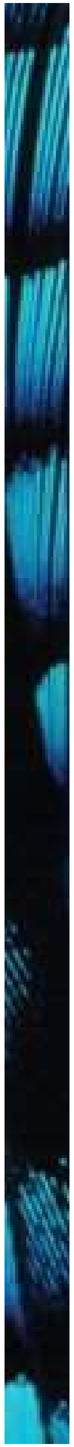
Understanding class definitions

Looking inside classes



Main concepts to be covered

- fields
- constructors
- methods
- parameters
- assignment statements

Ticket machines – an external view

- Exploring the behavior 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

Demo

Ticket machines – an internal view

- Interacting with an object gives us clues about its behavior.
- Looking inside allows us to determine how that behavior 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

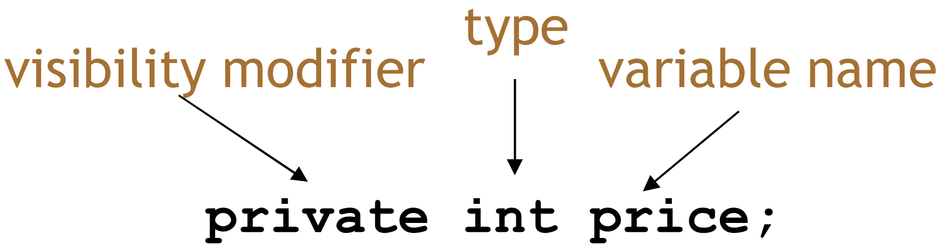
- Words with a special meaning in the language:
  - **`public`**
  - **`class`**
  - **`private`**
  - **`int`**
- Also known as *reserved words*.

