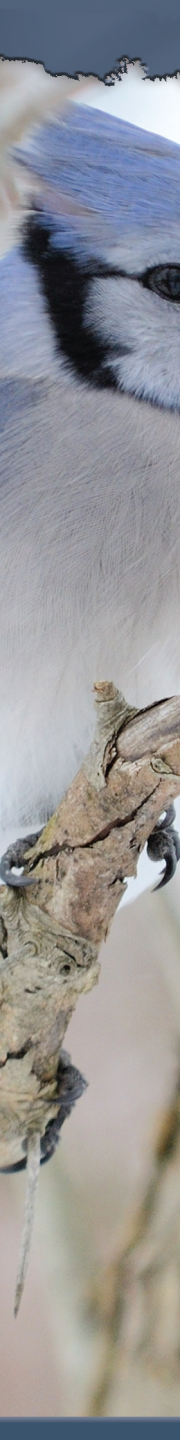
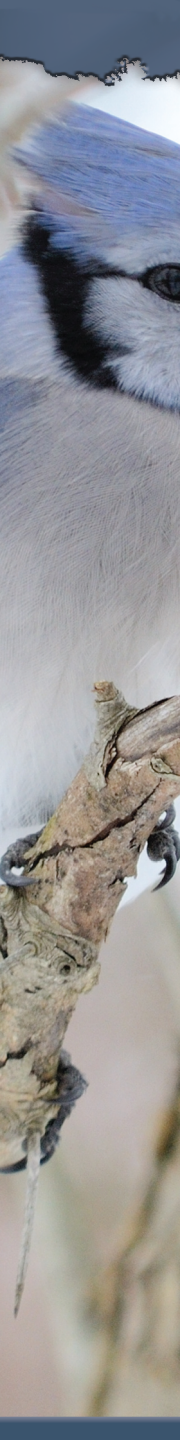


Understanding class definitions

Exploring source code



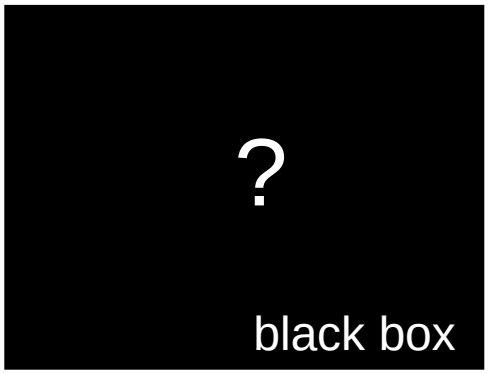
Ticket machine

Ticket machine

client view

Ticket machine

client view

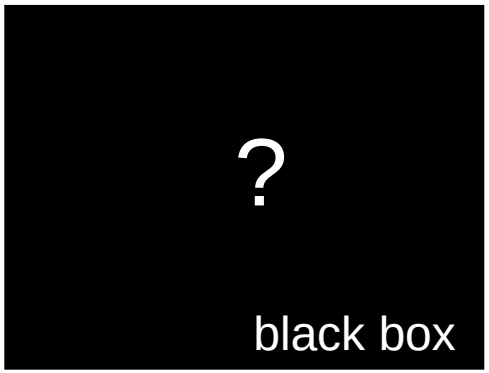


Ticket machine

client view

```
insertMoney

printTicket
```

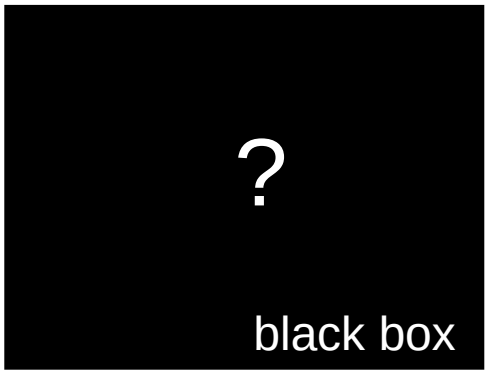


Ticket machine

client view

```
insertMoney

printTicket
```

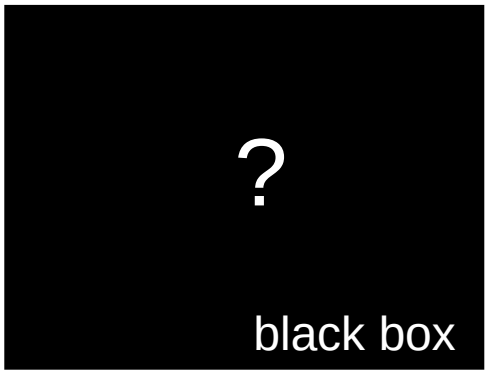


Ticket machine

client view

`insertMoney`

`printTicket`



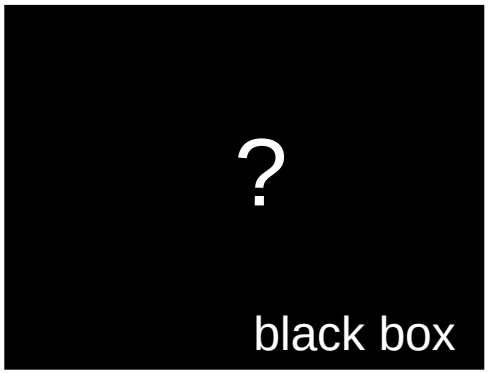

```
##################
# The BlueJ Line
# Ticket
# 10 cents.
##################
```

Ticket machine

client view

`insertMoney`

`printTicket`




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

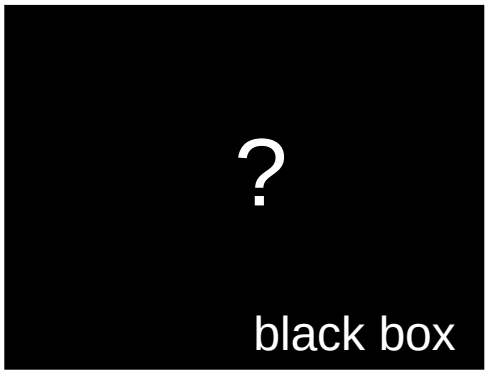
developer view

Ticket machine

client view

`insertMoney`

`printTicket`




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

white box

developer view

Basic class structure

```
package naiveticketmachine;
access class TicketMachine {
    Inner part omitted.
}
```

The outer wrapper of TicketMachine

```
package packagename;
access class ClassName {
    Fields
    Constructors
    Methods
}
```

package declaration

The inner contents of a class

```
access: _______ (default, package-private)
```

Basic class structure

```
package naiveticketmachine;
access class TicketMachine {
    Inner part omitted.
}
```

The outer wrapper of TicketMachine

```
package packagename;
access class ClassName {
    Fields
    Constructors
    Methods
}
```

package declaration

The inner contents of a class

```
access: _______ (default, package-private)
        public
```

