Object Oriented Programming
Part II of II

Steve Ryder
08/23/05, Session 8352
JSR Systems (JSR)
sryder@jsrsys.com

Interfaces

- Declaring an interface:

```
public interface Student {
    public abstract void checkCredits();
}
```

- Compiler won't let you instantiate an interface.

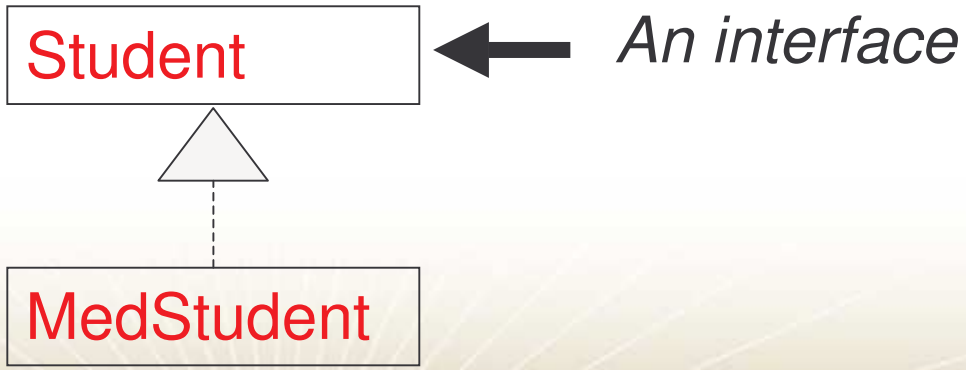
```
Student s;
S = new Student;
```

- Using an interface.

```
Class MedStudent extends Student {
    public void checkCredits();
}
```

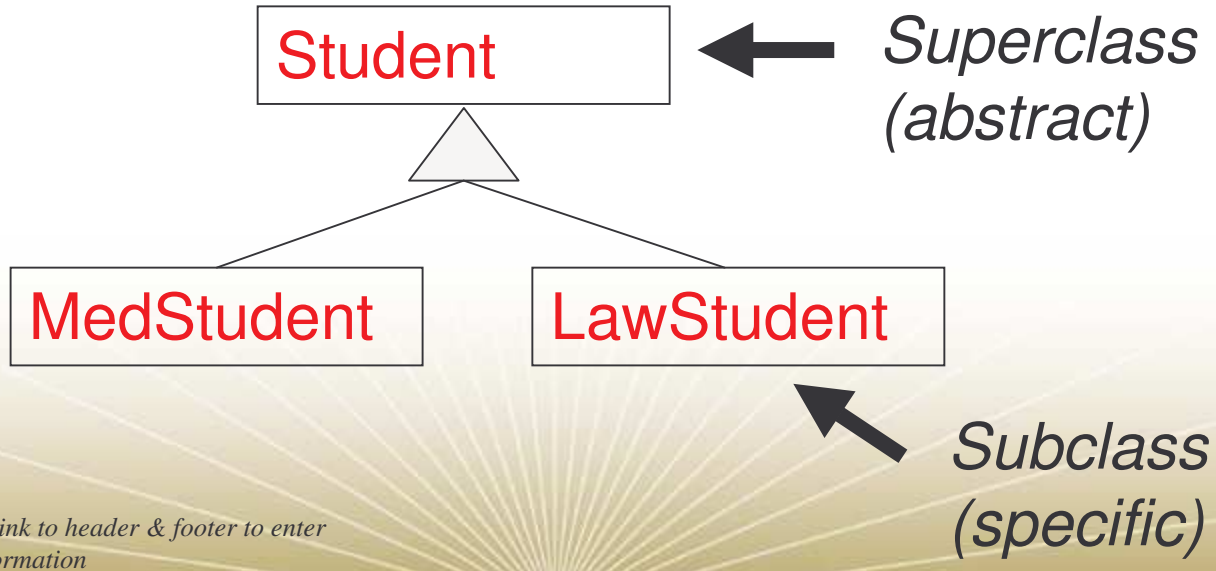
Interfaces – A Closer Look

- An interface is a variation of a class.
- An interface provides only a definition of functionality.
- A separate class actually implements that functionality.
- Use an interface when you want programmers to be able to use only the specific class and not the general one.



Inheritance

- Put common code in a class called a superclass. This is the more abstract class.
- The more specific class is called a subclass.
- A subclass extends the superclass.