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.
}
```

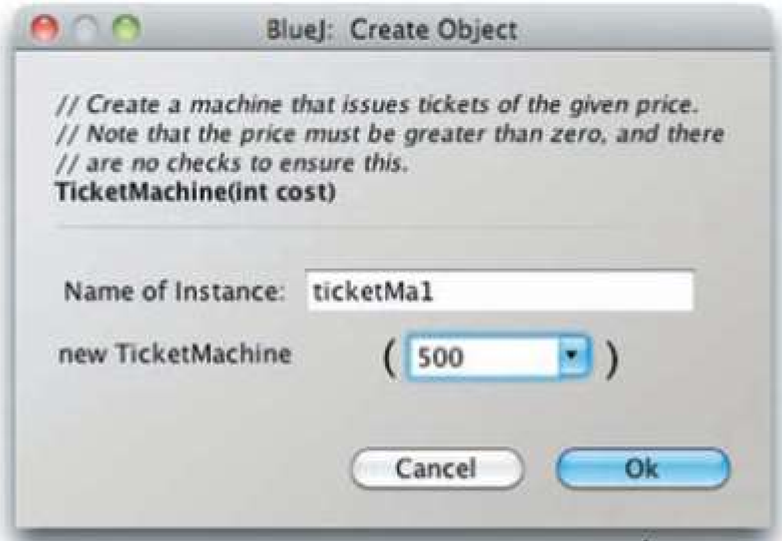


Constructors

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

- Initialize an object.
- Have the same name as their class.
- Close association with the fields.
- Store initial values into the fields.
- External parameter values for this.

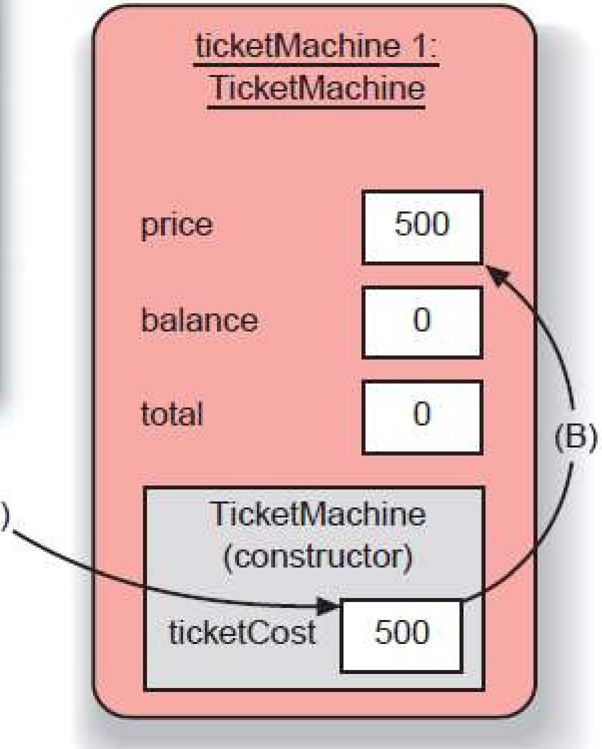
Passing data via parameters



// Create a machine that issues tickets of the given price.
// Note that the price must be greater than zero, and there
// are no checks to ensure this.
TicketMachine(int cost)

Name of Instance: ticketMa1

new TicketMachine (500)



Parameters are another sort of variable.

Assignment

- Values are stored into fields (and other variables) via assignment statements:
  - *variable = expression;*
  - `price = cost;`
- A variable stores a single value, so any previous value is lost.

Choosing variable names

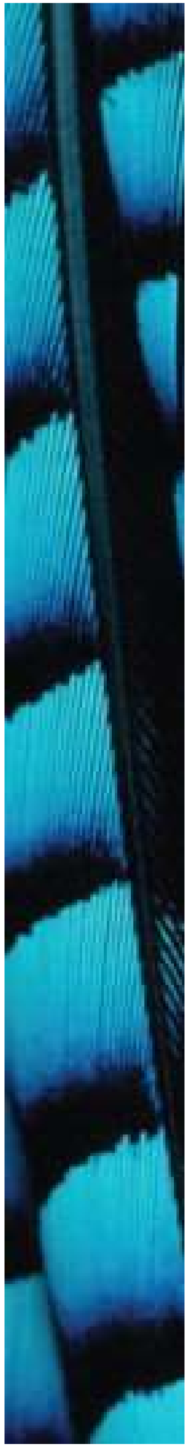
