Object-Oriented Programming: Constructors & Interfaces

Ewan Klein

Inf1 :: 2008/09

Current Topics

HFJ, Chs 8, 9

Abstract Classes: Abstract classes cannot be instantiated; they are just there so we can extend them with concrete classes.

Constructors: A constructor is the code that you run when you instantiate a new object.

Interfaces: An interface is a kind of abstract class, and allows for something like multiple inheritance.

Current Topics

HFJ, Chs 8, 9

Abstract Classes: Abstract classes cannot be instantiated; they are just there so we can extend them with concrete classes.

Constructors: A constructor is the code that you run when you instantiate a new object.

Interfaces: An interface is a kind of abstract class, and allows for something like multiple inheritance.

Abstract classes? Why, oh why?

Home Automation Scenario, 1

- Devices in the home can be controlled by software on a computer.
- To build the software, we need to build a model of the devices.
- Each device needs to support the API (set of methods) below.

Device API

```
switchOn()
switchOff()
getStatus() //Is it on or off?
```

If we want to add a new device to the home automation network, it must abide by this 'contract'.

Home Automation Scenario, 2

HomeDevice

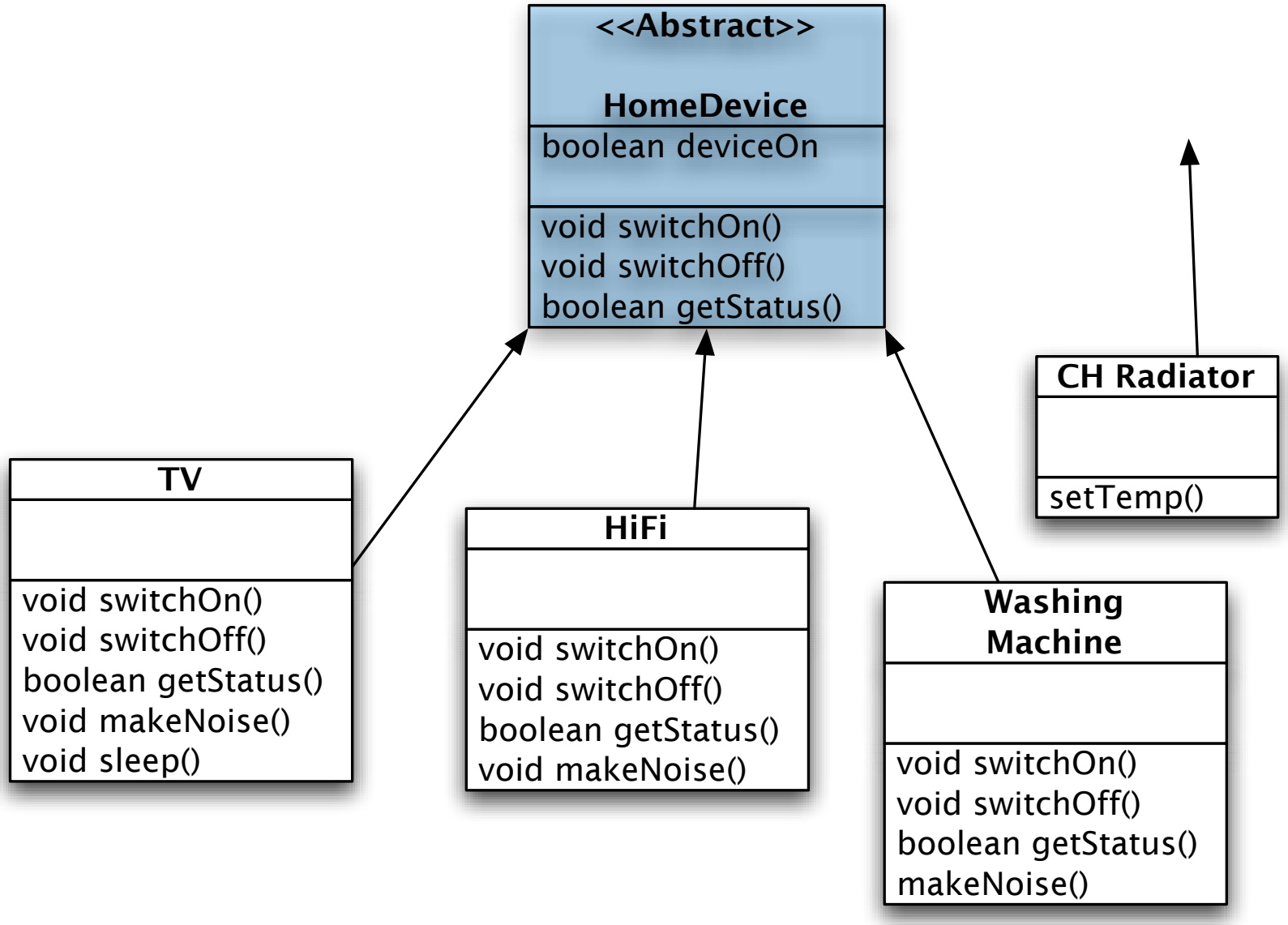
```
public abstract class HomeDevice {
    boolean deviceOn;
    public abstract void switchOn();
    public abstract void switchOff();
    public boolean getStatus() {
       return deviceOn;
}

}
```

