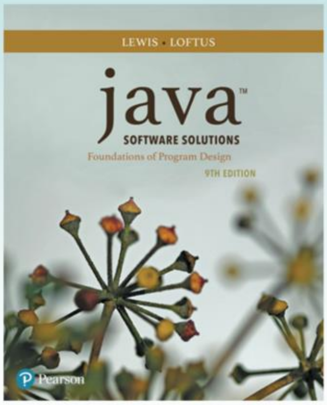
Chapter 9
Inheritance

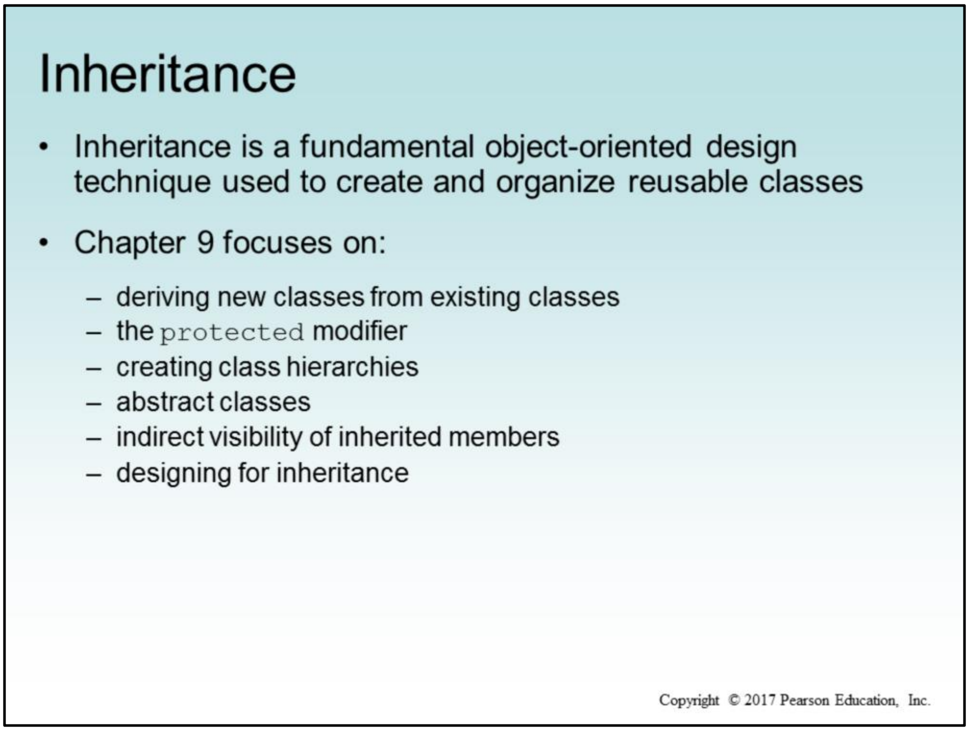


Java Software Solutions

Foundations of Program Design

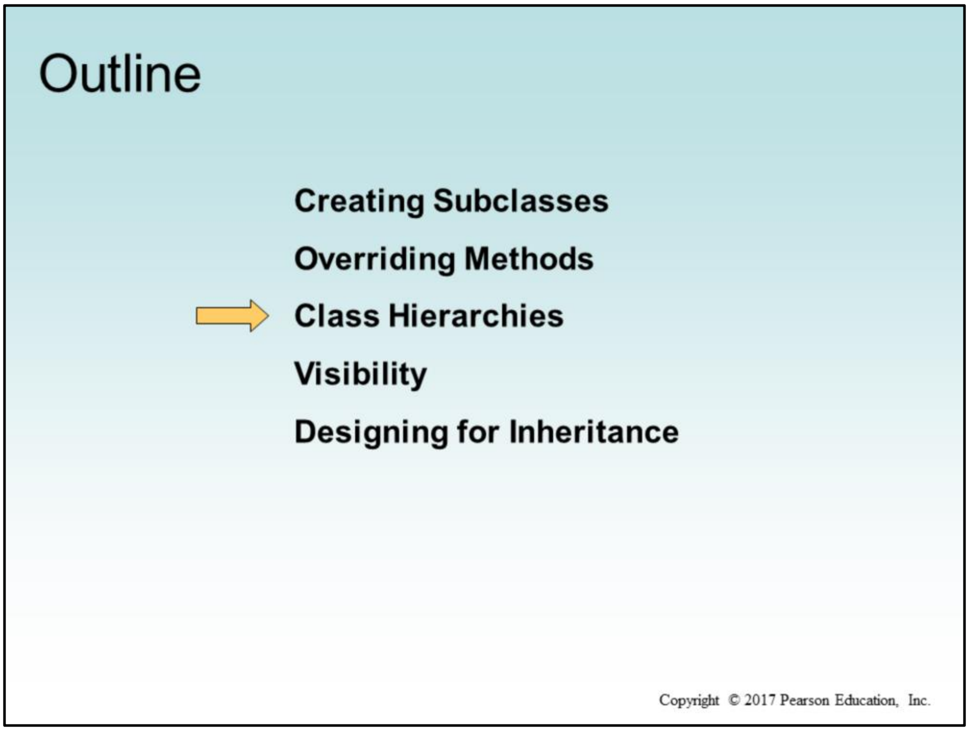
9th Edition

John Lewis
William Loftus



Inheritance

- Inheritance is a fundamental object-oriented design technique used to create and organize reusable classes
- Chapter 9 focuses on:
  - deriving new classes from existing classes
  - the `protected` modifier
  - creating class hierarchies
  - abstract classes
  - indirect visibility of inherited members
  - designing for inheritance



Outline

Creating Subclasses

Overriding Methods

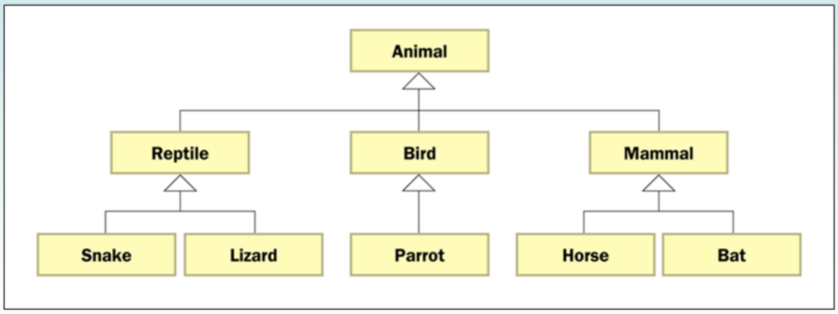
Class Hierarchies

Visibility

Designing for Inheritance

Class Hierarchies

- A child class of one parent can be the parent of another child, forming a *class hierarchy*



