Understanding Class Definitions

Java version

Objectives

- To learn more about developing classes.
- To get practical experience with objects and classes.

Main concepts discussed in this chapter

- fields
- methods (accessor, mutator)
- constructors
- assignment and conditional statement
- parameters

Resources

Classes needed for this lab - *chapter02.jar*.

To do

Ticket machines

Exploring the behaviour of a naïve ticket machine

Open the *naiveticketmachine* package. Ticket prices are given in cents. You will need to create a class with a **`main`** method to run the application.

Paranoia – WTF?

You're going to have trouble compiling the code! The **`naiveticketmachine.TicketMachine`** class has been secured, to the point of unusability – everything is declared **`private`**. You'll have to progressively ease up on the protections, but only as much as necessary, ie, don't just systematically replace **`private`** with **`public`**. Think!

Exercises

1. Create a **`TicketMachine`** object and take a look at its methods. You should see the following: **`getBalance`**, **`getPrice`**, **`insertMoney`**, and **`printTicket`**. Try out the **`getPrice`** method. You should see a return value containing the price of the tickets that was set when this object was created. Use the **`insertMoney`** method to simulate inserting an amount of money into the machine and

then use **getBalance** to check that the machine has a record of the amount inserted. You can insert several separate amounts of money into the machine, just like you might insert multiple coins or notes into a real machine. Try inserting the exact amount required for a ticket. As this is a simple machine, a ticket will not be issued automatically, so once you have inserted enough money, call the **printTicket** method. A facsimile ticket should be printed.

2. What value is returned if you check the machine's balance after it has printed a ticket?
3. Experiment with inserting different amounts of money before printing tickets. Do you notice anything strange about the machine's behavior? What happens if you insert too much money into the machine – do you receive any refund? What happens if you do not insert enough and then try to print a ticket?
4. Try to obtain a good understanding of a ticket machine's behaviour by trying out different methods before we start looking at how the **TicketMachine** class is implemented in the next section.
5. Create another ticket machine for tickets of a different price. Buy a ticket from that machine. Does the printed ticket look different?

Examining a class definition

Look at the source code for the **TicketMachine** class.

```
package naiveticketmachine;

/**
 * TicketMachine models a naive ticket machine that issues flat-fare
 * tickets. The price of a ticket is specified via the constructor.
 * It is a naïve machine in the sense that it trusts its users to insert
 * enough money before trying to print a ticket. It also assumes that
 * users enter sensible amounts.
 *
 * @author David J. Barnes and Michael Kölling
 * @version 2016.02.29
 */
class TicketMachine {
    // The price of a ticket from this machine.
    private final int price;
    // The amount of money entered by a customer so far.
    private int balance;
    // The total amount of money collected by this machine.
    private int total;

    /**
     * Create a machine that issues tickets of the given price.
     * Note that the price must be greater than zero, but there
     * are no checks to ensure this.
     */
    TicketMachine(int cost) {
        price = cost;
        balance = 0;
        total = 0;
    }

    /**
     * Return the price of a ticket.
     */
    int getPrice() {
```

```
        return price;
    }

    /**
     * Return the amount of money already inserted for the next ticket.
     */
    int getBalance() {
        return balance;
    }

    /**
     * Receive an amount of money from a customer.
     */
    void insertMoney(int amount) {
        balance = balance + amount;
    }

    /**
     * Print a ticket. Update the total collected and reduce the balance to
     * zero.
     */
    void printTicket() {
        // Simulate the printing of a ticket.
        System.out.println("####################");
        System.out.println("# The BlueJ Line");
        System.out.println("# Ticket");
        System.out.println("# " + price + " cents.");
        System.out.println("####################");
        System.out.println();

        // Update the total collected with the balance.
        total = total + balance;
        // Clear the balance.
        balance = 0;
    }
}
```

Class header

A Java class definition consists of a wrapper and an inner part

```
access class TicketMachine {
    Inner part of the class omitted.
}
```

The outer wrapping contains the class header which basically provides the name of the class (starting with a capital letter); the inner part does the work.

The access level for a class can be

- _____, ie, nothing. This is the default, or package-private level. This class can only be accessed from other classes within the same package. It is meant to only be used by developer-side code.
- **public** - this class is accessible from anywhere and can be used by client-side code.

Exercises

6. Write out what you think the outer layers of the **`Student`** and **`LabClass`** classes might look like – do not worry about the inner part (the class body).
7. Does it matter if we write
   `public class TicketMachine`
   or
   `class public TicketMachine`
   in the outer wrapper of a class? Edit the source of the **`TicketMachine`** to make the change. What error message do you get when you now compile the code?
8. Check whether or not it is possible to leave out the word **`public`** from the outer wrapper of the **`TicketMachine`** class.
9. Put the word **`public`** back and leave out the word **`class`**. Does this work?

Keywords

"**`public`**" and "**`class`**" are Java *reserved words* which can't be used for names. There are many others.

Fields, constructors, and methods

A Java class, eg, **`TicketMachine`** typically has the following structure

```
class TicketMachine {
    // fields
    // constructor(s)
    // methods
}
```

- Fields store persistent data within an object.
- Constructors ensure that an object is set up correctly when it is first built.
- Methods implement the behaviour of an object. They determine its functionality.

Exercises

10. Look at the class definition in the source code of **`TicketMachine`** and make a list of the names of the fields, constructors and methods in the **`TicketMachine`** class. Hint: There is only one constructor in the class.
11. Do you notice any features of the constructor that make it significantly different from the other methods of the class?

