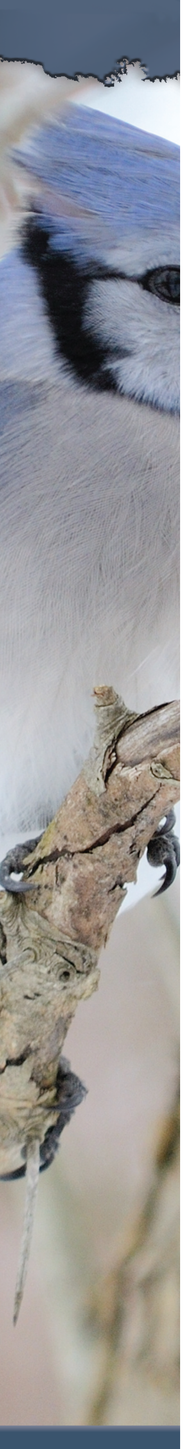
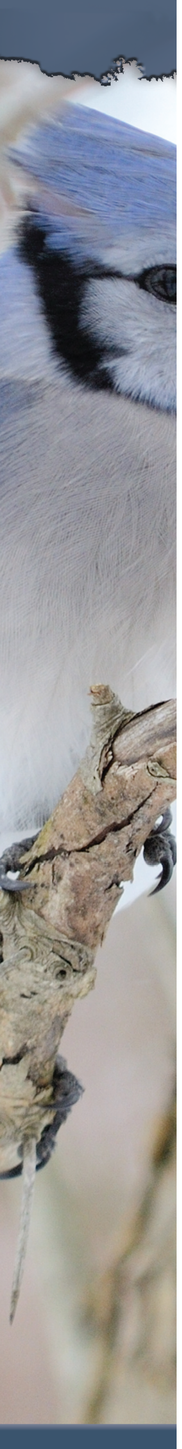


KING'S College LONDON

Programming Practice and Applications

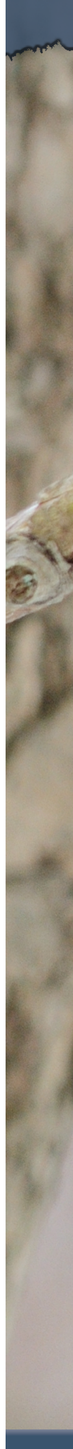
Understanding class definitions

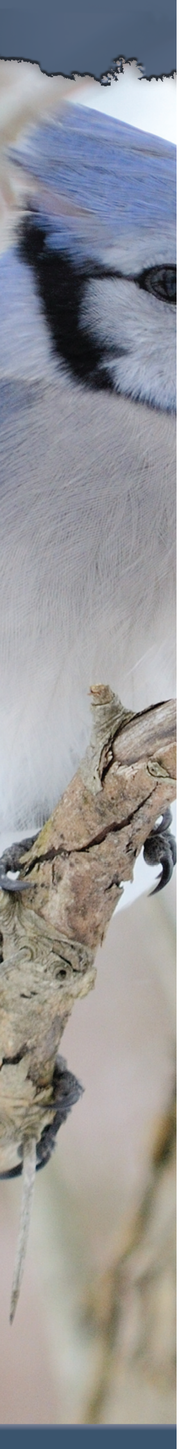
Michael Kölling



Main concepts to be covered

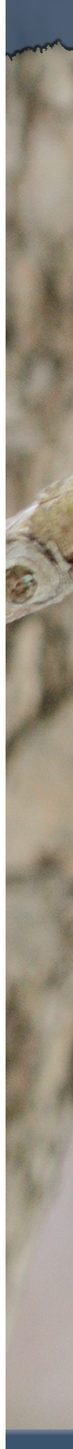
- fields
- constructors
- methods
- parameters
- assignment statements



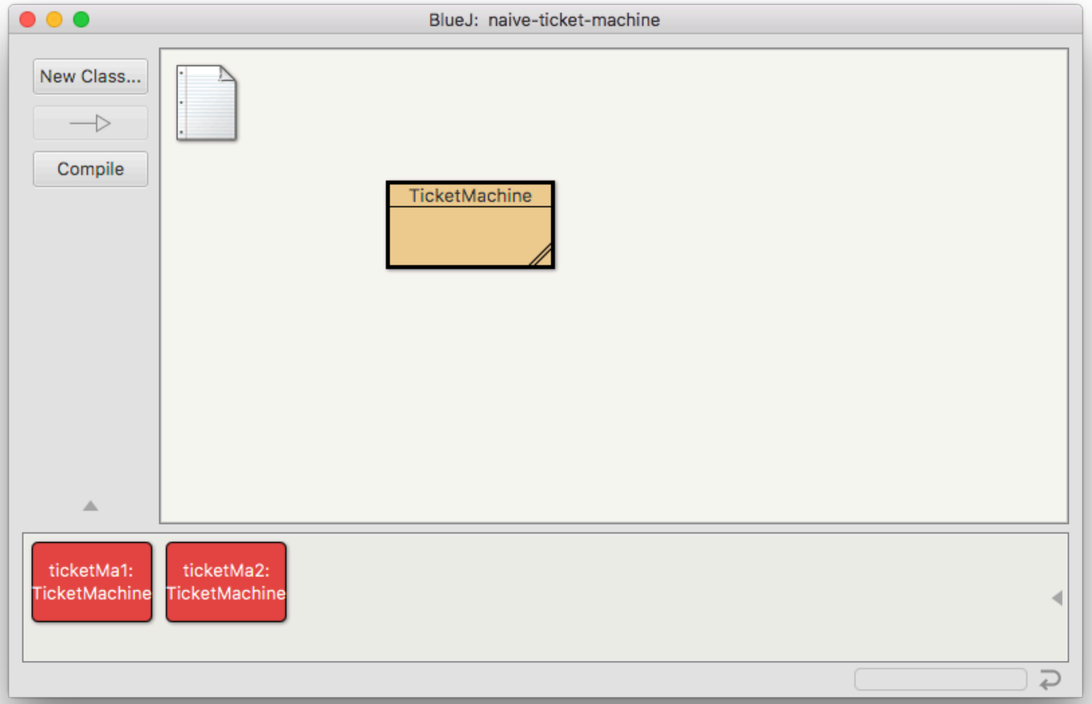


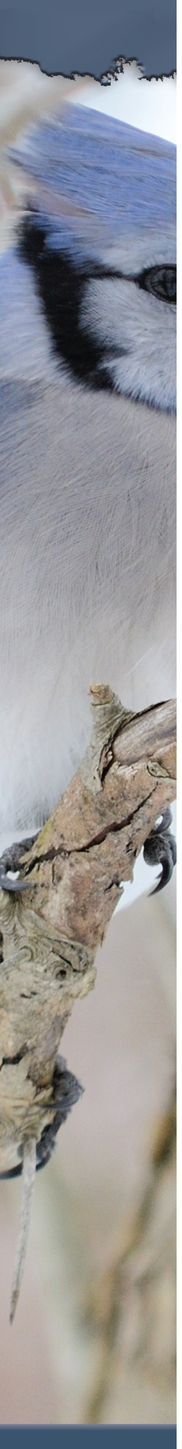
Ticket machines – an external view

- Exploring the behaviour of a typical ticket machine.
  - Use the *naive-ticket-machine* project.
  - Machines supply tickets of a fixed price.
    - How is that price determined?
  - How is ‘money’ entered into a machine?
  - How does a machine keep track of the money that is entered?



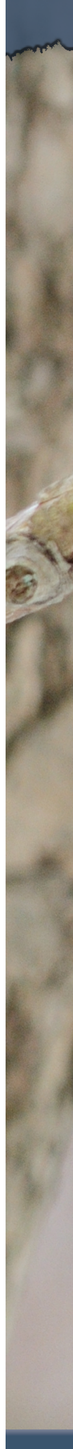
Ticket machines





Ticket machines – an internal view

- Interacting with an object gives us clues about its behaviour.
- Looking inside allows us to determine how that behaviour is provided or implemented.
- All Java classes have a similar-looking internal view.



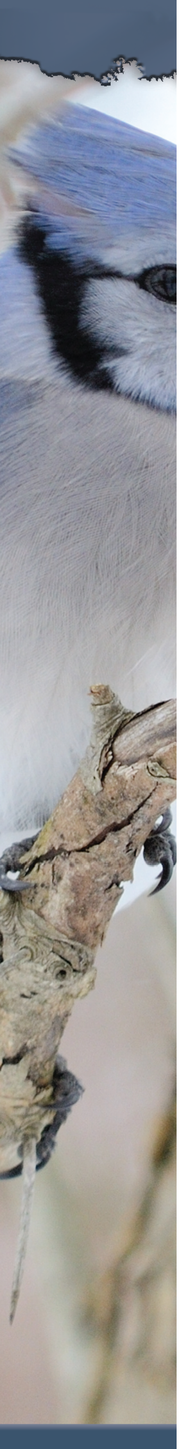
Basic class structure

```
public class TicketMachine
{
    Inner part omitted.
}
```

The outer wrapper of TicketMachine

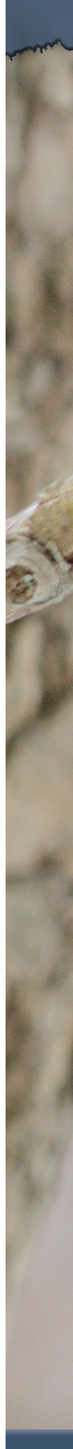
```
public class ClassName
{
    Fields
    Constructors
    Methods
}
```

The inner contents of a class



Keywords

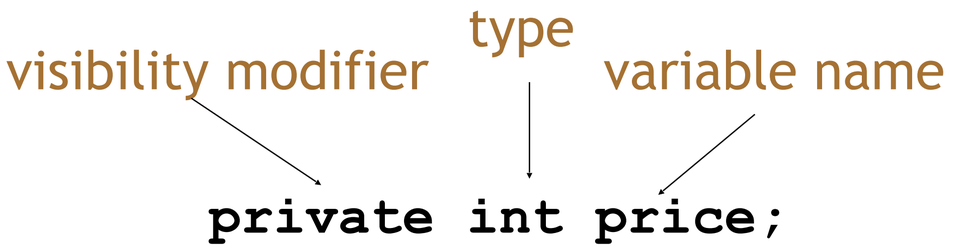
- Words with a special meaning in the language:
  - `public`
  - `class`
  - `private`
  - `int`
- Also known as *reserved words*.
- Always entirely lower-case.