Inheritance – A Closer Look

- When abstraction and inheritance is used correctly, other programmers will be able to add new 'kinds' of classes to the program at any time. Some rules to keep in mind:
  - DO - when one class is a more specific type of a superclass.
  - DO – when you have behavior that should be shared among multiple classes of the same general type.
  - DON'T – just so you can reuse code from another class if the relationship violates either of the two rules above.
  - DON'T – if the subclass and superclass do not pass the IS-A test

Inheritance – A Closer Look

Five steps to inheritance:

1. Look for objects that have common attributes and behaviors.

2. Design a class to represent common behaviors.

3. Decide if a subclass needs specific behaviors.

4. Look for more opportunities to use abstraction by finding two or more subclasses that might have common behavior.

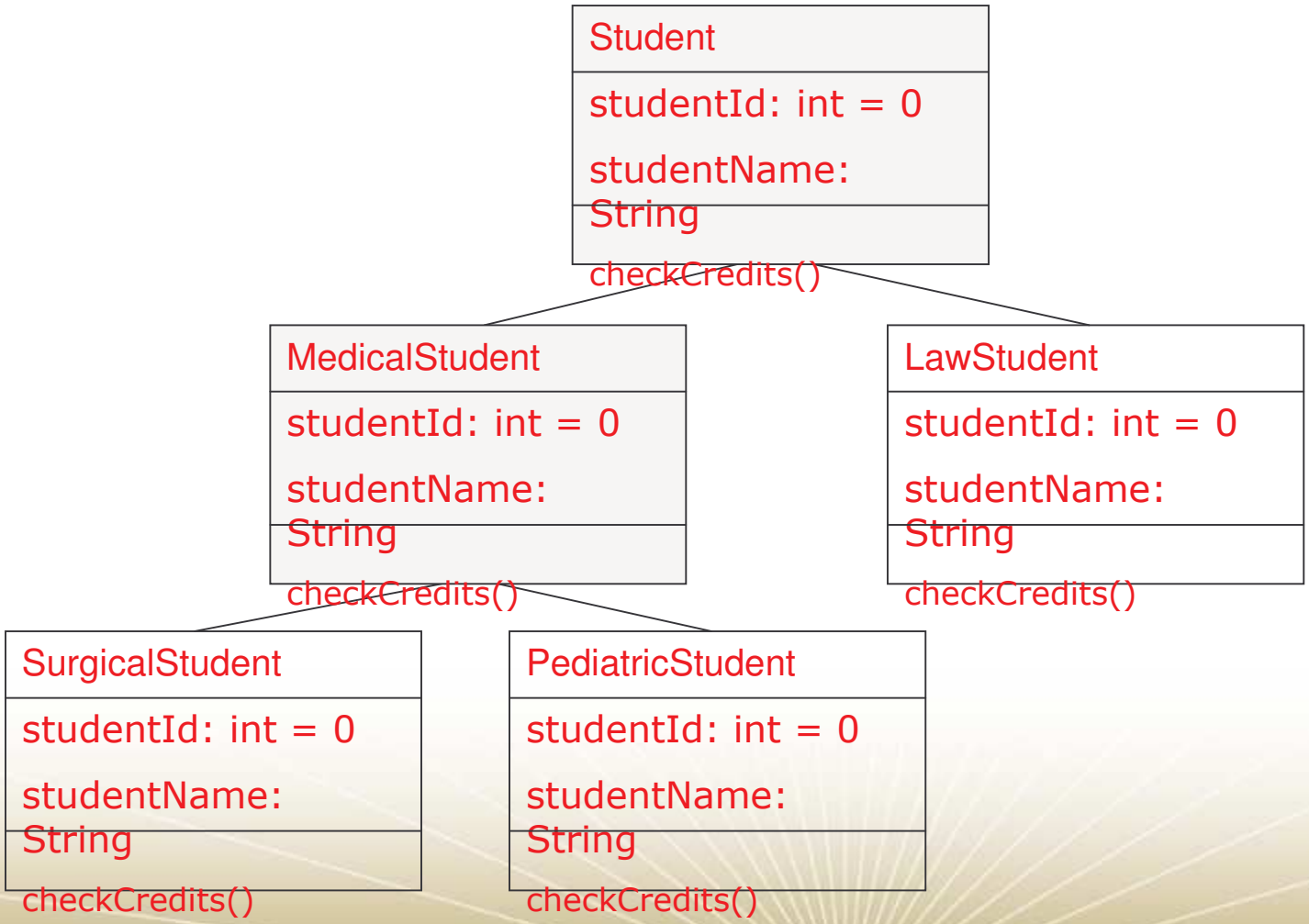
5. Finish the class hierarchy.

Inheritance – A Closer Look

Overriding methods –

- When you design superclasses and subclasses, there may be times that the subclass will have the same method as the superclass. This is called overriding.
- In this situation, the method for the lowest class (the most specific class) gets called.

