Lecture 10-1

Inheritance

Interfaces: an example where they come to play

"Interface" is an OO artifact that serves many purposes. Here is one of them:

A large scale software development project consists of many classes, designed and implemented by many developers

To promote consistency, coherence, correctness, and best practice, the project architect can diactate certain design rules

For example, she can insist that certain classes will have certain methods, and that the methods will have certain signatures – even if she is not the designer or implementor of these classes

Case in point: iterators. An iterator is an object that provides iteration services over the items of a collection. Which methods should the iterator have? And how should we name them?

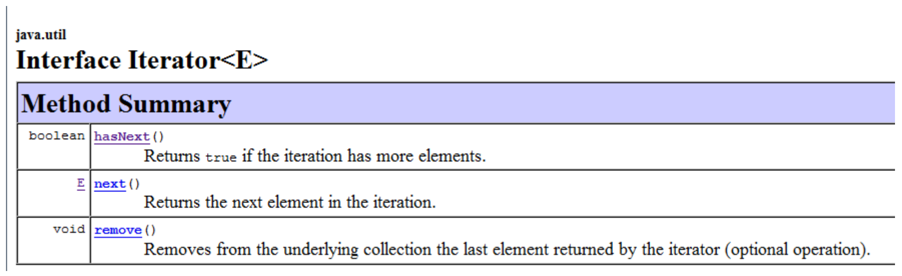
- `hasNext()` , `hasMore()` , `MoreWorkToDo()`, ... ?
- `next()`, `getNext()`, ... ?
- `advance()`, `advanceNext()`, `moveNext()`, ... ?

The architect can enforce a design convention that iterators will have to follow

This is done using a programming artifact called interface.

Example: Java's Iterator interface

Java specirfies an interface describing a standard mechanism to move through a collection of objects, one object at a time:


java.util
Interface Iterator<E>

Method Summary	
`boolean`	`hasNext()` Returns `true` if the iteration has more elements.
`E`	`next()` Returns the next element in the iteration.
`void`	`remove()` Removes from the underlying collection the last element returned by the iterator (optional operation).

Java programmers can write two kinds of iterators:

- "Free style" itertaors: don't follow the above design
- Iterators that implement the `java.util.Iterator` interface: must follow the design stated in the interface.

