- **Inheritance**
- **Method Overloading**
- **ArrayList**
- **Wrapper Classes**

Inheritance

- **What is inheritance with respect to classes ?**
- **Visibility modifiers (*protected* modifier)**
- **Inheritance and *Object* class**
- **Shadowing Variables**
- **Overriding methods**
- **Polymorphism**

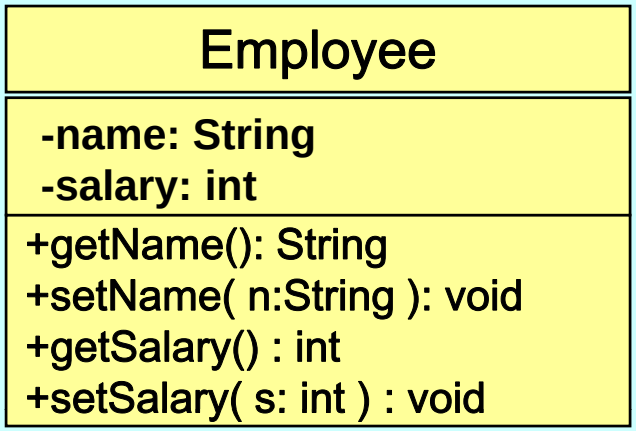
Motivation for Inheritance

Suppose you need to write class declarations for the following entities:

- **Employee**
- **Student**

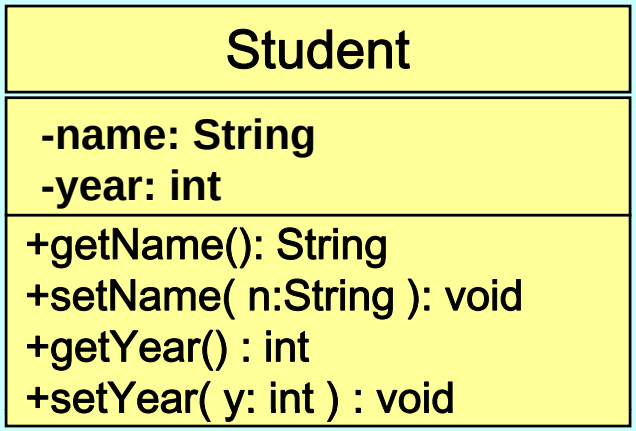
Class Declaration for Employee

```
public class Employee {
    private String name;
    private int salary;

    public int getName() {…}
    public void setName(String n) {…}

    public int getSalary() {…}
    public void setSalary(int s) {…}
}
```

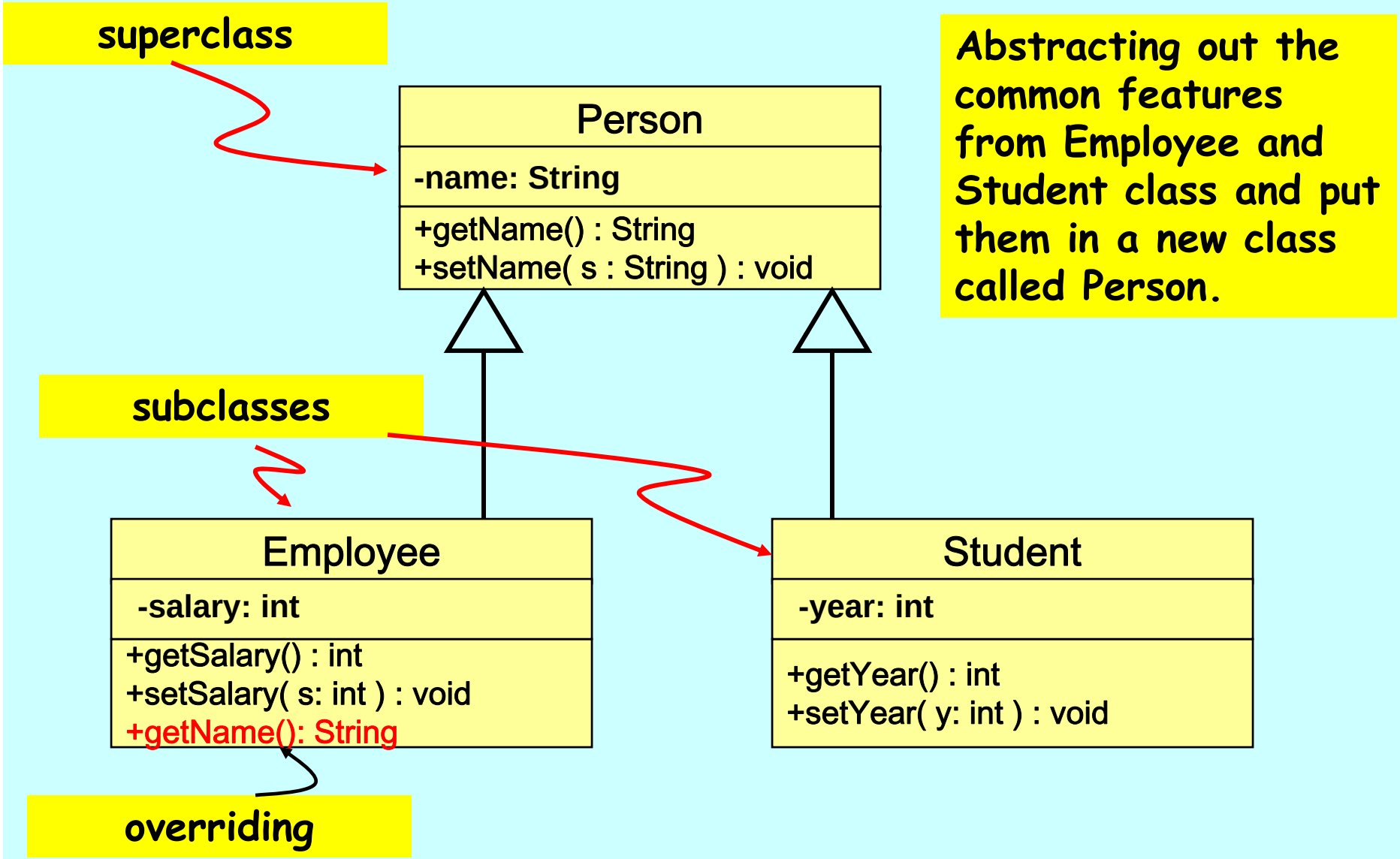


Class Declaration for Student

```
public class Student {
    private String name;
    private int year;

    public int getName() {…}
    public void setName(String n) {…}

    public int getYear() {…}
    public void setYear(int y) {…}
}
```



Creating Employee and Student classes from Person



What is inheritance with respect to classes ?

Creating new classes from existing classes by

- **inheriting all the instance variables and instance methods of the existing class**
- **adding additional state and behavior in the new class**

Supports the idea of software reuse

Creates an "is-a" relationship between classes

Class Declaration for Employee

subclass OR
child class

superclass OR
parent class OR
base class

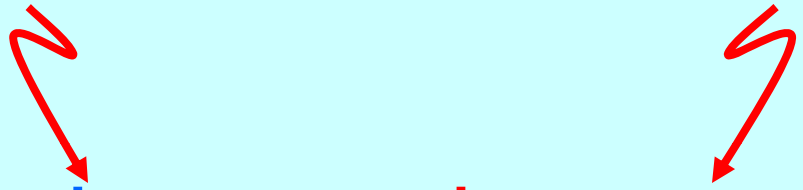
```
public class Employee extends Person {
    private int salary;
    public int getSalary() {…}
    public void setSalary(int s) {…}
    public String getName() {…}
}
```

inherits all instance variables and methods of Person and overrides **getName()**

Class Declaration for Student

subclass OR
child class

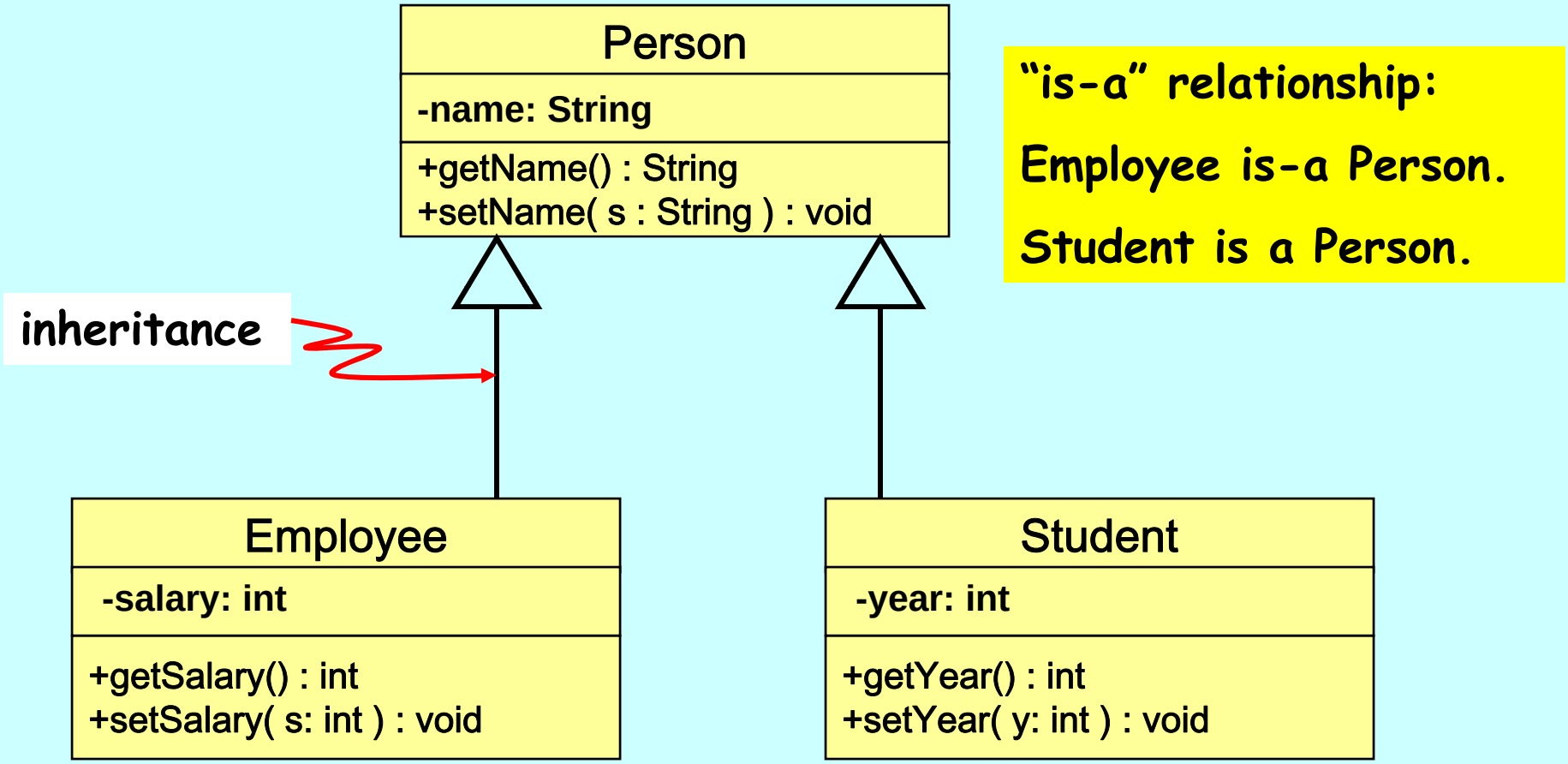
superclass OR
parent class OR
base class