Fields

- Fields store values for an object.
- They are also known as *instance variables*.
- Fields define the state of an object.
- Use *Inspect* to view the state.
- Some values change often.
- Some change rarely (or not at all).

```
public class TicketMachine
{
    private int price;
    private int balance;
    private int total;

    Further details omitted.
}
```

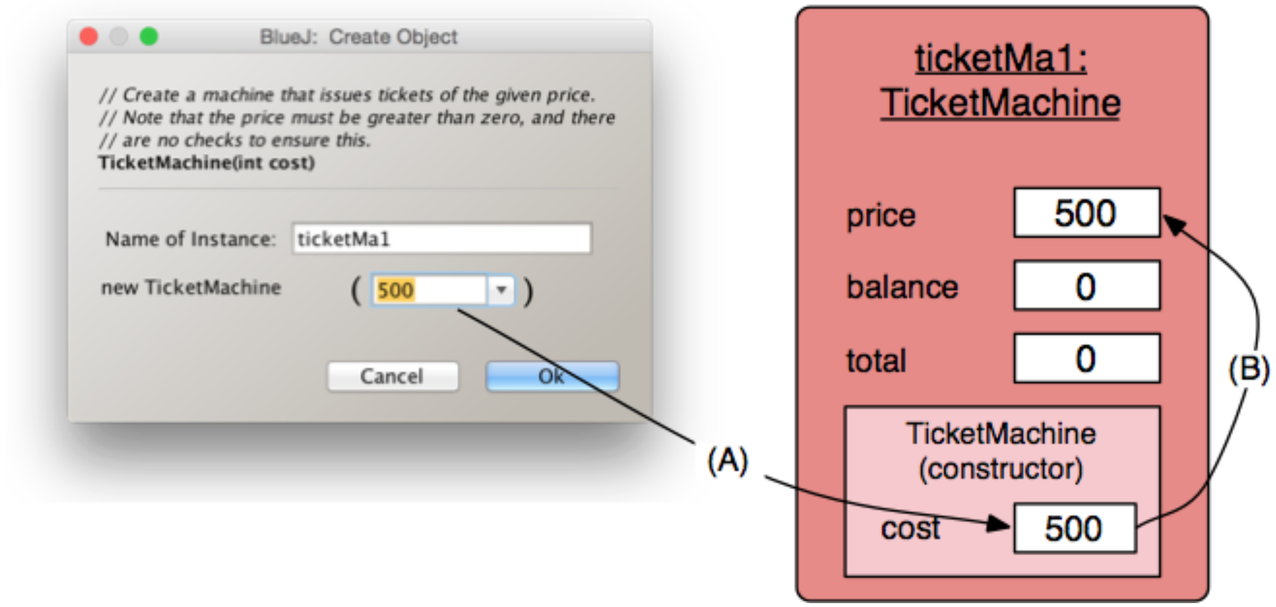


Constructors

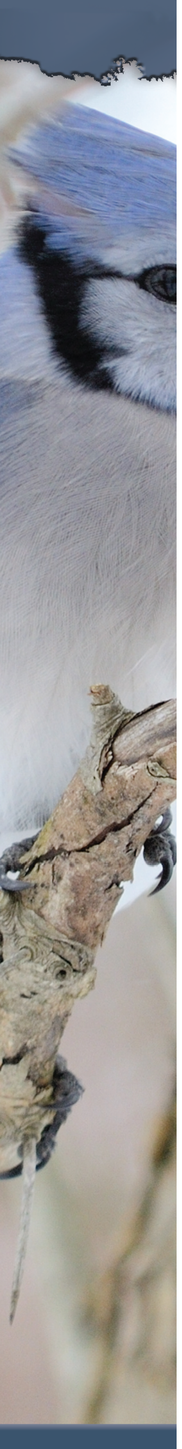
```
public TicketMachine(int cost)
{
    price = cost;
    balance = 0;
    total = 0;
}
```

- Initialise an object.
- Have the same name as their class.
- Close association with the fields:
  - Initial values stored into the fields.
  - Parameter values often used for these.

Passing data via parameters



Parameters are another sort of variable.



Assignment

- Values are stored into fields (and other variables) via assignment statements:

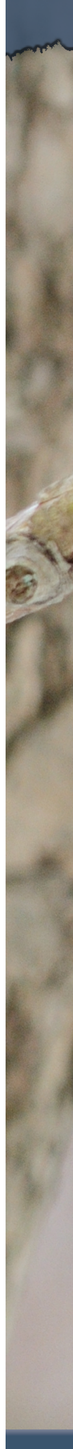
variable = *expression*;

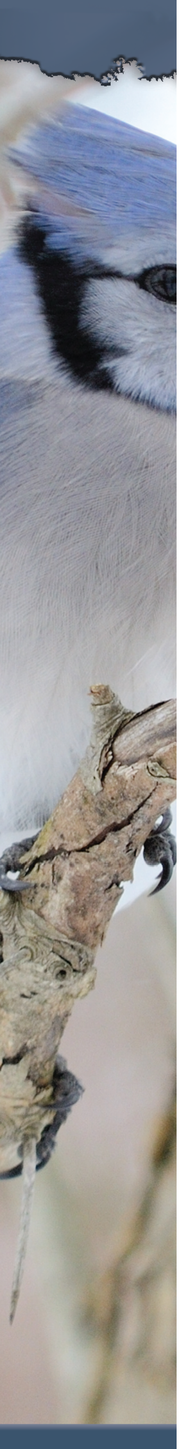
pattern

```
balance = balance + amount;
```

example

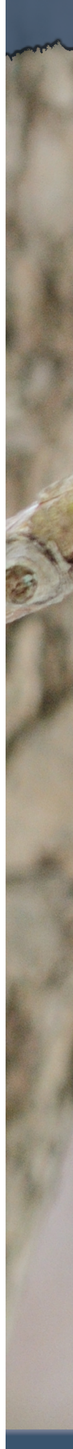
- A variable can store just one value, so any previous value is lost.

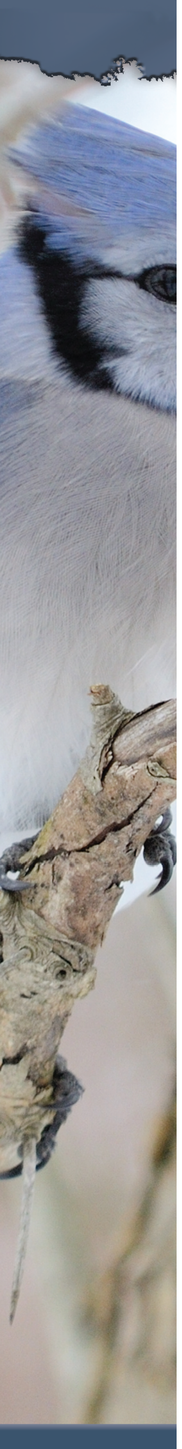




Choosing variable names

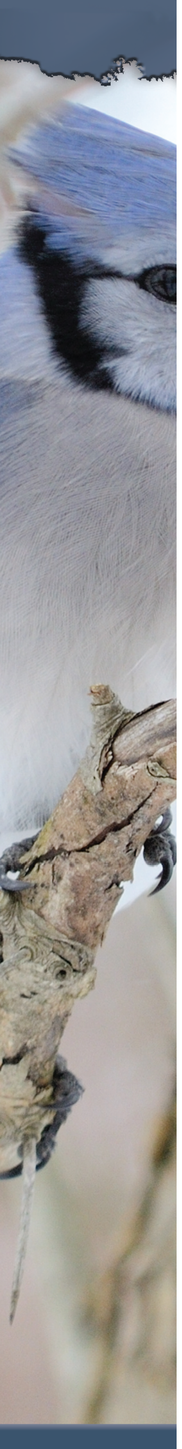
- There is a lot of freedom over choice of names. Use it wisely!
- Choose expressive names to make code easier to understand:
  - **`price`**, **`amount`**, **`name`**, **`age`**, etc.
- Avoid single-letter or cryptic names:
  - **`w`**, **`t5`**, **`xyz123`**





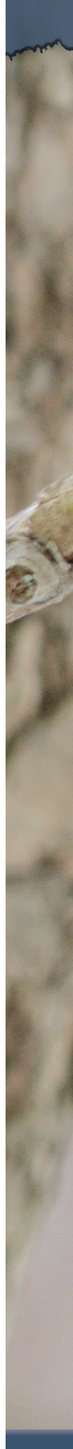
Method definitions

- Methods, including:
  - *method definitions*
  - *accessor* methods
  - *mutator* methods



Methods

- Methods implement the *behaviour* of objects.
- Methods have a consistent structure comprised of a *header* and a *body*.
- *Accessor methods* provide information about an object.
- *Mutator methods* alter the state of an object.
- Other sorts of methods accomplish a variety of tasks.



