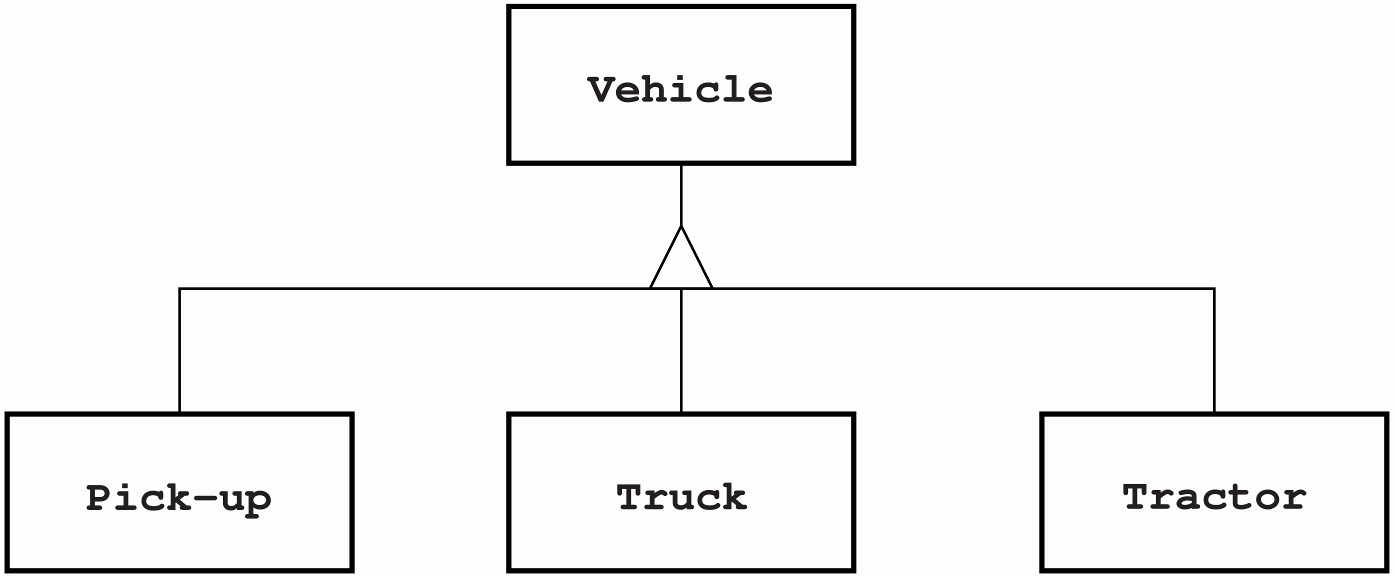
Inheritance

Object-oriented systems allow new classes to be defined in terms of a previously defined class.

All variables and methods of the previously defined class, called **superclass**, are **inherited** by **subclasses**. Subclasses can add some new variables and methods.

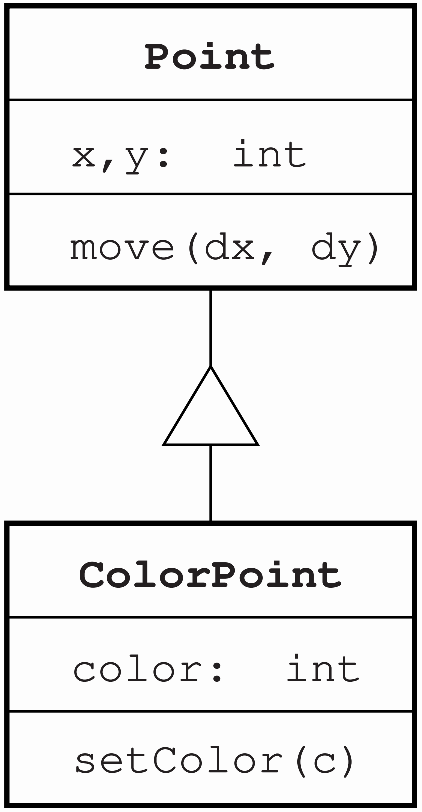
There is a hierarchical relationship between a superclass and its subclasses.



Inheritance (cont.)

Class Point represents a point in a plane.

Subclass ColorPoint adds information about color. It inherits attributes x and y and the method move() from its superclass Point.