Fields

Fields store values inside objects. Fields are also called *attributes* or *instance variables*. The **`TicketMachine`** class has three fields

- **`price`** stores the price of a ticket.
- **`balance`** is the amount of money inserted into the machine before printing a ticket.
- **`total`** is the amount of money inserted into the machine since the object was instantiated (excluding the current balance). The balance is added to the total when a ticket is printed.

```
class TicketMachine {
    private final int price;
```

```
    private int balance;
    private int total;

    Constructor and methods omitted.
}
```

price is declared **final** since once a value is assigned it never changes after it is initialized. The values of **balance** and **total** can change during execution. It is generally a good idea to declare variables which don't change as **final**.
We see that all the fields of **TicketMachine** are of type **int**, meaning integer. On the other hand, the fields of **LabClass** were declared as

```
class LabClass {
    private String instructor;
    private String room;
    private String timeAndDay;
    private List students;
    private int capacity;

    Constructor and methods omitted.
}
```

where three fields are of type **String**, one is of type **int**, and one is of type **List**, which is a Java type we'll encounter later. Since the values of the fields can change, the fields really are variables, called *instance variables*.
All fields are declared **private** – they are intended for use only in the class itself.
Source code following **//** is a comment for people reading the class definition (it does not form part of the executable code). Source code between **/*** and ***/** is a long comment.

Exercises

12. What is the type of each of the following variables or fields
```
private int count;
private Student representative;
private Server host;
```

13. What are the names of the following fields ?
```
private boolean alive;
private Person tutor;
private Game game;
```

14. Which of the names in 12, 13 are class names?
15. In the following field declaration from the **TicketMachine** class
```
private int price;
```
does it matter which order the three words appear in? Edit the **TicketMachine** class to try different orderings. Make sure you install the original version after your experiments!
16. Is it always necessary to have a semicolon at the end of a field declaration?
17. Write in full the declaration for a field of type **int** whose name is **status**.

Note that

- Field names start with a lower case letter.
- The field declaration starts with the key word **private**.
- The field declaration contains the key word **final** if its value never changes.

- The declaration includes the type of the field, **`int`**, **`String`**, **`Person`**, etc.
- The declaration ends with a `;`.

Constructors

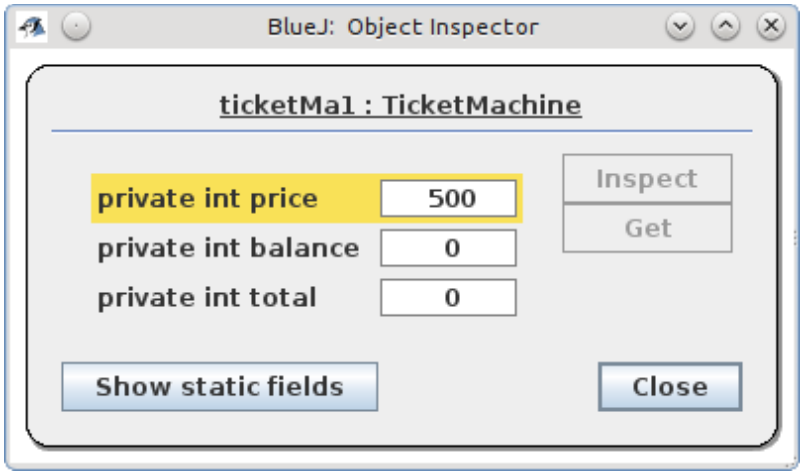
A constructor is called when an object is being built and its job is to ensure that its variables are correctly initialized. A constructor must have the same name as the class, and unlike a method it declares no return type. For instance, the **`TicketMachine`** constructor looks like

```
class TicketMachine {
    Fields omitted.

    /**
     * Create a machine that issues tickets of the given price.
     * Note that the price must be greater than zero, but there
     * are no checks to ensure this.
     */
    TicketMachine(int ticketCost) {
        price = ticketCost;
        balance = 0;
        total = 0;
    }

    Methods omitted.
}
```

For a class visible from the client-side, the declarations would be

```
public class TicketMachine {
    public TicketMachine(int ticketCost) {
        Code omitted.
    }
}
```

The values of the argument **`ticketCost`** is passed into the constructor as an argument, and then assigned to the **`ticketCost`** instance variable. After construction, a **`TicketMachine`** object could be represented as



