OLD-The bank account example

Description

Text from *OO-lec-v6-figures-ok*, which is in the DropBox *Papers* folder.

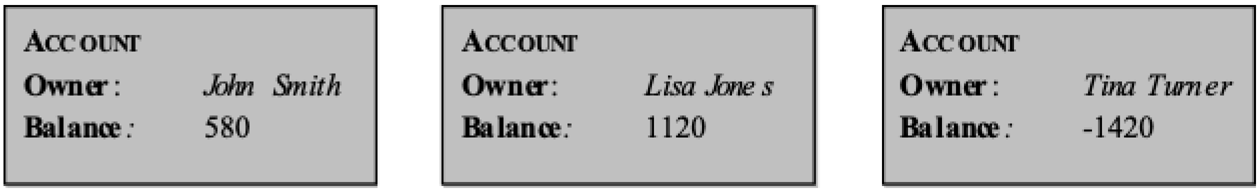
In this Section, we will try to envision how a computer may be used to support a simple administrative system from the real world using objects. As an example, we will use a bank.

1 Before computers

We will first describe how a bank might be organized before computers were invented. The bank has a number of customers. A customer may borrow money from the bank or deposit money in the bank. The bank keeps information about each customer in the form of an account that keeps track of the current balance for the customer. A bank clerk will have to be able to perform the following tasks:

- When a customer enters the bank, the clerk must be able to find the account of the customer.
- For a new customer, the clerk must be able to create a new account.
- The clerk must be able to receive money from the customer for deposit on his/her account.
- The clerk must be able to give money to the customer by withdrawing money from his/her account.
- The customer should also be able to loan money to the customer.

For each customer the bank keeps an account in the form of a file that consists of one or more pieces of paper. A piece of paper may have the format as shown in Figure 2.1, which shows three accounts.



ACCOUNT
Owner: *John Smith*
Balance: 580

ACCOUNT
Owner: *Lisa Jones*
Balance: 1120

ACCOUNT
Owner: *Tina Turner*
Balance: -1420

Figure 2.1 Example of paper-based accounts

The information in an account is the name of the owner of the account and the current balance of the account. One of the accounts is owned by *John Smith* and has a current balance of 580.

In this very simple example, we only have one type of account. If the balance on the account is positive, we assume that the customer has put money in the bank. If the balance is negative, we assume that the customer has borrowed money from the bank. As can be seen, *Tina Turner* has borrowed an amount of 1420.

The bank has an account for each customer, and all the accounts are kept in a cabin where there is a folder per account.

When a customer visits the bank and asks for service, a bank clerk finds his/her account in the cabin and performs the desired transaction (deposit or withdrawal) by changing the balance field.

If a new customer wants to use the bank, the clerk will have to create a new account. For this purpose, the bank has preprinted paper where the clerk may fill in the details of the customer, such as name of the owner, and the current balance.

We have of course made a number of simplifications compared to a real bank. In a real bank, there are many different types of accounts, such as saving accounts and check accounts. In addition, a record is kept for each transaction that has been made on a given account.

2 Bank with IT-support

Suppose the bank wants to handle its accounts using a computer. In order to do this, we must be able to represent the accounts in the computer. In addition, we must be able to view an account on a screen and modify the account using keyboard and mouse.

To represent an account in a computer, we use an *object*. An object is an entity that exists inside a computer – it may be used to keep all the information we need for an account. In addition, we may change the information stored in an object when transactions are performed on the account.

To be able to view and modify an account we must be able to display it on a screen and to enter information about transactions (deposits and withdrawals). To do this we must design and implement a user interface for a bank-system.

The paper-based account has the advantage that it is immediately readable and modifiable by the bank-clerk and when a transaction is performed, it is easy to update the balance field. For a computer-based system, we will have to split the paper-based account into two parts: an object, which represents the information on the account and a user interface for presentation and modification by the bank clerk. The user-interface may e.g. be a window on the screen. This is illustrated by in Figure 3.1.

Figure 3.1

2.1 Creating objects representing accounts

We will start by showing how to create objects that can represent accounts. In order to this, we will have to define a template for creating objects representing accounts. A template describes the structure of an object in terms of the elements of the object. The programming language construct for defining such an object-template is called a ***class***. An object created from a class consists of ***data-items*** and ***methods***. A class for defining accounts may be defined as shown in Figure 3.2.

