Inheritance and Polymorphism

Stony Brook University

Cats and Dogs

Cat

- A mammal.
- Has fur.
- Common house pet.
- Has several breeds.

- Meows.
- Scratches the furniture.
- Catches mice.

Dog

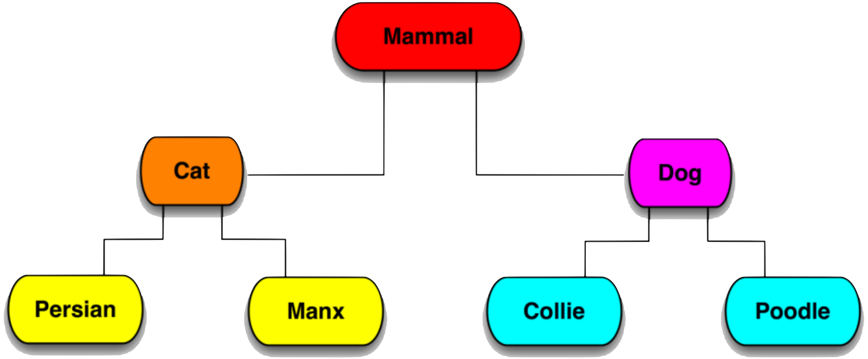
- Is a mammal.
- Has fur.
- Common house pet.
- Has several breeds.

- Barks.
- Catches Frisbees.
- Chases squirrels.
- Can guard a house.

Cats and Dogs cont..

- Some classes share properties that are similar.
- For example, cats and dogs are both mammals and common house pets.
- It doesn’t make sense to duplicate shared properties unnecessarily.
- Instead, we can abstract out common characteristics.

Cats and Dogs: Inheritance



Inheritance

- **Inheritance** A mechanism by which one class acquires (inherits) the properties (both data fields and methods) of another class.

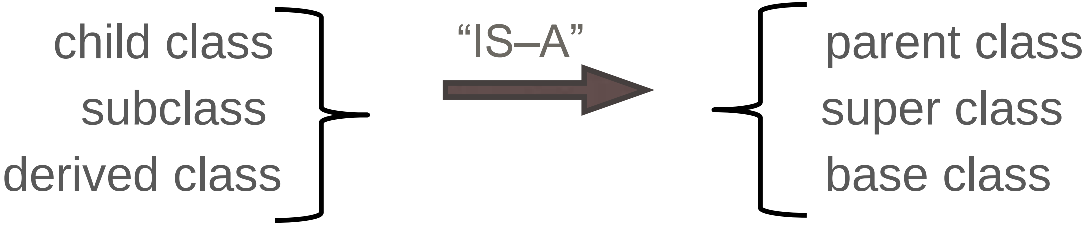
- **Superclass** The class being inherited from.

- **Derived class** The class that inherits.

- The derived class is **specialized** by adding properties specific to it.

Inheritance: Terminologies

- An object class that is derived from another class inherits its instance variables and methods.



child class

subclass

derived class

“IS–A”

parent class

super class

base class

- Any instance variable or method that is inherited by a subclass does not need to be redefined.

Why use Inheritance?

- Customize classes (from the JDK or your own).

- Benefits:
  - Don’t have to re-write code:
    - Use methods and variables from fully tested classes.
    - Superclass code can be used in any number of subclasses.
  - Changing common properties is easy – just change the parent class.

Inheritance Syntax

```
public class ChildClass extends ParentClass
{
 // instance variables for Child only
 // methods for Child only
}
```

- `ChildClass` now contains all instance variables and methods defined above, as well as those defined inside `ParentClass`

Sample Inheritance Code:

```
public class Mammal
{
   boolean hasFur = true;
   public void makeSound(){}
}
```

```
public class Cat extends Mammal
{
   boolean hasClaws;
   String furColor;
   public void makeSound()
   {
            System.out.println(“Meow!”);
   }
}
```

```
public class Persian extends Cat
{
  String furColor = “White”;
  public void makeSound()
  {
    System.out.println(“Hisss!”);
  }
}
```

- The `Persian` class *inherits* `hasClaws` from the `Cat` parent class. It ***overrides*** the `makeSound()` method, and ***hides*** the `furColor` field from `Cat`.