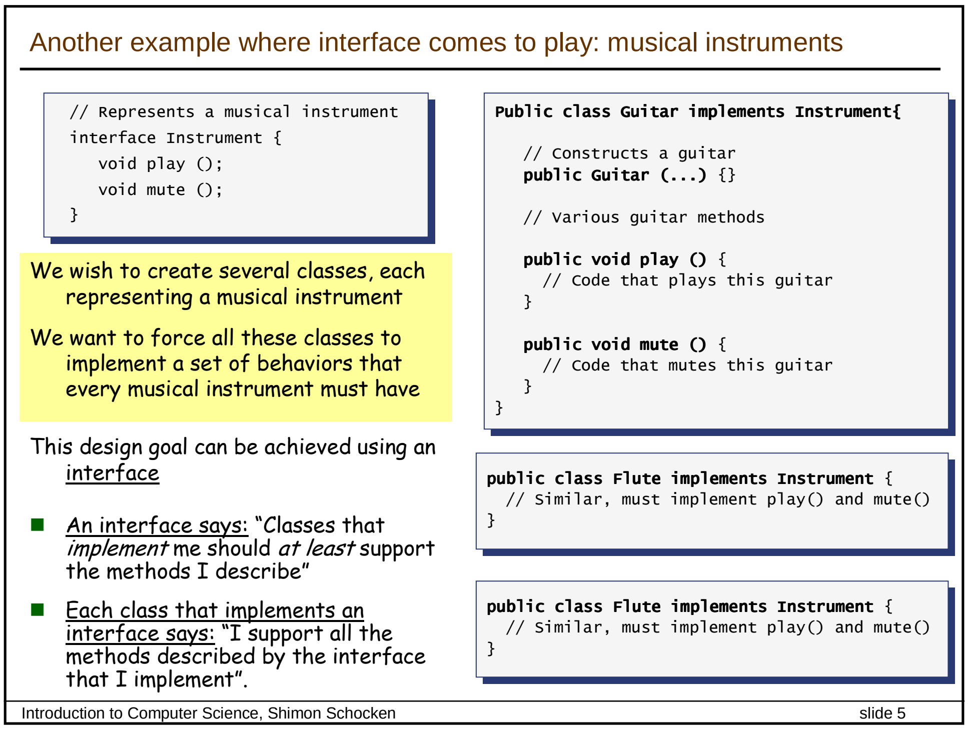
List iterator

List iterator implementation ("free style")

```
public class ListIterator {

    // current position in the list
    Item current;

    public ListIterator (Item item) {
        current = item;
    }

    public boolean hasNext () {
        return !(current == null);
    }

    public Item getNext () {
          return current;
    }

    public Item advance () {
          current = current.next;
    }

}
```

A list iterator implementation that implements `Iterator`

```
import java.util.Iterator;

public class ListIterator implements Iterator {

    // current position in the list
    Item current;

    public ListIterator (Item list) {
          current = list;
    }

    public boolean hasNext () {
          return !(current == null);
    }

    public Item next () {
          Item item = current;
          current = current.next;
          return item;
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }

}
```



Another example where interface comes to play: musical instruments

```
// Represents a musical instrument
interface Instrument {
   void play ();
   void mute ();
}
```

We wish to create several classes, each representing a musical instrument

We want to force all these classes to implement a set of behaviors that every musical instrument must have

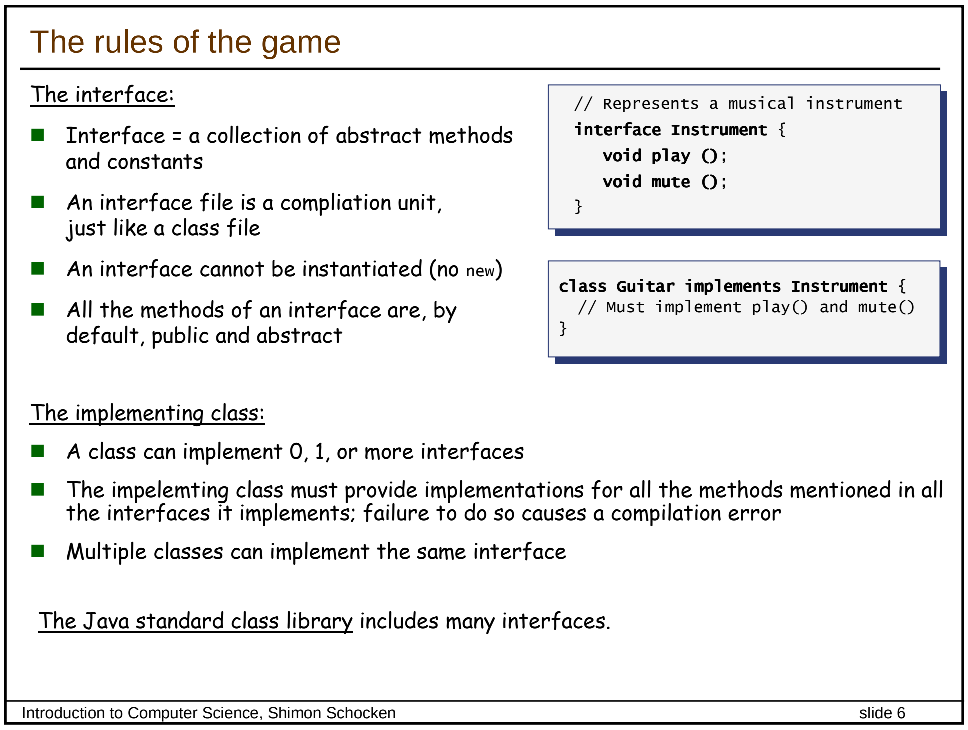
This design goal can be achieved using an <u>interface</u>