Figure 3.2

As mentioned, a class like `Account` defines a template for creating `Account`-objects. An `Account`-object has two data-items: `owner` and `balance`. The data-item `owner` is supposed to represent the name of the owner of the account. The data-item `balance` is supposed to represent the current balance of the account. During execution of the program, the value of `balance` may change when money is deposited or withdrawed from the account. The value of the `owner` may change if the account gets a new owner or the owner changes his name.

An Account-object also has three methods: setOwner, deposit, and withdraw. A method contains code that when executed may return data about the Account-object and/or modify the data-items of the Account-object. The method setOwner is supposed to represent and execute the action performed by a bank clerk when he/she needs to set the name of the owner. Similarly the methods deposit and withdraw are supposed to represent and execute the actions performed by the client when money is deposited or withdrawed respectively from the account.

In the following, we will describe in further detail the meaning of the constructs, class, data-item and method.

1.1 The class construct

As mentioned, a *class* is a template for creating objects. An object consists of the following elements:

- A set of *data-items* – also called *instance variables*, or just *variables*. A data-item is a container that may hold values. A value may be an integer value, like 5, 123, or –99.

- A set of *methods* – also called *member functions*. A method contains *code* that may be executed.

As mentioned, a class is a template for creating objects. A class definition contains a description of the data-items and methods that are present in the objects created from the class. A class declaration may have the following form:

```
class Account extends Object
{
   “declaration of data-items and methods
}
```

The important elements of this declaration are:

- The keyword **class** signals that a class is defined
- The name **Account** following class defines that the name of the class is Account
- The phrase **extends Object** will be ignored for the moment
- The braces { and } encapsulates the body of the class. The body contains declarations of data-items and methods.

1.1.1 Data-items

The following declaration is an example of a declaration of a of data-item:

```
int balance = 0;
```

This declaration states that objects created from the class will contain a data-item with the following properties:

- The name of the data-item is balance.
- The type of the data-item is int. The type of a data-item describes which values that can be stored in the data-item.
- The initial value of the data-item is zero (0).

The following declaration is also an example of a declaration of a data-item:

```
String owner;
```

Objects created from the class will have a data-item with the following properties:

- The name of the data-item is owner.
- The type of the data-item is string. The type String contains values that are sequences of characters. Examples of characters are “a”, “2”, “*”, etc. Notice that characters are written enclosed in quotes (“). Also note, that there is a difference between the number 2 and the character “2”. As mentioned, a string is a sequence of characters. Examples of string-values are: “John Smith”, and “LisaNelson@imv.au.dk”.
- No initial value is defined for owner.

A data-item is often called an *instance-variable* or just *variable*.

1.1.2 Methods

The following declaration is an example of a method declaration:

```
void setOwner(String N)
{ owner = N; }
```

Objects created by from the class will have a method that has the following properties:

- The ***name*** of the method is `setOwner`.
- The method has one ***argument*** (String N). An argument is similar to a data-item in the sense that it holds a value. The value of the argument is defined when the method is activated (see below). The value of the argument may be used in the body (see below) of the method.
- A method may compute a value that is returned to the caller (see below) of the method. The setOwner-method returns no value to the caller – indicated by the return type void. In a later section, we will give examples of methods

that return a value.
- The ***body*** of the method is { owner = N; }. In general, the body of a method may contain declarations of data-items and statements.
- A ***statement*** defines an action that when executed may have an effect on one or more data-items. The body of setOwner consist of one statement

The effect of executing this statement is that the data-item owner will be given the value of the data-item (argument) N of setOwner. The variable owner is said to be ***assigned*** the value of N. A statement of this form is called an assignment-statement. Below we will explain assignment-statements in more details and introduce a number of other kinds of statements.

```
owner = N;
```

A method may be invoked (or called) from another method. When a method is invoked, its arguments are given values, the code in the body of the method is executed and a possible return value is computed and returned to the caller of the method. We will explain this in further detail below.

1.2 Using class Account

The class Account may be used in other methods to create and manipulate Account-objects. In the following, we will show examples of using class Account.

1.2.1 Creating Account-objects

A class is a template for creating new objects. Execution of an expression of the form

```
new Account()
```

will result in the creation of a new Account-object.