Home Automation Scenario, 3

TV

```
public class TV extends HomeDevice {
    public boolean getStatus() {
        return deviceOn;
    }
    public void switchOff() {
        deviceOn = false;
    }
    public void switchOn() {
        deviceOn = true;
    }
}
```

Some Home Devices



In Breach of Contract

So, Abstract Classes

- Use an abstract class when you have several similar classes that:
  - have a lot in common — the implemented parts of the abstract class
  - have some differences — the abstract methods.

Hanging the Radiator

Where does `CH Radiator` belong in the class hierarchy?

Central Heating Radiator

```
public class CHRadiator {
    private int temperature;
    public void setTemp(int temp) {
        temperature = temp;
    }
```

- `Object` is the superclass of every class in Java!
- If a class doesn't explicitly extend some superclass, then it implicitly extends `Object`.

Hanging the Radiator

Where does `CH Radiator` belong in the class hierarchy?

Central Heating Radiator

```
public class CHRadiator  extends Object {
    private int temperature;
    public void setTemp(int temp) {
        temperature = temp;
    }
```

- `Object` is the superclass of every class in Java!
- If a class doesn't explicitly extend some superclass, then it implicitly extends `Object`.

Object as a Polymorphic Type

Javadoc for ArrayList's contains() method

```
public boolean contains(Object elem)
```

Returns `true` if this list contains the specified element.

Parameters: `elem` - element whose presence in this List is to be tested.

Returns: `true` if the specified element is present; `false` otherwise.

- Since every class in Java is a subclass of `Object`, the `contains()` method can take any object as an argument.

Some Methods of `Object`

`Object` defines methods that are available to every class. E.g.,

- `equals(Object o)` — test whether two objects are equal.
- `hashCode()` — numerical ID; equal objects must have equal hash codes.
- `toString()` — returns a textual representation of an object; automatically invoked by methods like `System.out.println()`; default implementation is not very useful.

Overriding toString(), 1

Number Six

```
public class NumberSix {
    public String toString() {
        return ”I am not a number – I am a free man!”;
    }
}
```

Number Two

```
public class NumberTwo {
    public static void main(String[] args) {
        NumberTwo n2 = new NumberTwo();
        NumberSix n6 = new NumberSix();
        System.out.println(”You are Number Two: ” + n2);
        System.out.println(”You are Number Six: ” + n6);
    }
}
```

Overriding toString(), 2

Output

```
You are Number Two: NumberTwo@10d448
You are Number Six: I am not a number - I am a free man!
```

Revisiting Constructors

Instantiating `Bicycle`

```
Bicycle bike = new Bicycle();
```

Default Constructor

```
public Bicycle() {
}
```

- Java compiler adds a default constructor.
- Name (e.g. `Bicycle`) is same as class name.
- No return type.

