract` methods?

- An `abstract` is defined by just the method header that declares: an access modifier, return type, and method signature followed by a semicolon. The method has no implementation.
- Abstract classes can (but don't have to) contain abstract methods.
- An abstract class can contain non-abstract methods, which will be inherited by the children.
- A non-abstract child class inherits the abstract method and must define a **concrete** method (i.e., implementation) for all abstract methods.

Concrete Classes extend Abstract Classes

```
public class Holiday extends Card {


    public Holiday( String r )  {
        super(r);
    }


    public void greeting()    {
        System.out.println("Dear " +
        getRecipient() + ",");
        System.out.println("Season's
        Greetings!\n");
    }
}
```

```
class Valentine extends Card {
   private int kisses;

    public Valentine (String r, int k ){
        super(r);
        kisses = k;
    }

    public void greeting(){
        System.out.println("Dear " +
        getRecipient() + ",");
        System.out.println("Love and
        Kisses,");
        for ( int j=0; j<kisses; j++ )
            System.out.print("X");
        System.out.println("\n");
    }
}
```

Important:
You cannot instantiate an `abstract` class

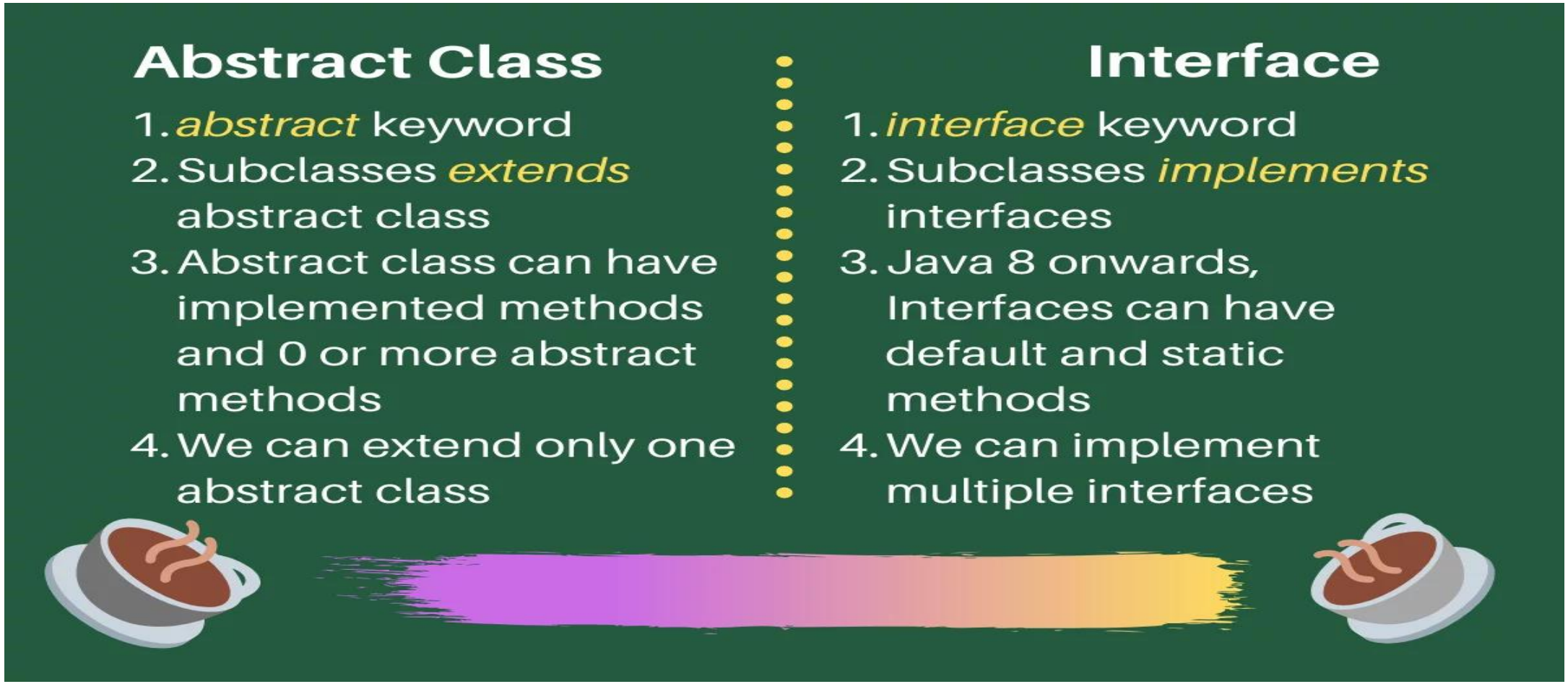
- The following statement will generate an error because you cannot instantiate an instance of the abstract class `Card`

```
Card card = new Card() ;
```

