JAVA™

AN INTRODUCTION TO
PROBLEM SOLVING
AND PROGRAMMING

WALTER SAVITCH

CLASS AND METHOD DEFINITIONS

Ch 5.1

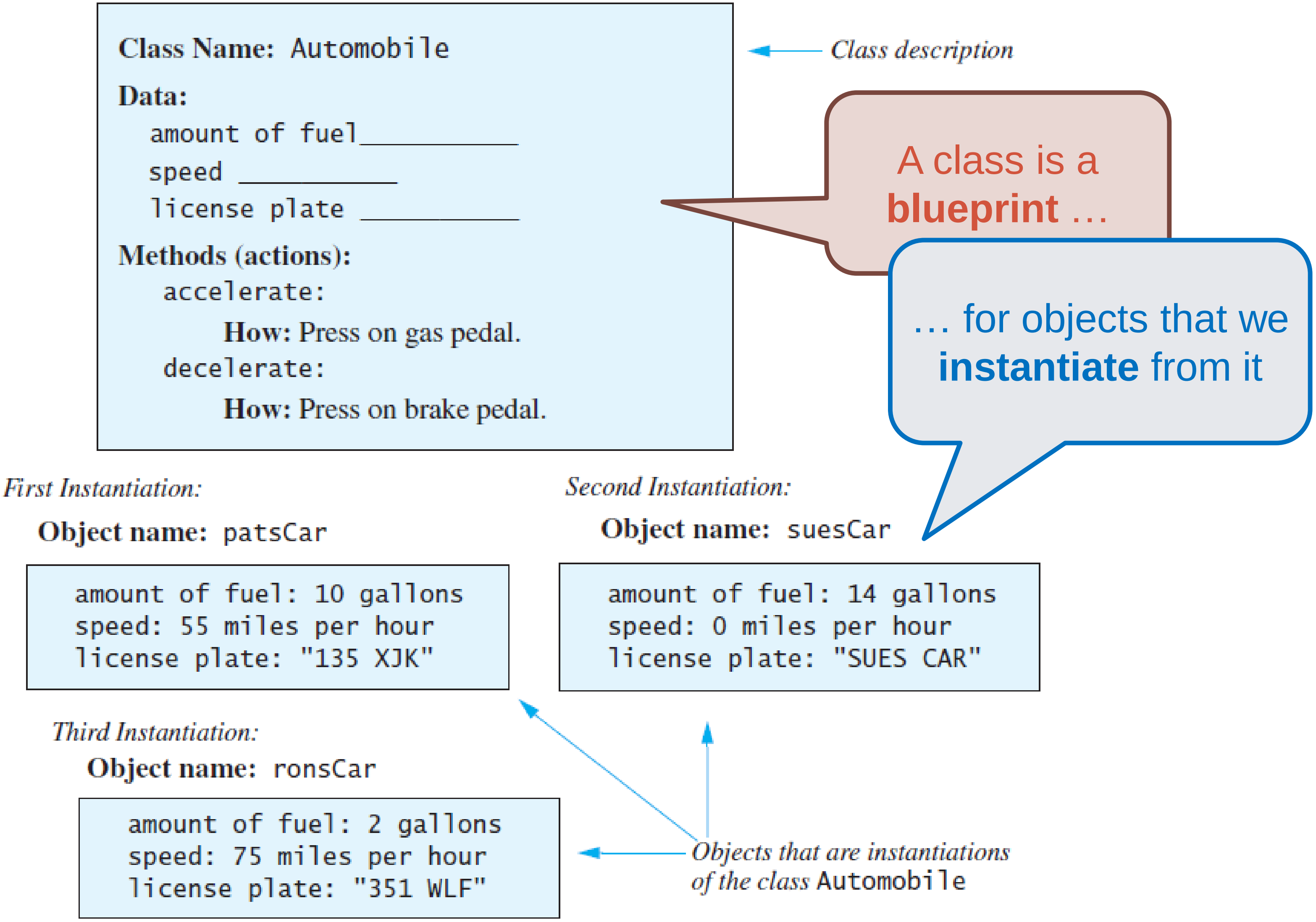
Class and Method Definitions: Outline

- Class Files and Separate Compilation
- Instance Variables
- Methods
- The Keyword this
- Local Variables
- Blocks
- Parameters of a Primitive Type

Class and Method Definitions

- Java program consists of objects
  - Objects of class types
  - Objects that interact with one another
- Program objects can represent
  - Objects in real world
  - Abstractions

FIGURE 5.1 **A Class as a Blueprint**



Class Name: `Automobile`

Data:

```
amount of fuel__________
speed __________
license plate __________
```

Methods (actions):

```
accelerate:
```

How: Press on gas pedal.

```
decelerate:
```

How: Press on brake pedal.

