Creating Classes and objects

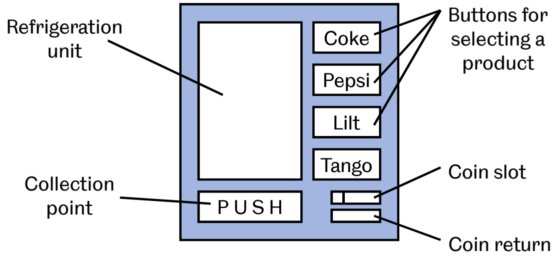
This lecture will

- Review the principles of object orientation
- Explain how to write methods in Java and the difference between an **actual parameter** and a **formal parameter**
- Explain how to write a simple class in Java
- Introduce class variables, methods and constants
- Explain **constructor chaining**
- Introduce the use of `private` methods

The rationale for object-orientation

- Many things in the real world can be described in terms of their **state**, and in terms of the **actions** they perform.
- Objects that have a state and associated actions are good models of real-world problems.
- **Example: A vending machine**
  - We can perform **actions** on the machine, e.g. selecting an item to buy, or requesting a coin refund.
  - The machine has a **state** that affects its behaviour, e.g. it may not be able to give change to a customer.
  - The inner workings of the machine are hidden; to the customer it is a 'black box'.

User interface of a vending machine



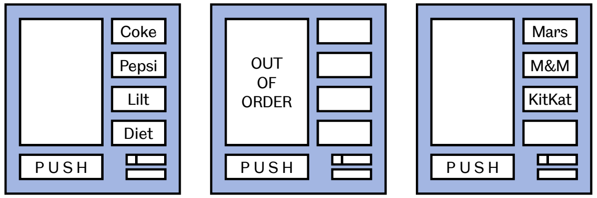
Principles of object-oriented approach

- An object provides operations that the user can invoke.
- These operations are called **methods**.
- An object has an internal **state**. Some of that state may be available to the user (either directly or via methods).
- An object is a black box. Some of its internal state is hidden from the user (**information hiding**).
- We don't reveal how methods work, or how they manipulate the internal state.

Objects and classes

- There are many types of vending machine, but all have the same core functionality.
- We say there is a **class** of vending machines.
- For this class, we can specify in general terms the state it will maintain and the actions (methods) it will provide, e.g. all vending machines will dispense a product of some kind.
- Specific vending machines are **instances** of the class of vending machines. They have the same methods, but different states.
- An **object** is an **instance** of a **class**.

Same structure, different state



Software classes and objects

- Software classes are used to package related values and the things one might want to do with them together in a sort of black box with the workings hidden
- For a **class**, we specify the information it will store and the actions (**methods**) it will provide
- A class can be used to create multiple **objects**, versions of the underlying class where each version is based on the class but has its own copies of the information
- Objects are created using the reserved word **new** and are **instances** of the class; they have the same methods, but different stored information
- Classes are useful because they can be used as the building blocks of larger and more complicated systems

Methods

- Methods are blocks of instructions which the program can **invoke**, cause to be obeyed
- They can, but need not, have **parameters** which act as input data to the method
- You have been using them all along
  - `System.out.println("Hello world");`
  - `Math.round(x);`
  - `String world = "Hello world".substring(6);`

Software objects

- Consider a class for representing meals. The **attributes** of the meal, the information we might want to store, might be the name, the price and the number of calories
- We need a Java class of the general form:

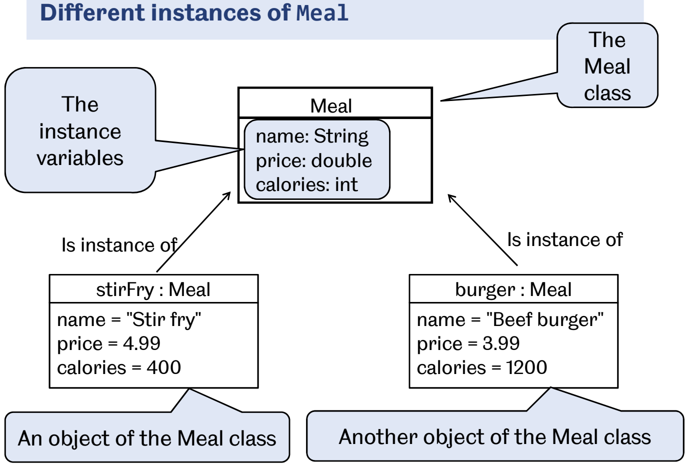
```
public class Meal {
  // instance variables
  // method declarations
}
```

State

- The **instance variables** model the attributes of a meal
- The **methods** model the operations on a meal, e.g. find its name or set its price

Actions

Different instances of `Meal`



The Meal class

The instance variables

Meal

name: String
price: double
calories: int

Is instance of

Is instance of

stirFry : Meal

name = "Stir fry"
price = 4.99
calories = 400

burger : Meal

name = "Beef burger"
price = 3.99
calories = 1200

An object of the Meal class

Another object of the Meal class

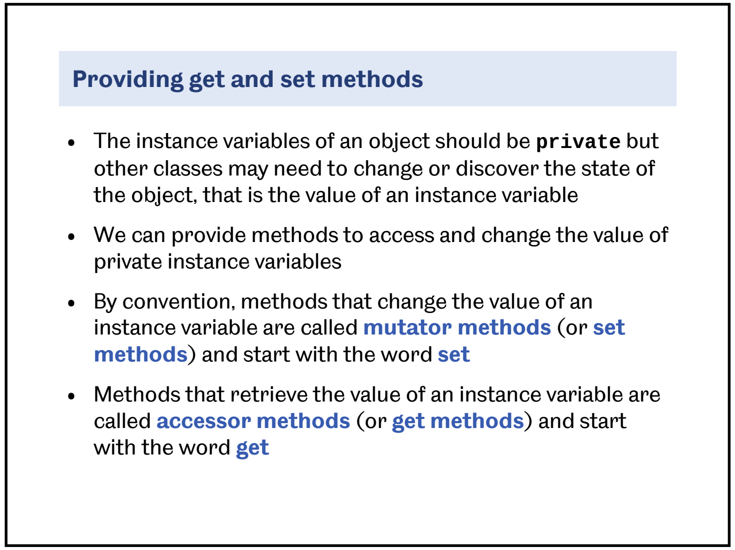
Public and private

- The class and its methods are usually declared **`public`** so that other classes can use them
- However instance variables are usually declared **`private`** so that their values are only available within the class
- This is called **information hiding** and is done to ensure that the state is always consistent
- Like the vending machine, we hide the internal workings of the object and provide an interface to the outside world (the methods)

The `Meal` class

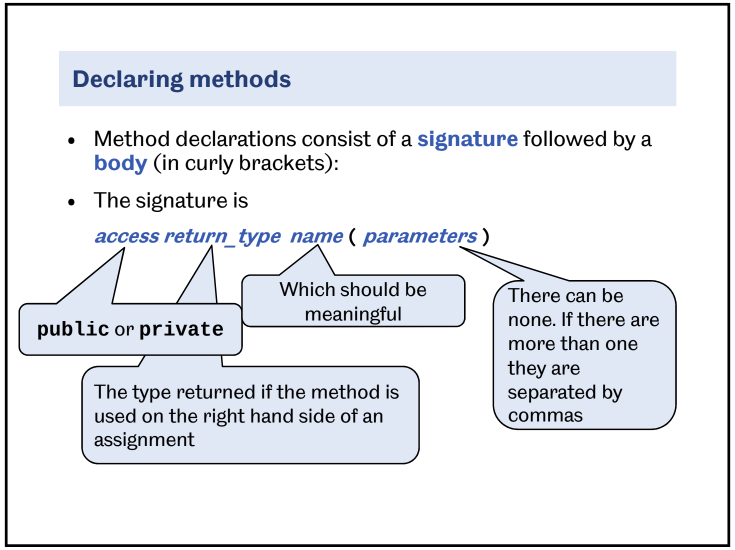
```
public class Meal {
  // instance variables
  private String name;
  private double price;
  private int calories;
  // method declarations
  ....
}
```