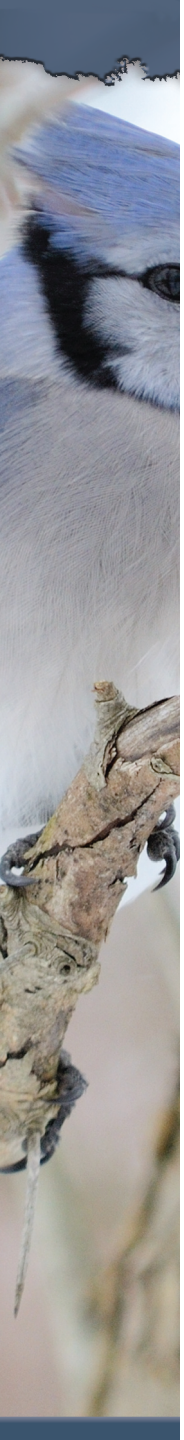
Class access

```
class TicketMachine {
    ...
}
```

Nothing, ie, package-private by default

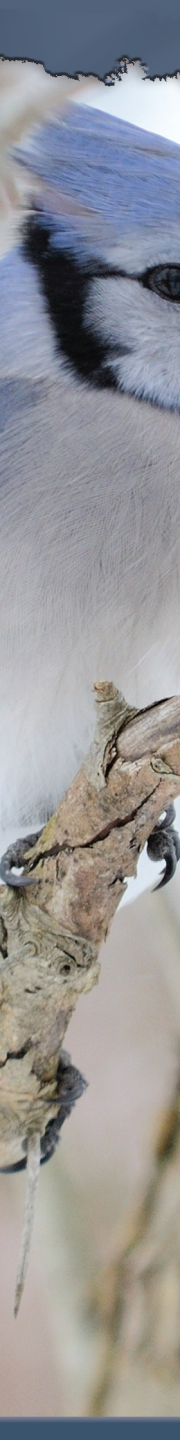
Need a *really* good reason to go public

```
public class TicketMachine {
    ...
}
```



Keywords

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



Fields

- Fields store values for an object.
- They are also known as *instance variables*, or *attributes*.
- Fields define the state of an object.
- Some values change often.
- Some change rarely

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

    Further details omitted.
}
```

access level, aka visibility modifier

type

variable name

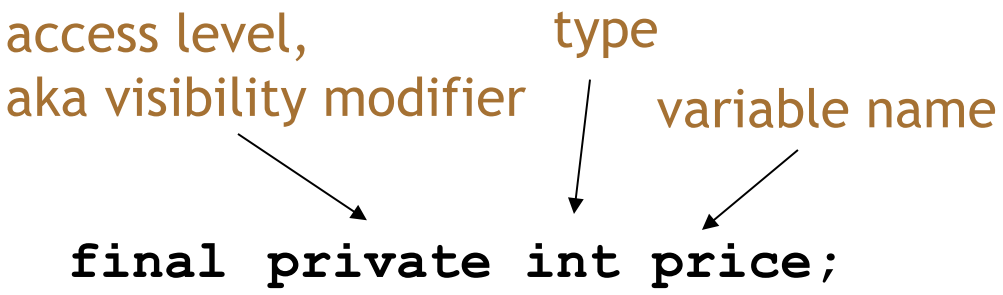
```
private int price;
```

Fields

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

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

    Further details omitted.
}
```

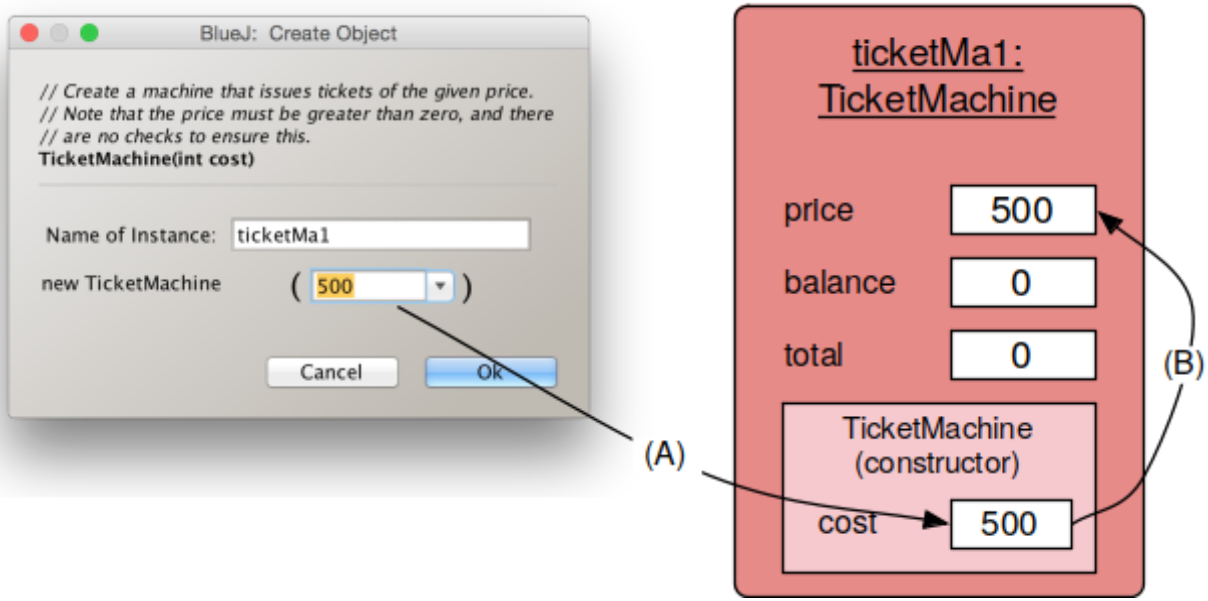


Constructors

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

- Initialize 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

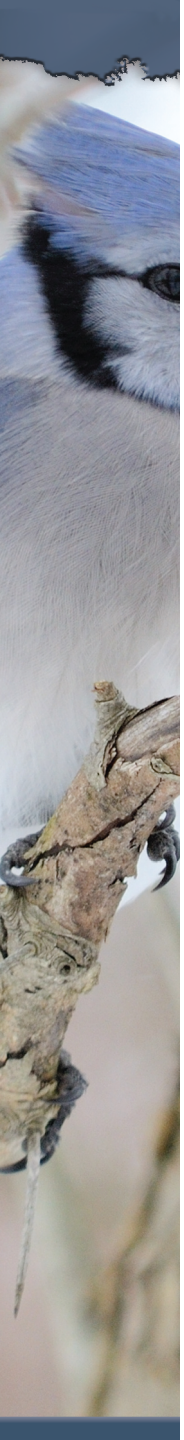
- Values are stored into fields (and other variables) via assignment statements:

  pattern

  - *variable* = *expression*;