Class description

A class is a **blueprint** …

… for objects that we **instantiate** from it

First Instantiation:

Object name: `patsCar`

```
amount of fuel: 10 gallons
speed: 55 miles per hour
license plate: "135 XJK"
```

Second Instantiation:

Object name: `suesCar`

```
amount of fuel: 14 gallons
speed: 0 miles per hour
license plate: "SUES CAR"
```

Third Instantiation:

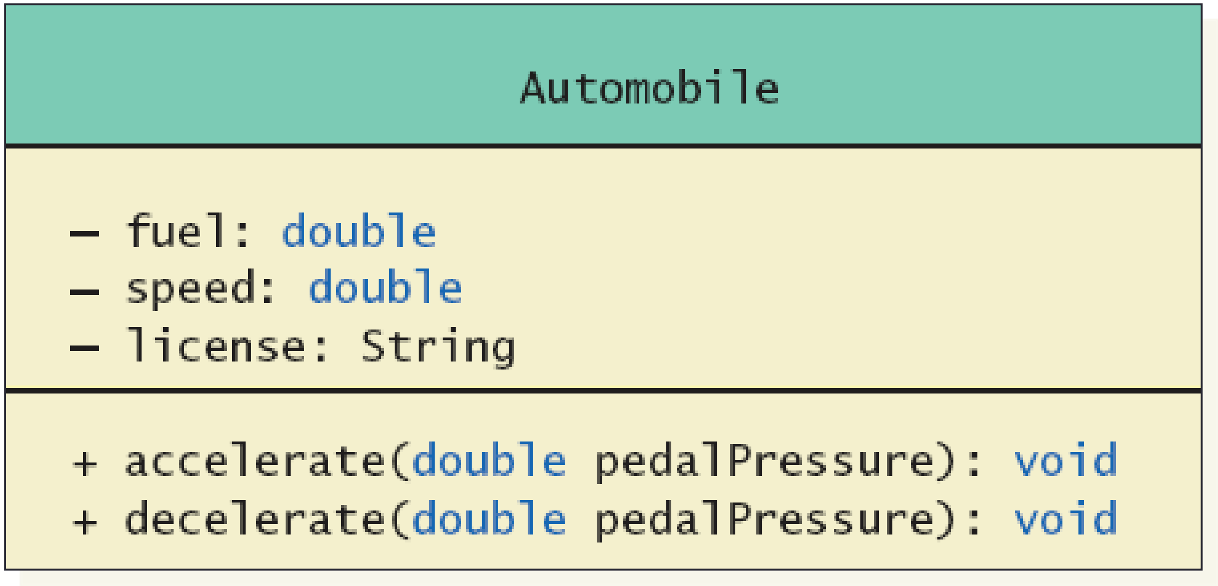
Object name: `ronsCar`

```
amount of fuel: 2 gallons
speed: 75 miles per hour
license plate: "351 WLF"
```

Objects that are instantiations of the class `Automobile`

Class and Method Definitions

- Figure 5.2 A class outline as a UML class diagram



```
Automobile

– fuel: double
– speed: double
– license: String

+ accelerate(double pedalPressure): void
+ decelerate(double pedalPressure): void
```

Methods

- Think of a method as defining an action to be taken (a segment of code)
- To start the action you invoke or call the method
- There are two kinds of Java methods
  - Return a single item
  - Return nothing – a **`void`** method
- The method **`main`** is a **`void`** method
  - Invoked by the system
  - Not by the application program

Methods

- To call a void method
  - Write the invocation followed by a semicolon
  - Resulting statement performs the action defined by the method
- To call a method that returns a quantity
  - Write the invocation anywhere a value matching the return type can be used
  - The call performs the action and the returned value will be used in the place of the invocation
- If you call a method that returns a value the same way you call a void method, the method will be executed, but the returned value will be lost.