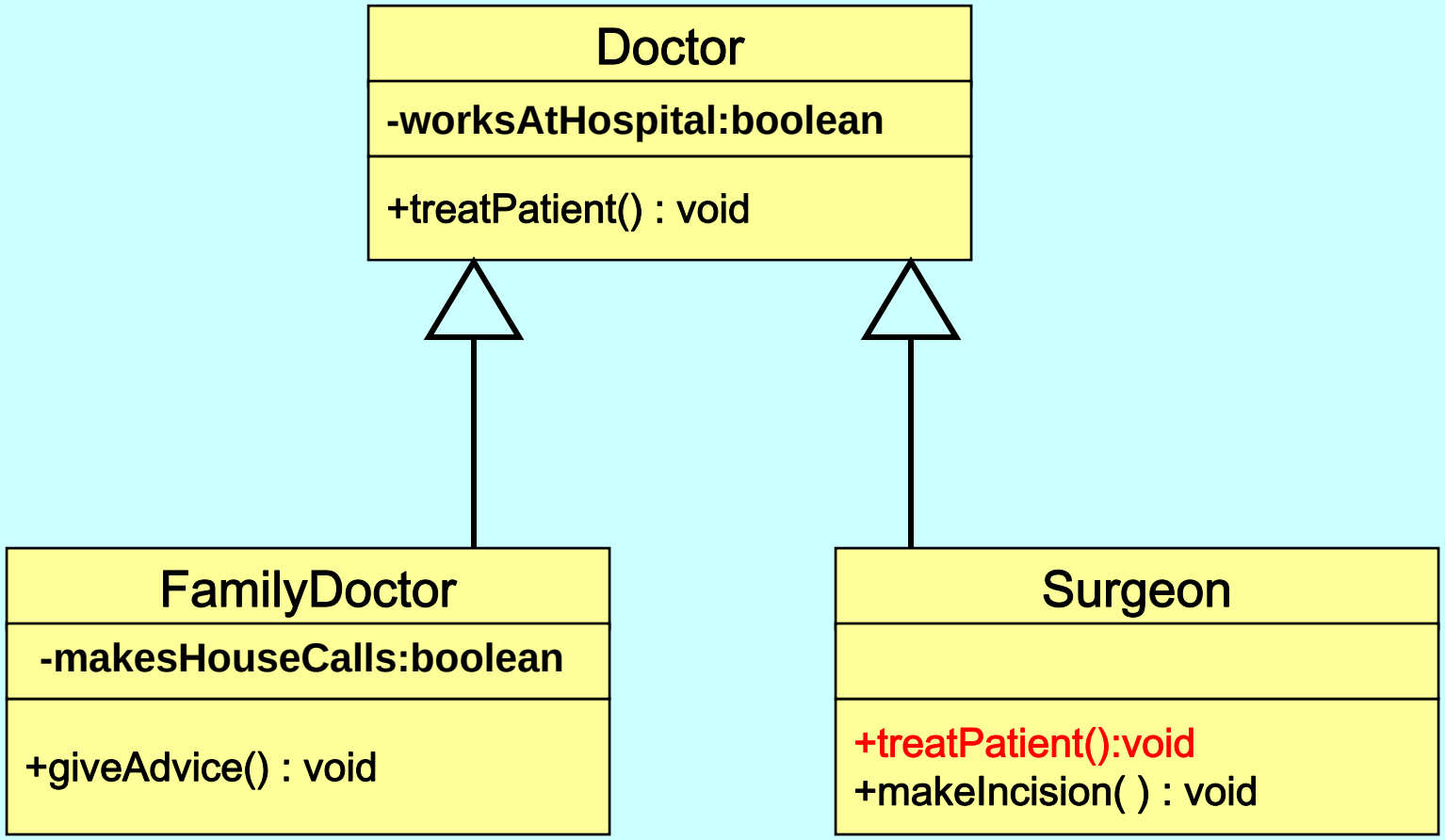
```
public class Student extends Person {
    private int year;

    public int getYear() {…}
    public void setYear(int y) {…}
}
```

inherits all instance variables and methods of Person

UML Notation for Inheritance



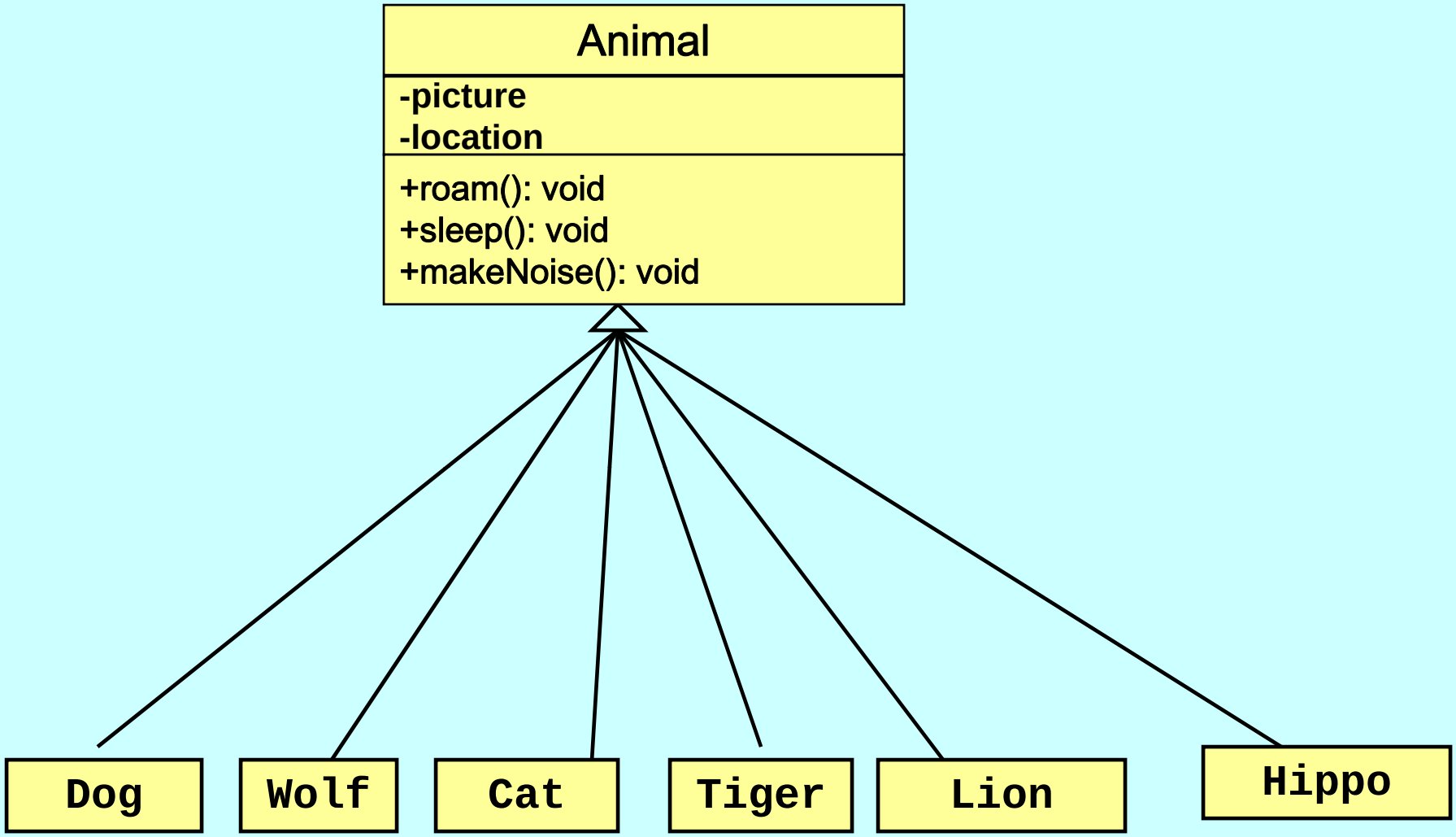
Example-1


Doctor
-worksAtHospital:boolean
+treatPatient() : void
FamilyDoctor
-makesHouseCalls:boolean
+giveAdvice() : void
Surgeon
+treatPatient():void
+makeIncision( ) : void

Design Inheritance Tree: Example-2

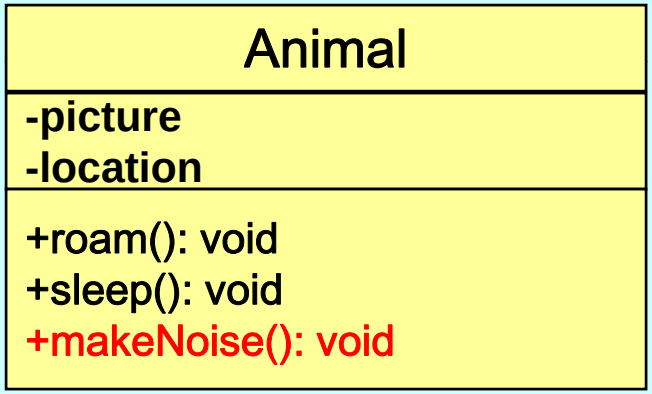
- Dog
- Wolf
- Cat
- Tiger
- Lion
- Hippo

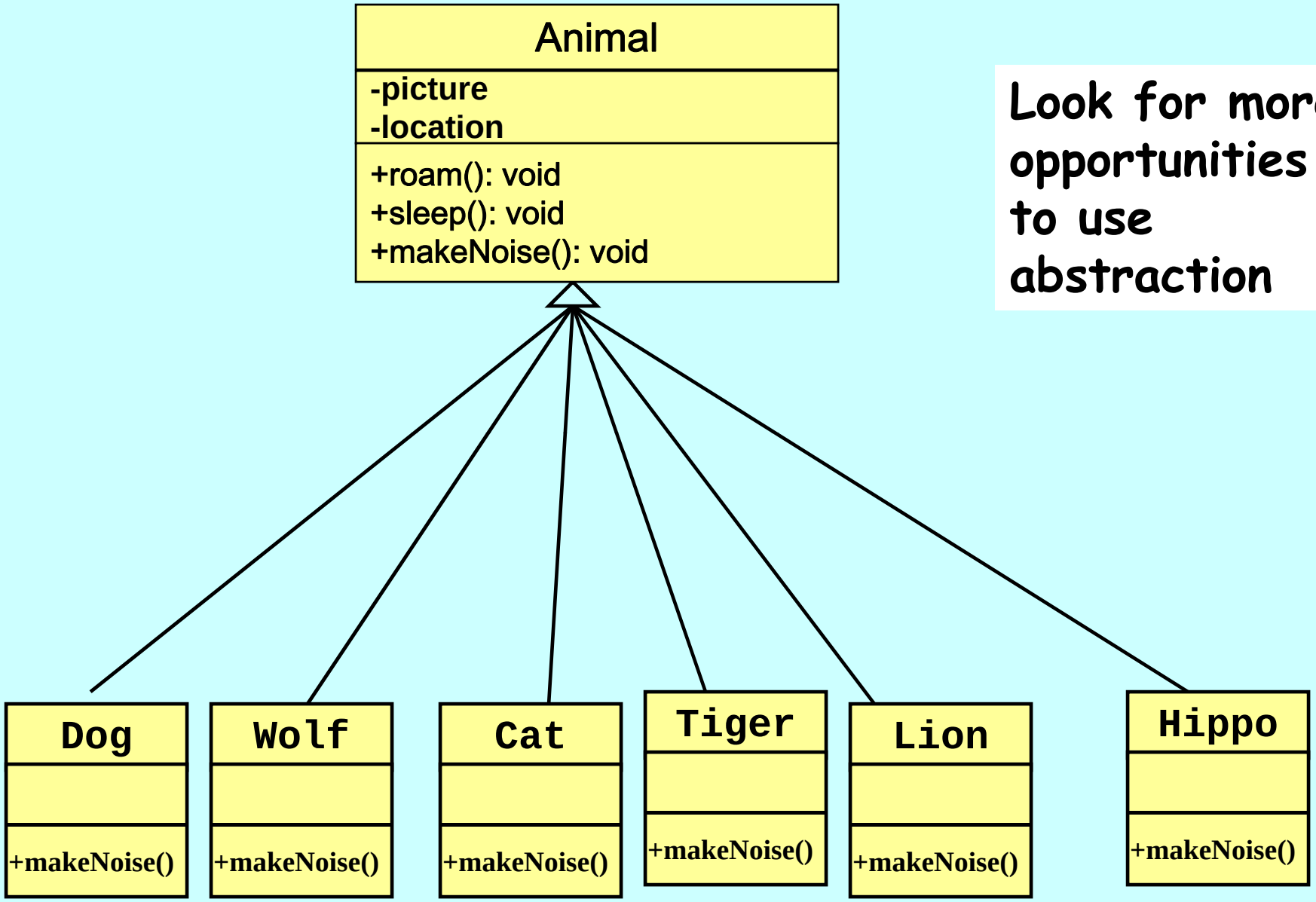
Look for objects that have common attributes and behaviors