1.2.2 Account-variables

The data-items balance and owner declared in class Account may hold values of type int or String respectively. A data-item may also hold a value that is a ***reference*** to an object. A declaration of the form:

```
Account JS;
```

declares JS to be a data-item that may refer to Account-objects. The data-item JS may be assigned a value by a statement like

```
JS = new Account()
```

The expression new Account() generates a new Account-object from the template defined by class Account. A reference to this new object is then assigned to the data-item JS.

Figure 3.4

A reference is a value that refers to an object: it uniquely identifies the object and it can be used to access the object by invoking its methods, reading the values of its data-items and/or assigning new values to its data-items. The term pointer is often used interchangeably with reference. In diagrams, a reference is often illustrated by an arrow for the data-item to the object. In Figure 3.4 is showed an object X with a data-item JS declared as Account JS, referring an Account-object.

1.1.1 Method invocation

We will now introduce a new form of statement, a ***method invocation***. A method invocation invokes a method on an object. The following statement is an example of a method invocation:

```
        JS.setOwner(“John Smith”);
```

The effect of this statement is that the method setOwner is invoked on the object JS. The argument N of setOwner is given the value “John Smith”. When setOwner is executed, this will then have the effect that the owner data-item of JS will be assigned the value “John Smith”.

Figure 3.5

The details of a method-call are shown in the diagram of Figure 3.5. The diagram consist of three rows marked 1, 2, and 3, and two columns marked a and b. The rows 1, 2, and 3 mark different points of time. The columns a and b shows two items that are of interest when illustrating the effect of the method call.

The item in column (b) is an Account-object labeled JS. The item in column (a) is a so-called ***method-activation*** (or just ***activation***) corresponding to the call of setOwner. An activation is similar to an object in the sense that it consists of data-items, including arguments. The activation of setOwner contains a data-item N, which is given the value “John Smith”. In addition, the activation contains a data-item called this. The data-item this is a reference to the object on which the method is invoked. Since setOwner is invoked on the Account-object JS, the data-item this is assigned a reference to the object JS. This is illustrated by the arrow from this to JS.

We may now explain the Figure 3.5in detail:

1. At time 1, the only item in existence is the Account-object JS in column (b) exists. The value of owner is “John Dowe” and the value of balance is 180.
2. At time 2, the method-call `JS.setOwner(“John Smith”)` is executed – at this point we do not care about who and where this method call is executed.
   To handle the method call, the method-activation in column (a) is generated with the argument N assigned the value “John Smith” and this assigned a reference to the object JS.
3. At time 3, the statement owner = N in the body of setOwner is executed. We mark this by an arrow from the lifeline of the setOwner-activation to the object JS.
   The result of this action is that the variable owner in JS has been assigned the value “John Smith”. This illustrated by redrawing the object JS in column (b). As can be seen owner has been changed but balance remains unchanged.

1.1.1 Statement sequence

Until now, we have seen two kinds of statements: assignment statements and method invocations. It is possible combine a sequence of statements and form a *statement sequence*. The following is an example of statement sequence:

```
    JS = new Account();
    JS.setOwner(“John Smith”);
    JS.deposit(200);
    JS.withdraw(75);
```

When a statement sequence is executed, the individual statements are executed in the order they appear in the sequence. We will elaborate on this below.

1.2 Putting it all together

A method consists of a number of declarations and a sequence of statements (statement sequence). The method main below includes a declaration of an Account-variable, generates a new Account-object and performs three method invocations on the Account-object:

```
static void main(String args[])
{ Account JS;
  JS = new Account();
  JS.setOwner("John Smith");
  JS.deposit(200);
  JS.withdraw(75);
}
```

The main-method differs from the previously defined methods setOwner, deposit and withdraw in two ways:

- The keyword static declares that the method is *class method*. A non-class method belongs to an object and can only be executed by invoking it on an object. A class method exists independently of any object and may be thought of as belonging to a class. We can thus execute main without any object.
- The name main is special. Java program consists of one or more classes. One of these classes must define a class method called main. When the program is executed, execution starts by invoking the main-method.
- The method main has an argument args of type array of String. We explain the details of args below.

We will later elaborate on the concept of class methods, including the method main.

