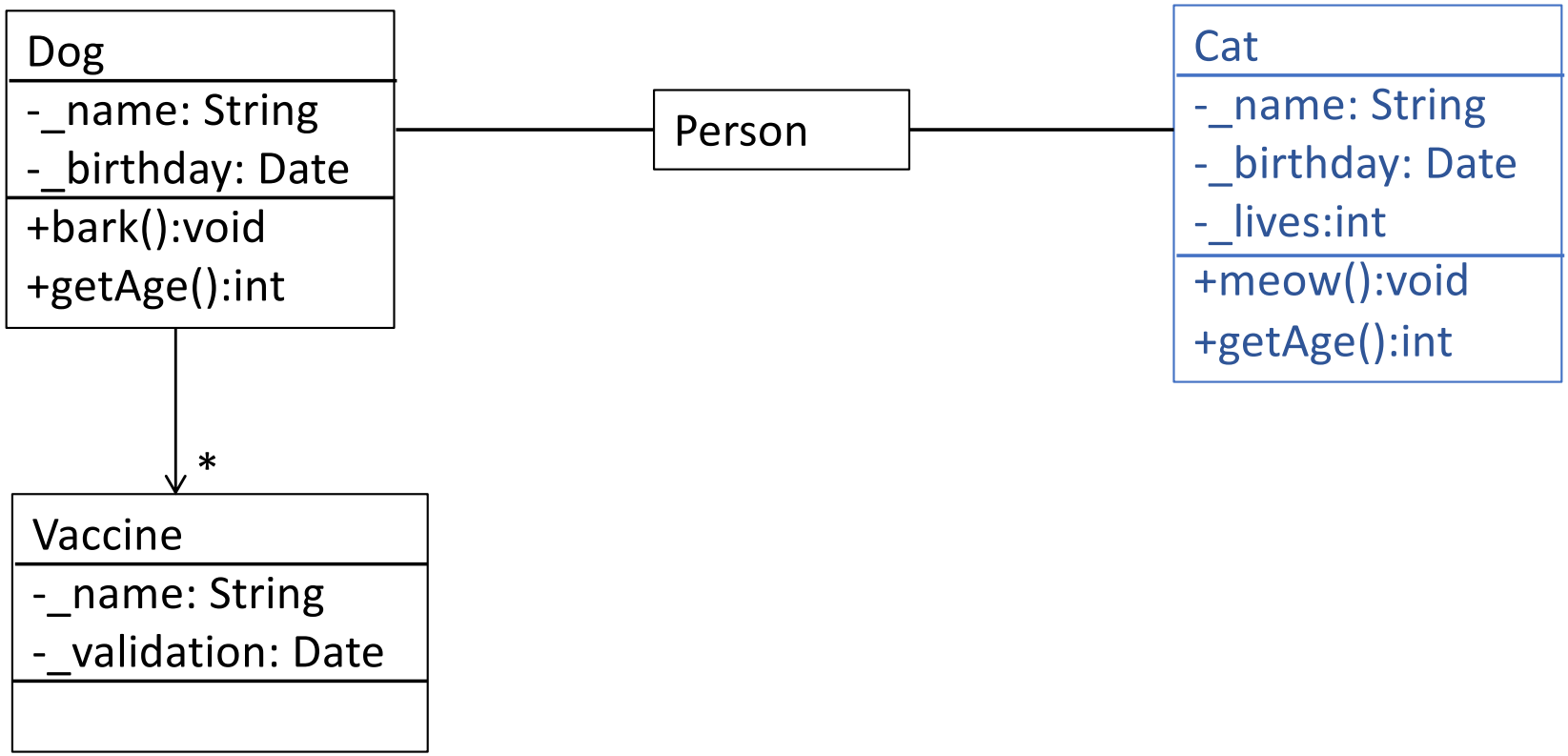
Inheritance

Code Reuse

Example: Shared Functionality - 1

A dog has a name, birthday, an owner and a set of vaccines

Behavior: barks and can know its age



A cat has a name, birthday, an owner and a number of lives

Behavior: meows and can know its age

Example: Shared Functionality - 2

```
public class Dog {
  private String _name;
  private Date _birthday;
  private Person _owner;
  private Vacine[] _vacine;

  public void bark() {
   System.out.println("ão ão");
  }

  public int getAge() {
    …
  }
}
```

```
public class Cat {
  private String _name;
  private Date _birthday;
  private Person _owner;
  private int _lifes;

  public void meow() {
    System.out.println("miau miau");
  }

  public int getAge() {
    …
  }
}
```

How to avoid code duplication?