- <u>An interface says:</u> "Classes that *implement* me should *at least* support the methods I describe"
- <u>Each class that implements an interface says:</u> "I support all the methods described by the interface that I implement".

```
Public class Guitar implements Instrument{

   // Constructs a guitar
   public Guitar (...) {}

   // Various guitar methods

   public void play () {
     // Code that plays this guitar
   }

   public void mute () {
     // Code that mutes this guitar
   }
}
```

```
public class Flute implements Instrument {
  // Similar, must implement play() and mute()
}
```

```
public class Flute implements Instrument {
  // Similar, must implement play() and mute()
}
```



The rules of the game

<u>The interface:</u>

- Interface = a collection of abstract methods and constants
- An interface file is a compliation unit, just like a class file
- An interface cannot be instantiated (no `new`)
- All the methods of an interface are, by default, public and abstract

```
// Represents a musical instrument
interface Instrument {
   void play ();
   void mute ();
}
```

```
class Guitar implements Instrument {
  // Must implement play() and mute()
}
```

<u>The implementing class:</u>

- A class can implement 0, 1, or more interfaces
- The impelemting class must provide implementations for all the methods mentioned in all the interfaces it implements; failure to do so causes a compilation error
- Multiple classes can implement the same interface

<u>The Java standard class library</u> includes many interfaces.

Set iterator abstraction and usage

`Set` API:

```
public class Set {

  public Set ()

  public void insert (int x)

  public boolean contains (int x)

  ...

  public Iterator iterator()

  ...
}
```

```
Set s = new Set();
for (int i = 0; i < 10; i++)
    s.insert(i*2);

// Prints the set's contents:
for (Iterator i = s.iterator(); i.hasNext();) {
    System.out.println(i.next());
}

// Prints 2 4 6 8 ...
```

How does the client programmer know which methods the `iterator` object provides?

She sees from the `Set` API that it is an Iterator, so she consult the Java `Iterator` API and know what to expect.

Set iterator implementation

```
import java.util.Iterator;
import java.util.NoSuchElementException;

public class Set {

  private int[] elements;
  private int size;

  // The regular Set methods come here

  public Iterator iterator() {
    return new SetIterator(elements, size);
  }
}

class SetIterator implements Iterator {

  // See definition on the right

}
```

```
class SetIterator implements Iterator {

  private int[] elements;
  private int size;
  private int index;

  SetIterator(int[] elements, int size) {
      this.elements = elements;
      this.size = size;
      this.index = 0;
  }

  public boolean hasNext() {
      return index < size;
  }

  public Object next() {
      if (!hasNext()) {
          throw new NoSuchElementException();
      }
      return elements[index++];
  }

  public void remove() {
    throw new UnsupportedOperationException();
  }
}
```

- Note how the iterator gives the client encapsulated access to the private data
- Note that `SetIterator` can be treated as an `Iterator`. That's an example of inheritance.

Outline

Interfaces

- Motivation
- Examples

Inheritance

- Motivation
- Examlpes

Sub-classing

- Constructors
- Methods

To be continued …

Why inheritance?

- We are asked to write an application that manages and displays analog clocks and digital clocks
- We notice that both abstractions have something in common: a clock behavior
- In other words:
  - An analog (digital) clock *is a clock*
  - It has all the basic features of a clock + some analog- (digital-) specific features

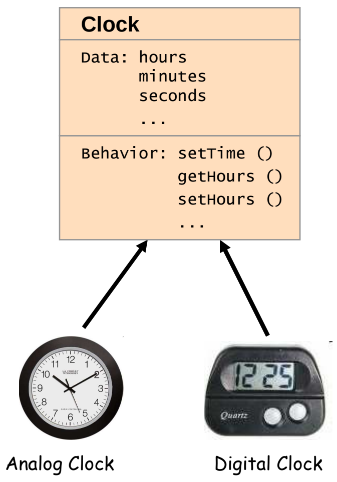
Inheritance

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

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

We could design other clock variants similarly

Implications: less work, less bugs, more consistency.