Why use User-defined methods?

Using methods has several advantages:

- While working on one method, you can **focus** on just that part of the program/class and construct it, debug it, and perfect it.
- Different people can work on different methods **simultaneously**.
- If a method is needed in more than one place in a class, or in different programs, you can write it once and use it many times.
- Using methods greatly enhances the program's **readability** because it reduces the complexity of the program.

```
public class Dog
{   public String name;
    public String breed;
    public int age;

    public void writeOutput()
    {    System.out.println("Name:  " + name);
         System.out.println("Breed:  " + breed);
         System.out.println("Age in calendar years:  " + age);
         System.out.println("Age in human years:  " +
                                           getAgeInHumanYears());
         System.out.println();
    } // end writeOutput

    public int getAgeInHumanYears()
    {    int humanYears = 0;
         if (age <= 2)
                 humanYears = age * 11;
         else
                 humanYears = 22 + ((age-2) * 5);

         return humanYears;
    } // end getAgeInHumanYears
}
```

3 Instance variables or Data members or attributes

Will have different values for each Dog instance created. Each object will have its own copy

2 behaviors or methods

Will be the same for all Dog instances created, but act on individual instance variables.

```
public class DogDemo
{
  public static void main(String[] args)
  {
      Dog balto = new Dog();
      balto.name = "Balto";
      balto.age = 8;
      balto.breed = "Siberian Husky";

      balto.writeOutput();

      Dog scooby = new Dog();
      scooby.name = "Scooby";
      scooby.age = 42;
      scooby.breed = "Great Dane";

      System.out.println(scooby.name + " is a " +  scooby.breed + ".");
      System.out.print("He is " + scooby.age + " years old, or ");

      int humanYears = scooby.getAgeInHumanYears();

      System.out.println(humanYears + " in human years.");
  }
}
```

Dog Example

```
public class Dog
{   public String name;
    public String breed;
    public int age;

    public void writeOutput()
    { System.out.println("Name:  " + name);
      System.out.println("Breed:  "+breed);
      System.out.println("Age..:  "+ age);
      System.out.println(
        "Age in human years: " +
        getAgeInHumanYears() );
      System.out.println();
    } // end writeOutput

    public int getAgeInHumanYears()
    { int humanYears = 0;
      if (age <= 2)
        humanYears = age * 11;
      else
        humanYears = 22 + ((age-2) * 5)
      return humanYears;
    } // end getAgeInHumanYears
}
```

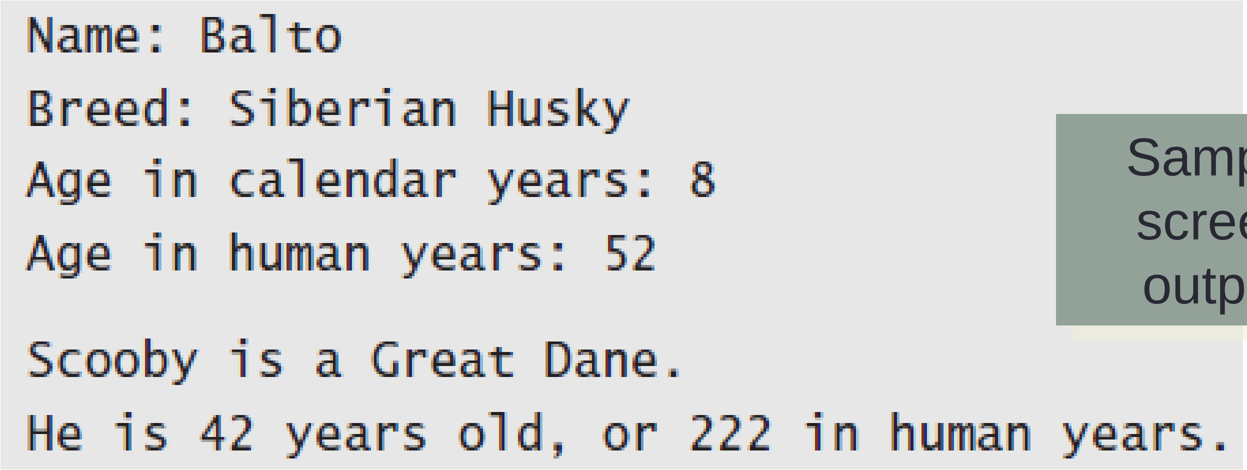
```
public class DogDemo {
public static void main(String[] args)
  {
     Dog balto = new Dog();
     balto.name = "Balto";
     balto.age = 8;
     balto.breed = "Siberian Husky
     balto.writeOutput();

     Dog scooby = new Dog();
     scooby.name = "Scooby";
     scooby.age = 42;
     scooby.breed = "Great Dane";
     System.out.println(scooby.name +
      " is a "+  scooby.breed +".");
     System.out.print("He is " +
      scooby.age + " years old, or ");
     int humanYears =
         scooby.getAgeInHumanYears();
     System.out.println(humanYears +
       " in human years.");
  } // end main
}
```