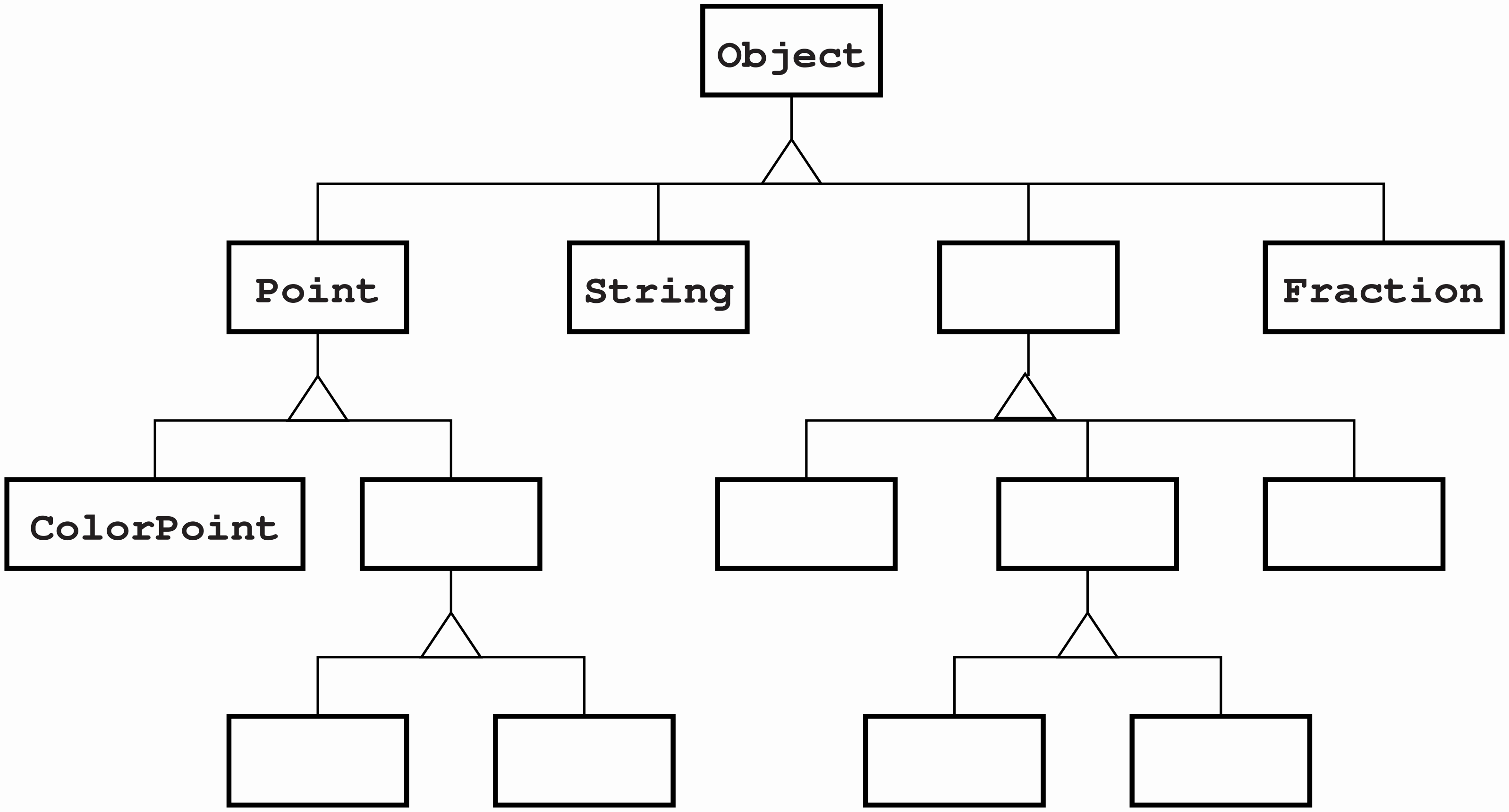
Inheritance (cont.)

Java supports inheritance through the **extends** keyword. Only single inheritance is supported, i.e. a subclass can be inherited from exactly one superclass.

```
class Point {
    private int x, y;
      ...

    public void move(int dx, int dy) {
        x += dx; y += dy;
    }
}

class ColorPoint extends Point {
    private int color;

    public void setColor(int c) {
        color = c;
    }
}
```

Hierarchy of Classes

Inheritance gives rise to a whole hierarchy of classes, because other subclasses can be inherited from subclasses of a class. Every class is a subclass of the special class Object.