- Instance variables appear in the **body** of the class declaration; they are **persistent** (their values are retained so long as the object exists) and have **global scope** (their values are accessible within any method contained in the class)
- The order in which the variables and methods are declared is irrelevant to how they work but it should have some logic to it and be commented



Providing get and set methods

- The instance variables of an object should be **`private`** but other classes may need to change or discover the state of the object, that is the value of an instance variable
- We can provide methods to access and change the value of private instance variables
- By convention, methods that change the value of an instance variable are called **mutator methods** (or **set methods**) and start with the word **set**
- Methods that retrieve the value of an instance variable are called **accessor methods** (or **get methods**) and start with the word **get**



Declaring methods

- Method declarations consist of a **signature** followed by a **body** (in curly brackets):
- The signature is

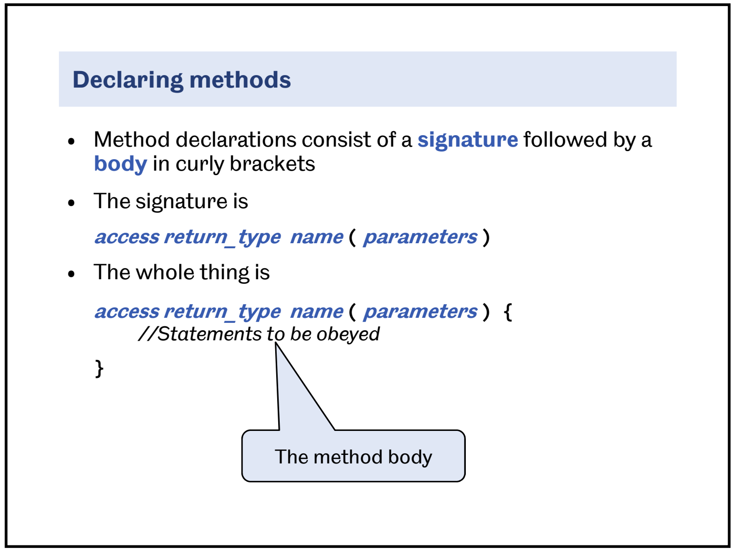
access return_type name (***parameters***)

`public` or **`private`**

The type returned if the method is used on the right hand side of an assignment

Which should be meaningful

There can be none. If there are more than one they are separated by commas



Declaring methods

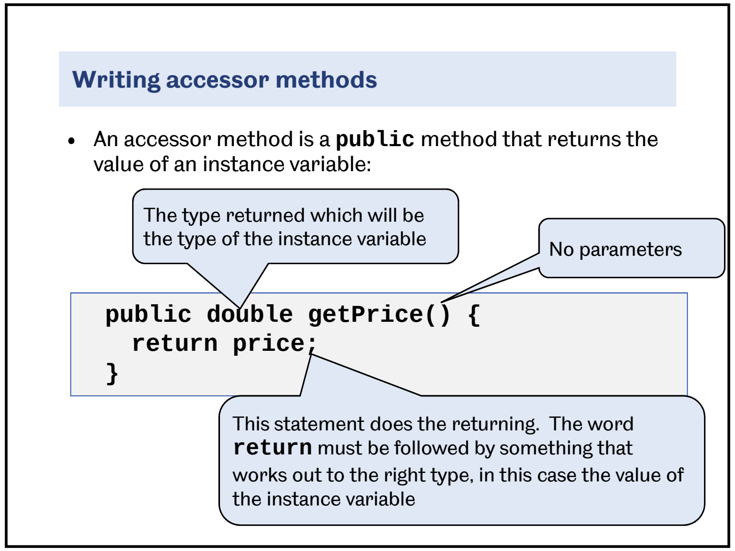
- Method declarations consist of a **signature** followed by a **body** in curly brackets
- The signature is

access return_type name (***parameters***)

- The whole thing is

```
access return_type name ( parameters ) {
    //Statements to be obeyed
}
```

The method body



Writing accessor methods

- An accessor method is a **`public`** method that returns the value of an instance variable:

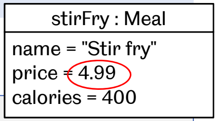
The type returned which will be the type of the instance variable

No parameters

```
public double getPrice() {
  return price;
}
```

This statement does the returning. The word **`return`** must be followed by something that works out to the right type, in this case the value of the instance variable

Writing accessor methods


stirFry : Meal

name = "Stir fry"

price = 4.99

calories = 400

```
public double getPrice() {
  return price;
}
```

- So, if **stirFry** is an instance of **Meal** then we can write:

```
 double cost = stirFry.getPrice();
```

Writing mutator methods

- This method sets the price of a **Meal**:

```
public void setPrice(double p) {
  price = p;
}
```

- Set methods are also **public**
- Set methods do not return a value. We indicate this with the return type **void**
- Methods with the return type **void** have no **return** statement
- They have a single parameter which looks like a variable declaration whose type is that of the instance variable being set

Mutator method's parameters

- This method sets the price of a **Meal**:

```
public void setPrice(double p) {
  price = p;
}
```

- Information is passed into the method by a process of **parameter passing**
- There is a **formal parameter** **p**, of type **double**. When the method is invoked, we specify an **actual parameter** whose value is assigned to the formal parameter before the body of the statement is executed

Calling a set method

- If **stirFry** is an instance of Meal, we can write:

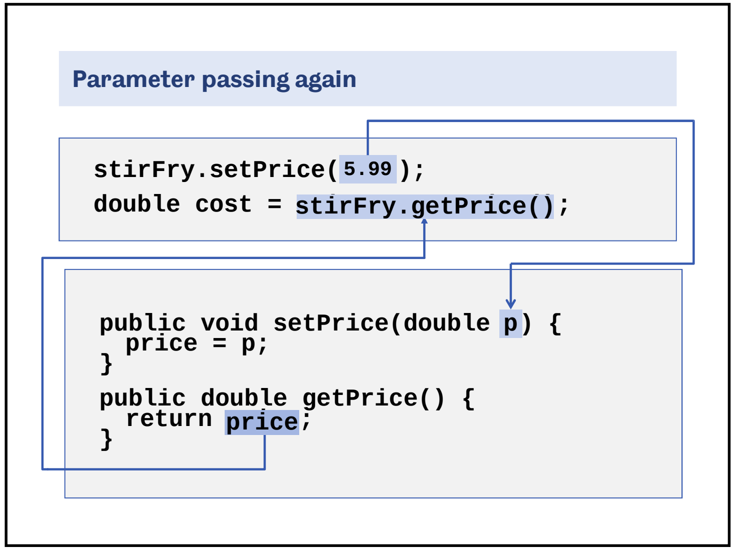
```
stirFry.setPrice(5.99);
```

- The actual parameter (**5.99**) is assigned to the formal parameter, as if we executed **p=5.99;**
- Hence, the body of the **setPrice** method during this invocation is equivalent to

```
price = 5.99;
```

- Calling a set method changes the state of the object:

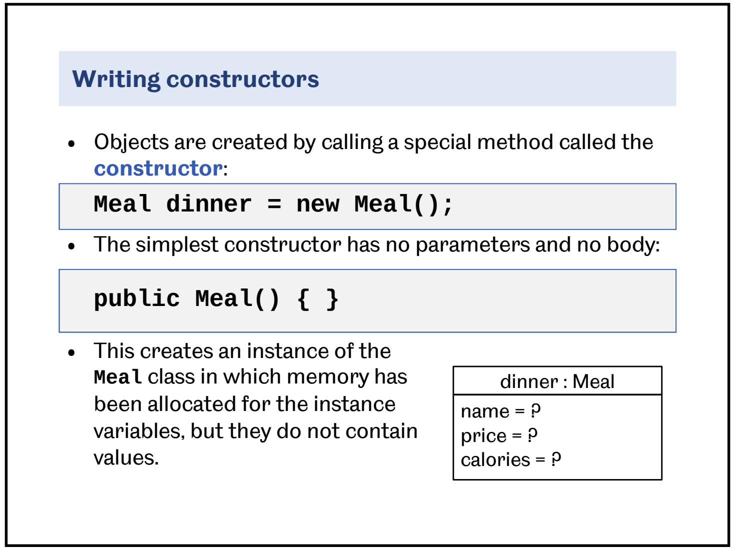
```
stirFry.setPrice(5.99);
double cost=stirFry.getPrice(); // cost is 5.99
```



Parameter passing again

```
stirFry.setPrice(5.99);
double cost = stirFry.getPrice();
```

```
public void setPrice(double p) {
  price = p;
}
public double getPrice() {
  return price;
}
```



Writing constructors

- Objects are created by calling a special method called the **constructor**:

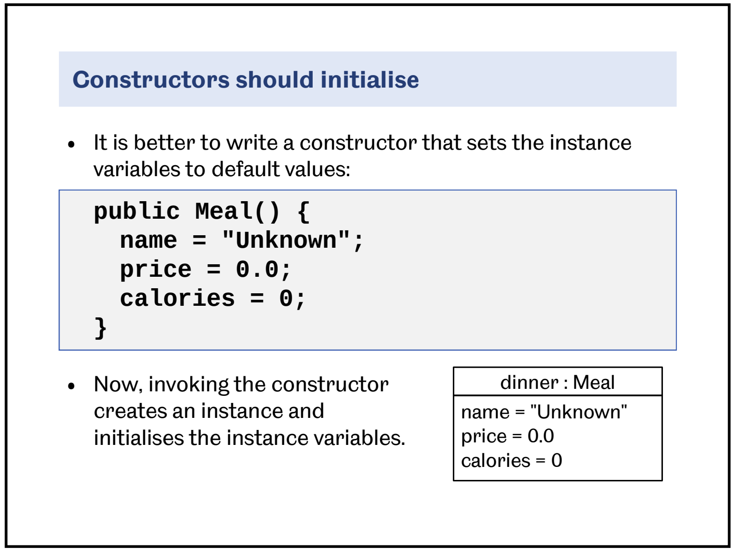
```
Meal dinner = new Meal();
```

- The simplest constructor has no parameters and no body:

```
public Meal() { }
```

- This creates an instance of the `Meal` class in which memory has been allocated for the instance variables, but they do not contain values.

dinner : Meal
name = ?
price = ?
calories = ?



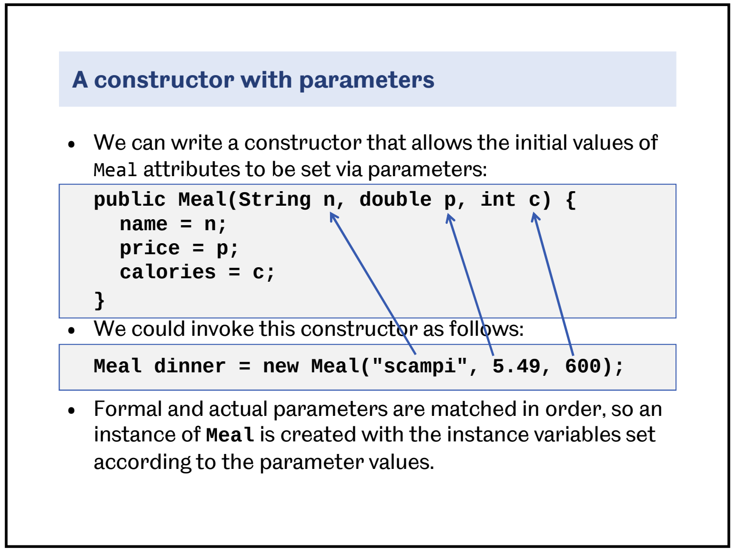
Constructors should initialise

- It is better to write a constructor that sets the instance variables to default values:

```
public Meal() {
  name = "Unknown";
  price = 0.0;
  calories = 0;
}
```

- Now, invoking the constructor creates an instance and initialises the instance variables.

dinner : Meal
name = "Unknown"
price = 0.0
calories = 0



A constructor with parameters

- We can write a constructor that allows the initial values of `Meal` attributes to be set via parameters:

```
public Meal(String n, double p, int c) {
  name = n;
  price = p;
  calories = c;
}
```

- We could invoke this constructor as follows:

```
Meal dinner = new Meal("scampi", 5.49, 600);
```

- Formal and actual parameters are matched in order, so an instance of `Meal` is created with the instance variables set according to the parameter values.

Overloading the constructor

- We now have **two** constructors for the `Meal` class:

```
public Meal() { ... }
public Meal(String n, double p, int c) { ... }
```

- The compiler knows which constructor to call by matching the actual parameters against the formal parameters.
- We can have several methods with the same name but different formal parameters.
- This is called **overloading**.

❓ **Where else have you seen overloading?**

The `toString` method

- It would be convenient if we could display the attributes of a `Meal` object by invoking a single method.
- The solution is to provide a `toString` method:

```
public String toString() {
  return "Meal Name=" + name +
             ", Price=" + price +
             " and Calories=" + calories;
}
```

- Java automatically invokes the `toString` method in any expression that requires a `String` argument.

Invoking the `toString` method

- Consider the following program fragment:

```
Meal pizza =
    new Meal("Special Pizza", 8.99, 800);
System.out.println(pizza);
```

- The output from this code is:

```
Meal Name=Special Pizza, Price=8.99 and Calories=800
```

- It is as though we had written the following:

```
System.out.println(pizza.toString());
```

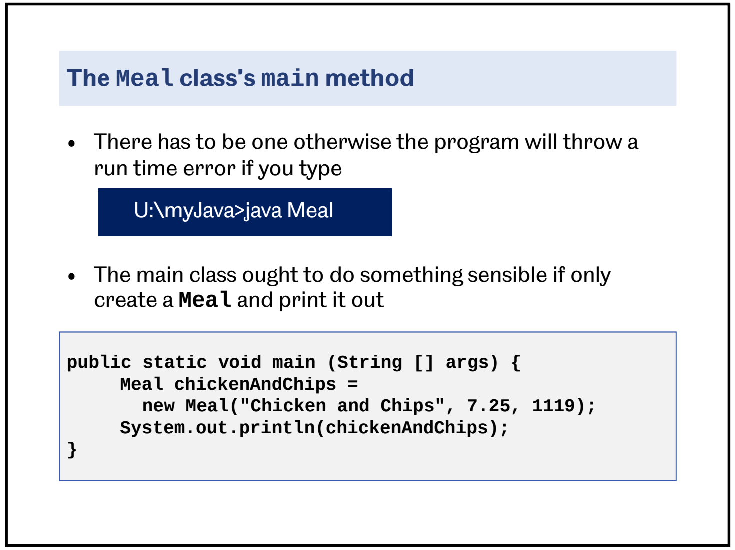
The `main` Method

- When we started this course we wrote programs like this

```
public class Simple {
   public static void main(String[] args) {
      System.out.print("Running a Java application");
      System.out.println("...finished.");
   }
}
```

A class which contained nothing except a `main` method which did everything

- Any program we write will still need a `main` method as an entry point and won't work without it



