Lectures 12.1 - 12.2

Inheritance

Why inheritance?

- We are asked to write a class that manages and displays an analog clock
- An analog clock has various properties and behaviors. Some of them are features that *every clock has*
- In other words:
  - An analog clock *is a clock*
  - It has all the features of a clock + some analog clock specific features

Inheritance

If we already have a `Clock` class, we can:

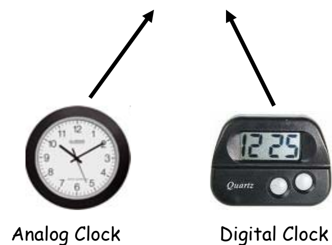
1. Define a new class, say `AnalogClock`, and make it inherit the functionality of the `Clock` class
2. Further, we can endow `AnalogClock` with additional functionality of its own

We can design other clock variants similarly

Implications: less work, less bugs, more consistency.

Clock

```
- Hours : int
- Minutes : int
- Seconds : int
...
+ setTime() : void
+ getSeconds() : int
+ setSeconds(int seconds) : int
+ secondElapsed() : void
+ secondElapsed(int n) : void
...
```



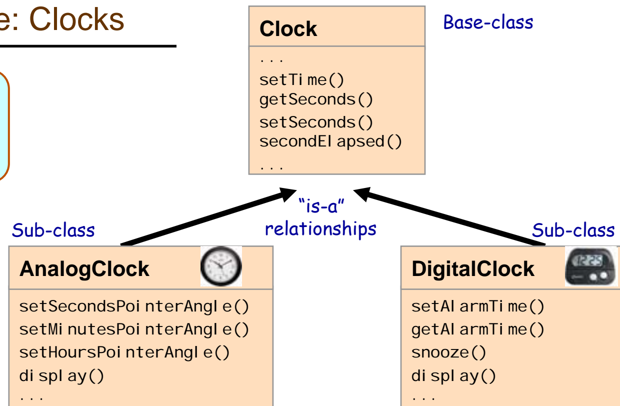
Analog Clock　　Digital Clock

Outline

- Why inheritance?
  - Motivation
  - Examples
- Inheritance anatomy
  - Constructors
  - Methods
  - Example
- The Object class
- Method overriding
- Inheritance implications
  - Narrowing / Widening
  - Run-time types
  - Virtual method calling
- Packages
- Visibility modifiers
- Design issues
  - Multiple inheritance
  - Overloading vs. overriding
  - When to sub-class?
- Benefits of inheritances

Inheritance example: Clocks

Anything that can be done with Clock objects can also be done with AnalogClock and DigitalClock objects



Inheritance = the ability to *derive* one class definition from another

- The derived sub-class *inherits* the members of the base-class
- Usually, the sub-class designer will endow it with additional, sub-class-specific members

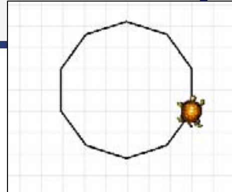
OO terminology:

- Base-class = super-class = parent class
- Sub-class = derived class = child class
- To extend = to derive = to sub-class

Inheritance example: Turtles

```
// A turtle that draws geometric figures like polygons.
public class GeoTurtle extends Turtle {
  // Draws a perfect polygon with n edges of size edgeSize
  public void drawPolygon(int n, double edgeSize) {
    int theta = 360 / n;
    for (int i=0; i<n; i++) {
      moveForward(edgeSize);
      turnLeft(theta);
    }
  }
}
```

The syntax
`C1 extends C2`
implies:
`C1` is a sub-class of `C2`
`C2` is a super-class of `C1`



Turtle
- Turtle()
- moveForward()
- turnLeft()
- turnRight()
- tailUp()
- tailDown()
- ...