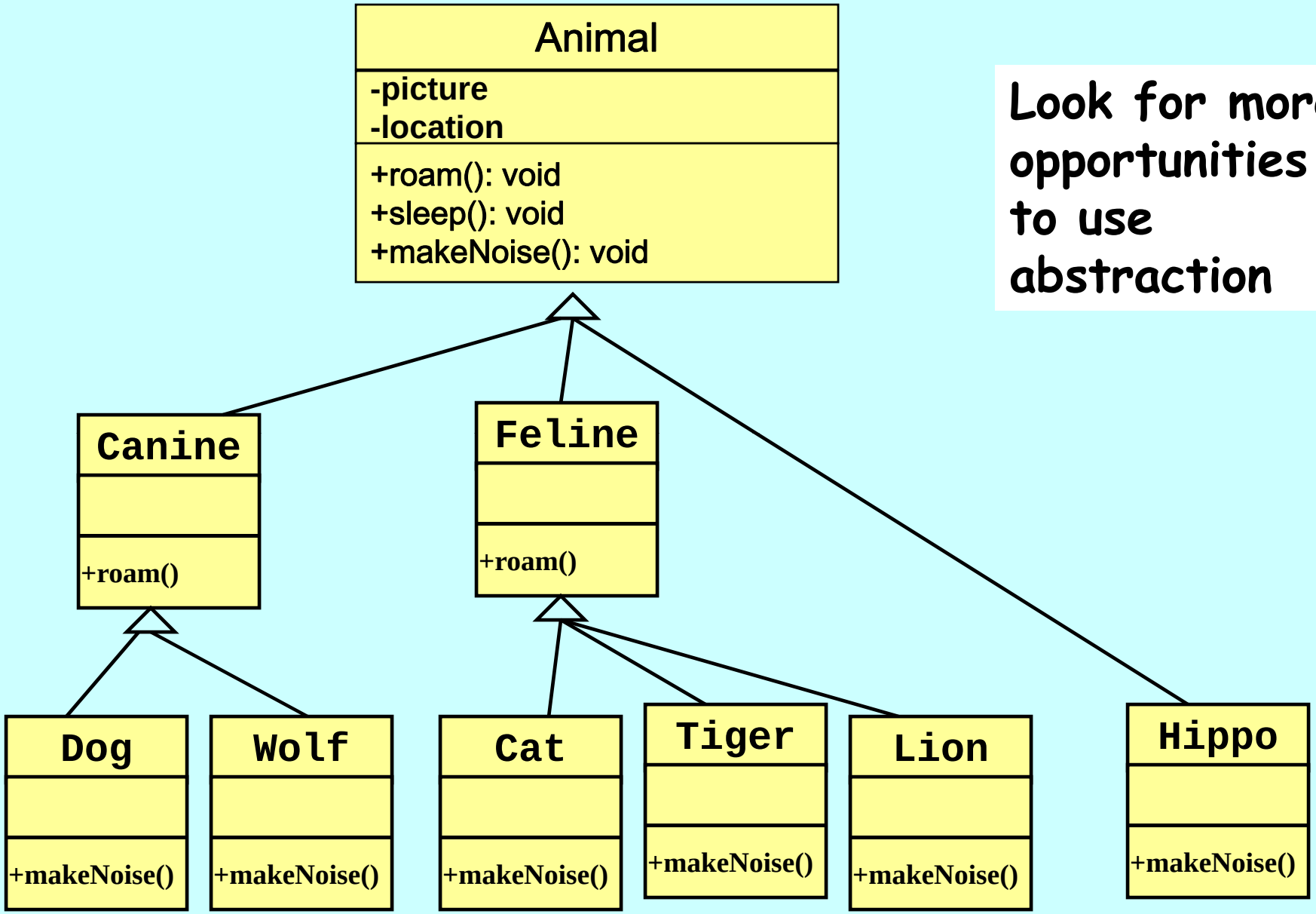


Design a class that represents the common state and behavior

Decide if a subclass needs behaviors that are specific to that particular class type.



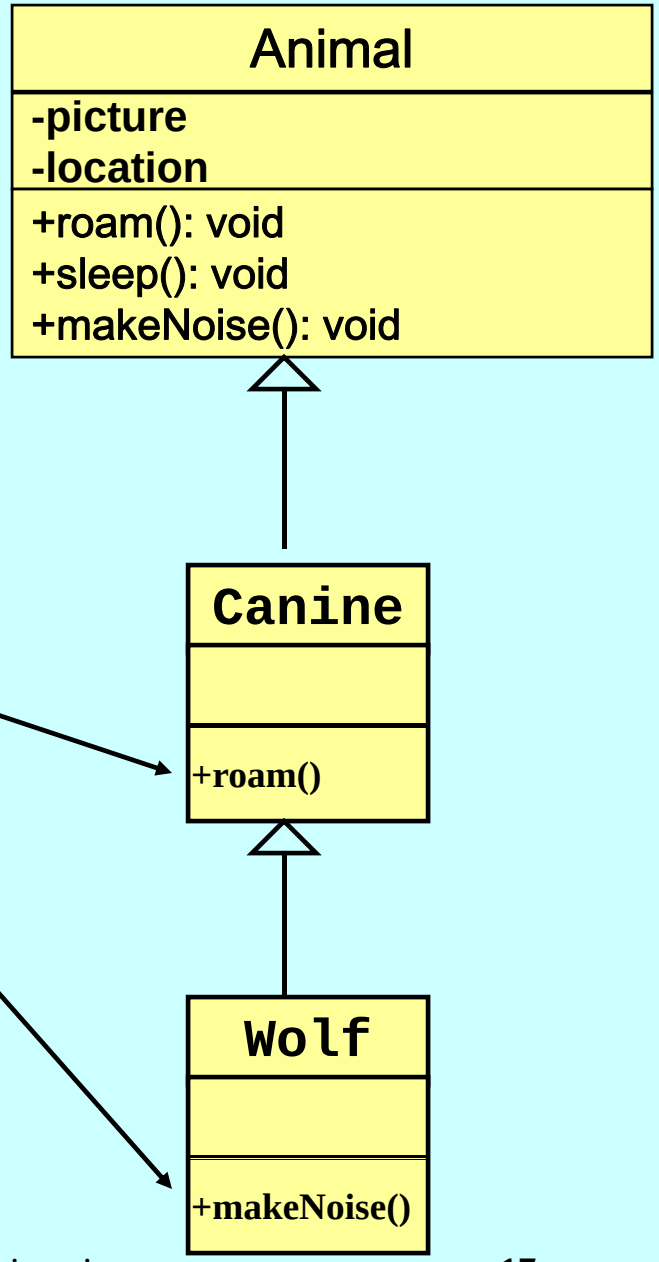

Animal
-picture
-location
+roam(): void
+sleep(): void
+makeNoise(): void
Look for more opportunities to use abstraction
Dog
+makeNoise()
Wolf
+makeNoise()
Cat
+makeNoise()
Tiger
+makeNoise()
Lion
+makeNoise()
Hippo
+makeNoise()



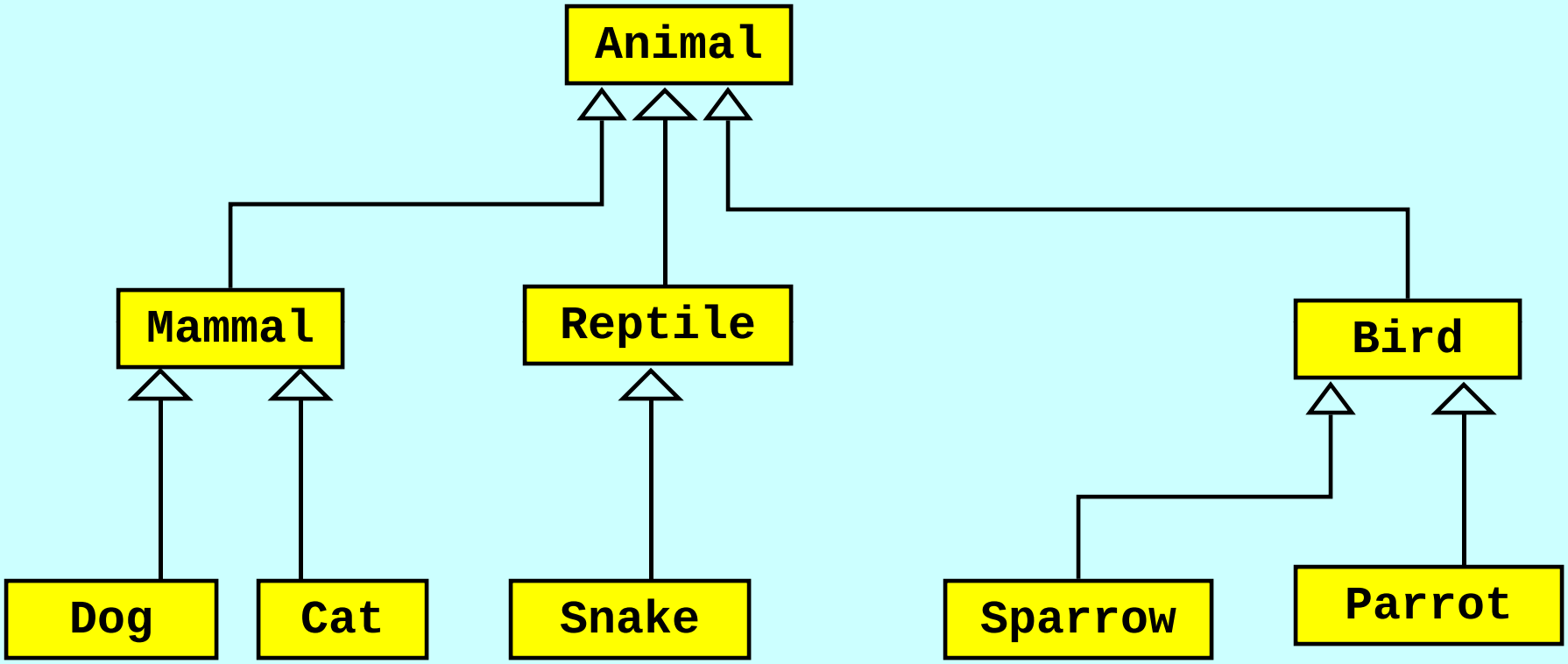
Look for more opportunities to use abstraction

Which method is called..

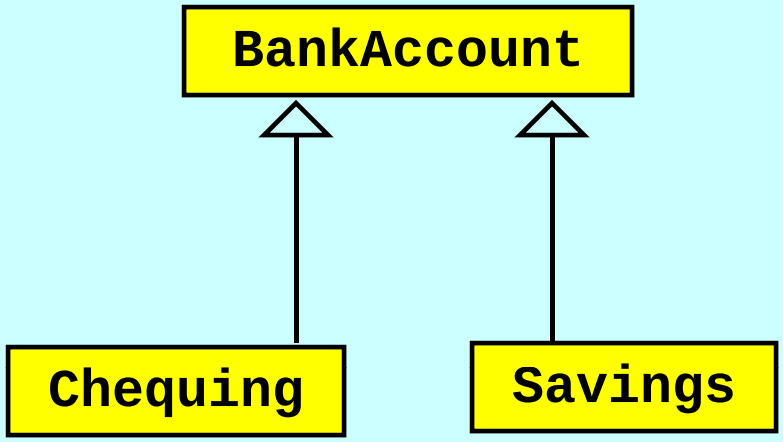
```
Wolf w = new Wolf();
w.makeNoise();
w.roam();
w.sleep();
```