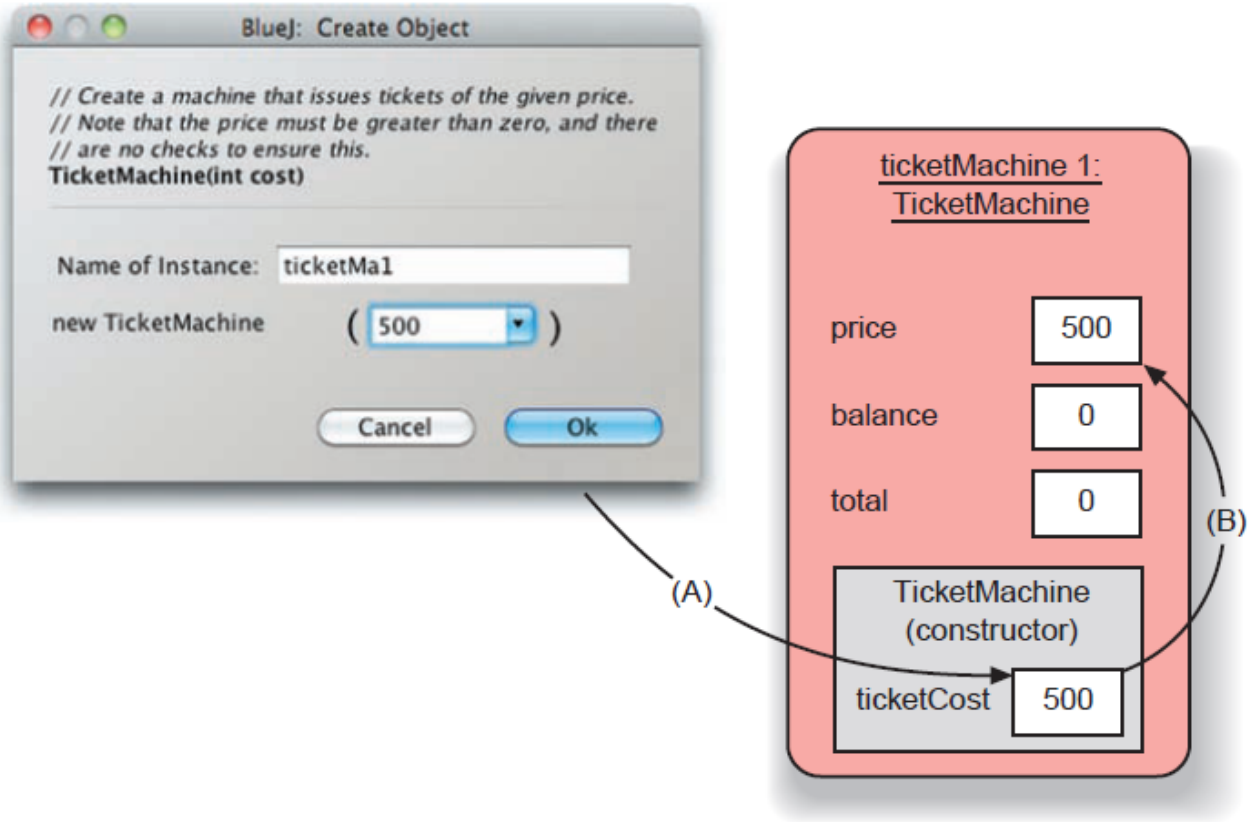
By default, attributes are initialized to the equivalent of zero (**`0`**, **`null`**, **`false`**). Thus the **`balance`** and **`total`** initialization statement could have been omitted, but it's considered good practice to make them explicit.

Parameters: receiving data

The line below is the signature of the constructor of the **`TicketMachine`** class. Its name is the name of the class (**`TicketMachine`**) and it indicates the number of arguments and their types (a single argument called **`ticketCost`** of type **`int`**), and the level of access for the method (no keyword, so package-private here). Note that there is no return type.

```
TicketMachine(int ticketCost) {
    Code omitted.
}
```

The ticket cost has to be passed into the constructor because its value is set by the user, see the first arrow in the diagram below. In the code, **`ticketCost`** is called the *formal parameter* of the constructor and the value 500 is called the *real parameter*. When the constructor executes, the real parameter (500) is assigned as the value of the formal parameter **`ticketCost`**. The formal parameter is actually a variable since it can take whatever value the user gives it. Thus the attribute **`price`** is assigned the value 500, the value of the formal parameter **`ticketCost`**. A formal parameter can also be called an *argument*.
In the constructor code, the curly brackets **`{`** and **`}`** delimit the *scope* of the constructor, and any variable declared inside the parameters, including the argument, can only be seen from within the scope. It is not visible from outside the scope. An attribute is declared within the scope of a class and can thus be seen from anywhere in the class.



The *lifetime* of a variable is also related to its scope, eg, the argument is available while the constructor is

executing, and once it has finished executing its three lines of code, the argument no longer exists. The lifetime of any variable is only within its scope (we'll see more on lifetime and scope later).

Exercises

18. To what class does the following constructor belong?
    `Student(String name)`
19. How many parameters does the following constructor have and what are their types?
    `Book(String title, double price)`
20. Can you guess what types some of the **`Book`** class's fields might be? Can you assume anything about the names of its fields?

Choosing variable names

Variables, methods and classes all should have names that relate to their purpose. **`price`** says more about the function of that variable than would, eg, **`p1`**.

Assignment

The following line in the constructor code

```
price = ticketCost;
```

is a Java *assignment statement* (and not an equality!). It says that the value of the variable **`ticketCost`** is to be assigned as the new value for the variable **`price`**.
In an assignment statement, the variables must be of the correct type.

Exercises

21. Suppose that the class **`Pet`** has a field called **`name`** that is of the type **`String`**. Write an assignment statement in the body of the following constructor so that the **`name`** field will be initialized with the value of the constructor's parameter.
    ```
    Pet(String petName) {
        ...
    }
    ```
22. What would be the constructor signature that would correspond to the following expression?
    `new Date("March", 23, 1861)`

Methods

Methods have a *signature* and a *body*. The first line after the comment is the *signature* of **`getPrice`**

```
/**
 * Return the price of a ticket.
 */
public int getPrice() {
    return price;
}
```

The *signature* indicates its name (**`getPrice`**), the number of arguments (zero), the type of value returned by the method (**`int`**), and the level of access for the method (here **`public`**). The signature of a method can also be called the method *header*.

Between the **{}**s is the code in the body of the method. The code implements the behaviour of the method as given by its signature. Here, it returns that value **price** as the result of executing the method.
A method doesn't have to return a result, and this is indicated by the return type **void**. When a result is returned, its type must agree with the type declared in the method header. A constructor doesn't declare that it returns anything, and has no return type.

Accessor and mutator methods

Methods such as **getPrice** and **getBalance** are called *accessor methods* (or *getter* methods). They return information about an object's state, generally its value.

