



CS 112 – MiraCosta College

Introduction to Computer Science II Java

Module 3 – Abstract Classes

Chris Merrill

Computer Science Dept
cmerrill@miracosta.edu

Agenda

- Review – Polymorphism
 - Homework for Module 1
 - Algorithms Exercise
 - Polymorphism
 - Abstract Classes
- Quiz 2 – Polymorphism and Abstract Classes
- Polymorphic arrays and parameters
- The clone() method and copy constructors
- Geometric Figures Lab

Introduction to Polymorphism

- Polymorphism is the ability to associate many meanings with one method name
 - Actually, it references both the name and parameter list, otherwise known as the method's *signature*
- Polymorphism allows changes to be made to method definitions in the derived classes, *and have those changes apply to the software written for the base class*

How does Polymorphism Work?

- For polymorphism to work, a variable of the base class type can be assigned to an object of a derived class.

```
PersonAtMCC susan = new Instructor(...)
```

- This does NOT work the other way around:

```
Instructor jose = new PersonAtMCC(...)
```



Won't work!

How does Polymorphism Work?

Conversely, many times we can use a variable of a derived class when we normally would use a variable of a parent class, for example, a copy constructor:

```
Instructor joe = new Instructor(...);
```

```
PersonAtMCC jack = new PersonAtMCC(joe);
```



Copy constructor

Late Binding

- The process of associating a method definition with a method invocation is called *binding*
- If the method definition is associated with its invocation when the code is compiled or loaded, that is called *early binding*
- If the method definition is associated with its invocation when the method is invoked (at run time), that is called *late binding* or *dynamic binding*

Late Binding

- Because of late binding, a method can be written in a base class to perform a task, even if portions of that task aren't yet defined
- Java uses late binding for all methods *except* **final**, **private**, and **static** methods
- Let's look at examples of polymorphism

Upcasting and Downcasting

- *Upcasting* is when an object of a derived class is assigned to a variable of a base class

```
Animal anAnimal ;           //Parent class
Cat felix = new Cat() ;     //Child class
anAnimal = felix ;          //Upcasting
System.out.println(anAnimal.toString());
```

- Because of late binding, the `toString` method above uses the definition in the `Cat` class

Upcasting and Downcasting

In fact, upcasting works when assigning an object of a descendant class to a variable of *any* ancestor class:

```
Object anObject ;           //Ancestor class
Cat felix = new Cat() ;     //Descendant class
anObject = felix ;          //Upcasting
System.out.println(anObject.toString());
```

Upcasting and Downcasting

Downcasting is when a type cast is performed from a base class to a derived class (or from any ancestor class to any descendent class)

- Downcasting has to be done very carefully
- In many cases it doesn't make sense (run-time error), or is simply illegal (compiler error):

```
cat felix = (Cat) anAnimal ; // run-time error?  
cat felix = new Animal() ;   // compiler error
```

Downcasting Problems

Run-time error:

```
cat felix = (Cat) anAnimal ;
```

- Will run properly if `anAnimal` references a `Cat` or any descendant object
- Won't work if `anAnimal` references `Dog` object

Compiler error:

```
cat felix = new Animal() ;
```

- No way this is going to work...

Downcasting

It is the responsibility of the programmer to use downcasting only in situations where it makes sense

- The compiler *does* check if a variable of an ancestor class is being assigned to a new descendant class object (a syntax or compiler error).
- The compiler does *not* check to see if downcasting is a reasonable thing to do (possibly resulting in a run-time error).

How to Check if Downcasting is OK

Downcasting to a specific type is sensible only if the object being cast is an instance of that type

- This is what the `instanceof` operator tests:

`object instanceof ClassName`

- It returns `true` if `object` is of type `ClassName`
- It also returns `true` if `object` is an instance of any *descendent* class of `ClassName`

Variables and Objects Knows the Definitions of Their Methods

- The *type* of a class variable determines the *method names* which can be used with that variable
 - The Java compiler cannot see method signatures in descendant classes that are not in that class
- The *object named by the variable* determines *which definition* with the method name is used during run-time
 - Methods which are overridden in descendant classes cause the Java Virtual Machine to use polymorphism to determine which definition will be used.

