## CS102

Introduction to
data structures, algorithms, and object-oriented programming



## Inheritance & Class Hierarchies

The central new idea in object-oriented programming -- the idea that really distinguishes it from traditional programming -- is to allow classes to express the similarities among objects that share **some**, but not all, of their structure and behavior.

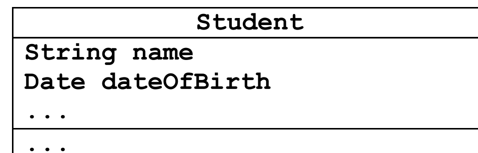
## has-a Relationship

```
class Student
{
   private String name;
   private Date dateOfBirth;
   ...
}
```

- a **`Student`** *has-a* **`String`** for **its** `name`
- a **`Student`** *has-a* **`Date`** for its `dateOfBirth`



## In UML Terms



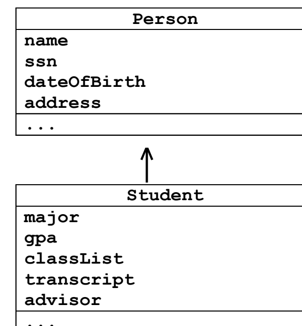



## Terminology for Inheritance

The term inheritance refers to the fact that one class can inherit part or all of its structure and behavior from another class. A subclass is created by using the extends keyword in the subclass signature.

- parent class ⟵ •child class
- superclass ⟵ •subclass
- base class ⟵ •derived class





## Syntax to Set Up Inheritance

- class Student **extends** Person

  Makes an object of Student class to also be an object of Person class.

7

## is-a Relationship

- a **`Student`** is-a **`Person`**
- Every **`Student`** object is also a **`Person`** object.
- Class **`Student`** inherits non-private data & methods from Class **`Person`**.

8

## Inheritance Promotes Software Reuse

- Start with a class to which you want to add fields/methods.
- Extend that class and add additional capabilities to the subclasses.
- Classes that are specified as final cannot be extended.

9

## Programmer Decisions

- What methods and variables of the parent class can the child class see?
- What if the child wants to add a method or variable *with the same name* as a method or variable of the parent? Method overloading

10

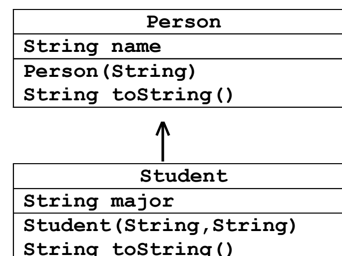
## Why Not Modify Existing Code...

...instead of subclassing?

- Source code may not be available to us and the implementation may be difficult to understand.
- Our changes could break the existing code.

11

## Smaller Example of subclassing




12

## Simple Person class

```
class Person
{
   protected String name;
   public Person ( String name )
   {    this.name = name;    }
   public String toString() { return name; }
}
```



```
public class Student extends Person {
   private String major;  // added field in subclass
   public Student ( String name, String major )
   {  super(name);  // call to Person constructor
      this.major = major;
   }
   public String toString()
   {
      return this.name + " (" + this.major + ")";
   }
}
```



- Use the parent's constructor to initialize parent variables (in a subclass, this will be a call to super() with zero or more parameters).
- Inside a subclass constructor, a call to the superclass constructor must be the first line. If not, there will be an error.



## Another Person class

```
class Person
{
   // name is non-accessible, even from subclasses
   private String name;
   public String toString()
   {
      return name;
   }
}
```



```
class Student extends Person
{
   private String major;
   public String toString()
   {
      return super.toString() + " (" + major + ")";
   }
}
```



## Object — The Top of All Hierarchies

Every class is a subclass of the Object class (i.e., extends the Object class). Inherited methods include:

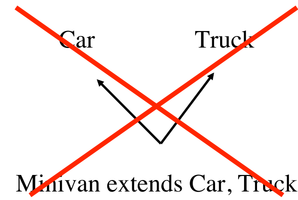
- public String toString()
- public boolean equals(Object)

## Overriding vs Overloading

- Don't confuse overriding with overloading.
- Can anyone tell me the difference?

