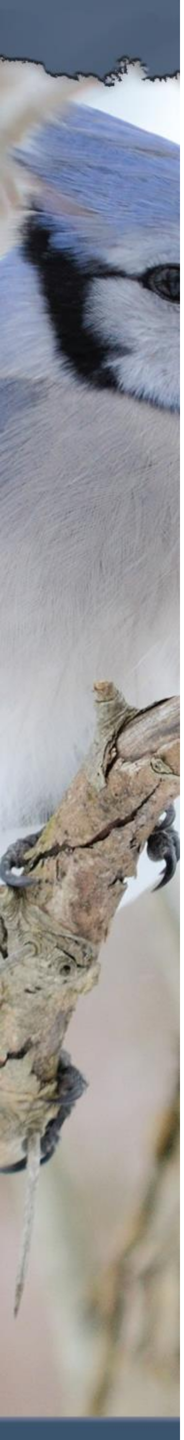
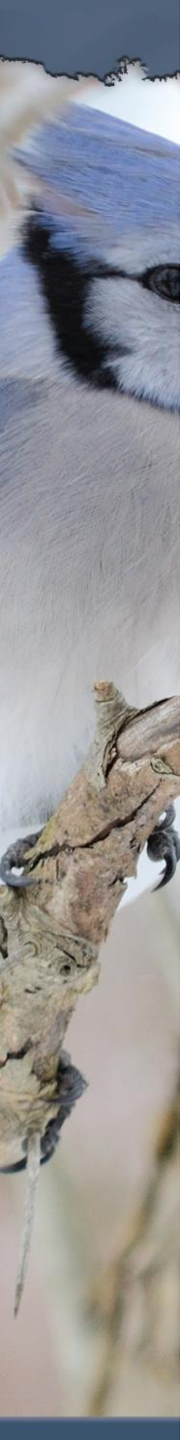


Understanding class and object definitions

Looking inside classes and exploring source code

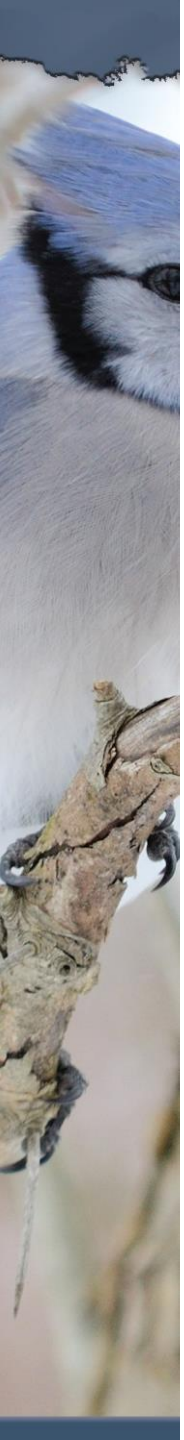


Classes and objects

- Fundamental to much of the early parts of this course
- Class: category or type of ‘thing’ (Like a template or blueprint)
- Object: belongs to a particular class and has individual characteristics
- Explore through BlueJ …

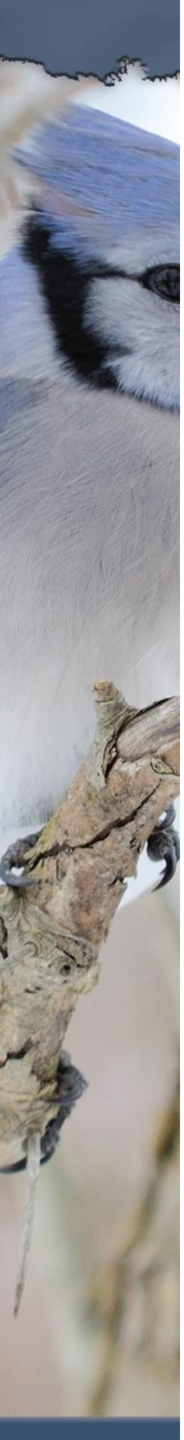
Classes and Objects

- Classes (noun)
  - Represents *ALL generic objects* of a similar kind or type
  - e.g. Car
- Objects (proper noun)
  - Represents *ONE specific thing* from the real world or some problem domain
  - e.g. THAT red car in the garage or YOUR green car in the parking lot