Downcasting

When is downcasting is necessary?

- A good example of downcasting is in our version of the `equals` method for a class:

```
Employee staff = (Employee) anObject ;
```

- Remember, though, that we first verified that `anObject` was an `Employee` object by using `anObject.getClass()` (see next slide).

Our equals Method for Employee

```
public boolean equals(Object anObject) {  
  
    if (anObject == null) {  
        return false ;  
    }  
    if (getClass() != anObject.getClass()) {  
        return false ;  
    }  
    Employee otherEmployee = (Employee) anObject;  
    return (name.equals(otherEmployee.name) &&  
            hireDate.equals(otherEmployee.hireDate)) ;  
}
```


Downcasting Explanation

In each version of `equals`, an `Object` parameter is used in the method heading

- There are no instance variables called `name` or `hireDate` in the `Object` class, so the compiler can't associate them with the `anObject` variable.
- When `anObject` is downcast to `otherEmployee` (a variable of type `Employee`) the compiler now sees those instance variables – as well as any other methods available in the `Employee` class

Early Binding for **private**, **final**, and **static** Methods

Java uses static (or early) binding with **private**, **final**, and **static** methods

- Methods which are **private** are not available outside of the class in which they are defined, and therefore are not inherited and cannot be overridden (though they *can* be redefined)

```
private void someMethod() { ... }
```

Early Binding for **static** Methods

The argument *for* using early binding for static methods is that the method should be invoked using the Class name, not an instance variable.

- If **bark** is a **static** method in the **Dog** class, then **Dog.bark** should be used, not **fido.bark**

Of course, the argument against this is why should Java allow instance variables to invoke static methods at all?

Derived vs. Base Class Members

- A derived-class object has all of the instance variables and constants of a base-class object
- It also inherits all methods from its base class which aren't private. Since these are essentially “helper” methods, we can ignore these.
- So we can say that the capabilities of a derived-class object include those of a base-class object.

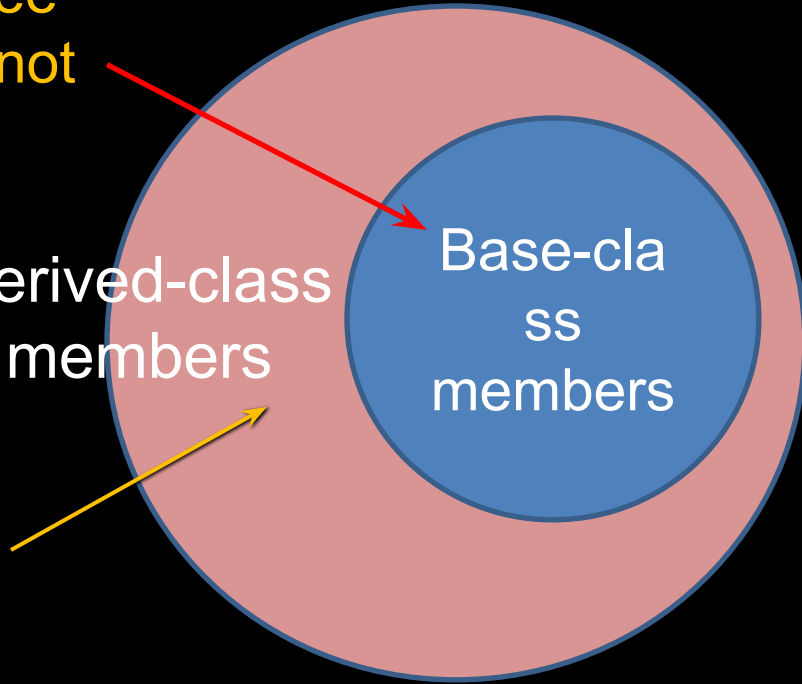
Derived vs. Base Class Capability

A base-class variable will see only these members. It cannot see derived-class data.

Derived-class members

Base-class members

A derived-class variable will see *all* members



Polymorphism Is One-Way Only

Is it ok to assign a base-class variable to a derived-class object (upcasting)?