- There is a lot of freedom over choice of names. Use it wisely!
- Choose expressive names to make code easier to understand:
  - **`price`**, **`amount`**, **`name`**, **`age`**, etc.
- Avoid single-letter or cryptic names:
  - **`w`**, **`t5`**, **`xyz123`**

Main concepts to be covered

- methods
  - including accessor and mutator methods
- conditional statements
- string concatenation
- local variables

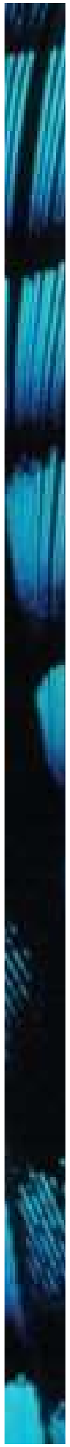
Methods

- Methods implement the behavior 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.

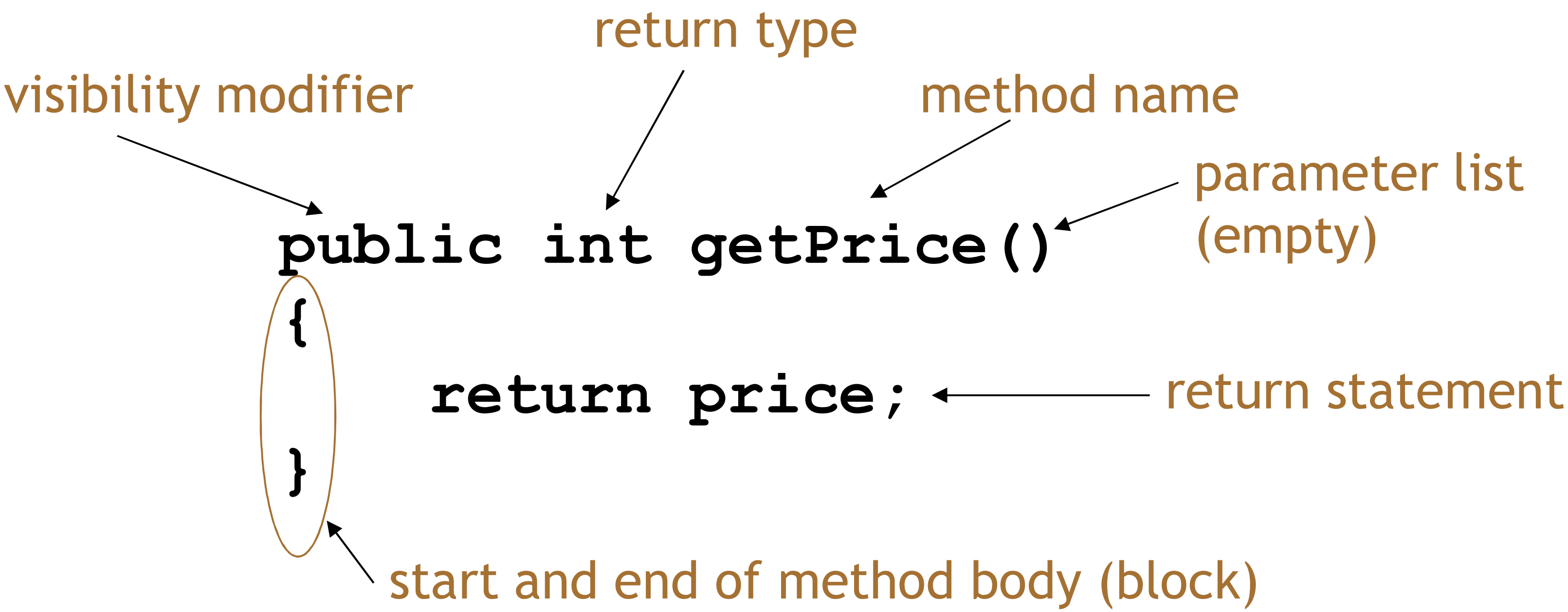


Method structure

- The header provides the method’s *signature*:
  - `public int getPrice()`
- The header tells us:
  - the name of the method
  - what parameters it takes
  - whether it returns a result
  - its visibility to objects of other classes
- The body encloses the method’s statements.



Accessor (`get`) methods



return type

visibility modifier

method name

parameter list (empty)