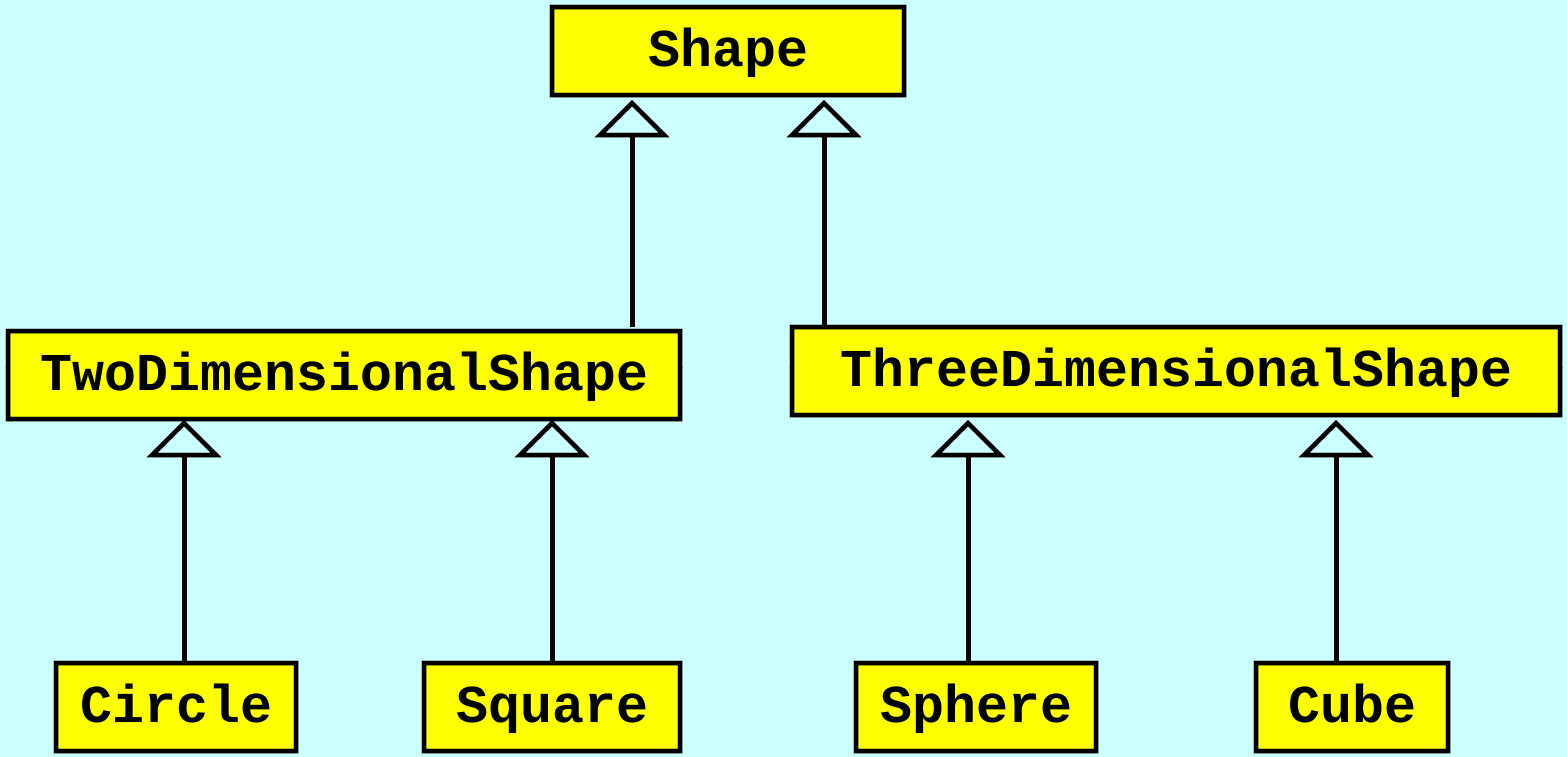
Other Inheritance Examples



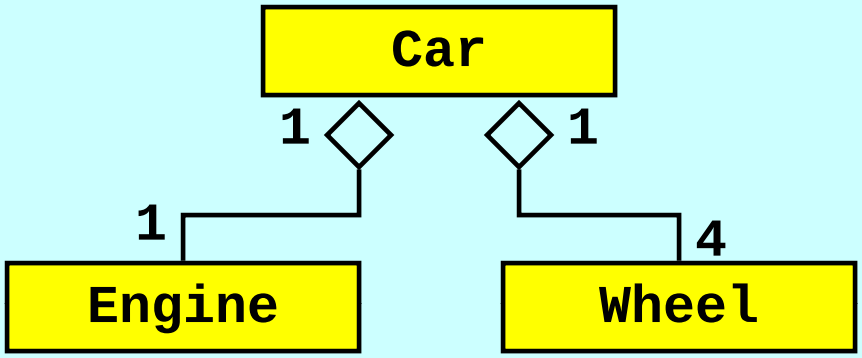
Inheritance Examples

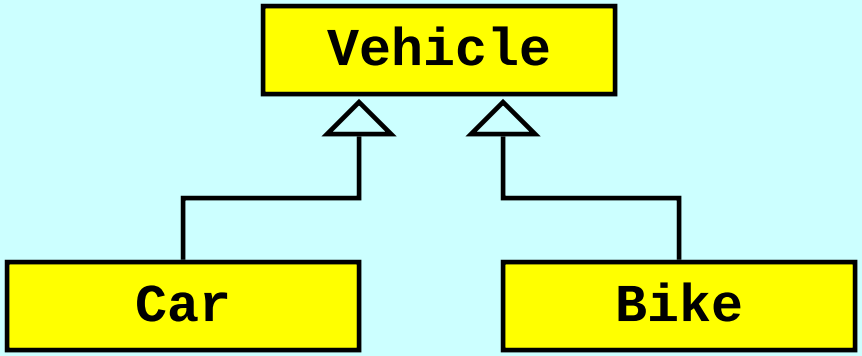


Inheritance Examples



Be careful about “is-a” and “has-a” relationships while designing classes



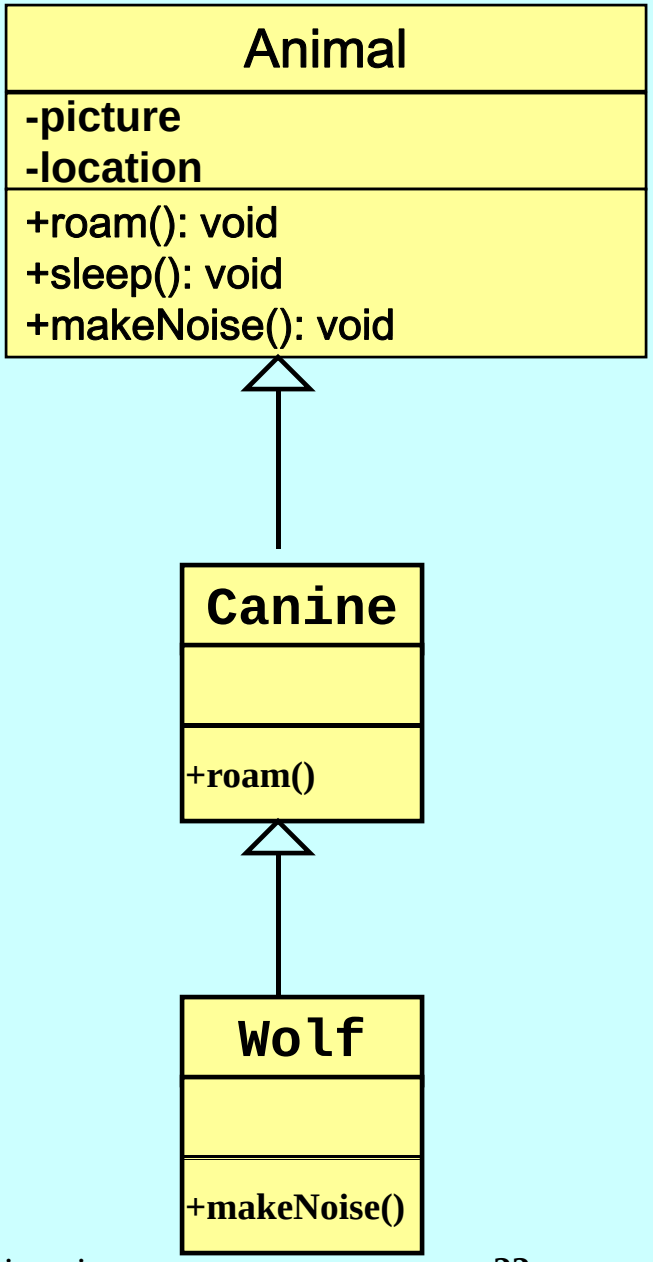


Canine is-a Animal

Wolf is-a Canine

Wolf is-a Animal

A Wolf can roam, sleep and makeNoise.

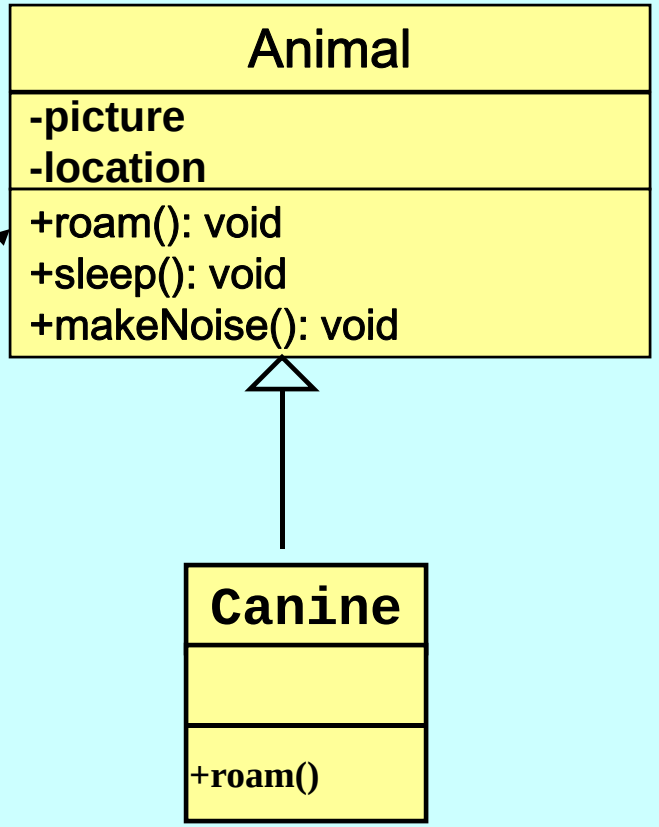


Inheritance relationship works in only one direction

~~Animal is-a Wolf~~

What if in roam method of Canine, I do not want to completely replace the superclass version, I just want to add more stuff..

```
public class Canine {
    …
    public void roam() {
        super.roam();
        // subclass specific code
    }
}
```



Salient features of Inheritance

When we extend class A from class B,

- **class A inherits the state (variables) and behaviors (methods) of class B**
- **we can add new variables to class A**
- **we can add new methods to class A**
- **we can override methods of class B in class A**

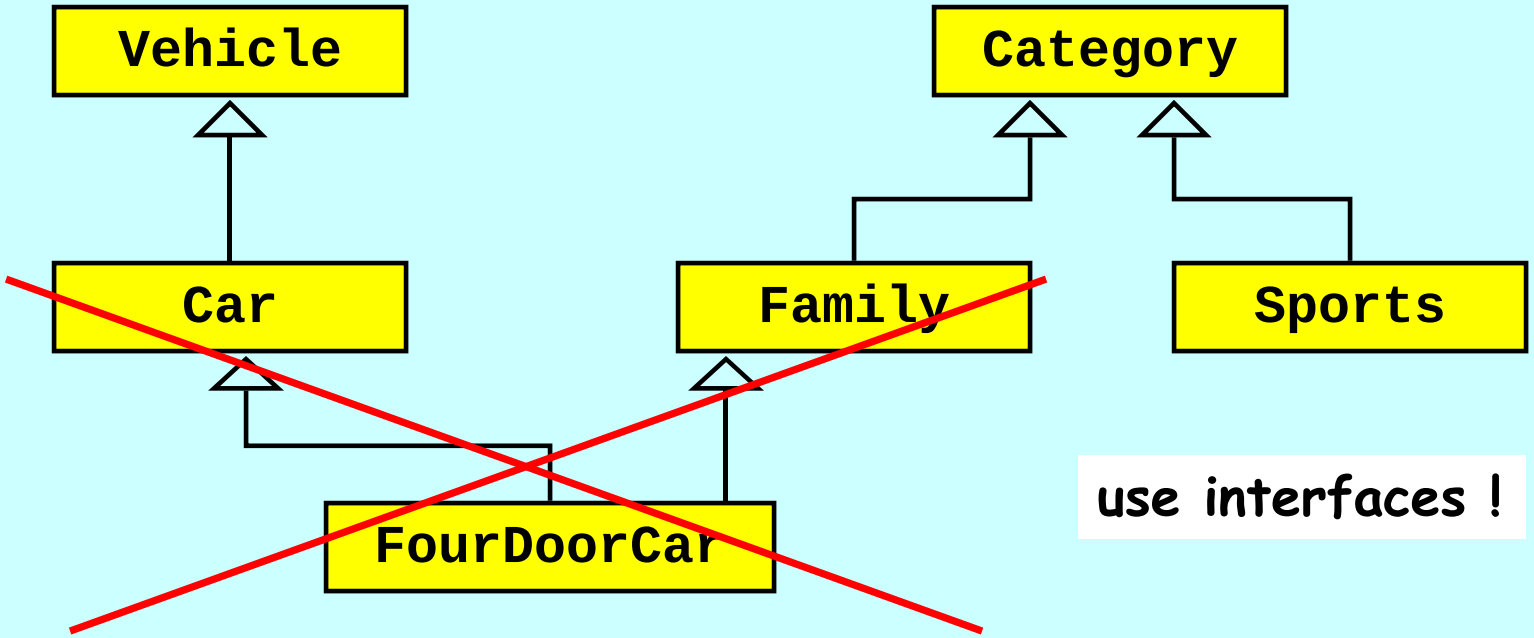
Class Hierarchies

In Java:

- **a class can have only one super-class**
- **a class can have any number of sub-classes**
- **two classes inherited from same parent – *Siblings***
- **inheritance is *transitive***

Class Hierarchies

- In some O-O languages, such as C++, multiple inheritance is permitted.
- In Java, only single inheritance supported



Access Modifiers

- private
- public
- default
- protected: can be accessed in the subclasses even if the subclasses are outside the package of the superclass

Class Declaration for Vehicle

```
public class Vehicle {
    private int vin;

    public Vehicle() { vin = 0; }
    public int getVIN() {…}
    public void setVIN(int v) {…}
}
```

Class Declaration for Car

```
public class Car extends Vehicle {
    private int mileage;

    public Car() { mileage = 0; }
    public int getMileage() {return mileage;}
    public void setMileage(int m) {mileage = m;}

    public String toString() {
        return "VIN: " + vin + " Mileage: " + mileage;}
    }
}
```

compilation error: although Car inherits from Vehicle, it cannot directly access the private members of Vehicle

