public void setName(String name)
{
    this.name = name;
}

public void setAge(int n)
{
    age = n;
}

public void incrementAge()
{
    age = age + 1;
}

public String getName()
{
    return name;
}

public int getAge()
{
    return age;
}

// file Person.java
public class Person
{
    // constructor methods
    public Person()
    {
        name = "UNKNOWN";
        age = 0;
    }

    public Person(String n, int a)
    {
        name = n;
        age = a;
    }

    // instance methods
    public void setNameAndAge(String n, int a)
    {
        name = n;
        age = a;
    }

    // instance variables/attributes
    private String name;
    private int age;
}

// end class Person

Defining a Class — A Very Simple E.g.

public class Person
{
    // constructor methods
    public Person()
    {
        name = "UNKNOWN";
        age = 0;
    }

    public Person(String n, int a)
    {
        name = n;
        age = a;
    }

    // instance methods
    public void setNameAndAge(String n, int a)
    {
        name = n;
        age = a;
    }

    // instance variables/attributes
    private String name;
    private int age;
}

// end class Person

Recommended Readings:
Horstmann Ch. 2-4 & 6-9, especially 2, 3 & 9
Defining Methods
When you define a method, you take the steps required to solve a subproblem and give them a name. Afterwards, the method can be called without knowing how it is implemented. This is called procedural abstraction.

A method definition specifies:
- the name of the method,
- the name and types of parameters it takes,
- the type of result it returns,
- its visibility, i.e. public, private, etc.
- whether it is an instance or class (static) method,
- the steps required to execute it — the body of the method.

Elements of a Class Definition
A class definition may include the following:
- instance methods,
- constructors,
- instance variables/attributes/fields,
- class/static methods,
- class/static variables/constants.

Instance Variables/Attributes
For each piece of information that needs to be maintained about the instances of a class, you need to define an instance variable/attribute/field in the class, e.g.

```java
public class Person {
    // methods
    ... // instance variables/attributes
    private String name;
    private int age;
} // end class Person
```

If an attribute is public, users can access and change its value directly. But if the attribute is private, users can only access it indirectly by calling methods. E.g.
public class Car
{
  ...
  private String model;
  public int mileage;
}

These access restrictions only apply to code outside the class. The methods of the class always have access to the class’s attributes, e.g. `setAge` can change the value of the `age` attribute in a `Person` even though it is `private`.

Instance attributes are usually `private`. In this way, the class designer retains the right to change the way the data in the attributes is represented.

Parameters

When we call a method, we often want to pass some data to it; the method can then use the data, save it in an attribute, or examine it to decide what actions to take. We do this by having the method take parameters. E.g. we need to pass the person’s age to the `setAge` method; the age can be any value we want; the method uses the parameter `n` for this.

Parameters are declared in the header of the method definition. Both the parameter `name` and `type` are given, e.g.

```java
public void setAge(int n)
{
  age = n;
}
```

```java
public void setNameAndAge(String n, int a)
{
...
```

Returning Results from Methods

A method’s header specifies whether or not it returns a `result`, and if it does, what the result’s type is. When no result is returned, the method’s result type is declared to be `void`.

After a method has been called and its body has finished executing, the execution of the program continues from the point where the method was called. If a method is to return a value to the place where it was called, it must terminate by executing the statement “return expression”; e.g. `return age;`.

Then, the expression is evaluated and its value is passed back to the point of the call as the method terminates.

Methods that return a value are often called `functions` and methods that do not are often called `procedures`.

When you call a method, you supply an `argument` or `actual parameter` for each `formal parameter` in the method definition header. Arguments are associated to parameters by the order in which they appear. The number and type of arguments must match that of the parameters. E.g.

```java
Person p1 = new Person();
int uAge = 44;
p1.setAge(uAge);
p1.setNameAndAge("Yves", 44);
```

When a method is called, first the `parameters are passed`, and then the body of the method is executed.
Constructors

Constructors have the same name as their class. Their job is adequately initializing the new object's attributes.

A class can have several constructor methods. This is an example of "overloading", i.e. having several methods with the same name in one class. The overloaded methods must have a different number or types of arguments. For e.g., the class Person has 2 constructors:

1. A 2 arguments constructor that initializes the name and age of the new object to the values supplied, e.g.
   Person p1 = new Person("Yves", 44);

2. A 0 arguments constructor that initializes the attributes to default values, e.g.
   Person p1 = new Person();

In defining constructors, the result type need not be specified as it is always the constructor's class.

If you don't define any constructors, a 0 arguments constructor is automatically provided; it initializes the numeric attributes to 0, booleans to false, and objects to null.

About this

Within a class definition, this without parentheses always refers to the current instance of the class. It can be used to refer to the instance's attributes in a method that has a variable or parameter with the same name, e.g.

```java
public class Person
{
    ... public void setName(String name)
    {
        this.name = name;
    }
    ... private String name;
    private int age;
} // end class Person
```

Here the parameter name defined in the method hides the attribute name defined in the class; but you can refer to the latter using this.name. this(...) is a call to the class's constructor; see Horstmann p. 94.