```
public int getPrice()
{
    return price;
}
```

return statement

start and end of method body (block)

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!

Test

```
public class CokeMachine
{
private price;

public CokeMachine()
{
   price = 300
}

public int getPrice
{
   return Price;
}
```

- What is wrong here?

(there are five errors!)

Test

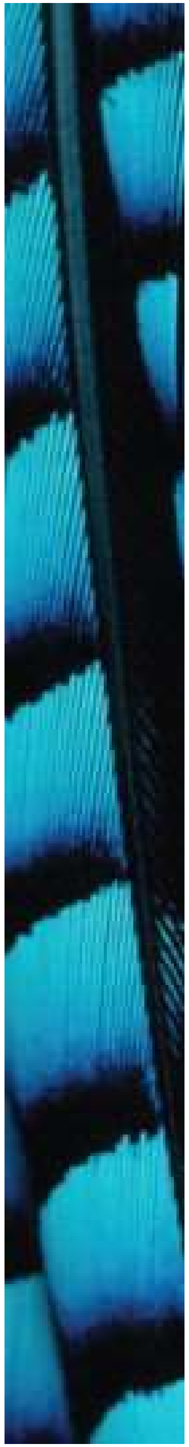
```
public class CokeMachine
{       int
private price;

public CokeMachine()
{
    price = 300;
}

public int getPrice()
{
    return Price;
}
}
```

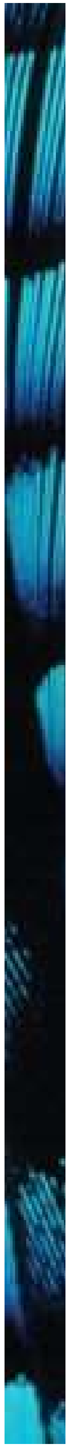
- What is wrong here?

(there are five errors!)

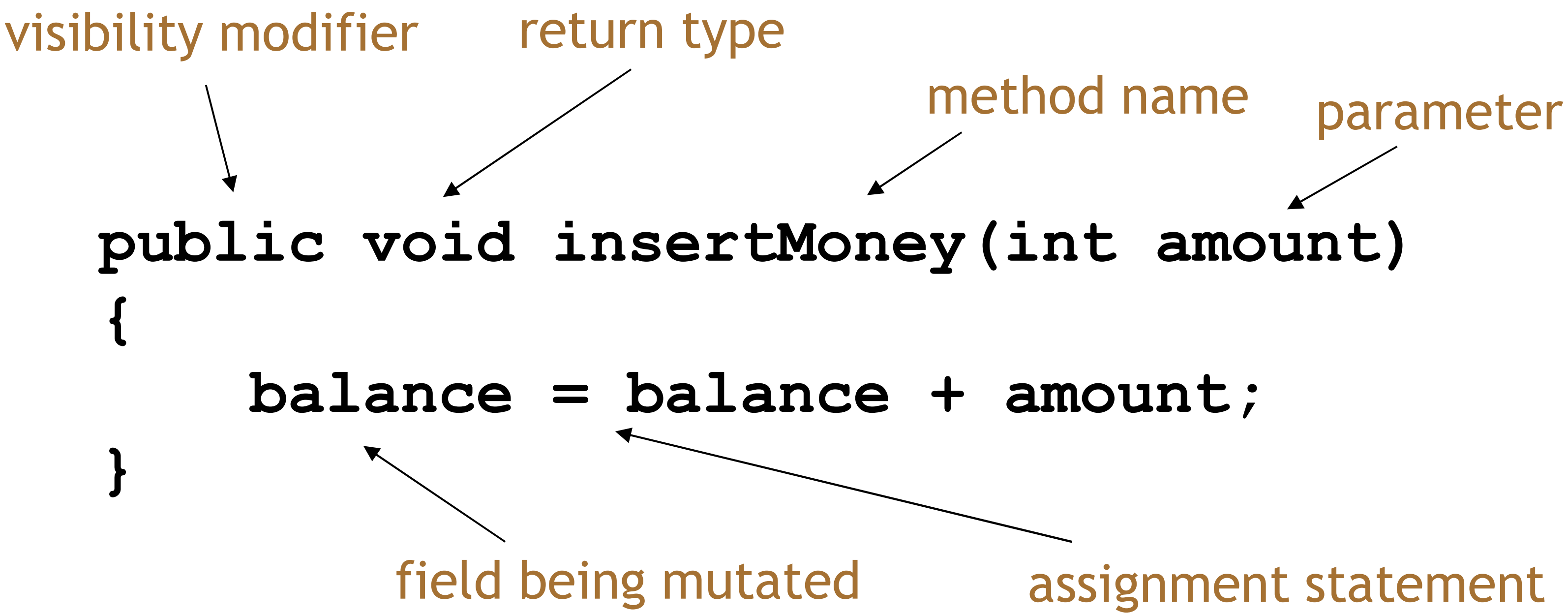


Mutator methods

- 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.
  - Typically contain assignment statements.
  - Often receive parameters.



Mutator methods



visibility modifier

return type

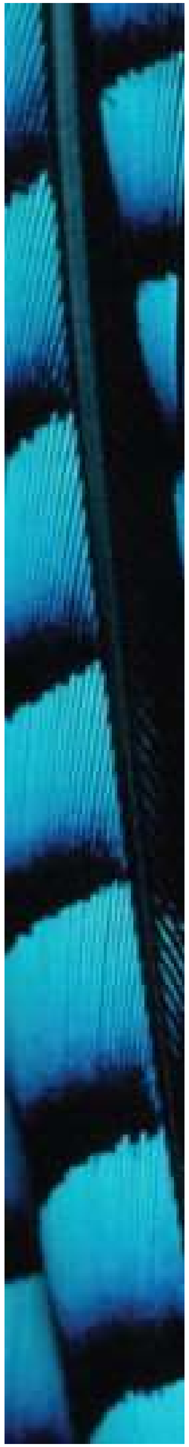
method name

parameter

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

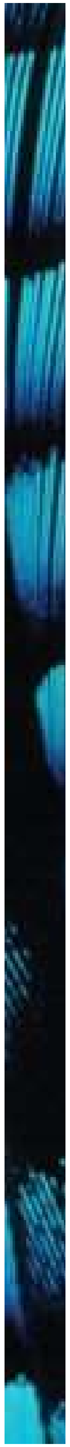
field being mutated

assignment statement



`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 parameter, with the same type as the type of the field
  - a single assignment statement



A typical `set` method

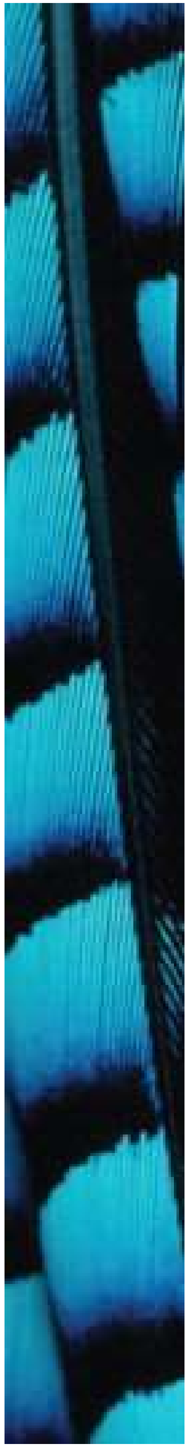
```
public void setDiscount(int amount)
{
    discount = amount;
}
```

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