Protected Members

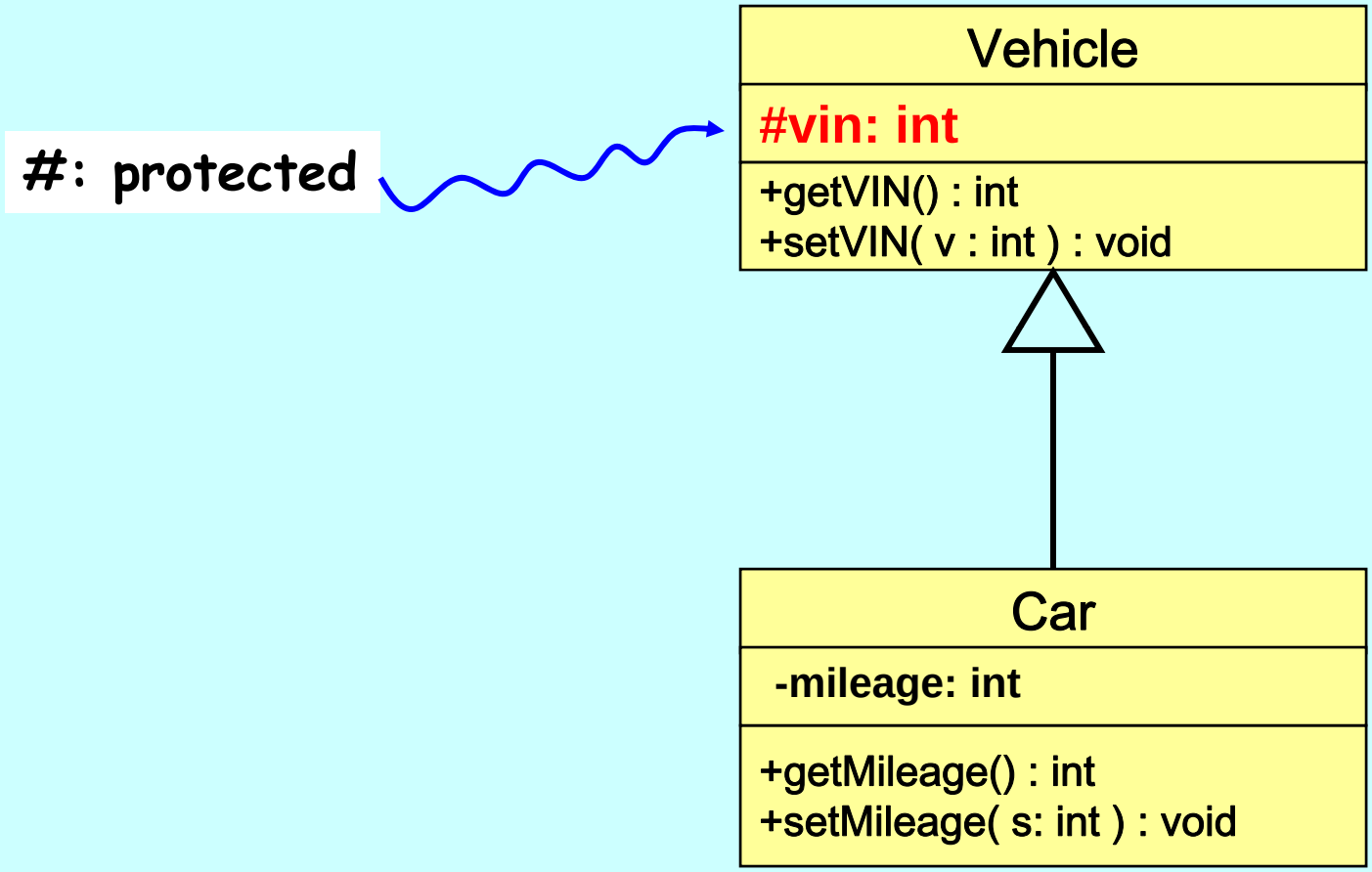
```
public class Vehicle {
    protected int vin;

    public Vehicle() { vin = 0; }
    public int getVIN() {…}
    public void setVIN(int v) {…}
}
```

Accessing Protected Members

```
public class Car extends Vehicle {
    private int mileage;

    public Car() { mileage = 0; }
    public int getMileage() {return mileage;}
    public void setMileage(int m) {mileage = m;}

    public String toString() {
        return "VIN: " + vin + " Mileage: " + mileage;}
    }
}
```

no compilation error since vin is a protected member

UML for Protected Members



When designing with inheritance,

- When one class is more specific type of a superclass
- When you have behavior that should be shared among multiple classes of the same general type
- is-a relationship holds

Inheritance let you guarantee that all classes grouped under a certain supertype have all the methods that the supertype has..

Inheritance and the *Object* class

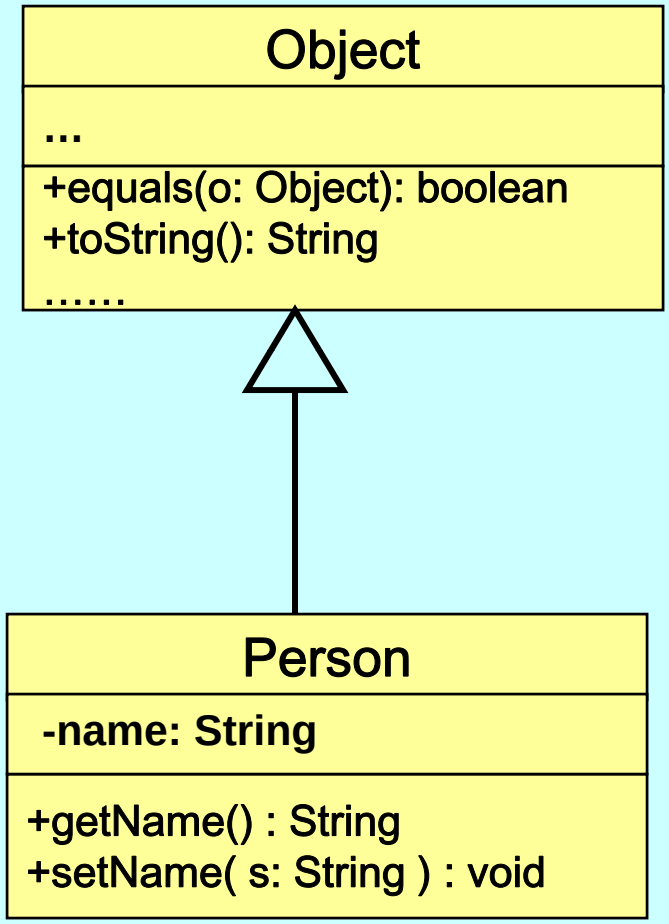
In Java:

- **all classes are derived from Object class**

- **If a class definition does not contain an extends clause, then that class is automatically derived from the Object class**

- **this means all classes in Java are subclasses of class Object**

Every class extends Object class (by default)




```
public class Person
{
}
```

is equivalent to:

```
public class Person
   extends Object
{
}
```

Inheritance and the *Object* class

- **All public methods of class Object are inherited by every Java class.**

- **Object class defined in java.lang package**

- **Some public methods of Object class:**

```
boolean equals( Object obj )
String toString()
Object  clone()
```

Shadowing Variables: Not recommended

- **possible for a child class to declare a variable with the same name as one that is inherited from the parent.**

- **if a variable of same name is declared in child class, it is called *shadow variable*.**

- **difference between redeclaring a variable and giving an inhertited variable a particular value**

Overriding Methods

- **When a class defines a method with the same name and signature as a method in the parent class, we say that child's method overrides the parent's method.**

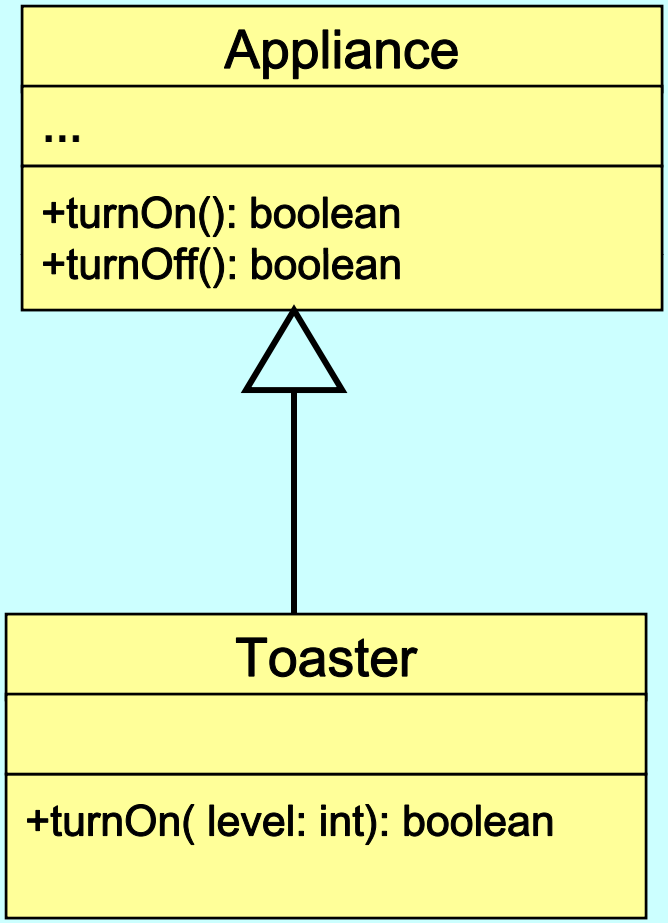
- **The object that is used to invoke a method determines which version of the method is actually executed.**

- **if a method is *final* in parent, child cannot override it.**

Rules for Overriding Methods

- **Arguments must be the same and return type must be compatible**

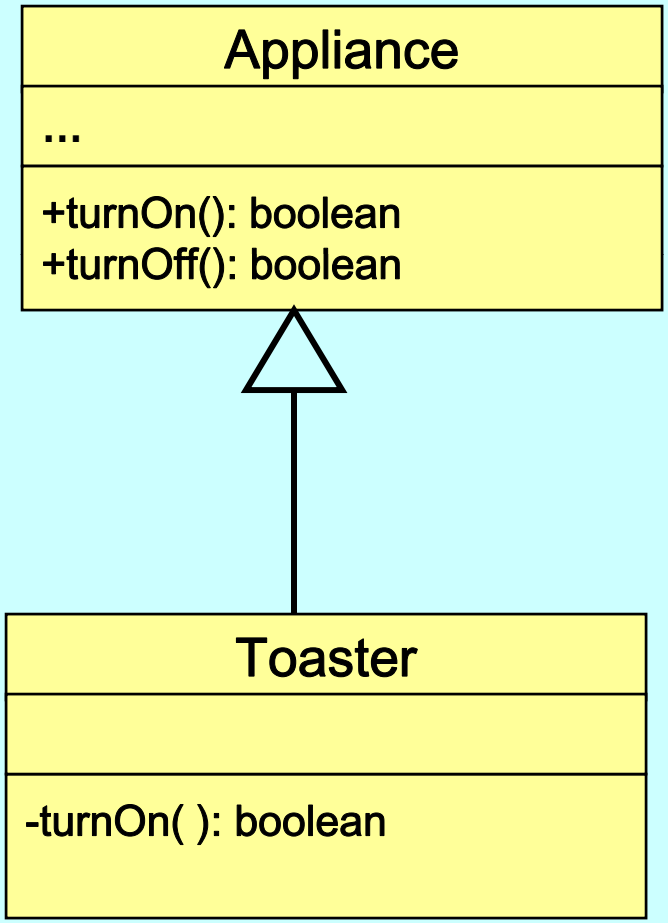
turnOn() overriden ??



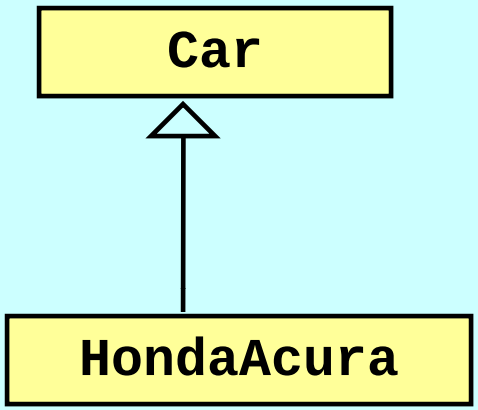
Rules for Overriding Methods

- **The method cannot be less accessible**

turnOn() overriden ??



