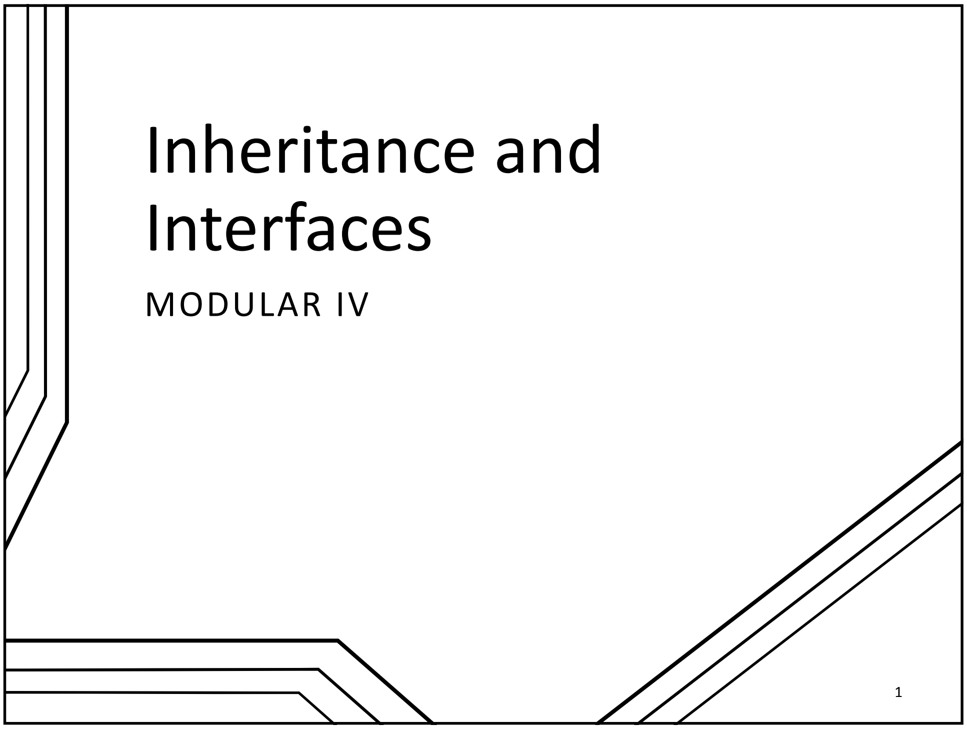
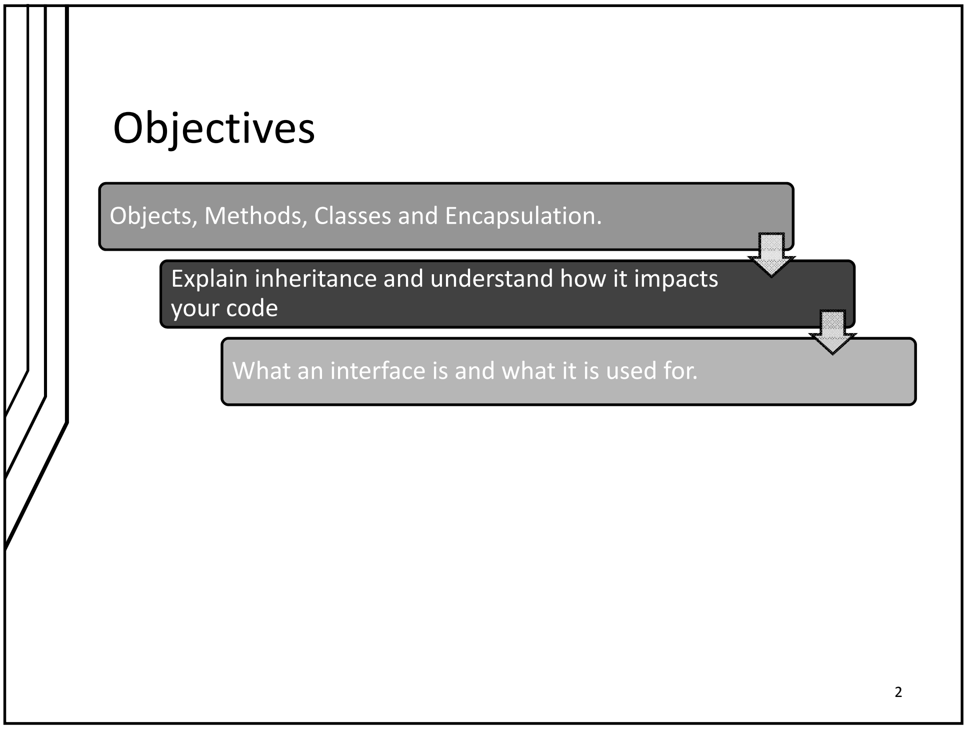


Inheritance and Interfaces

MODULAR IV

1



Objectives

Objects, Methods, Classes and Encapsulation.

Explain inheritance and understand how it impacts your code

What an interface is and what it is used for.

2

Object Oriented Programming

- Programs are constructed from modules called *objects*.
- An *object* is grouping of some data and instructions that act on data.
- Data and actions are strongly related so the object is a meaningful thing. OOP models real-world problems. A model is built from objects that interact with each other.
- E.g . school - objects: student, class, teacher. Actions: enroll student, assign grade, display info . . .