Possible Solution: Use Composition

```
public class Animal {
  private String _name;
  private Date _birthday;
  private Person _owner;

  int getAge() {
    …
  }
}
```

```
public class Dog {

  private Animal _animal;
  private Vacine[] _vacine;

  public void bark() {
    …
  }
}
```

- Code duplication is avoided
- But there are problems with this solution:
  - The public interface of Dog and Cat is not the same as before
  - The relationship between **Cat** (**Dog**) and **Animal** is not a **has-a**
    - **Cat** is a **Animal**
    - **Dog** is a **Animal**

```
public class Cat {

  private Animal _animal;
  private int _numberLifes;

  public void meow() {
    …
  }
}
```

Possible Solution: Use Composition - 2

```
public class Animal {
  private String _name;
  private Date _birthday;
  private Person _owner;

  int getAge() {
    …
  }
}
```

```
public class Dog {

  private Animal _animal;
  private Vacine[] _vacine;

  public void bark() {
    …
  }
  public int getAge() {
    return _animal.getAge();
  }
}
```

- Code duplication is avoided
- But there are problems with this solution:
  - The public interface of Dog and Cat is not the same as before
    - Can solve this
      - **but duplicate code**
  - The relationship between **Cat** (**Dog**) and **Animal** is not a **has-a**
    - **Cat** is a **Animal**
    - **Dog** is a **Animal**

```
public class Cat {

  private Animal _animal;
  private int _numberLifes;

  public void meow() {
    …
  }
  public int getAge() {
    return _animal.getAge();
  }
}
```

Relationships

- Object oriented programming leads to programs that are models
  - sometimes models of things in the real world
  - sometimes models of contrived or imaginary things
  - Two types of relationships between the modelled entities
    - **Has-a**
    - **Is-a**
- Chess
  - The Board **has** 32 chess pieces
  - A chess piece **has a** position
  - A chess piece **has a** color
  - A rook **is a** type of chess piece
  - A chess piece moves (changes position)
    - Behavior is specific for each type of chess piece

The **has-a** Relationship

- Objects are often made up of many parts or have sub data
  - chess piece: position, color, …
  - animal: owner, birthday, …
- The **has-a** relationship is modeled by composition
  - Each **has-a** relationship is implemented by one field internal to objects
  - Type of internal field depends on multiplicity of composition

The **is-a** Relationship

- Another type of relationship found in the real world
  - a rook is a chess piece
  - a queen is a chess piece
  - a student is a person
  - a teacher is a person
  - an undergraduate student is a student
- **is-a** usually denotes some form of specialization
  - May offer more functionalities (*methods*)
  - And/or some functionalities have distinct implementation
- It is **not** the same as **has-a**

Is-a relationship - Inheritance

- The **is-a** relationship is modeled in object oriented languages via *inheritance*
- Classes can inherit from other classes
  - **Base** inheritance in a program on the real world things being modeled
  - Does “an A is a B” make sense? Is it logical?
- In Java the `extends` keyword is used in the class header to specify which preexisting class a new class is inheriting from

```
public class Dog extends Animal
```

- Animal is said to be
  - the parent class of Dog
  - the super class of Dog
  - the base class of Dog
  - *an ancestor of Dog*
- Dog is said to be
  - a child class of Animal
  - a subclass of Animal
  - a derived class of Animal
  - *a descendant of Animal*

Results of Inheritance

public class B extends A

- The subclass inherits (gains) all fields and instance methods of the super class, **automatically**
  - The non-static fields defined in A are also part of the state of every B object
- Additional methods can be added to subclass
  - Called specialization
- The subclass can replace (redefine, **override**) methods from the super class
- Inheriting all member does not mean direct access
  - private and package-private members not accessible

Example

Animal

- **Has**:
  - birthday, name and owner
- **Do**:
  - getAge(), getOwner()

- **Dog**
  - **Is-a** Animal
  - **Has**:
    - birthday, name, owner
    - vacines
  - **Do**:
    - getAge(), getOwner(),
    - bark(), waggingTail()

```
class Animal {
  private String _name;
  private Date _birthday;
  private Person _owner;

  public int getAge() {
    … }

  public Person getOwner() {
    …}
}
```

```
class Dog extends Animal {
  private Vacine[] _vacine;

  public void bark() {
    … }

  public void waggingTail()
  { … }
}
```

```
class Cat extends Animal {
  private int _lifes;

  public void meow() {
    …
  }
  public void climb() {
    …
  }
}
```

Overriding Methods

- A subclass can override (redefine) the methods of the superclass
  - Objects of the subclass type will use the new method
  - Objects of the superclass type will use the original
- This allows to make more methods that are common to several subtypes using the right abstraction
- Cat and Dog have a similar method (same semantic) but with a different implementation
  - bark() and meow(): *animal talk or make noise*
  - But should not have both methods at Animal
  - Solution: have a method makeNoise() at Animal
  - And must be implemented in both classes (Cat and Dog)
    - Otherwise, the computation would be the same

Solution with Overriding

```
public class Animal {
  private String _name;
  private Date _birthday;
  private Person _owner;

  public int getAge() {
    …
  }

  public Person getOwner() {
    …
  }
  public void makeNoise() {
  }
}
```

```
public class Dog extends Animal {
  private Vacine[] _vacine;

  public void waggingTail() {
    …
  }
  public void makeNoise() {
   System.out.println("ão ão");
  }
}
```

```
public class Cat extends Animal {
  private int _lifes;

  public void climbTree() {
    …
  }
  public void makeNoise() {
    System.out.println("miau miau");
  }
}
```