Sample design (using UML)



HondaAcura is a kind of Car

"is-a" relationship

Overriding Methods: Example

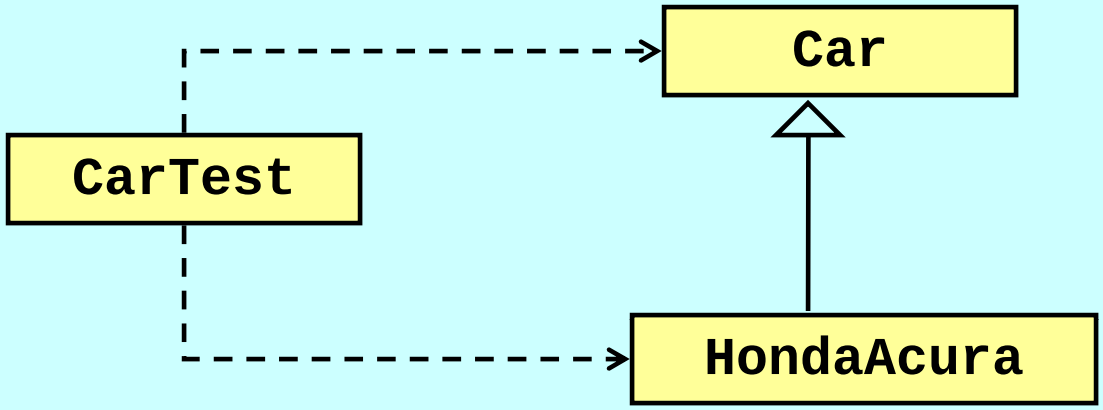
```
public class Car {

    //let default color of any car be blue
    public void color() {
        System.out.println( "Blue" );
    }
    public final int numberOfWheels() { return 4; }
}
```

subclass of Car (if any) won't be able to override the numberOfWheels method ! WHY ??

Overriding Methods: Example

```
public class HondaAcura extends Car {

    //let color of 1999 HondaAcura be red
    public void color() {
       System.out.println("Red");
    }
}
```

Class diagram (with test driver CarTest)



CarTest class uses Car and HondaAcura classes

HondaAcura is a kind of Car

"uses" relationship ------->

"is-a" relationship

Overriding Methods: Example

```
public class CarTest {

   public static void main( String[] args ) {
      Car c = new Car();
      HondaAcura h = new HondaAcura();

      c.color();
      h.color();
   }
}
```

Output ??

Use of keyword super to invoke parent's method

```
public class Car {

    //let default color of any car be blue
    public void color() {
        System.out.println( "Blue" );
    }
}
```

Use of keyword super to invoke parent's method (contd.)

```
public class HondaAcura extends Car {
    private int year;
    public int  getYear() { return year; }
    public void setYear(int y) { year = y; }

    public void color() {
       if( getYear() == 1999 )
          System.out.println("Red");
       else
          super.color();
    }
}
```

Use of keyword super to invoke parent's method (contd.)

```
public class SuperTest {
    public static void main( String[] args ) {
        HondaAcura h = new HondaAcura();

        h.setYear( 2000 );
        h.color();
  }
}
```

Output ??

Overriding Methods: equals and hashCode

- **The methods equals and hashcode are inherited from Object class**
- **If you override equals for a class, you must override hashcode for that class !**
- **If two objects are equal according to the equals method, then calling the hashcode method on each of those two objects must produce the same integer result !**

Overriding Methods: equals and hashCode

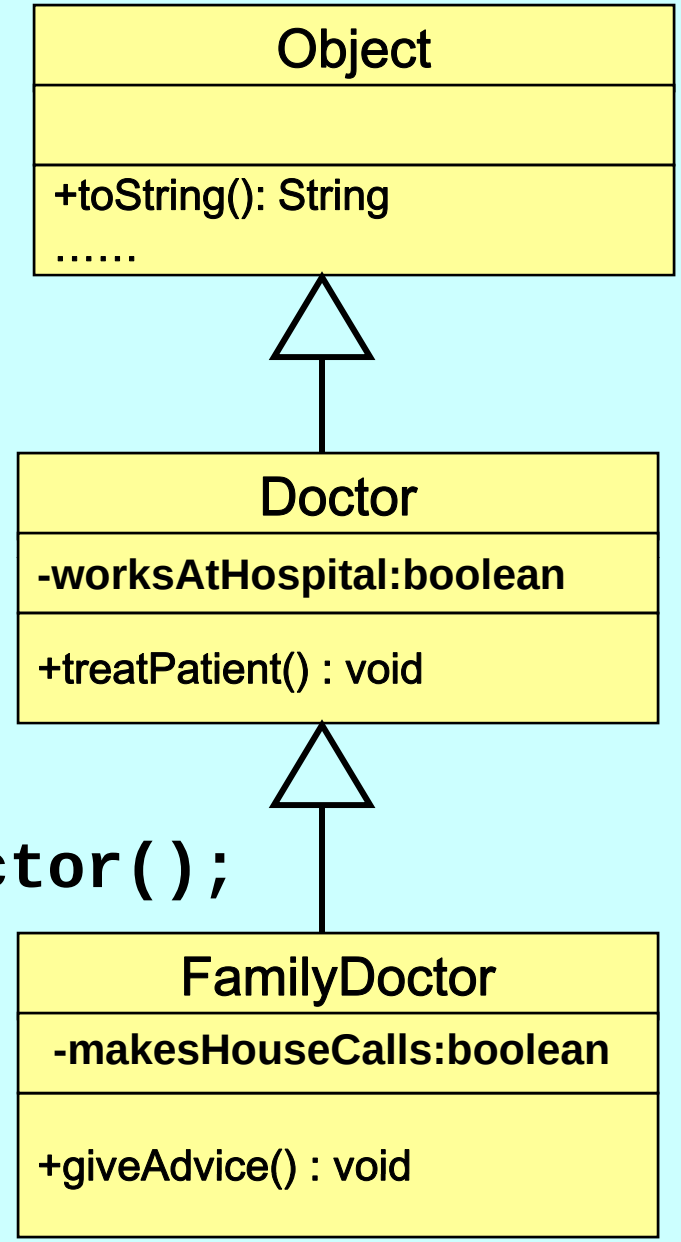
```
public class MyInteger {
    private int value;
    public MyInteger( int v ) { value = v; }
    public int intValue() { return value; }

    public boolean equals( Object o ) {
      if( this == o ) return true;
      if( o instanceof MyInteger ) {
        MyInteger other = ( MyInteger )o;
        return ( value == other.intValue() );
      }
      return false;
    }
}
```

Overriding Methods: equals and hashCode (contd.)

```
public class MyInteger {
      ..........
      ..........
      ..........

      public int hashCode() { return value; }
}
```

The MyInteger class discussed so far is motivated by the code in Integer.java file in JDK source code.

Constructor Chaining



Object()

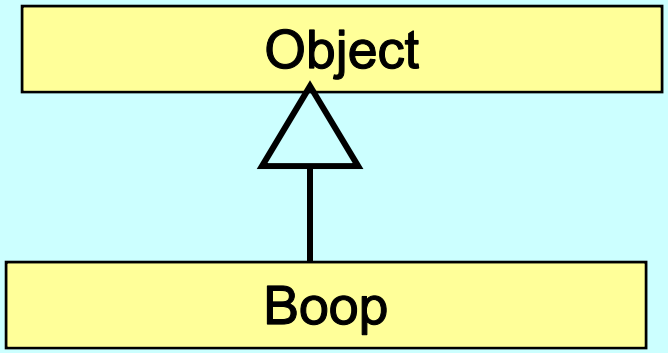
Doctor()

FamilyDoctor()

```
FamilyDoctor fd = new FamilyDoctor();
```

Possible constructors for class Boop

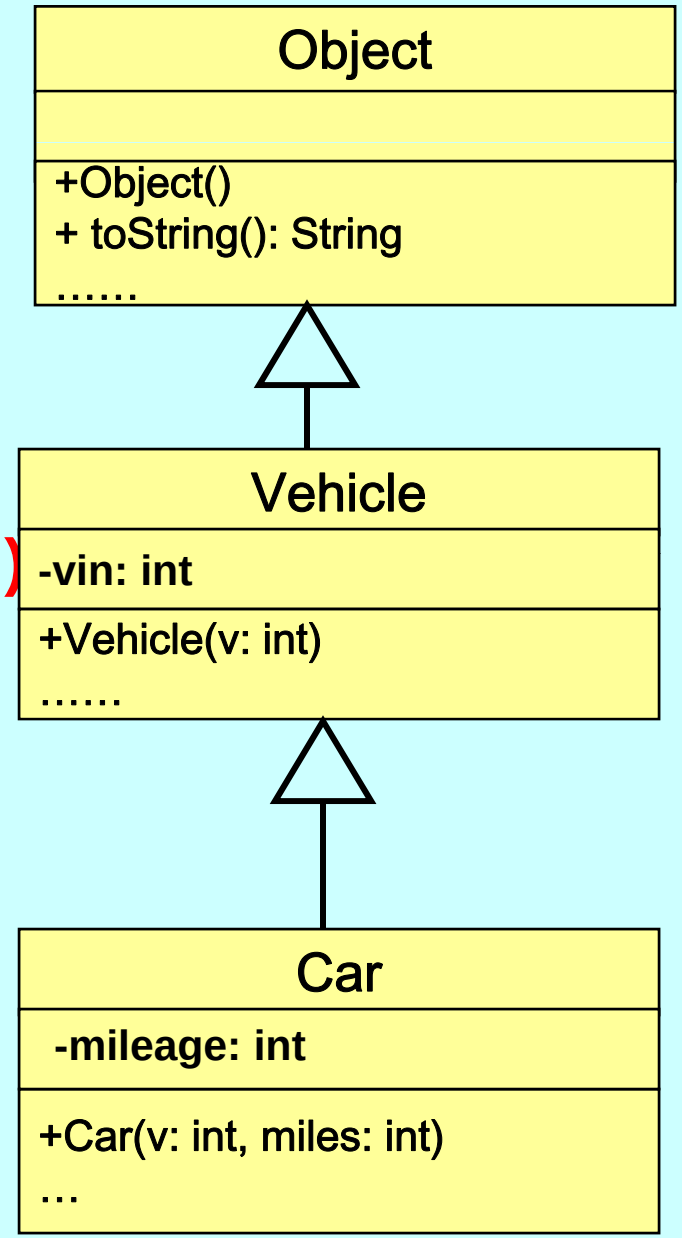
```
public Boop() {
    super();
}
public Boop(int j) {
    super();
    size = j;
}
public Boop(){
}
public Boop(int j) {
  size = j;
}
```




```
public Boop(int j) {
  size = j;
  super();
}
```

Superclass constructors with arguments

```
public class Vehicle {
    private int vin;

    public Vehicle( int v )
    {
       vin = v;
    }
    ……
}
```



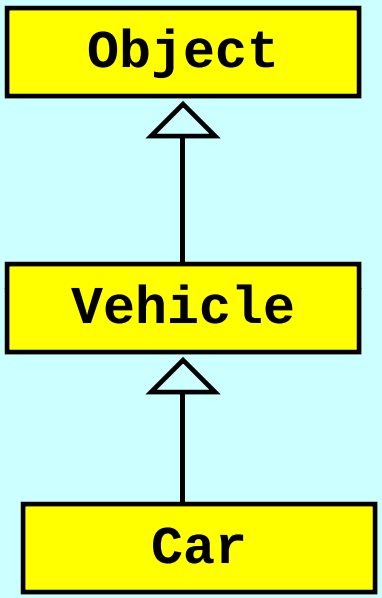
```
public class Car extends Vehicle {
    private int mileage;

    public Car(int v, int miles) {
        mileage = miles;
    }

    ……..
}
```

compilation error ?

Explicitly invoke the superclass constructor

```
public class Car extends Vehicle {
    private int mileage;

    public Car(int v, int miles) {
        super( v );
        mileage = miles;
    }

    ……
}
```

Order of Constructor Invocation

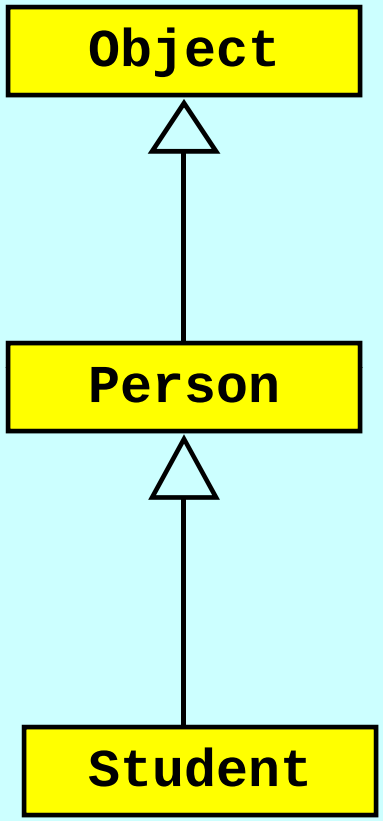


- super() or super(…) must be the first statement in the constructor

- if no explicit invocation of superclass constructor, Java automatically calls super()

- if superclass has no default constructor, compilation error occurs

Order of Constructor Invocation




```
public Object()
{ … }
```

if present, must be the first statement in the ctor

```
public Person(String n) {
    super();
    name = n;
}
```

```
public Student(String n, int i) {
    super( n );
    age = i;
}
```

Invoking one overloaded constructor from another

```
public class Car extends Vehicle {
    private int mileage;

    public Car(int v) {
        this( v, 0 );
    }

    public Car(int v, int m) {
        super( v );
        mileage = m;
    }

    ……
}
```

- this() or this(…) invokes another constructor in the same class

- If the first line of a constructor is this() or this(…), super() is not called automatically

- Every constructor can have a call to super(..) or this(..), but never both !