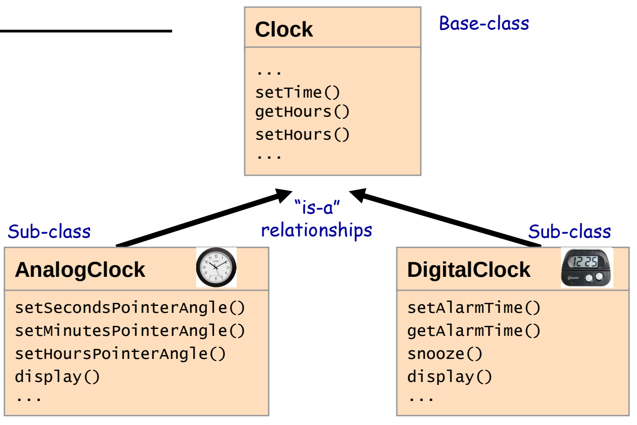


Clock

```
Data: hours
      minutes
      seconds
      ...
Behavior: setTime ()
          getHours ()
          setHours ()
          ...
```

Example: Clocks

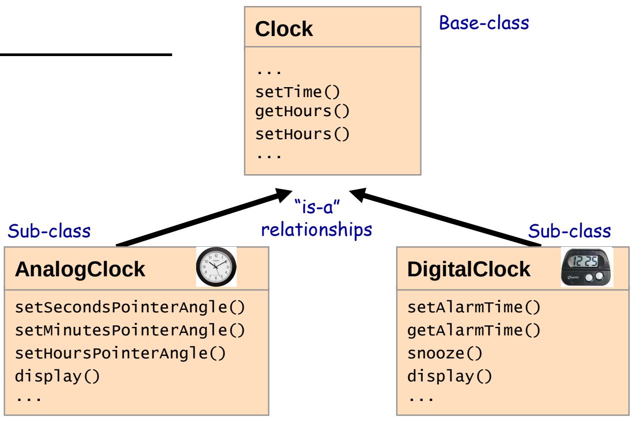
- The derived sub-class *inherits* the non-private members of the base-class
- Usually, the sub-class will have additional, sub-class-specific members



OO terminology:

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

Client view



```
public class SomeClass {
    ...
    DigitalClock dc = new DigitalClock();
    ...
    dc.setTime(10,0,0);
    ...
    dc.setAlarmTime(5,30,0);
    ...
}
```

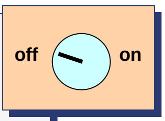
Clients of the sub-class can invoke methods of both the sub-class and its base-class

Example: Switches

Base-class abstraction

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

  // The state of this switch
  private boolean isOn;

  public Switch (boolean isOn) {
    setIsOn(isOn);
  }

  public boolean isOn () {
    return isOn;
  }

  public void setIsOn (boolean isOn) {
    this.isOn = isOn;
  }

}
```

Switch

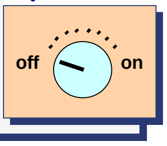


Best practice advice:
use inherotance only when the sub-class is a sub-type of the base class.

Sub-class implementation

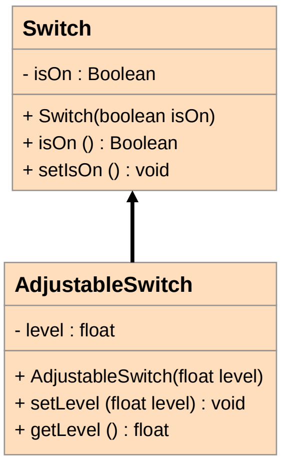
```
public class AdjustableSwitch extends Switch {

   private float level;

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

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

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

Adjustable switch



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

The inheritance hierarchy



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

The non-private members of the base-class may be used just like sub-class members

Outline

Interfaces
- Motivation
- Examples

Inheritance
- Motivation
- Examlpes

➡ Sub-classing
- Constructors
- Methods

To be continued ...

Sub-class constructors

Constructors are not inherited.