Objects and Methods

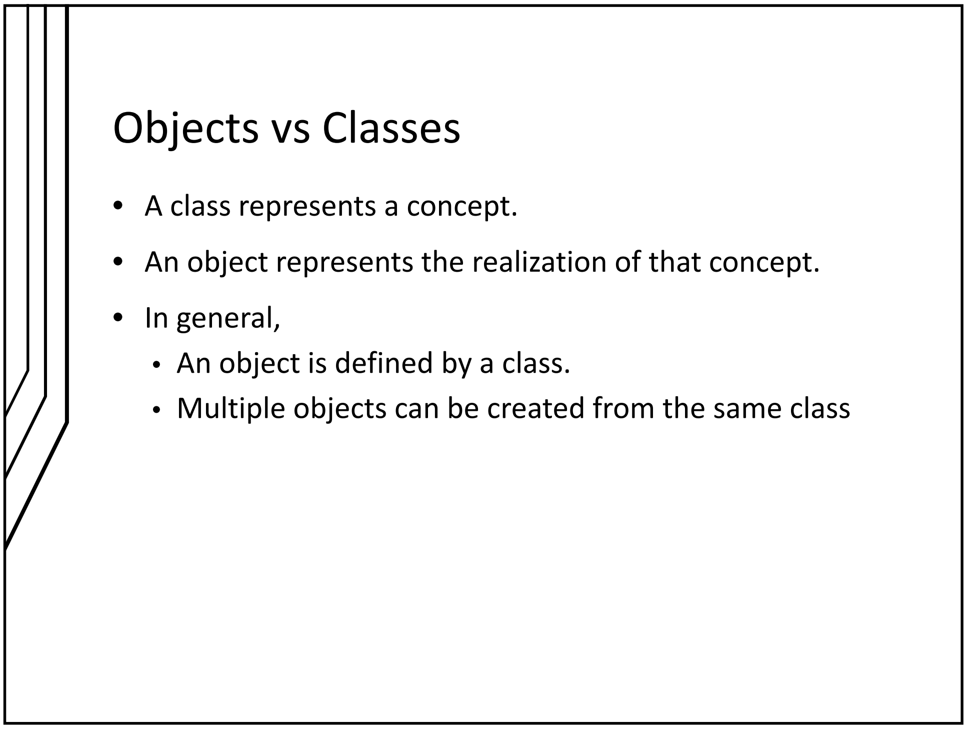
- An object is a collection of data and operations (variables and methods)
- A variable of a given (primitive) type has:
  - storage for a single value
  - a predefined set of operators
  - eg. + - * / % operations for int variable.
- An object of a given class has:
  - storage for several values
  - defines its own set of methods to operate on instance variables

Objects

- An object has:
  - state - descriptive characteristics
  - behaviors - what it can do (or be done to it)
- Example, consider a coin that can be flipped so that it's face shows either "heads" or "tails"
  - The <u>state</u> of the coin is its current face (heads or tails)
  - The <u>behavior</u> of the coin is that it can be flipped
- Note: that the behavior of the coin might change its state.

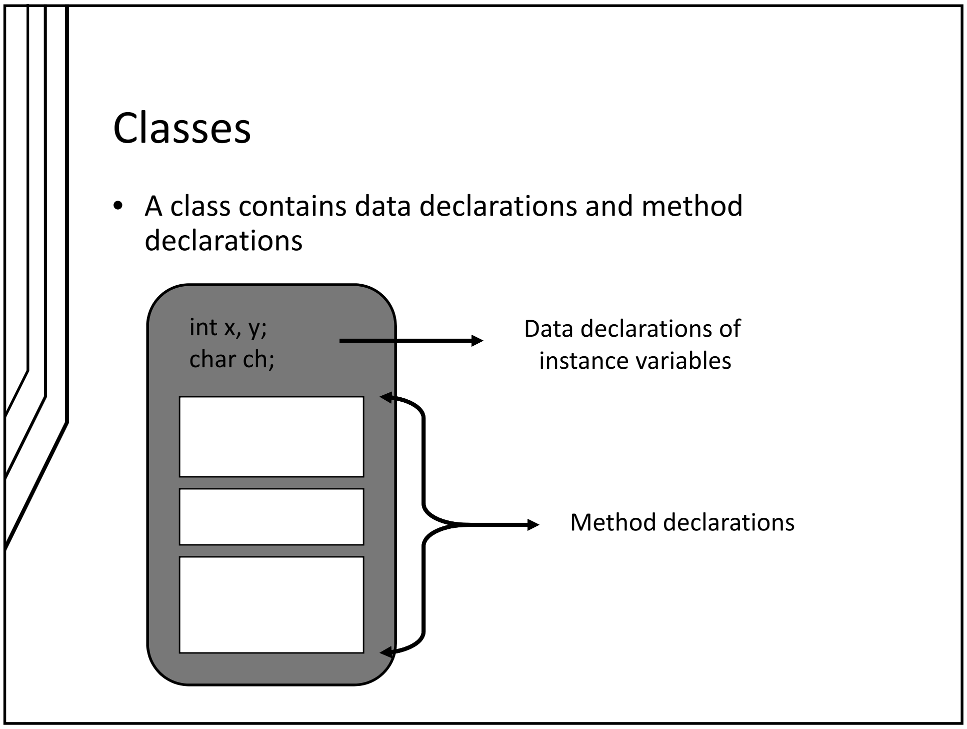
Classes

- *class* is a reserved word used to define a blueprint for creating objects.
- A class is a model that defines the variables and methods an object will contain when *instantiated*.
- Example, the String class is used to define String objects.
  - Each String object contains specific characters (<u>state</u>).
  - Each String object can perform services (<u>behaviors</u>) such as toUpperCase.



Objects vs Classes

- A class represents a concept.
- An object represents the realization of that concept.
- In general,
  - An object is defined by a class.
  - Multiple objects can be created from the same class



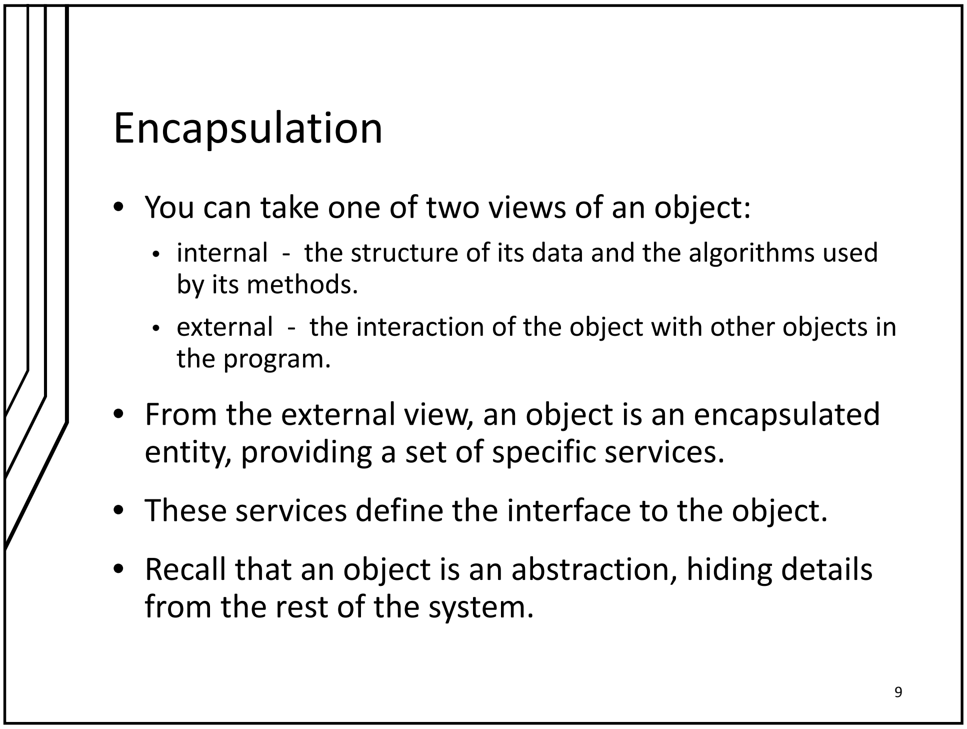
Classes

- A class contains data declarations and method declarations

```
int x, y;
char ch;
```