```
private int discount;
```

Protective mutators

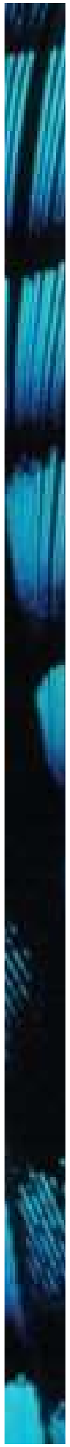
- A set method does not have to assign the parameter to the field.
- The parameter may be checked for validity and rejected if inappropriate.
- Mutators thereby protect fields.
- Mutators support *encapsulation*.



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

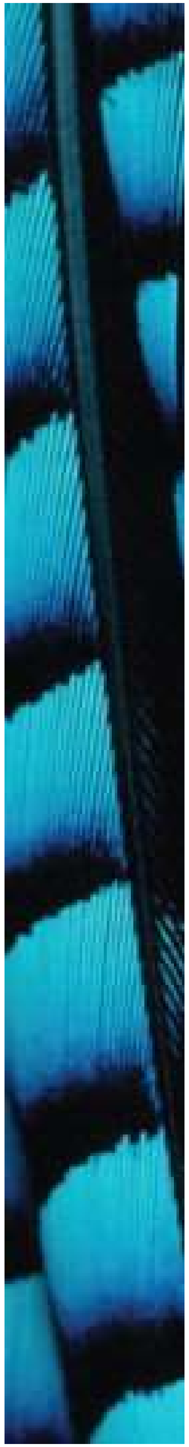
Quiz

- System.out.println(5 + 6 + "hello");

```
11hello
```

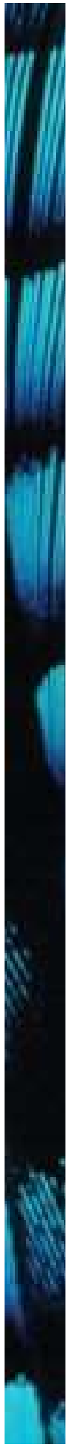
- System.out.println("hello" + 5 + 6);