GeoTurtle
- drawPolygon()
- drawSquare()
- drawCircle()
- ...

```
public class GeoTurtleDemo {
  public static void main(String[] args) {
    GeoTurtle gTurtle = new GeoTurtle();
    gTurtle.tailDown();
    gTurtle.drawPolygon(10, 50);
  }
}
```

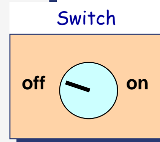
What happens when these methods are invoked?

Inheritance example: Switches

Base-class

```
// A switch that can be on/off.
public class Switch {

  // Records the state of the switch
  private boolean isOn;

  public Switch(boolean isOn) {
    this.isOn = isOn;
  }
  public boolean isOn() {
    return isOn;
  }
  public void setOn(boolean state) {
    isOn = state;
  }
}
```

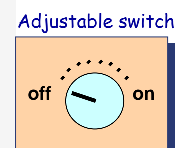
Switch



Sub-class

```
public class AdjustableSwitch extends Switch {

  private float level;

  public AdjustableSwitch(float level) {
     // later ...
  }

  public void setLevel(float level) {
    this.level = level;
    setOn(level > 0);
  }

  public float getLevel() {
    return (isOn() ? level : 0);
  }
}
```

Adjustable switch



- The sub-class is always *more specific* than the base-class.

Outline

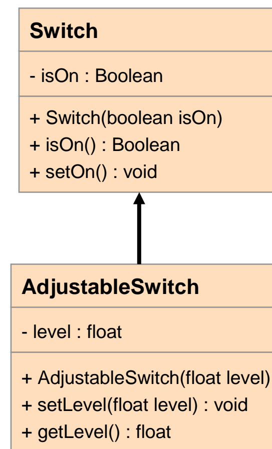
- Why inheritance?
  - Motivation
  - Examples
- ⇨ Inheritance anatomy
  - Constructors
  - Methods
  - Example
- The Object class
- Method overriding
- Inheritance implications
  - Narrowing / Widening
  - Run-time types
  - Virtual method calling
- Packages
- Visibility modifiers
- Design issues
  - Multiple inheritance
  - Overloading vs. overriding
  - When to sub-class?
- Benefits of inheritances

Inheritance anatomy

The sub-class inherits the non-private members of the base-class.

Within the sub-class code:

- Non-private fields and methods of the base-class may be used just like sub-class members
- Private fields of the base-class can be accessed via super-class public methods, as usual (example: see how `isOn` is treated in the `AdjustableSwitch` example)
- How to expose private members of the base-class to the sub-class: later.



Sub-classing constructors

Constructors are not inherited.

A sub-class constructor always has the same logic:

1. Mandatory: First, it invokes a constructor of the base-class. This is done in order to initialize the state of the sub-class object from the base-class perspective
2. Optional: Second, it may do some sub-class construction work.

Base-class

```
public class Switch {
  private boolean isOn;

  public Switch (boolean isOn) {
    this.isOn = isOn;
  }

  // Other Switch methods
```

Example:

Sub-class

```
public class AdjustableSwitch extends Switch {
   private float level;

   public AdjustableSwitch (float level) {
       super(level > 0);
       this.level = level;
   }

   // Other AdjustableSwitch methods
}
```

A sub-class constructor must begin with

Either: `super(...)`

Or: `this.anotherSubClassConstructor(...)`

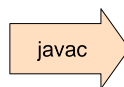
Sub-classing constructors (cont.)

If we don't declare any constructor in the sub-class, the compiler automatically:

1. Adds an empty (default) constructor to the sub-class
2. Puts in it a `super()` call to the constructor of the base-class

A sub-class without any constructor:

```
public class C1 extends C2 {
  // C1 fields
  // C1 methods (no constructor)
}
```