In Figure 3.6, we have extended class Account from Figure 3.2 with the above main-method.

```
class Account extends Object {
   String owner;
   int balance = 0;

   void setOwner(String N)
   { owner = N; };

   void deposit(int amount)
   { balance += amount; };

   void withdraw(int amount)
   { balance -= amount;}

   static void main(String args[])
   { Account JS;
     JS = new Account();
     JS.setOwner("John Smith");
     JS.deposit(200);
     JS.withdraw(75);
   }
}
```

The class Account in Figure 3.6 may be compiled and executed. During execution the following happens:

1. Execution starts be invocation of the method main.
2. Execution of main consists of execution of the following statements:
3. A new Account-object is created and assigned to JS.
4. The method setOwner with argument "John Smith" is invoked on the object JS.
5. The method deposit is invoked with argument 200 on JS.
6. The method withdraw is invoked with argument 75 on JS. At this point the object JS has the state shown in Figure 3.3.

In the next section, we will explain the execution of Account in further details.

1.1 Sequence diagrams

To illustrate the effect of actions executed by a Java program, we will use a diagrammatic notation called sequence-diagrams. A sequence may be used to show the life cycle of one or more *items* during the execution of part of a program. In Figure 3.5 above, we have already seen an example of a sequence diagram.

For most programs, a large number of items are generated execution of the program. The most common examples of items are objects, method-invocations, classes and methods. In most cases, we will only show objects and method-activations on the diagrams. The life cycle of the Account-object may be illustrated by the sequence diagram in Figure 3.7.

Figure 3.7

In a sequence diagram, an item is shown as a box at the top of a vertical line. The diagram in Figure 3.7 has six items:

1. The Java-item in column (a) represents the Java-system of the computer.
2. The item in column (b) represents an activation of the main-method.
3. Column (c) shows three items representing activations of setOwner, deposit, and withdraw.
4. Column (d) represents an Account-object labeled JS.

The vertical line below a box is called the item's lifeline. The lifeline represents the life of the item during program execution.

A box shows the state of the item at a given point in time during execution of the program. The *state* of an item represents the information held in the item at the given point in time. For an object or method-activation, the values of the data-items are included in the state.

An item may be shown several times along its lifeline. This is done to show how the state of the item may change during the life cycle of the item. Even if the Account-object JS is drawn four times in column (d) in Figure 3.7, there is only one Account-object in this example. A box showing the state of the Account-object is often called a *snapshot* of the state of the object.

An arrow between the lifelines of two items represents an *action* invoked by one of the items and executed by the other item. The item at the start of the arrow is the *invoker* and the item at the end of the item is the *receiver*.

An item may be shown several times on its lifeline to illustrate that the values of its instance variables may have changed as the effect of a method-call.

In Figure 3.8 is shown the elements of a sequence diagram:

- An object and its lifeline.
- Two action arrows.
- An activation box.

Figure 3.8

An action like new Account() results in the creation of a new Account-object. To illustrate that a new Account-object is created, the arrow points to the box of the object being created. This is shown in part (a) of Figure 3.9. In this way, the diagram illustrates that the life cycle of a new Account-objects starts by the execution of new Account().

Figure 3.9

In part (b) of Figure 3.9, part of the life cycle of the Account-object is shown. Initially the variables name and balance of the Account-object has the values undefined and 0 respectively. The action owner="John Smith" is then executed. During execution of this action, the Account-object is active as shown by the activation box below the arrow. After execution of owner="John Smith", the variable owner has the value "John Smith".

We may now explain the details of Figure 3.7 – The numbers in the list below refers to the numbers of the rows in the figure:

1. At the start of the scenario, only the Java-item representing the Java-system on the computer exists.
2. The user then executes the Java-program illustrated by the arrow labeled Account.java. This implies that the Java-system invokes the main-method resulting in creation of the main-activation shown in column (b).

3. The action JS=new Account() is then invoked by the main-activation. The result is creation of the Account-object JS in column (d). For the Account-object, the value of owner is undefined and the value of balance is zero (0). In addition, the data-item JS in main-activation is assigned a reference to the new Account-object. This is illustrated by the new snapshot of main following row 3.
4. The main-activation then executes the method invocation JS.SetOwner("John Smith"). This implies that an activation of setOwner is created as shown in column (c). The argument N has assigned the value "John Smith" and the this-reference refers to the Account-object. The setOwner-activation then becomes active and starts executing the statements in its body.
5. The body setOwner consists of only one statement owner = N. Execution of this statement implies that the owner data-item of the Account-object is assigned the value N, which is "John Smith".
6. The main-activation then executes the method invocation JS.desposit(200). This implies that an activation of deposit is created as shown in column (c). The argument amount has been assigned the value 200 and this has been assigned a reference to the Account-object. The deposit-activation then becomes active and starts execution of the statements in its body.
7. The deposit-activation executes the statement balance += amount which implies that the value of balance in the Account-object is incremented by 200. The result is that the value of balance becomes 200.
8. Finally, main-activation executes JS.withDraw(75): An activation of withdraw is created with deposit assigned the value 75 and this assigned a reference to the Account-object.
9. The withdraw-activation executes the statement balance -= amount which implies that the value of balance in the Account-object becomes 125.
10. At this point, the execution of the program is finished – the program is said to terminate.