- Yes, though the compiler will not have visibility to those members that are unique to the derived class?

Is it ok to assign a derived-class variable to a base-class object (downcasting)?

- Not always, as the Java compiler may think it has visibility to members that don't exist in the base-class

When Does Downcasting Make Sense?

Polymorphism allows us to “upcast” a derived-class object to a base-class variable:

```
Animal mammal = new Dog(...);
```

We can then “downcast” back to a derived-class variable to access all of the capabilities of the `Dog` class which are missing from the `Animal` class

```
Dog fido = (Dog) mammal;
```

What would happen if we had instantiated a `Cat` object?

Polymorphism and Method Parameters

Since derived-class objects have all of the members of base-class objects, they can be substituted for base-class arguments to a method.

```
Boat titanic = new Boat(. . .) ;  
Vehicle transport = new Vehicle(titanic) ;
```

Here the **Vehicle** copy constructor will use only those members in the **Boat** object referenced by the **titanic** variable which are inherited from the **Vehicle** class.

Questions?

Which of the following statements are true?

- An **Object** class variable can be assigned to *any* object in Java.

true when using the **new** operator to instantiate an object

true when using an assignment statement to assign to an existing object

*The Java compiler will see only the methods in the **Object** class, though.*

Questions?

In our definition of the `equals` method, we use a parameter of type `Object`. This means that we can pass *any* object of *any* type to this method.

Always `true`

We can compare Dogs to Cats, Rectangles to Students, Cars to Wombats... (Fortunately, these will all return false)

Questions?

An object of a descendant class can be upcast to an ancestor class variable.

Always **true**

However, the Java compiler will see only the methods in the ancestor class, as it looks only there to verify the existence of a method's signature

Methods in descendant classes are not available

Questions?

An object of a ancestor class can be downcast to a descendant class variable

true only if the variable is the same type as the descendant object (or one of its descendants).


The Java compiler will see all the methods in the descendant class.

Declaring An Abstract Class

A class that has one or more abstract methods must be declared an *abstract class*

It also must have the modifier **abstract** included in its heading:

```
public abstract class Employee {  
    . . .  
    public abstract double getPay() ;  
    . . .  
}
```



A diagram consisting of a brown rectangular box with the word "Mandatory" in yellow text. Three yellow arrows originate from this box: one points to the word "abstract" in the class declaration line, another points to the word "abstract" in the method signature line, and a third points to the semicolon at the end of the method signature line.

Abstract Class

- An abstract class can have any number of abstract and/or fully defined methods, including none
- If a derived class of an abstract class adds to or does not define all of the abstract methods, then it also is abstract, and must add **abstract** to its modifier
- A class that is not abstract (and therefore has no abstract methods) is called a concrete class

An Abstract Class Cannot be Instantiated

An abstract class can be used *only* to derive more specialized classes

- For example, while it may be useful to discuss employees in general, in reality an employee must be either a salaried worker or an hourly worker
- The Java compiler will not allow the name of an abstract class to follow the **new** operator

Should All Base Classes be Abstract?

This does *not* mean that all base classes should be made abstract

- We started by defining inheritance by creating classes that are more “generalized” version of their children
- What if we look in the other direction derived more “specialized” versions of parent classes

Some Base Classes are Concrete!

For example, a **Doctor** class, adequate for describing general practitioners and primary care physicians, could have many specialized descendants, such as:

- Cardiologist
- Psychiatrist
- Dermatologist
- Ophthalmologist
- Anesthesiologist
- Pathologist
- Obstetrician
- Podiatrist
- Plastic Surgeon
- Orthopedist

Abstract Classes Have Constructors, Too

We know that an abstract class constructor cannot be invoked to create an object of the abstract class

- However, a derived class constructor will include an invocation of the abstract class constructor in the form of **super**
- The abstract class constructor, like other base class constructors, is called either implicitly or explicitly.

Can Abstract Classes have Instance Variables?

Yes, of course.

