Scarborough College
University of Toronto
Fall 1997

CSC A06F: Introduction to Computer Programming
Lecture notes 4:
Classes and objects

Reading:

- Lewis and Loftus, chapters 4.

Methods

```
double averageSongLength (int songs, double totalTime) {
   double avg;
   avg = totalTime / songs;
   return avg;
}
```

When the method is called, its statements are executed one after the other. It can even call other methods.

Variables can be declared inside the method (e.g., `avg`), but these are **local variables** . That means they are not accessible outside the method.

The **parameters** (i.e., `tracks`, `totalTime`) are local variables within the method. Their values are **passed** in from the calling program.

It finishes execution when it reaches a `return` statement.

The `return` statement will send a value back to the calling program.

Anatomy of a method

So, a method has 4 parts:

- name
- return-type (use `void` if no value is returned)
- parameters (use empty brackets, `()`, if none)
- executable code

Syntax:

```
return-type method-name (parameter-list) {
   statement-list
}
```

Why have methods at all? Two main reasons...

How parameter passing works – I

For ***primitive data types:***

- Makes a ***copy*** of the variable's value.
- The method only sees the copy.
- That means it *can't* affect the original variable's value.

Why won't this method to swap two values do what you want?

```
void swap (int num1, int num2) {
   int temp = num1;
   num1 = num2;
   num2 = temp;
}

...

int a = 10;
int b = 20;
swap (a, b);
```

This is **pass by value.**

How parameter passing works – II

For ***objects:*** (i.e., all non-primitive types)

- Passes a **reference** to the actual object. *Not a copy of the object* .
- The method sees the *same* object as the calling program.
- Any change to the object carries back to the calling program.

This will have the expected effect. Why?

```
void swap (Num num1, Num num2) {
   int temp = num1.value;
   num1.value = num2.value;
   num2.value = temp;
}

...

Num a = new Num();  a.value = 10;
Num b = new Num();  b.value = 20;
swap (a, b);
```

This is **pass by reference.**

Classes

Example:

```
class CD {

   String title;
   String double totalTime;
   String int songs;

   CD (t, tt, s) {
      title = t;
      totalTime = tt;
      songs =  s;
   }

   double averageSongLength () {
      return ( totalTime / songs );
   }
}
```

Anatomy of a class

A class is a blueprint for building or **instantiating** new objects.

Syntax:

```
class class-name {
    data-declarations
    constructors
    methods
}
```

A class is just a description. ***You can't execute a class*** .

So, the variables declared in a class *do not* exist, and you can't call the methods either.

Instantiation of a class

Use `new` to **instantiate** a new object from a class:

```
CD my_cd;
my_cd = new CD("Trace -- Son Volt", 52.5, 13);
```

The new object is an **instance** of the class.

Instantiation

- allocates memory for the new object,
- creates the variables, and
- calls the constructor.

Each instance of the class has its own unique variables—**instance variables** .

But the methods are the same for all instances.

Initialization of an object

A **constructor** is a special method used to initialize a new instance of the class.

It has the same name as the class, but no return type.

The `new` command calls the constructor and passes the parameters to the constructor.

Constructors are mainly used to initialize the instance variables.

Only called once.

Now you're ready to use the object!

`static` methods

Every method has to be defined in a class, but you need an object to call a method.

Sometimes instantiating a new object just to call a method is *too much work.*

So, we have **class methods** , or **static methods** , which are associated with a class, and not objects.

> If you use the **modifier** `static` when you declare a method, then you can call the method directly without needing an object.

E.g., methods for mathematical functions: `Math.sqrt(2)` (`Math` is a predefined class full of formulas)

E.g., the `main` method.

A **class method** can't access any instance variables, because there is no instance.

Philosophy

Why classes?

1. Saves your time in writing big programs that have lots of similar components. (Write the code once, and use instantiation to create lots of 'copies' of the same code.)
2. Can enforce good design so that large complex systems are easier to understand, develop, debug, and maintain.
3. Because object-oriented programming is cool.

But what is 'good design'?

Good design involves **encapsulation** and **abstraction** .

Encapsulation

Think of an object as a **black box** .

> You can't see how it works. And you can't see how its state is represented.
>
> But you can request it to do things, because it provides **services** .

Some methods implement the services: **service methods** .

Others help the services, but aren't services themselves: **support methods** .

Other objects should have access to the the service methods. Why?

Other objects should ***not*** have access to the support methods. Why?

We **encapsulate** the state and behaviour of an object, and we specify which of these is accessible to others.

Visibility Modifiers

Can specify which methods are service and support.

This enforces encapsulation, which enforces good design.

- Use the `public` visibility modifier to specify service methods. I.e., any method that another object should be able to call.
- Use the `private` modifier to specify support methods. No one else can use the method.

Can also specify visibility for instance variables. As a rule: All variables that represent the state of the object should be `private`. Why?

Abstraction

You use abstraction all the time. E.g., driving a car, or using an airport.

An abstraction:

- hides details

A good abstraction

- hides the right details (i.e., the irrelevant ones)
- and reveals the important ones.

What is abstraction good for?

Encapsulation is an abstraction.