Exercises

23. Compare the **getBalance** method with the **getPrice** method. What are the differences between them?
24. If a call to **getPrice** can be characterized as "What does a ticket cost?", how would you characterize a call to **getBalance**?
25. If the name of **getBalance** is changed to **getAmount**, does the return statement in the body of the method need to be changed, too? Try it out.
26. Define an accessor method, **getTotal**, that returns the value of the **total** field.
27. Try removing the return statement from the body of **getPrice**. What error message do you see now when you try compiling the class?
28. Compare the method signatures of **getPrice** and **printTicket** in the **TicketMachine** source code. Apart from their names, what is the main difference between them?
29. Do the **insertMoney** and **printTicket** methods have return statements? Why do you think this might be? Do you notice anything about their headers that might suggest why they do not require return statements?

Mutator methods change the state of an object. The are often called *setter* methods because their names are frequently **setSomething**. In the same way as we think of accessors as requests for information (questions), we can think of mutators as requests for an object to change its state.

Exercise

30. Create a ticket machine with a ticket price of your choosing. Before doing anything else, call the **getBalance** method on it. Now call the **insertMoney** method and give a non-zero positive amount of money as the actual parameter. Now call **getBalance** again. The two calls to **getBalance** should show different output because the call to **insertMoney** had the effect of changing the machine's state via its **balance** field.

The **insertMoney** method has a return type of **void** indicating that the method returns nothing.

```
void insertMoney(int amount) {
    balance = balance + amount;
}
```

Exercise

31. How can we tell from just the header that **setPrice** is a method and not a constructor?
    void setPrice(int ticketCost)
32. Complete the body of the **setPrice** method so that it assigns the value of its parameter to the **price** field. What do you have to change?
33. Complete the body of the following method, whose purpose is to add the value of its parameter to a field

named **score**.

```
/**
 * Increase score by the given number of points.
 */
void increase(int points) {
    ...
}
```

34. Is the **increase** method a mutator? How can you show this?
35. Add the following method which subtracts a given value from the price attribute

```
/**
 * Reduce price by the given amount.
 */
void discount(int amount) {
    ...
}
```

Getters and setters are generally frowned upon, and setters are considered by some to be downright evil. They are not object-oriented - they give away too much information about the implementation of objects. More on this later.

Beware of development environments which automatically generate getters and setters – never give in to the temptation!

Printing from methods

Calling the **printTicket** method produces something like the following output

```
##################
# The BlueJ Line
# Ticket
# 500 cents.
##################
```

Lets look at the **printTicket** method itself to see where this comes from

```
void printTicket() {
    // Simulate the printing of a ticket.
    System.out.println("##################");
    System.out.println("# The BlueJ Line");
    System.out.println("# Ticket");
    System.out.println("# " + price + " cents.");
    System.out.println("##################");
    System.out.println();

    // Update the total collected with the balance.
    total = total + balance;
    // Clear the balance.
    balance = 0;
}
```

- The signature says it takes no arguments and returns no value.
- There are eight statements in the body.

- The first six statements print the output, including the blank line.
- The next statements increment the total and set the balance to zero.

The result of **`"# " + price + " cents."`** is a character string (a **`String`**) like **`"# 500 cents."`** when the value of **`price`** is 500. The **`+`** operator here is *concatenation* and not addition; it says to concatenate two character strings together. If one of the operands of **`+`** is a character string and the other argument is not a character string, then Java automatically converts the non-character string into a character string before concatenating them. It seems clear that **`System.out.println`** is a magic incantation for printing character strings passed as arguments.

Exercises

36. What *exactly* is printed by
    `System.out.println("My cat has green eyes.");`
37. Add a method called **`prompt`** to the **`TicketMachine`** class. This should have a **`void`** return type and take no parameters. The body of the method should print something like:
    `Please insert the correct amount of money.`
38. What do you think would be printed if you altered the fourth statement of **`printTicket`** so that price also has quotes around it, as follows?
    `System.out.println("# " + "price" + " cents.");`
39. What about the following version?
    `System.out.println("# price cents.");`
40. Could either of the previous two versions be used to show the price of tickets in different ticket machines? Explain your answer.
41. Add a **`showPrice`** method to the **`TicketMachine`** class. This should have a **`void`** return type and take no parameters. The body of the method should print something like:
    `The price of a ticket is` *`xyz`* **`cents.`**
    where *`xyz`* should be replaced by the value held in the **`price`** field when the method is called.
42. Create two ticket machines with differently priced tickets. Do calls to their **`showPrice`** methods show the same output or different? How do you explain this effect?

Exercise

I. Continuing with Ex.41: for the naive ticket machine, is there any point in having the **`getPrice`** and **`getBalance`** methods? Can they be replaced with **`showPrice`** and **`showBalance`**? What are the advantages / disadvantages of each approach?

Method summary

Methods define the behaviour of objects.

Some methods (but not all) take arguments which are necessary for their behaviour.

Some methods return data to where they were called from. Some methods, with return type **`void`**, do not return anything but just manipulate the object's attributes or perform some function.

Accessor methods return information about an object's state. They have no argument and return a value. Mutator methods change an object's state. They generally take arguments but return nothing.

> ☞ Note: before writing any accessor / mutator (getter / setter) methods, think very carefully about whether you really really need them. They are considered un-object-oriented as they can reveal too much about an object's data. It is better to *tell* an object what to do with its data, eg, **`ticketMachine.showPrice()`**, than to *ask* an object for its data in order to do something with it, eg, **`System.out.println(ticketMachine.getPrice())`**. This is sometimes called the principle, *Tell Don't Ask*. We'll be seeing more of this later.