Method structure

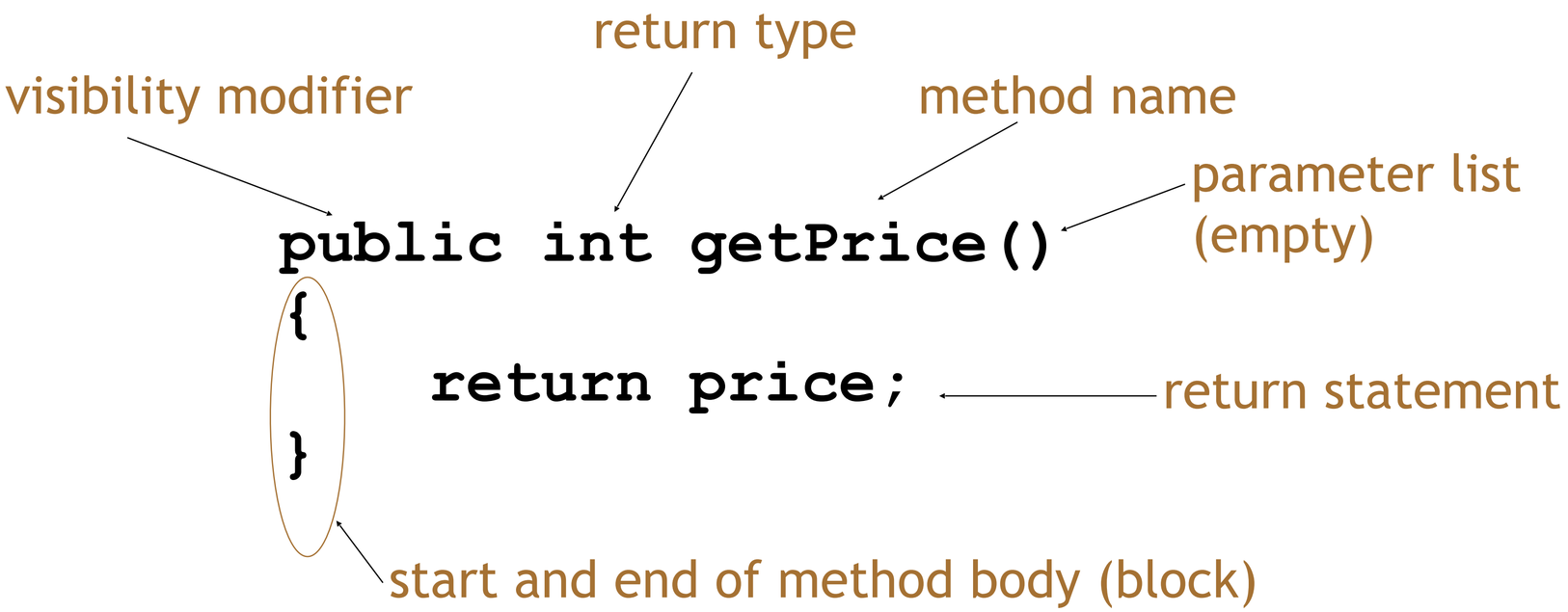
- The header:

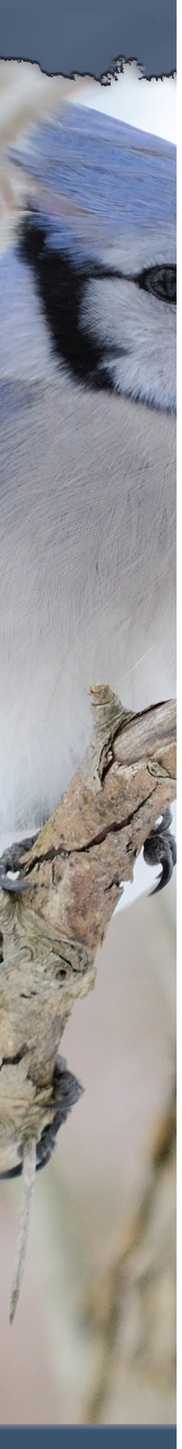
```
public int getPrice()
```

- The header tells us:
  - the *visibility* to objects of other classes;
  - whether the method *returns a result*;
  - the *name* of the method;
  - whether the method takes *parameters*.

- The body encloses the method's *statements*.

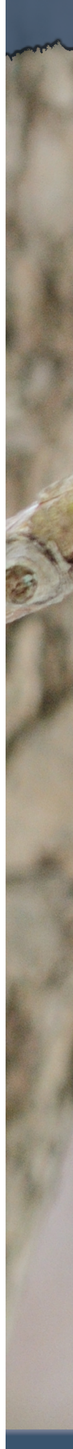
Accessor (`get`) methods

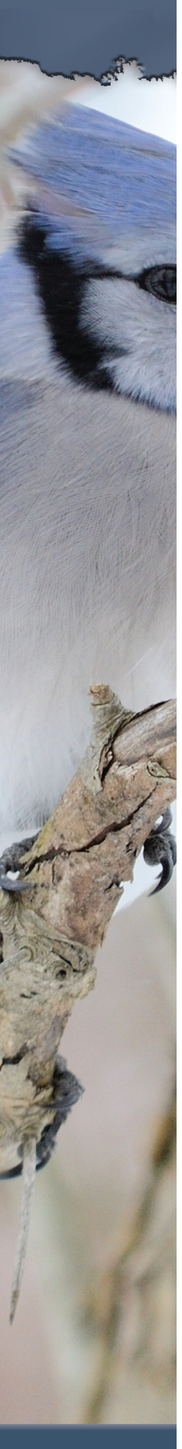




Accessor methods

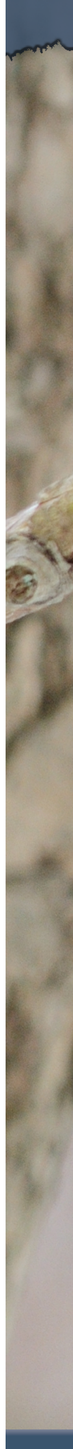
- An accessor method always has a return type that is not **`void`**.
- An accessor method returns a value (*result*) of the type given in the header.
- The method will contain a **`return`** statement to return the value.
- NB: Returning is *not* printing!



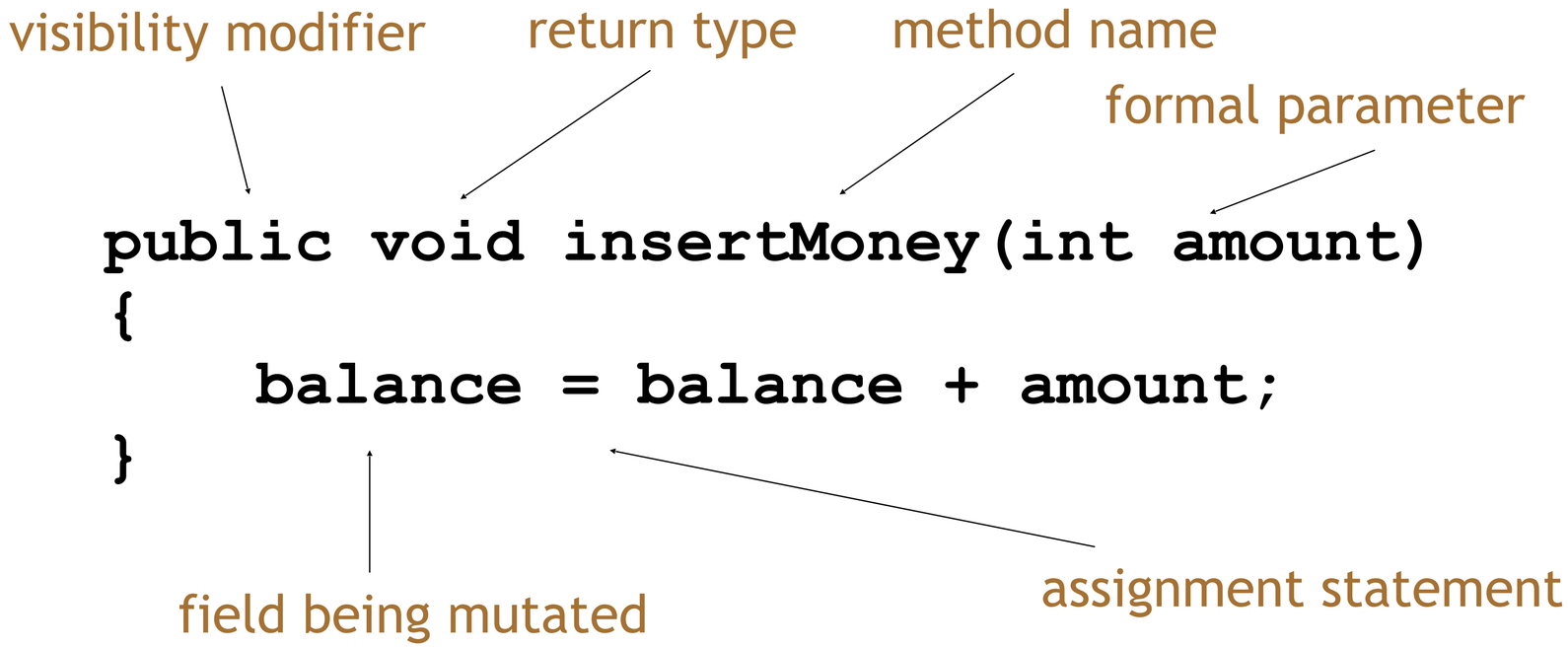


Mutator methods

- Have a similar method structure: header and body.
- Used to *mutate* (i.e., change) an object's state.
- Achieved through changing the value of one or more fields.
  - They typically contain one or more assignment statements.
  - Often receive parameters.