Data declarations of instance variables

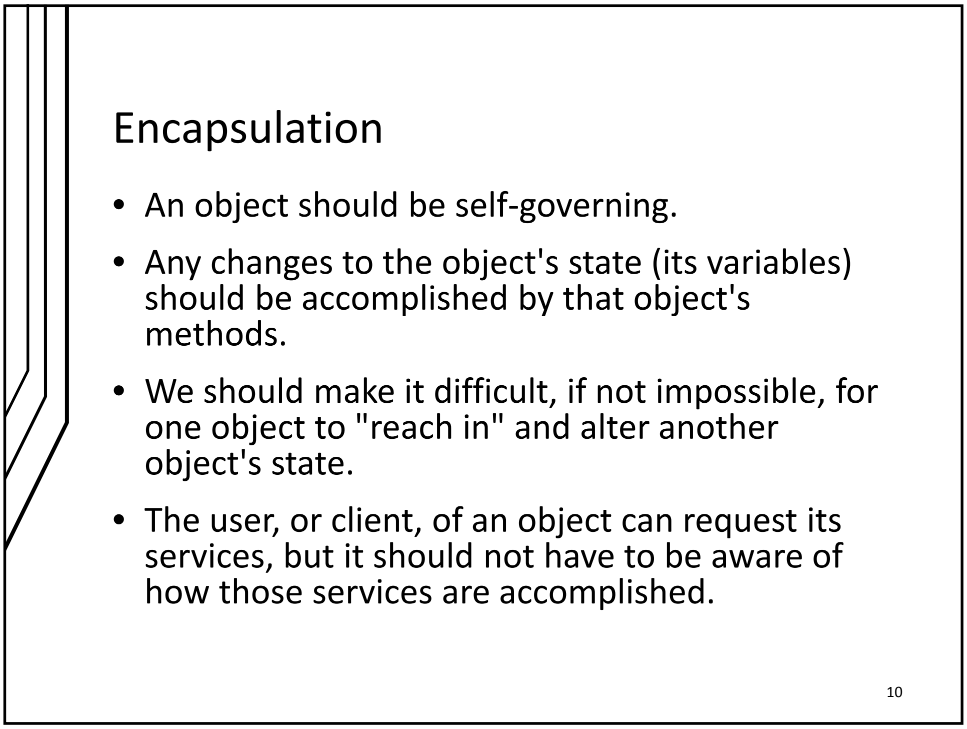
Method declarations



Encapsulation

- You can take one of two views of an object:
  - internal - the structure of its data and the algorithms used by its methods.
  - external - the interaction of the object with other objects in the program.
- From the external view, an object is an encapsulated entity, providing a set of specific services.
- These services define the interface to the object.
- Recall that an object is an abstraction, hiding details from the rest of the system.

9



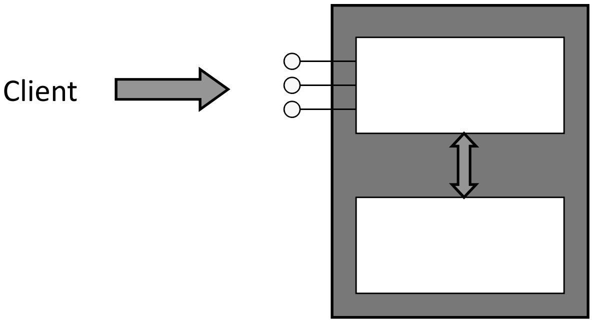
Encapsulation

- An object should be self-governing.
- Any changes to the object's state (its variables) should be accomplished by that object's methods.
- We should make it difficult, if not impossible, for one object to "reach in" and alter another object's state.
- The user, or client, of an object can request its services, but it should not have to be aware of how those services are accomplished.

10

Encapsulation

- An encapsulated object can be thought of as a black box.
- Its inner workings are hidden to the client, which only invokes the interface methods.

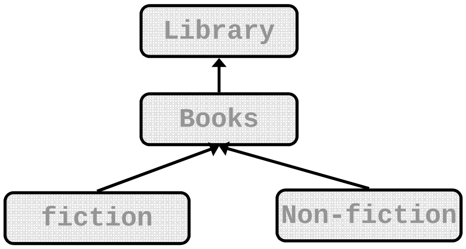


Inheritance

- Inheritance allows a software developer to derive a new class from an existing one
- The existing class is called the parent class, or superclass, or base class
- The derived class is called the child class or subclass.
- As the name implies, the child inherits characteristics of the parent
- That is, the child class inherits the methods and data defined for the parent class

Inheritance

- Inheritance relationships are often shown graphically in a class diagram, with the arrow pointing to the parent class



Library

Books

fiction

Non-fiction

- Inheritance should create an is-a relationship, meaning the child is a more specific version of the parent

Deriving Subclasses

- In Java, we use the reserved word extends to establish an inheritance relationship

```
 class Book extends Library { ... }
```

or

```
   class Fiction extends Books { ... }
```

- Book is a derived class of Library
- Fiction is a derived class of Books

Controlling Inheritance

- Visibility modifiers determine which class members get inherited and which do not.
- Variables and methods declared with *public* visibility are inherited, and those with *private* visibility are not.
- But *public* variables violate our goal of encapsulation.
- There is a third visibility modifier that helps in inheritance situations: *protected*

The *protected* Modifier

- The *protected* visibility modifier allows a member of a base class to be inherited into the child.
- But *protected* visibility provides more encapsulation than public does.
- However, *protected* visibility is not as tightly encapsulated as *private* visibility.

The super Reference

- Constructors are not inherited, even though they have public visibility.
- However, we often want to use the parent's constructor to set up the "parent's part" of the object.
- The *super* reference can be used to refer to the parent class and is used to invoke the parent's constructor.
- Ex:
  - From the Book Class, the constructor looks like:
    - public Book (String author, String title, int pages) {…}
  - We would make the call from Library’s constructor as
    - super(author, title, pages);
  - where author and title are strings and pages is an integer

Overriding Methods

- A child class can override the definition of an inherited method in favor of its own.
  - That is, a child can redefine a method that it inherits from its parent.
- The new method must have the same signature as the parent's method, but can have different code in the body.
- The type of the object executing the method determines which version of the method is invoked.
- Note: that a parent method can be explicitly invoked using the super reference.
- If a method is declared with the *final* modifier, it cannot be overridden.
- The concept of overriding can be applied to data (called shadowing variables), there is generally no need for it.