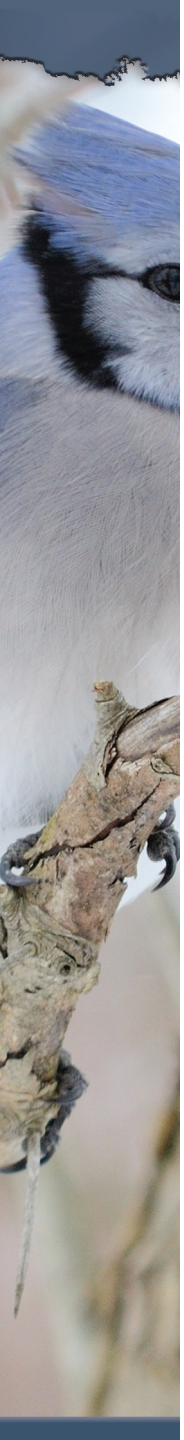
  example

  - `balance = balance + amount;`
  - `balance += amount;`
- A variable can store just one value, so any previous value is lost.



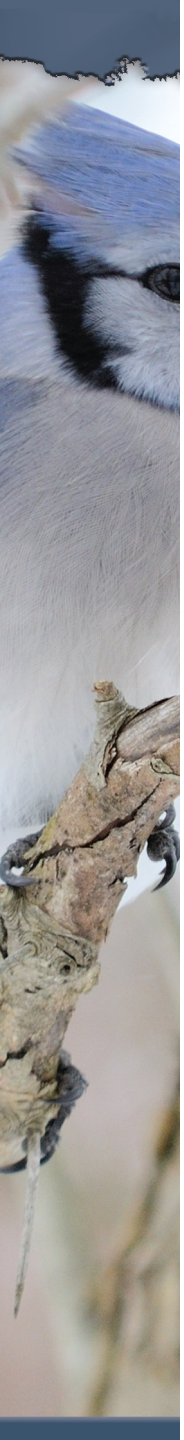
Choosing variable names

- 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`**



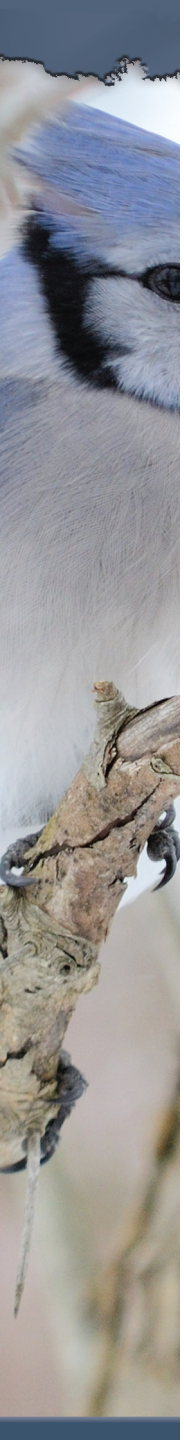
Methods

- 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

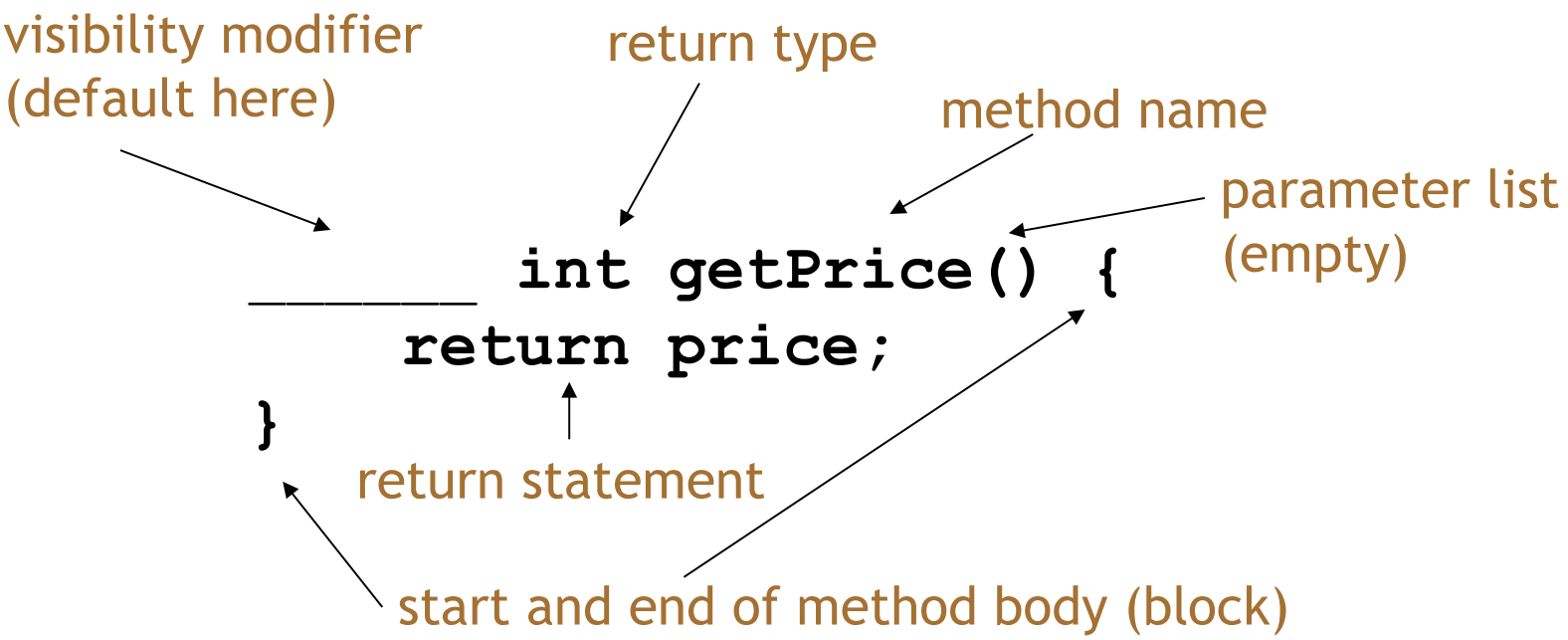
- The header:
  - *access* **`int getPrice()`**
- The header tells us:
  - the *visibility (access)* to objects;
    - **`private`**, package-private, **`public`**
  - whether the method *returns a result*;
  - the *name* of the method;
  - whether the method takes *parameters*.
- The body encloses the method's *statements*.