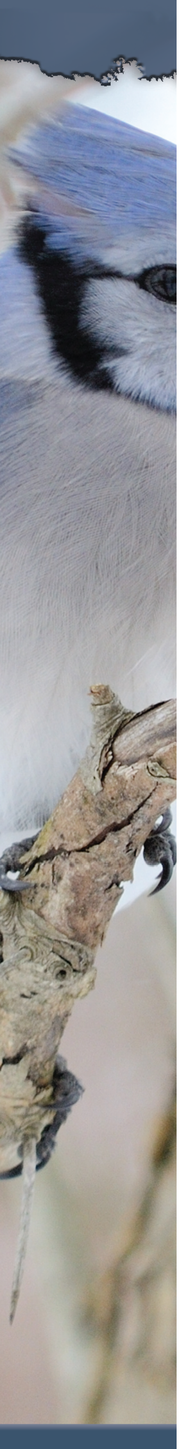


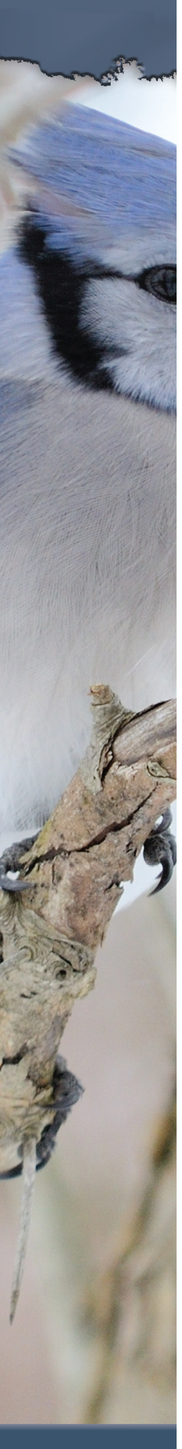
Mutator methods



`set` mutator methods

- Fields often have dedicated **`set`** mutator methods.
- These have a simple, distinctive form:
  - **`void`** return type
  - method name related to the field name
  - single formal parameter, with the same type as the type of the field
  - a single assignment statement



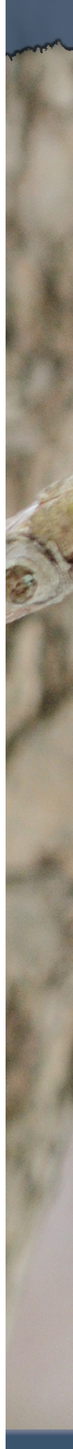


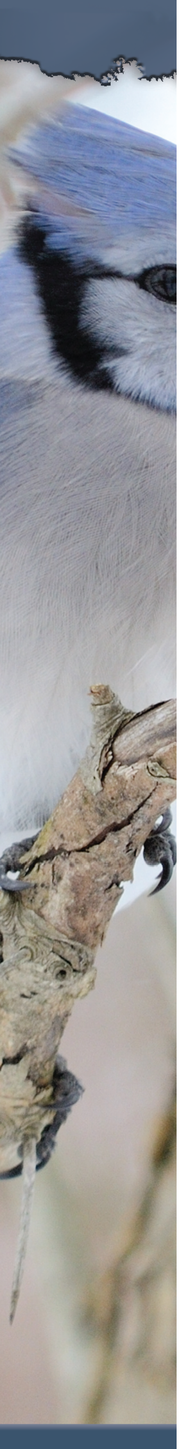
A typical `set` method

```
public void setDiscount(int amount)
{
    discount = amount;
}
```

We can easily infer that `discount` is a field of type `int`, i.e:

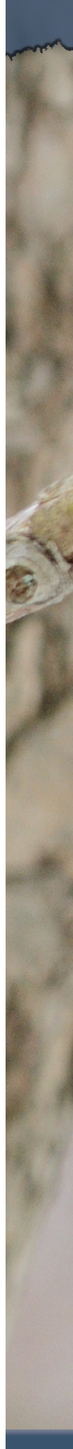
```
private int discount;
```

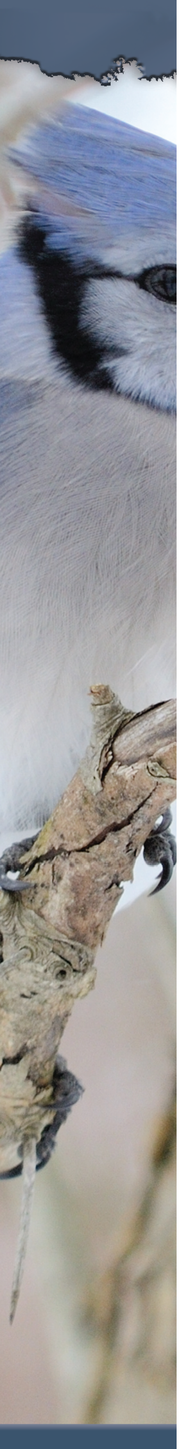




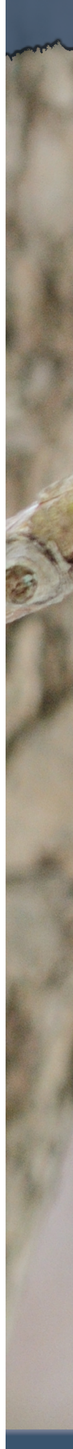
Method summary

- Methods implement all object behaviour.
- A method has a name and a return type.
  - The return-type may be **`void`**.
  - A non-**`void`** return type means the method will return a value to its caller.
- A method might take parameters.
  - Parameters bring values in from outside for the method to use.