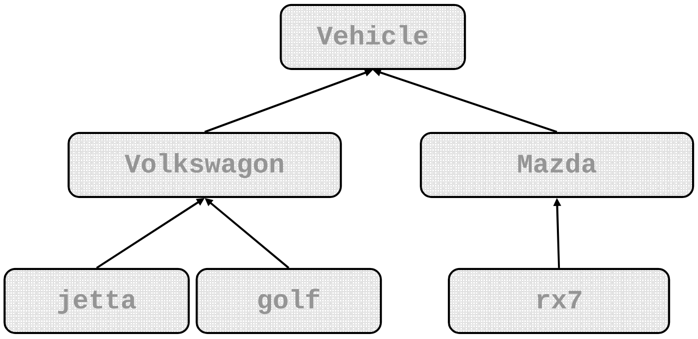
Overloading vs. Overriding

- Don't confuse the concepts of *overloading* and *overriding.*
- *Overloading* deals with multiple methods in the same class with the same name but 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 data.
- *Overriding* lets you define a similar operation in different ways for different object types.

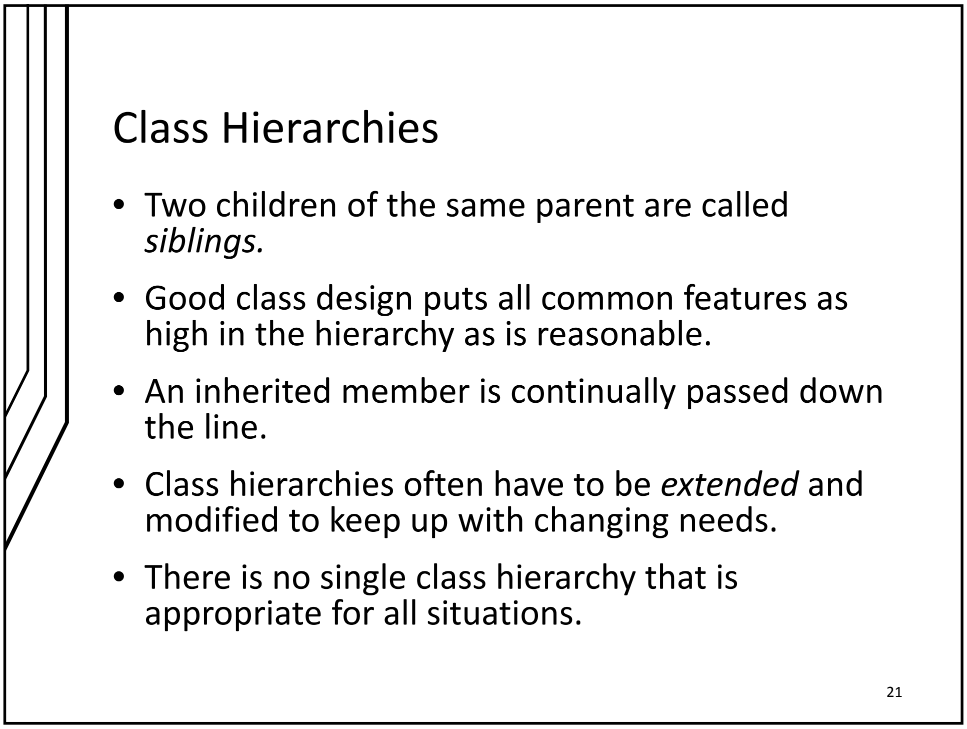
19

Class Hierarchies

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

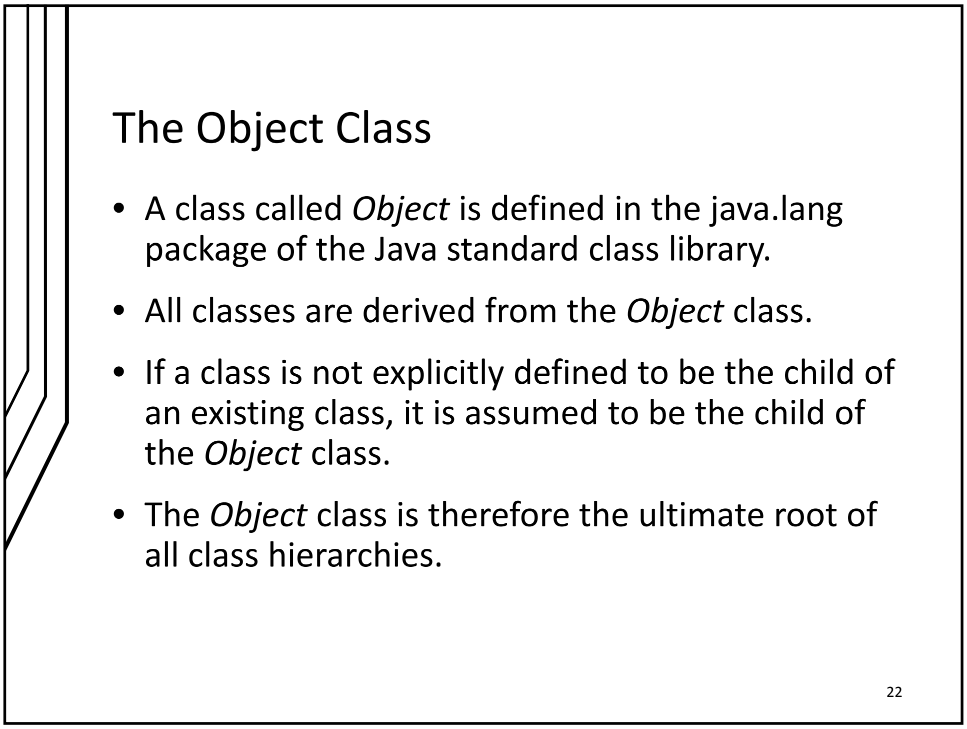


20



Class Hierarchies

- Two children of the same parent are called *siblings.*
- Good class design puts all common features as high in the hierarchy as is reasonable.
- An inherited member is continually passed down the line.
- Class hierarchies often have to be *extended* and modified to keep up with changing needs.
- There is no single class hierarchy that is appropriate for all situations.