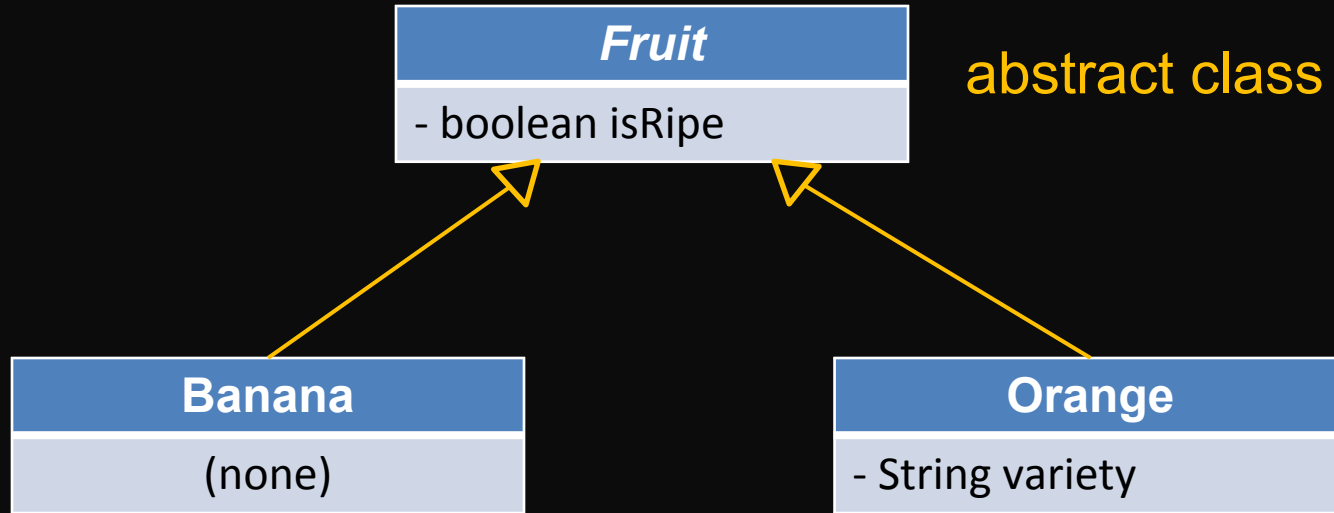
- Static and instance variables are inherited from abstract parents just like non-abstract parents
- They should be accessed and modified by getters and setters, again just like using non-abstract parents
- They also can be set by a call to **super** in the constructors of subclasses

An Abstract Class Is a Type

Although an object of an abstract class cannot be created, it is legal to declare a variable of an abstract class type

- However, it can reference only objects of its concrete descendent classes
- It's also ok to have a parameter of an abstract class type
 - Polymorphism makes it possible to reference an object of any of its descendent classes

Fruits (Oranges, Bananas, Strawberries)



An Example of an Abstract Class

The base (parent) class will be named **Fruit**

- It will contain a single instance variable: **isRipe**
- It also will contain an **abstract** method named **prepare**
- Since it contains an **abstract** method, the class itself also must be declared **abstract**!

An Example of an Abstract Class

The derived classes will be specific types of fruit, namely **Banana** and **Orange**.

- In addition, the **Orange** class will have another instance variable named **variety**.
- Java will force each subclass to implement a version of a **prepare** method, because **prepare** is declared **abstract** in the parent class.

(FruitDemo)

A First Look at the `clone` Method

Every object inherits a method named `clone` from the class `Object`

- The method `clone` has no parameters
- It should return a *deep* copy of the calling object
- The *inherited* version of the method was not designed to be used as is
 - Instead, each class is expected to *override* it with a more appropriate version

A First Look at the `clone` Method

The heading for the `clone` method defined in the `Object` class is as follows:

```
protected Object clone()
```

- The heading for a `clone` method that overrides the `clone` method in the `Object` class can differ somewhat from the heading above...