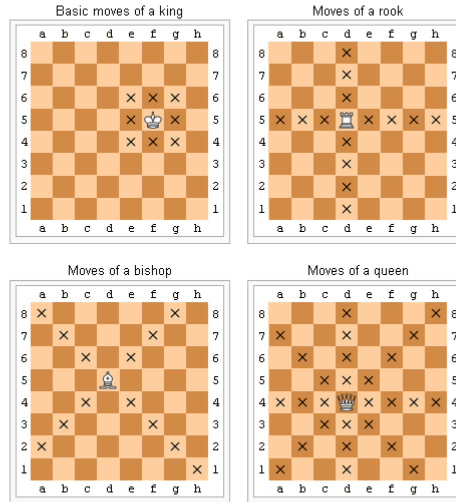
Becomes (implicitly):

```
public class C1 extends C2 {
  // C1 fields
  public C1() {
    super();
  }
  // C1 methods
  ...
```

- If we declare a sub-class constructor, but don't say explicitly `super()` as its first instruction, the compiler will automatically insert `super()` as the first instruction
- If there is no empty constructor in the super-class, the sub-class code will not compile!

Sub-classing methods

Example: a `Rook` is-as `ChessPiece`.
It inherits a `move(x,y)` method. But, being a `Rook`, its movements are restricted. We may want to create a new, `Rook`-specific `move()` method

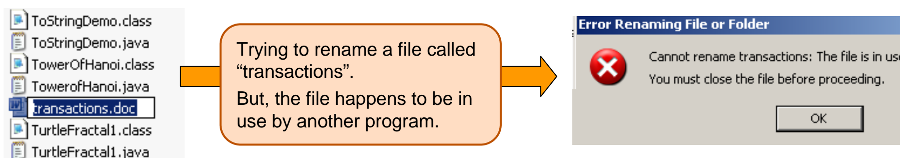


The sub-class inherits all the public methods of the base-class.
For each inherited method, the sub-class code can either:

- Use the inherited method as-is
- Overload the method
- Override the method:
  1. Keep the method signature
  2. Re-write its implementation.

Example: file management (the story)

Typically, a file system lets users access a file only if it in a stable state. Example:



"Stable state" depend on the application. Examples:

- Operating system: As long as some program is doing something to a file, we don't want to let any other program touch this file.
- Transaction processing: As long as some user reserves a seat in a flight, we don't want other users to access the reservations file

Typical solution: the file system designer defines a Boolean attribute that stores the file state (open / not open). Clients can access the file only if it's not open.

Example: file management

Server

```
// File implementation
public class File {

  private String name;
  private boolean isOpen;

  public File(String name) {
    this.name = name;
    this.isOpen = true;
  }

  public void open() {
    if (isOpen)
      // code for denying access
    else
      isOpen = true;
  }

  public void close() {
    isOpen = false;
  }

  // Other File methods

}
```

Client

```
public class FileDemo {

  public static void main (String args[]) {
    File f = new File("reservations");
    // Add some data to the file
    // File processing operations ...
    f.close();

    // Do something else, unrelated to the file
    ...

    // More file processing:
    f.open();
    // File processing operations
    f.close();

    // Etc.
  }
}
```

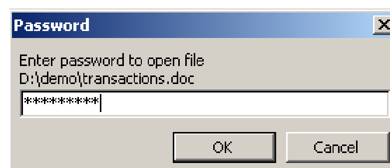
Access control:

- When a client wants to access a file, it calls `f.open()`
- When file processing ends, the client calls `f.close()`
- This allows safe file sharing among clients.

Example: file management (the story, take 2)