Using a Class and Its Methods

- View sample program, listing 5.2

  `class DogDemo`



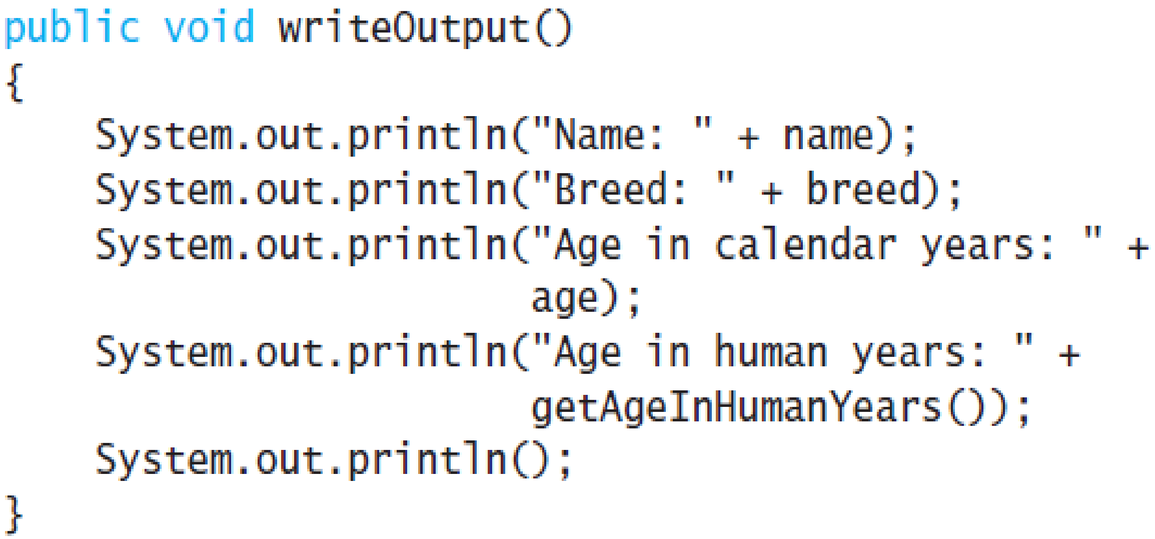
```
Name: Balto
Breed: Siberian Husky
Age in calendar years: 8
Age in human years: 52

Scooby is a Great Dane.
He is 42 years old, or 222 in human years.
```

Sample screen output

Defining Methods

- Consider method **`writeOutput`** from **`class dog`**



```
public void writeOutput()
{
    System.out.println("Name: " + name);
    System.out.println("Breed: " + breed);
    System.out.println("Age in calendar years: " +
                       age);
    System.out.println("Age in human years: " +
                       getAgeInHumanYears());
    System.out.println();
}
```

- Method definitions appear inside class definitions
- Methods can only be used with objects of that class

Defining Methods

- Most method definitions we will see as `public`
- A method that does **not** return a value is specified as a `void` method
- A method that does return a values must specify the **type** of the returned value.
- Heading includes possible parameters
- Body enclosed in braces `{ }`