E.g. Implementing the CreditCard Class

```java
// file CreditCard.java

public class CreditCard
{
    // constructors
    public CreditCard(int no, String aName, double aLimit)
    {
        assert 0 < no && no <= 999999 && aLimit > 0; // precondition
        number = String.format("%06d", no) + ";
        int digitSum = 0;
        while(no > 0)
        {
            digitSum = digitSum + no % 10;
            no = no / 10;
        }
        number = number + (MOD = digitSum % MOD);
        name = aName;
        limit = aLimit;
        balance = 0;
    }

    public CreditCard(int no, String aName)
    {
        this(no, aName, DEFAULT_LIMIT);
    }
```

public void pay(double amount)
{ assert amount >= 0; // precondition
balance = balance - amount;
}

// standard methods
public boolean equals(Object anObject)
{ return (anObject instanceof CreditCard &&
        number.equals(((CreditCard)anObject).number));
}

public String toString()
{ String res = "CARD [";
res = res + "NO=": number;
res = res + ", BALANCE=";
res = res + String.format("%.2f", balance) + "]";
return res;
}

// instance methods - accessors
public double getBalance()
{ return balance;
}
public double getLimit()
{ return limit;
}
public String getName()
{ return name;
}
public String getNumber()
{ return number;
}

public void credit(double amount)
{ assert amount >= 0; // precondition
balance = balance + amount;
}

// mutators
public boolean setLimit(double newLimit)
{ if (newLimit >= 0 && newLimit >= balance)
{ limit = newLimit;
return true;
}
else
{ return false;
}
}

// instance variables/attributes/fields
private String number;
private String name;
private double limit;
private double balance;

// class/static variables/attributes/fields
public static double DEFAULT_LIMIT = 1000.0;
public static int MIN_NAME_LENGTH = 3;
public static int MOD = 9;
public static int SEQUENCE_NUMBER_LENGTH = 6;

// end class CreditCard
Parameter Passing — The Details

As we saw earlier, parameter passing proceeds as follows:

1. the arguments are evaluated,
2. parameter variables are created,
3. the values of the arguments are copied into the parameter variables.

E.g. in the method call c1.charge(amount), the value of the argument amount, say 20.0 is first obtained, then a new formal parameter variable amount is created, and then the value of the amount argument, 20.0, is copied into this parameter variable. After this has been done, the body of the charge method is executed.

Same when the argument is an expression, e.g.
c1.charge(c1.getLimit() - c1.getBalance())

If method has several parameters, they are all passed in this way, e.g.
CreditCard c1 = new CreditCard(703,"John",5000.0).

Since the method is working with a copy of the argument, any changes made to the parameter variable don’t affect the argument. So you cannot use parameters of primitive types to return values in Java. E.g.
public class App
{
    public static void main(String[] args)
    {
        int n = 99;
        System.out.println("in main n = " + n);
        EgC1 e = new EgC1();
        e.increment(n);
        System.out.println("in main n = " + n);
    }
}

public class EgC1
{
    public void increment(int m)
    {
        System.out.println("in increment m = " + m);
        m = m + 1;
        System.out.println("in increment m = " + m);
    }
}

The mode of parameter passing used by Java is named call by value because it is the value of the argument which is passed to the formal parameter.

When the type of a parameter is an object type, only the reference in the argument gets copied in the parameter, and both the argument and parameter refer to the same object. So the method is working on the original object, and any change to its attributes persists when the method returns. E.g.
So, object parameters can be used by a method to send results back to the caller just as well as to receive data from the caller.

In languages like C++ and Pascal, there is a parameter passing mode named **call by reference** where the parameter receives a reference to the argument. The fact that object-type variables always contain references in Java makes object parameters behave somewhat as if they had been passed using call by reference.

---

**Control Flow and the Execution Stack**

```java
public class App
{
    public static void main(String[] args)
    {
        int n = 1;
        EgCl e = new EgCl();
        e.m1(n);
    }

    public class EgCl
    {
        public void m1(int n1)
        {
            n1 = n1 + 1;
            m2(n1);
        }

        private void m2(int n2)
        {
            n2 = n2 + 1;
            System.out.println(n2);
        }
    }
}
```

But unlike true call by reference, changing which object the parameter is referring to does not change which object the argument is referring to. E.g.

```java
public class App
{
    public static void main(String[] args)
    {
        String c = new CreditCard(703, "John");
        System.out.println("in main c = " + c.toString());
        EgCl e = new EgCl();
        e.m1(c);
        System.out.println("in main c = " + c.toString());
    }
}
```

```java
public class EgCl
{
    public void makeNewCard(CreditCard cm)
    {
        cm = new CreditCard(704, "Mary");
        System.out.println("in makeNewCard cm = " + cm.toString());
    }
}
```

When a method, say `main`, calls a method `m1` (on `e`), the execution of `main` is suspended and `m1` starts executing. Only when the execution of `m1` terminates will the execution of `main` resume. If `m1` calls a third method `m2`, `m1` is also suspended until `m2` terminates. The chain of method calls can get arbitrarily long.