Make Your Own Constructor, 1

Declared Constructor

```
public class Bicycle {
    private int gears;
    private String brand;
    public Bicycle() {
        gears = 15;
        brand = "Cannondale";
        System.out.printf("Bike: gears=%d, brand=%s",
                          gears,
                          brand);
    }
}
```

Make Your Own Constructor, 2

BikeLauncher

```
public class BikeLauncher {
    public static void main(String[] args) {
        Bicycle myBike = new Bicycle();
    }
}
```

Output

```
Bike: gears=15, brand=Cannondale
```

Using Setters to Initialize Instance Variables, 2

Rectangle1

```
public class Rectangle1 {
    private int height;
    private int width;
    int area = height * width; // Doesn't work!
    public void setHeight(int newHeight) {
        height = newHeight;
    }
    public void setWidth(int newWidth) {
        width = newWidth;
    }
}
```

Using Setters to Initialize Instance Variables, 2

RectangleLauncher

```
public class RectangleLauncher {
    public static void main(String[] args) {
        Rectangle1 r1 = new Rectangle1();
        r1.setHeight(3);
        r1.setWidth(6);
        System.out.println("Area of r1: " + r1.area);
    }
}
```

Output

```
Area of r1: 0
```

Using a Constructor to Initialize Instance Variables, 1

Rectangle2

```
public class Rectangle2 {
    private int height;
    private int width;
    int area;
    // Add a two-parameter constructor
    public Rectangle2(int startHeight, int startWidth) {
        height = startHeight;
        width = startWidth;
        area = height * width;
    }
}
```

Using a Constructor to Initialize Instance Variables, 2

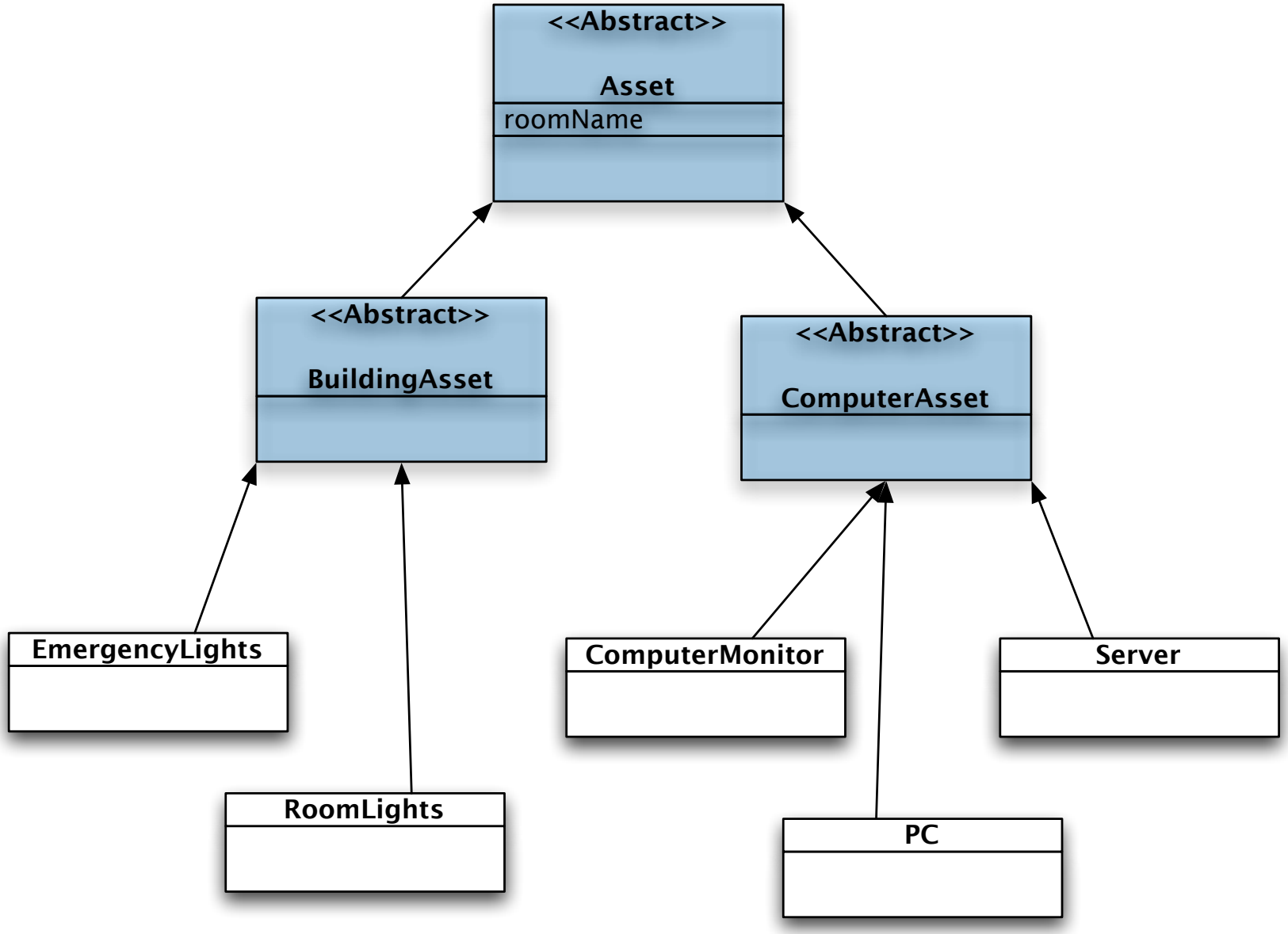
Rectangle1

```
public class RectangleLauncher {
    public static void main(String[] args) {
        Rectangle1 r1 = new Rectangle1();
        r1.setHeight(3);
        r1.setWidth(6);
        System.out.println("Area of r1: " + r1.area);

        Rectangle2 r2 = new Rectangle2(3, 6);
        System.out.println("Area of r2: " +r2.area);
    }
}
```