Summary of the naïve ticket machine

Exercises

43. Modify the constructor of **`TicketMachine`** so that it no longer has a parameter. Instead, the price of tickets should be fixed at 1,000 cents.
44. Give the class two constructors. One should take a single parameter that specifies the price, and the other should take no parameter and set the price to be a default value of your choosing. Test your implementation by creating machines via the two different constructors.
45. Implement a method, **`empty`**, that simulates the effect of removing all money from the machine. This method should have a **`void`** return type, and its body should simply set the **`total`** field to zero. Does this method need to take any parameters? Test your method by creating a machine, inserting some money, printing some tickets, checking the total and then emptying the machine. Is this method a mutator or an accessor?

Reflecting on the design of the ticket machine

Basically, the naïve ticket machine is not very good, and might only be used by the SNCF. Among other things

- It contains no check that the customer has entered enough money to pay for a ticket.
- It does not refund any money if the customer pays too much for a ticket.
- It does not check to ensure that the customer inserts sensible amounts of money: experiment with what happens if a negative amount is entered, for instance.
- It does not check that the ticket price passed to its constructor is sensible.

We'll now work on a *betterticketmachine* - open the package with that name.

Making choices: the conditional statement

Here's the initial code for a better machine:

```
package betterticketmachine;

/**
 * TicketMachine models a ticket machine that issues flat-fare tickets. The
 * price of a ticket is specified via the constructor. Instances will check
 * to ensure that a user only enters sensible amounts of money, and will
 * only print a ticket if enough money has been input.
 *
 * @author David J. Barnes and Michael Kölling
 * @version 2016.02.29
 */
class TicketMachine {
    // The price of a ticket from this machine.
    private final int price;
    // The amount of money entered by a customer so far.
    private int balance;
    // The total amount of money collected by this machine.
    private int total;

    /**
     * Create a machine that issues tickets of the given price.
     */
    TicketMachine(int cost) {
        price = cost;
```

```
    balance = 0;
    total = 0;
}

/**
 * Return the price of a ticket.
 */
int getPrice() {
    return price;
}

/**
 * Return the amount of money already inserted for the next ticket.
 */
int getBalance() {
    return balance;
}

/**
 * Receive an amount of money from a customer. Check that the amount is
 * sensible.
 */
void insertMoney(int amount) {
    if (amount > 0) {
        balance = balance + amount;
    } else {
        System.out.println("Use a positive amount rather than: "
                + amount);
    }
}

/**
 * Print a ticket if enough money has been inserted, and reduce the
 * current balance by the ticket price. Print an error message if more
 * money is required.
 */
void printTicket() {
    if (balance >= price) {
        // Simulate the printing of a ticket.
        System.out.println("####################");
        System.out.println("# The BlueJ Line");
        System.out.println("# Ticket");
        System.out.println("# " + price + " cents.");
        System.out.println("####################");
        System.out.println();

        // Update the total collected with the price.
        total = total + price;
        // Reduce the balance by the price.
        balance = balance - price;
    } else {
        System.out.println("You must insert at least: "
                + (price - balance)
                + " more cents.");
```

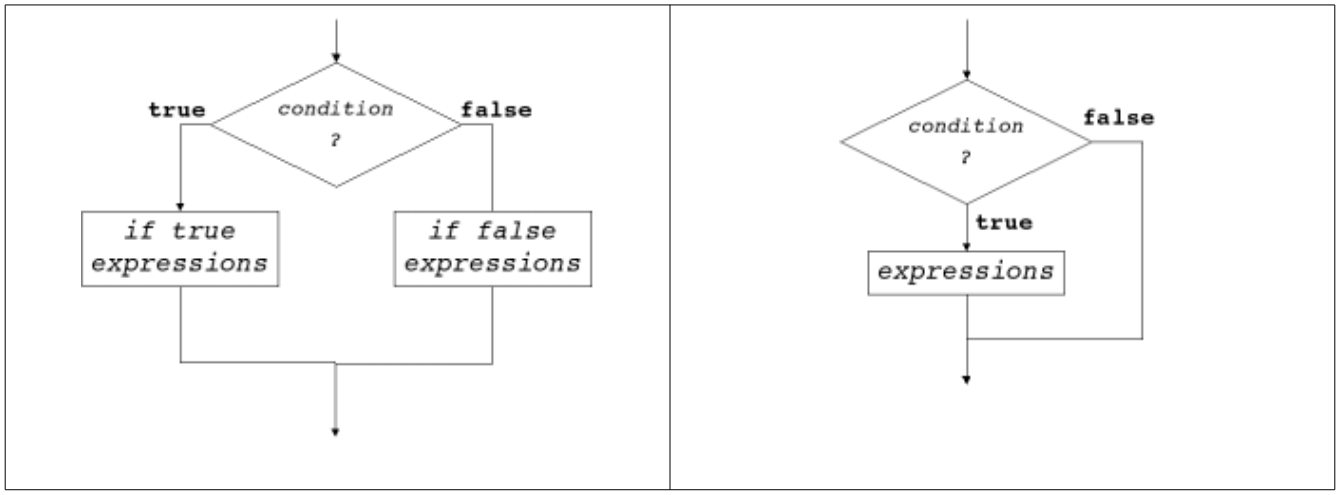
```
        }
    }

    /**
     * Return the money in the balance. The balance is cleared.
     */
    int refundBalance() {
        int amountToRefund;
        amountToRefund = balance;
        balance = 0;
        return amountToRefund;
    }
}
```

Note that this class has the same name as the previous version, **`TicketMachine.`** However, the two classes are in different packages, and this allows Java to distinguish between them. We do this often.

Look first at the new **`insertMoney`** method

```
void insertMoney(int amount) {
    if (amount > 0) {
        balance = balance + amount;
    }
    else {
        System.out.println("Use a positive amount rather than: "
                + amount);
    }
}
```

The body of the method forms a *conditional statement* which allows the program to take different actions based on a test. If the amount is positive we add it to the total, otherwise we complain and don't do anything. This corresponds to the general schema on the left below.