```
hello56
```



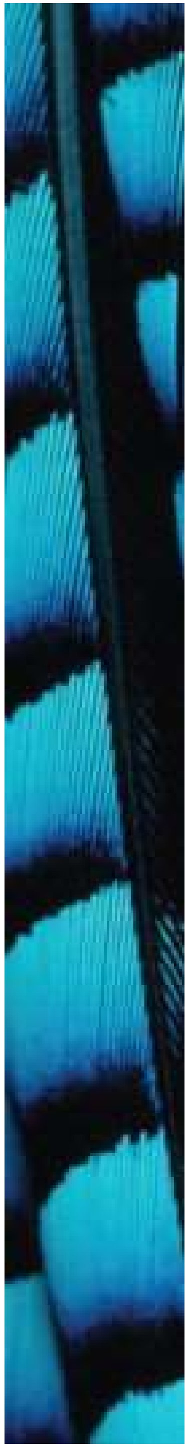
Method summary

- Methods implement all object behavior.
- 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.



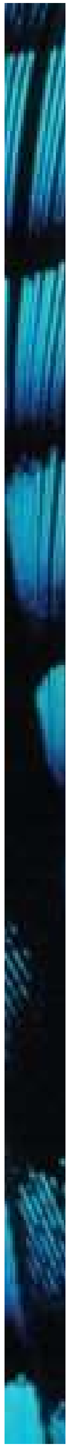
Reflecting on the ticket machines

- Their behavior is inadequate in several ways:
  - No checks on the amounts entered.
  - No refunds.
  - No checks for a sensible initialization.
- How can we do better?
  - We need more sophisticated behavior.



Making choices in everyday life

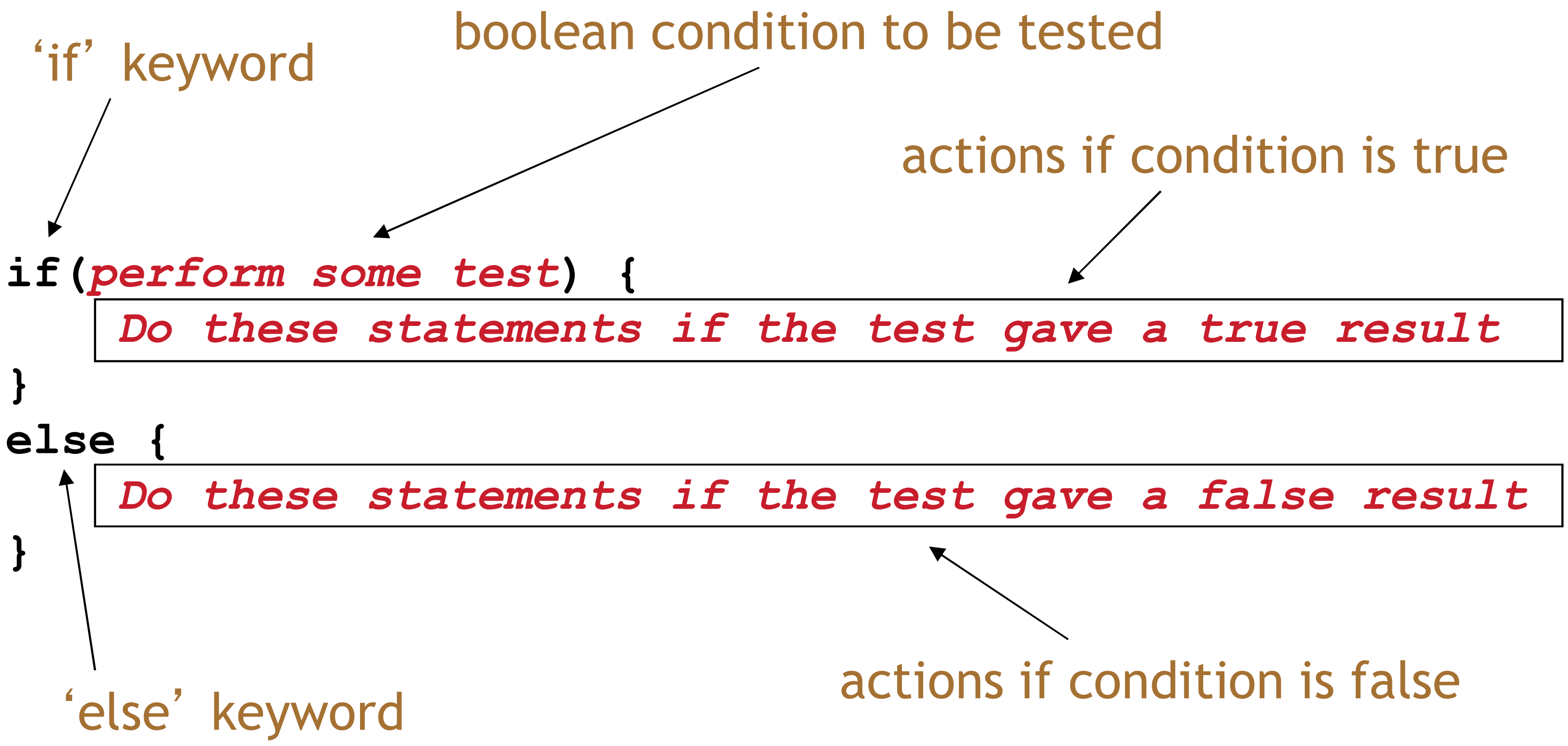
- If I have enough money left, then I will go out for a meal
- otherwise I will stay home and watch a movie.



Making a choice in everyday life

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

Making choices in Java



‘if’ keyword

boolean condition to be tested

actions if condition is true