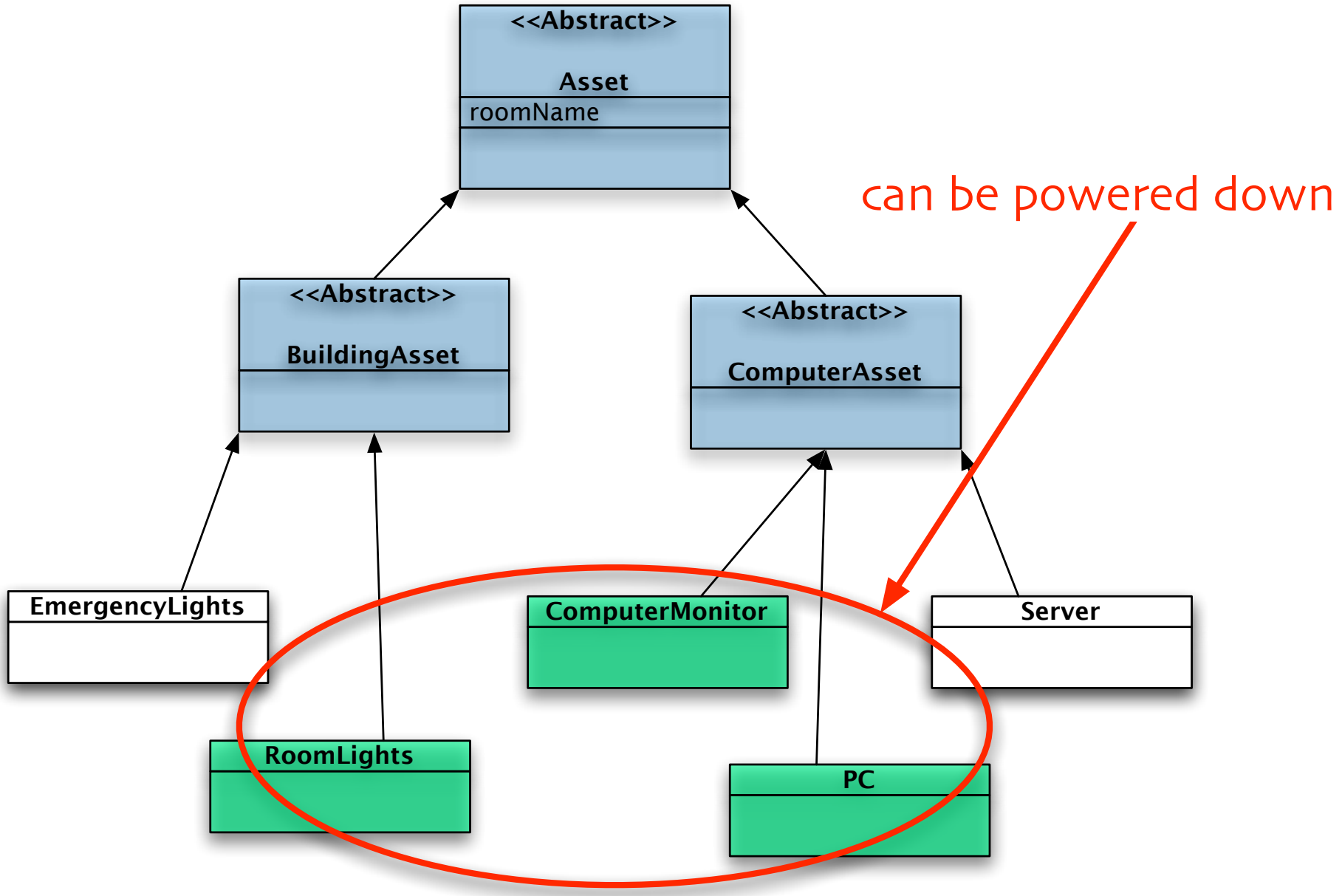
Output

```
Area of r1: 0
Area of r2: 18
```