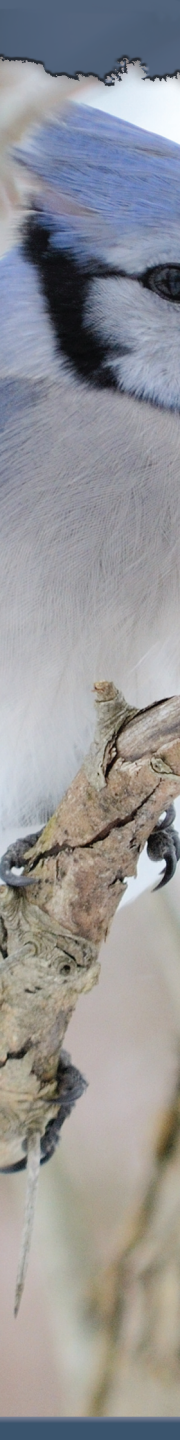


Accessor methods

- 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!

Accessor (`get`) methods

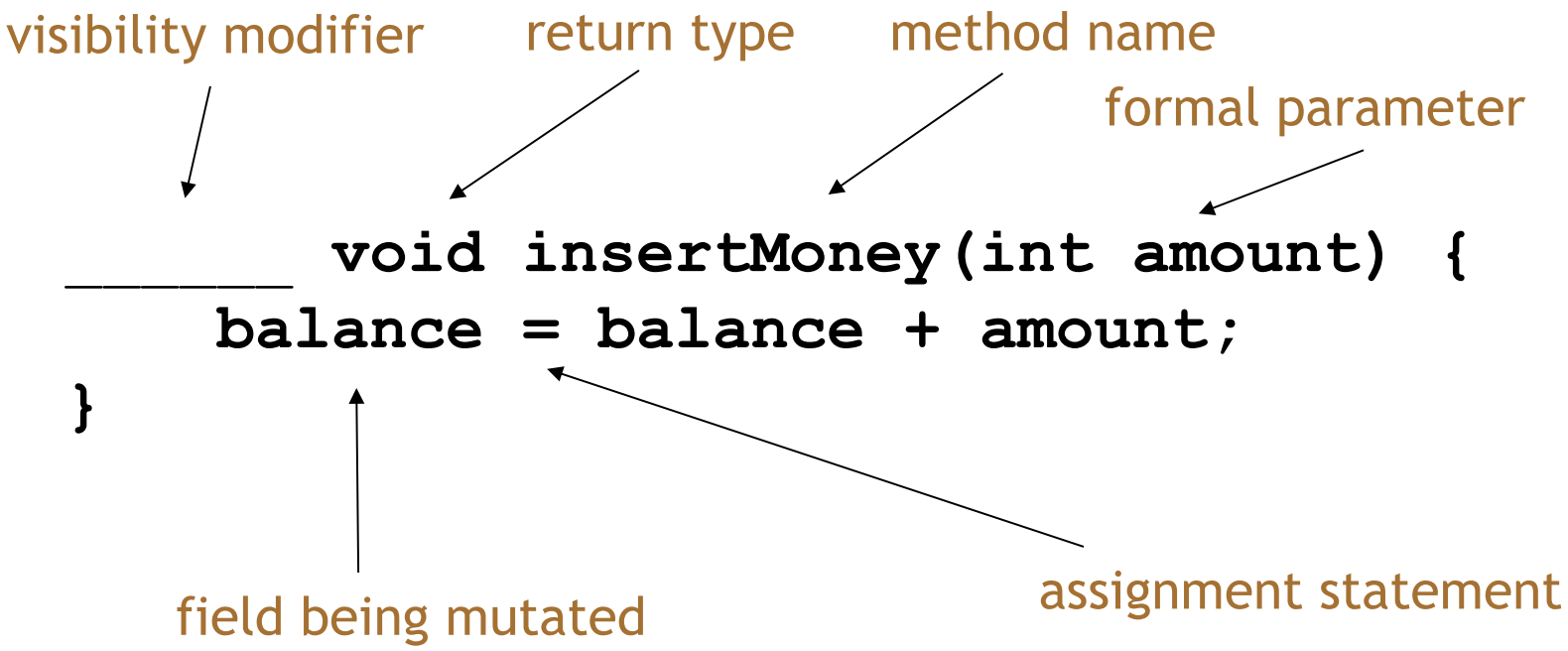


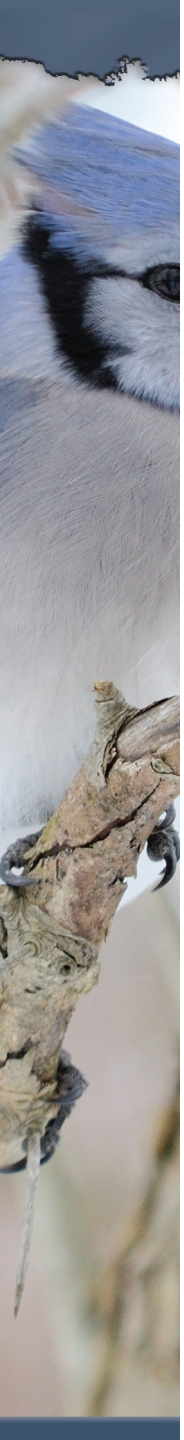


Mutator methods

- 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.

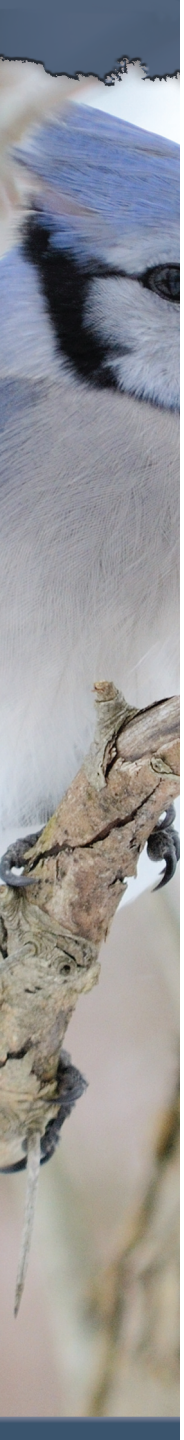
Mutator methods





`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



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 (sort of) support *encapsulation*.

Accessors and mutators - a word of caution

- Do not overuse accessors and mutators.
- They can break *encapsulation*.
  - We'll see more about this later on...
- They are generally considered evil… especially when added automagically by your favourite IDE.