@Override Annotation

- Use **@Override** annotation every time a method is overridden

```
class ParentClass {

    public void displayMethod(String msg) {
      System.out.println(msg);
    }
}
```

```
class Subclass extends ParentClass {

  @Override
  public void displayMethod(String msg) {
    System.out.println("Message is: "+ msg);
  }
}
```

- Advantages:
  - If programmer makes a mistake (wrong name or wrong parameter list) and method is not overridden, compiler gives error
  - Improves readability of the code

Solution with Overriding

```
public class Animal {
  private String _name;
  private Date _birthday;
  private Person _owner;

  public int getAge() {
    …
  }

  public Person getOwner() {
    …
  }
  public void makeNoise() {
  }
}
```

```
public class Dog extends Animal {
  private Vacine[] _vacine;

  public void waggingTail() {
    …
  }
  @Override public void makeNoise() {
   System.out.println("ão ão");
  }
}
```

```
public class Cat extends Animal {
  private int _lifes;

  public void climbTree() {
    …
  }
  @Override public void makeNoise() {
    System.out.println("miau miau");
  }
}
```

Attendance Question 2

What is output when the `main` method is run?

```
public class Foo{
   public static void main(String[] args){
      Foo f1 = new Foo();
      System.out.println(f1.toString());
   }
}
```

A. `0`

B. `null`

C. Unknown until code is actually run

D. No output due to a syntax error

E. No output due to a runtime error

Possible: Foo@677327b6

Inheritance in Java

- Java is a pure object-oriented language
  - All code is part of some class
- All classes, except one, **must** inherit from exactly one other class
- The `Object` class is the *cosmic superclass*
  - The `Object` class does not inherit from any other class
  - The `Object` class has several important methods: `toString()`, `equals()`, `hashCode()`, `clone()`, `getClass()`
- Implications:
  - All classes are descendants of `Object`
  - All classes and thus all objects have a `toString()`, `equals()`, `hashCode()`, `clone()`, and `getClass()` method
    - `toString()`, `equals()`, `hashCode()`, `clone()` normally overridden

Inheritance in Java

```
public class Foo {
  public static void main(String[] args){
    Foo f1 = new Foo();
    System.out.println(f1.toString());
  }
}
```

Where is Object?

- If a class header does not include the extends clause the class extends the `Object` class by default
  - `Object` is an ancestor to all classes
  - it is the only class that does not extend some other class
- A class extends exactly one other class
- Extending two or more classes at the same time is designated as *multiple inheritance.*
  - Java does not support this directly, rather it uses *Interfaces*.
  - C++ supports

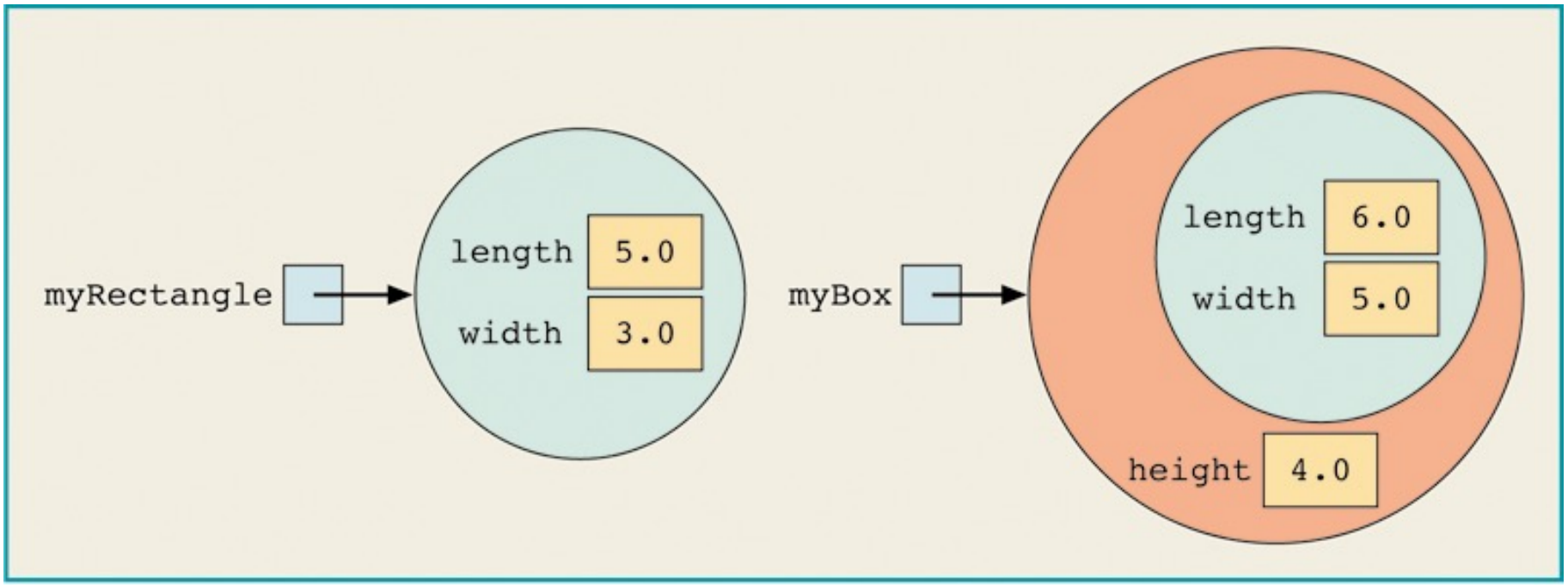
Objects `myRectangle` and `myBox`

```
public class Rectangle {
  private int _length;
  private int _width
  …
}
```

```
public class Box extends Rectangle {
  private int _height;

  …
}
```

```
Rectangle myRectangle = new Rectangle(5, 3);
Box myBox = new Box(6, 5, 4);
```