Animal

Reptile

Bird

Mammal

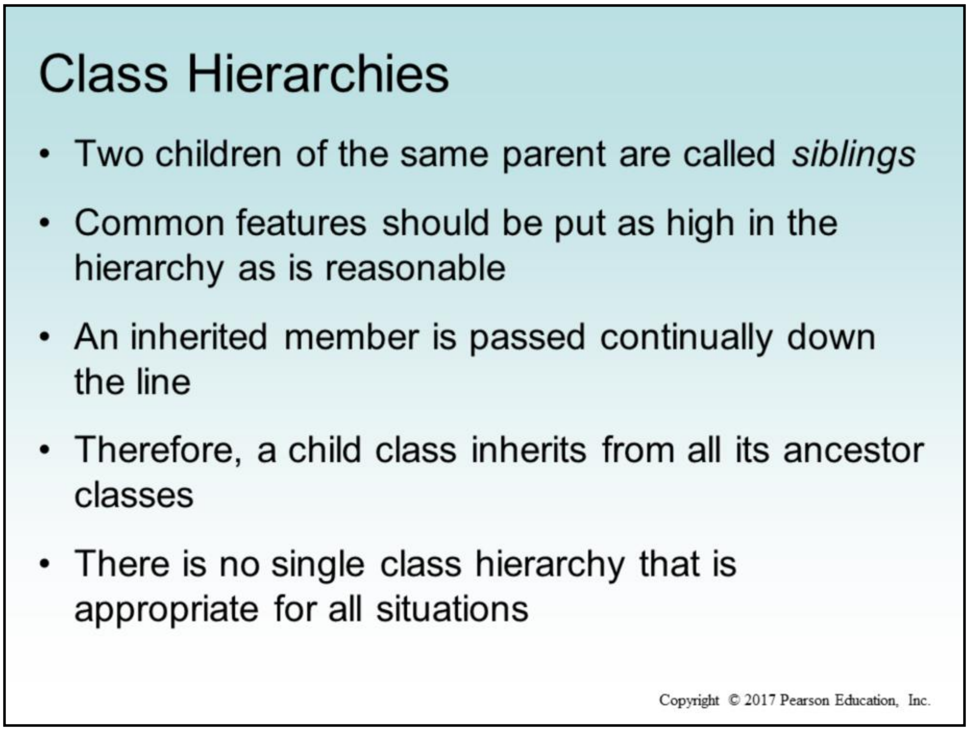
Snake

Lizard

Parrot

Horse

Bat

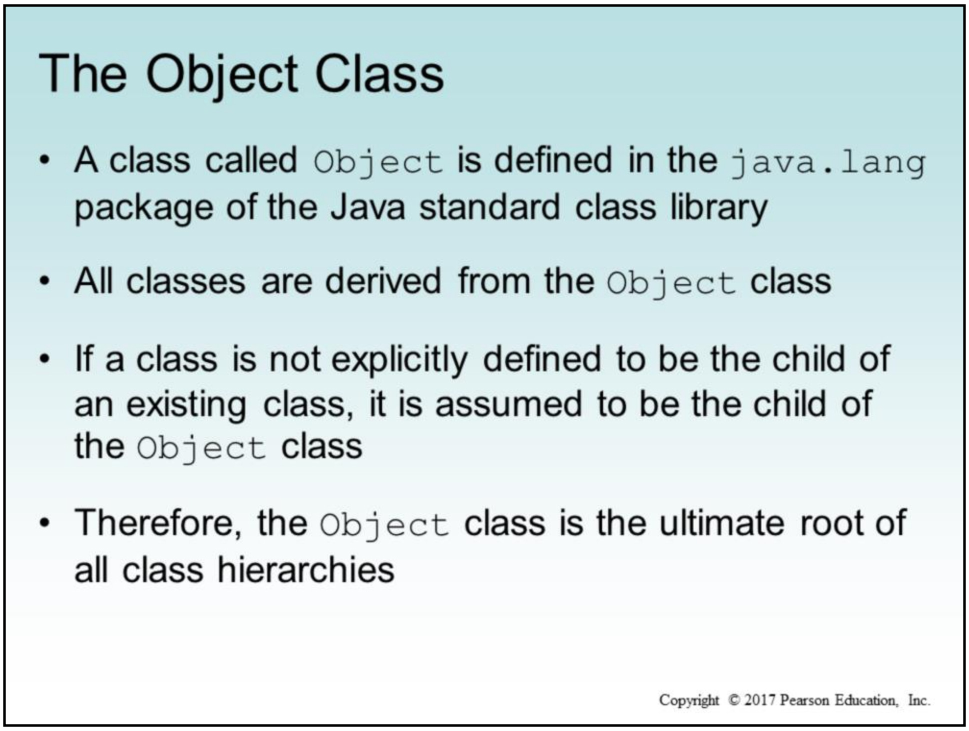


Class Hierarchies

- Two children of the same parent are called *siblings*
- Common features should be put as high in the hierarchy as is reasonable
- An inherited member is passed continually down the line
- Therefore, a child class inherits from all its ancestor classes
- There is no single class hierarchy that is appropriate for all situations

-Attributes and behaviors progress from very general at the top to very specific at the bottom of a hierarchy

-Classes farther down the hierarchy take up more memory because they inherit attributes from all ancestors!

