INFO1113 Object-Oriented Programming

Week 3A: Objects, Classes and UML
Structuring data and methods

Copyright Warning

Topics

- Classes (s. 5)
- Object attributes (s. 14)
- Instance Methods (s. 29)
- UML Class Diagram (s. 40)

Classes
“Where Reference Types Come From”

Classes

There is a clear distinction between a **primitive type** and a **reference type** but how is the distinction made?

Classes

There is a clear distinction between a **primitive type** and a **reference type** but how is the distinction made?

Reference Types are **Classes**.

We have already used classes within our programs since the start of semester. However now we are able to define our own classes.

Most programming languages have some mechanism of structuring data for reuse.

Refer to Chapter 5.3, pages 362-366, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

But what are they?

What's a class

"A class defines a type or kind of object. It is a blueprint for defining the objects. All objects of the same class have the same kinds of data and the same behaviours. When the program is run, each object can act alone or interact with other objects to accomplish the program's purpose."

Sometimes it is simply conveyed as a **blueprint/template/concept** of an object.

In Java, this is a primary way of structuring data.

Refer to Chapter 5.1, page 303, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

What's a class

Every java program we have ever written so far has included the idea of a class in some form.

However we have never **instantiated** an instance of our own class yet. We have been merely using inbuilt classes within java such as:

- Scanner
- String
- StringBuilder
- **ArrayList**

Objects

The *type* of an object variable is its *class*

```
Point p;
```

Objects are *instance* of a particular *class*.

```
Point topleft = new Point(-1, -1);
Point right = new Point(1, 0);
Point home = new Point(-3388797, 15119390);
```

Objects

The *type* of an object variable is its *class*

```
Point p;
```

Shiny **new** keyword! As discussed last week, this allocates memory and instantiates an object.

Objects are *instance* of a particular *class*.

```
Point topleft = new Point(-1, -1);
Point right = new Point(1, 0);
Point home = new Point(-3388797, 15119390);
```

Objects

The *type* of an object variable is its *class*

```
Point p;
```

Shiny **new** keyword! As discussed last week, this allocates memory and instantiates an object.

Objects are *instance* of a particular *class*.

```
Point topleft = new Point(-1, -1);
Point right = new Point(1, 0);
Point home = new Point(-3388797, 15119390);
```

We have to ask what where this **method** exists

Can I make my own?

Classes

Yes!

However let's start off with a basic **class definition**.

```
public class Cupcake {
    boolean delicious;
    String name;
}
```

We can **instantiate** this class with the following line of code

```
Cupcake c = new Cupcake()
```

Refer to Chapter 5.3, pages 306-307, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Classes

Yes!

However let's start off with a basic **class definition**.

```
public class Cupcake {
    boolean delicious;
    String name;
}
```

This is the **body** of the class. We define **attributes** of **object** within this space.

We can **instantiate** this class with the following line of code

```
Cupcake c = new Cupcake()
```

Refer to Chapter 5.3, pages 306-307, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Yes!

However let's start off with a basic **class definition**.

```
public class Cupcake {
    boolean delicious;
    String name;
}
```

This is the **body** of the class. We define **attributes** of **object** within this space.

We can **instantiate** this class with the following line of code

```
Cupcake c = new Cupcake()
```

Declared a **Cupcake** object.

Java is **allocating** space for a **Cupcake** object and **invoking the constructor** to initialise it.

Refer to Chapter 5.3, pages 306-307, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Classes

Yes!

However let's start off with a basic **class definition**.

```
public class Cupcake {
    boolean delicious;
    String name;
}
```

This is the **body** of the class. We define **attributes** of **object** within this space.

We can **instantiate** this class with the following line of code

```
Cupcake c = new Cupcake();
```

Declared a **Cupcake** object.

Java is **allocating** space for a **Cupcake** object and **invoking the constructor** to initialise it.

Refer to Chapter 5.3, pages 306-307, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Constructors

Every class in **Java** has a **Constructor** even if it is not **explicitly defined**.

Extending our **Cupcake** class we can write our own constructor.

```
public class Cupcake {
    public boolean delicious;
    public String name;

    public Cupcake() { /* NO OP */ }
}
```

Refer to Chapter 6.1, page 419 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Constructors

Every class in **Java** has a **Constructor** even if it is not **explicitly defined**.