Use of Subclasses

Subclasses can be used as any other classes. Attributes and methods can be accessed as usual:

```
ColorPoint p = new ColorPoint();
p.setColor(3);
p.x = 45;
p.move(10, 20);
```

Reference to an instance of a class can also point to an instance of its subclass. For example a reference to the class Point can point to an instance of its subclass ColorPoint:

```
ColorPoint p = new ColorPoint();
Point q = p;
q.move(60, -40);
Point r = new ColorPoint();
```

Use of Subclasses (cont.)

However only attributes and methods declared in the class can be accessed using reference of the given type. Attributes and methods declared in its subclasses can not be accessed using this reference.

```
Point r = new ColorPoint();
r.setColor(10);  // Compile error! Method setColor()
                 // is not defined in the class Point.
```

An instance of a subclass of a class can be assigned to a reference to the given class. On the other hand, it is a compile-time error to assign to a reference to some class an expression of type reference to its superclass:

```
ColorPoint c = new ColorPoint();
Point q = c;         //  O.K.
ColorPoint t = q;    //  Compile error!
```

Note: Every instance of ColorPoint is also an instance of Point. There can be instances of Point that are not instances of ColorPoint.

Cast Operator

It is possible to use cast operator to convert a reference to some class to a reference to its subclass:

```
Point q = new ColorPoint();
ColorPoint t = (ColorPoint)q;   // Both q and t point to
                                // the same object.
t.setColor(4);    // O.K.
q.setColor(4);    // Compile error!
```

The following usage is also possible:

```
((ColorPoint)q).setColor(4);
```

When the instance is not an instance of the class used in the cast operator, a run-time error occurs (an exception is thrown).

```
Point q = new Point();
ColorPoint t = (ColorPoint)q;   // Run-time error occurs.
```

The `instanceof` Operator

The **instanceof** operator determines whether a given object is an instance of particular class or type.

The syntax is:

```
expr instanceof type
```

where expr represents an expression that evaluates to a reference and type is a name of a class.

The result of the **instanceof** operator is **true** if the value of expr is not **null** and could be cast to the type without raising an exception. Otherwise the result is **false**.

```
Point p;
  ...
if (p instanceof ColorPoint) {
    ((ColorPoint)p).setColor(5);
}
```

Note: If it is clear at compile-time that the value of the expression can not be an instance of the given class, a compile error is produced.

The `final` Classes

When a class is marked as **final** no subclasses can be inherited from this class.

In the following example, we can not declare the class ColorPoint as a subclass of the class Point, since the class Point is final.

```
final class Point
{
        ...
}

class ColorPoint extends Point    // Compile error!
{
        ...
}
```

Polymorphism

A subclass can **override** methods of its superclass, i.e., it can provide its own implementation of these methods.

```
class Point {
        ...
    public void print() {
        System.out.print("(" + x + "," + y + ")");
    }
}

class ColorPoint extends Point {
        ...
    public void print() {
        System.out.print("(" + x + "," + y + ", color=" +
                                 color + ")");

    }
}
```

Polymorphism (cont.)

If an overridden method is called, the method in the subclass is always used. The overridden method in the superclass is not accessible from other objects.

```
ColorPoint c = new ColorPoint();
Point p = c;
c.print();  // The method print() defined in the class
p.print();  // ColorPoint is called in both cases.
```

The code that calls overridden methods does not need to be aware of different implementations of the methods in different subclasses.

```
Point[] points = new Point[10];
points[0] = new ColorPoint();
points[1] = new Point();
  ...
for (int i = 0; i < points.length; i++) {
    points[i].print();
}
```

The super Keyword

The **super** keyword can be used to access members of a class inherited by the class in which it appears.

In particular it is the only way to access overridden methods.

```
class Point {
        ...
    public void print() {
        System.out.print("(" + x + "," + y + ")");
    }
}

class ColorPoint extends Point {
        ...
    public void print() {
        super.print();   // calls the method print() in
                         // the class Point
        System.out.print(", color=" + color);
    }
}
```

The `final` Methods

A method can be marked as **final**. Such methods can not be overridden in subclasses.

```
class Point {
        ...
    public final void print() {  // Marked as 'final'.
        System.out.print("(" + x + "," + y + ")");
    }
}

class ColorPoint extends Point {
        ...
    public void print() {  // Compile error! Can not override
                           // final method.
        System.out.print("(" + x + "," + y + ", color=" +
                         color + ")");

    }
}
```

Inheritance and Constructors

Constructors are not inherited. The subclass must define its own constructors.