Since the first method to be called is always the last to resume, the Java interpreter uses a stack to keep track of control flow in a program — the **execution stack**. We will see that this is important when encountering methods that call themselves, i.e., recursion. The stack is also used to store a method’s local variables.
Scope of Variables

The scope of a variable is the part of the program where it is visible, where it can be accessed. The variables declared inside a method, as well as its parameters, are said to be local to the method. One can only refer to them in the method or code block where they are declared. E.g.

Access Control Revisited

For attributes and methods, one specifies where they are visible using access control modifiers such as public and private.

public means that the attribute or method is accessible everywhere. Normally we use this only for methods and class constants that are made available to users of the class.

private means that the attribute or method is only accessible inside the class where it is declared. Normally we use this for all attributes and for methods that are defined by the implementor for his own use and are not provided to users of the class.

Besides these, there are other access control modifiers such as protected (accessible in subclasses and other classes in the same package) and the default/no modifier (accessible other classes in the same package), which we will discuss later.

Class Methods

Suppose we want to add a method isLegal(int no) to the CreditCard class to allow users to check whether the passed number would be a legal credit card number (before calling the constructor):

```java
public static boolean isLegal(int no) {
    return (0 < no && no <= 999999);
}
```

This cannot be an instance method because there is no instance yet. We can make it a class method. It would belong to the class, not to one of its instances. The method must be called on the class, e.g.

```java
if (CreditCard.isLegal(703)) ...
```

In Java, class methods (and attributes) are declared static.
Similarly, for the class Person discussed earlier, we could define a class method that returned the maximum legal age:

```java
public static int getUpperAgeLimit()
{ return 150;
}
```

To use it, you write for e.g.
```java
if (a > Person.getUpperAgeLimit())
```

And we would add a method:
```java
public static int getCount()
{ return count;
}
```

Then, we could use these as follows:
```java
System.out.println("Person count is " + Person.getCount());
Person p1 = new Person();
System.out.println("Person count is " + Person.getCount());
Person p2 = new Person();
System.out.println("Person count is " + Person.getCount());
```

Class Attributes

Suppose we wanted to add a counter to the class Person to keep track of how many Person objects have been created. This counter would have to be a class attribute variable.

To declare it, we would add
```java
private static int count = 0;
```

to the class definition. We would also change the constructors to increment the counter, e.g.
```java
public Person()
{ ... // as before
    count++;
}
```

Class Constants

A class may also define some constants for its users. For e.g., the class Stock defines the constant TSE_URL, to store the URL used by the refresh method to establish a connection with the TSE. To declare TSE_URL for e.g., we would write
```java
public static final String TSE_URL = "http://tse.com";
```

to the Stock class definition.

These are constant attributes of the class, not of its instances. You must refer to them using the class name, e.g.
```java
System.out.println(Stock.TSE_URL);
```
Why Define a Class?

There are two cases where defining a class is useful.

1. Your program needs to work with some kind of data, e.g. Persons. You want to group together the data and the operations that manipulate it.

You also want to hide the details of how the data is represented and how the operations are implemented from users of the class. The class will make some operations public, i.e. available to the users, and provide information on how to use them. This is the class’s interface. The rest of the class’s definition is private and hidden from users.

When such a class allows many different possible implementations, one says that the class defines an abstract data type; e.g. stack, list, binary tree, etc.

2. You want to group together a set of related operations in a module, e.g. the Math class. In this case, class users won’t create instances of the class. The methods are associated with the class itself. In Java, they are labeled static.

Here too, the class supplies some public operations to users and provides information on how to use them in its interface. The rest of its definition is private.

In both cases, we say that the class encapsulates, i.e. hides, the details of its definition.

javadoc: A Documentation Utility

Important to have good documentation of classes’ APIs.

Can use javadoc utility to help produce this.

You put special comments in the class’s file and then run javadoc on it to produce an HTML API documentation file.

javadoc comments start with /**. Put one immediately before each method, non-private field, and before the class itself.

Special tags (must start line):
@param parameter-name description
@return description
@exception fully-qualified-class-name description etc.

Can include other HTML tags e.g. <code>, <it>, etc.

See lab handbook and Horstmann for examples.

javadoc automatically adds links to existing classes.

When designing a class, document API using javadoc before writing code.

Use normal comments to document class implementation.
Steps to Class Implementation

Study API.

Write 1st version of class with fields and methods required by API, leaving out implementation for now; document using javadoc.

Write test harness that tests every feature of the class.

Identify private attributes and declare them.

Implement constructors, accessors, mutators, standard methods, specialized methods. Avoid redundancy by delegating and defining private methods.

Add new test cases as you implement methods. Test methods as early as possible. Fix bugs and run all tests again (bug fix may introduce new bugs).