1 Creating several accounts

In the previous section, we have shown how to define a Java-program that defines class Account with a main-method that creates just one Account-object. On e of the advantages of a class is that it can be used for creating several objects. In this section, we will show an example that creates more than one instance of class Account. We will also show how to make a Java-program that makes use of two classes.

We will define class Bank that creates three Account-objects. For simplicity, we have assumed that the bank has just three customers. In Figure 4.1 we define a class Bank that has three Account-objects, JS, LJ, and TT and a static main-method. When main is executed, it creates the three accounts.

```
public class Bank extends Object
{ public static void main(String args[])
     { Account JS,LJ,TT; // variables to hold 3 accounts
       // create the account for John Smith and deposit 100
       JS = new Account();
       JS.setOwner("John Smith");
       JS.deposit(100);
       // create the account for Lisa Jones and deposit 200
       LJ = new Account();
       LJ.setOwner("Lisa Jones");
       LJ.deposit(200);
       // create the account for Tina Turner and deposit 50
       TT = new Account();
       TT.setOwner("Tina Turner");
       TT.deposit(50);
    }
}
```

Figure 4.1. Class Bank defining three Account-objects.

The Bank-class makes use of the Account-class defined in Figure 3.2[1].

We may now compile and execute Bank.java. When Bank.java is executed, the first thing that happens is that method main is executed and an activation for main is created.

For this example, the main-activation has data-items corresponding to args, JS, LJ and TT.

[1] At this point, we have not discussed how to organize class-definitions on a computer. Usually a class is stored in a file. For a class to use another class, there are some rules that must be followed. This is an issue we will return to later,

We may now compile and execute Bank.java. When Bank.java is executed, the first thing that happens is that method main is executed and an activation for main is created.

For this example, the main-activation has data-items corresponding to args, JS, LJ and TT.

When the main-activation has been created, the statements in the body of main are executed. Execution of theses statements will result in the creation of three Account-objects. For each Account-object, the name of the owner is set by execution of the method setOwner(). In addition, some money is deposited on each account by execution of the method deposit().

Just after the final statement (TT.deposit(50)) in main has been executed, the state of execution may be illustrated by Figure 4.3

Figure 4.3

In the following, we will describe step-by-step what happens during execution of Bank.java. The state of execution is illustrated by the diagram shown in Figure 4.4. The labels in the leftmost column of Figure 4.4, refers to the items in the numbered list below:

1. The user executes Bank.main().
2. The method main defined in class Bank is executed. This happens in two steps:
   - A method-activation for main is created. This main-activation has data-members corresponding to args, JM, LJ and TT.
   - The statements defined in the body of main() are executed. The execution thus continues along the lifeline for the main-activation.
3. The main-activation executes JS = new Account();
   The expression new Account() implies that a new object is created from the template defined by class Account. The new Account-object is shown in column d. The instance variable JM in the Bank-object is assigned a reference to the new Account-object. That is JM refers to the new Account-object. This is illustrated by the new drawing of main-activation after row 3.
4. The main-activation executes `JM.setOwner(“John Smith”)`.
   The method setOwner is executed (invoked) on the object JM with argument “John Smith”. This implies that the body of the method setOwner defined in class Account (line xxx) is executed. The argument N of setOwner has the value “John Smith”.
   Execution of the statement name = N; implies that the instance variable name of the Account is assigned the value of N. I.e. name is assigned the value “John Smith”. This is illustrated by the new drawing of the Account-object in column c.The main-activation executes JM.deposit(100).
5. The method deposit is executed on the object JM with argument 100. This implies that the body of deposit is executed. The argument amount of deposit has the value 100.
6. Execution of balance += amount; implies that the instance variable balance is incremented with the value of amount. Since balance was 0 (zero) before the statement was executed, the value of balance is 100 after the statement has been executed. This is illustrated by the new drawing of the Account-object in column d.
7. LJ = new Account();
   An instance (object) of class Account is generated and a reference to the new Account is assigned to the variable LJ.
8. LJ.setOwner(“Lisa Jones”);
   The method setOwner of LJ is executed with the argument “Lisa Jones”. The result is that the instance variable name of LJ is assigned the value “Lisa Jones”.
9. LJ.deposit(200);
   The method deposit of LJ is executed with the argument 200. The result is that the instance variable balance of LJ gets the value 200.
10. TT = new Account();