```
if(perform some test) {
    Do these statements if the test gave a true result
}
else {
    Do these statements if the test gave a false result
}
```

‘else’ keyword

actions if condition is false

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: " +
            amount);
    }
}
```

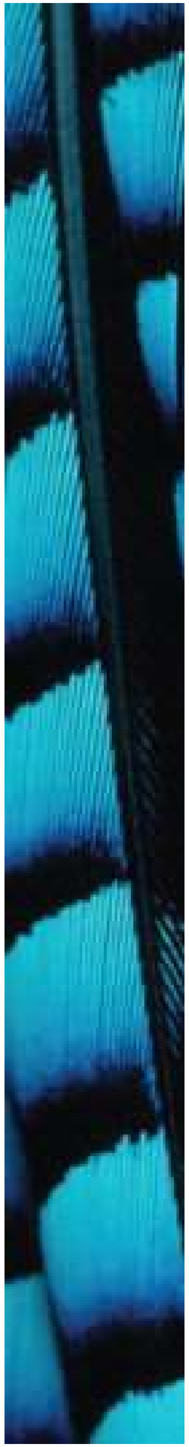
How do we write 'refundBalance'?

Variables – a recap

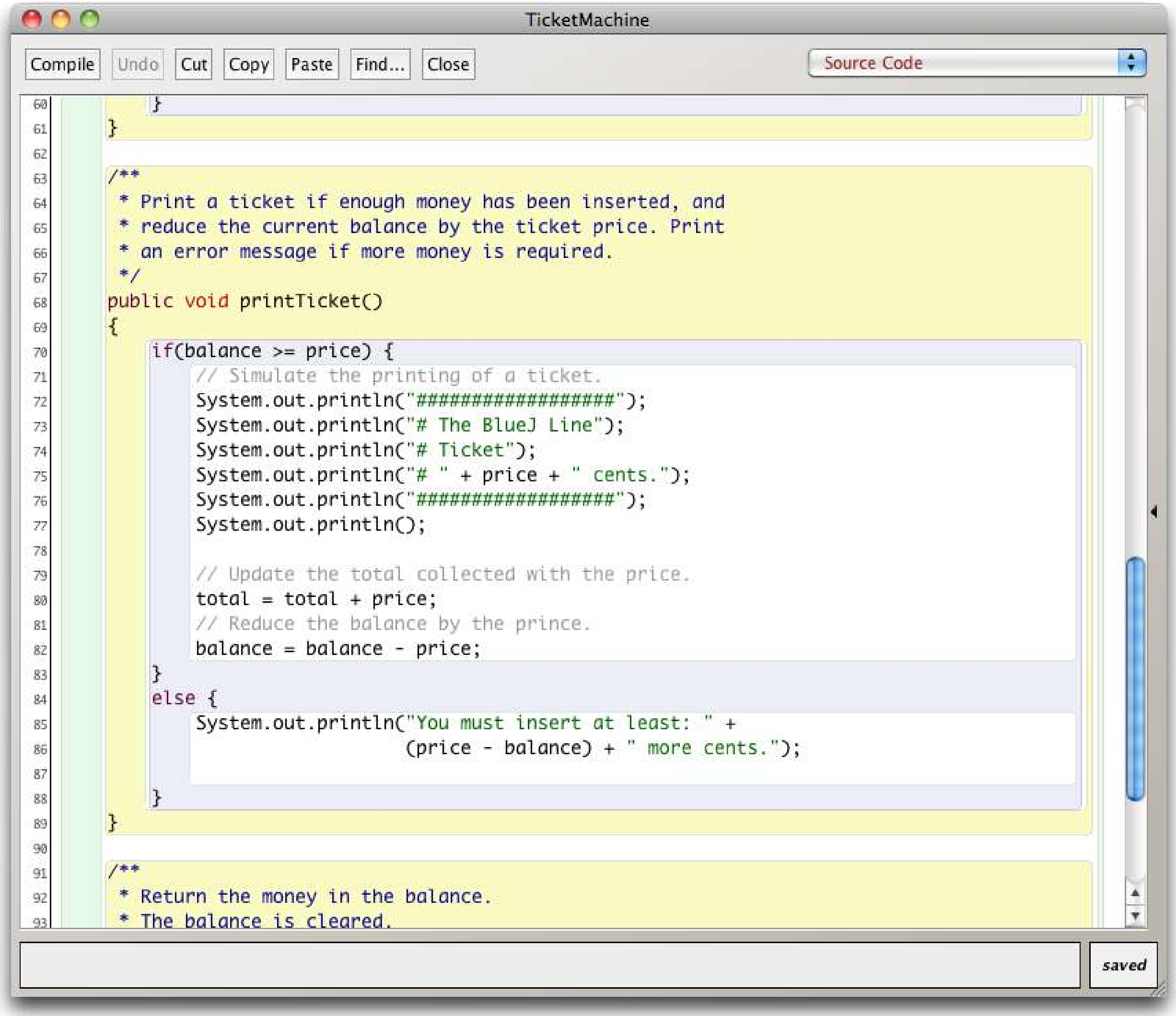
- Fields are one sort of variable.
  - They store values through the life of an object.
  - They are accessible throughout the class.
- Parameters are another sort of variable:
  - They receive values from outside the method.
  - They help a method complete its task.
  - Each call to the method receives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