More Definitions

- **Override** When an **instance method** in a derived class has the same form of heading ***(signature)*** as an **instance method** in its superclass, the method in the derived class **overrides** (redefines) the method in the superclass.

- **Hide** When a data **field** in a derived class has the same name as one in its superclass or a **class method** has the same form of heading **(signature)** as a **class method** in its superclass, the data field or class method **hide** the corresponding component in the superclass.

Say again?

```
public class Example
{
  char letter;
  public static String lineIs();
  …
}


public class ExtExample extends Example
{
  char letter;
  public static String lineIs();
  …
}
```

Hiding or overriding?

```
public class Example
{
   char letter;
   public String lineIs();
   …
}


public class ExtExample extends Example
{
  String letter;
  public String lineIs();
  …
}
```

Hiding or overriding?

Organizing classes with Inheritance

- Determine what data you need to store. For example, for storing student and employee data:
    - Student data: name, age, **GPA**
    - Employee data: name, age, **salary**

- Divide up your classes by state – Students and Employees store different data, so make them into separate classes.

- Pool common data into a common parent class.
    - Person data: name, age

- Have Student and Employee customize Person.

Example: Parent Class: `Person`

```
public class Person{
        private String name;
        private int age;
        // constructor for a Person
        public Person(String initName){
          age = 0;    // just born
          name = initName;
        }
        // accessor method to get Person's name
        public String getName() { return name; }
        // accessor method to get Person's age
        public int getAge() { return age; }
        // mutator method to set person's age
        public void setAge(int newAge){
           if (newAge < 0)
               age = 0;
           else
               age = newAge;
         }
}
```

Example: Child (Sub-)Class: `Student`

```
public class Student extends Person{
   private double gpa;
   // constructor for a Student
   public Student(String initName){
      super(initName);
      gpa = 0.0;
   }
   public double getGpa() { return gpa; }
   public void setGpa(double newGpa){
      if (newGpa < 0.0 || newGpa > 4.0)
           gpa = 0.0;
      else
        gpa = newGpa;
   }
}
```

Means `Student` inherits all instance variables and methods from `Person`

Runs **`Person`**’s constructor

Example: Child (Sub-)Class: Employee

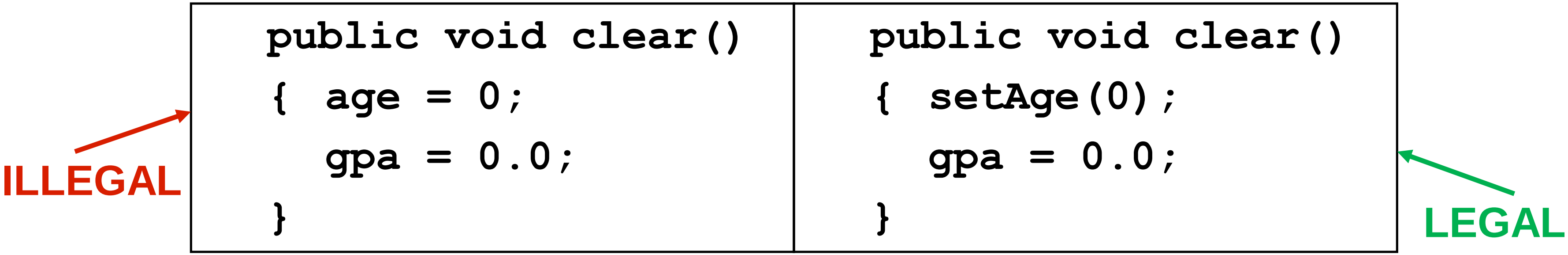
```
public class Employee extends Person{
  private int salary;
   // constructor for an Employee
  public Employee(String initName){
   super(initName);
   salary = 0;
  }
  public int getSalary() { return salary; }
  public void setSalary(int newSalary){
    if (newSalary < 0 || newSalary > 400000)
      salary = 0;
   else
      salary = newSalary;
  }
}
```