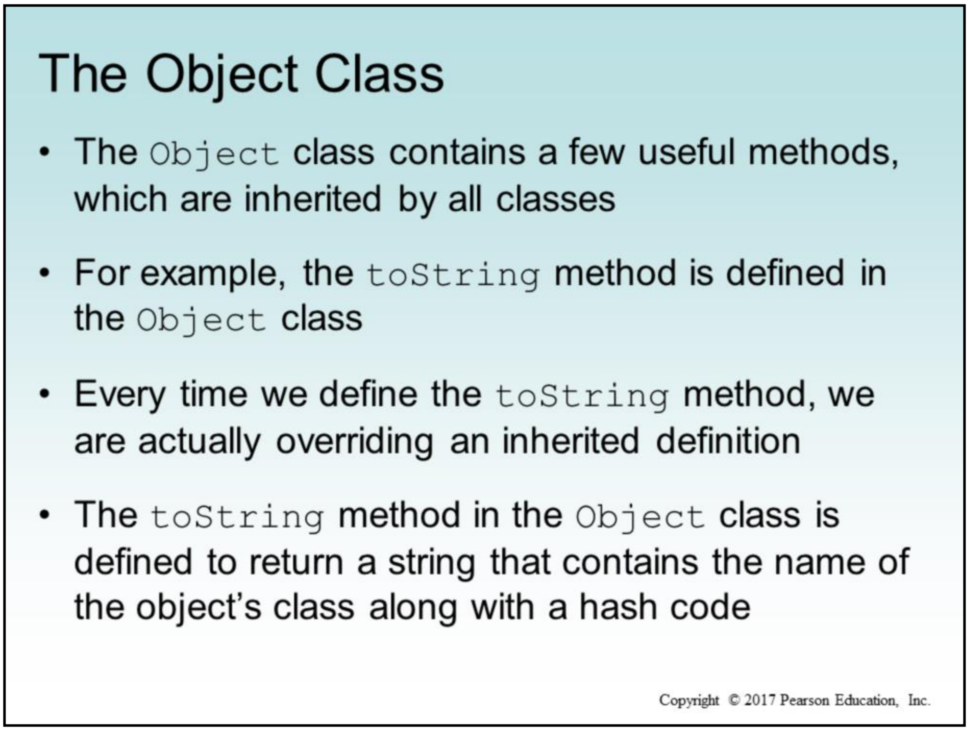


The Object Class

- A class called `Object` is defined in the `java.lang` package of the Java standard class library
- All classes are derived from the `Object` class
- If a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the `Object` class
- Therefore, the `Object` class is the ultimate root of all class hierarchies

-Whether we realized it or not, every class we have been writing is actually derived from a class called Object

-This is the top level class for all Java classes – all classes are derived from this Object class



The Object Class

- The `Object` class contains a few useful methods, which are inherited by all classes
- For example, the `toString` method is defined in the `Object` class
- Every time we define the `toString` method, we are actually overriding an inherited definition
- The `toString` method in the `Object` class is defined to return a string that contains the name of the object's class along with a hash code

-Because of inheritance, we inherit some very general and useful methods from the Object class

-These methods are part of the Object class because they are applicable to all classes!

-We've actually been using a couple in our studies; namely the toString method and the equals method

-These are general methods that we have been **overriding** to make useful for the classes we've been writing

-Recall how we've been overriding the toString class to return a String to describe our classes!

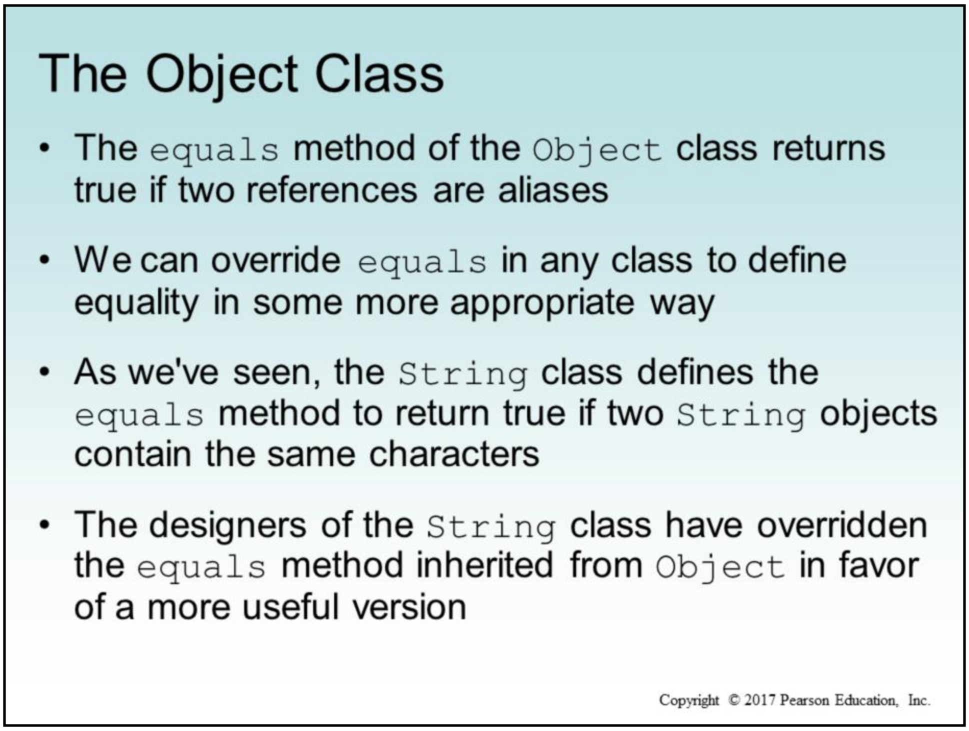
-In this way, we have been practicing methods of inheritance and object-oriented design!

-If we don't override this toString method, the Object's toString method is called

-The Object's toString method returns a string that is the name of the class followed by some unique value

-In Eclipse, we can see all the methods that we can override from the Object class (and any parent class)

-Simply, RMB, Source->Override/Implement methods….