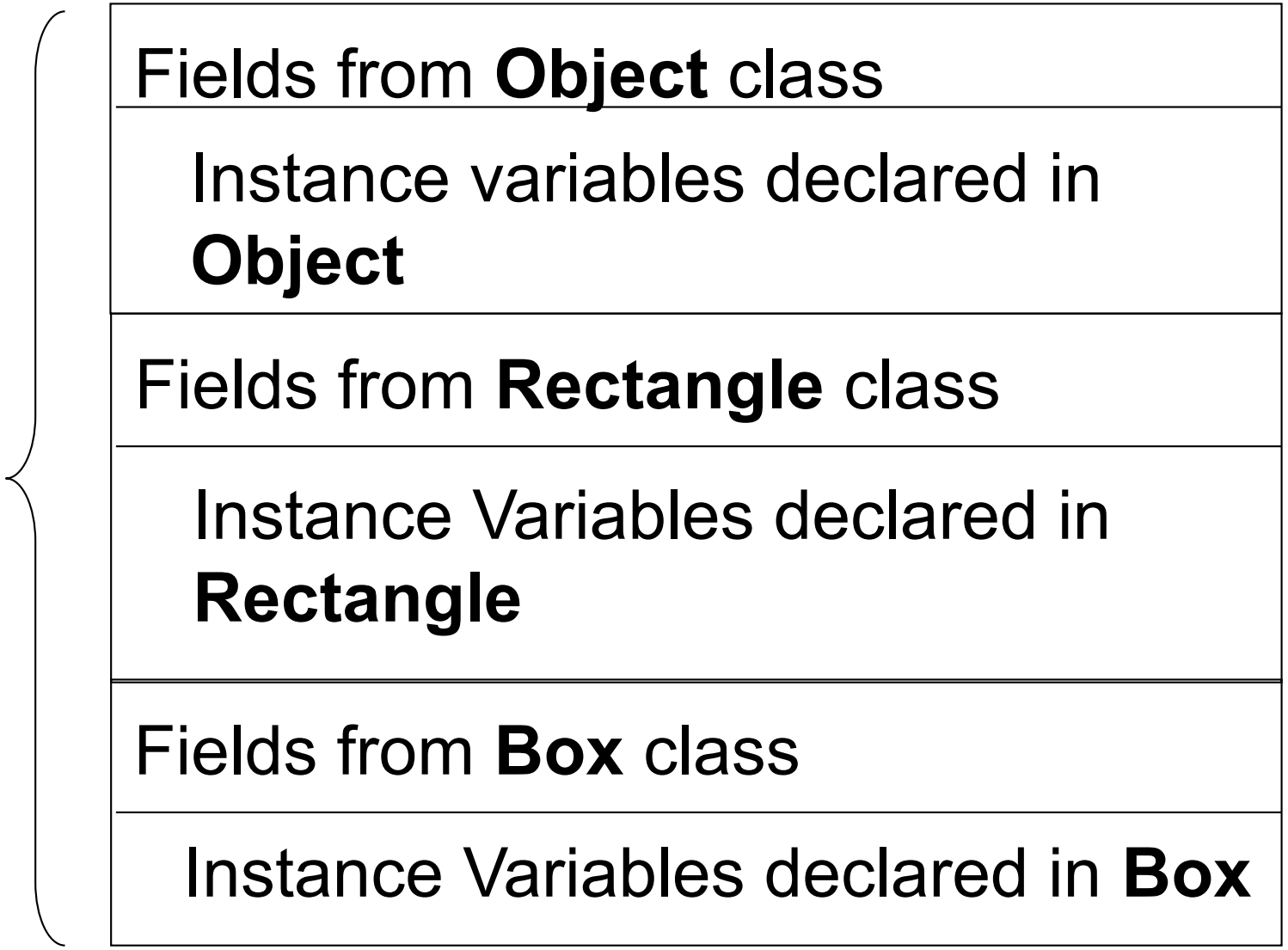
The Real Picture with Inheritance

A **Box** object

state

behavior

Available methods are all methods from **Object**, **Rectangle** and **Box**

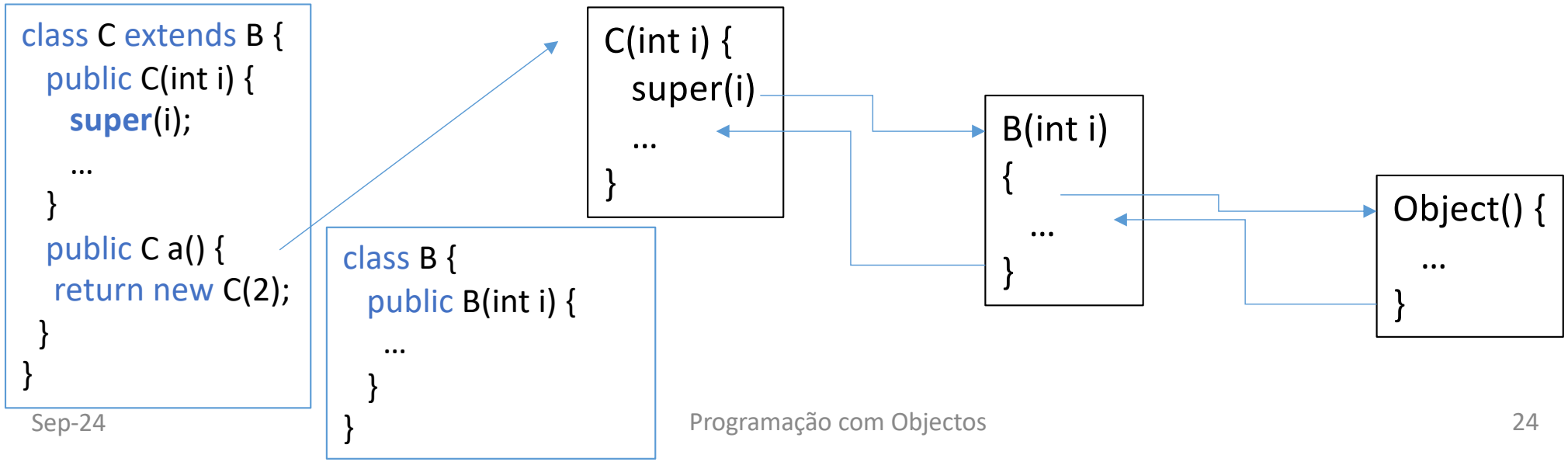


Constructors with Inheritance

- Constructors handle initialization of objects
- When creating an object with one or more ancestors (every type except Object) a chain of constructor calls takes place
- The reserved word **`super`** may be used in a constructor to call one of the parent's constructors
  - must be first line of constructor
- If no parent constructor is explicitly called the default **no-arg** constructor of the parent is called
  - if no default constructor exists a syntax error results
- Cannot invoke parent constructor and another constructor in the same class
  - `super();this();` is **not** allowed
  - One or the other, not both

Constructors in Subclasses

- Constructors are *not* inherited!
- Chain of constructor calls
  - subclass constructor invokes superclass constructor
    - Implicitly or explicitly
    - To call explicitly, use *super(params)*. Otherwise, *no-arg ctor* is called
    - Superclass constructor call must be first statement in subclass constructor
  - `Object` constructor is always fired last
    - But it is the first to be executed



Invoking methods of the superclass

- How to invoke an accessible method of the superclass in the context of the subclass?

- Method not overridden in subclass