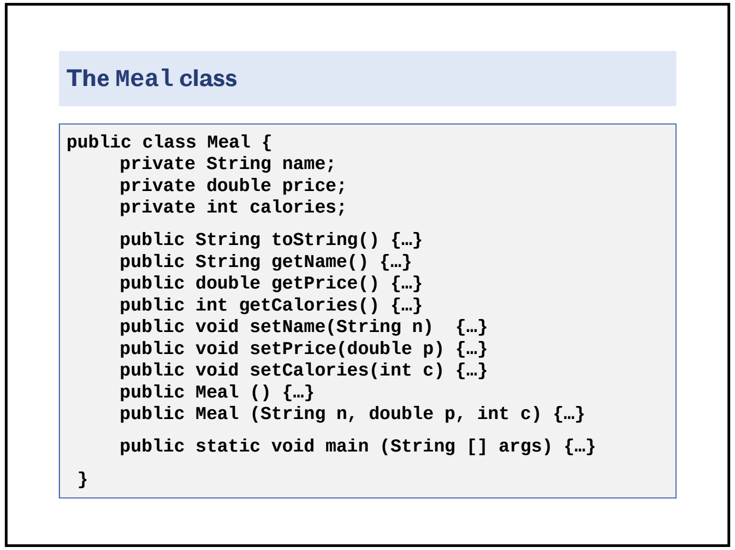
The `Meal` class’s `main` method

- There has to be one otherwise the program will throw a run time error if you type

```
U:\myJava>java Meal
```

- The main class ought to do something sensible if only create a `Meal` and print it out

```
public static void main (String [] args) {
     Meal chickenAndChips =
       new Meal("Chicken and Chips", 7.25, 1119);
     System.out.println(chickenAndChips);
}
```

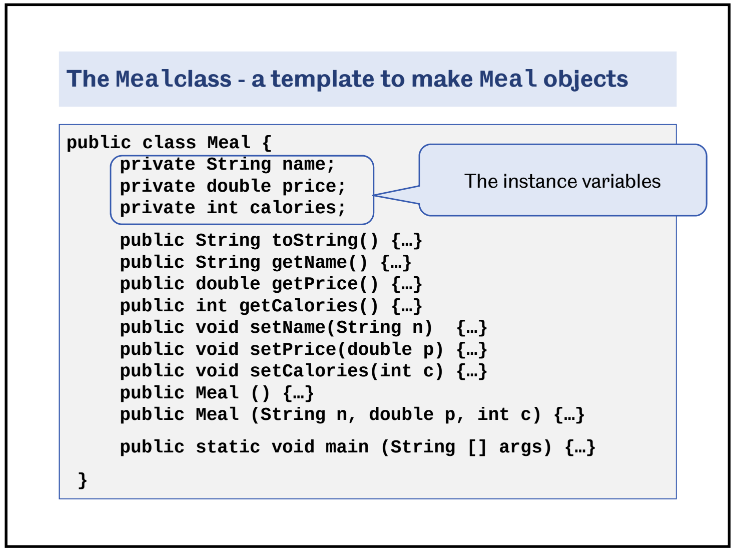


The `Meal` class

```
public class Meal {
     private String name;
     private double price;
     private int calories;

     public String toString() {…}
     public String getName() {…}
     public double getPrice() {…}
     public int getCalories() {…}
     public void setName(String n)  {…}
     public void setPrice(double p) {…}
     public void setCalories(int c) {…}
     public Meal () {…}
     public Meal (String n, double p, int c) {…}

     public static void main (String [] args) {…}
 }
```

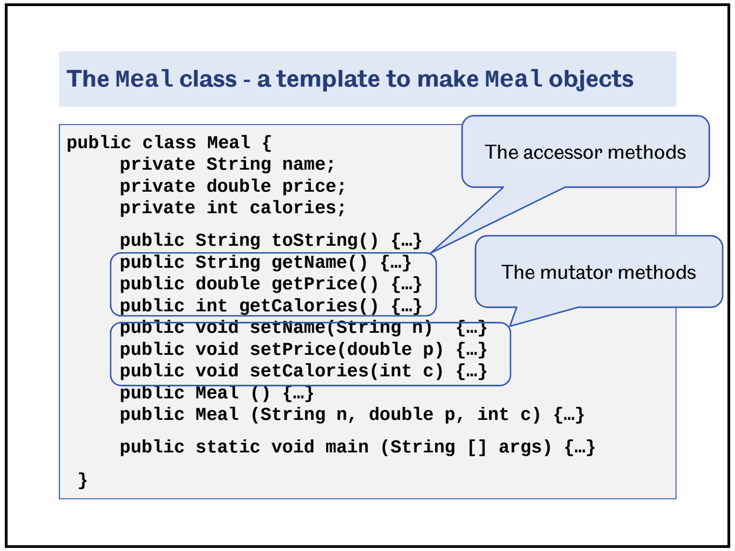


The `Meal`class - a template to make `Meal` objects

```
public class Meal {
     private String name;
     private double price;
     private int calories;

     public String toString() {…}
     public String getName() {…}
     public double getPrice() {…}
     public int getCalories() {…}
     public void setName(String n)  {…}
     public void setPrice(double p) {…}
     public void setCalories(int c) {…}
     public Meal () {…}
     public Meal (String n, double p, int c) {…}

     public static void main (String [] args) {…}
 }
```

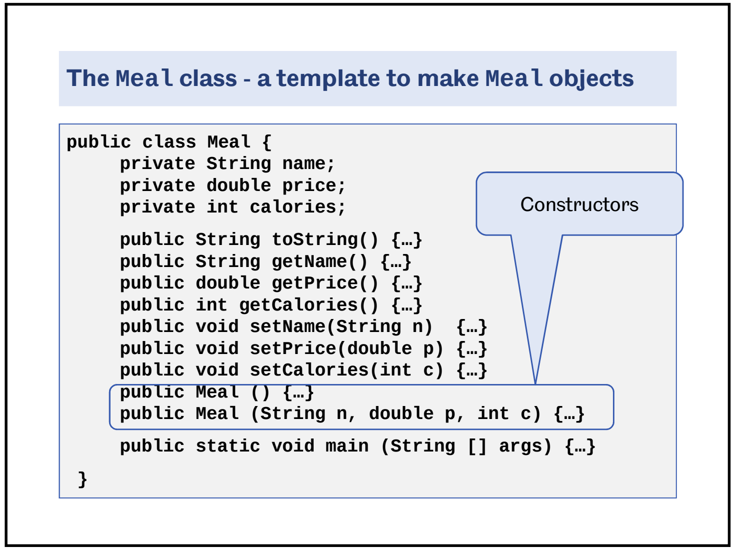


The `Meal` class - a template to make `Meal` objects

```
public class Meal {
     private String name;
     private double price;
     private int calories;

     public String toString() {…}
     public String getName() {…}
     public double getPrice() {…}
     public int getCalories() {…}
     public void setName(String n)  {…}
     public void setPrice(double p) {…}
     public void setCalories(int c) {…}
     public Meal () {…}
     public Meal (String n, double p, int c) {…}

     public static void main (String [] args) {…}
 }
```