Methods That Return a Value

- Consider method **`getAgeInHumanYears( )`**

```
public int getAgeInHumanYears()
    {   int humanYears = 0;
        if (age <= 2)
              humanYears = age * 11;
        else
              humanYears = 22 + ((age-2) * 5);

        return humanYears;
    } // end getAgeInHumanYears
```

`return` statement argument must match return type

- Heading declares type of value to be returned
- Last statement executed is **`return`**

The `return` statement

- Make sure of the following in the value-returning methods:
  - A value is returned.
  - Only a single value is returned to the caller method
  - The returned value has the same data type as the method

- Remember that the `return` statement:
  - is the last to execute in the method
  - make sure all paths are considered

Covering all paths

- Assume you want a method `hasLetter` for the class `Dog`, that checks if a given letter is contained in the name of the dog and returns its index, otherwise it prints an error message:

```
public int hasLetter (char letter){

int x = name.indexOf(letter);

if (x != -1)
   return x;
else
   System.out.print("Doesn't contain this letter");

System.out.println();
return -99;
}
```

What is wrong with this method?

The `return` is only in one path of all possible paths of execution.

How can we fix that?

There are multiple possibilities, and easy one is to add a return at the end

`return` is last to execute

- Assume you want a method `hasLetter` for the class `Dog`, that checks if a given letter is contained in the name of the dog and returns its index, otherwise it prints an error message:

```
public int hasLetter (char letter){

int x = name.indexOf(letter);

if (x != -1)
   return x;
else
   System.out.print(“Doesn’t contain this letter”);

return -99;
System.out.println();
}
```

What is wrong with this method now?

You can NOT have statements AFTER the `return`.

`return` in `void` Methods

- You can use `return` in void methods
- The syntax is simply:

```
return;
```

- No value is returned, but the control of the program is transferred back to the caller method.

Second Example – Account Class

```
public class Account {
    public String id, name;
    public double balance;
    public void readInput(){
        Scanner keyboard = new Scanner(System.in);
        System.out.println("Enter the account number: ");
        id = keyboard.nextLine();
        System.out.println("Enter the account holder name: ");
        name = keyboard.nextLine();
        System.out.println("Enter the account balance in riyals: ");
        balance = keyboard.nextDouble();}
    public void display(){
        System.out.println("\tAccount information");
        System.out.println("ID: " + id);
        System.out.println("Name: " + name);
        System.out.println("Balance: " + balance);
        System.out.println();}
    public double balanceInDollars() {
        double balanceDollars;
        balanceDollars = balance / 3.75;
        return balanceDollars;}}
```

```
public class AccountTest {
 public static void main(String[] args)
  {Account acc1 = new Account();
   acc1.readInput();
   acc1.display();
   Account acc2 = new Account();
   acc2.readInput();
   acc2.display();}}
```

The Keyword `this`

- Referring to instance variables:
  - outside the class – must use:
    - Name of an object of the class
    - Followed by a dot
    - Followed by Name of instance variable
  - Inside the class,
    - Use name of variable alone
    - The object (unnamed) is understood to be there
    - It is the receiving object

The Keyword `this`

- Inside the class the unnamed object can be referred to with the name `this`
- Example

```
this.name = keyboard.nextLine();
```

- The keyword `this` stands for the receiving object
- For simplicity Java allows you to omit it.
- We will see some situations later that require the use of `this`

Local Variables

- Variables declared inside a method are called local variables
    - Can only be used inside the method
    - For example:
        - All variables declared inside method **`main`** are local to **`main`**
- Local variables having the same name and declared in different methods are considered different variables