A sub-class constructor always has the same logic:

1. First, it must invoke 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. Second, it may do some additional sub-class construction work.

base-class

```
public class Switch {
  private boolean isOn;

  public Switch (boolean isOn) {
    setIsOn(isOn);
  }

  // Other Switch methods
```

sub-class

Example:

```
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-class 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)
}
```

javac

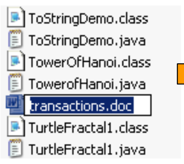
Becomes (implicitly):

```
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 argument-less constructor in the super-class, the sub-class code will not compile!

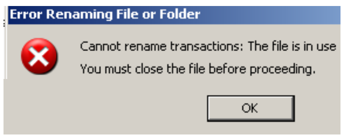
Example: file management (the story)

<u>Required:</u> a system that lets users access a file only if it in a stable state.



Trying to rename a file called "transactions".

But, the file happens to be in use by another program.


Error Renaming File or Folder

Cannot rename transactions: The file is in use

You must close the file before proceeding.

"Stable state" depend on the application. Examples:

- Operating system: As long as some program is doing something to a file, no other program is allowed to access this file
- Transaction processing: As long as some user reserves a seat in a flight, no other user is allowed to access the reservations file

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

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

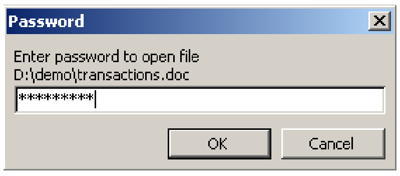
```
File f = new File("reservations");
// Some file processing code
file.close();
...

f.open();
// Some file processing code
f.close();
...
```

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.

File management (the story continues)

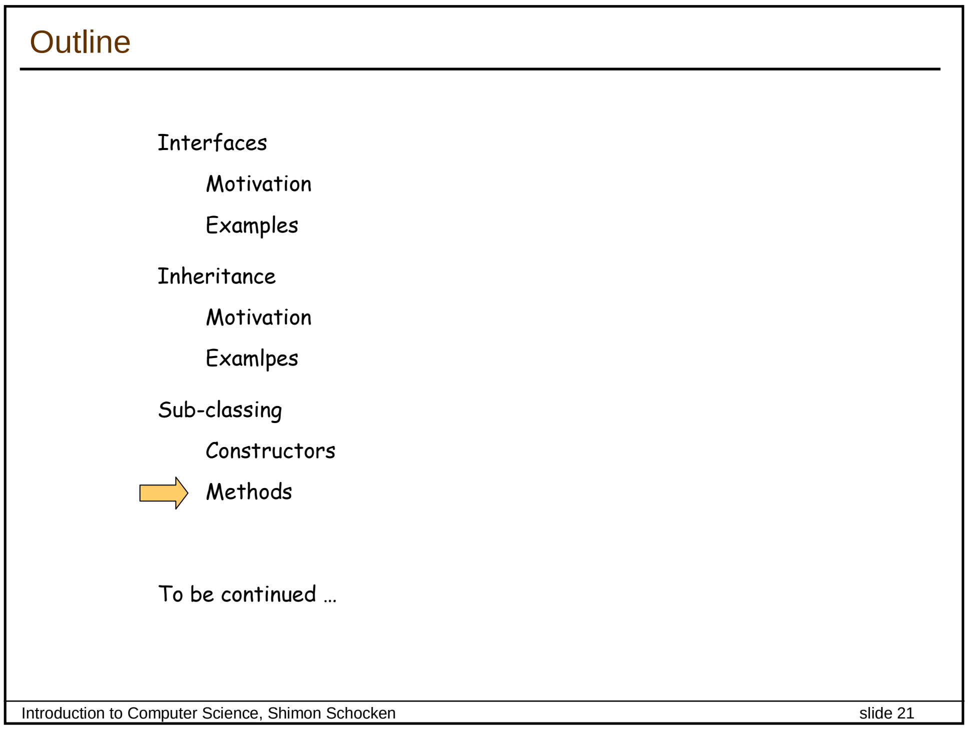
Required: In addition to access control, which is mandatory for all files, we want to allow creation and access of password-protected files:


Password

Enter password to open file
D:\demo\transactions.doc

****

- 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.



Outline

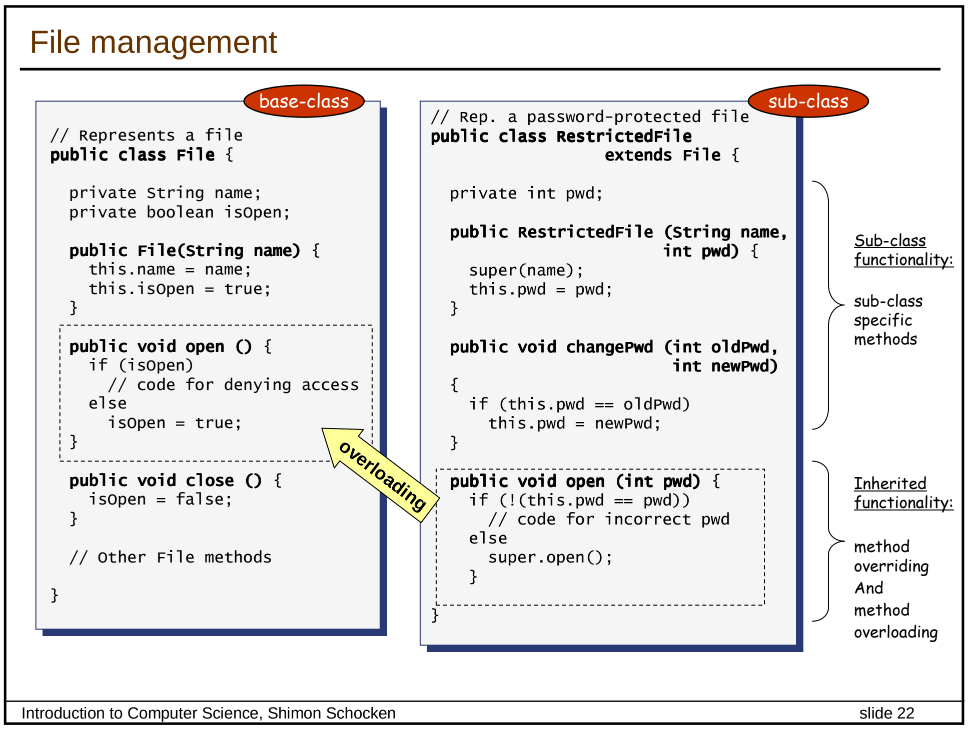
Interfaces
- Motivation
- Examples

Inheritance
- Motivation
- Examlpes

Sub-classing
- Constructors
- Methods

To be continued …

Introduction to Computer Science, Shimon Schocken — slide 21



File management

base-class