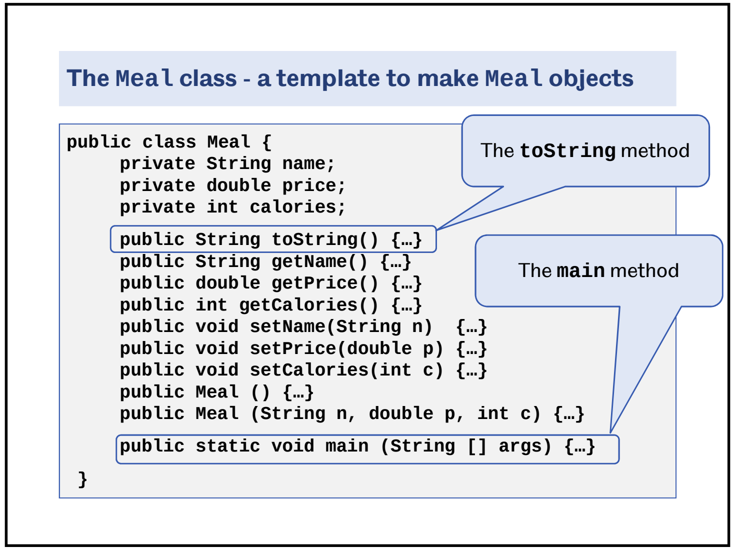
The `Meal` class - a template to make `Meal` objects

```
public class Meal {
    private String name;
    private double price;
    private int calories;

    public String toString() {…}
    public String getName() {…}
    public double getPrice() {…}
    public int getCalories() {…}
    public void setName(String n)  {…}
    public void setPrice(double p) {…}
    public void setCalories(int c) {…}
    public Meal () {…}
    public Meal (String n, double p, int c) {…}

    public static void main (String [] args) {…}
 }
```

Constructors



The `Meal` class - a template to make `Meal` objects

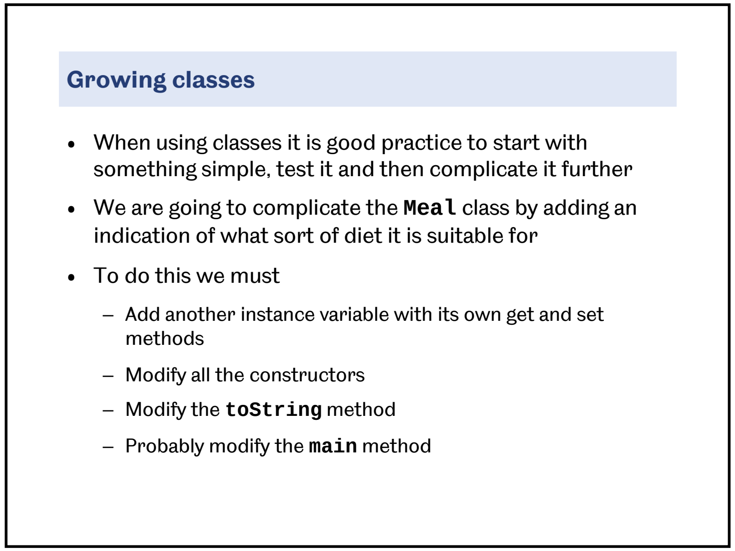
```
public class Meal {
    private String name;
    private double price;
    private int calories;

    public String toString() {…}
    public String getName() {…}
    public double getPrice() {…}
    public int getCalories() {…}
    public void setName(String n)  {…}
    public void setPrice(double p) {…}
    public void setCalories(int c) {…}
    public Meal () {…}
    public Meal (String n, double p, int c) {…}

    public static void main (String [] args) {…}
 }
```

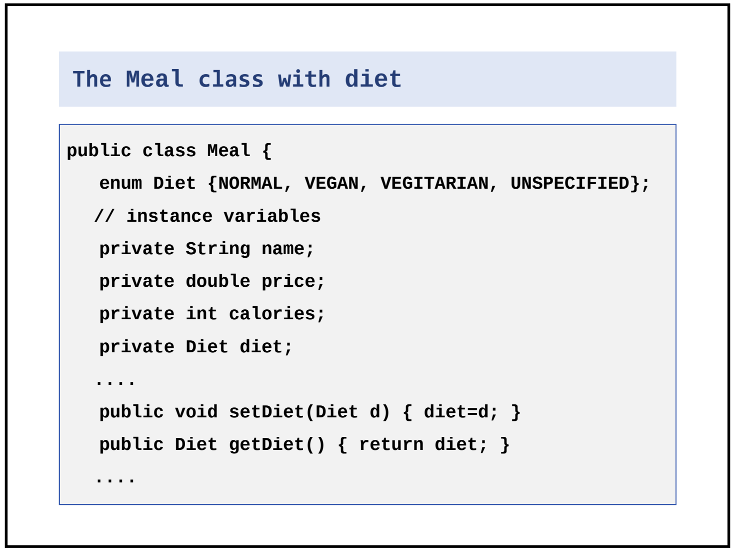
The **`toString`** method

The **`main`** method



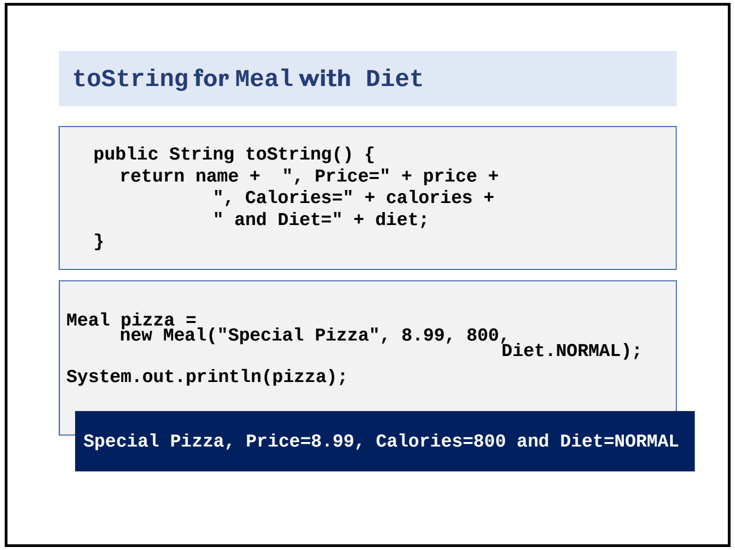
Growing classes

- When using classes it is good practice to start with something simple, test it and then complicate it further
- We are going to complicate the **`Meal`** class by adding an indication of what sort of diet it is suitable for
- To do this we must
  - Add another instance variable with its own get and set methods
  - Modify all the constructors
  - Modify the **`toString`** method
  - Probably modify the **`main`** method



The Meal class with diet

```
public class Meal {
    enum Diet {NORMAL, VEGAN, VEGITARIAN, UNSPECIFIED};
   // instance variables
    private String name;
    private double price;
    private int calories;
    private Diet diet;
   ....
    public void setDiet(Diet d) { diet=d; }
    public Diet getDiet() { return diet; }
   ....
```

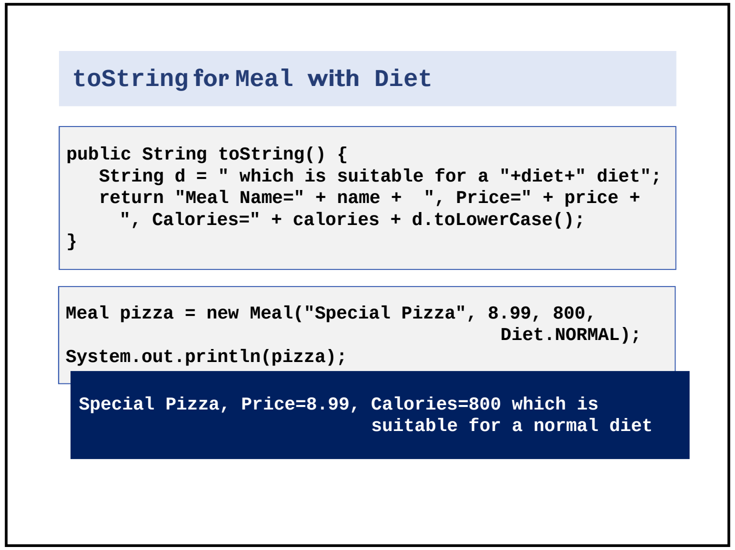


toString for Meal with Diet

```
   public String toString() {
     return name +  ", Price=" + price +
             ", Calories=" + calories +
             " and Diet=" + diet;
   }
```

```
Meal pizza =
     new Meal("Special Pizza", 8.99, 800,
                                    Diet.NORMAL);
System.out.println(pizza);
```

```
Special Pizza, Price=8.99, Calories=800 and Diet=NORMAL
```