Methods and Parameters

- Methods (verbs)
  - Objects have operations which can be invoked on a specific object
  - e.g. *drive* the red car
- Parameters (adverbs)
  - Additional necessary information may be passed to the method to help with its execution
  - e.g. drive the red car *for 10 miles*



Other observations

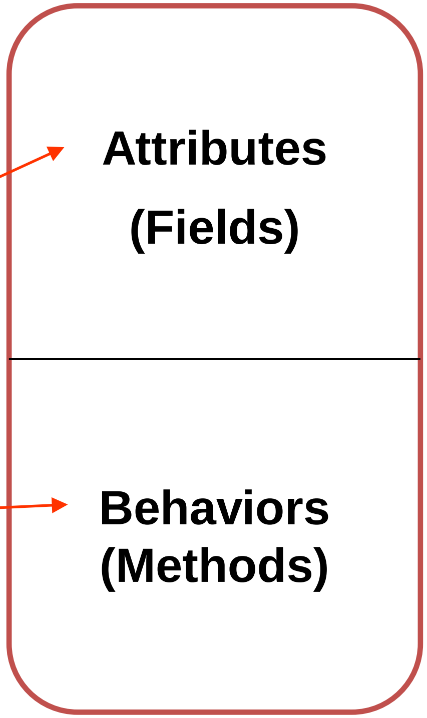
- Many distinct *instances* can be created from a single class
- An object has *attributes* that are values stored in *fields*
- The CLASS defines what FIELDS an object has
- But each OBJECT stores its own set of VALUES (the *state* of the object)

Definitions summary

Class

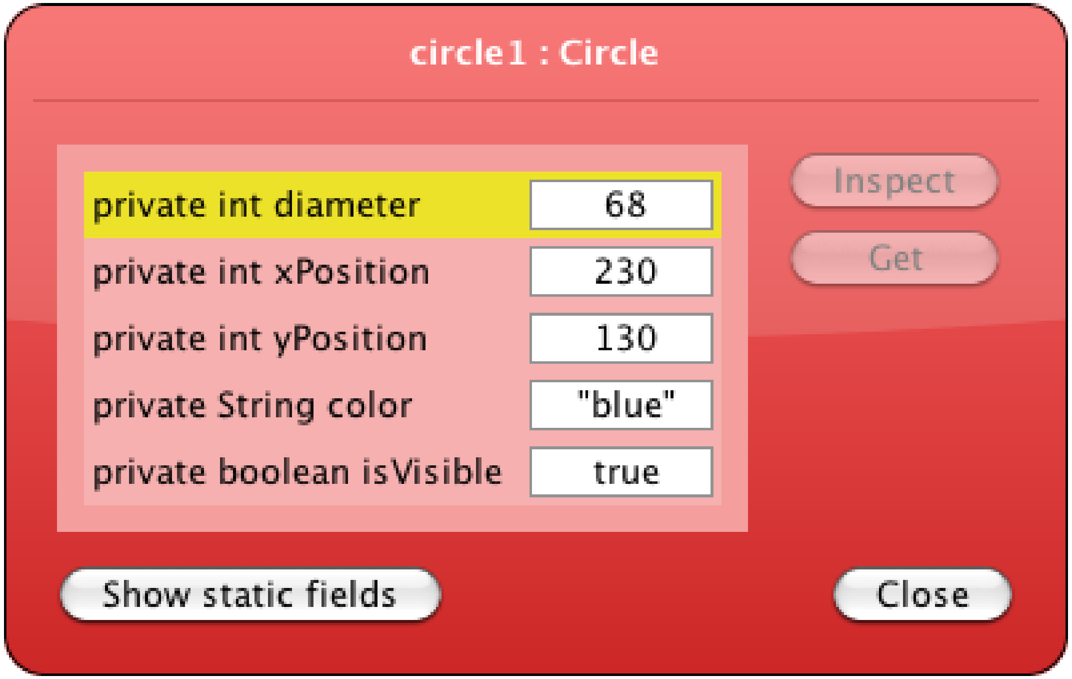
- A blueprint for objects of a particular type
- Defines the structure (number, types) of the attributes
- Defines available behaviors of its objects