Note that it's also possible to have no else statement in the conditional, corresponding to the general schema on the right.
The condition, here **`amount > 0`**, must return one of the Boolean values **`true`** or **`false`**. Other relational

operators include the following

Math	Java
<	<
≤	<=
=	==
≥	>=
>	>
≠	!=

Note the difference between the mathematical relation = and the Java relation **==**.

Exercises

46. Check that the behavior we have discussed here is accurate by creating a **TicketMachine** instance and calling **insertMoney** with various actual parameter values. Check the **balance** both before and after calling **insertMoney**. Does the balance ever change in the cases when an error message is printed? Try to predict what will happen if you enter the value zero as the parameter, and then see if you are right.
47. Predict what you think will happen if you change the test in **insertMoney** to use the greater-than or equal-to operator:
    ```
    if (amount >= 0)
    ```
    Check your predictions by running some tests. What difference does it make to the behavior of the method?
48. Rewrite the if-else statement so that the error message is printed if the condition is true, and the balance is increased if the condition is false.

Exercises

II. Which of the following work as expected? Note that **a -= b** is a short form for **a = a - b**.
    a.
    ```
    if (speed = 160) {
      points == points - 8;
    }
    ```
    b.
    ```
    if (speed == 130) {
      points -= 6;
    }
    ```
    c.
    ```
    if (speed = 110) {
      points -= 4;
    }
    ```
    d.
    ```
    if (speed <= 90) {
      points -= 0;
    }
    ```
III. Write some code to test your hypotheses. You should create a new Java package, let's call it **radar**. Then create a new class and call it, uh, **Radar**.

A further conditional-statement example

The **printTicket** method contains a further example of a conditional statement.

```
void printTicket() {
```

```
        if (balance >= price) {
            // Simulate the printing of a ticket.
            Printing details omitted.

            // Update the total collected with the price.
            total = total + price;
            // Reduce the balance by the prince.
            balance = balance - price;
        }
        else {
            System.out.println("You must insert at least: "
                    + (price - balance) + " more cents.");

        }
    }
```

This version of the method checks that the customer has put in enough money to pay for the ticket; if not then the machine complains.

Exercise

50. In this version of **printTicket** we also do something slightly different with the **total** and **balance** fields. Compare the naïve implementation of the method with that above to see whether you can tell what those differences are. Then check your understanding by experimenting.
51. Is it possible to remove the else statement and block of code? Does the code compile? What happens when it runs?

Scope

Scope (also called a *block*) is the code delimited by **{}**s.

Exercises

52. After a ticket has been printed, could the value in the **balance** field ever be set to a negative value by subtracting **price** from it? Justify your answer.
53. So far you have seen just two arithmetic operators, **+** and **-**, that can be used in arithmetic expressions in Java. Take a look at *http://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html* to find out what other operators are available.
54. Write an assignment statement that will store the result of multiplying two variables, **price** and **discount**, into a third variable, **saving**.
55. Write an assignment statement that will divide the value in **total** by the value in **count** and store the result in **mean**.
56. Write an if statement that will compare the value in **price** against the value in **budget**. If **price** is greater than **budget** then print the message **Too expensive**, otherwise print the message **Just right**.
57. Modify your answer to the previous exercise so that the message if the **price** is too high includes the value of your budget.

Local variables

Consider the method **refundBalance** which is new in the better **TicketMachine**

```
int refundBalance() {
    int amountToRefund;
    amountToRefund = balance;
    balance = 0;
    return amountToRefund;
}
```

amountToRefund is a *local variable* of the method. It is declared within the scope of the method and its lifetime is the scope of the method - once the method is finished executing, its local variables no longer exist. The variable could have been declared and initialized in one expression

```
int amountToRefund = balance;
```

Exercises

58. Why does the following version of **refundBalance** not give the same results as the above?

```
int refundBalance() {
    balance = 0;
    return amountToRefund;
}
```

What tests can you run to demonstrate that it does not?

59. What happens if you try to compile the **TicketMachine** class with the following version:

```
int refundBalance() {
    return amountToRefund;
    balance = 0;
}
```

What do you know about return statements that helps to explain why this version does not compile?

60. What is wrong with this constructor?

```
TicketMachine(int cost) {
    int price = cost;
    balance = 0;
    total = 0;
}
```

A local variable of the same name as an instance variable (field) will prevent the instance variable from being accessed from within the method. We will see how to get around this later.

Fields, parameters, and local variables

	field	parameter	local variable
AKA*	instance variable	argument	
declared in	class, outside methods	method header	inside method
persistence	lifetime of object	lifetime of method	lifetime of method
scope	whole class	method	method

* Also known as

Exercises

61. Add a new method, **emptyMachine,** that is designed to simulate emptying the machine of money. It should both return the value in **total** and reset **total** to be zero.
62. Rewrite the **printTicket** method so that it declares a local variable, **amountLeftToPay**. This should then be initialized to contain the difference between **price** and **balance**. Rewrite the test in

the conditional statement to check the value of **amountLeftToPay**. If its value is less than or equal to zero, a ticket should be printed, otherwise an error message should be printed stating the amount still required. Test your version to ensure that it behaves in exactly the same way as the original version.

63. What would you need to change on the better **TicketMachine** if you wanted to have a single ticket machine issue tickets where the price of the ticket is entered by the user before inserting his money? Copy the *betterticketmachine* package and paste it as a new package *multipriceticketmachine* and make the necessary changes to the new package code.

Summary of the better ticket machine

It's, well, uh...better.