The Object Class

- The `equals` method of the `Object` class returns true if two references are aliases
- We can override `equals` in any class to define equality in some more appropriate way
- As we've seen, the `String` class defines the `equals` method to return true if two `String` objects contain the same characters
- The designers of the `String` class have overridden the `equals` method inherited from `Object` in favor of a more useful version

-Another method we've overridden is the equals method to determine how to test two objects for equality

-Recall we can't test equality with object reference variables using == since that simply compares their addresses

-Instead when we write a class, we typically override this method to allow objects of our class to be compared

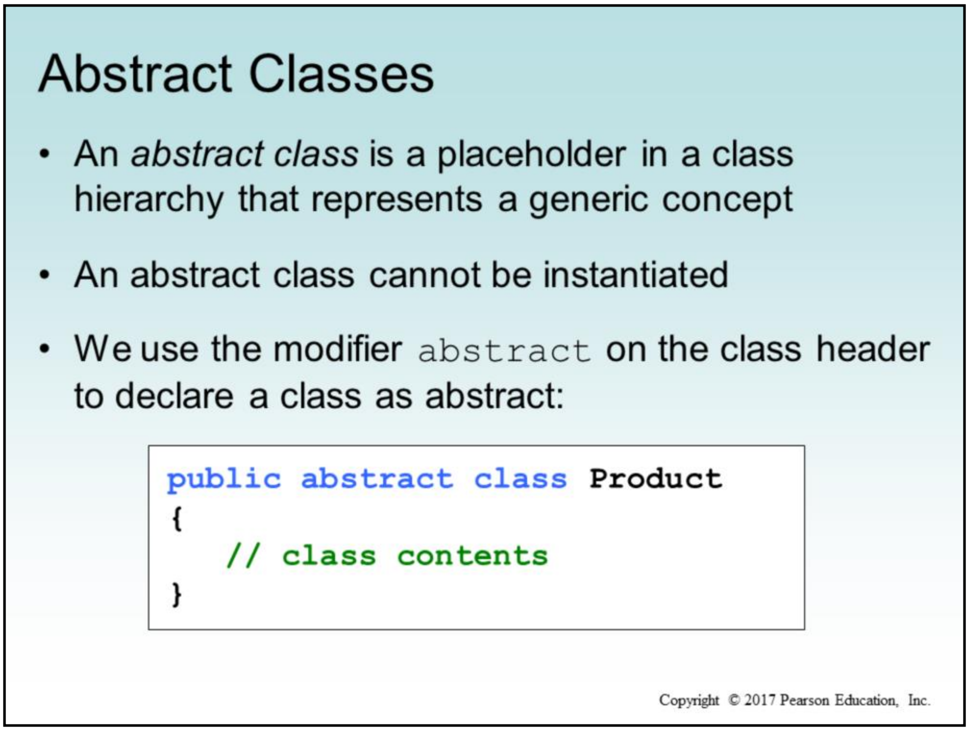
-Instead of testing for equality using ==, we use the equals method, for example:

```
MyClass c1 = new MyClass();
MyClass c2 = new MyClass();

if( c1.equals(c2) )
{
        ….
}
```

NOT

```
if( c1 == c2 )
{
        …
}
```



Abstract Classes

- An *abstract class* is a placeholder in a class hierarchy that represents a generic concept
- An abstract class cannot be instantiated
- We use the modifier `abstract` on the class header to declare a class as abstract:

```
public abstract class Product
{
   // class contents
}
```

-Think of an **abstract** class as a very **general** type of class that doesn’t “make sense” to instantiate

-For example, consider a hierarchy of different types of shapes with a top base class named Shape