Mutators can be harmful

- Bad. Why?

```
public class Car {
    private int speed;

    public int getSpeed() {
        return speed;
    }

    public void setSpeed(int speed) {
        this.speed = speed;
    }
}
```

Mutators can be harmful

- Bad. Why?

```
public class Car {
    private int speed;

    public int getSpeed() {
        return speed;
    }

    public void setSpeed(int speed) {
        this.speed = speed;
    }
}
```

```
// breaks encapsulation (and the car)
Car car = new Car();
int newSpeed = car.getSpeed() + 300;
car.setSpeed(newSpeed);
```

Protective mutators are better

```
public class Car {
    public static final int MAX_SPEED = 130;
    private int speed;

    public void setSpeed(int newspeed) {
        if (newSpeed < MAX_SPEED) {
            speed = newSpeed;
        }
    }
}
```

Protective mutators are better

```
public class Car {
    public static final int MAX_SPEED = 130;
    private int speed;

    public void setSpeed(int newspeed) {
        if (newSpeed < MAX_SPEED) {
            speed = newSpeed;
        }
    }
}
```

```
// no collateral damage
Car car = new Car();
car.setSpeed(3000);
```

Immutable classes

- Best, if possible – simple to use.
- Class state should be initialized in the constructor.
- Never changed afterwards.
  - No setters.
- Not always possible 😟.

static fields and methods

- Until you know what you're doing*, avoid declaring things static.
  - ~~`static`~~ `double someValue;`
  - ~~`static`~~ `int calc(double num);`
- Only use for

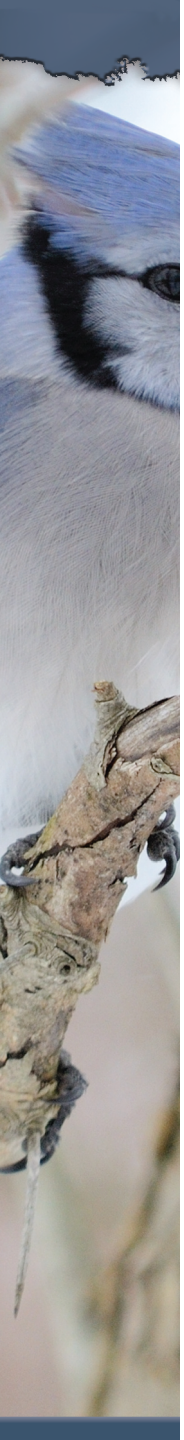
```
public static void main(String[] args)
```

* Seen later in course.

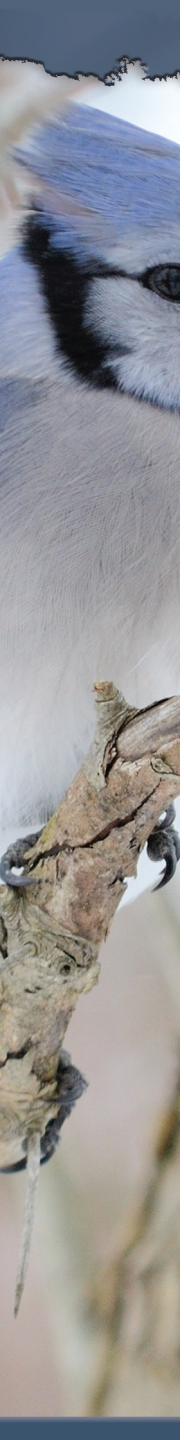
Printing from methods

```
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;
}
```

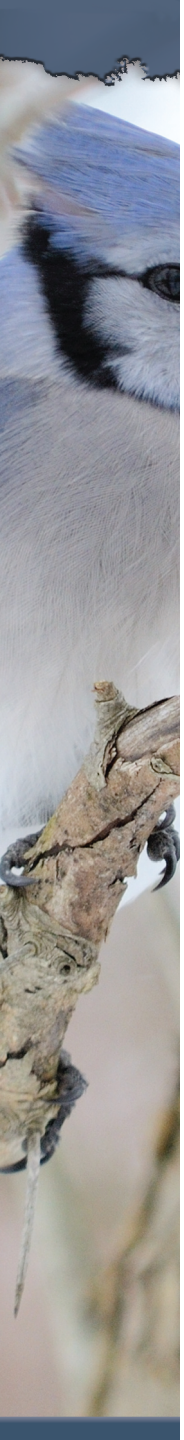


String concatenation



String concatenation

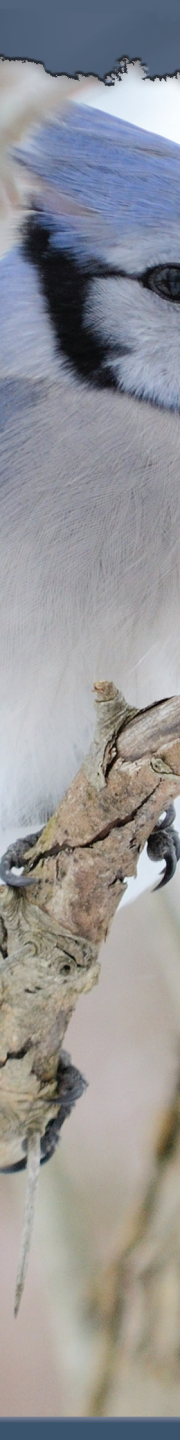
- 4 + 5



String concatenation

- 4 + 5

  9

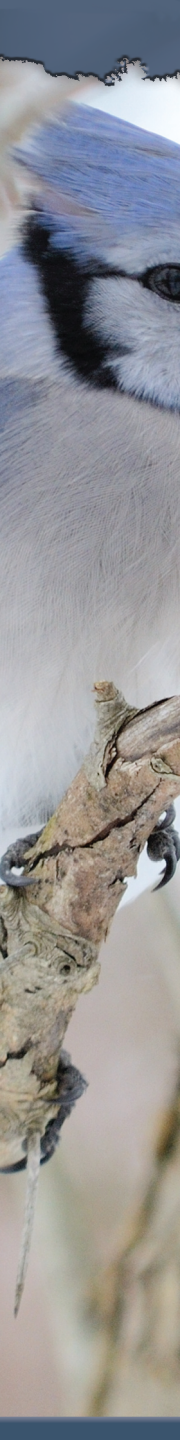


String concatenation

- 4 + 5

  9

- "wind" + "ow"



String concatenation

- 4 + 5

  9

- "wind" + "ow"

  "window"

String concatenation

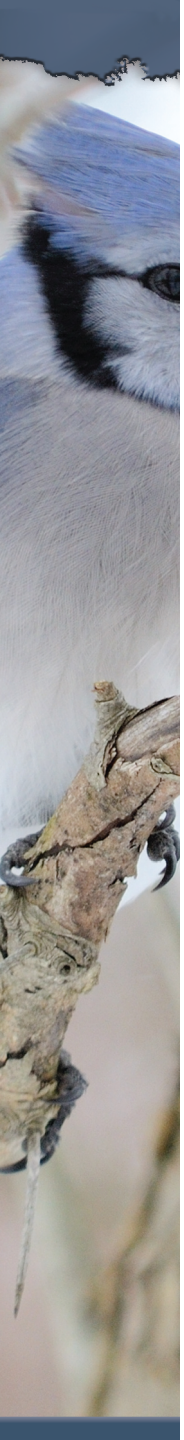
- 4 + 5

  9

- "wind" + "ow"

  "window"