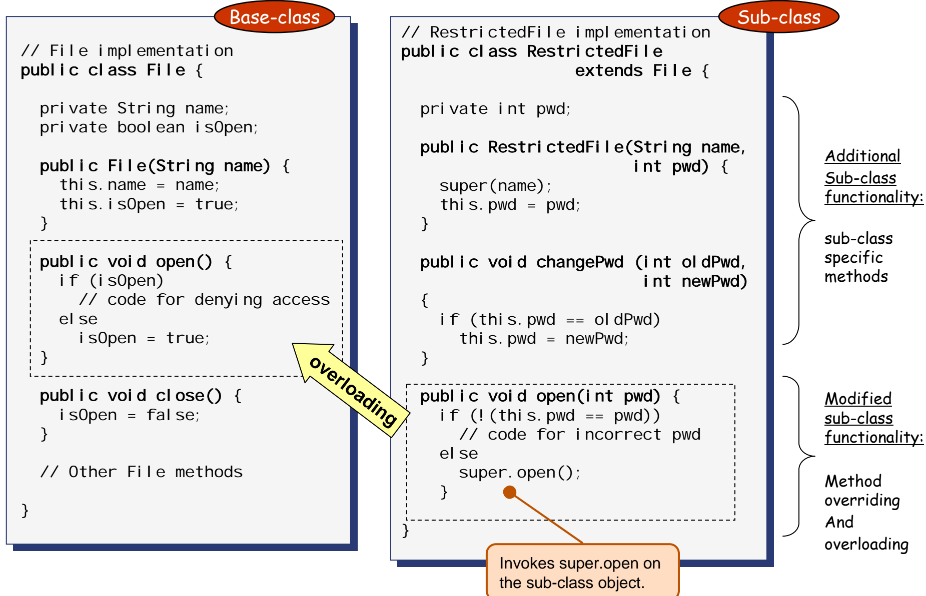
Password control:

In addition to access control, which is mandatory for all files, we may want to allow creation and access of password-protected files:

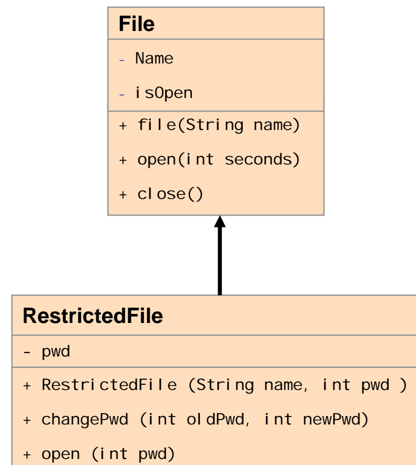


- A protected file should have all the features of a regular file, plus password protection
- This extension can be handled by sub-classing the `File` class.

Example: file management



Example: file management – inheritance hierarchy



Outline

- Why inheritance?
  - Motivation
  - Examples
- Inheritance anatomy
  - Constructors
  - Methods
  - Example
- The Object class
- Method overriding
- Inheritance implications
  - Narrowing / Widening
  - Run-time types
  - Virtual method calling
- Packages
- Visibility modifiers
- Design issues
  - Multiple inheritance
  - Overloading vs. overriding
  - When to sub-class?
- Benefits of inheritances

`Object`: the parent class of all Java classes

java.lang
Class Object

From the `Object` class API

`java.lang.Object`

public class **Object**

Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class.

Method Summary	
protected `Object`	**`clone`**`()` Creates and returns a copy of this object.
`boolean`	**`equals`**`(Object obj)` Indicates whether some other object is "equal to" this one.
protected `void`	**`finalize`**`()` Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
`Class`	**`getClass`**`()` Returns the runtime class of an object.
`int`	**`hashCode`**`()` Returns a hash code value for the object.
`void`	**`notify`**`()` Wakes up a single thread that is waiting on this object's monitor.
`void`	**`notifyAll`**`()` Wakes up all threads that are waiting on this object's monitor.
`String`	**`toString`**`()` Returns a string representation of the object.

- All classes implicitly extend the class `Object`
- Thus, the `Object` methods are inherited by all Java classes
- Designers of sub-classes often override these base methods.

The `toString` method

java.lang
Class Object

From the `Object` class API

java.lang.Object

toString

`public String toString()`

Returns a string representation of the object. In general, the `toString` method returns a string that "textually represents" this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method.

The `toString` method for class `Object` returns a string consisting of the name of the class of which the object is an instance, the at-sign character `@', and the unsigned hexadecimal representation of the hash code of the object. In other words, this method returns a string equal to the value of:

- When you write a class, you often override the `toString()` method in order to create a suitable textual representation of objects of that class.
- What is expected from overriding methods? See the base class API.