Example: Using all three classes

```
public class PeopleTester
{ public static void main(String[] args)
  {      Person moe = new Person("Moe Stooge");
         Student larry = new Student("Larry Stooge");
         Employee curly = new Employee("Curly Stooge");
         moe.setAge(106);               // LEGAL?    YES!
         moe.setGpa(2.2);               // LEGAL?    NO!
         moe.setSalary(100000);         // LEGAL?    NO!
         larry.setAge(101);             // LEGAL?    YES!
         larry.setGpa(1.2);             // LEGAL?    YES!
         larry.setSalary(50000);        // LEGAL?    NO!
         curly.setAge(100);             // LEGAL?    YES!
         curly.setGpa(0.5);             // LEGAL?    NO!
         curly.setSalary(25000);        // LEGAL?    YES!
  }
}
```

Inheritance: public vs. private

- Inherited methods are accessible by the derived class if they are **`public`** or **`protected`**.
- Inherited instance variables are **not directly** accessible by the derived class if they are private.
  - Use public accessor / mutator methods of parent class instead.
  - For example, if we added a clear method inside Student:



ILLEGAL

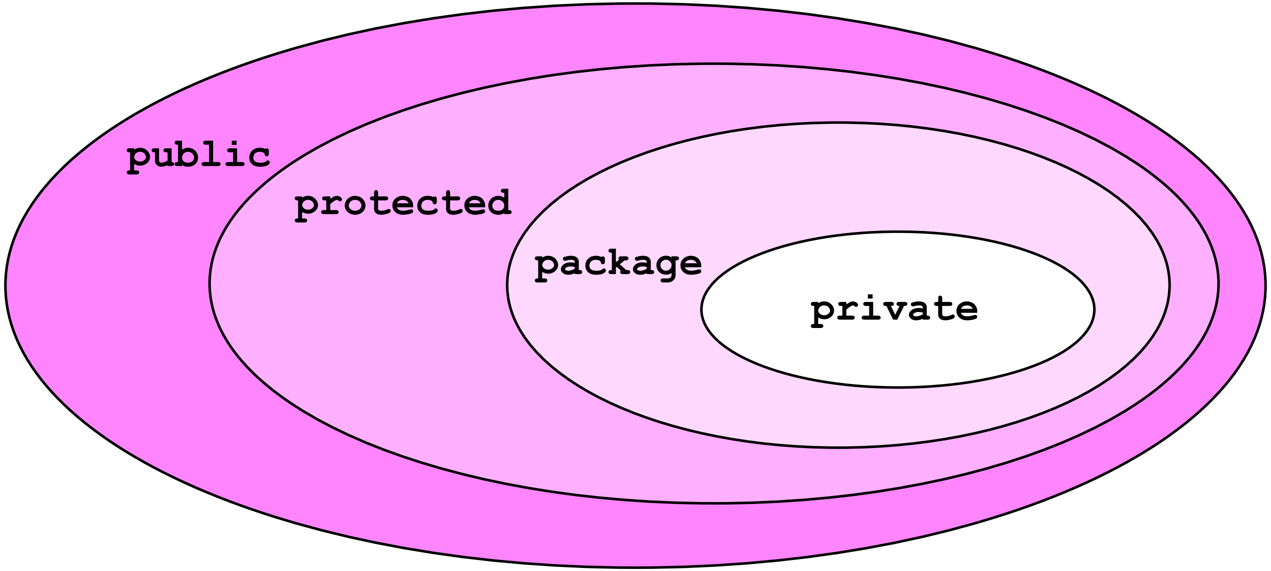
```
public void clear()
{  age = 0;
   gpa = 0.0;
}
```

LEGAL

```
public void clear()
{  setAge(0);
   gpa = 0.0;
}
```

- **`private`** methods of a base class are not accessible by the derived class.

Four Levels of Class Member Access



Member Accessibility

External Access	public	protected	Default (package private)	private
Same package	Yes	Yes	Yes	No
Derived class in another package	Yes	Yes (inheritance only)	No	No
User code	Yes	No	No	No

Inheritance: `super()` and `this()`

- `super()` runs base (parent) class' constructor.
  - Must be first statement in a derived class' constructor.
  - If it is left out, `super()` is still executed using the base class' default constructor (with no parameters).
- `this()` runs a class' own constructor.
  - Used on first line of another constructor.
  - May pass parameters to this() as long as they correspond to parameters for another constructor.
  - For example, if we added a new constructor for Student:

```
public Student(String initName, double initGpa)
  {
       this(initName);
        gpa = initGpa;
  }
```

Runs the previously defined `Student` constructor

`this` Keyword

- **`this`** keyword is the name of a reference that refers to an object itself.