- However, the following is correct because `Valentine` **is-a** `Card`

```
Card card = new Valentine( "Joe", 14 ) ;
```

Abstract class vs. Interface



Source: https://www.journaldev.com/1607/difference-between-abstract-class-and-interface-in-java

`abstract class` or `interface`? (1/2)

Whether to choose between `interface` or `abstract class` for providing a contract for subclasses is a design decision and depends on many factors.

- Java doesn't support multiple class level inheritance, so every class can extend only one superclass, however, a class can implement multiple interfaces.

- If there are a lot of methods in the contract, then abstract class is more useful because we can provide a default implementation for some of the methods that are common for all the subclasses.

Source: https://www.journaldev.com/1607/difference-between-abstract-class-and-interface-in-java

abstract class or interface?(2/2)

- When abstract class is used, if subclasses don't need to implement a particular method, they can avoid providing the implementation but in case of interface, the subclass will have to provide the implementation for all the methods even though it's of no use and implementation is just empty block.

- If our base contract keeps on changing then interfaces can cause issues because we can't declare additional methods to the interface without changing all the implementation classes, with the abstract class we can provide the default implementation and only change the implementation classes that are actually going to use the new methods.

Source: https://www.journaldev.com/1607/difference-between-abstract-class-and-interface-in-java

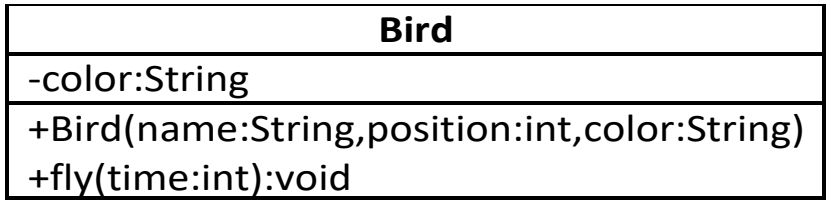
Example: `AnimalApplication`

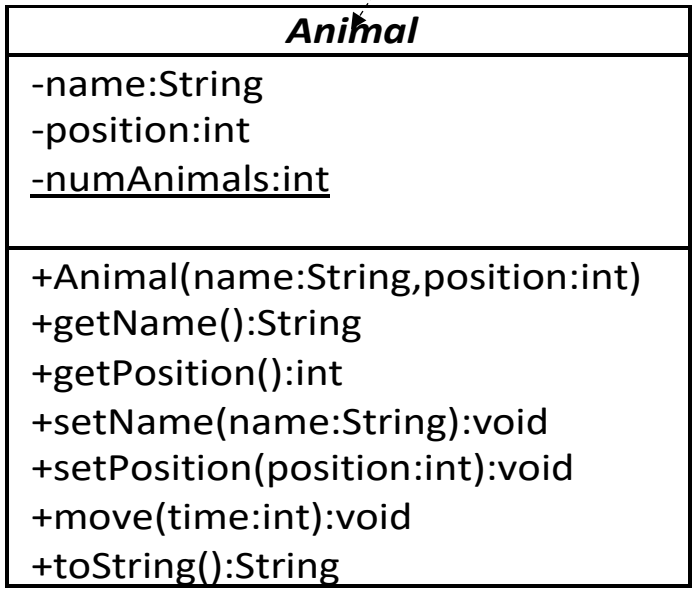
- Create `abstract` class Animal:
  - Instance variables:
    - `name:String`
    - `position: int`
    - `numAnimals:int`
      - a static variable that is used to represent the number of created animals
  - Methods:
    - `Constructor`
    - `Getters and setters`
    - `toString`
    - `move(time:int)`
      - assume by default an animal moves 2 miles per unit of time

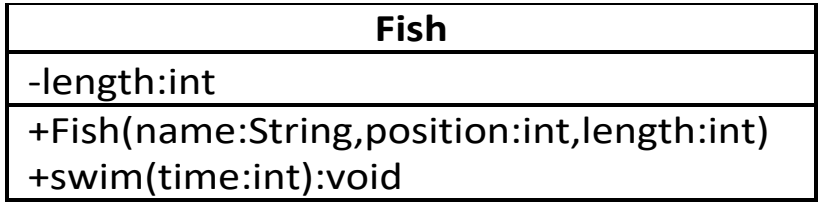
`AnimalApplication` (continued)