Overriding `toString`

Base class

```
// Represents a point on a grid.
public class Point {

  // The coordinates of the point
  private int x, y;

  // Constructs a point
  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public String toString() {
    return "(" + x + "," + y + ")";
  }
}
```

```
public enum Color
    {red, blue, green, yellow};
```

Sub class

```
// Represents a colored point on a grid.
public class ColoredPoint extends Point {

  private Color color;

  // Constructs a new colored point
  public ColoredPoint(int x, int y, Color color) {
    super(x,y);
    this.color = color;
  }

  public String toString() {
    return super.toString() + " "
           + color.toString();
  }

  . . .
}
```

overriding

The `equals` method

java.lang
Class Object

From the `Object` class API

`java.lang.Object`

equals

```
public boolean equals(Object obj)
```

The `equals` method for class `Object` implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values `x` and `y`, this method returns `true` if and only if `x` and `y` refer to the same object (`x == y` has the value `true`).

- Like `toString()`, class designers normally override `equals()` to suit their needs.

Overriding `equals`

```
public class Point {
  // The coordinates of the point
  private int x, y;
  ...
  public boolean equals(Object other) {
    if (other instanceof Point) {
      Point otherPoint = (Point)other;
      return (x == otherPoint.x && y == otherPoint.y);
    }
    return false;
  }
  ...
}
```

`x instanceof C`
Means
"x refers to an object of type C"

```
public class EqualsDemo {
  public static void main(String[] args) {
    Object obj1 = new Object();
    Object obj2 = new Object();
    Point p1 = new Point(10, 20);
    Point p2 = new Point(10, 20);
    System.out.println(obj1.equals(obj2));
    System.out.println(p1 == p2);
    System.out.println(p1.equals(p2));
  }
}
```

This example illustrates how to design a domain-specific definition of equivalence, overriding the base definition of `equals`

C:\WINDOWS\system32\cmd.exe

```
D:\demo>java EqualsDemo
false
false
true
```

Further sub-classing and overriding

base-class

```
public class Point {
  // The coordinates of the point
  private int x,y;
  ...
  public boolean equals(Object other) {
    if (other instanceof Point) {
      Point otherPoint = (Point)other;
      return (x == otherPoint.x && y == otherPoint.y);
    }
    return false;
  }
  ...
}
```

overriding

sub-class

```
public class ColoredPoint extends Point {

  private Color color;
  ...
  public boolean equals(Object other) {
    if (other instanceof ColoredPoint) {
      ColoredPoint otherPoint = (ColoredPoint)other;
      return (super.equals(other) && color == otherPoint.color);
    }
    return false;
  }
  ...
}
```

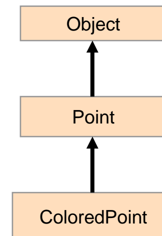

Overriding rules

- The overridden methods of the base-class can still be accessible from the sub-class code using the syntax `super.method(...)`
- The *contract* of a method states, among other things, what is expected of overriding implementations of the method
- The method signature (name, number and type of the parameters) of the overriding method must be the same as that of the overridden method
- The overriding method can have a different `throws` clause as long as it doesn't declare any types not declared by the `throws` clause in the overridden method
- The access modifier of the overriding method can allow more access than the overridden method, but not less. For example, a `protected` method in the base-class can be made `public` but not `private`.