String concatenation



Printing from methods

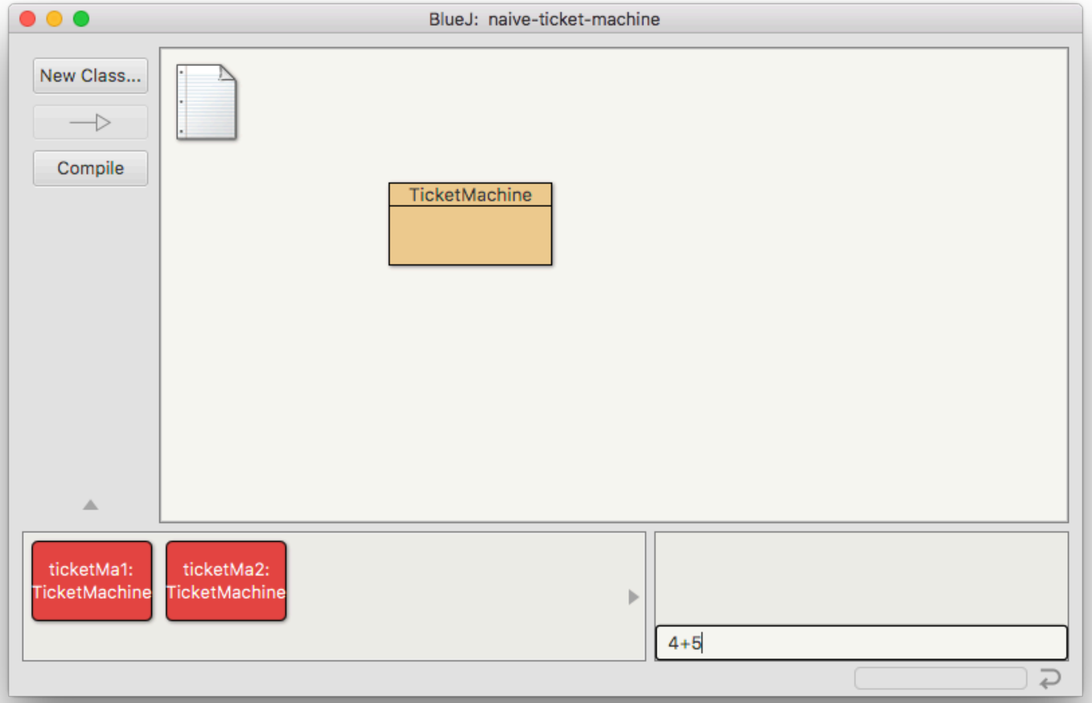
```
public void printTicket()
{
    // Simulate the printing of a ticket.
    System.out.println("####################");
    System.out.println("# The BlueJ Line");
    System.out.println("# Ticket");
    System.out.println("# " + price + " cents.");
    System.out.println("####################");
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

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

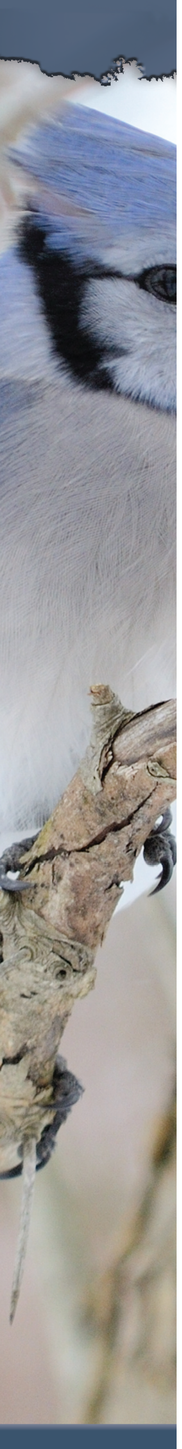
String concatenation

- 4 + 5
  9
- "wind" + "ow"
  "window"
- "Result: " + 6
  "Result: 6"
- "# " + price + " cents"
  "# 500 cents"

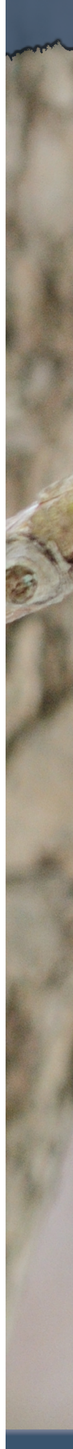
→ overloading

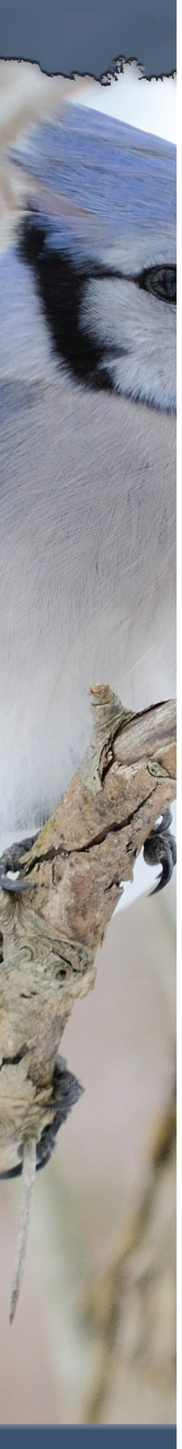
The codepad


BlueJ: naive-ticket-machine
New Class...
Compile
TicketMachine
ticketMa1:
TicketMachine
ticketMa2:
TicketMachine
4+5



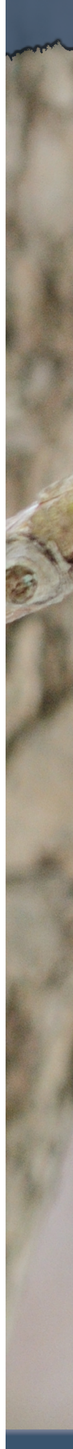
Conditional statements

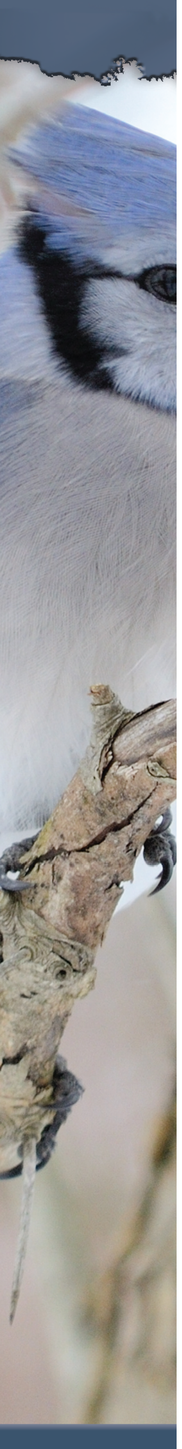




Reflecting on the ticket machines

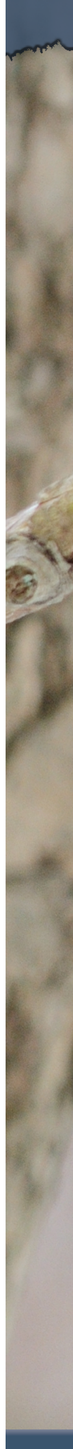
- Their behaviour is inadequate in several ways:
  - No checks on the amounts entered.
  - No refunds.
  - No checks for a sensible initialisation.
- How can we do better?
  - We need the ability to choose between different courses of action.





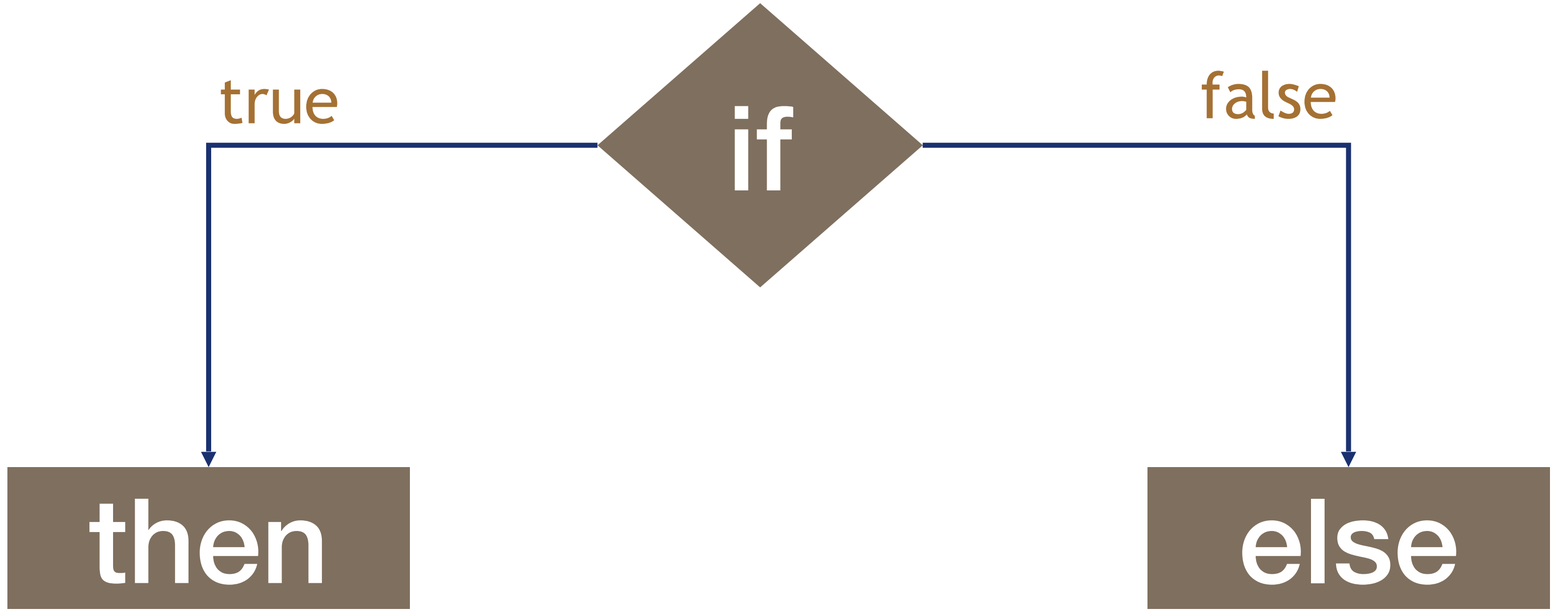
Making choices in everyday life

- If I have enough money left, then I will go out for a meal
- otherwise I will stay home and watch a movie.



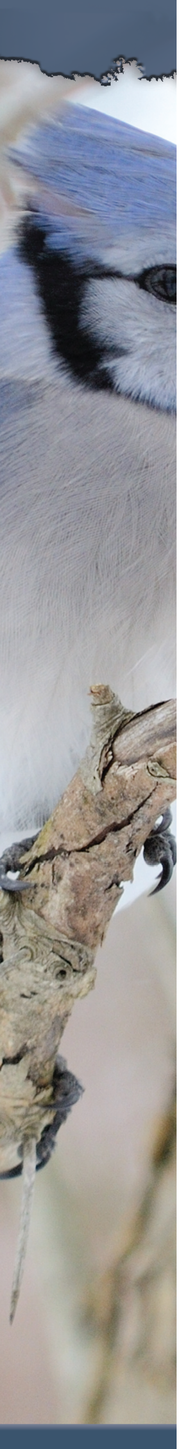
Making a choice in everyday life

If I have enough money left



Then I will go out for a meal

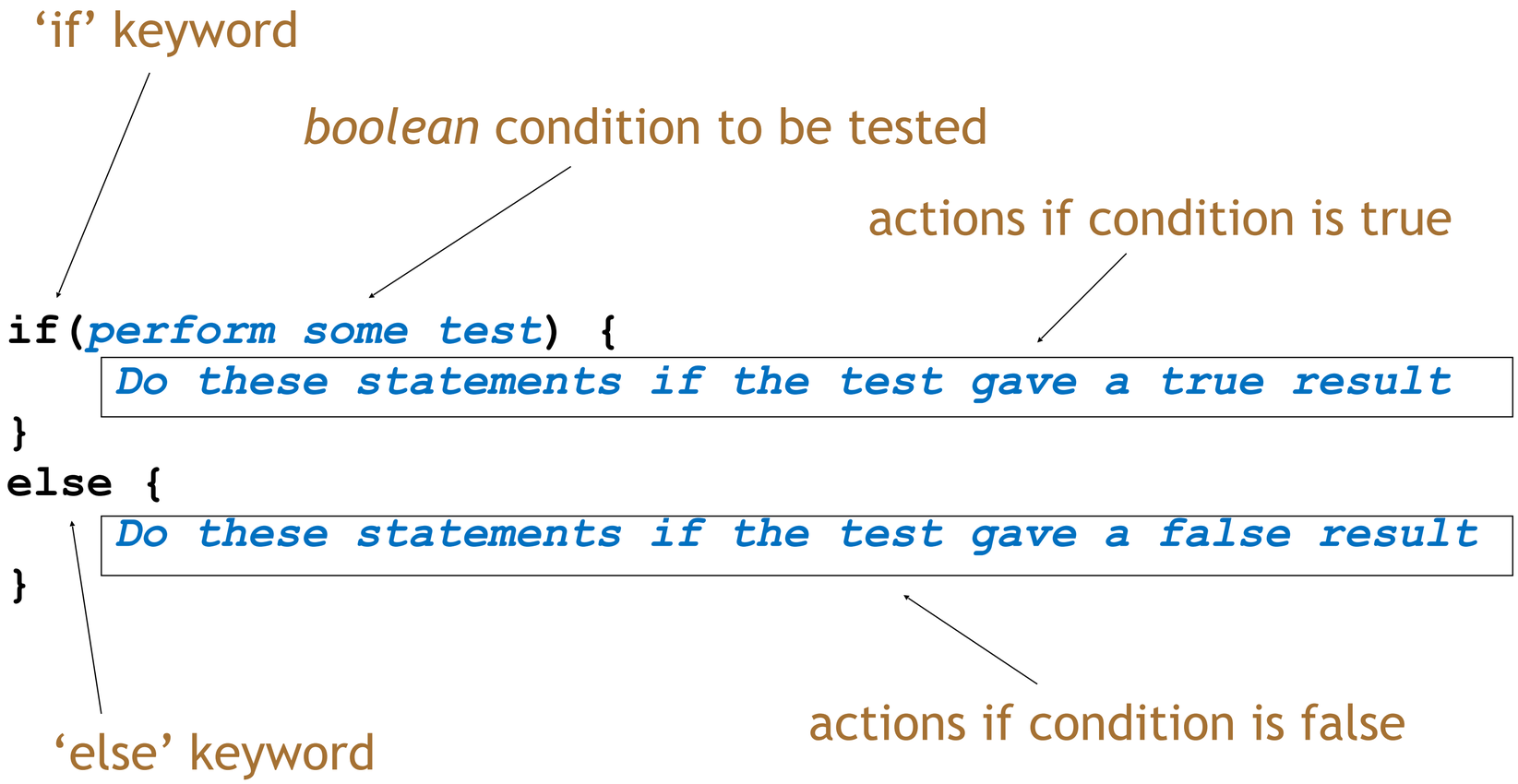
Else I will stay home and watch a movie.



Making a choice in everyday life