- "Result: " + 6



String concatenation

- 4 + 5

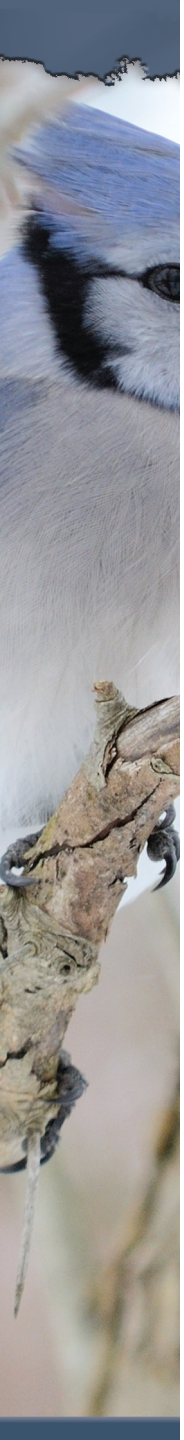
  9

- "wind" + "ow"

  "window"

- "Result: " + 6

  "Result: 6"



String concatenation

- 4 + 5

  9
- "wind" + "ow"

  "window"
- "Result: " + 6

  "Result: 6"
- "# " + price + " cents"

String concatenation

- 4 + 5

  9
- "wind" + "ow"

  "window"
- "Result: " + 6

  "Result: 6"
- "# " + price + " cents"

  "# 500 cents"

String concatenation

- 4 + 5

  9

- "wind" + "ow"

  "window"

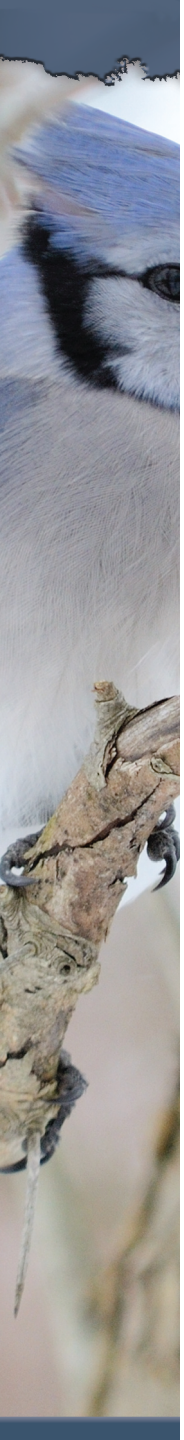
- "Result: " + 6

  "Result: 6"

- "# " + price + " cents"

  "# 500 cents"

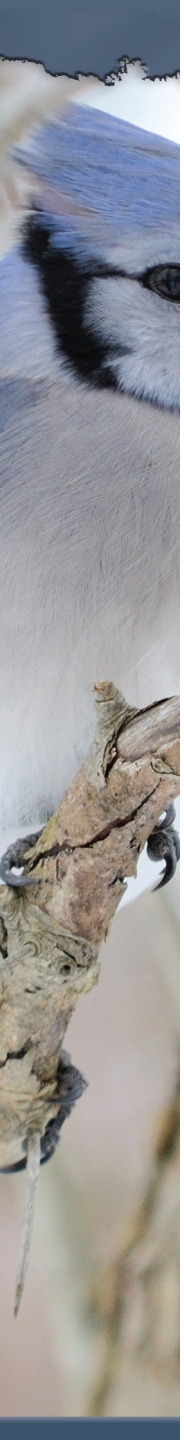
→ overloading +



Quiz

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

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

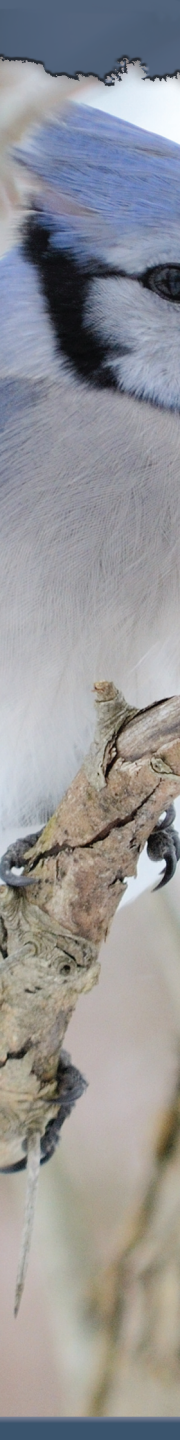


Quiz

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

  `11hello`

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



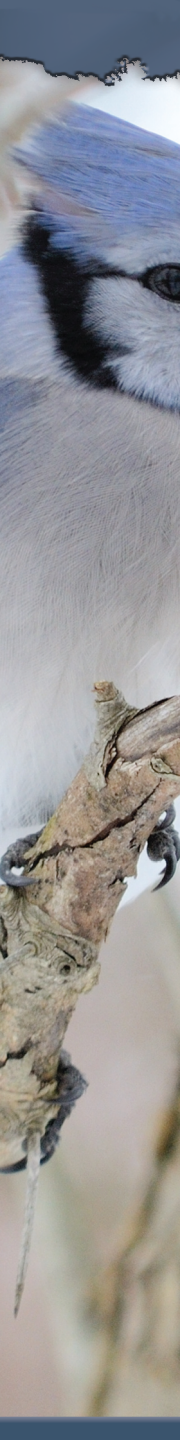
Quiz

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

```
11hello
```

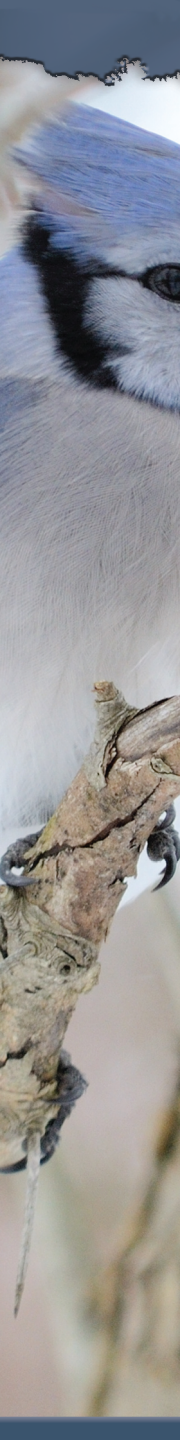
- System.out.println("hello" + 5 + 6);

```
hello56
```



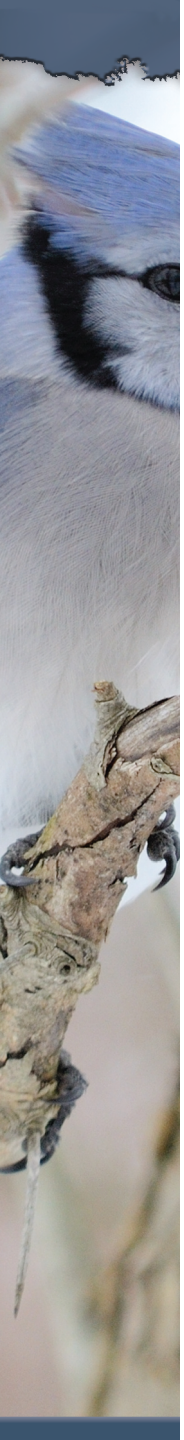
Method summary

- 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.