Extending our **Cupcake** class we can write our own constructor.

```
public class Cupcake {
    public boolean delicious;
    public String name;

    public Cupcake() { /* NO OP */ }
}
```

This looks like a method but has no **return type?**
The **constructor's** role is to **construct** an **object** of the **type Cupcake.**

Refer to Chapter 6.1, pages 419 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Constructors

Every class in **Java** has a **Constructor** even if it is not **explicitly defined**.

Extending our **Cupcake** class we can write our own constructor.

```
public class Cupcake {
    public boolean delicious;
    public String name;

    public Cupcake() {
        delicious = true; //Aren't they all?
        name = "Chocolate Cupcake";
    }
}
```

We can expand on this to provide **default values**.

Refer to Chapter 6.1, pages 419 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Constructors

Every class in **Java** has a **Constructor** even if it is not **explicitly defined**.

Extending our **Cupcake** class we can write our own constructor.

```
public class Cupcake {
    public boolean delicious;
    public String name;

    public Cupcake(boolean isTasty) {
        delicious = isTasty; //Aren't they all?
        name = "Chocolate Cupcake";
    }
}
```

We can expand on this to provide **default values** and **parameters** for our constructor . We can then invoke the parameter with arguments that relate to the object.

Refer to Chapter 6.1, pages 419 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Let’s make some classes!

Objects

Now using our nice cupcake class, let's see what we can do with it!

```
public class Cupcake {
    public boolean delicious;
    public String name;

    public Cupcake(boolean isTasty) {
        delicious = isTasty;
        name = "Chocolate Cupcake";
    }
}
```

Let's instantiate our own instance!

```
Cupcake mine = new Cupcake(true);
Cupcake toShare = new Cupcake(false);
System.out.println(mine.delicious);
System.out.println(toShare.delicious);
```

Refer to Chapter 6.1, pages 424-429 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Objects

Now using our nice cupcake class, let's see what we can do with it!

```
public class Cupcake {
    public boolean delicious;
    public String name;

    public Cupcake(boo
        delicious = isTasty;
        name = "Chocolate Cupcake";
    }
}
```

We have instantiated a cupcake to the variable **mine** and inputted **true**. This will set the **delicious** attribute to **true.**

Let's instantiate our own instance!

```
Cupcake mine = new Cupcake(true);
Cupcake toShare = new Cupcake(false);
System.out.println(mine.delicious);
System.out.println(toShare.delicious);
```

Refer to Chapter 6.1, pages 424-429 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Objects

Now using our nice cupcake class, let's see what we can do with it!

```
public class Cupcake {
    public boolean delicious;
    public String name;

    public Cupcake(boolean isTasty) {
        delicious = isTasty;
        name = "Chocolate Cupcake";
    }
}
```

I have deliberately provided a bland cupcake to everyone by setting the **isTasty** to **false.**

Let's instantiate our own instance!

```
Cupcake mine = new Cupcake(true);
Cupcake toShare = new Cupcake(false);
System.out.println(mine.delicious);
System.out.println(toShare.delicious);
```

Refer to Chapter 6.1, pages 424-429 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Objects

Now using our nice cupcake class, let's see what we can do with it!

```
public class Cupcake {
    public boolean delicious;
    public String name;

    public Cupcake(boolean isTasty) {
        delicious = isTasty;
        name = "Chocolate Cupcake";
    }
}
```

We can access the attributes of the object by using the .<attribute>

Let's instantiate our own instance!

```
Cupcake mine = new Cupcake(true);
Cupcake toShare = new Cupcake(false);
System.out.println(mine.delicious);
System.out.println(toShare.delicious);
```

Refer to Chapter 6.1, pages 424-429 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Objects

Now using our nice cupcake class, let's see what we can do with it!

```
public class Cupcake {
    public boolean delicious;
    public String name;

    public Cupcake(boolean isTasty) {
        delicious = isTasty;
        name = "Chocolate Cupcake";
    }
}
```

We can access the attributes of the object by using the **.<attribute>**

Let's instantiate our own instance!

```
Cupcake mine = new Cupcake(true);
Cupcake toShare = new Cupcake(false);
System.out.println(mine.delicious);
System.out.println(toShare.delicious);
```

You may have already of picked up on that from using **Scanner** and **String**

Refer to Chapter 6.1, pages 424-429 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Let’s go back and use that class!