  - Just invoke the method
  - `methodName(parameter list)`

- Method Overridden in subclass
  - If invoke `MethodName(parameter list)` the invoked method is the one defined in subclass
  - Must use the **super** keyword to specify that we want to invoke the version defined for the superclass

    `super.MethodName(parameter list)`

Override and Overloading

```
public class Rectangle{
  private int _length, _width;

  public Rectangle(int l, int w) {
     _length = l;
     _width = w;
  }
  public int area() {
   return _length * _width;
  }
  public void print() {
     System.out.println("length : " + _length);
     System.out.println("width : " + _width);
  }
  public void setDimension(double l, double w) {
     _length = (l >= 0) ? l : 0;
     _width = (w >= 0) ? w : 0;
  }
}
```

Override and Overloading

```
public class Rectangle{
  public void area() { … }
  public void print() { … }
  public void setDimension(double l, double w) { … }
}
```

```
public class Box extends Rectangle {
 private int _height;
 public Box(int l, int w, int h) {
  super(l, w);
  _height = h;
 }
 public void volume() {
  return area() * _height;
 }
 public void print() {
  super.print();
  System.out.println("h: " + _height);
 }
 public void setDimension(double l,
                          double w, double h) {
  super.setDimension(l, w);
  if (h >= 0) _height = h;
  else _height = 0;
 }
}
```

Inherited methods?

- *area* *setDimension(l,w)*

Overridden methods?