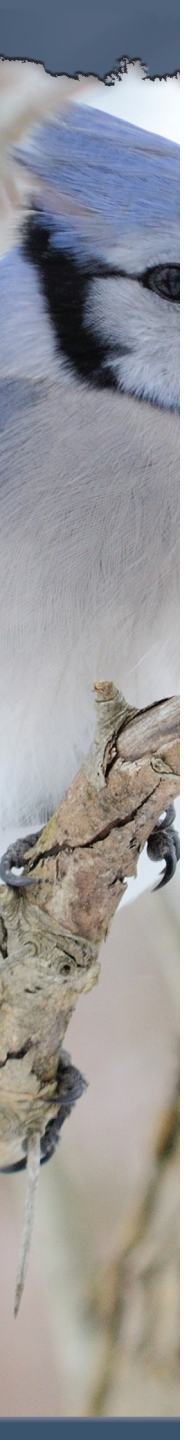
Reflecting on the ticket machines

- 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 the ability to choose between different courses of action.



Making choices in everyday life

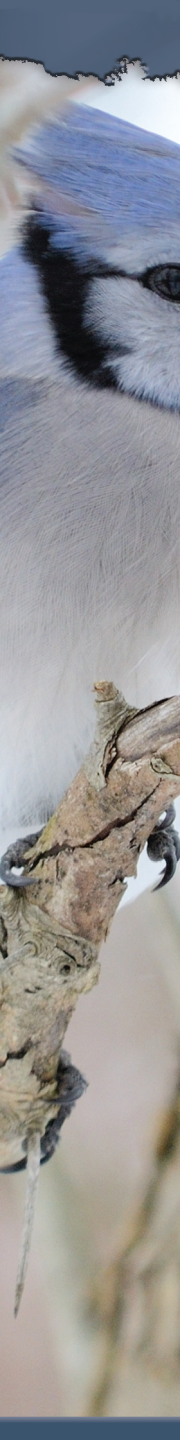
- If I have enough money left, then I will go out for a meal
- otherwise I will stay home and watch a movie.



Making a choice in everyday life

```
if (I have enough money left) {
    I will go out for a meal;
} else {
    I will 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
}
```

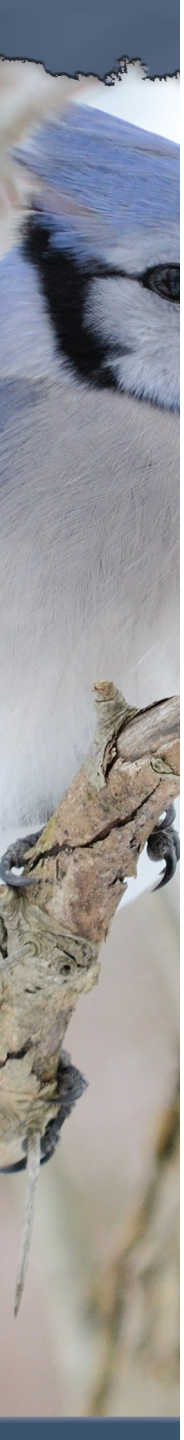
'else' keyword

actions if condition is false

Making a choice in the ticket machine

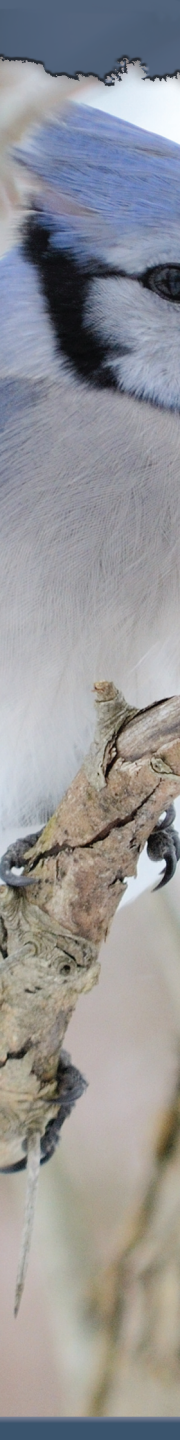
```
void insertMoney(int amount) {
    if (amount > 0) {
        balance = balance + amount;
    } else {
        System.out.println(
            "Use a positive amount: "
            + amount);
    }
}
```

conditional statement avoids an inappropriate action



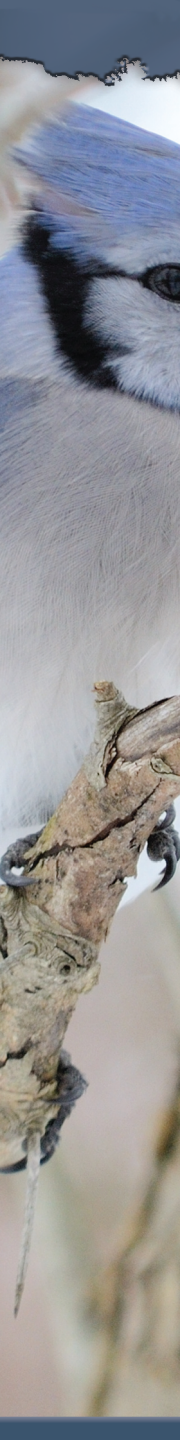
Variables – a recap

- 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.

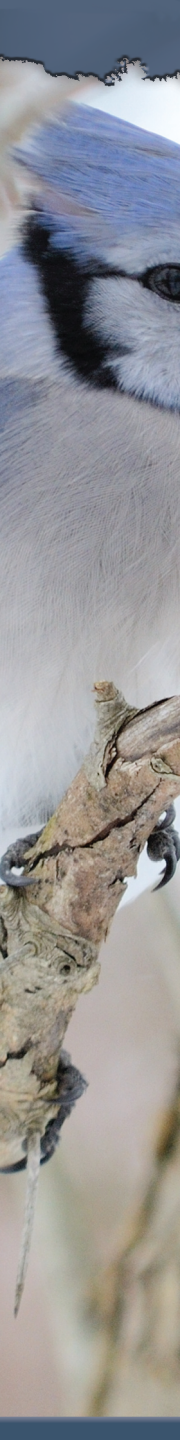


Scope and lifetime

- Each block defines a new *scope*.
  - Delimited by { }
  - Class, method and statement.
- Scopes may be nested:
  - statement block inside another block inside a method body inside a class body.
- Scope is static (compile-time).
- Lifetime is dynamic (runtime).



