# COMP 202 - Week 5

- We've been using predefined classes. Now we will learn to write our own classes to define new objects.
- This week we focus on:
  - Objects: attributes, state and behaviour
  - Anatomy of a Class: attributes and methods
  - Classes as Types
  - Scope
  - Creating new objects
  - Parameter passing



# Objects

- An object has:
  - *state* - descriptive characteristics
  - *behaviors* - what it can do (or be done to it)
- For example, consider a coin that can be flipped so that it's face shows either "heads" or "tails"
- The state of the coin is its current face (heads or tails)
- The behavior of the coin is that it can be flipped
- Note that the behavior of the coin might change its state

## Classes

- A *class* is a blueprint of an object
- It is the model or pattern from which objects are created
- For example, the `String` class is used to define `String` objects
- Each `String` object contains specific characters (its state)
- Each `String` object can perform services (behaviors) such as `toUpperCase`

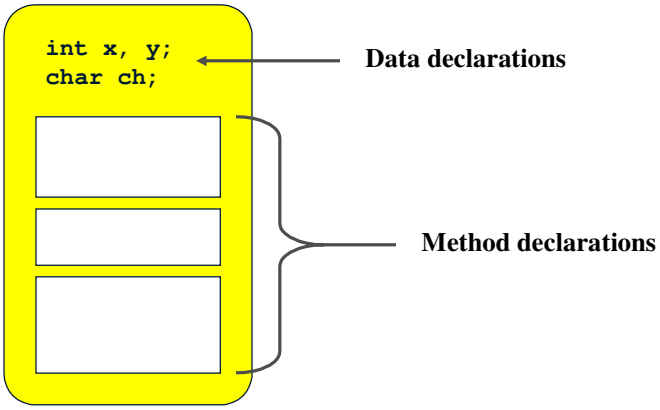


## Classes

- The `String` class was provided for us by the Java standard class library
- But we can also write our own classes that define specific objects that we need
- For example, suppose we wanted to write a program that simulates the flipping of a coin
- We could write a `Coin` class to represent a coin object

# Classes

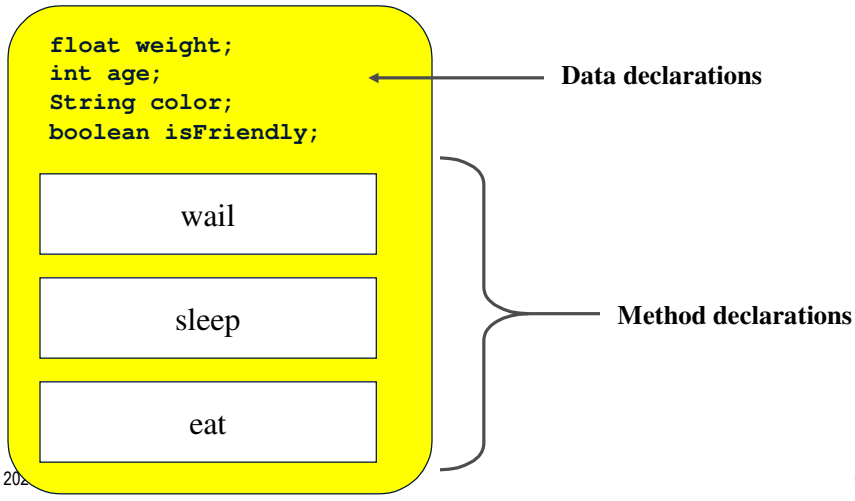
- A class contains data declarations and method declarations (collectively called *members* of the class)






# Classes

- A cat has a weight, an age, a color, and a friendliness factor
- A cat can wail, sleep, and eat



## Function

- There are many situations in life in which one number is completely dependent on another, ex:
  - The amount you pay in tuition is dependent on the number of credits you take.
  - We say that tuition is a function of the number of credits.
- When an input value generates a unique output value, the relationship is called a function.
- A function is a correspondence between a set of input values X (called the domain) and a set of output values Y (called the range) where exactly one y value in the range corresponds to each number x in the domain.
- What this is saying is that in a function, an input, x, can never correspond to more than one output, y.
  - 13 units cannot cost one amount for your friend and a different amount for you.



## Functions with multiple input variables

- In everyday life, many quantities depend on more than one changing variable, ex:
  - plant growth depends on sunlight and rainfall
  - speed depends on distance travelled and time taken
  - voltage depends on current and resistance
- A function is a rule that relates how one quantity depends on other quantities, ex:
  - $s=d/t$ where
    - $s$ = speed (m / s)
    - $d$ = distance (m)
    - $t$ = time taken (s)

## Java Methods

- **Method: A function defined in a class.**
  - **Class method: A method that is invoked without reference to a particular object (more details later).**
  - **Class methods affect the class as a whole, not a particular instance of the class. Also called a *static method*.**
  - **instance method: Any method that is invoked with respect to an instance of a class. Also called simply a *method*.**
- **Unless specified otherwise, a method is not static.**
- **The input variables of a method are called its *parameters*.**
- **The output variable of a method is called its *return value*.**
- **A method in Java does not have to return a value, in which case we declare the return type as *void* (ex: `main` method).**