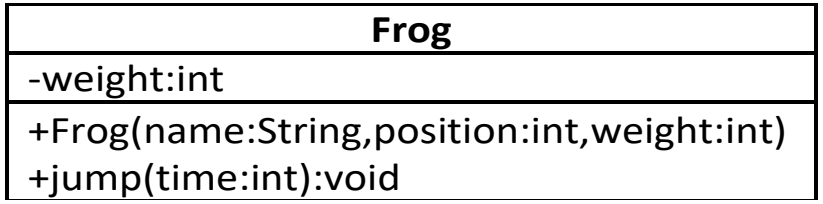
- Declare three subclasses:
  - Bird:
    - `color:String`
    - `fly(time:int):void`
      - A bird flies 5 miles per unit of time
  - Fish:
    - `length: int`
    - `swim(time:int):void`
      - A fish swims 3 miles per unit of time
  - Frog:
    - `Weight: int`
    - `jump(time:int):void`
      - A frog jumps 1 miles per unit of time

abstract class is *italic* in UML



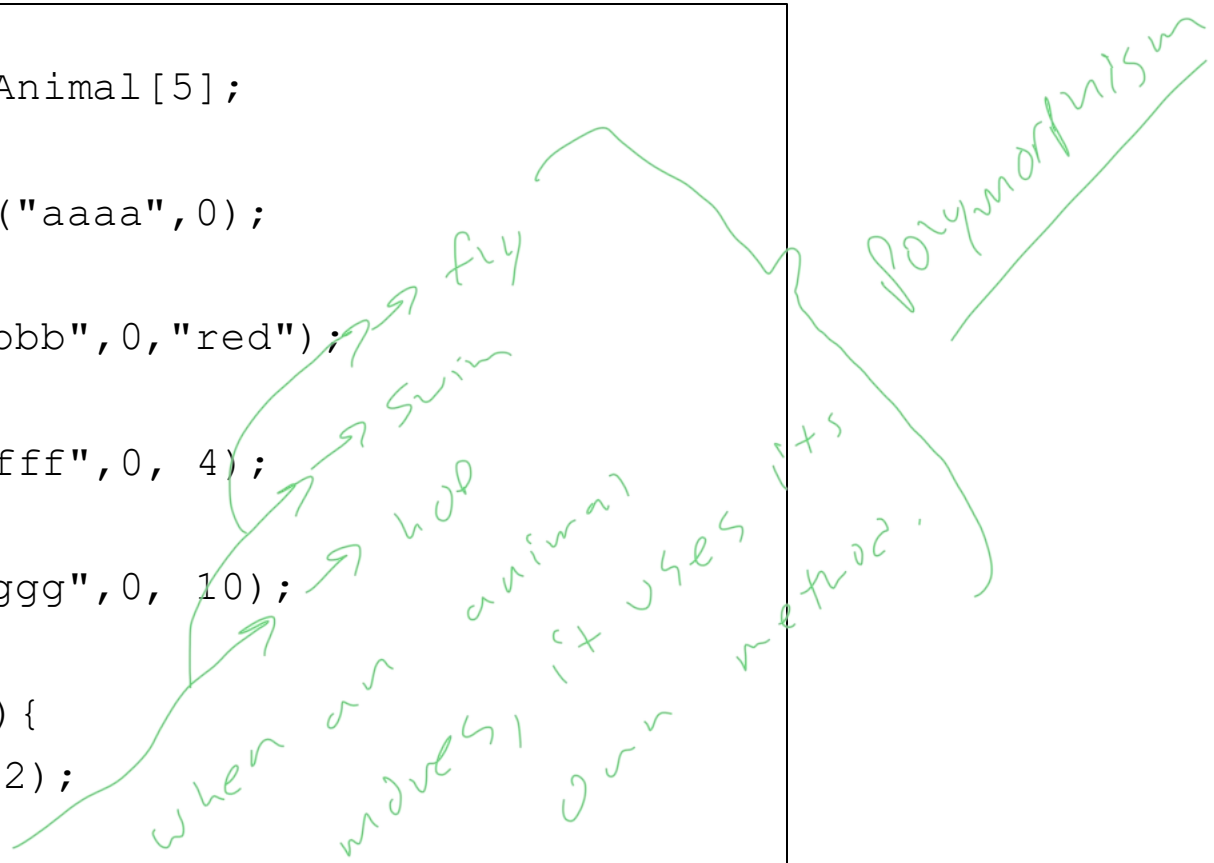






AnimalApplication Driver

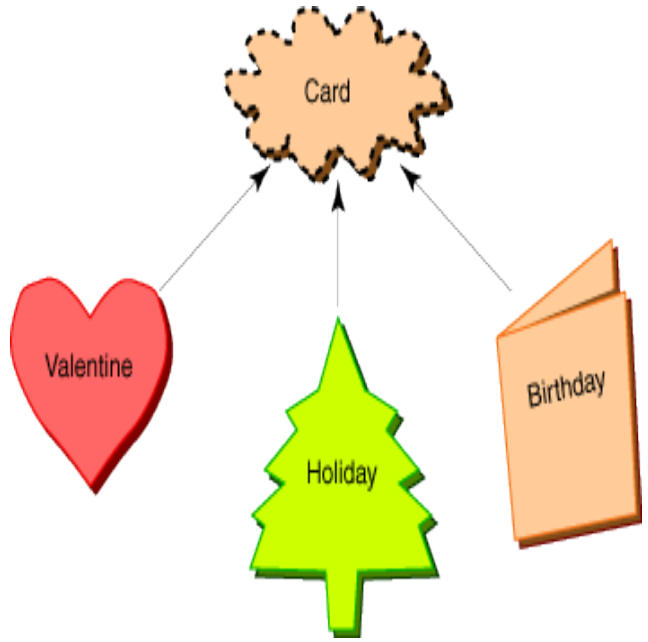
```
Animal[] myPets = new Animal[5];

myPets[0] = new Animal("aaaa",0);

myPets[1] = new Bird("bbb",0,"red");

myPets[2] = new Fish("fff",0, 4);

myPets[3] = new Frog("ggg",0, 10);

for (int i=0; i<4; i++){
        myPets[i].move(2);
```



Polymorphism

Example: `GreetingCard` Hierarchy

- Assume you want to implement a Java application that represents greeting cards where the parent class is `Card` and a card object will have a `greeting()` method that writes out a greeting.



- Assume further that there are three children classes as follows:
  - `Valentine`: prints "Love and Kisses."
  - `Holiday`: prints "Season's Greetings"
  - `Birthday`: prints "Happy Birthday."

Using Parent Class Reference Variables

- A reference variable of a parent class (e.g., `Card`) can be used for any object that inherits from that class (e.g., `Holiday` and `Birthday`)
- When a method is invoked, it is the class of the instantiated object (not of the type of the reference variable) that determines which method runs
- This concept is called **polymorphism**

Descendant

- A variable can hold a reference to an object whose class is a descendant of the class of the variable

```
Card myCard = new Holiday( "Maria" );
```

- A **descendant** of a class is a child of that class, or a child of a child of that class, and so on