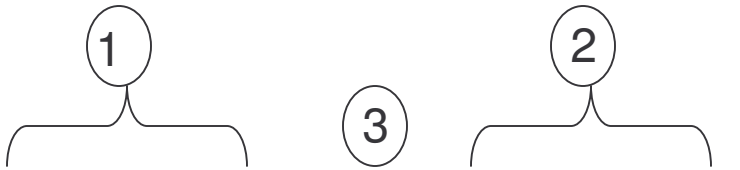
Inheritance – A Closer Look



Polymorphism

Three steps of object declaration and assignment

1. Declare a reference variable



2. Create an object
3. Link the object and the reference

Polymorphism

Power references

- The reference and the object can be different

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

Polymorphic Arrays

- Power references allow you to make polymorphic arrays

```
Animal [ ] animals = new Animal[5];
animals[0] = new Dog();
animals[1] = new Cat();

for (int I = 0; I < animals.length; I ++) {
    animals[i].eat();
    animals[i].roam();
}
```

Arguments/Return Types

You can use polymorphic arguments and return types….

```
class Vet {
    public void giveShot(Animal a) {
        //do vet stuff
        a.makeNoise;
    }
}
```

Overloading

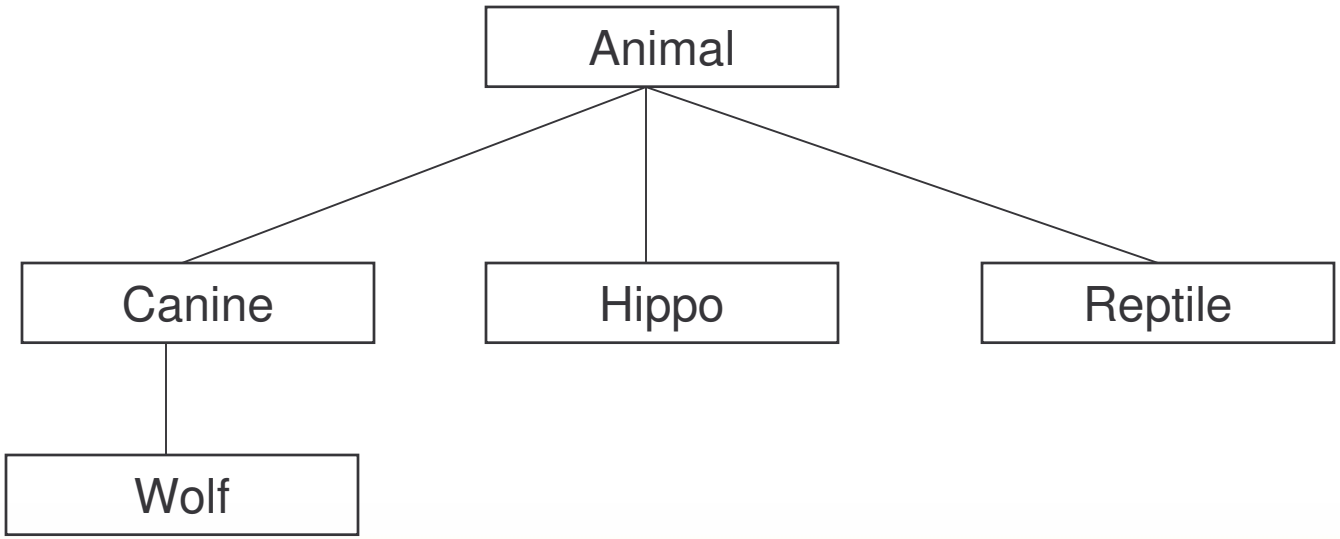
- Overloading is having two methods with the same name but different arguments.
- Overloaded methods have great flexibility:
  1. Return types can be different as long as the arguments are different types.
  2. The return type can not be the only thing changed.
  3. You can vary the access levels in any manner.

Abstract Classes

- Keep duplicate code to a minimum.
- Override generic methods.
- Flexible because of Animal subtypes that can be designed in the future and used in any method expecting an Animal object as an argument.
- Creates a common protocol for all animals that are related to the Animal superclass.