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.



Scope highlighting



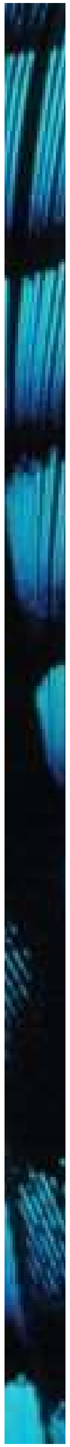
```
        }
    }

    /**
     * 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.
     */
    public 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 prince.
            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.
```



Scope and lifetime

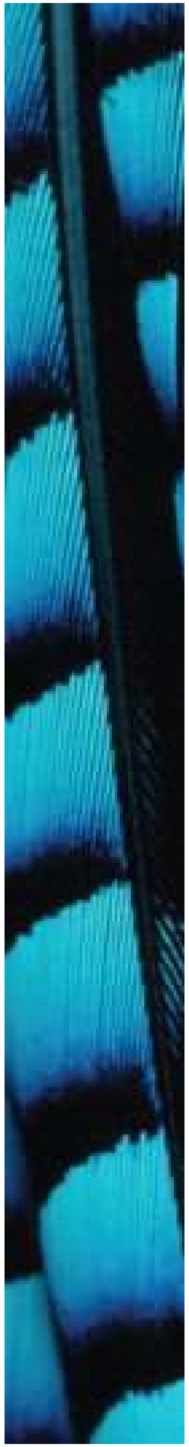
- Each block defines a new scope.
  - Class, method and statement.
- Scopes may be nested:
  - statement block inside another block inside a method body inside a class body.
- Scope is static (textual).
- Lifetime is dynamic (runtime).

Local variables

A local variable

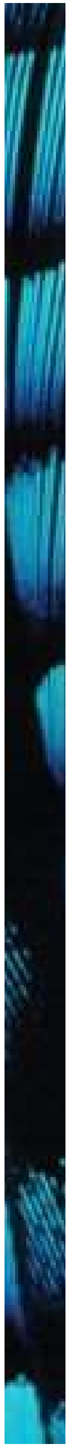
No visibility modifier

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



Scope and lifetime

- 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.
- The scope of a field is its whole class.
- The lifetime of a field is the lifetime of its containing object.



Review (1)

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

Review (2)

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

Review (3)

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

Review (4)

- ‘Correct’ behavior 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.