Exercise 1

```
C obj = new C();
```

```
public class A {
    public A() { System.out.println("ctor A");}
}

public class B extends A {
    public B() { System.out.println("ctor B");}
}

public class C extends B {
    public C() { System.out.println("ctor C");}
}
```

Output ??

Exercise 2

```
public class A {
    public A( int j ) { }
}

public class B extends A {
    public B( int j ) { }
}
```

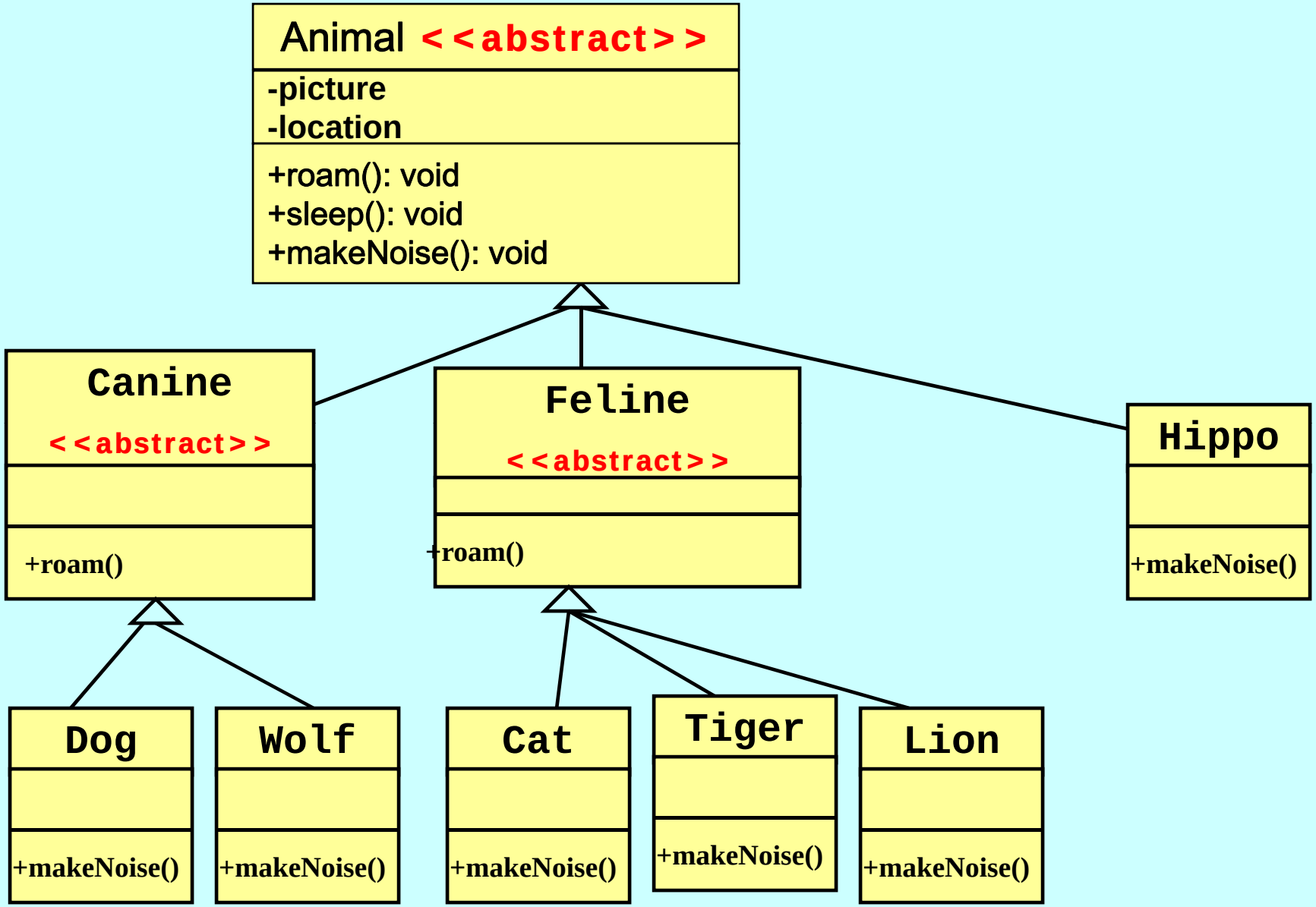
Compile ??

Abstract Classes

- **represents an abstract/generic concept in a class hierarchy**

- **cannot be instantiated**

- **can be subclassed, can have a superclass**


Animal <<abstract>>
-picture
-location
+roam(): void
+sleep(): void
+makeNoise(): void
Canine
<<abstract>>
+roam()
Feline
<<abstract>>
+roam()
Hippo
+makeNoise()
Dog
+makeNoise()
Wolf
+makeNoise()
Cat
+makeNoise()
Tiger
+makeNoise()
Lion
+makeNoise()

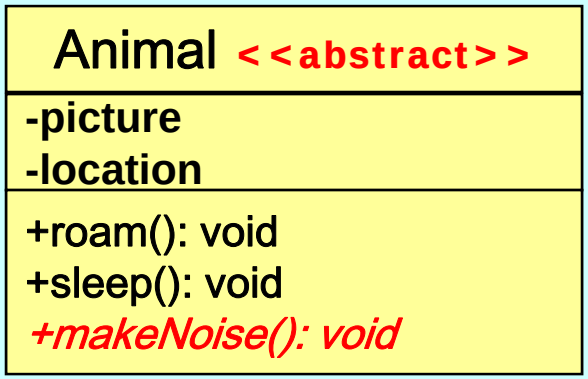
```
public abstract class Animal {
    ……
}
public abstract class Canine extends Animal {
    ……
}
```

```
public class CreateAnimal {
    public static void main(String[] args) {
        Animal a = new Animal();
        Canine c = new Canine();
        Animal a = new Dog();
        Canine c = new Wolf();
    }
}
```

Abstract Methods

- some methods in an abstract class does not make any sense unless they are implemented by a more specific subclass.

- An abstract method has NO BODY.




```
public abstract class Animal {
   public abstract void makeNoise();
   ……
}
```

If you declare an abstract method in a class, you must declare that class abstract as well.

Abstract Classes and Abstract Methods

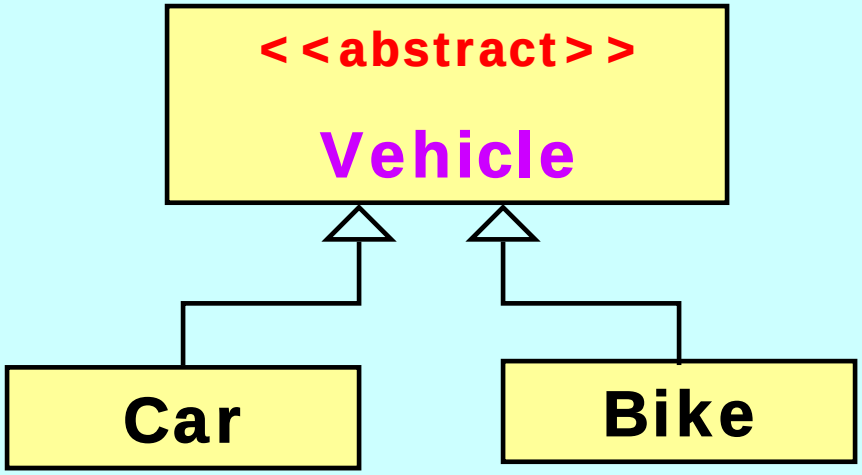
- **usually contains one or more abstract methods (i.e. methods with no body, just showing return type and signature)**

- **a subclass of an abstract class is instantiable only when all its abstract methods have been implemented (i.e. method body provided) in the subclass or its ancestors decending from that abstract class.**

- **may contain methods that are not abstract (i.e. methods with body)**

Abstract Class: An example

Let Vehicle be an abstract class – It makes sense to model vehicles in a program but it may not make sense to create a generic vehicle object. Instead, Car or Bike might seem to imply specific kinds of vehicles.



A class that is not abstract is a concrete class. Car and Bike are concrete classes.

Declaration for abstract class Vehicle

```
public abstract class Vehicle {

   public abstract boolean hasEngine();

}
```

Declaration for concrete class Car

```
public class Car extends Vehicle {

   public Car() {}

   public boolean hasEngine() { return true; }
}
```

Declaration for concrete class Bike

```
public class Bike extends Vehicle {

  public Bike() {}
  public boolean hasEngine() { return false; }

}
```

Abstract concept testing

```
public class AbstractTest {
   public static void main( String[] args ) {
      Vehicle v = new Vehicle();
      Vehicle b = new Bike();
      System.out.println( b.hasEngine() );
      Vehicle  c = new Car();
      System.out.println( c.hasEngine() );
    }
}
```

when is this error detected ?

outputs ??

Implementing an abstract method: provide the method body

```
public abstract class A {
  abstract void foo();
}
public class B extends A{
  void foo() { int j = 0;}
}
```

compile ??

```
public abstract class A {
  abstract void foo();
}
public class B extends A{
}
```

compile ??

Implementing an abstract method: provide the method body

```
public abstract class A {
  abstract void foo();
}
public abstract class B
extends A {
  void foo() { int j = 0;}
}
public class C extends B {
}
```

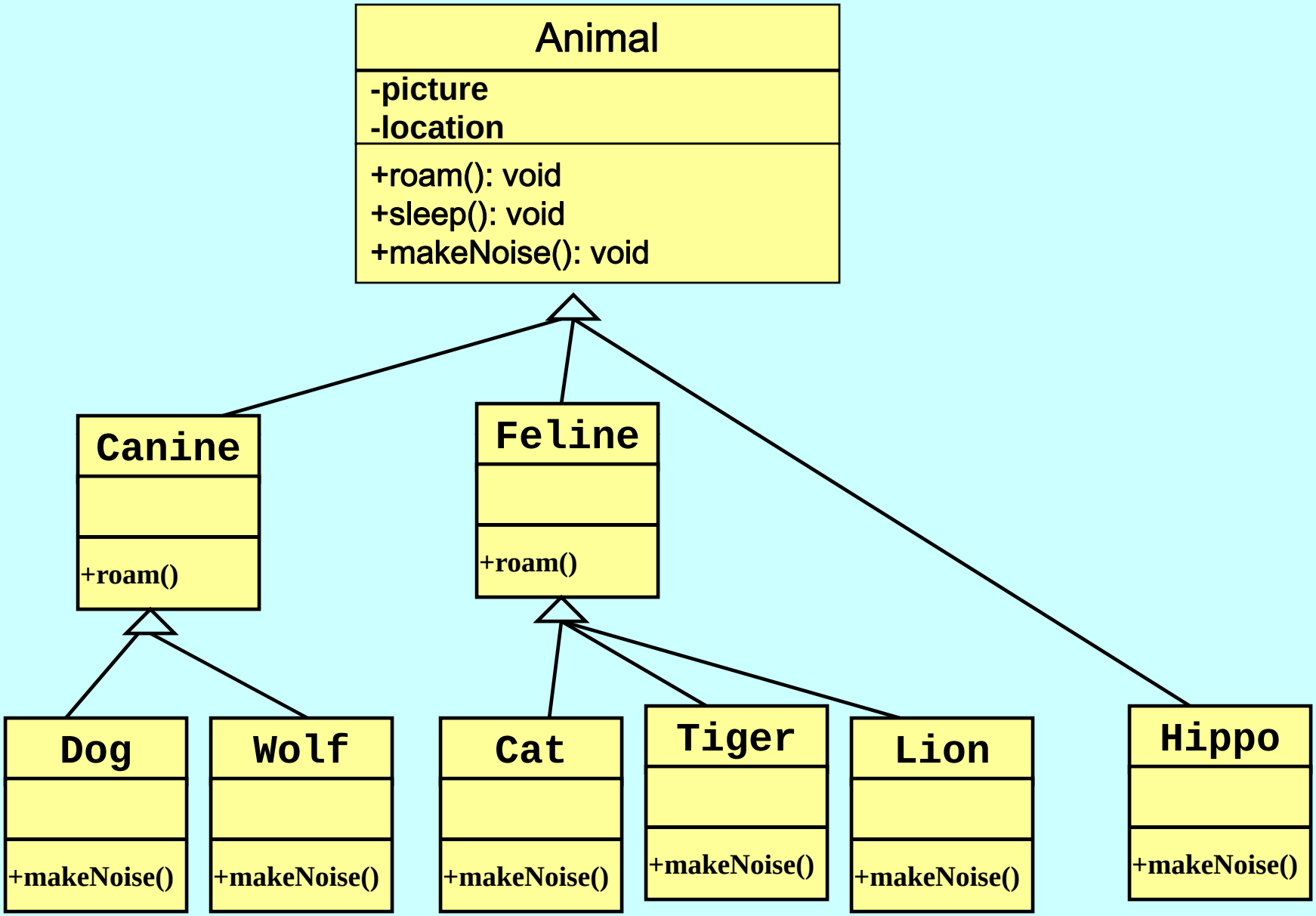
compile ??

```
public abstract class A {
  abstract void foo();
}
public abstract class B
extends A {
}
public class C extends B {
  void foo() { int j = 0;}
}
```

compile ??