Exercise

IV. Why?

V. Did the better ticket machine need getter / setter methods?

Self-review exercises

Exercises

64. List the name and return type of this method:

```
String getCode() {
    return code;
}
```

65. List the name of this method and the name and type of its parameter:

```
void setCredits(int creditValue) {
    credits = creditValue;
}
```

66. Write out the outer wrapping of a class called **Person**. Remember to include the curly brackets that mark the start and end of the class body, but otherwise leave the body empty.

67. Write out definitions for the following fields:

    - a field called **name** of type **String**
    - a field of type **int** called **age**
    - a field of type **String** called **code**
    - a field called **credits** of type **int**

68. Write out a constructor for a class called **Module**. The constructor should take a single parameter of type **String** called **moduleCode**. The body of the constructor should assign the value of its parameter to a field called **code**. You don't have to include the definition for **code**, just the text of the constructor.

69. Write out a constructor for a class called **Person**. The constructor should take two parameters. The first is of type **String** and is called **myName**. The second is of type **int** and is called **myAge**. The first parameter should be used to set the value of a field called **name**, and the second should set a field called **age**. You don't have to include the definitions for the fields, just the text of the constructor.

70. Correct the error in this method:

```
public void getAge() {
    return age;
}
```

71. Write an accessor method called **`getName`** that returns the value of a field called **`name`**, whose type is **`String`**.
72. Write a mutator method called **`setAge`** that takes a single parameter of type **`int`** and sets the value of a field called **`age`**.
73. Write a method called **`printDetails`** for a class that has a field of type **`String`** called name. The **`printDetails`** method should print out a string of the form "The name of this person is", followed by the value of the name field. For instance, if the value of the **`name`** field is **`"Helen"`**, then **`printDetails`** would print:

    The name of this person is Helen

Reviewing a familiar example

Back to school - look at the **`Student`** class code from the *labclasses* package.

```
package labclasses;

/**
* The Student class represents a student in a student administration
 * system. It holds the student details relevant in our context.
 *
 * @author Michael Kölling and David Barnes
 * @version 2016.02.29
 */
class Student {
    // the student's full name
    private String name;
    // the student ID
    private final String id;
    // the amount of credits for study taken so far
    private int credits;

    /**
     * Create a new student with a given name and ID number.
     */
    Student(String fullName, String studentID) {
        name = fullName;
        id = studentID;
        credits = 0;
    }

    /**
     * Return the full name of this student.
     */
    String getName() {
        return name;
    }

    /**
     * Set a new name for this student.
     */
    void changeName(String replacementName) {
```

```
        name = replacementName;
    }

    /**
     * Return the student ID of this student.
     */
    String getStudentID() {
        return id;
    }

    /**
     * Add some credit points to the student's accumulated credits.
     */
    void addCredits(int additionalPoints) {
        credits += additionalPoints;
    }

    /**
     * Return the number of credit points this student has accumulated.
     */
    int getCredits() {
        return credits;
    }

    /**
     * Return the login name of this student. The login name is a
     * combination of the first four characters of the student's name and
     * the first three characters of the student's ID number.
     */
    String getLoginName() {
        return name.substring(0,4) + id.substring(0,3);
    }

    /**
     * Print the student's name and ID number to the output terminal.
     */
    void print() {
        System.out.println(name + ", student ID: "
                + id + ", credits: " + credits);
    }
}
```

In this small example, the pieces of information we wish to store for a student are their name, their student ID, and the number of course credits they have obtained so far. All of this information is persistent during their time as a student, even if some of it changes during that time (the number of credits). We want to store this information in fields, therefore, to represent each student's state.

The class contains three fields: **`name`**, **`id`**, and **`credits`**. Each of these is initialized in the single constructor. The initial values of the first two are set from parameter values passed into the constructor. Each of the fields has an associated **`get`** accessor method, but only **`name`** and **`credits`** have associated mutator methods. This means that the value of an id field remains fixed once the object has been constructed. If a field's value cannot be changed once initialized, we say that it is *immutable*. Sometimes we make the complete state of an object immutable once it has been constructed; the **`String`** class is an important example of this.

Exercises

74. Draw a picture representing the initial state of a **`Student`** object following its construction with the following actual parameter values:
    ```
    new Student("Benjamin Jonson", "738321")
    ```
75. What would be returned by **`getLoginName`** for a student with the name "Henry Moore" and the id "557214"?
76. Create a **`Student`** with name "djb" and id "859012". What happens when **`getLoginName`** is called on this student? Why do you think this is?
77. The **`String`** class defines a length accessor method with the following signature:
    ```
    /**
    * Return the number of characters in this string.
    */
    public int length()
    ```
    Add conditional statements to the constructor of **`Student`** to print an error message if either the length of the **`fullName`** parameter is less than four characters or the length of the **`studentId`** parameter is less than three characters. However, the constructor should still use those parameters to set the **`name`** and **`id`** fields, even if the error message is printed. Hint: use if statements of the following form (that is, having no else part) to print the error messages.
    ```
    if (perform a test on one of the parameters) {
        Print an error message if the test gave a true result.
    }
    ```
78. Modify the **`getLoginName`** method of **`Student`** so that it always generates a login name, even if either of the **`name`** and **`id`** fields is not strictly long enough. For strings shorter than the required length, use the whole string.

Both **`name`** and **`id`** were declared of type **`String`**. To find out what **`name.substring(0,4)`** does, we need to look up the documentation for **`String`**'s **`substring`** method.
It's a Very Good IdeaTM to start finding your way around the official documentation for Java, available at http://docs.oracle.com/javase/8/docs/api/.