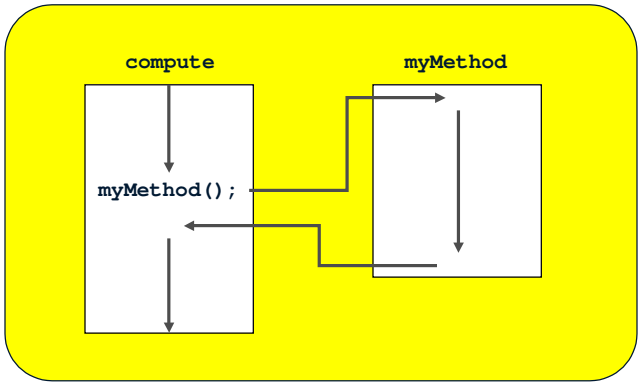


## Writing Methods

- **A *method declaration* specifies the code that will be executed when the method is invoked (or called)**
- **When a method is invoked, the flow of control jumps to the method and executes its code**
- **When complete, the flow returns to the place where the method was called and continues**
- **The invocation may or may not return a value, depending on how the method was defined**

# Method Control Flow

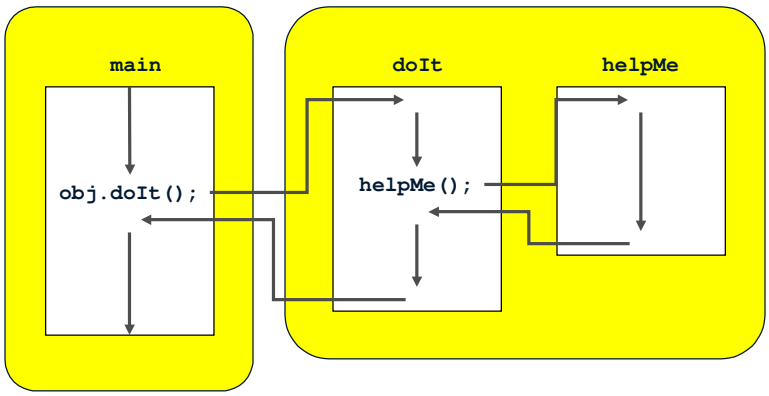
- The called method could be within the same class, in which case only the method name is needed






# Method Control Flow

- The called method could be part of another class or object



## The Coin Class

- In our `Coin` class we could define the following data:
  - `face`, an integer that represents the current face
  - `HEADS` and `TAILS`, integer constants that represent the two possible states
- We might also define the following methods:
  - a `Coin` constructor, to set up the object
  - a `flip` method, to flip the coin
  - a `getFace` method, to return the current face
  - a `toString` method, to return a string description for printing



## The Coin Class

- See CountFlips.java
- See Coin.java
- Once the `Coin` class has been defined, we can use it again in other programs as needed
- Note that the `CountFlips` program did not use the `toString` method
- A program will not necessarily use every service provided by an object
- See Cat.java
- See FeedTheCat.java

## Data Scope

- The *scope* of data is the area in a program in which that data can be used (referenced)
- Data declared at the class level can be used by all methods in that class
- Data declared within a method can only be used in that method
- Data declared within a method is called *local data*



## Scope

- A variable's scope is the region of a program within which the variable can be referred to by its simple name.
- Secondarily, scope also determines when the system creates and destroys memory for the variable.
- Scope is distinct from visibility, which applies only to member variables (and methods) and determines whether the variable can be used from outside of the class within which it is declared. Visibility is set with an access modifier (more detail later).

# Scope

- The location of the variable declaration within your program establishes its scope and places it into one of these 3 categories:
  - member variable
  - method parameter
  - local variable

```
class MyClass
{
    ...
    member variable declarations
    ...
    public void aMethod(method parameters)
    {
        ...
        local variable declarations
        ...
    }
}
```



# Scope

- A member variable is a member of a class or an object.
- It is declared within a class but outside of any method.
- A member variable's scope is the entire declaration of the class.
- You declare local variables within a block of code.
- In general, the scope of a local variable extends from its declaration to the end of the code block in which it was declared.
- Parameters are formal arguments to methods and are used to pass values into methods.
- The scope of a parameter is the entire method for which it is a parameter.

## Scope Example

- Consider the following example:

```
if (...)
{
  int i = 17;
   ...
}
System.out.println("The value of i = " + i);
```

  - The final line won't compile because the local variable `i` is out of scope.
  - Either the variable declaration needs to be moved outside of the `if` statement block, or the `println` method call needs to be moved into the `if` statement block.