A First Look at the `clone` Method

- Changes to the heading of the `clone` method
 1. A change to a more permissive access, such as from `protected` to `public`, is always allowed when overriding a method definition
 2. Changing the return type from `Object` to the type of the class being cloned is allowed because every class is a descendent class of the class `Object`
(This is an example of a covariant return type)

A First Look at the `clone` Method

If a class has a copy constructor, the `clone` method for that class can use the copy constructor to create the copy returned by the `clone` method

```
public Dog clone() {  
    return new Dog(this) ;  
}
```

Note that copy constructors should *always* make deep copies of the objects being copied

Invoking the `clone` Method

Using the clone method of the `Dog` class from the prior slide, we can make a clone of a `Dog` object using the following code:

```
Dog fido = new Dog("Fido", "Collie", ...) ;  
Dog anotherFido = fido.clone() ;
```

How does this differ from a copy constructor?

Older `clone` Methods Return An `Object`

Prior to version 5.0, Java did not allow covariant return types, so no changes to the return type of an overridden method were allowed

- Since the return type of the `clone` method of the `Object` class is `Object`, the `clone` method for all classes had `Object` as its return type

Older `clone` Methods Return An `Object`

Pre-5.0 Java, the `clone` method for the `Dog` class would have looked like this:

```
public Object clone() {  
    return new Dog(this) ;  
}
```

- Therefore, the result always would have to be typecast back to the correct class type

```
Dog newFido = (Dog) oldFido.clone() ;
```

Limitations of Copy Constructors

Although the copy constructor and `clone` method for a class appear to do the same thing, there are cases where only a `clone` will work

- If we pass an object of a descendant class to a copy constructor of an ancestor class, the copy constructor will return an object of the *ancestor* class!

```
Animal newDog = new Animal(oldDog) ;
```

- The above example, the copy constructor returns an `Animal` object, not a `Dog` object!

Limitations of Copy Constructors

If the `clone` method is used instead of a copy constructor, then (because of late binding) a true copy is made, even from objects of a derived class:

```
Animal newDog = oldDog.clone() ;
```

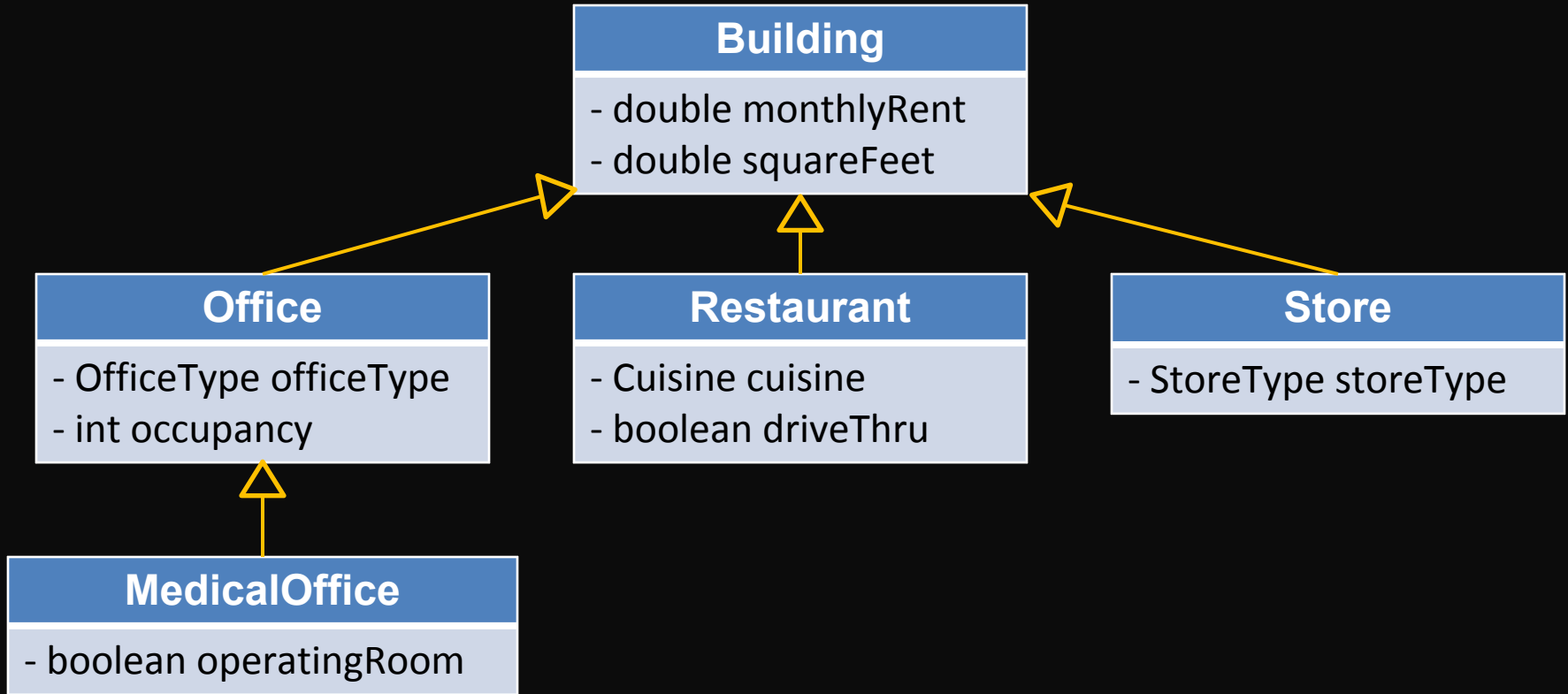
- This works because the method `clone` has the same name and parameter list (i.e., signature) in both classes, so polymorphism will work!
- The copy constructors for `Animal` and `Dog` have *different* names, so polymorphism doesn't apply.