A new Account-object is generated and assigned to TT.

11. TT.setOwner(“Tina Turner”);
    The instance variable TT.name gets the value “Tina Turner”.
12. TT.deposit(50);
    The instance variable TT.balance gets the value 50.

Figure 4.4

The Java code in class Account and class Bank is not in itself very useful. We need to extend the example to be able to do something interesting, such as performing transactions on the accounts. To do this we will also have to implement a user interface. Therefore, we have a long way to go!

Summary of concepts

Incomplete

In this section, a summary of central constructs in object-oriented programming and Java are given.

1.1 Object

An object is an entity that exists in the memory of a computer. An object may be used to store data about a given phenomenon. Data stored in an object may be changed during the execution of a program. <methods?>, …

In Java, an object consists of

- data-items or instance variables
- Methods

Modeling aspects

Examples

1.2 Class

A class is a template for creating objects

A class may be used to represent a concept

In Java:

A class definition has the following form:

public class Name extends Object

```
{
   “description of data-items and methods”
}
```

At this point the important elements of this description is

- **class** Name – which declares that we are defining class with the name Name
- **public** is a so-called modifier, which we will explain later
- **extends Object** will also be ignored for the moment

1.3 Data-item/instance variable

A *data-item* – also called *instance variable* or just *variable*, is a container that may hold a value. A data-item has a name and a type. The name is used to denote the data-item from other places in the code. The type of the data-item defines the set of possible values that the data-item may have during execution of the program.

A data-item is defined by means of a declaration, which in its simplest version has the form

```
<type> <name>;
```

An example of a declaration is

```
int balance;
```

1.3.1 Initial value of a data-item

A data-item declaration may also define an initial value for the data-item. In this case, the declaration has the form:

```
<type> <name> = <expression>;
```

An example of a declaration with an initial value is:

```
int balance = 0;
```

1.3.2 Modifiers

A data-item declaration may contain a modifier (or several)

```
<access> <type> <name> = <expression>;
```

1.4 Simple types

int

double

- The type double contains so-called floating-point values, which are comma-values. Examples of such values are 3.14, -113.123, 0.0, 999.11, etc.

char

1.5 Reference values

1.6 Method

A method is a template for creating method-activations

1.7 Method-activation

1.8 Constructor

…

2 Sequence diagrams

The diagrammatic notation used in Figure 6 illustrates how execution progresses in a Java-program.

1. A vertical column represents an entity such as object, class or method-activation.
2. An entity is shown as a box. The top of a column, is always a box representing the entity
3. The thin vertical line below the entity is called the object's lifeline. The lifeline represents the life of the object during the scenario described by the diagram.
4. A thick line represents that the entity is active by executing actions in that period.
5. A horizontal arrow between the lifelines of two entities represents a method invocation. A method arrow is labeled with the name of the method and sometimes with the full statement.
6. Entities at the top of the diagram are in existing when the scenario starts. During execution of the scenario, new entities (objects or method-activations) may be created.
7. Creation of a new object is shown by a method-arrow with the label new.
8. Creation of a method-activation is shown with a method-arrow labeled by the name of the method being invoked
9. A given entity may be drawn several times along its lifeline. This is done when the state of the entity is changed during execution.

Extra stuff:

If the arrow does not begin at a lifeline, then the action is invoked by some item not shown on the diagram.

The two items are also shown in **Error! Reference source not found.** (a) and (b).

Execution of an action new Account() may be illustrated by the diagram in **Error! Reference source not found.**.

The last item in **Error! Reference source not found.** is the box shown in **Error! Reference source not found.**d. Such a box is called an activation-box and shows when an item is active executing actions.