Object



Demo of figures project

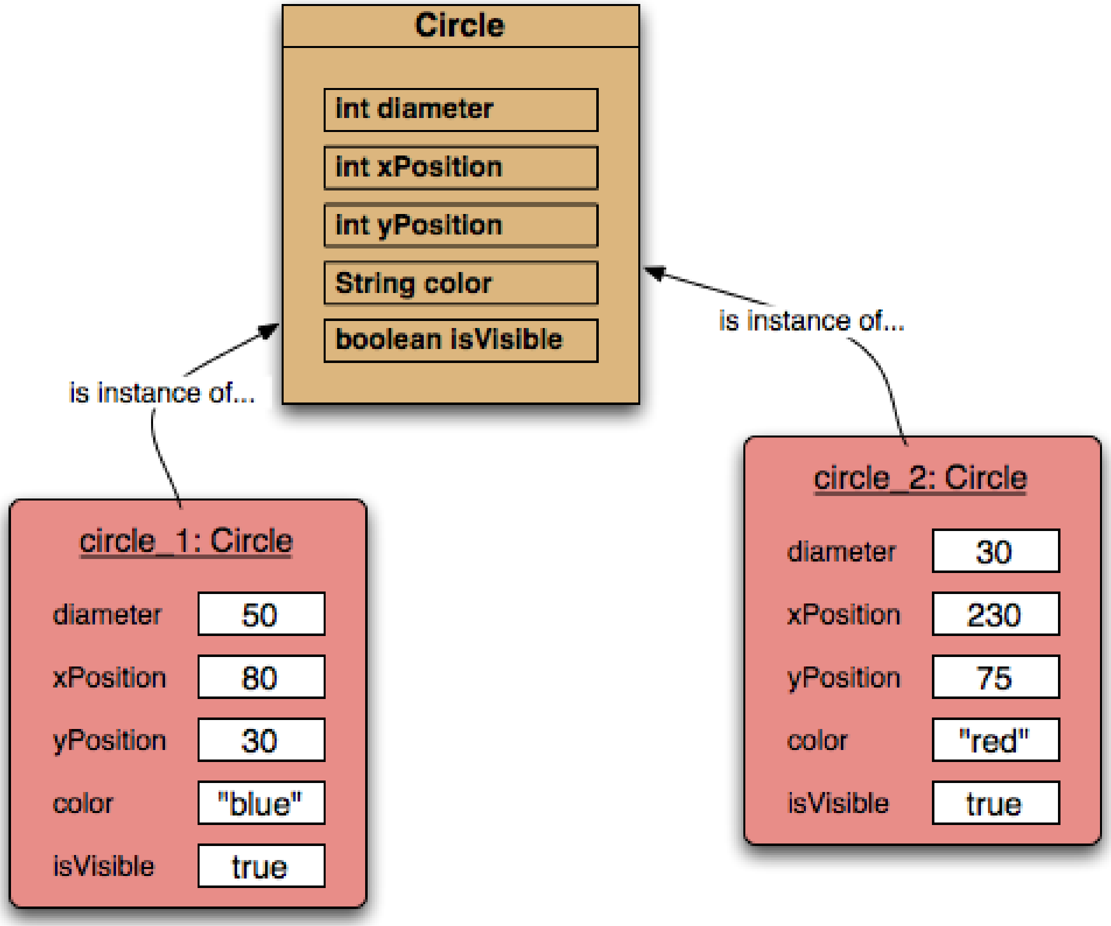
State



circle1 : Circle

private int diameter	68
private int xPosition	230
private int yPosition	130
private String color	"blue"
private boolean isVisible	true

Two circle objects



Circle

- int diameter
- int xPosition
- int yPosition
- String color
- boolean isVisible

is instance of...