```
// Represents a file
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

}
```

sub-class

```
// Rep. a password-protected file
public class RestrictedFile
                    extends File {

  private int pwd;

  public RestrictedFile (String name,
                        int pwd) {
    super(name);
    this.pwd = pwd;
  }

  public void changePwd (int oldPwd,
                         int newPwd)
  {
    if (this.pwd == oldPwd)
      this.pwd = newPwd;
  }

  public void open (int pwd) {
    if (!(this.pwd == pwd))
      // code for incorrect pwd
    else
      super.open();
    }
}
```

Sub-class functionality:

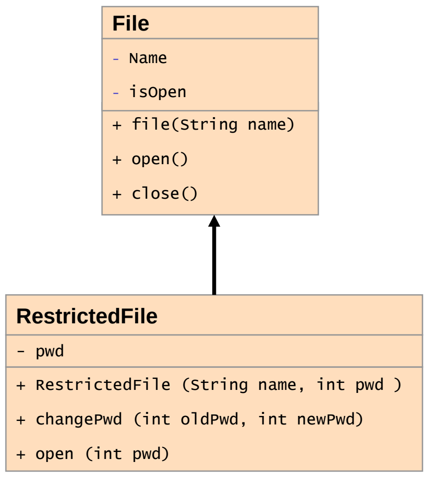
sub-class specific methods

Inherited functionality:

method overriding And method overloading

Introduction to Computer Science, Shimon Schocken — slide 22

The inheritance hierarchy


File
- Name
- isOpen
+ file(String name)
+ open()
+ close()
RestrictedFile
- pwd
+ RestrictedFile (String name, int pwd)
+ changePwd (int oldPwd, int newPwd)
+ open (int pwd)