19

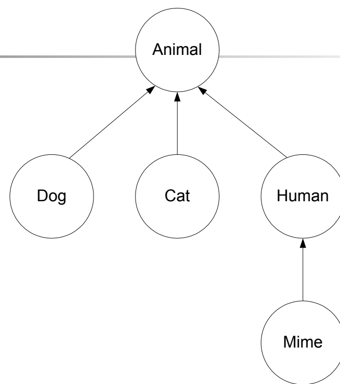
## Single vs Multiple Inheritance




Minivan extends Car, Truck

- Java allows only single inheritance, i.e., a subclass can extend only one superclass.

20




21

class Animal

class Dog extends Animal

class Cat extends Animal

class Human extends Animal

class Mime extends Human

- *Each has*
  public void speak()

22

**public void speak()** in each class contains the single print statement:

- *Animal:*
  System.out.println("* generic animal noise *");
- *Dog:*
  System.out.println("woof");
- *Cat:*
  System.out.println("meow");
- *Human:*
  System.out.println("hello");
- *Mime:*
  System.out.println();

23

```
Animal [] animals = new Animal[MAX];
animals[0] = new Dog();
animals[1] = new Cat();
animals[2] = new Human();
animals[3] = new Dog();
animals[4] = new Mime();
animals[5] = new Cat();
animals[6] = new Animal();
for ( int i = 0 ; i < MAX ; i++ )
{
  animals[i].speak();
}
```

24

## Motivation for Abstract Classes

- We shouldn't instantiate `Animal` —
  The class is too generic!
- Let's make `Animal` an *abstract class*.



```
abstract class Animal
{
//-----------------------------------------------------
   abstract public void speak();
//-----------------------------------------------------
}
```



```
Animal [] animals = new Animal[MAX];
animals[0] = new Dog();
animals[1] = new Cat();
animals[2] = new Human();
animals[3] = new Dog();
animals[4] = new Mime();
animals[5] = new Cat();
animals[6] = new Cat();
```



## Interfaces

- Java only allows *single* inheritance, so a subclass can only have one superclass, including abstract classes.
- Interfaces can be used to achieve most of the effects of multiple inheritance. A class that implements an interface is a subtype of the interface.



## Interfaces

Interfaces are like abstract classes, but they can contain no method bodies, only method *stubs*.

Classes that *implement* an interface are required to provide full method bodies for each method stub in the interface.



## Interfaces

Interfaces can be a data type for a declared variable, but the instantiated type must be a subtype of the interface.

Classes can implement any number of interfaces and they must have method bodies for all method stubs in the interface.

## Interfaces

Because of the inheritance model Java provides, an object can have multiple data types.



## Structural recursion in Java

Recall the definition of a list in Scheme:

A list of integers (IList) is either:

- empty, or it is a
- constructed list, formed by using (cons element LON), where LON is an IList.

We can make a similar structure in Java using an Interface:

```
public interface IList {
    // stub for method to return length of this IList
    public int length();
    // stub for method to sum all integers in this IList
    public int sum();
}
```



## Structural recursion in Java

A subtype of interface IList to represent a constructed list:

```
public class ConsList implements IList {
    int first;
    IList rest;
    // fully implemented method returns length of this IList
    public int length() {
        return 1 + this.rest.length();
    }
    // fully implemented method returns sum of numbers in this IList
    public int sum() {
        return this.first + this.rest.sum();
    }
}
```



## Structural recursion in Java

A subtype of interface IList to represent an empty list:

```
public class MTList implements IList {
    // implementation of length method for empty list
    public int length() {
        return 0;
    }
    // implementation of sum method for empty list
    public int sum() {
        return 0;
    }
}
```