```
if(I have enough money left) {
    I will go out for a meal;
} else {
    I will stay home and watch a movie;
}
```

Making choices in Java

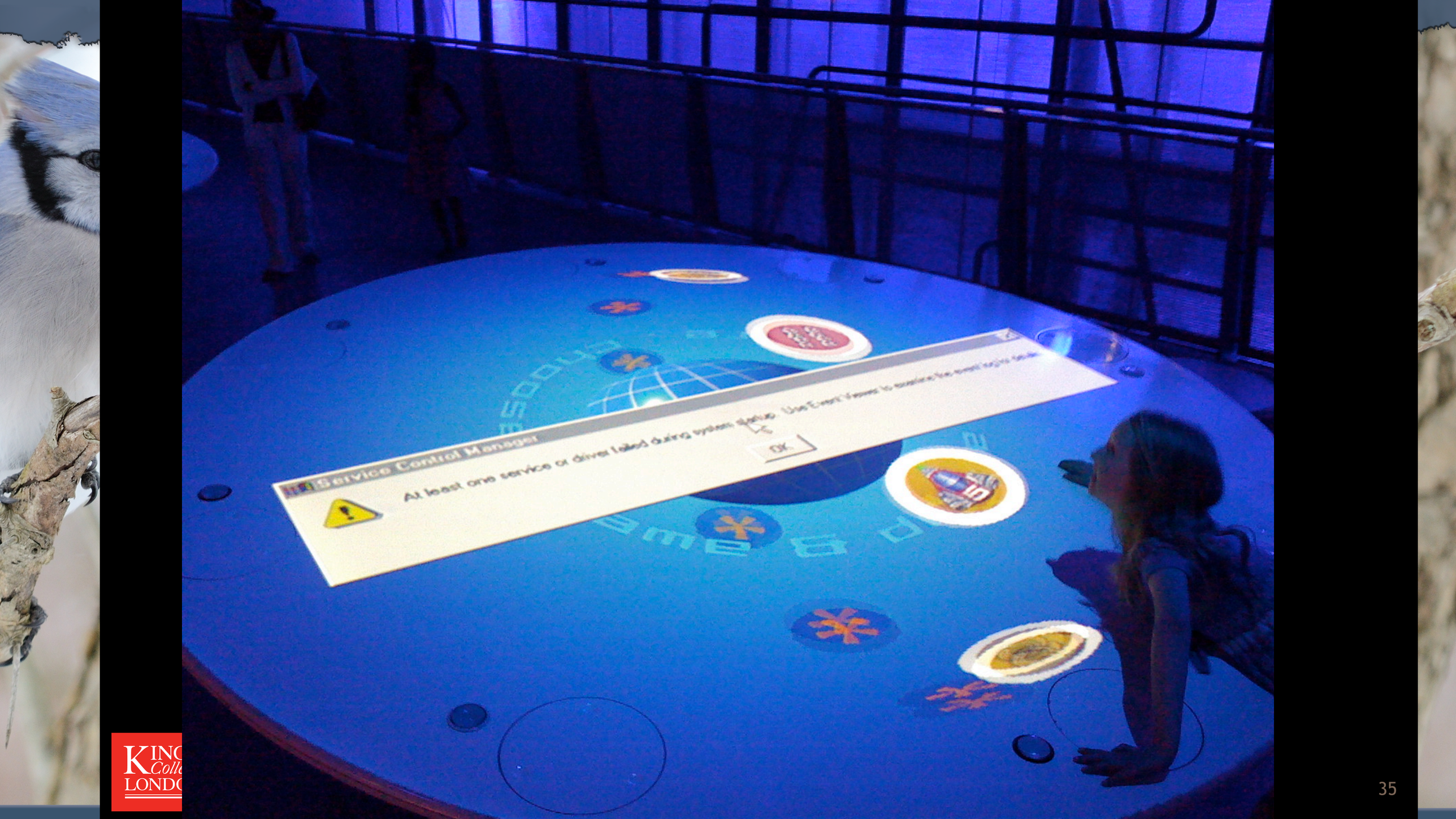


Making a choice in the ticket machine

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

conditional statement avoids an inappropriate action


Time Destination Flight Status
08:30 Puerta Plata MON303 Delayed to 1830
09:15 St Lucia VS031 Delayed to 1700
11:40 Sanford TCX011K Closing Gate 25
15:25 Dublin FR1125 Board Gate 10
16:10 Isle Of man BE7336 Closing Gate 04
16:10 Malaga TOM6273 Closing Gate 13
16:15 Jersey BE947 Closing Gate 12
16:15 Ibiza MON5982 Closing Gate 14
Time Now: 16:21 Date: 17 August 2007
colourmaster

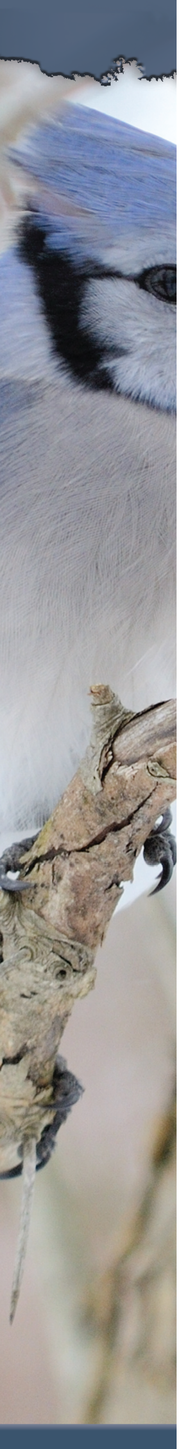

Service Control Manager
At least one service or driver failed during system startup. Use Event Viewer to examine the event log for
OK

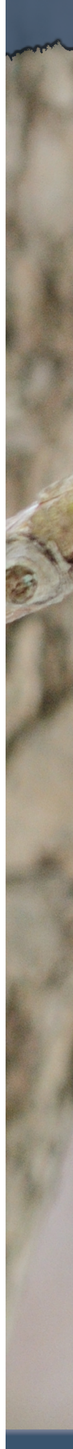

Time Flight Gate Remarks
10:08a 6668 B57 On Time
1:15p 6554 B84 On Time
0:28a 5062 A61 On Time
1:30a 5159 A63 On Time
0:07a 6718 B95 On Time
2:07p 6675 B73 On Time
0:40a 5042 A63 On Time
0:10a 5192 A68 On Time
2:21a 6050 B77 On Time
2:10a 5192 A68 On Time, 2 Stops
2:58a 5180 A68 On Time
2:24a 7544 B83 On Time
2:23a 7393 B86 On Time
0:00a 5070 A68 On Time, 1 Stop
2:53a 6615 B59 On Time
0:00a 5068 A61 Departs 11:20a
0:04a 6950 B94 On Time
0:08p 6097 B43 On Time
2:28a 6614 B53 On Time
2:25a 58 B16 On Time
0:0p 6657 B46 On Time
2:2a 6570 B89 On Time
2:5a 5127 A61 On Time, 1 Stop
2:6a 6534 B67 Departs 11:37a
0:0a 7512 B58 On Time
2:5a 361 B32 On Time
2:6a 6399 B93 On Time
2:7a 6888 B48 On Time
2:5a 5001 A63 Departs 11:09a
0:0a 6722 B84 On Time
0:0a 5143 A68 On Time
0:0a 6575 B27 Boarding
Departures Time Flight Gate Remarks
Little Rock 10:55a 6730 B93 On Time
Los Angeles 8:30a 5705 B38 On Time
Los Angeles 9:32a 9043 B44 Aircraft Delayed
Los Angeles 10:45a 5103 B46 On Time
Los Angeles 11:44a 5715 B44 On Time
Louisville 10:09a 6710 B77 On Time
Madison 10:40a 7668 B67 On Time
Memphis 10:05a 7029 B69 On Time
Midland 10:51a 8098 B94 On Time
Milwaukee 10:05a 8088 B90 On Time
Minneapolis/St. Paul 8:25a 4670 B52 On Time
Minneapolis/St. Paul 10:00a 4656 B54 On Time
Missoula 8:02a 6721 B69 On Time
Missoula
Moline
Monterey
Montrose
Montrose
ddcexe.exe - Common Language Runtime Debugging Services
Application has generated an exception that could not be handled.
Process id=0x290 (656), Thread id=0x288 (648).
Click OK to terminate the application.