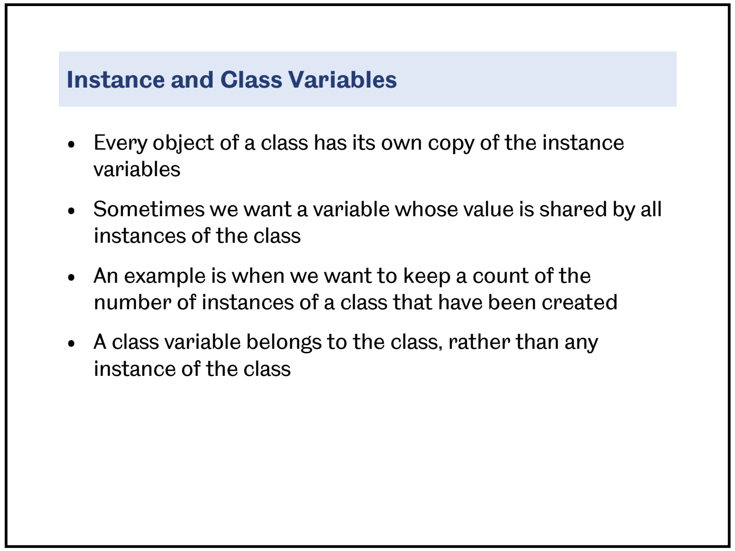


toString for Meal with Diet

```
public String toString() {
   String d = " which is suitable for a "+diet+" diet";
   return "Meal Name=" + name +  ", Price=" + price +
     ", Calories=" + calories + d.toLowerCase();
}
```

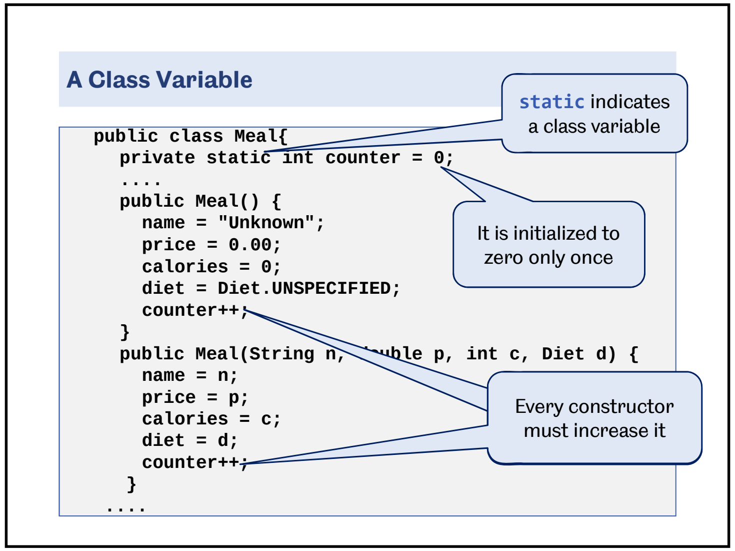
```
Meal pizza = new Meal("Special Pizza", 8.99, 800,
                                        Diet.NORMAL);
System.out.println(pizza);
```

```
Special Pizza, Price=8.99, Calories=800 which is
                           suitable for a normal diet
```



Instance and Class Variables

- Every object of a class has its own copy of the instance variables
- Sometimes we want a variable whose value is shared by all instances of the class
- An example is when we want to keep a count of the number of instances of a class that have been created
- A class variable belongs to the class, rather than any instance of the class



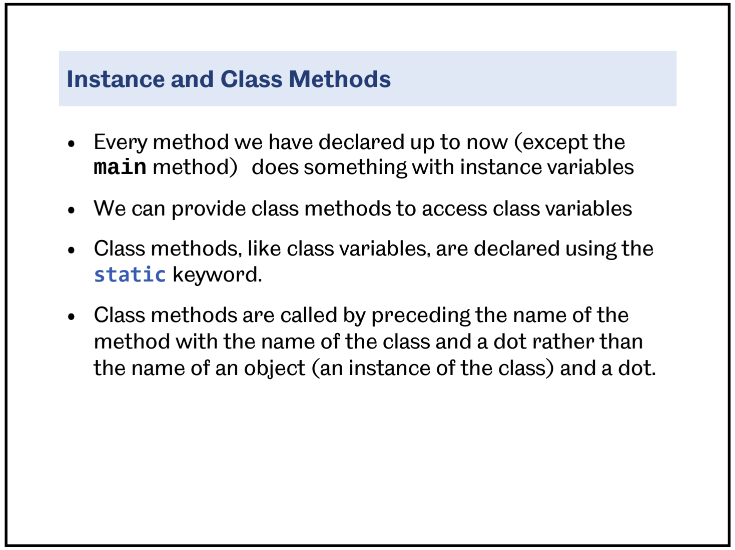
A Class Variable

```
  public class Meal{
    private static int counter = 0;
    ....
    public Meal() {
      name = "Unknown";
      price = 0.00;
      calories = 0;
      diet = Diet.UNSPECIFIED;
      counter++;
    }
    public Meal(String n, double p, int c, Diet d) {
      name = n;
      price = p;
      calories = c;
      diet = d;
      counter++;
     }
   ....
```

static indicates a class variable

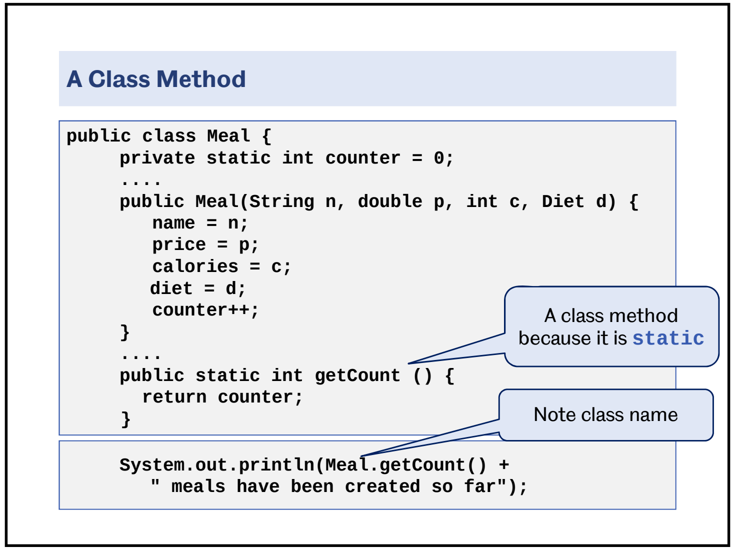
It is initialized to zero only once

Every constructor must increase it



Instance and Class Methods

- Every method we have declared up to now (except the **`main`** method) does something with instance variables
- We can provide class methods to access class variables
- Class methods, like class variables, are declared using the `static` keyword.
- Class methods are called by preceding the name of the method with the name of the class and a dot rather than the name of an object (an instance of the class) and a dot.