The Object Class

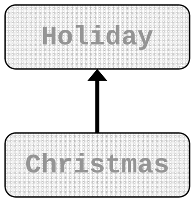
- A class called *Object* is defined in the java.lang package of the Java standard class library.
- All classes are derived from the *Object* class.
- If a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the *Object* class.
- The *Object* class is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

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 have defined *toString*, we have actually been overriding it.
- The *toString* method in the *Object* class is defined to return a string that contains the name of the object's class and a hash value (memory location).

References and Inheritance

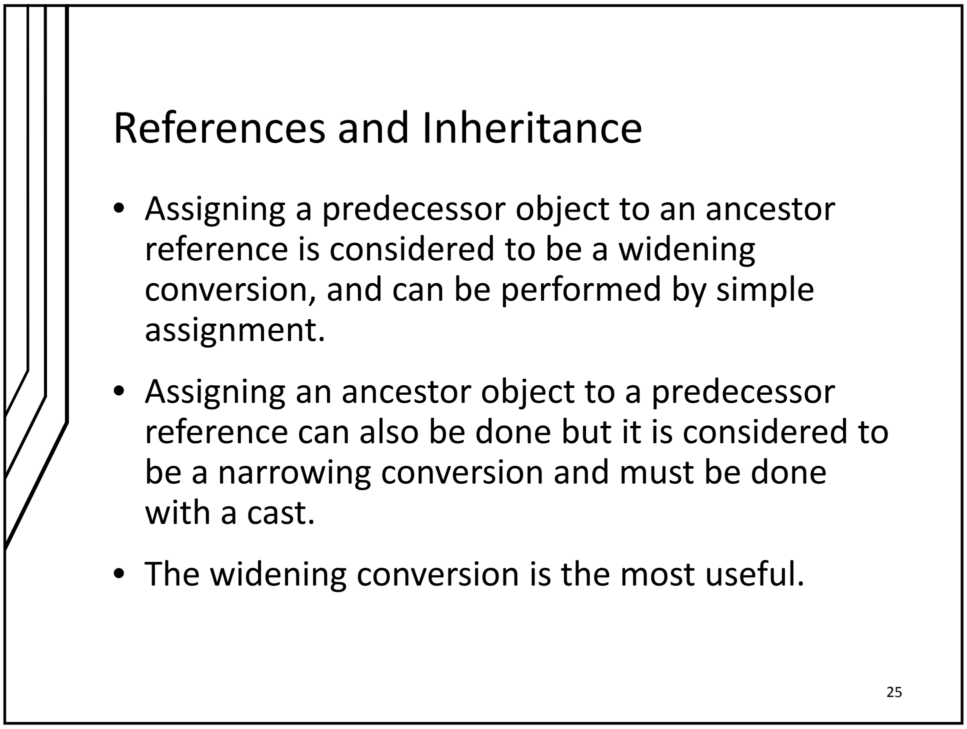
- An object reference can refer to an object of its class or to an object of any class related to it by inheritance.
- For example, if the Holiday class is used to derive a child class called Christmas, then a Holiday reference could actually be used to point to a Christmas object.



Holiday

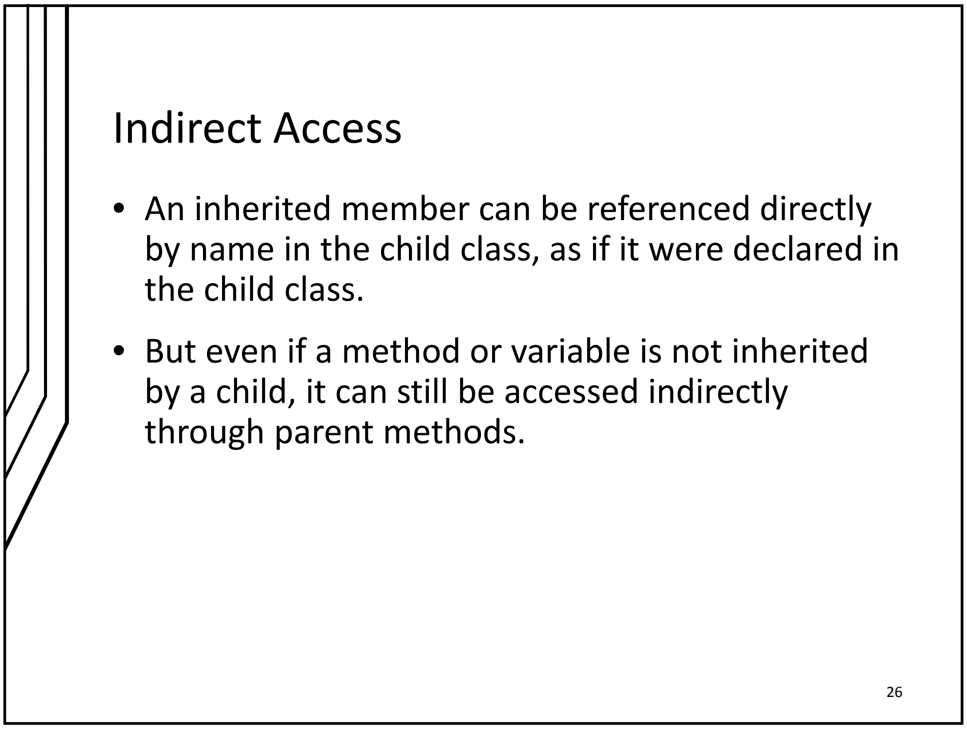
Christmas

```
Holiday day;
day = new Christmas();
```