Inheritance hierarchy

The is-a relationship is transitive:

- `ColoredPoint` is-a `Point`
- `Point` is-a `Object`
- `ColoredPoint` is-a `Object`



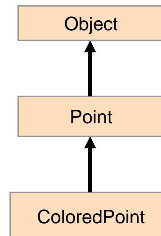
When you call a method on some object:

- If the compiler finds a matching method declaration in the object's class, it uses it
- Otherwise it searches in its immediate parent
- If not found, it searches in the parent's immediate parent, and so on,
- All the way up to `java.lang.Object`

Outline

- Why inheritance?
  - Motivation
  - Examples
- Inheritance anatomy
  - Constructors
  - Methods
  - Example
- The Object class
- Method overriding
- ⇨ Inheritance implications
  - Narrowing / Widening
  - Run-time types
  - Virtual method calling
- Packages
- Visibility modifiers
- Design issues
  - Multiple inheritance
  - Overloading vs. overriding
  - When to sub-class?
- Benefits of inheritances

Different views on the same object



Inheritance enable us to treat objects from multiple perspectives

For example, a `ColoredPoint` object can be treated as a …

- `ColoredPoint:` The narrowest, and most specific, point of view
- `Point:` Less specific: we forget about the special characteristics of the object as a `ColoredPoint`
- `Object:` the widest point of view.

Narrowing / widening

Sometimes we want to "widen-up" an object and treat it like its parent object

Sometimes we want to "narrow down" an object and treat it like one of its child objects

```
// Recall that ColoredPoint extends Point.
Point p1;
ColoredPoint p2 = new ColoredPoint(2, 3, Color.red);
p1 = (Point) p2;          // p2 is widened up
p1 = p2;                  // Likewise
p2 = (ColoredPoint) p1 ;  // p1 is narrowed down
```

In Java:

- Narrowing requires explicit casting
- Widening can be done in several different ways.

Widening via parameter passing

case-class

```
// A point on a grid
public class Point {
  private int x,y;
  ...
  // Computes the distance from another point
  public double distanceFrom(Point p) {
    int dx = x - p.x;
    int dy = y - p.y;
    return Math.sqrt(dx * dx + dy * dy);
  }
  ...
}
```

```
public enum Color
      {red, blue, green, yellow};
```

sub-class

```
// A colored point on a grid
public class ColoredPoint extends Point {
  private Color color;
  ...
  // Constructs a new colored point
  public ColoredPoint(int x, int y, Color color) {
    super(x,y);
    this.color = color;
  }
  ...
```

client code

```
...
Point p1;
ColoredPoint p2;
p1 = new Point(2, 3);
p2 = new ColoredPoint(5, 6, Color.red);
double d = p1.distanceFrom(p2);
```

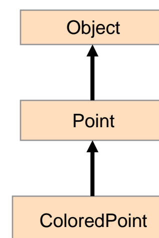
`distanceFrom()` expects to get a `Point` parameter;
Instead, it gets a `ColoredPoint` argument (`p2`).

That's OK – `p2`. has been widened by the type of the formal parameter.

Run-time type

Consider the following code:

```
Point p;
ColoredPoint cp = new ColoredPoint(10, 20, Color.green);
p = cp;
```



Observations:

The compile-time type of `p` is `Point`

But, `p` now contains a reference to a `ColoredPoint` object

Terminology: we say that the run-time type of `p` is `ColoredPoint`

Run-time type

- The run-time type is always some subclass of the compile-time type
- The same object can have different run-time types in different program runs.

Virtual method calling

```
public class Animal {
  public String eats() { return "food"; }
}
```

```
public class Cow extends Animal {
 public String eats() { return "grass"; }
}
```

```
public class NarrowingDemo {
  public static void main(String[] args) {
    Animal a;
    Cow c = new Cow();
    a = c;
    System.out.println(a.eats());
  }
}
```

- In OO, (non-static) methods are always invoked on some object
- If the object has been narrowed or widened, which method implementation should be used?

Virtual method calling:
The method to be invoked is determined by the run-time type of the object and not by its compile-time type

In Java, all methods invocations are virtual.