- *print*, *setDimension(l,w)*

Box **overloads** *setDimension*

- Same name but different parameters

New methods (specialization)

- volume()
- setDimension(l,w,h)

super not really needed here

The Keyword `super`

- `super` is used to access something (any protected or public **field** or **method**) from the super class that has been overridden
- Box's `print` makes use of the `print` in Rectangle by calling `super.print()`
- Without the super calling `print()` would result in infinite recursive calls

```
public class Box extends Rectangle {
  // ….
  public void print() {
    print();
    System.out.println("height: " + _height);
  }
}
```

- Java does not allow nested super's

  `super.super.print()`

Overloading vs. Overriding

- Don't confuse the concepts of overloading and overriding
- Overloading deals with multiple methods with the same name in the same class, but with different signatures
- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
- Overloading lets you define a similar operation in different ways for different parameter types
- Overriding lets you define a similar operation in different ways for different object types

Hiding Fields

- The concept of *overriding* can also be applied to fields and is called *shadowing variables* or *hiding fields*
- Define a field in subclass with the same name as a field in the superclass
  - Type can be different
  - Code in subclass accesses the field defined in subclass
  - Code in superclass accesses the field defined in superclass
  - To access superclass field in subclass use *super* keyword
    - super.fieldName
- Hiding fields makes code difficult to read

Access Modifiers and Inheritance

- **public**
  - accessible to all classes
- **private**
  - accessible only within that class. Hidden from all sub classes.
- **protected**
  - accessible by classes within the same *package* and all descendant classes
- Fields ***should*** be private
- **protected** methods are used to allow descendant classes to modify instance variables in ways other classes can't

Comments on Private vs. Protected

- Use *private* so that
  - Superclass implementation can change without affecting subclass implementations
- Use *protected* when
  - Superclass should provide a service only to its subclasses
  - Should not provide service to other clients
- Avoid *protected* fields
  - Do not preserve the *encapsulation* principle
  - Provide set and get methods to access *private* fields
    - *public* or *protected*

Using protected Fields

- Advantages
  - Subclasses can modify values directly
  - Slight increase in performance
    - Avoid set/get method call overhead (may not be true)
- Disadvantages
  - No validity checking
    - subclass can assign illegal value
  - Implementation dependent
    - subclass methods more likely dependent on superclass implementation
    - superclass implementation changes may result in subclass modifications
      - Fragile (brittle) software

Example

package a

```
public class Employee {
  protected Date hireDay;
  . . .
}
```

package b

```
public class Department {
  public Date getDate(Employee p){
    return p.hireDay;  ✗
  }
}
```

- Valid accesses?

package a

```
public class Department {
  public Date getDate(Employee p) {
    return p.hireDay;  ✓
  }
}
```

package b

```
public class Manager extends Employee {
  public Date getDate(){
    return hireDay;    ✓
  }

  public Date getDate(Employee p){
    return p.hireDay;  ✗
  }

  public Date getDate(Manager m){
    return m.hireDay;  ✓
  }
}
```

```
Manager.java:12: error: hireDay has protected access in Employee
   return p.hireDay;
             ^
1 error
```

What `protected` really means

- Precisely, a `protected` member is accessible
  1. within the class itself
  2. within code in the same package
  3. within code of a subclass through object references that are of at least the same type as the subclass

```
public class Manager extends Employee {
  public Date getDate(){
    return hireDay;
  }

  public Date getDate(Employee p){
    return p.hireDay;
  }

  public Date getDate(Manager m){
    return m.hireDay;
  }
}
```

final Method

- Can declare a method of a class *final* using the keyword `final`

```
public class A {
   public final void doSomeThing() {
      //...
    }
}
```

- **final** methods **cannot** be overridden in subclasses

```
public class B extends A {
    public void doSomeThing() {
       //...
    }
}
```

- Compilation error

final Class

- Can also declare a `class` final using the keyword `final`
- If a `class` is declared `final`, then no other class can be derived from this class
- Class String is final

Invoking methods in Constructors

- What is the problem with

```
public class A {
  public A() {
    doSomeThing();
  }

  public void doSomeThing() {
    //...
  }
}
```

```
public class B extends A {
  private String _myStr;

  public B(String str) {
    _myStr = str;
  }

  public void doSomeThing() {
    System.out.println(_myStr);
  }
}
```

- None
  - Unless we consider inheritance
  - What is written if "**new B("abc");**"?
    - **null** and not "**abc**"
  - Invoked method can be overridden in subclasses
  - May access to fields that are not initialized yet
- Safe methods in constructor - cannot be overridden
  - final
  - private

Any Problem?

```
public class Animal {
  private String _name;
  private Person _owner;

  public Animal(String n, Person o) {
    _name = n;
    _owner = o;
  }

  public void makeNoise() {
  }
  // …
}
```

```
public class Dog extends Animal {
  private Vacine[] _vacine;

  public Dog(String n, Person p, Vacine[] v) {
    super(n, p);
    _vaccine = v;
  }

  public void makeNoise() {
    System.out.println("ão ão");
  }
}
```

```
public class Cat extends Animal {
  private int _numberLifes;

  public Cat(String n, Person p) {
    super(n, p);
    _numberLifes = 7;
  }

  public void makeNoise() {
    System.out.println("miau miau");
  }
}
```

Yes!