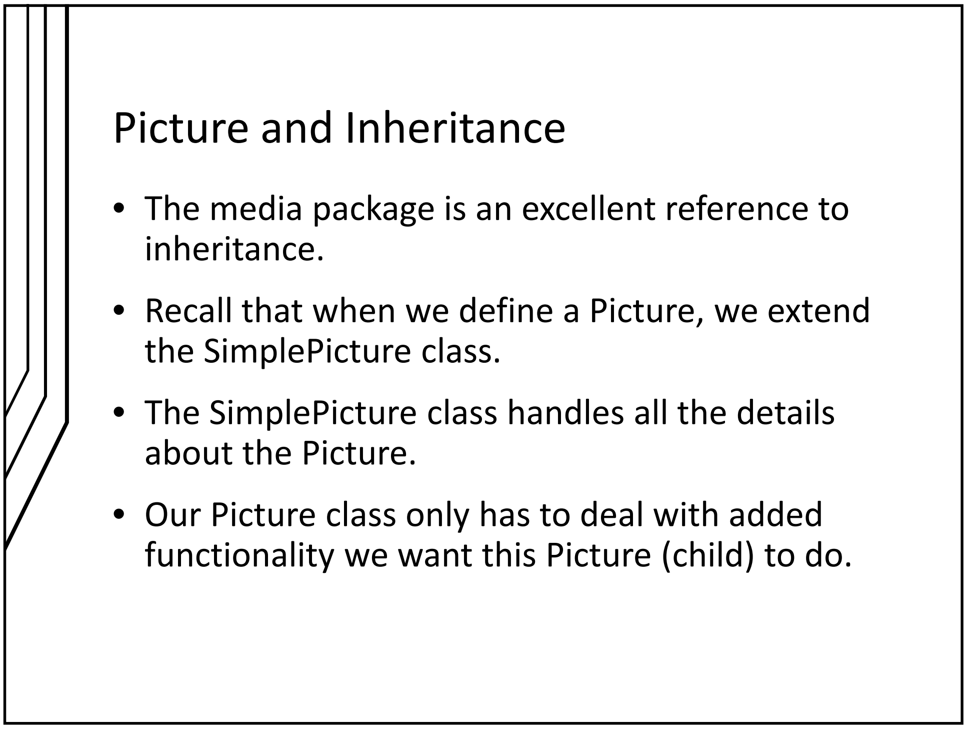
References and Inheritance

- Assigning a predecessor object to an ancestor reference is considered to be a widening conversion, and can be performed by simple assignment.
- Assigning an ancestor object to a predecessor reference can also be done but it is considered to be a narrowing conversion and must be done with a cast.
- The widening conversion is the most useful.



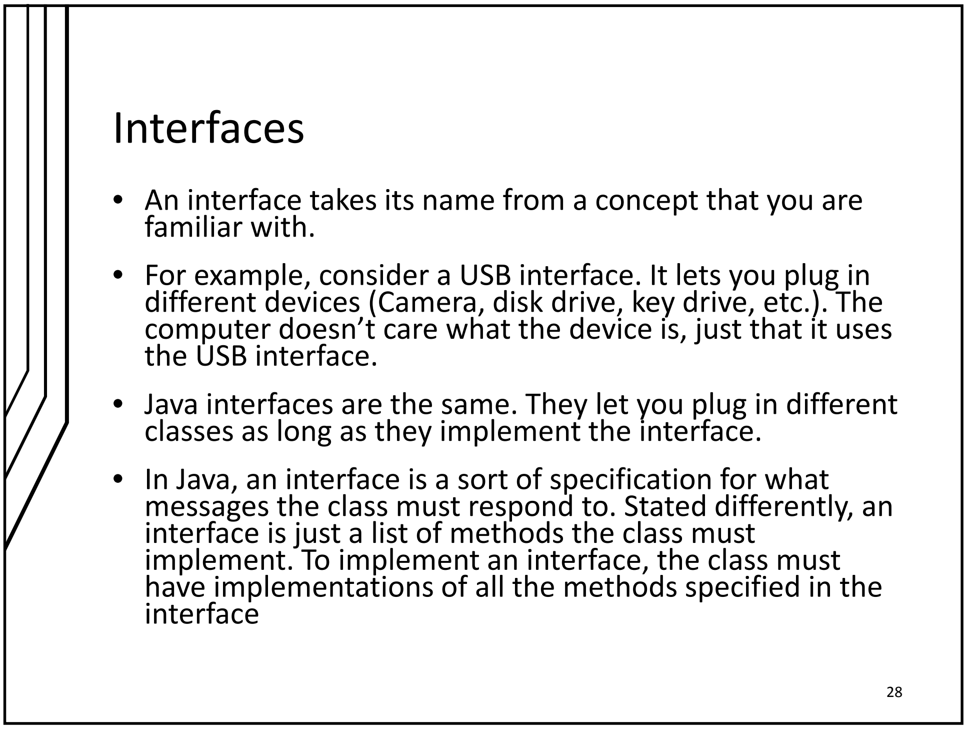
Indirect Access

- An inherited member can be referenced directly by name in the child class, as if it were declared in the child class.
- But even if a method or variable is not inherited by a child, it can still be accessed indirectly through parent methods.



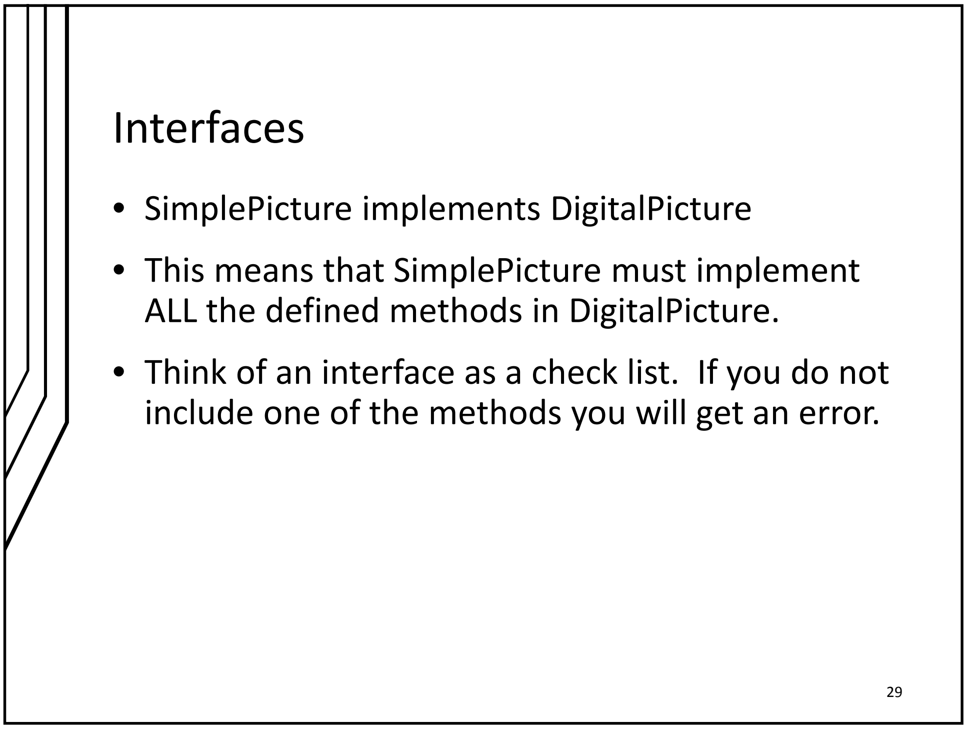
Picture and Inheritance

- The media package is an excellent reference to inheritance.
- Recall that when we define a Picture, we extend the SimplePicture class.
- The SimplePicture class handles all the details about the Picture.
- Our Picture class only has to deal with added functionality we want this Picture (child) to do.