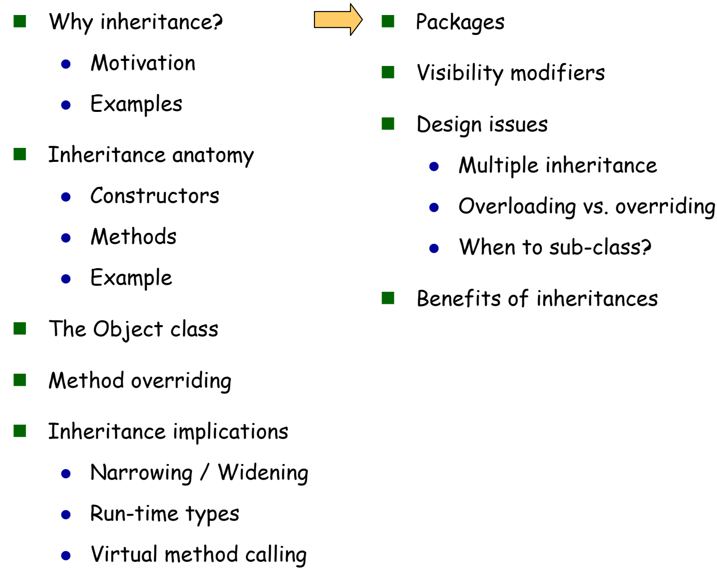
```
C:\WINDOWS\system32\cmd.exe
D:\demo>java NarrowingDemo
grass
```


InstanceOf

- Problem: If the run-time type of an object is unknown, how can we tell which methods we are allowed to invoke on it?
- Solution: The run-time type can be checked using the operator `instanceOf`.

```
public class RunTimeTypeDemo {

  public static void main(String[] args) {
    File f;
    if (Math.random() >= 0.5)
      f = new File("reservations");
    else
      f = new RestrictedFile("reservations", 56789);
    f.close();

    // Many lines of code later, we want to process the f file.
    // How shall we open it? Is it a restricted file, or just a file?

    if (f instanceof RestrictedFile) {
      RestrictedFile rf = (RestrictedFile) f;
      rf.open(56789);
    }
    else
      f.open();
  }
}
```

Outline



Packages

- If you have various classes that do something together, you may want to put them in a package
- Package = a collection of classes and interfaces providing access protection and namespace management
- Classes in a package do not have to be a part of the same inheritance tree

How to group classes into a package:

1. Write `package packageName` at the beginning of each class file
2. Put all class files under a directory whose name is `packageName`.
   The directory that contains this directory should be in our `CLASSPATH`.

Class visibility:

- A class declared as `public` can be used from outside the package
- A class declared without a `public` modifier can be used only inside the package.

Visibility modifiers

The visibility modifier of a member determines which other classes can access the member. There are four possibilities:

`public`: Accessible to any class

`protected`: (1) Accessible to any class in the same package as this class
(2) Accessible to any sub-class of this class

None (default)): Accessible to any class in the same package as this class

`private`: Accessible to this class only

Access Levels

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Best practice advice

- Use `protected` if you want to expose a member to sub-classes and hide it from the rest of the world
- Protected methods are useful for customizing the behavior of our class by sub-classing.
- Avoid defining too many protected variables: it hurts encapsulation
- A `protected` member is part of the class interface; Therefore it must be documented in the class API.

Protected fields: usage example

Base class

```
package screen;

// Represents a point on a grid.
public class Point {

  // The coordinates of the point
  protected int x,y;

  // Constructs a point
  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  // public getx() and setx() methods
  // public gety() and sety() methods

  ...

}
```

Sub class

```
package screen;

// Represents a colored point on a grid.
public class ColoredPoint extends Point {

  private Color color;

  // Constructs a new colored point
  public ColoredPoint(int x, int y, Color color) {
    super(x,y);
    this.color = color;
  }
  . . .
  public void draw() {
    // This method will need to refer to x,y
    // and it may be cumbersome to do it
    // through accessor methods
  }
  . . .
}
```

The `final` modifier

The `final` modifier can be used for classes, methods and variables; in each case it has a different meaning:

- A `final` class cannot have derived classes
- A `final` method cannot be overridden
- A `final` variable can be initialized only once
  - If the variable is `static` you must specify its value in the declaration and it becomes a constant
  - If the variable is not static, you should either specify a value in the declaration or in the constructor, but only once, and only in either of these two places .