Instance methods

We're finally getting rid of the static (training wheels off!)

Syntax:
`[final] return_type name ([parameters])`

An instance method operates on attributes associated with the instance. These methods can **only** be used with an object.

Instance Methods

Let's extend our **Cupcake** class!

```
public class Cupcake {
    public boolean delicious;
    private String name;

    public Cupcake(boolean isTasty, String cupcakeName) {
        delicious = isTasty;
        name = cupcakeName;
    }

    public void setName(String n) { name = n; }

    public String getName() { return name; }
}
```

Refer to Chapter 5.2, pages 342-344 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Instance Methods

Let's extend our **Cupcake** class!

```
public class Cupcake {
    public boolean delicious;
    private String name;

    public Cupcake(boolean isTasty, String cupcakeName) {
        delicious = isTasty;
        name = cupcakeName;
    }

    public void setName(String n) { name = n; }

    public String getName() { return name; }
}
```

A **getter** method has been specified here. This method merely returns the attribute **name.**

Refer to Chapter 5.2, pages 342-344 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Instance Methods

Let's extend our **Cupcake** class!

```
public class Cupcake {
    public boolean delicious;
    private String name;

    public Cupcake(boolean isTasty, String cupcakeName) {
        delicious = isTasty;
        name = cupcakeName;
    }

    public void setName(String n) { name = n; }

    public String getName() { return name; }
}
```

A **getter** method has been specified here. This method merely returns the attribute **name.**

A **setter** method specified. This allows us to modify the **name** attribute

Refer to Chapter 5.2, pages 342-344 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Instance Methods

Let's extend our **Cupcake** class!

```
public class Cupcake {
    public boolean delicious;
    private String name;

    public Cupcake(boolean isTasty, String cupcakeName) {
        delicious = isTasty;
        name = cupcakeName;
    }

    public void setName(String n) { name = n; }

    public String getName() { return name; }
}
```

A **getter** method has been specified here. This method merely returns the attribute **value.**

A **setter** method specified. This allows us to modify the **name** attribute.

```
Cupcake mine = new Cupcake(true, "My Cupcake!");
Cupcake toShare = new Cupcake(false, "Everyone's Cupcake");
mine.setName("My Cupcake, Don't touch!");
```

Refer to Chapter 5.2, pages 342-344 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Instance Methods

Let's extend our **Cupcake** class!

```
public class Cupcake {
    public boolean delicious;
    private String name;

    public Cupcake(boolean isTasty, String cupcakeName) {
        delicious = isTasty;
        name = cupcakeName;
    }

    public void setName(String n) { name = n; }

    public String getName() { return name; }
```

private modifier limits how where the attribute can be accessed. **public** allows access outside of the class while **private** limits itself to the scope of the class.

A **getter** method has been specified here. This method merely returns the attribute **name.**

A **setter** method specified. This allows us to modify the **name** attribute

```
Cupcake mine = new Cupcake(true, "My Cupcake!");
Cupcake toShare = new Cupcake(false, "Everyone's Cupcake");
mine.setName("My Cupcake, Don't touch!");
```

Refer to Chapter 5.2, pages 356-359 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Instance Methods

Let's extend our **Cupcake** class!

```
public class Cupcake {
    public boolean delicious;
    private String name;

    public Cupcake(boolean isTasty, String cupcakeName) {
        delicious = isTasty;
        name = cupcakeName;
    }

    public void setName(String n) { name = n; }

    public String getName() { return name; }
```

private modifier limits how where the attribute can be accessed. **public** allows access outside of the class while **private** limits itself to the scope of the class.

A **getter** method has been specified here. This method merely returns the attribute **name.**

A **setter** method specified. This allows us to modify the **name** attribute

```
Cupcake mine = new Cupcake(true, "My Cupcake!");
Cupcake toShare = new Cupcake(false, "Everyone's Cupcake");
mine.setName("My Cupcake, Don't touch!");
```

If we were to use **.getName()** on **mine** and **toShare**. What would the return value be?