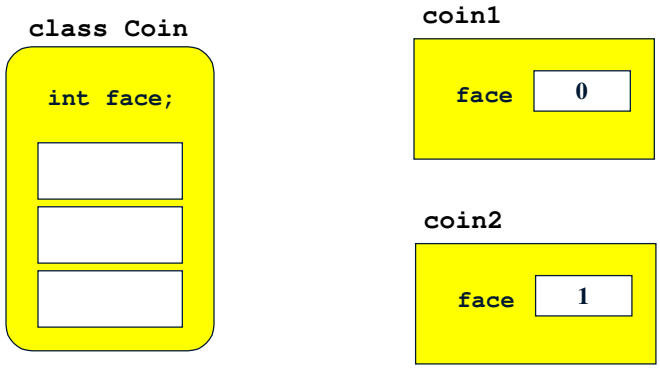


## Instance Data

- The `face` variable in the `Coin` class is called *instance data* because each instance (object) of the `Coin` class has its own
- A class declares the type of the data, but it does not reserve any memory space for it
- Every time a `Coin` object is created, a new `face` variable is created as well
- The objects of a class share the method definitions, but they have unique data space
- That's the only way two objects can have different states

# Instance Data

- See FlipRace.java






# Encapsulation

- You can take one of two views of an object:
  - internal - the structure of its data, the algorithms used by its methods
  - external - the interaction of the object with other objects in the program
- From the external view, an object is an *encapsulated* entity, providing a set of specific services
- These services define the *interface* to the object
- An object is an *abstraction*, hiding details from the rest of the system

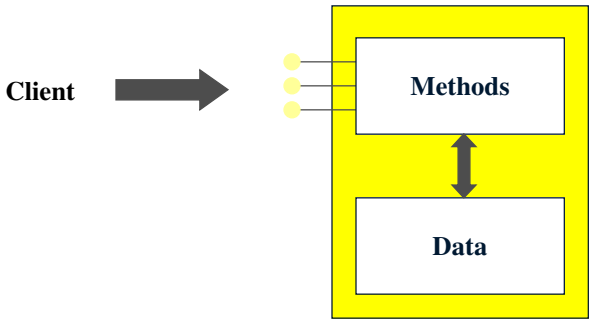
# Encapsulation

- An object should be *self-governing*
- Any changes to the object's state (its variables) should be accomplished by that object's methods
- We should make it difficult, if not impossible, for one object to "reach in" and alter another object's state
- The user, or *client*, of an object can request its services, but it should not have to be aware of how those services are accomplished



# Encapsulation

- An encapsulated object can be thought of as a *black box*
- Its inner workings are hidden to the client, which only invokes the interface methods



## Visibility Modifiers

- In Java, we accomplish encapsulation through the appropriate use of *visibility modifiers*
- A *modifier* is a Java reserved word that specifies particular characteristics of a method or data value
- We've used the modifier `final` to define a constant
- Java has three visibility modifiers: `public`, `private`, and `protected`
- We will discuss the `protected` modifier later



## Visibility Modifiers

- Members of a class that are declared with *public visibility* can be accessed from anywhere
- Members of a class that are declared with *private visibility* can only be accessed from inside the class
- Members declared without a visibility modifier have *default visibility* and can be accessed by any class in the same package
- Java modifiers are discussed in detail in Appendix F

## Visibility Modifiers

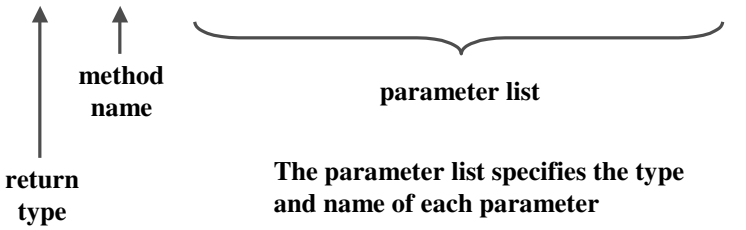
- As a general rule, no object's data should be declared with public visibility
- Methods that provide the object's services are usually declared with public visibility so that they can be invoked by clients
- Public methods are also called *service methods*
- A method created simply to assist a service method is called a *support method*
- Since a support method is not intended to be called by a client, it should not be declared with public visibility



## Method Declarations Revisited

- A method declaration begins with a *method header*

```
char calc (int num1, int num2, String message)
```




The parameter list specifies the type and name of each parameter

The name of a parameter in the method declaration is called a *formal parameter*

# Method Declarations

- The method header is followed by the *method body*

```
char calc (int num1, int num2, String message)
{
   int sum = num1 + num2;
   char result = message.charAt (sum);

   return result;
}
```

The return expression must be consistent with the return type

`sum` and `result` are *local data*

They are created each time the method is called, and are destroyed when it finishes executing



# The return Statement

- The *return type* of a method indicates the type of value that the method sends back to the calling location
- A method that does not return a value has a `void` return type
- The *return statement* specifies the value that will be returned
- Its expression must conform to the return type

# Parameters

- Each time a method is called, the *actual parameters* in the invocation are copied into the *formal parameters*

```
ch = obj.calc (25, count, "Hello");
```

```
char calc (int num1, int num2, String message)
{
   int sum = num1 + num2;
   char result = message.charAt (sum);

   return result;
}
```



# Constructors Revisited

- Recall that a constructor is a special method that is used to set up a newly created object
- When writing a constructor, remember that:
  - it has the same name as the class
  - it does not return a value
  - it has no return type, not even `void`
  - it often sets the initial values of instance variables
- The programmer does not have to define a constructor for a class
- See SuperCat.java
- See FeedTheCats.java
- See BankAccounts.java
- See Account.java