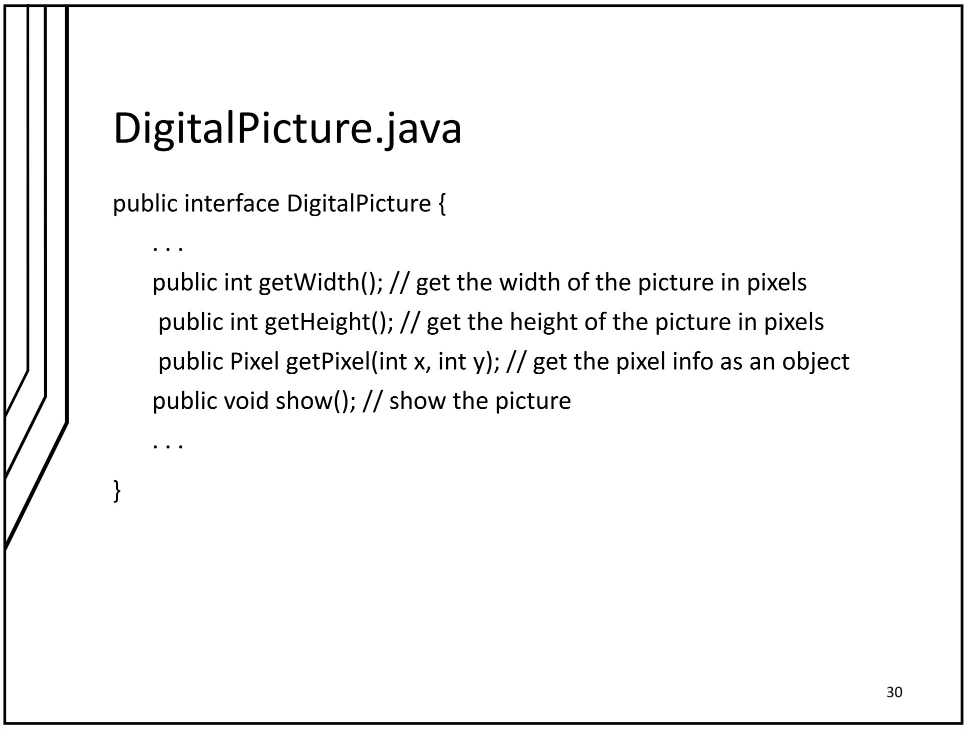
Interfaces

- An interface takes its name from a concept that you are familiar with.
- For example, consider a USB interface. It lets you plug in different devices (Camera, disk drive, key drive, etc.). The computer doesn’t care what the device is, just that it uses the USB interface.
- Java interfaces are the same. They let you plug in different classes as long as they implement the interface.
- In Java, an interface is a sort of specification for what messages the class must respond to. Stated differently, an interface is just a list of methods the class must implement. To implement an interface, the class must have implementations of all the methods specified in the interface



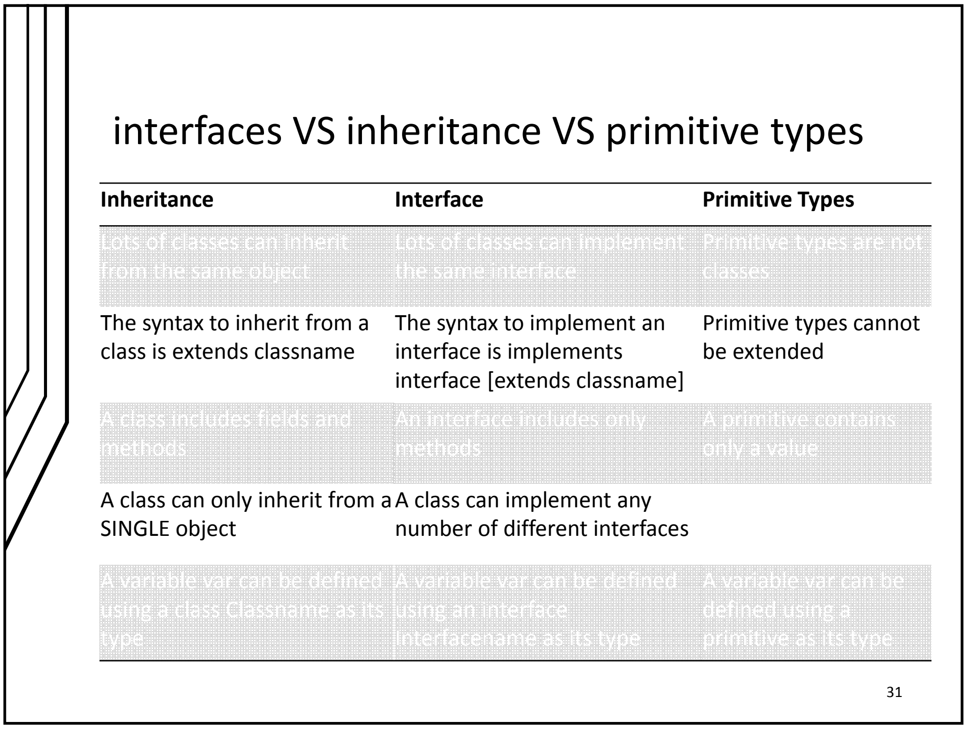
Interfaces

- SimplePicture implements DigitalPicture
- This means that SimplePicture must implement ALL the defined methods in DigitalPicture.
- Think of an interface as a check list. If you do not include one of the methods you will get an error.