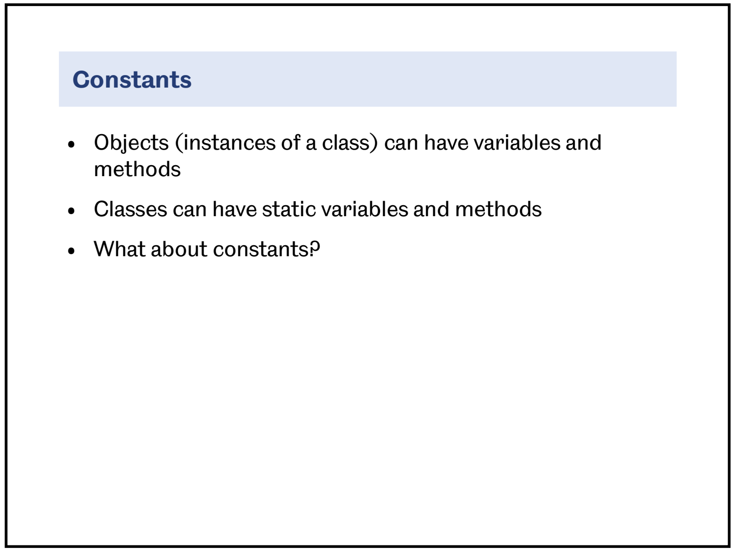
A Class Method

```
public class Meal {
     private static int counter = 0;
     ....
     public Meal(String n, double p, int c, Diet d) {
        name = n;
        price = p;
        calories = c;
        diet = d;
        counter++;
     }
     ....
     public static int getCount () {
       return counter;
     }
```

```
     System.out.println(Meal.getCount() +
        " meals have been created so far");
```

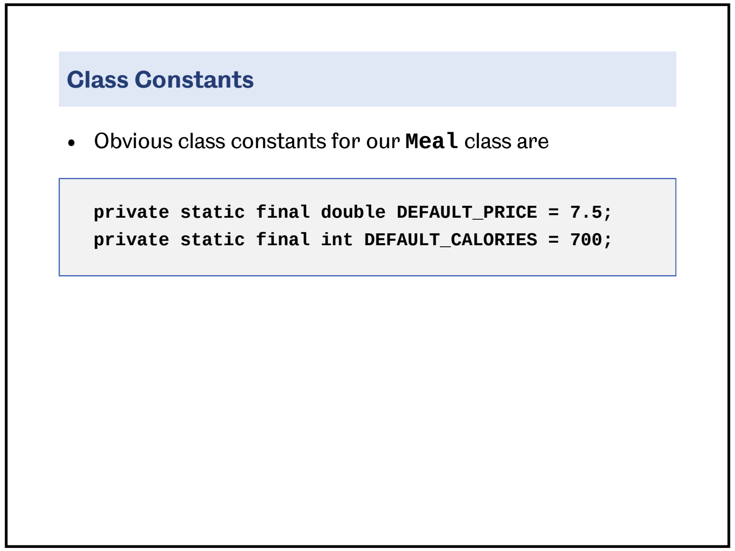
A class method because it is `static`

Note class name



Constants

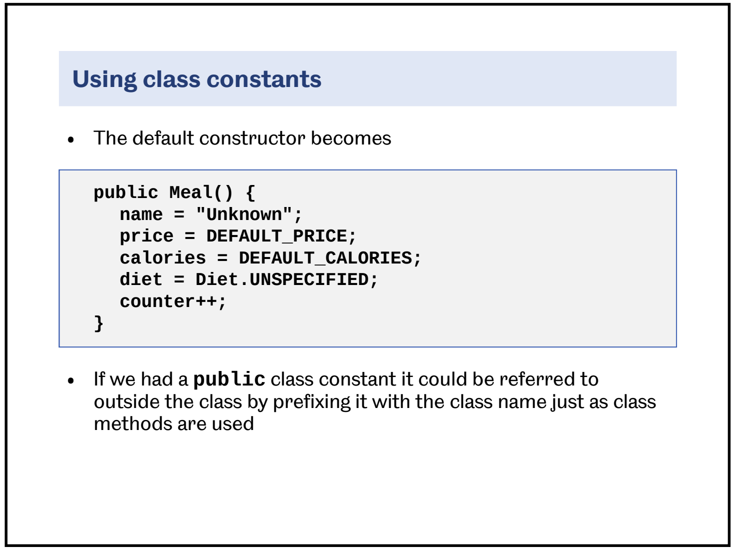
- Objects (instances of a class) can have variables and methods
- Classes can have static variables and methods
- What about constants?



Class Constants

- Obvious class constants for our **`Meal`** class are

```
private static final double DEFAULT_PRICE = 7.5;
private static final int DEFAULT_CALORIES = 700;
```

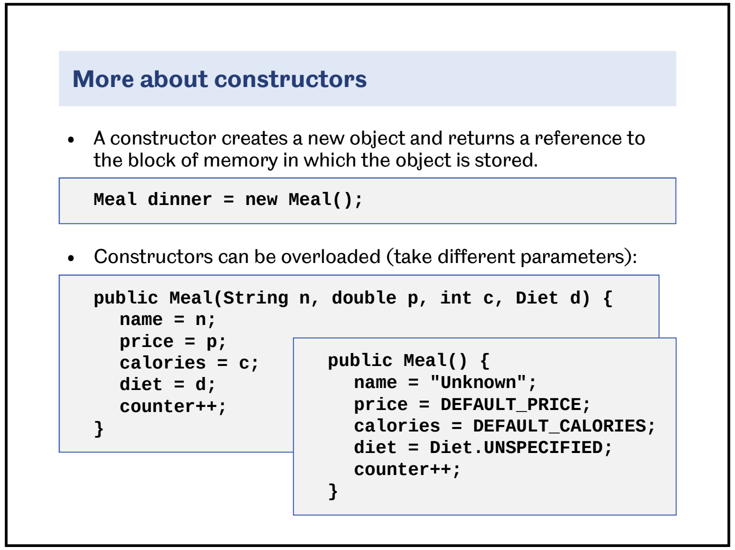


Using class constants

- The default constructor becomes

```
public Meal() {
   name = "Unknown";
   price = DEFAULT_PRICE;
   calories = DEFAULT_CALORIES;
   diet = Diet.UNSPECIFIED;
   counter++;
}
```

- If we had a **`public`** class constant it could be referred to outside the class by prefixing it with the class name just as class methods are used



More about constructors

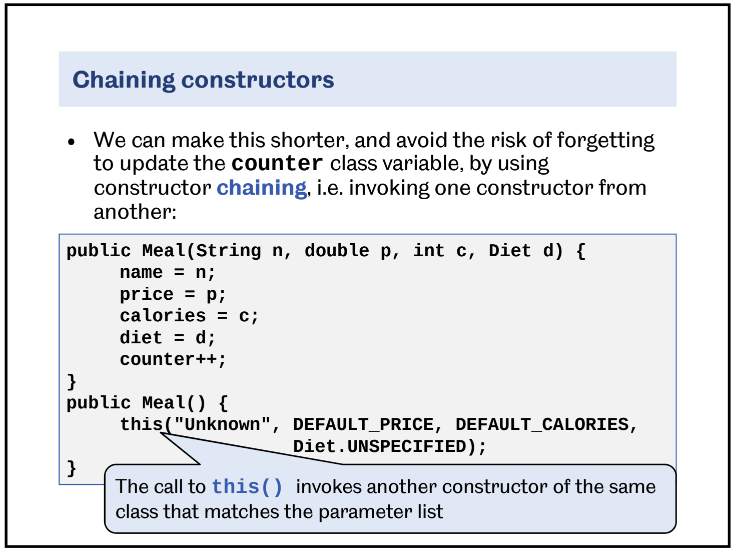
- A constructor creates a new object and returns a reference to the block of memory in which the object is stored.

```
Meal dinner = new Meal();
```

- Constructors can be overloaded (take different parameters):

```
public Meal(String n, double p, int c, Diet d) {
   name = n;
   price = p;
   calories = c;
   diet = d;
   counter++;
}
```

```
public Meal() {
   name = "Unknown";
   price = DEFAULT_PRICE;
   calories = DEFAULT_CALORIES;
   diet = Diet.UNSPECIFIED;
   counter++;
}
```