Summary

Concept summary

object creation

Some object creation requires arguments providing extra information.

field

Fields store data for an object to use. Fields are also known as *instance variables*.

comment

Comments are inserted into the source code of a class to provide explanations to human readers. They have no effect on the functionality of the class.

constructor

Constructors allow each object to be set up properly when it is first created.

scope

The scope of a variable defines the section of source code from where the variable can be accessed.

lifetime

The lifetime of a variable describes how long the variable continues to exist before it is destroyed.

assignment

Assignment statements store the value represented by the right-hand side of the statement in the variable named on the left.

method

Methods consist of two parts: a *header* and a *body*.

accessor method

Accessor methods return information about the state of an object. But be careful, they can reveal *too much* about an object, thus breaking encapsulation.

mutator method

Mutator methods change the state of an object. But be careful, they can reveal too much about an object, thus breaking encapsulation.

println

The method **`System.out.println(`**…**`)`** prints its parameter to the text terminal.

conditional

A conditional statement takes one of two possible actions based upon the result of a test.

boolean expression

Boolean expressions have only two possible values: **`true`** and **`false`**. They are commonly found controlling the choice between the two paths through a conditional statement.

local variable

A local variable is a variable declared and used within a single method. Its scope and lifetime are limited to that of the method.

Additional exercises

Exercises

83. Below is the outline for a **`Book`** class, which can be found in the *bookexercise* package. The outline already defines two fields and a constructor to initialize the fields. In this exercise and the next few, you will add further features to the class outline. Add two accessor methods to the class – **`getAuthor`** and **`getTitle`** – that return the **`author`** and **`title`** fields as their respective results. Test your class by creating some instances and calling the methods.

```
/**
 * A class that maintains information on a book.
 * This might form part of a larger application such
```

```
 * as a library system, for instance.
 *
 * @author Insert your name here.
 */
class Book {
    // The fields.
    private String author;
    private String title;
    /**
     * Set the author and title fields when this object
     * is constructed.
     */
    Book(String bookAuthor, String bookTitle) {
        author = bookAuthor;
        title = bookTitle;
    }
    Add the methods here.
}
```

84. Add two methods, **printAuthor** and **printTitle**, to the outline **Book** class. These should print the **author** and **title** fields, respectively, to the terminal window.
85. Add a further field, **pages**, to the **Book** class to store the number of pages. This should be of type **int**, and its initial value should be passed to the single constructor, along with the **author** and **title** strings. Include an appropriate **getPages** accessor method for this field.
86. Are **Book** objects immutable? Justify your answer.
87. Add a method, **toString**, to the **Book** class. This should return details of the **author**, **title** and **pages**.
    Add a method, **printDetails**, to the **Book** class, calling **toString** and printing details of **author**, **title** and **pages** to the terminal window. It is your choice how the details are formatted. For instance, all three items could be printed on a single line, or each could be printed on a separate line. You might also choose to include some explanatory text to help a user work out which is the author and which is the title, for example
    Title: Robinson Crusoe, Author: Daniel Defoe, Pages: 232
88. Add a further field, **refNumber**, to the **Book** class. This field can store a reference number for a library, for example. It should be of type **String** and initialized to the zero length string (**""**) in the constructor as its initial value is not passed in a parameter to the constructor. Instead, define a mutator for it with the following signature:
    void setRefNumber(String ref)
    The body of this method should assign the value of the parameter to the **refNumber** field. Add a corresponding **getRefNumber** accessor to help you check that the mutator works correctly.
89. Modify your **toString** method to include returning the reference number. However, the method should return the reference number only if it has been set – that is, the **refNumber** string has a non-zero length. Hint: Use a conditional statement whose test calls the **length** method on the **refNumber** string.
90. Modify your **setRefNumber** mutator so that it sets the **refNumber** field only if the parameter is a string of at least three characters. If it is less than three, then print an error message and leave the field unchanged.
91. Add a further integer field, **borrowed**, to the **Book** class. This keeps a count of the number of times a book has been borrowed. Add a mutator, **borrow**, to the class. This should update the field by 1 each time it is called. Include an accessor, **getBorrowed**, that returns the value of this new field as its result. Modify **toString** so that it includes the value of this field with an explanatory piece of text.
92. Add a boolean attribute **courseText** to record whether the book is used in a course. The value is set

through an argument to the constructor, and the attribute is immutable (can't be changed). Provide an accessor method called **`isCourseText`**.

93. Create a new package, *heaterexercise*. Create a class, **`Heater`**, that contains a single integer field, **`temperature`**. Define a constructor that takes no parameters. The **`temperature`** field should be set to the value 15 in the constructor. Define the mutators **`warmer`** and **`cooler`**, whose effect is to increase or decrease the value of temperature by 5° respectively. Define an accessor method to return the value of **`temperature`**.
94. Modify your **`Heater`** class to define three new integer fields: **`min`**, **`max`** and **`increment.`** The values of **`min`** and **`max`** should be set by parameters passed to the constructor. The value of **`increment`** should be set to 5 in the constructor. Modify the definitions of **`warmer`** and **`cooler`** so that they use the value of **`increment`** rather than an explicit value of 5. Before proceeding further with this exercise, check that everything works as before. Now modify the **`warmer`** method so that it will not allow **`temperature`** to be set to a value greater than **`max`**. Similarly modify **`cooler`** so that it will not allow **`temperature`** to be set to a value less than **`min`**. Check that the class works properly. Now add a method, **`setIncrement`**, that takes a single integer parameter and uses it to set the value of **`increment`**. Once again, test that the class works as you would expect it to by creating some **`Heater`** objects. Do things still work as expected if a negative value is passed to the **`setIncrement`** method? Add a check to this method to prevent a negative value from being assigned to **`increment`**.