Abstract Classes

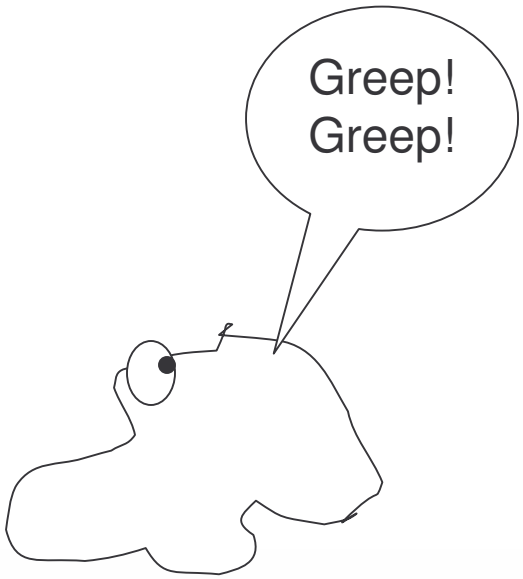
Sample Animal class hierarchy



Abstract Classes

Given the class design on the previous slide, the following declarations are valid:

```
Animal aHippo = new Hippo();
Canine aWolf = new Wolf();
Wolf aWolf = new Wolf();
```



But what about this?

```
Animal anim = new Animal();
```

What would an Animal object look like?

Abstract Classes

- The Animal class is necessary for the inheritance and polymorphism we've been covering. However …
  - Programmers should only be able to instantiate the more concrete subclasses like Wolf or Hippo because those have shapes, sizes, and behaviors that are well-defined.
- To stop a class from being instantiated, make the class abstract.

```
abstract class Animal

abstract class Canine extends Animal
```

Abstract Methods

- An abstract method must be overridden.
- An abstract method has no body.

```
public abstract void eat();
```

- If you declare a method as abstract, you must declare the class abstract as well.

Abstract Methods

- What can an abstract method be used for?
  - The point of an abstract method is that even without any actual code, you still have defined part of the protocol for a group of subclasses.

Abstract Methods

- What if there are two abstract classes in the hierarchy?
  - A subclass can ‘pass the buck.’
  - If Animal and Canine are both abstract, the first concrete class to extend Canine must implement all abstract methods from both Animal and Canine.

Review

- Abstract classes and methods are useful for keeping duplicate code to a minimum while maintaining a protocol for a group of classes.
- An abstract class can not be instantiated. This forces the programmer to instantiate only the more specific (or concrete) subclasses.
- Abstract methods define the behaviors that all subclasses must have. Each subclass has its own unique way to implement the behaviors.
- The first concrete class in the hierarchy (Wolf from Canine and Animal) must implement all methods from both Canine and Animal.