The constructors in the subclass can call a constructor of the superclass using the keyword **super**.

```
class Point {
    public Point(int x, int y) {
        this.x = x; this.y = y;
    }
      ...
}

class ColorPoint extends Point {
    public ColorPoint(int x, int y, int c) {
        super(x, y);  // The constructor in the class
                      //  Point is called.
        color = c;
    }
      ...
}
```

Inheritance and Constructors (cont.)

When no constructor of the superclass is called explicitly in the constructors in the subclass, the constructor with no parameters is used, as if the following construction would be put at the beginning of the constructor:

```
{
    super();
      ...
}
```

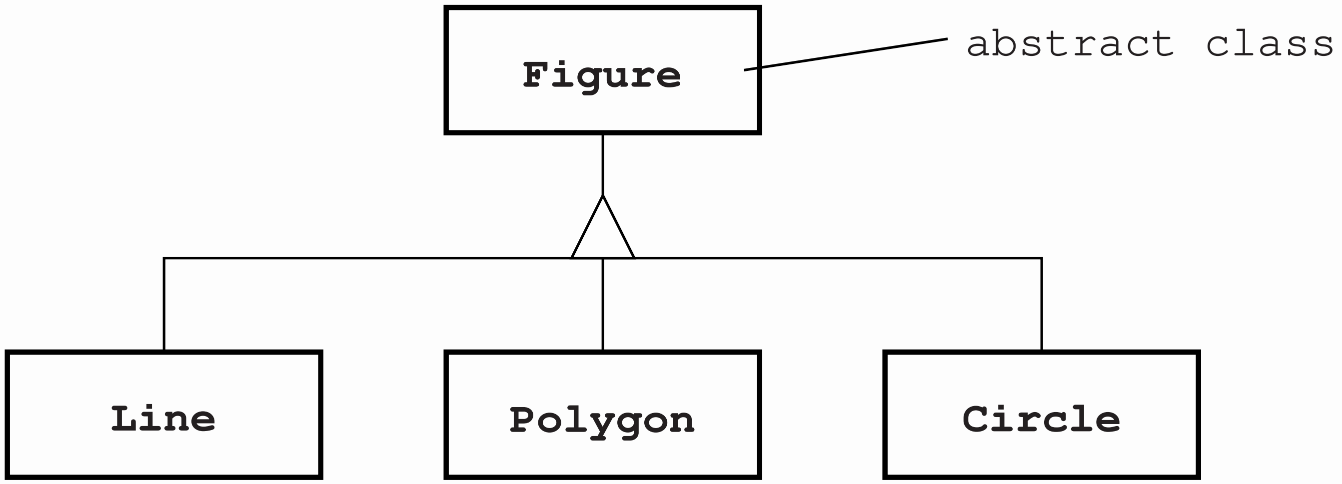
It is an compile-time error if the superclass does not define (either implicitly or explicitly) the constructor with no parameters in this case.

Abstract Classes

An **abstract** class is an incomplete description of something; a set of operations and attributes that, in themselves, do not fully describe an object.

Abstract classes are used as common superclasses of some classes and they contain common attributes and methods of these classes.

Abstract classes can not be instantiated, but their non-abstract subclasses can.



Abstract Classes (cont.)

Abstract classes are declared with the keyword **abstract**.

```
abstract class Figure {         // an abstract class
    ...
}

class Line extends Figure {   // a non-abstract class
    ...
}
```

It is possible to use references to instances of an abstract class.

```
Figure a = new Line();
a.move(10, 20);
```

It is not possible to create instances of an abstract class.

```
Figure b = new Figure();  // Compile error!
```

Abstract Methods

Abstract classes can contain abstract methods. Such methods are marked with the keyword **abstract** and have only header, their body is replaced with a semicolon (;).

```
abstract class Figure {
        ...
    abstract void draw();   // abstract method
}
```

Every abstract method must be **implemented** in non-abstract subclasses.

```
class Line extends Figure {
        ...
    void draw() {   // implementation of the abstract method
        ...   // <- draws the line
    }
}
```

Note: Every class containing a non-implemented abstract method (either directly or inherited) must be declared as an abstract class.

Abstract Methods (cont.)