- Common uses of **`this`** keyword:

1. Reference a class’s “*hidden*” *data fields.*
2. To enable a constructor to invoke another constructor of the same class as the first statement in the constructor.

Reference the Hidden Data Fields

```
public class Foo {
  private int i = 5;
  private static double k = 0;

  void setI(int i) {
    this.i = i;
  }

  static void setK(double k) {
    Foo.k = k;
  }
}
```

```
Suppose that f1 and f2 are two objects of Foo.

Invoking f1.setI(10) is to execute
   this.i = 10, where this refers f1

Invoking f2.setI(45) is to execute
   this.i = 45, where this refers f2
```

Calling Overloaded Constructor

```
public class Circle {
  private double radius;

  public Circle(double radius) {
    this.radius = radius;
  }

  public Circle() {
    this(1.0);
  }

  public double getArea() {
    return this.radius * this.radius * Math.PI;
  }
}
```

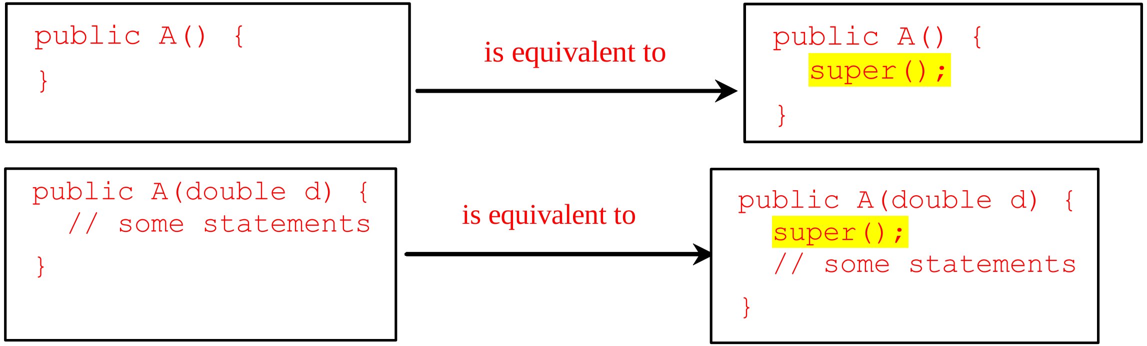
this must be explicitly used to reference the data field radius of the object being constructed

this is used to invoke another constructor

Every instance variable belongs to an instance represented by this, which is normally omitted

Are superclass’s Constructor Inherited?

- **No. They are not inherited.**
- They are invoked explicitly or implicitly:
    - Explicitly using the **`super`** keyword
    - If the keyword **`super`** is not explicitly used, the superclass's no-arg constructor is automatically invoked as the first statement in the constructor, unless another constructor is invoked (when the last constructor in the chain will invoke the superclass constructor)



```
public A()  {
}
```

is equivalent to

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

```
public A(double d)  {
   // some statements
}
```

is equivalent to

```
public A(double d)  {
   super();
   // some statements
}
```

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: Java requires that the statement that uses the keyword **`super`** appear first in the constructor (unless another constructor is called or the superclass constructor is called implicitly) to call a superclass method

Constructor Chaining

- ***Constructor chaining*** **: constructing an instance of a class invokes all the superclasses’ constructors along the inheritance chain.**

```
public class Faculty extends Employee {
  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
  public static void main(String[] args) {
    new Faculty();
  }
}
class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) 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");
  }
}
```

Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) 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");
  }
}
```

1. Start from the main method

Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) 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");
  }
}
```

2. Invoke Faculty constructor

Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) 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");
  }
}
```

3. Invoke Employee’s no-arg constructor

Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) 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");
  }
}
```

4. Invoke Employee(String) constructor

Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) 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");
  }
}
```

5. Invoke Person() constructor

Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) 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");
  }
}
```

6. Execute println

Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) 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");
  }
}
```

7. Execute println

Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) 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");
  }
}
```

8. Execute println

Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) 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");
  }
}
```

9. Execute println

Calling Superclass Methods

```
public void printCircle() {
  System.out.println(
    "The circle is created " +
    super.getDateCreated() +
    " and the radius is " +
    radius);
}
```

Stony Brook
University