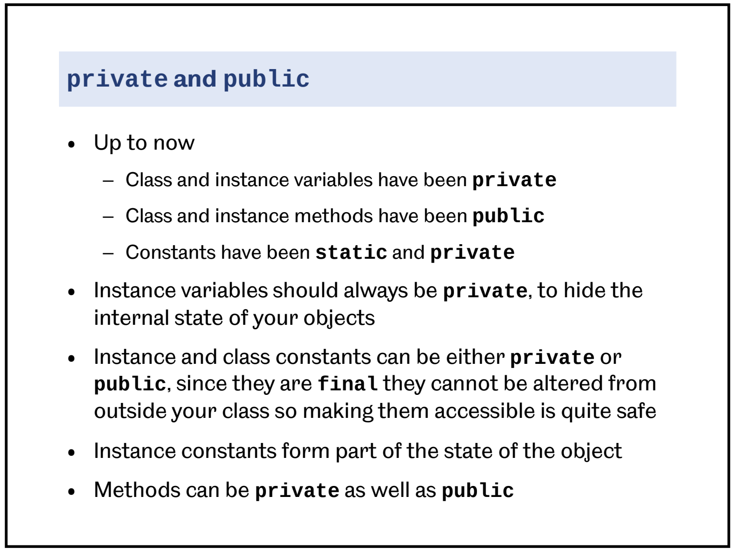


Chaining constructors

- We can make this shorter, and avoid the risk of forgetting to update the **`counter`** class variable, by using constructor **chaining**, i.e. invoking one constructor from another:

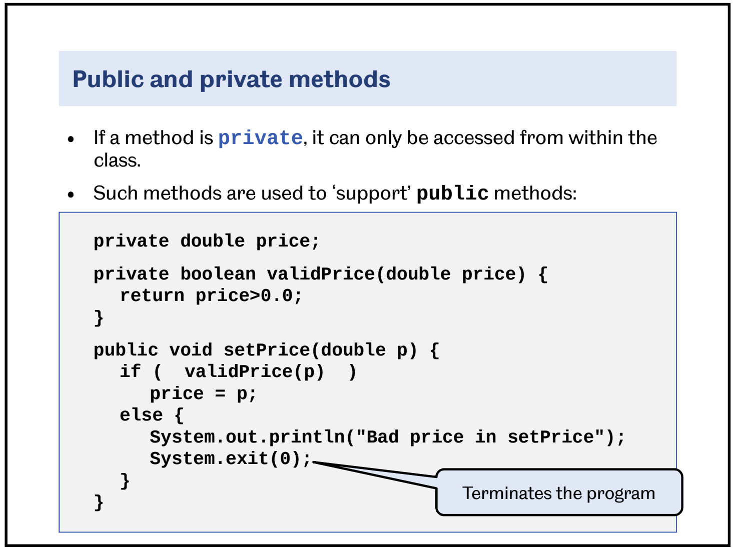
```
public Meal(String n, double p, int c, Diet d) {
     name = n;
     price = p;
     calories = c;
     diet = d;
     counter++;
}
public Meal() {
     this("Unknown", DEFAULT_PRICE, DEFAULT_CALORIES,
                     Diet.UNSPECIFIED);
}
```

The call to **`this()`** invokes another constructor of the same class that matches the parameter list



`private` and `public`

- Up to now
  - Class and instance variables have been **`private`**
  - Class and instance methods have been **`public`**
  - Constants have been **`static`** and **`private`**
- Instance variables should always be **`private`**, to hide the internal state of your objects
- Instance and class constants can be either **`private`** or **`public`**, since they are **`final`** they cannot be altered from outside your class so making them accessible is quite safe
- Instance constants form part of the state of the object
- Methods can be **`private`** as well as **`public`**



Public and private methods

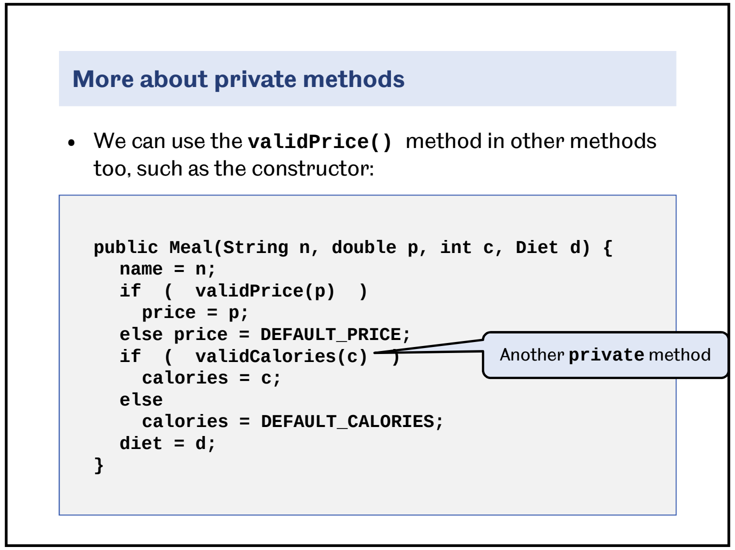
- If a method is **private**, it can only be accessed from within the class.
- Such methods are used to 'support' **public** methods:

```
private double price;

private boolean validPrice(double price) {
   return price>0.0;
}

public void setPrice(double p) {
   if (  validPrice(p)  )
      price = p;
   else {
      System.out.println("Bad price in setPrice");
      System.exit(0);
   }
}
```

Terminates the program

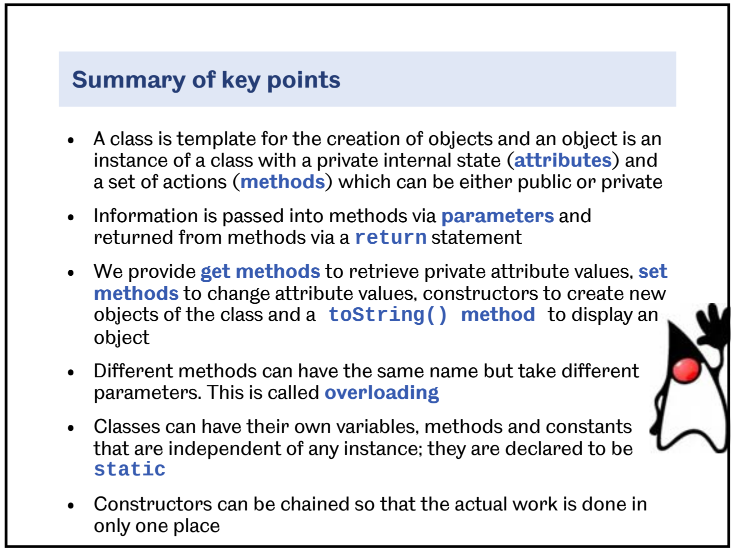


More about private methods

- We can use the **validPrice()** method in other methods too, such as the constructor:

```
public Meal(String n, double p, int c, Diet d) {
   name = n;
   if  (  validPrice(p)  )
     price = p;
   else price = DEFAULT_PRICE;
   if  (  validCalories(c)  )
     calories = c;
   else
     calories = DEFAULT_CALORIES;
   diet = d;
}
```

Another **private** method



Summary of key points

- A class is template for the creation of objects and an object is an instance of a class with a private internal state (**attributes**) and a set of actions (**methods**) which can be either public or private
- Information is passed into methods via **parameters** and returned from methods via a **return** statement
- We provide **get methods** to retrieve private attribute values, **set methods** to change attribute values, constructors to create new objects of the class and a **toString() method** to display an object
- Different methods can have the same name but take different parameters. This is called **overloading**
- Classes can have their own variables, methods and constants that are independent of any instance; they are declared to be **static**
- Constructors can be chained so that the actual work is done in only one place