Instance Methods

We want to eat the cupcake

```
public class Cupcake {
    public boolean delicious;
    private String name;
    public boolean eaten;

    public Cupcake(boolean isTasty, String cupcakeName) {
        delicious = isTasty;
        name = cupcakeName;
        eaten = false;
    }

    public void setName(String n) { name = n; }

    public String getName() { return name; }

    public void eat() { eaten = true; }

}
```

Instance Methods

We want to eat the cupcake

```
public class Cupcake {
    public boolean delicious;
    private String name;
    public boolean eaten;

    public Cupcake(boolean isTasty, String cupcakeName) {
        delicious = isTasty;
        name = cupcakeName;
        eaten = false;
    }

    public void setName(String n) { name = n; }

    public String getName() { return name; }

    public void eat() { eaten = true; }

}
```

Now we have an extra property called **eaten** and we can write a method called **eat()** that will change the state of the object.

Instance Methods

```
public class Cupcake {
    public boolean delicious;
    private String name;
    public boolean eaten;

    public Cupcake(boolean isTasty, String cupcakeName) {
        delicious = isTasty;
        name = cupcakeName;
        eaten = false;
    }

    public void setName(String n) { name = n; }

    public String getName() { return name; }

    public void eat() {
        if(!eaten) {
            System.out.println("That was nice!");
        eaten = true;
    }
}
```

Expanding on this method, we can output to the user when it has been eaten.

Let’s make extend this class and test it!

UML

Unified Modelling Language, a visual language to assist with designing applications and systems.

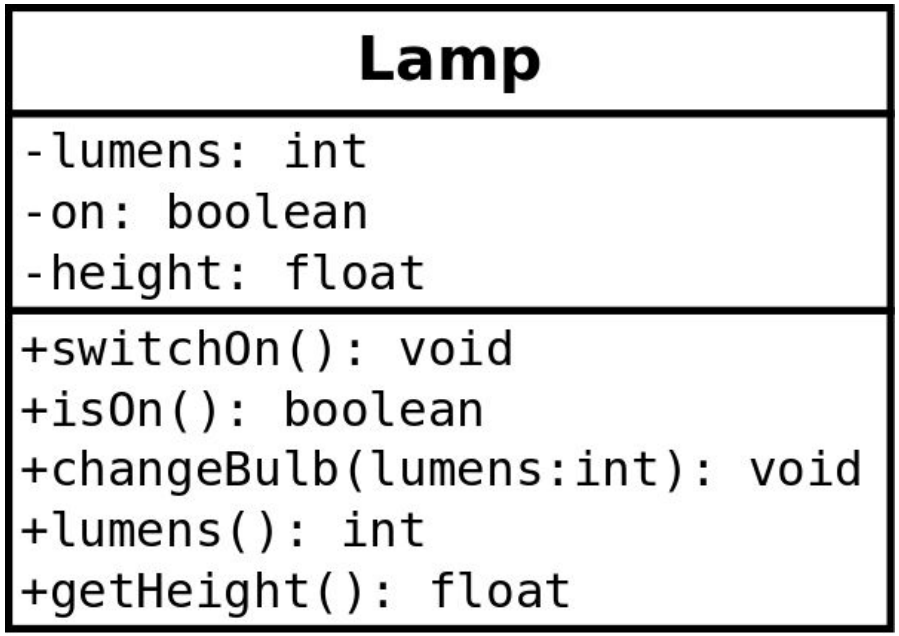
UML offers a the ability to purely design a system in how objects will interact with each other as well as describing interaction a user may have with the system.

Specifically in this course we are focused on **UML Class Diagrams**.

Class diagrams allow us to design classes prior to implementing them. Giving the ability to model the system without implementing it first.