- Does it make sense to have the following?
  - Animal a = new Animal("42", aPerson);
  - **No!**
- More problems?

```
Public class Cow extends Animal {
  private String _color;

  public Cow(String n, Person p, String c) {
     super(n, p);
    _color = c;
  }
}
```

  - makeNoise() is not overridden and it should be!

Abstract Classes

- Sometimes, when refactoring code, superclasses do not correspond to real entities
  - Are an abstraction
    - Example: *Dog* and *Animal*
  - Represented as an abstract class in Java
- Abstract class in Java
  - public **abstract** class A { …. }
  - Cannot be instantiated (but can define constructors)
    - Unlike *normal* classes – concrete classes
  - Can have methods without any implementation
    - Called ***abstract*** methods
    - A method that has only the heading with no body
    - And use *abstract* keyword
      - public abstract void doSomething();

Abstract Classes - 2

- An abstract class can contain fields, constructors, concrete and abstract methods
- Static, private, and final methods cannot be abstract
- When you extend an *abstract* class with *abstract* methods:
  1. Subclass does not override all abstract methods
     - Subclass must be declared abstract
  2. Subclass overrides all abstract methods
     - Subclass can be a concrete class
       - Can instantiate subclass
     - Or it can be declared abstract

Correct Animal Implementation

```
public abstract class Animal {
  private String _name;
  private Person _owner;

  public Animal(String n, Person p) {
    _name = n;
    _owner = p;
  }

  public abstract void makeNoise();
// …
}
```

```
public class Cow extends Animal {
  private String _color;

  public Cow(String n, Person p,
             String c) {
    super(n, p);
    _color = c;
  }
}
```

Compile this code.
What happens?

Compilation error in Cow. Why?

- Inherits an abstract method
- Does not implement it

- Solutions

1. Implement method in *Cow*
2. Or declare *Cow* as abstract

Any Problem?

```
public abstract class Animal {
  private String _name;
  private Person _owner;

  public Animal(String n, Person p) {
    _name = n;
    _owner = o;
  }
  public String getName() {
    return _name;
  }
  public abstract void makeNoise();
}
```

```
public class Dog extends Animal {
  private Vacine[] _vacine;

  public Dog(String n, Person p,
             Vacine[] v) {
    super(n, p);
    _vaccine = v;
  }
  public void makeNoise() {
    System.out.println("ão ão");
  }
}
```

Consider the following code:

```
Person p = new Person();
Vacine[] vacines = new Vacine[0];

Animal animal;
Dog dog = new Dog("dog", p, vacines);
System.out.printf("Nome: %s\n",
                  dog.getName());
dog.makeNoise();
animal = dog;
animal.makeNoise();
```

- Any compilation error?
- **No!**
- Any execution error?
- **No!**
- Result of execution:

```
Nome: dog
ão ão
ão ão
```

Polymorphism

- Another feature of OOP
- Literally “having many forms”
- Object variables in Java are polymorphic
- Can treat an object of a subclass as an object of its superclass
  - A reference variable of a given type can refer to objects of its own type or to objects of subtypes from its type
- When a method is invoked, which method body is executed?
  - Superclass version or
  - Subclass version

Polymorphism - Code Binding

- Binding: connecting a method call to a method body
- Two kinds of binding
  - Static binding
  - Dynamic binding
- Static binding (Early binding)
  - Binding is performed before the program is run
  - Based on the type of the **variable**
- Dynamic binding
  - Binding occurs at run time, based on the type of the **object** not on the type of the **variable**
  - Method body to be executed is determined at execution time, not compile time
- Java applies dynamic binding for non-static, non-final and non-private methods
  - Designated as **polymorphic** methods
- Java applies static binding for static methods and private or final non-static methods

Method Lookup

- It is a two-step process

1. To determine if a method is legal the compiler looks in the class based on the declared type
   - if it finds it great, if not go to the super class and look there
   - continue until the method is found, or the Object class is reached and the method was never found. (Compile error)
2. To determine which polymorphic method is actually executed by the run time system
   - starts with the actual run time **class of the object** that is calling the method
   - search the class for that method
   - if found, execute it, otherwise go to the super class and keep looking
   - repeat until a version is found

Attendance Question

What is output by the code to the right when run?

A. `!!live`

B. `!eggegg`

C. `!egglive`

D. `!!!`

E. `eggegglive`

```
public class Animal{
  public String bt(){ return "!"; }
}

public class Mammal extends Animal{
  public String bt(){ return "live"; }
}

public class Platypus extends Mammal{
  public String bt(){ return "egg";}
}

public static void main(String args[]) {
  Animal a1 = new Animal();
  Animal a2 = new Platypus();
  Mammal m1 = new Platypus();

  System.out.print( a1.bt() );
  System.out.print( a2.bt() );
  System.out.print( m1.bt() );
}
```

Polymorphism (continued)

- Operator `instanceof` - determines whether a reference variable that references an object of a particular type

- Example:

```
p instanceof BoxShape
```

- Evaluates to a boolean
  true if p refers an object of the class *BoxShape (or subclass)*
  false otherwise
- Should be **avoided**
  - Usually, it is not OOP
- **Allowed in**
  - **public boolean equals(Object)**

Type Compatibility

- Java is a *strongly typed* language.
- Compatibility
  - when you assign the value of an expression to a variable, the type of the expression must be compatible with the declared type of the variable: it must be the same type as, or a subtype of, the declared type
  - `null` object reference is compatible with all reference types.
- What about primitive types?