Local Variables

```
public class Account {
    public String id;
    public String name;
    public double balance;

    public void display(){
        System.out.println("\tAccount
        information");
        System.out.println("ID:  " +
        id);
        System.out.println("Name:  " +
        name);
        System.out.println("Balance:  "
        + balance);
        System.out.println();
    }
    public double balanceInDollars()  {
        double balanceDollars;
        balanceDollars = balance *
        3.75;
        return balanceDollars;
    }
}
```

```
public class AccountTest {
    public static void main(String[] args)
    {

    Account acc1 = new Account();
    acc1.id = "1111";
    acc1.name = "Mohammad";
    acc1.balance = 3000;
    acc1.display();
    Account acc2 = new Account();
    acc2.id = "2222";
    acc2.name ="Saad";
    acc2.balance = 1000;
    acc2.display();

    double balanceDollars;
    balanceDollars =
    acc1.balanceInDollars();
    System.out.println("Balance of " +
    acc1.name
    + " in dollars is "+ balanceDollars);
    }
```

Two different variables

Blocks

- Recall compound statements
  - Enclosed in braces { }
- When you declare a variable **within** a compound statement
  - The compound statement is called a block
  - The scope of the variable is from its declaration to the end of the block
- A variable declared **outside** the block is usable both outside and inside the block

Parameters of Primitive Type

```
public class Account {
    public String id;
    public String name;
    public double balance;

    public double credit(double amount){
        balance+= amount;
        return balance;
    }
    public double debit(double amount){
        if(amount <= balance)
                balance-=amount;
        else
                System.out.println("Amount exceeded
        balance");

        return balance;
    }
    // the rest of the previously defined methods
}
```

- Note that both credit and debit methods take one parameter which is double
- The **formal** parameter is `amount`

Parameters of Primitive Type

```
public class AccountTest {
    public static void main(String[] args) {

        Account acc1 = new Account();
        acc1.id = "1111"; acc1.name = "Mohammd";
        acc1.balance = 3000;
        double newBalance = acc1.credit(1000);
        System.out.println("The new balance "
        + "(after calling credit) is " + newBalance);
        newBalance = acc1.debit(500);
        System.out.println("The new balance "
        + "(after calling debit) is " + newBalance);
    }
}
```

- Calling the method

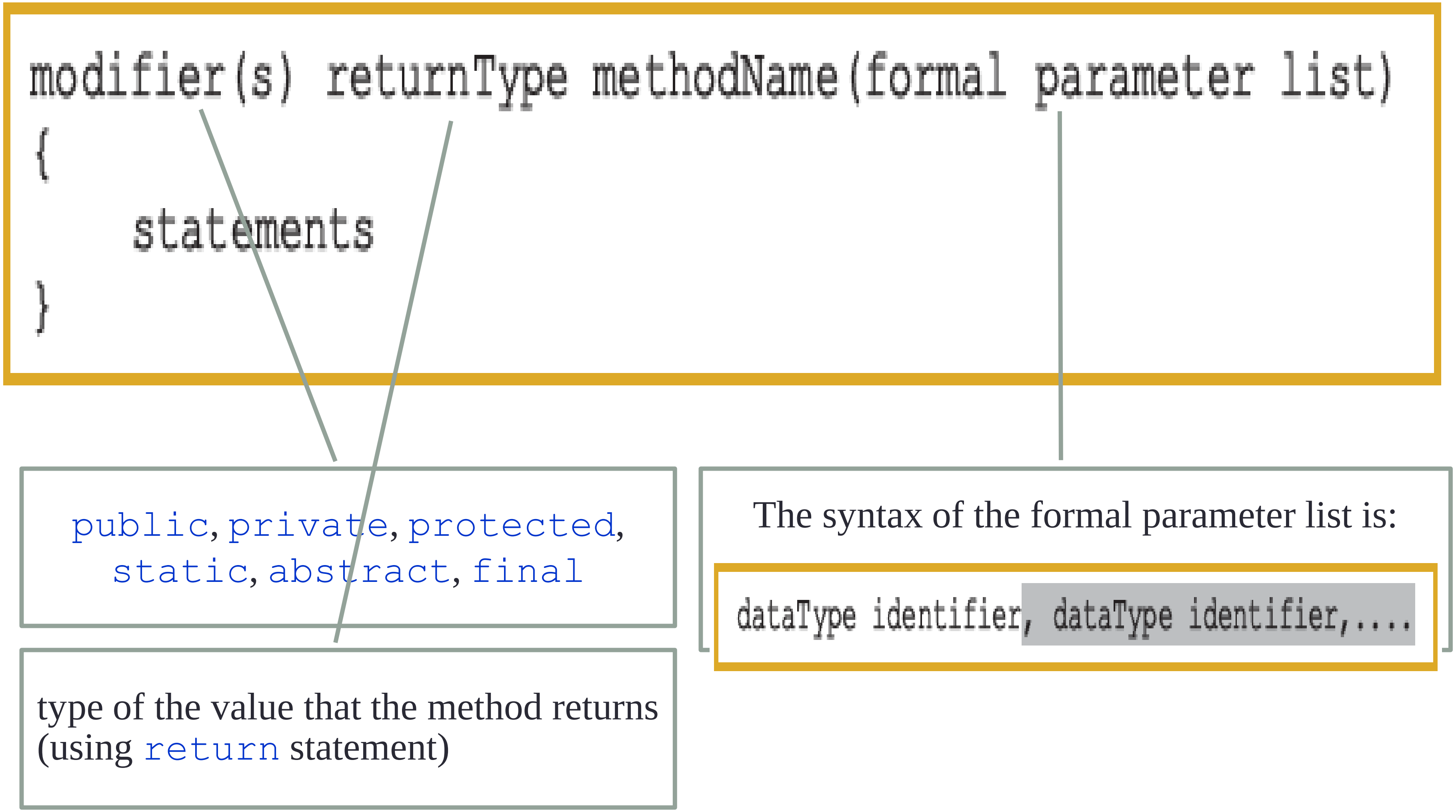
```
double newBalance =
acc1.credit(1000);
```