UML Class Diagram

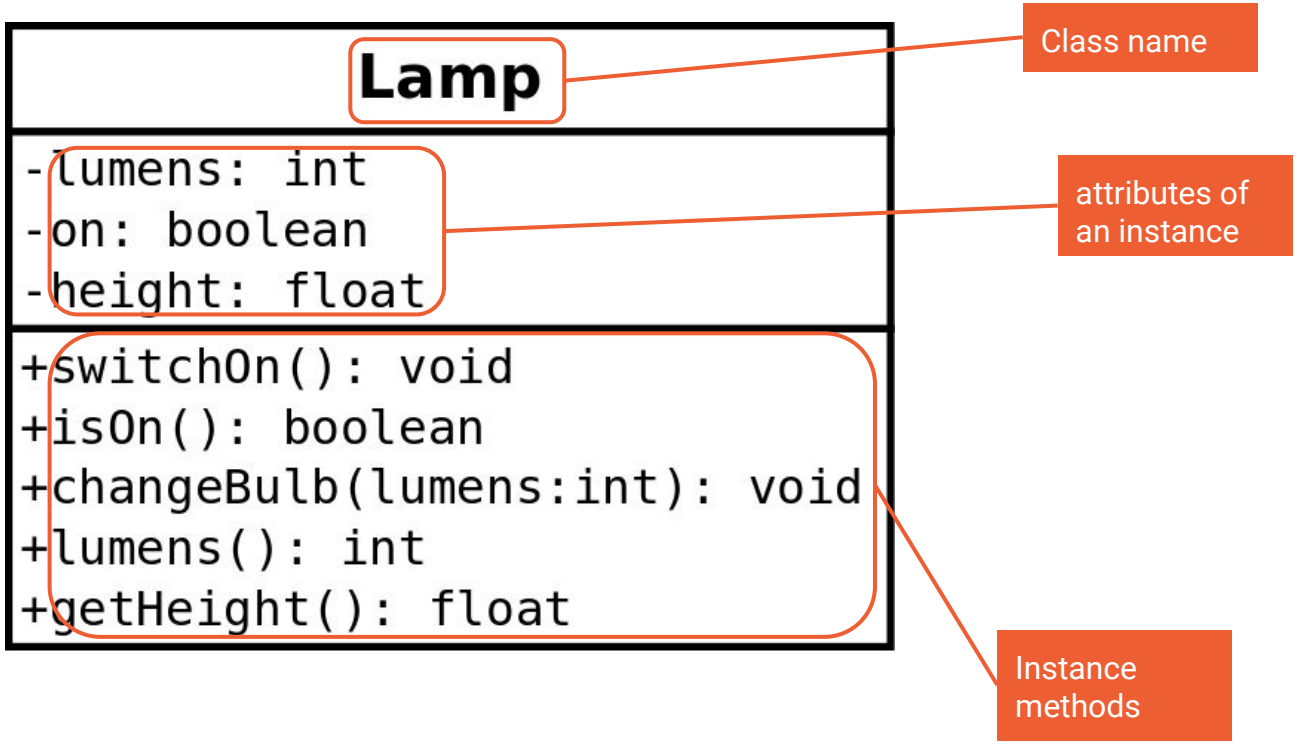
We already have a class here! Let's create it's UML Class Diagram



Refer to Chapter 5.3, page 374 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

UML Class Diagram

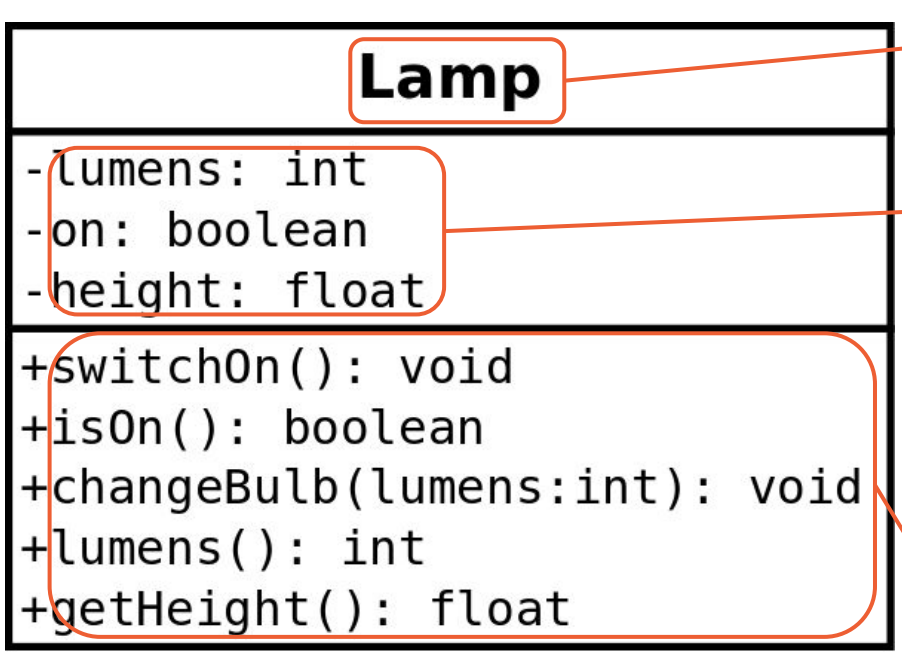
We already have a class here! Let's create it's UML Class Diagram