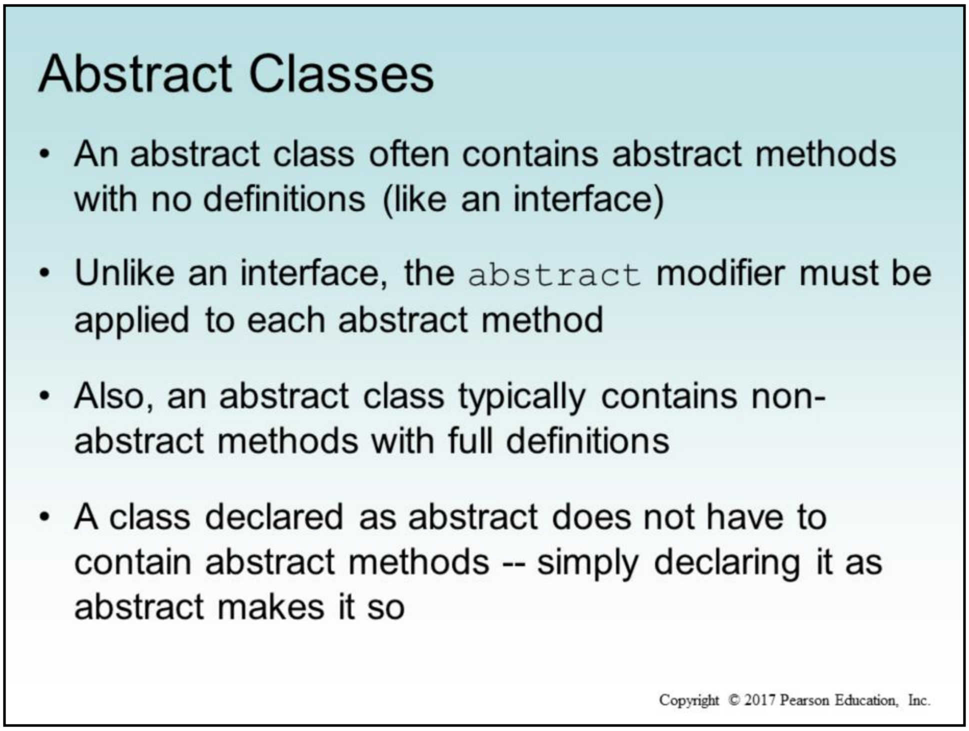
-It really doesn’t make sense to instantiate (create) an object of a Shape type because… what is a Shape??

-Instead, we create objects of specific **types of** shapes (e.g. Rectangle, Triangle, Circle)

-Even though we never actually create a Shape object, we need the class, however, in our class hierarchy

-We need it to store general attributes (**instance data**) applicable to all types of Shapes

-We also need it to define **behaviors** applicable to all types of Shapes



Abstract Classes

- An abstract class often contains abstract methods with no definitions (like an interface)
- Unlike an interface, the `abstract` modifier must be applied to each abstract method
- Also, an abstract class typically contains non-abstract methods with full definitions
- A class declared as abstract does not have to contain abstract methods -- simply declaring it as abstract makes it so

-The behaviors (methods) we specify in a base abstract class may also be abstract

-An abstract method is simply a method with a name and signature, but no method definition!

-Methods are abstract because we cannot implement such methods in the base class

-In other words, it doesn’t “make sense” to implement some methods in a base class

-Consider our Shape hierarchy again with a method to compute the area

-What does it mean to compute the area of a Shape??

-Since we don’t know the type of Shape, we don’t know how to compute the area!

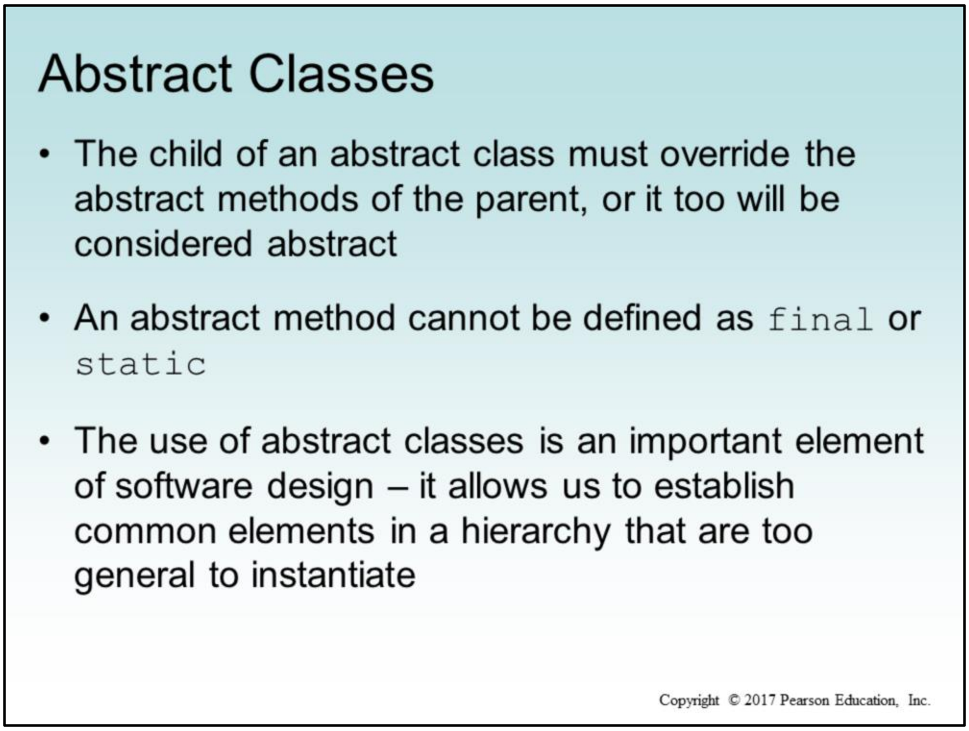
-As a result, we make such a method abstract in the base Shape class and allow the children to implement it!