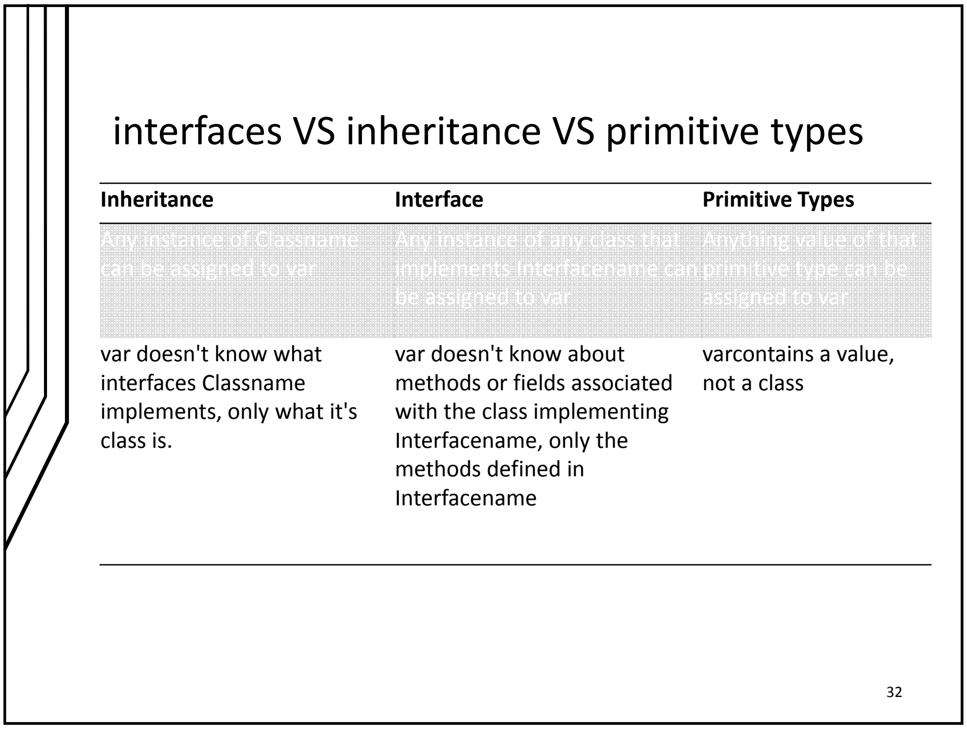
DigitalPicture.java

```
public interface DigitalPicture {
      . . .
      public int getWidth(); // get the width of the picture in pixels
       public int getHeight(); // get the height of the picture in pixels
       public Pixel getPixel(int x, int y); // get the pixel info as an object
      public void show(); // show the picture
      . . .
}
```



interfaces VS inheritance VS primitive types

Inheritance	Interface	Primitive Types
Lots of classes can inherit from the same object	Lots of classes can implement the same interface	Primitive types are not classes
The syntax to inherit from a class is extends classname	The syntax to implement an interface is implements interface [extends classname]	Primitive types cannot be extended
A class includes fields and methods	An interface includes only methods	A primitive contains only a value
A class can only inherit from a SINGLE object	A class can implement any number of different interfaces	
A variable var can be defined using a class Classname as its type	A variable var can be defined using an interface Interfacename as its type	A variable var can be defined using a primitive as its type



interfaces VS inheritance VS primitive types

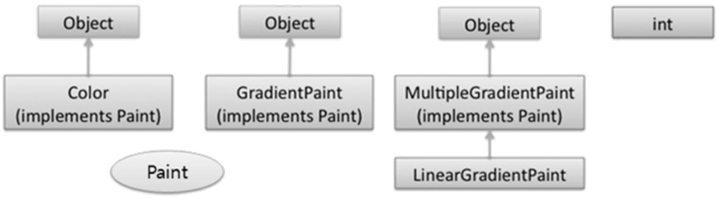
Inheritance	Interface	Primitive Types
Any instance of Classname can be assigned to var	Any instance of any class that implements Interfacename can be assigned to var	Anything value of that primitive type can be assigned to var
var doesn't know what interfaces Classname implements, only what it's class is.	var doesn't know about methods or fields associated with the class implementing Interfacename, only the methods defined in Interfacename	varcontains a value, not a class

interfaces VS inheritance VS primitive types

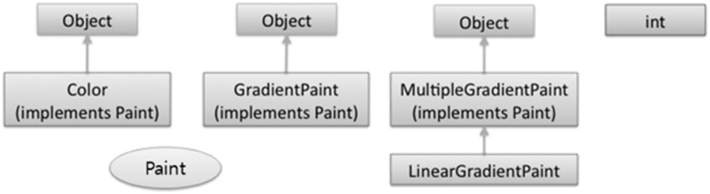
Inheritance	Interface	Primitive Types
If var's type (Classname) implements an interface Interfacename, and var2 is of type Interfacename, then var2 = var is a valid statement. This is because Java knows that Classname implements Interfacename	If var's type is Interfacename; var contains an instance of Classname; and var2 is of type Classname, then var2 = var is not a valid statement. Instead, you must use var2 = (Classname) var. This is because Java has no way of knowing which of the classes that implement Interfacename are contained in var	If var and var2 are of the same primitive type, then var2 = var is a valid statement. Otherwise you must use casting to assign var to var2
Inheritance is single, not multiple	Interfaces support "multiple inheritance"	primitives do not support inheritance

Summary

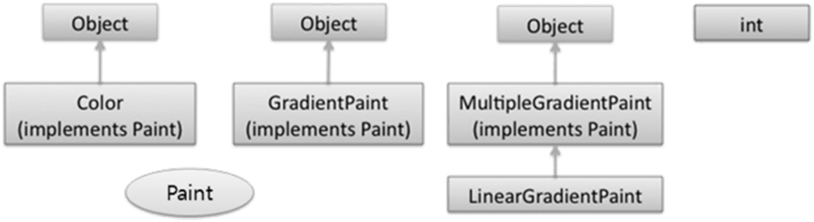
- To summarize, here are all the different sorts parameters we can use in a method.
- This picture shows several things that could be used as method parameters.


Object
Object
Object
int
Color (implements Paint)
GradientPaint (implements Paint)
MultipleGradientPaint (implements Paint)
Paint
LinearGradientPaint

- A primitive:
  - A parameter can be a primitive type such as **int** (red box)


Object
Object
Object
int
Color (implements Paint)
GradientPaint (implements Paint)
MultipleGradientPaint (implements Paint)
Paint
LinearGradientPaint

- An object whose type is a Class:
  - A parameter can have the type of a class.
  - For example, a method could be defined that takes type Object as a parameter (public void myMethod(Object myObj). In that case, any of the classes shown in the image (blue and green rectangles) would be valid parameters.
  - This is because they all inherit from Object or from something that inherits from Object.
  - Similarly, a method that takes an object of type MultipleGradientPaint could also be passed an object of type LinearGradientPaint because LinearGradientPaint inherits from MultipleGradientPaint.


Object
Object
Object
int
Color (implements Paint)
GradientPaint (implements Paint)
MultipleGradientPaint (implements Paint)
Paint
LinearGradientPaint

- An object whose type is an Interface:
  - All of the blue boxes are classes that implement Paint (an interface) either explicitly (Color, GradientPaint, MultipleGradientPaint), or through inheritance (LinearGradientPaint).
  - An object created from any of the four classes shown in blue could be passed into a method requiring a parameter of type Paint.