- Siblings are not descendants of each other (you did not inherit anything from your brother)

Polymorphism

- **Polymorphism** means "having many forms".
- In Java, it means that one variable might be used with several objects of related classes at different times in a program **AND THE TYPE OF THE OBJECT DETERMINES WHICH VERSION OF THE METHOD WILL BE INVOKED**

```
public static void main ( String[] args )  {

    Card myCard = new Holiday( "Amy" );
    myCard.greeting();                          //invoke a Holiday greeting()

    myCard = new Valentine( "Bob", 3 );
    myCard.greeting();                          //Invoke a Valentine greeting()

    myCard = new Birthday( "Cindy", 17 );
    myCard.greeting();                          //Invoke a Birthday greeting()

  }
```

Why Polymorphism?

- Polymorphism enables you to write programs to process objects that share the same superclass as if they're all objects of the superclass

  "**Program in the general**" rather than "**Program in the specific**"

- Allows the design and implementation of systems that are easily **extensible**
- New classes can be added with little or no modification to the general portions of the program, as long as the new classes are part of the inheritance hierarchy the program processes generically
- The only parts of a program that must be altered to accommodate new classes are those that require direct knowledge of the new classes that are added to the hierarchy

Dynamic Binding

- Assume C_n is the most general class, and C_1 is the most specific class (in Java, C_n is the `Object` class).

- If `o` invokes a method `m`, the JVM searches the implementation for the method `m` in C_1, C_2, ..., C_{n-1} and C_n, in this order, until it is found.

- Once an implementation is found, the search stops and the first-found implementation is invoked.