- The **actual** parameter is *the double 1000*

Sample screen output

```
The new balance (after calling credit) is 4000.0
The new balance (after calling debit) is 3500.0
```

Syntax: Method



```
modifier(s) returnType methodName(formal parameter list)
{
    statements
}
```

`public`, `private`, `protected`, `static`, `abstract`, `final`

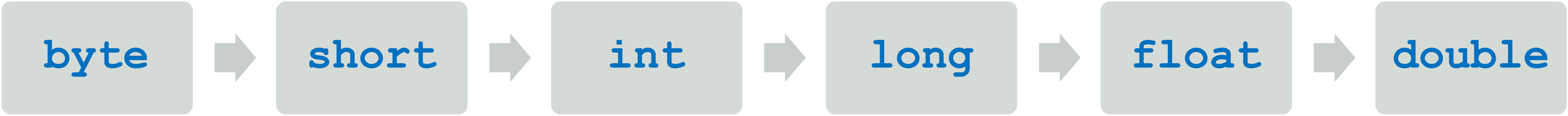
type of the value that the method returns (using `return` statement)

The syntax of the formal parameter list is:

```
dataType identifier, dataType identifier,...
```

Parameters of Primitive Type

- Parameter names are local to the method
- When a method is invoked
  - Each parameter is initialized to the value in the corresponding actual parameter
  - A primitive actual parameter is not (and cannot be) altered by invocation of the method
    - We will learn later that this is not the case for actual parameters of non-primitive types.
- Automatic type conversion is performed



byte → short → int → long → float → double

Passing +1 Parameters of Primitive Type

```
public class X {

   public double n;
    public void Y(int i, int j){
         System.out.println(i + j);
         n++;
    }
    public void Z(double i) {
         System.out.println(n + i);
    }
}
```

```
public class Test {
     public static void main(String[] args)
     {
          X x = new X();
          x.n = 2;
          x.Y(5,6);
          int t1= 1, t2 = 3;
          x.Y(t1,t2);
          x.Z(x.n);
     }
}
```

Sample screen output

```
11
4
8.0
```

The use of the Keyword **this**

```
public class X {

   public double n;
    public void Y(int i, int j){
        System.out.println(i + j);
        n++;
    }
    public void Z(double n)  {
        System.out.println(this.n + n);
    }
}
```

```
public class Test {
    public static void
    main(String[] args)  {

        X x = new X();
        x.n = 2;
        x.Y(5,6);
        int t1= 1, t2 = 3;
        x.Y(t1,t2);
        x.Z(6);
        System.out.println(x.n);

    }
}
```

Sample screen output

```
11
4
10.0
4.0
```