Appleton Tower Devices



Powersaving for Appleton Tower Devices



Interfaces, 1

- Interfaces are a special kind of abstract class — also for imposing 'contracts'.
- Can cut across class hierarchies — gives the effect of multiple inheritance.
- All methods in an interface are abstract — so these must be implemented in the subclasses that conform.

The PowerSwitchable Interface

```
public interface PowerSwitchable {
            public void powerDown();
            public void powerUp();
}
```

Interfaces, 1

- Interfaces are a special kind of abstract class — also for imposing 'contracts'.
- Can cut across class hierarchies — gives the effect of multiple inheritance.
- All methods in an interface are abstract — so these must be implemented in the subclasses that conform.

The `PowerSwitchable` Interface

```
public interface PowerSwitchable {
    abstract public void powerDown();
    abstract public void powerUp();
}
```

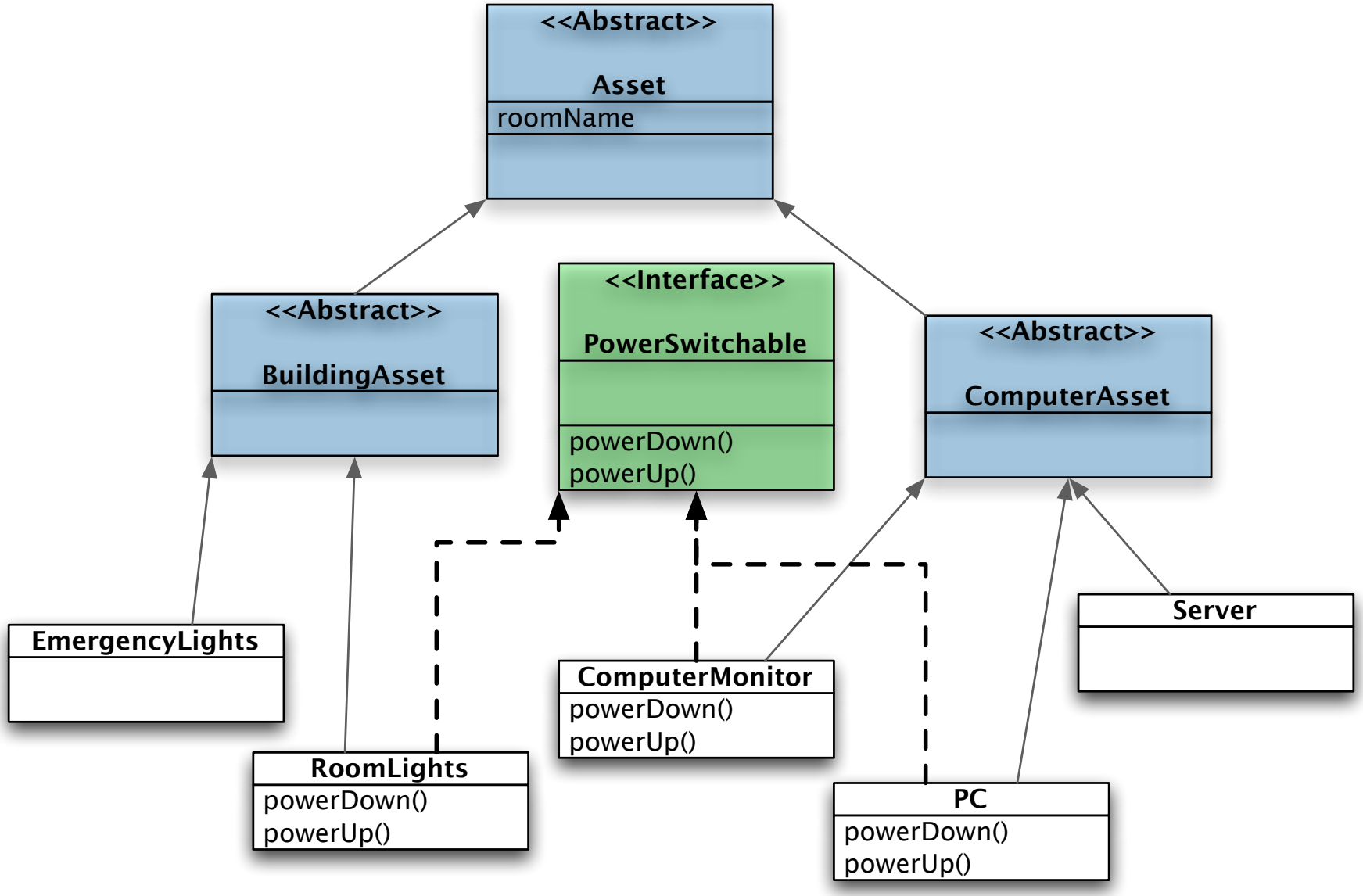
We don't *need* to use the modifier `abstract` in an interface.

Interfaces, 2

Implementing the PowerSwitchable Interface

```
public class Monitor extends ComputerAsset implements PowerSwitchable {
    ComputerMonitor(String roomName) {
        super(roomName);
    }
    public void powerDown() {
        System.out.println("Dousing monitor in " + roomName);
    }
    public void powerUp() {
        System.out.println("Warming up monitor in " + roomName);
    }
    public String toString() {
        return "Monitor in " + roomName;
   }
}
```

Powersaving for Appleton Tower Devices



When to Use What

- Make a subclass when you need to add or extend code that is already present in an existing class.
- Use an abstract class to enforce a contract, and you have a partial implementation; i.e., at least some code that some subclasses can inherit. If an abstract class only has abstract methods, make it into an interface.
- Use an interface to define a 'role' that cuts across existing classes. These can easily be retrofitted to implement a new interface.

Reading, etc

- I can't cover all the material in Chapters 8 & 9 of *Head First Java* in the lectures, so do read them carefully.
- No lecture this Friday!
- Next Tuesday, guest lecture by Stuart Anderson on software engineering and software testing.
- Exam environment: normal DICE environment except firewalled down. All applications should work exactly as normal. You won't have your normal home directory however —- so any local configuration and setup will not be available in the exam.
- Reminder — Revision Tutorial / Lab Sessions:
  - Today, 3-4pm, AT 4.12
  - Thursday 26th Feb, 2-3pm, AT 4.12