Abstract methods are called as any other methods – the implementation in the corresponding subclass is called.

```
Figure[] figures = new Figure[100];
figures[0] = new Line();
figures[1] = new Circle();
figures[2] = new Polygon();
    ...
for (int i = 0; i < figures.length; i++) {
    figures[i].draw();  // The method draw() of the
                        // corresponding class is called.
}
```

Interfaces

An **interface** is a named collection of method definitions (without implementations). An interface can also declare constants.

A definition of an interface resembles a definition of a class, but the keyword **interface** is used instead of the keyword **class**.

```
interface Drawable
{
    void draw();                        // methods
    void highlight(int mode);

    int HM_DARK  = 0;                   // constants for
    int HM_LIGHT = 1;                   // highlight mode
}
```

The definitions of methods must be the same as definitions of abstract methods except that the keyword **abstract** is not used.

Interfaces (cont.)

We say a class **implements** an interface if it provides implementations of methods in the interface (in the same way as it implements abstract methods).

The used syntax is illustrated in the following example.

```
class Line implements Drawable
{
        ...
    public void draw() {
        ...              // a method that actually draws the line
    }

    public void highlight(int mode) {
        ...              // a method that actually highlights
                         // the line using the specified mode
    }
}
```

Interfaces (cont.)

A references that point to any object implementing the given interface can be used in the same way as references pointing to class instances.

```
Line l = new Line();
   ...
Drawable d = l;
d.draw();                          // O.K.
d.highlight(Drawable.HM_DARK);     // O.K.
d.move(10, 20);                    // Compile error. The method
                                   // move() is not deklared in
                                   // the interface Drawable.
```

Only methods declared in the interface can be called using a reference type corresponding to this interface.

Interfaces (cont.)

- An **interface** defines a protocol of behavior that can be implemented by any class **anywhere** in the class hierarchy.
- An interface **declares** a set of methods but **does not** implement them.
- A class that **implements** the interface agrees to implement all the methods defined in the interface, thereby agreeing to certain behavior.
- There is a hierarchy of interfaces similar to hierarchy of classes. We talk about **superinterfaces** and **subinterfaces**.

```
interface DrawableFull extends Drawable
{
    void fill(int color);
}
```

Interfaces (cont.)

- A class can implement more than one interface. Names of multiple interfaces are separated by comma (,).

```
        class Polygon extends Figure implements
                                     Drawable, Rotating {
            ...
```

- Methods declared in an interface are implicitly **public** and **abstract**. It is not possible to change this.
- Attributes declared in an interface are implicitly **public**, **static** and **final**, i.e., they represent constants. It is not possible to change this.
- When a class implements an interface, it is essentially signing a contract. Either the class must implement **all** the methods declared in the interface and its superinterfaces, or the class must be declared **abstract**.

Interfaces (cont.)

The most significant differences between interfaces and abstract classes:

- An interface cannot implement any methods, whereas an abstract class can.
- An interface cannot declare any static methods, whereas an abstract class can.
- An interface cannot declare instance variables, whereas an abstract class can.
- An interface cannot declare non-final static attributes, whereas an abstract class can.
- A class can implement many interfaces but can have only one superclass.
- An interface is not part of the class hierarchy – unrelated classes can implement the same interface.

The `final` Attributes

Constant values can be declared using the keyword **`final`**.

```
final int NUMBER = 10;
```

We can assign a value to **`final`** attributes and (local) variables only in their declarations or in constructors. An attempt to assign them a value in normal methods results in a compile-time error.

```
NUMBER = 5;  //  Compile error!
```

A **`final`** attribute is usually declared as **`static`**, since it is not necessary to have a copy of the same value in all instances, and one common copy is sufficient.

```
static final int NUMBER = 10;
```

Note: Names of constant values are by convention formed from upper-case letters and underscores (_).

The `final` Attributes (cont.)

One common usage of **`final`** attributes is to use them for representation of possible values from some finite set of values – **enumeration** of these values. To each possible element of the set we assign some arbitrary integer value. In program we always use the assigned symbolic names instead of integer values.

In this case names of attributes representing values from the set share a common prefix.

For example in a chess-playing program we can represent different pieces using the following declarations.

```
// chess pieces
public static final int P_NONE   = 0,
                        P_KING   = 1,
                        P_QUEEN  = 2,
                        P_BISHOP = 3,
                        P_KNIGHT = 4,
                        P_ROOK   = 5,
                        P_PAWN   = 6;
```