Polymorphism:

When you define a supertype for a group of classes, any subclass of that supertype can be substituted where the supertype is expected..


Animal
-picture
-location
+roam(): void
+sleep(): void
+makeNoise(): void
Canine
+roam()
Feline
+roam()
Dog
+makeNoise()
Wolf
+makeNoise()
Cat
+makeNoise()
Tiger
+makeNoise()
Lion
+makeNoise()
Hippo
+makeNoise()

To see how polymorphism works..

```
Dog myDog = new Dog();
```

- Here reference type and object type are same..

- But with polymorphism, the reference and object type can be different..

```
Animal a = new Dog();
```

Polymorphism: the reference type can be a superclass of the actual object type

```
Animal[] animals = new Animal[3];
animals[0] = new Dog();
animals[1] = new Cat();
animals[2] = new Wolf();

for( Animal a: animals ) {
    animals[i].makeNoise();
}
```

Polymorphism: can have polymorphic arguments and return type

```
public class Vet {
    ……
    public void giveShot(Animal a) {……}
}
public class PetOwner {
     public void start() {
         Vet v = new Vet();
         v.giveShot( new Dog() );
         v.giveShot( new Cat() );
     }
}
```

Type Checking

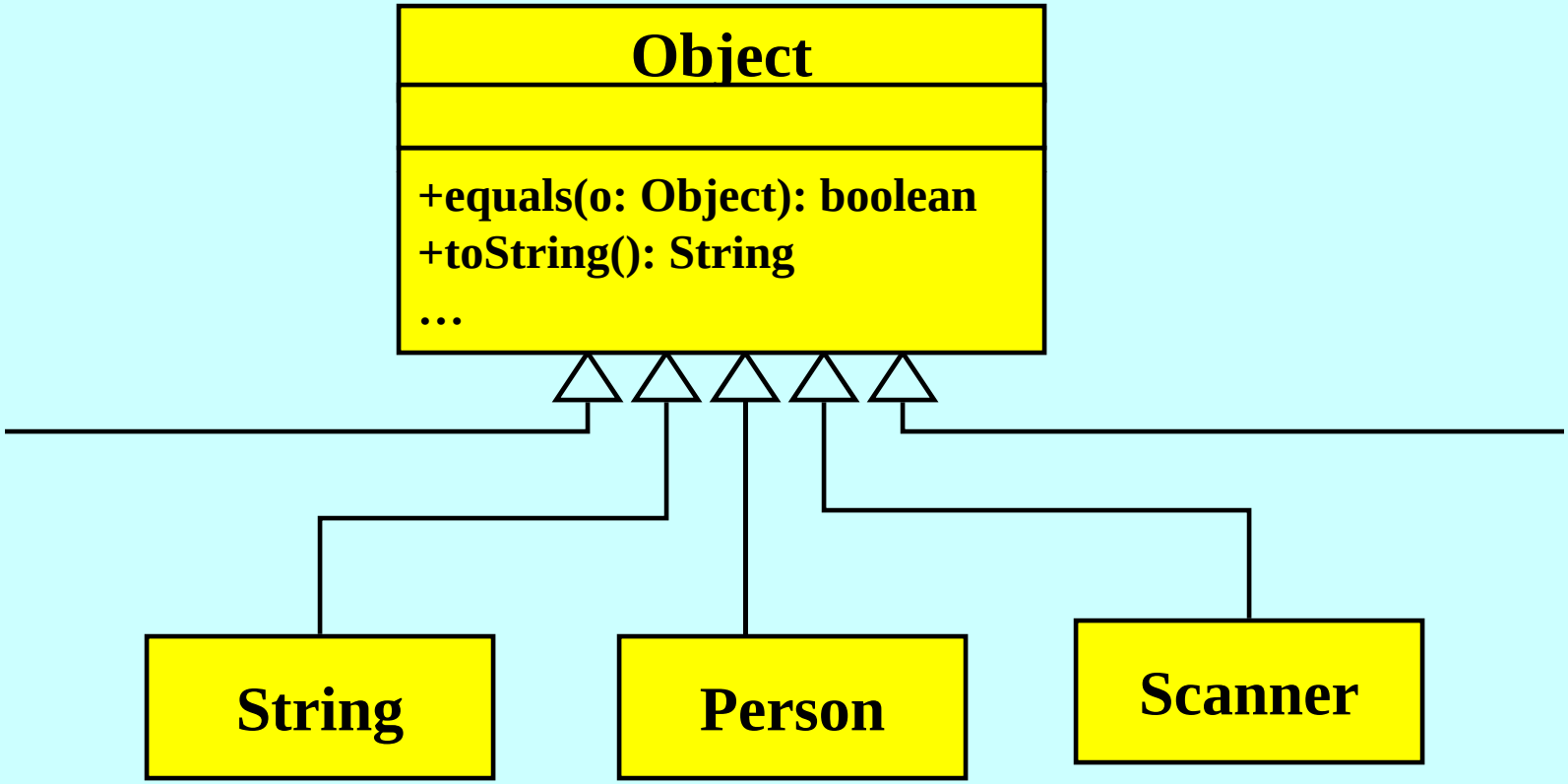
Java is Strongly Typed.

```
int y = 7, z = 3;
int x = Num.gcd (z,y);
```

```
public class Num {
   public static int gcd (int n, int d) {
      while (n != d)
      if (n > d)
         n = n - d;
      else d = d - n;
      return n;
   }
}
```

Type Hierarchy

```
Object o1 = new String( “abc” ); // legal
Object o2 = new String( “def” );  // legal
```



Type Hierarchy

```
if ( o1.equals(“abc”) ) // legal
if ( o2.equals(“abc”) ) // legal
if ( o1.length() )  // illegal , o1 is
                    //object, no length method

String s = o1; // illegal, o1 is an object
               // of String
if ( ( (String)o1 ).length() ) // legal
String s = (String) o1; // legal
```

Method Overloading

- ***Method overloading:*** **In Java, you can use same method name with different parameter lists for multiple methods**

- ***method signature:*** **a method's name, along with the number, type and order of its parameters**

- **signature of each overloaded method must be different**

Method Overloading: Example

```
public class Adder {
  public int add(int n1, int n2) {
      return n1 + n2;
  }

  public int add(int n1, int n2, int n3) {
      return n1 + n2 + n3;
  }
}
```

Invocation

```
Adder adder = new Adder();
adder.add(1, 3);
```

```
adder.add(1, 3, 4);
```

Method Overloading: Example

- **`println` method in PrintStream class is overloaded:**

```
println( String s )
println( int i )
  …
```

```
System.out.println( “Number of students: ” );
System.out.println( 5 );
```

Method Overloading

- **Return type is NOT part of method signature**
- **overloaded methods cannot differ only be their return type**

- **Constructors can be overloaded**

```
public Circle()
public Circle( int radius )
```

Method Overloading

- ***Method overloading:*** **In Java, you can use same method name with different parameter lists for multiple methods**

- ***method signature:*** **a method's name, along with the number, type and order of its parameters**

- **signature of each overloaded method must be different**

Method Overloading: Example

```
public class Adder {
  public int add(int n1, int n2) {
      return n1 + n2;
  }

  public int add(int n1, int n2, int n3) {
      return n1 + n2 + n3;
  }
}
```

Invocation

```
Adder adder = new Adder();
adder.add(1, 3);
```

```
adder.add(1, 3, 4);
```

Method Overloading: Example

- **`println` method in PrintStream class is overloaded:**

```
println( String s )
println( int i )
  …
```

```
System.out.println( “Number of students: ” );
System.out.println( 5 );
```

Method Overloading

- **Return type is NOT part of method signature**
- **overloaded methods cannot differ only by their return type**

- **Constructors can be overloaded**

```
public Circle()
public Circle( int radius )
```

Vector Class

- **Vectors are extensible arrays; they are empty when first created and can grow and shrink on the high end**

- **defined in java.util package**

- **Like an array, it contains elements indexed from 0 to (current length – 1)**

Vector Class (contd.)

- **Element type: Object**

- **int size() method : returns the length of a vector**

- **Object get(int index) : returns the indexed element**

- **Different elements of a Vector can be objects of different types**

Adding elements

```
Vector v = new Vector();  //creates empty Vector


if (v.size() == 0) // true
v.add( new String(“abc”) );//increases size by 1
                           //and stores argument
                           //in the new location


v.add( new String(“def”) );
v.add( new String(“gh”) );
System.out.println( v.size() ); //size is 3
```

Accessing elements

```
String s = (String) v.get(0);//cast is necessary
                             //because get
                             //returns an Object

System.out.println( s ); // output: abc

String s = (String) v.get(3);
                //throws IndexOutOfBoundsException
```

Setting elements

```
v.set( 0, new String("ijk") ); //changes the
                               //element at index 0
```

Removing elements

```
v.remove( 1 ); //removes the element at index 1
               //and shifts the remainder
               //to the left
```

The ArrayList Class

- **`ArrayList` class is part of java.util package**

```
ArrayList band = new ArrayList();
```

- **add things in it**

```
band.add( new String("Paul") );
band.add( new Integer(2) );
```

- **[ ] syntax cannot be used**

also Vector class…

The ArrayList Class **(contd.)**

- **We can declare an ArrayList object to accept a particular type of object (Java 5.0)**

```
ArrayList<String> list =
                        new ArrayList<String>();
list.add( new String("Paul") );


list.add( new Integer(2) );
```

compile error

The ArrayList Class **(contd.)**

- **each value is stored at a specific position**

```
String s1 = list.get(0);
System.out.println( s1 );
```

no need to cast to String

- **Important: grows and shrinks as needed**

```
String s2 = new String("Anil");
list.add( s2 );
list.remove( s1 );
```

When an element is removed, the list "collapses" to close the gap

(indexes adjust accordingly)

The ArrayList Class **(contd.)**

- **number of elements ??**

```
int i = list.size();
System.out.println( i );
```

- **index of an element ??**

```
int j = list.indexOf( s2 );
System.out.println( j );
```

- **elements can be removed from any location (index) with a single method invocation**

```
list.remove( 0 );
```

The ArrayList Class **(contd.)**

- **remove all elements ??**

```
list.clear();
```

- **find out if it contains something**

boolean isIn = `list.contains( s1 );`

- **find out if its is empty**

boolean empty = `list.isEmpty();`

- **other methods: …**

Comparing ArrayList to a regular array

- **A regular array has to know the size at the time it is created. ArrayList grows and shrinks as needed.**

- **To put an object in a regular array, you must assign it to a specific location. With ArrayList, you can specify an index or you can just say add.**

Comparing ArrayList to a regular array (contd.)

- **Arrays use array syntax that is not used anywhere else in Java. ArrayList have no special syntax just like other objects.**

- **ArrayLists in Java 5.0 are parameterized.**

The ArrayList Efficiency

- **ArrayList class is implemented using an array.**

• The underlying array is manipulated so that the indexes remains continuous when elements are added or deleted

• If added or removed from END of the list, processing is efficient

• But as elements are added or removed from the front or middle of the list, processing can become inefficient

Wrapper Classes

```
int x = 32;
ArrayList list = new ArrayList();
list.add( x );
```

This won't work unless you are using Java 5.0

There is a wrapper class (in java.lang package) for every primitive type.

Boolean, Character, Byte, Short, Integer, Long, Float, Double

Wrapping and Unwrapping a primitive value

```
int i = 288;

Integer iWrap = new Integer( i );
```

When you need to treat a primitive like an object, wrap it.

```
int unwrapped = iWrap.intValue();
```

Boolean class has booleanValue(), Character class has charValue(),…

Without Autoboxing (Java versions before 5.0)

```
ArrayList list = new ArrayList();
list.add( new Integer(3) );
Integer one = (Integer) list.get( 0 );
int intOne = one.intValue();
```

With Autoboxing (Java version 5.0 or greater)

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add( 3 );
int intOne = list.get( 0 );
```

Autoboxing feature added to Java 5.0 does the conversion from primitive to warpper object automatically.

Wrapper classes also provide useful static methods:

Converting a String to a primitive type

```
String s = "2";
int x = Integer.parseInt( s );
double d = Double.parseDouble("420.24");
boolean b = new Boolean("true").booleanValue();
```

```
If you try:
String t = "three";
int y = Integer.parseInt( t );
```

Cause a NumberFormatException

Wrapper classes also provide useful static methods:

Converting a primitive number into a String

```
double d = 42.6;
String dStr = “” + d;
```

```
double d = 42.6;
String dStr = Double.toString( d );

int i = 99;
String iStr = Integer.toString( i );
```