Outline

- Why inheritance?
  - Motivation
  - Examples
- Inheritance anatomy
  - Constructors
  - Methods
  - Example
- The Object class
- Method overriding
- Inheritance implications
  - Narrowing / Widening
  - Run-time types
  - Virtual method calling
- Packages
- Visibility modifiers
- Design issues
  - Multiple inheritance
  - Overloading vs. overriding
  - When to sub-class?
- Benefits of inheritances

When to derive a sub-class?

That's a design issue.

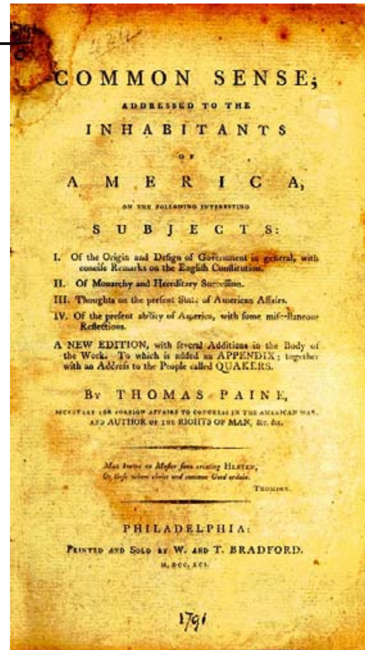
Use inheritance when it makes sense from the abstraction standpoint

Example: we wish to implement dashed lines. Two options:

- Add a `dashed` field to the `Line` class
- Create a `DashedLine` class that extends `Line`

What to do?

- Use judgment and common sense
- Consult the abstraction.



COMMON SENSE;
ADDRESSED TO THE
INHABITANTS
OF
AMERICA,
ON THE FOLLOWING INTERESTING
SUBJECTS:

I. Of the Origin and Design of Government in general, with concise Remarks on the English Constitution.
II. Of Monarchy and Hereditary Succession.
III. Thoughts on the present State of American Affairs.
IV. Of the present ability of America, with some miscellaneous Reflections.

A NEW EDITION, with several Additions in the Body of the Work. To which is added an APPENDIX; together with an Address to the People called QUAKERS.

BY THOMAS PAINE,
AND AUTHOR OF THE RIGHTS OF MAN, &c. &c.

PHILADELPHIA:
PRINTED AND SOLD BY W. AND T. BRADFORD.
M, DCC, XCI.

1791

To overload, or to override?

Overloading:

- Multiple methods, each with the same name but different signatures
- Enables to define a similar operation in different ways for different data

Overriding:

- Multiple methods, one in a base-class and the other in sub-classes, each having the same signature but different implementations
- Enables to define a similar operation in different ways for different object types.

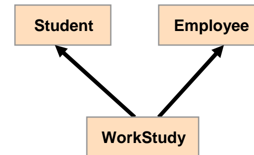
Is this overloading or overriding?

Multiple inheritance

A university typically employs some of its students in all sorts of jobs

Such students / employees are called "work-study"

How to model a WorkStudy abstraction?



<u>Multiple inheritance:</u>

A class is allowed to extend more than one base-class

- Supported by some OO languages, e.g. C++
- Not supported in Java
- But … a workaround is possible, using interfaces.

Advantages of inheritance

- Code re-use: you implement only the *necessary additional functionality*
  - Time
  - Money
  - Ease of use: if users are familiar with the base-class, they need learn only the additional functionality
  - Debugging / QA only the added functionality
  - Errors minimization
  - Inherited maintenance: the sub-class inherits all the improvements made to the base class
  - Compactness: the code is smaller and easier to handle
  - Inheritance is typically part of the abstraction
- Obligations of the super-class:
  - Interface
  - API
  - Documentation.