

INFO1113 Object-Oriented Programming

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

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act.
Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

- Classes
- Object attributes
- Instance Methods
- UML Class Diagram

Classes

“Where Reference Types Come From”

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.

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.

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**

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

```
Point p;
```

Shiny **new** keyword! As discussed before, 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?

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()
```

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()
```

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.

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.

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 */ }  
}
```

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**.

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**.

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.

Let's make some classes!

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);
```

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, String name) {  
        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);
```

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  
        delicious = isTas  
        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);
```

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);
```

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**

Let's go back and use that class!

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; }  
}
```

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**.

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.

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.

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

Instance Methods

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

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

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

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

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

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

Instance Methods

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

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

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

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

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

```
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!**

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

UML offers 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.

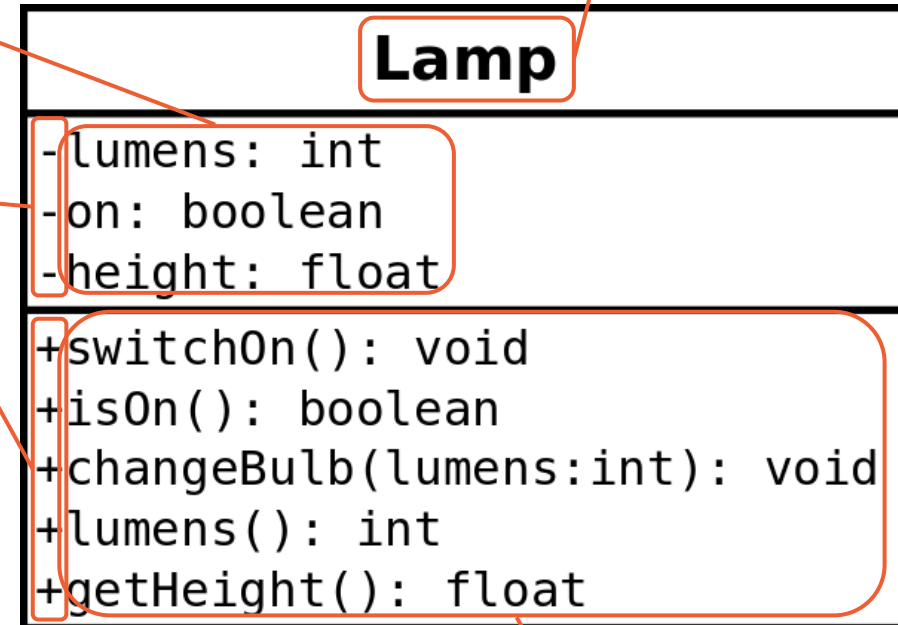
UML Class Diagram

```
public class Lamp {  
    private int lumens;  
    private boolean on;  
    private float height;  
  
    public void switch  
        .....  
}  
    public boolean isOn() {  
        .....  
}  
    public void changeBulb(int lumens) {  
        .....  
}  
    public int lumens() {  
        .....  
}  
    public float getHeight() {  
        .....  
}  
}
```

attributes of
an instance

- specifies **private**
+ specifies **public**

Class name



Instance
methods

Okay but what about the + and - annotations?

See you next time!