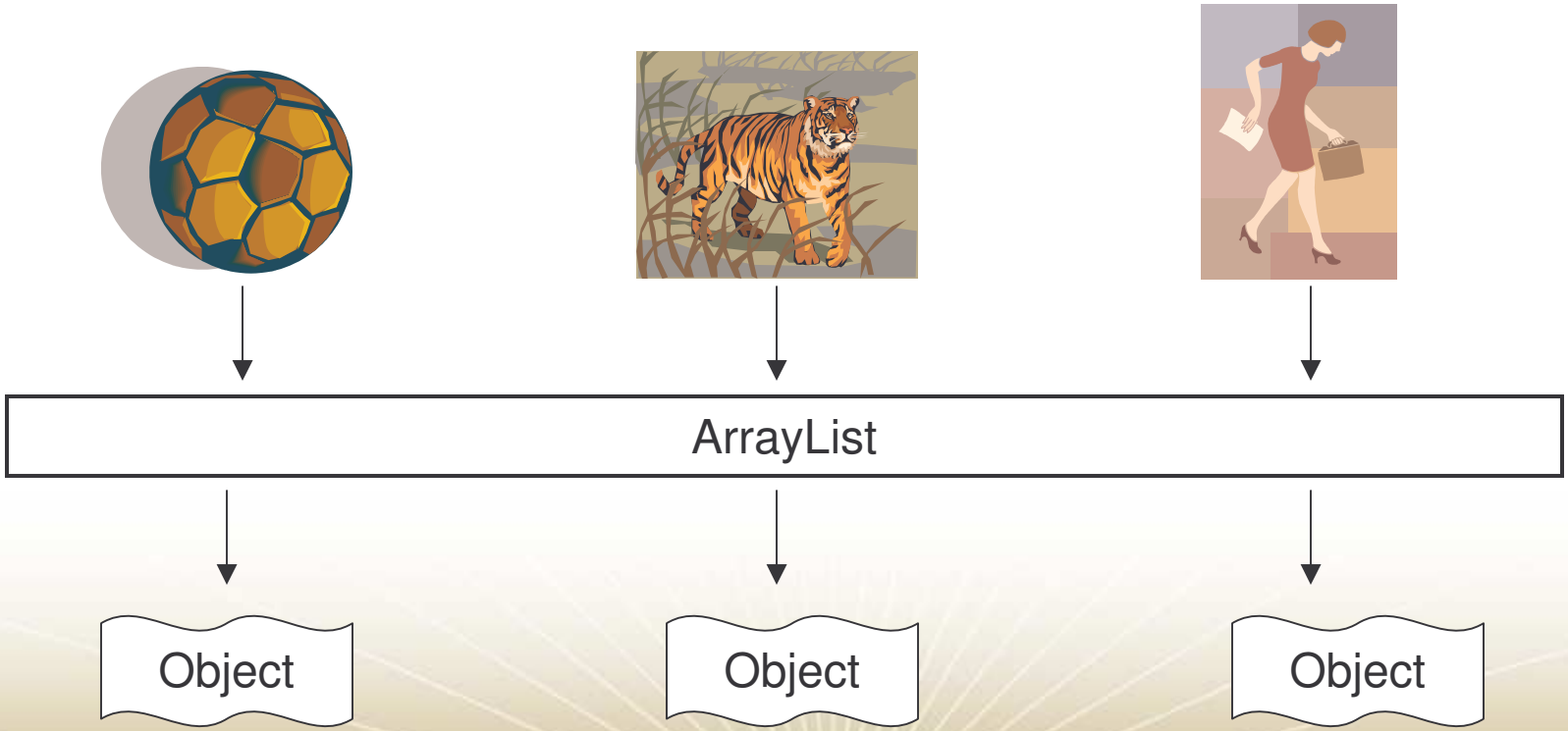
The Dot Operator

- The Dot operator (.) gives you access to an object's state and behavior.

```
//Make a new Object
Dog d = new Dog();
//Call the Dog's bark method
d.bark();
//Set the size of the Dog
d.size = 40;
```

Objects in an ArrayList

- Objects come out of an ArrayList acting like they are generic objects.



Objects in an ArrayList

1. `ArrayList al = new ArrayList();`
2. `Tiger t = new Tiger();`
3. `al.add(t);`
4. `//Make Tiger in ArrayList growl`

1) Instantiate ArrayList object
2) Instantiate Tiger object
3) Add Tiger object to ArrayList

Objects in an ArrayList

1. ArrayList al = new ArrayList();
2. Tiger t = new Tiger();
3. al.add(t);
4. //Make Tiger in ArrayList growl

Can you call the Tiger's makeNoise method here?

No. Only the methods in the Object class are available at this point.

Casting an Object Reference Back to its Real Type

1. Object o = al.get(index);
2. Tiger t = (Tiger) o;
3. t.makeNoise();

1) Get the object from ArrayList

2) Generic object ‘o’ is casted to a Tiger object and assigned to the ‘t’ reference variable

3) The makeNoise() method of the Tiger class is called

Review

- Objects go into an ArrayList as the specified type but come out as generic objects.
- To access the methods of the specific type, you must cast the object to the specific object.

Pet Shop Program

- What if the Dog class that was written for any type of dog was needed as a pet in another program?
- The Dog class would need new pet-oriented methods such as play(), sit(), rollover(), etc..
- Let's review three design options to make this happen…

Pet Shop – Design Option 1

- Put pet methods in Animal class.
- Pros
  - All Animals instantly inherit pet behaviors.
  - We won't have to touch existing Animal subclasses.
  - Any Animal subclass created in the future will get the pet methods.
  - Any program wanting to treat animals as pets can use the Animal class as a polymorphic argument or return type.
- Cons
  - ALL animals inherit pet behaviors even lions, tigers, and bears – oh, my!
  - There are sure to be changes required to the subclasses like Dog and Cat because they would implement pet behaviors very differently.

Pet Shop – Design Option 2

- Put pet methods in the Animal class but make the methods abstract, forcing the subclasses to override them.
- Pros
  - All the benefits of option1 are realized plus there would be no unwanted animals with pet attributes.
  - The abstract methods that must be overridden can be empty.
- Cons
  - Every subclass of Animal would have to have pet methods even if they aren't needed.
  - The existence of Pet methods in the subclasses would be misleading as pet behaviors would be expected from those methods.

Pet Shop – Design Option 3

- Put the pet methods only in the classes where they belong.
- Pros
  - The pet methods are only where they belong.
- Cons
  - There is no way for other programmers to know what the protocol for establishing or using pet behaviors and no way for the compiler to make sure pet-like methods are implemented correctly.
  - The Animal class could not be used as the polymorphic type because the compiler will not let you call a pet method on an Animal reference.

Pet Shop – Best Design

- Create two superclasses: Animal and Pet.
- Give the Pet class all of the Pet methods.
- Have subclasses that should use Pet methods extend both the Animal and Pet classes.