-Such an abstract method in the Shape class would look like (note that there is no definition)

```
abstract public void computeArea();
```

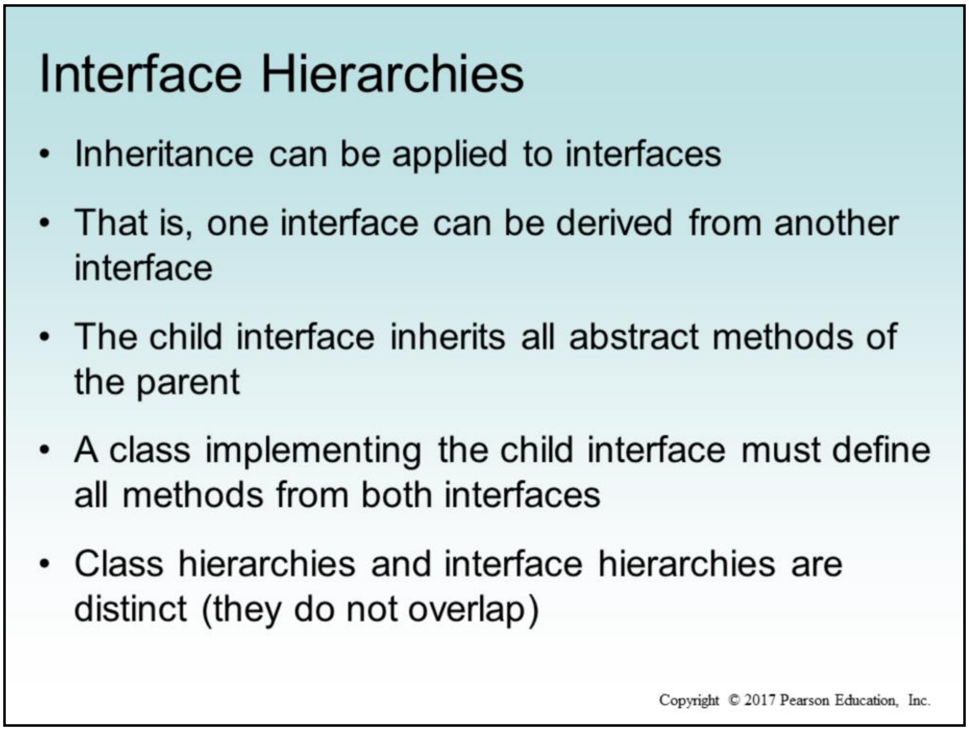
-And a derived Rectangle class, for example, would then provide the implementation for the abstract method:

```
public void computeArea()
{
        area = width * height;
}
```