Polymorphic Arrays

An array of a base type can contain objects of the base type or any of its descendants

- In the next demo, we'll create objects of several descendant types in one array
- We'll then use the copy constructor of the base class and see what type of objects it creates.
- We'll do the same thing using a **clone** method and compare the results

Buildings (Offices, Restaurants, and Stores)



The **clone** Method vs. Copy Constructors Using Several Types of Buildings

An “ah-ha” moment came when I saw a
demo similar to this one.

(Building)

Results

Copy constructors are not polymorphic, because there is only one method with the exact signature of the copy constructor

`clone` methods are polymorphic, as there are several methods with the `clone()` signature, so Java is able to pick the most appropriate one based on the object's type, *not* the copy constructor's name.

Introduction to Abstract Classes and Methods

In Chapter 7, the `Employee` class and its two derived classes, `HourlyEmployee` and `SalariedEmployee`, were defined

- We tried to add to the `Employee` class, which compared two employees to see if they have the same pay:

```
public boolean samePay(Employee other) {  
    return (getPay() == other.getPay());  
}
```

Introduction to Abstract Classes and Methods

There is a big problem with this method:

- There are `getPay` methods in each of the derived classes
- There is no `getPay` method in the `Employee` class, nor is there any way to define it without knowing whether the employee is hourly or salaried
- The compiler won't allow this approach to work

Introduction to Abstract Classes and Methods

The ideal situation would be if there were a way to

- postpone the definition of a `getPay` method until the type of the employee were known (i.e., in the derived classes)
- leave some kind of note in the `Employee` class to indicate that it was accounted for
- Java allows this using *abstract classes and methods*

Abstract Methods

- In order to postpone the definition of a method, Java allows an *abstract method* to be declared
 - An abstract method has a heading, *but no body*
 - Instead, the body of the method *must* be defined in at least one of the derived classes
- The class that contains an abstract method is called an *abstract class*

Abstract Methods

- An abstract method is a placeholder for a method that must be fully defined in a descendent class
 - Therefore, it cannot be **private**
- It has a complete method heading, to which has been added the modifier **abstract**
- It has no method body, and ends with a semicolon in place of its body

Homework

- Complete Lab 3 – Abstract Geometric Figures
- Complete all prior lab and homework assignments
- Read chapter 9 – Exception Handling
- Homework Module 3, projects 1, 2, and 3, and turn in your results at the beginning of next class

Polymorphism & Abstract Class Lab

Create an abstract parent class named **Figure**, and concrete child classes describing **Triangle**, **Rectangle**, **Square**, and **Circle** objects.