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



Unsuccessful attempt

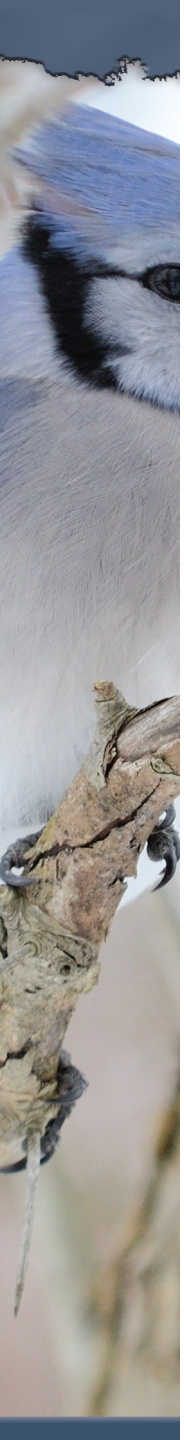
```
/**
 Clear and return balance.
 */
int refundBalance() {
    // Clear the balance.
    balance = 0;
    // Return the amount left.
    return balance;
}
```

It works, but it's not right.

Another unsuccessful attempt

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

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



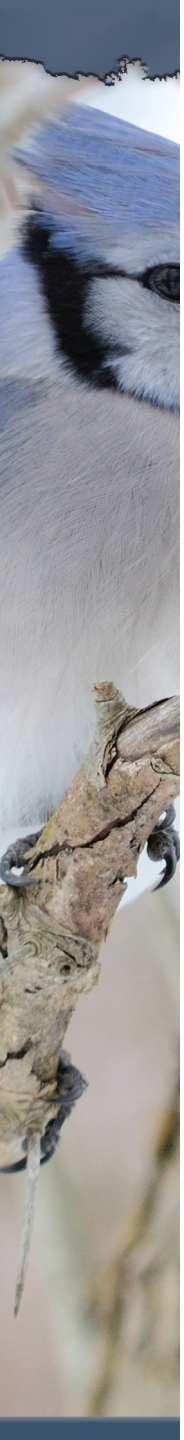
Local variables

- 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

A local variable

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



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.

Variable scope (visibility)

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

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

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

Variable scope (visibility)

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

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

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

attributes visible from all methods in class

Variable scope (visibility)

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

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

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

parameter **`cost`** only visible from constructor

Variable scope (visibility)

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

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

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

local variable **amountToRefund** only visible from method

Variable lifetime

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

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

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

Variable lifetime

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

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

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

attributes
exist throughout object lifetime

Variable lifetime

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

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

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

attributes
exist throughout object lifetime

parameter **`cost`**
only exists while constructor running

Variable lifetime

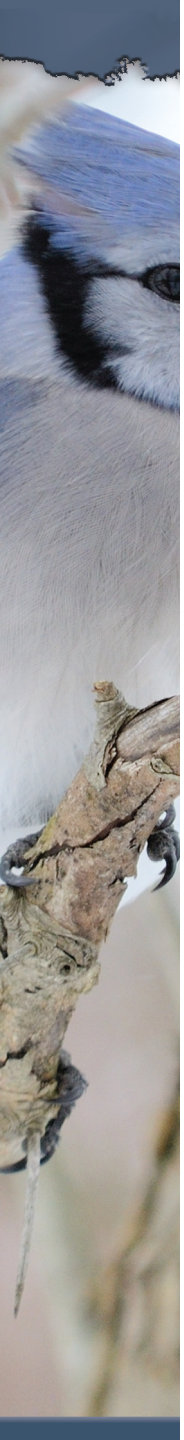
```
public final class TicketMachine {
    private final int price;
    private int balance;
    private int total;

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

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

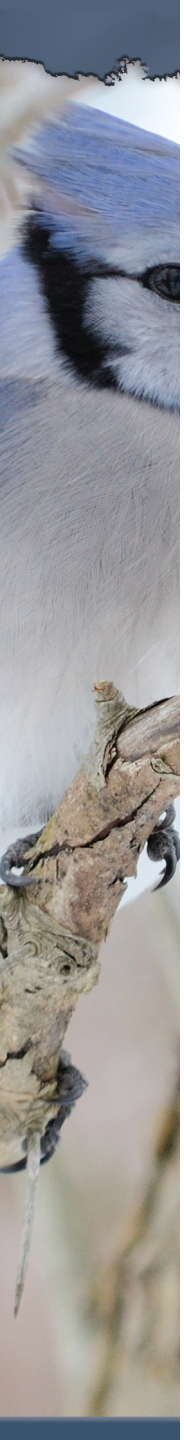
attributes
exist throughout object lifetime

local variable
amountToRefund
only exists while method running



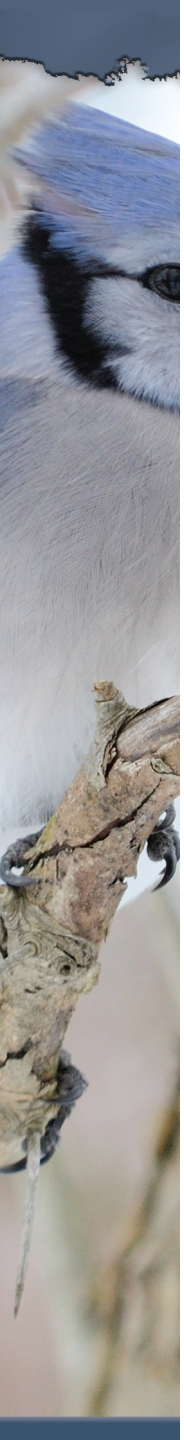
Review (1)

- 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.



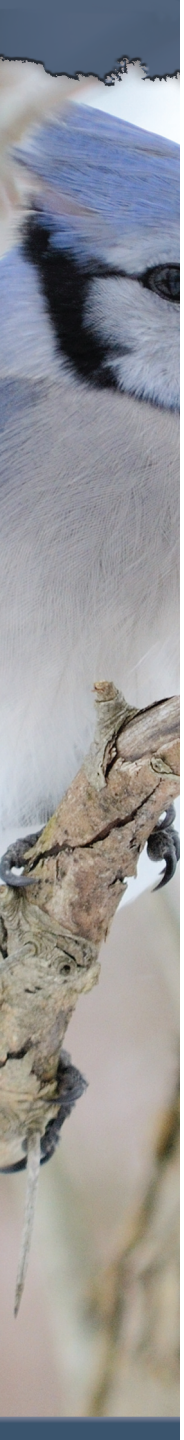
Review (2)

- Fields, parameters and local variables are all variables.
- Fields persist for the lifetime of an object.
- 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’ 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.

Review (5)

- Methods provide access to class fields.
- Think carefully before allowing methods to modify field values.
-  Beware of IDEs which automatically generate accessors and mutators.
- Immutable objects simplify code.