Refer to Chapter 5.3, page 374 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

UML Class Diagram

We already have a class here! Let's create it's UML Class Diagram

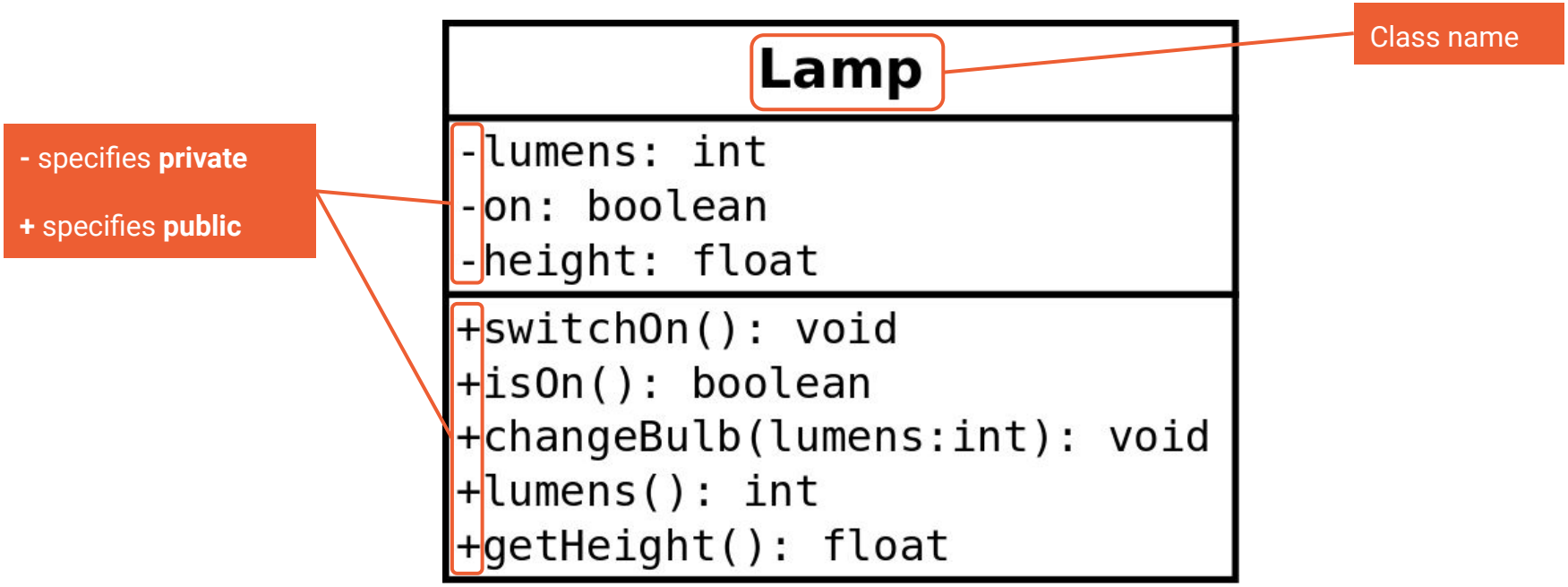


Okay but what about the + and - annotations?

Refer to Chapter 5.3, page 374 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

UML Class Diagram

We already have a class here! Let's create it's UML Class Diagram



Refer to Chapter 5.3, page 374 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

See you next time!