Abstract Classes

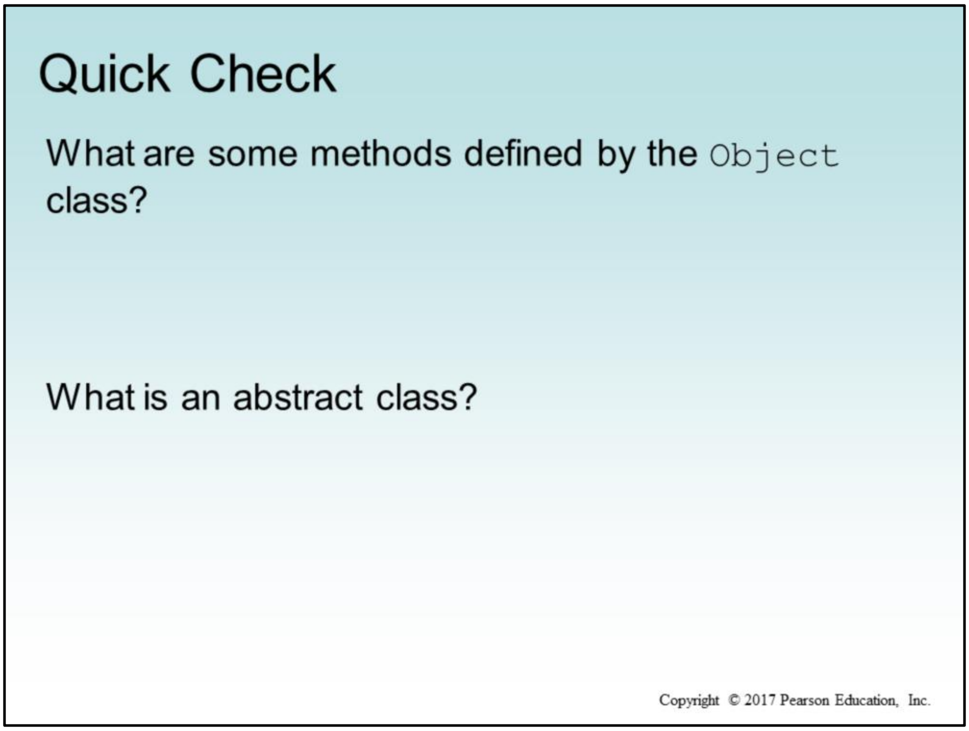
- The child of an abstract class must override the abstract methods of the parent, or it too will be considered abstract
- An abstract method cannot be defined as `final` or `static`
- The use of abstract classes is an important element of software design – it allows us to establish common elements in a hierarchy that are too general to instantiate



Interface Hierarchies

- Inheritance can be applied to interfaces
- That is, one interface can be derived from another interface
- The child interface inherits all abstract methods of the parent
- A class implementing the child interface must define all methods from both interfaces
- Class hierarchies and interface hierarchies are distinct (they do not overlap)

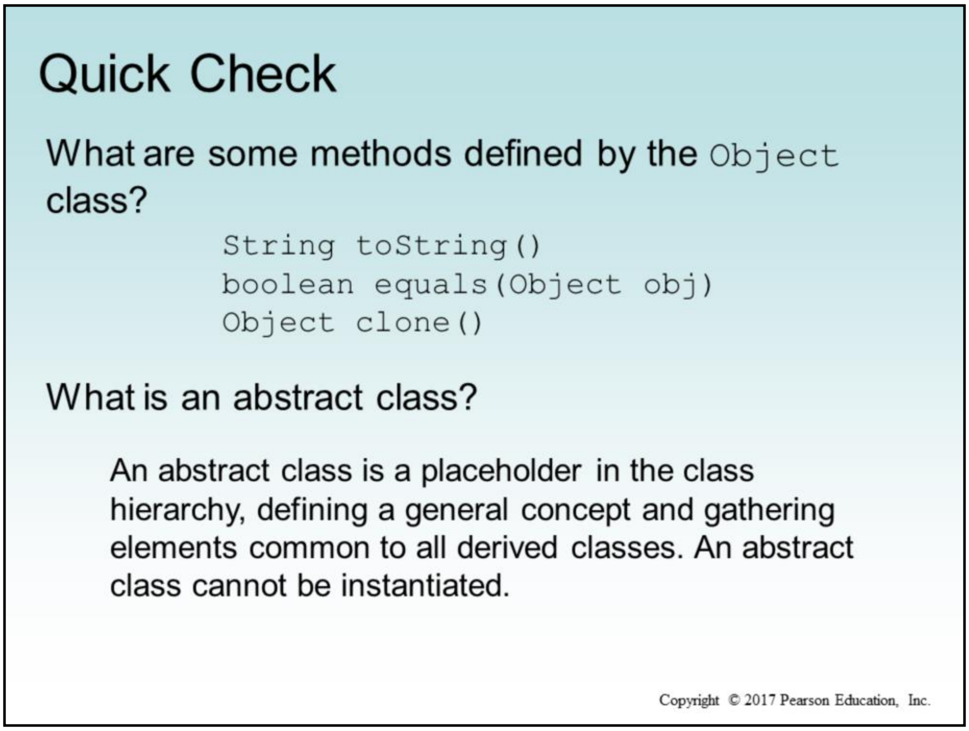
-Just as classes can derive from one another to form hierarchies, so can interfaces



Quick Check

What are some methods defined by the `Object` class?

What is an abstract class?



Quick Check

What are some methods defined by the `Object` class?

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

What is an abstract class?

An abstract class is a placeholder in the class hierarchy, defining a general concept and gathering elements common to all derived classes. An abstract class cannot be instantiated.