Click CANCEL to debug the application.
OK
Cancel
Munich 10:23a 902 B45 On Time, 1 Stop
Nashville 10:05a 6042 B94 On Time
New Orleans 10:05a 5080 B24 On Time
New York LaGuardia 8:25a 7097 B29 On Time
New York LaGuardia 10:54a 7095 B22 On Time
Newark 10:10a 3470 B16 On Time
Newark 10:37a 6445 B25 On Time
North Platte 11:30a 5137 A61 On Time
Oklahoma City 8:40a 4888 B83 On Time
Oklahoma City 10:08a 5997 B49 On Time
Omaha 7:55a 4474 B73 On Time
Omaha 10:11a 5494 B28 On Time
Ontario, CA 9:33a 4157 B75 On Time
Ontario, CA 10:28a 5062 A61 On Time, 2 Stops
Orange County 9:07a 5839 B46 On Time
Departures Time
Ontario, CA 12:11p
Orange County 11:52a
Orlando 10:05a
Orlando 12:30p
Page 11:30a
Palm Springs 11:29a
Pasco 12:29p
Philadelphia 10:25a
Philadelphia 11:10a
Phoenix 10:54
Phoenix 11:30
Phoenix 11:54
Phoenix 12:1
Pierre 10:3
Pittsburgh 10:0
Portland, OR 12:
Prescott 10:
Pueblo 12:
Rapid City 10
Rapid City 12
Regina 12
Reno/Tahoe 1
Riverton 1
Rochester
Sacramento
Sacramento
Salt Lake City
San Antonio
San Diego
San Francisco
San Francisco
San Jose, CA
Santa Barbara
NEC
MultiSync LCD 4610

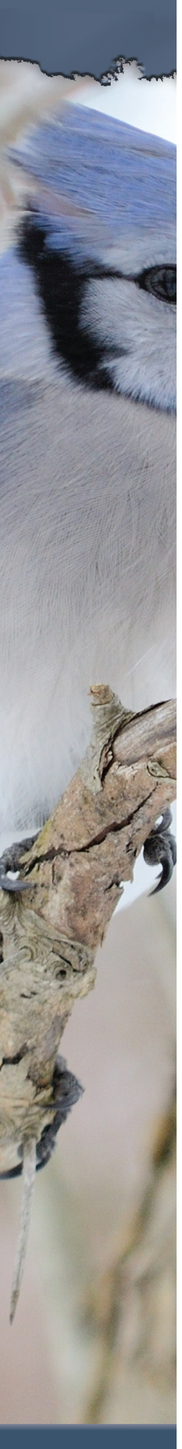

NCR
Please wait...
Microsoft Windows xp
Professional
Copyright © 1985-2001
Microsoft Corporation
Microsoft
Windows is shutting down...
Insert Here
CARD
LINK
MasterCard
Maestro
Cirrus
VISA
VISA
PLUS
CANCEL

Protective mutators

- A set method does not have to always assign unconditionally to the field.
- The parameter may be checked for validity and rejected if inappropriate.
- Mutators thereby protect fields.
- Mutators support *encapsulation*.

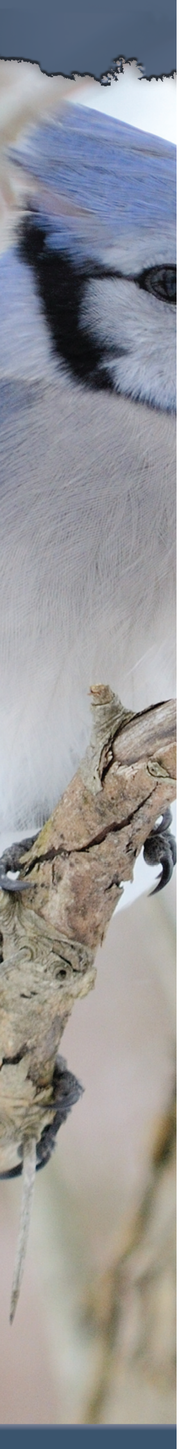






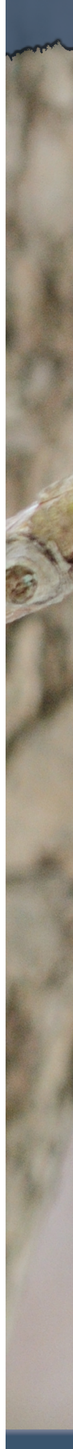
Variables

- Fields (instance variables)
- Parameters
- Local variables

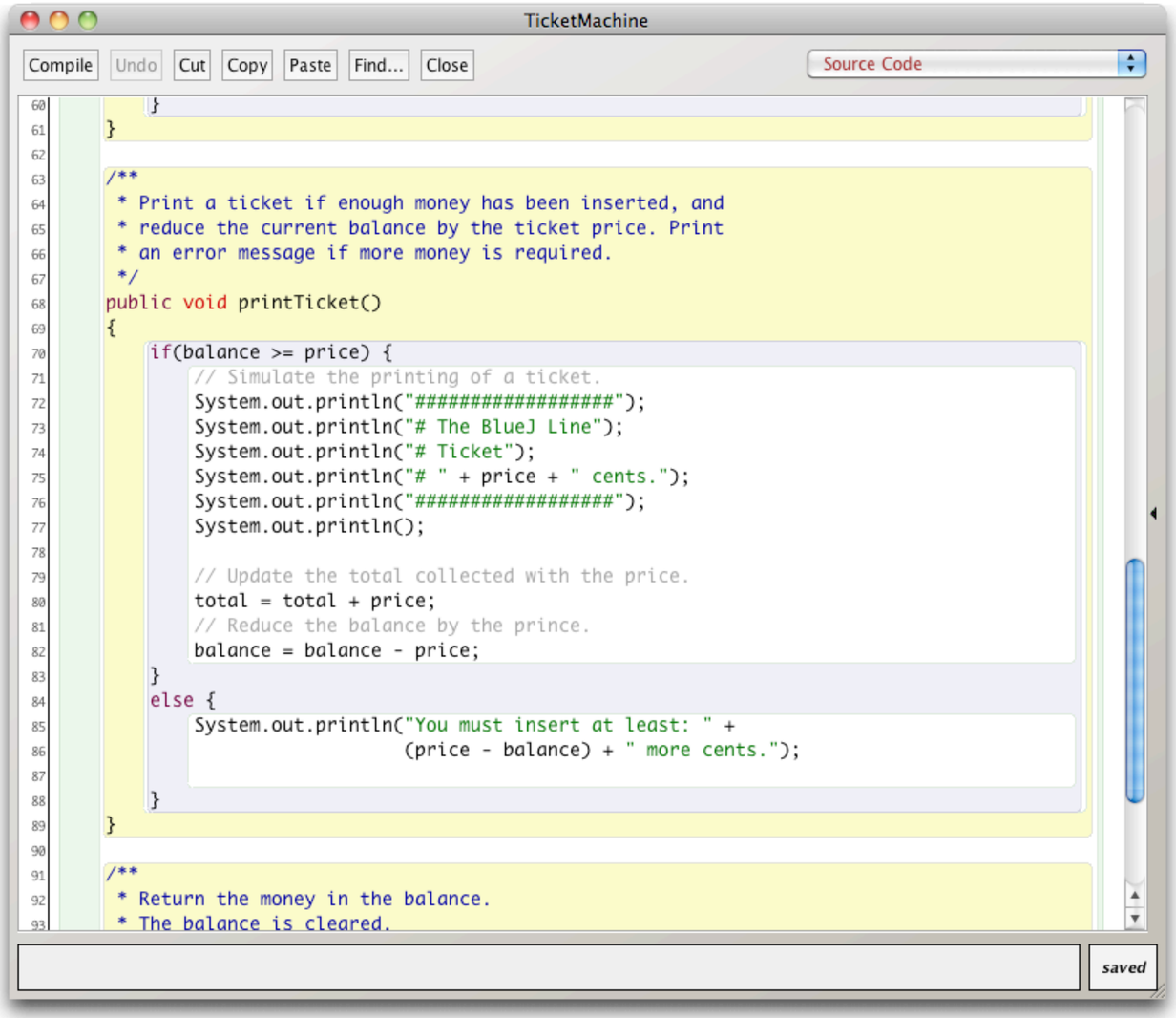


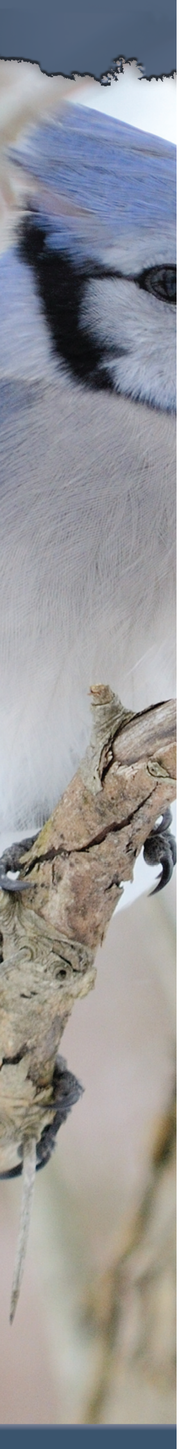
Variables – a recap

- Fields are one sort of variable.
  - They store values through the life of an object.
  - They are accessible throughout the class.
- Parameters are another sort of variable:
  - They receive values from outside the method.
  - They help a method complete its task.
  - Each call to the method receives a fresh set of values.
  - Parameter values are short lived.



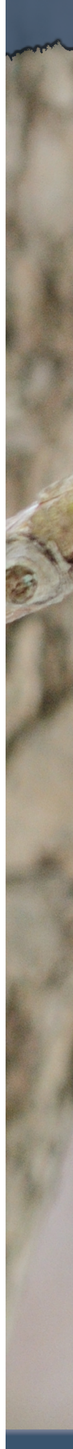
Scope highlighting

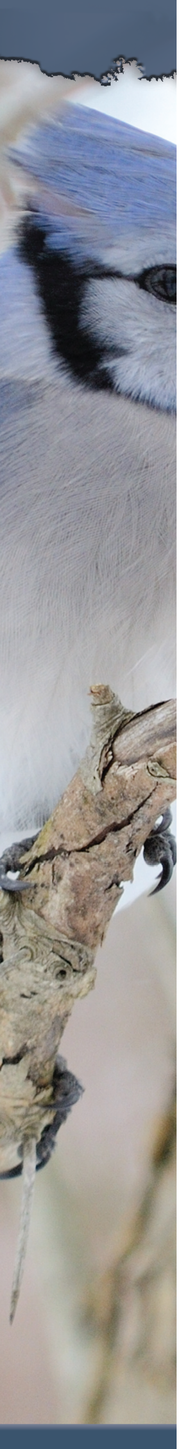




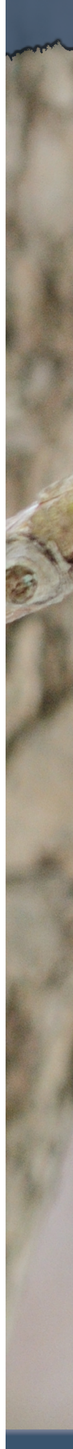
Scope and lifetime

- Each block defines a new scope.
  - Class, method and statement.
- Scopes may be nested:
  - statement block inside another block inside a method body inside a class body.
- Scope is static (textual).
- Lifetime is dynamic (runtime).





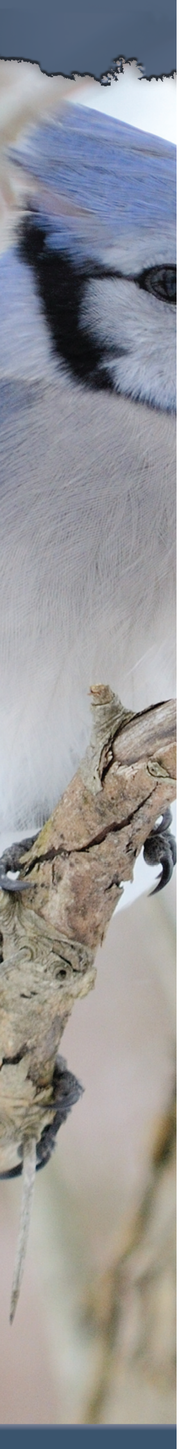
How do we write a method to ‘refund’ an excess balance?



Unsuccessful attempt

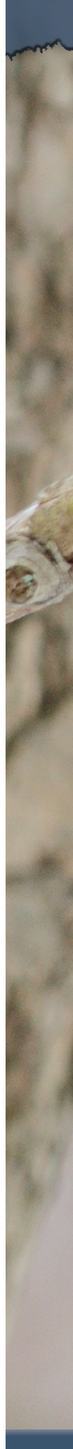
```
public int refundBalance()
{
    // Return the amount left.
    return balance;
    // Clear the balance.
    balance = 0;
}
```

It looks logical, but the language does not allow it.

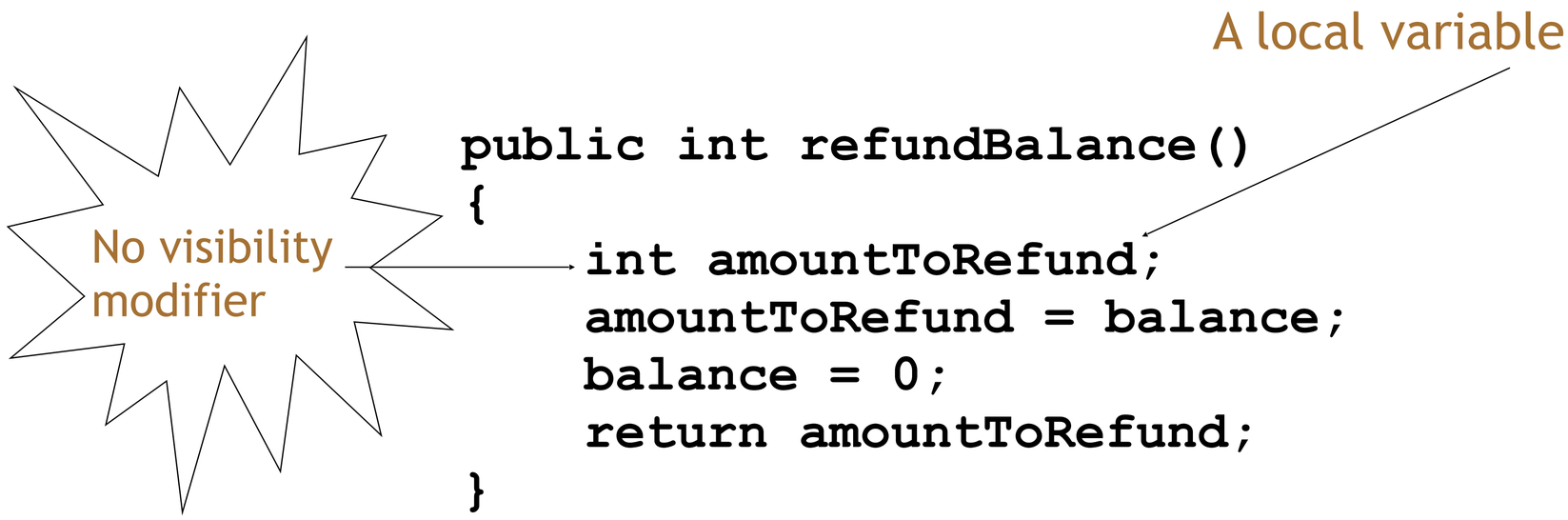


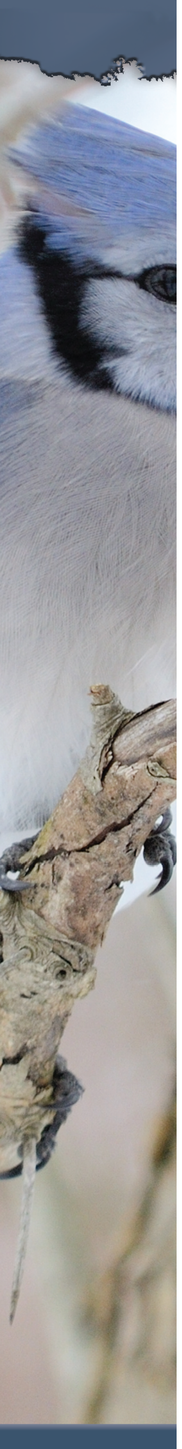
Local variables

- Methods can define their own, *local* variables:
  - Short lived, like parameters.
  - The method sets their values – unlike parameters, they do not receive external values.
  - Used for ‘temporary’ calculation and storage.
  - They exist only as long as the method is being executed.
  - They are only accessible from within the method.
  - They are defined within a particular *scope*.



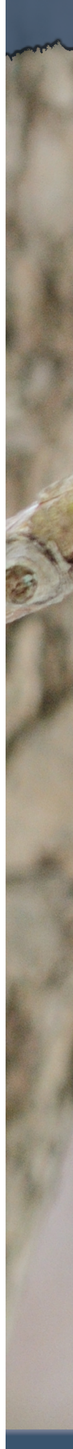
Local variables

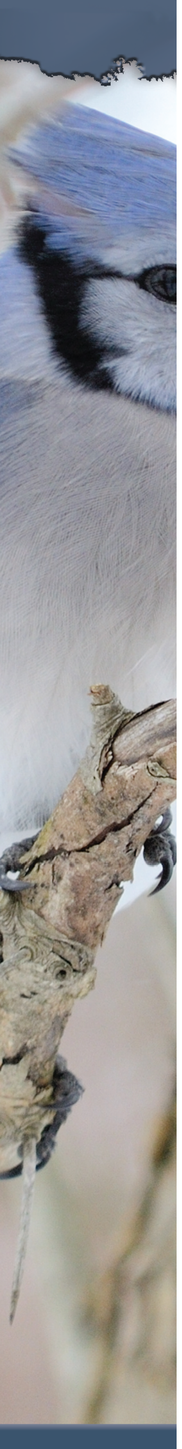




Scope and lifetime

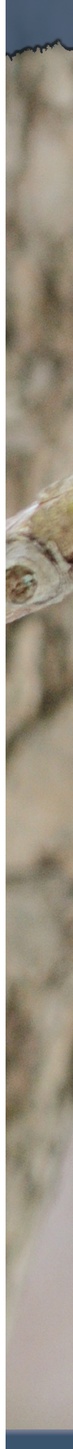
- The scope of a field is its whole class.
- The lifetime of a field is the lifetime of its containing object.
- The scope of a local variable is the block in which it is declared.
- The lifetime of a local variable is the time of execution of the block in which it is declared.

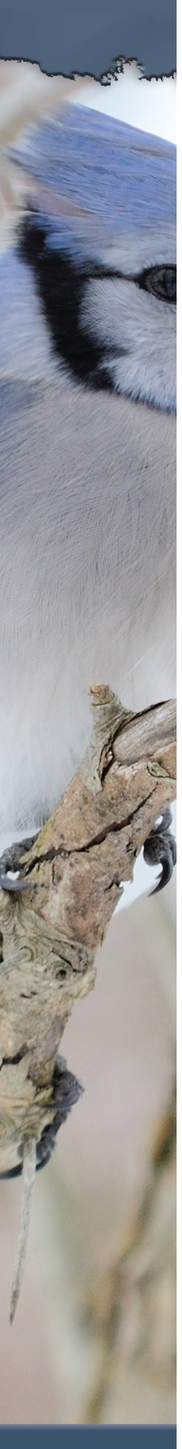




Review (1)

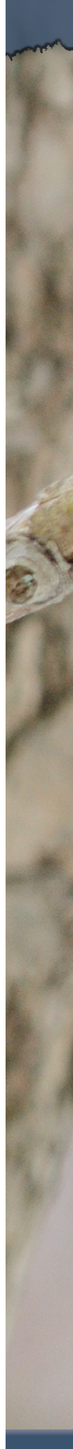
- Class bodies contain fields, constructors and methods.
- Fields store values that determine an object's state.
- Constructors initialise objects – particularly their fields.
- Methods implement the behaviour of objects.

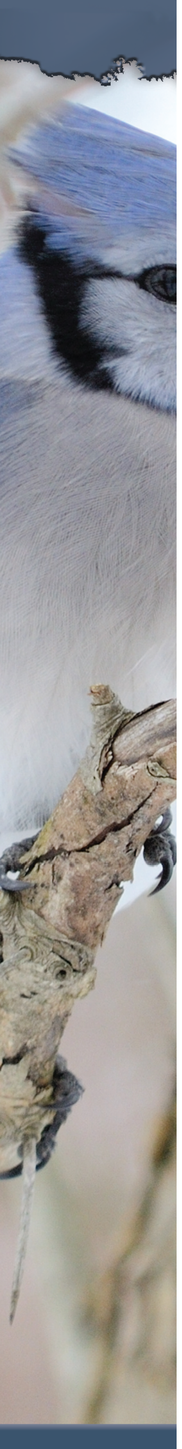




Review (2)

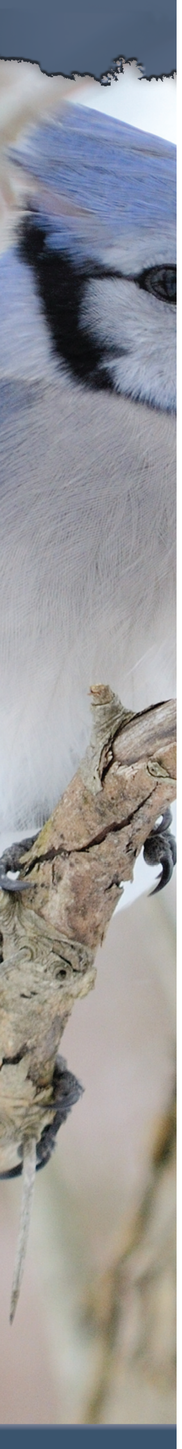
- Fields, parameters and local variables are all variables.
- Fields persist for the lifetime of an object.
- Local variables are used for short-lived temporary storage.
- Parameters are used to receive values into a constructor or method.





Review (3)

- Methods have a return type.
- **`void`** methods do not return anything.
- non-**`void`** methods always return a value.
- non-**`void`** methods must have a return statement.



Review (4)

- ‘Correct’ behaviour often requires objects to make decisions.
- Objects can make decisions via conditional (if) statements.
- A true-or-false test allows one of two alternative courses of actions to be taken.