Type conversion - 1

- The types higher up the type hierarchy are said to be *wider*, or *less specific* than those lower down the hierarchy. Similarly, lower types are said to be *narrower*, or *more specific*.
- *Widening conversion*: assign a subtype to a supertype
  - Upcasting - It is safe
  - Can be checked at compile time. No action needed
- *Narrowing conversion:* convert a reference of a supertype into a reference of a subtype
  - Downcasting – Not safe
  - *M*ust be explicitly converted by using the *cast* operator
    - **(Type)**

```
Rectangle r = new Box();
Box b = (Box)r;
```

Type conversion - 2

- Explicit type casting: a type name within parentheses, before an expression
  - For upcasting: **not** necessary
  - For downcasting: **must** be provided

e.g.

```
String str = "test";
Object obj1 = (Object)str;
Object obj2 = str;
String str1 = obj1;
String str2 = (String)obj1;
Double num = (Double)obj1;
```

- If the compiler can tell that a narrowing cast is incorrect, then a compile-time error will occur
- If the compiler cannot tell, the run time system will check it. If the cast is incorrect, then a *ClassCastException* will be thrown

E.g. Student is subclass of Person

```
public class TypeTest {
   static Person[] p = new Person[10];

   static
   {
            for (int i = 0; i < 10; i++) {
               if(i<5)
                  p[i] = new Student();
               else
                  p[i] = new Person();
            }
   }

   public static void main (String args[]) {
            Person o1 = (Person)p[0];
            Person o2 = p[0];
            Student o3 = p[0];
            Student o4 = (Student)p[0];
            Student o5 = p[9];
            Student o6 = (Student)p[9];
            int x = p[0].getStudentNumber();
   }
}
```

```
%> javac TypeTest.java

TypeTest.java:17 incompatible types
found    : Person
required : Student
            Student o3 = p[0];
                         ^

TypeTest.java:19 incompatible types
found    : Person
required : Student
            Student o5 = p[9];
                         ^


TypeTest.java:21: cannot resolve  symbol
symbol   : method getStudentNumber ()
location:  class Person
         int x = p[0].getStudentNumber();
                      ^

3 errors
```

<u>After commenting out these three ill lines:</u>

```
%> java typeTest
Exception in thread "main"
      java.lang.ClassCastException: Person
            at typeTest.main(typeTest.java:20)
```

Why Bother with Inheritance?

- Inheritance allows programs to model relationships in the real world
  - if the program follows the model, it may be easier to write
- Inheritance allows code reuse
  - complete programs faster (especially large programs)
- Polymorphism allows code reuse in another way
  - Assign multiple meanings to the same method name
- Inheritance and polymorphism allow programmers to create *generic algorithms*

Subclasses and Superclass Contract

- Polymorphism implies that you can make code that is client of a given type
  - public void printAll(Animal[] animals) { … }
- and that code works well with instances of that type and instances of subtypes of that type
- This requires that subtypes must respect the contract of the superclass
  - The semantic of each overridden method is preserved
  - Each overridden method has an equal or less restricted access modifier
    - A public method of the superclass cannot be overridden as protected
    - But a protected method of the superclass can be overridden as public
- Otherwise, *is-a* relationship would be broken

Software Development - Design Stage

- Characterize classes that model the entities of the domain problem
  1. Fields
  2. Methods
  3. Relationships
- Some classes found to be closely related
  - Factor out common fields, behaviors
  - Design superclasses to store common characteristics
  - Use inheritance to develop subclasses with inherited capabilities
- Makes the code more flexible (see Polymorphism)

Design Hints for Inheritance

1. Place common operations and fields in the superclass
2. Try not to use `protected` fields
3. Use inheritance to model a Is-A relationship
4. Respect the contract of the superclass
   1. Don't use inheritance unless all inherited methods make sense
   2. Don't change the expected behavior when you override a method
5. Use polymorphism, not type information

```
if (x instanceof Type1)
        ((Type1)x).action1();
else if (x instanceof Type2)
        ((Type2)x).action2();
```

- Solution?

  Do action1 and action2 represent a common concept? If it is, make the concept a method of a common superclass or interface of both types, and then you can simply call `x.action()`.

Object-Oriented (OO) Paradigm

- *Everything* is an object
  - Each object has a state
  - Each object has a type
    - Defines the set of valid operations
- State of application is a graph of objects
- Each class encapsulates the data (fields) and functionality (methods) of the objects.
- When a method is called on an object, *"the object knows what to do on its own."* The caller doesn't need to know what to do.
- Object-Oriented
  - Think what the objects (data) are first
  - Then their functionality (methods)
  - Then how the objects and their functionality can be used to create more complex functionality
    - Reuse code