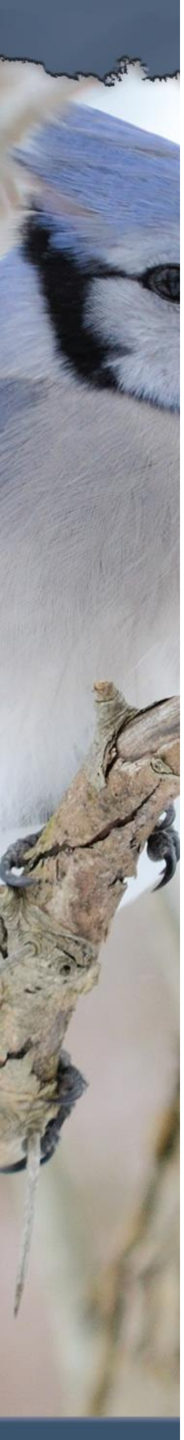
circle_1: Circle

diameter	50
xPosition	80
yPosition	30
color	"blue"
isVisible	true

is instance of...

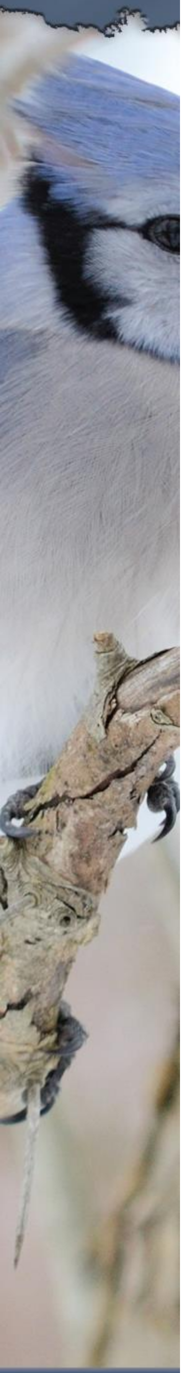
circle_2: Circle

diameter	30
xPosition	230
yPosition	75
color	"red"
isVisible	true



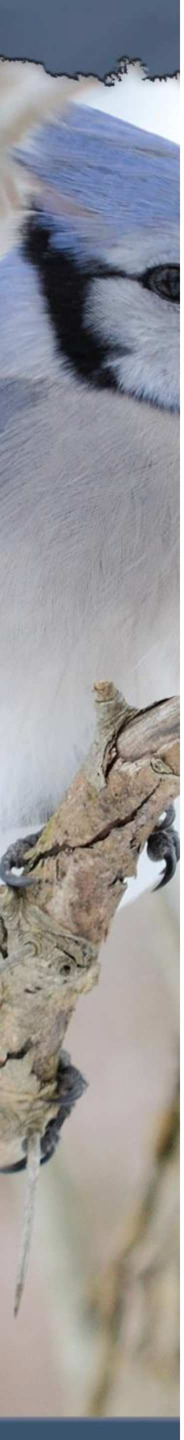
Return values

- All the methods in the *figures* project have **`void`** return types
- But methods may return a result via a return value that is not **`void`**
- Such methods will have a specific non-**`void`** return data type
- More on this in the next chapter



Ticket machines

Demo of
naïve-ticket-machine



Ticket machines – an external view

- Exploring the behavior of a typical ticket machine using *naive-ticket-machine* project that supplies tickets of a fixed price
  - How is that price determined?
  - How does a machine keep track of the money that is entered so far?
  - How does a machine keep track of the total amount of money collected?
  - How is ‘money’ entered into a machine?
  - How does the machine issue the ticket?

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

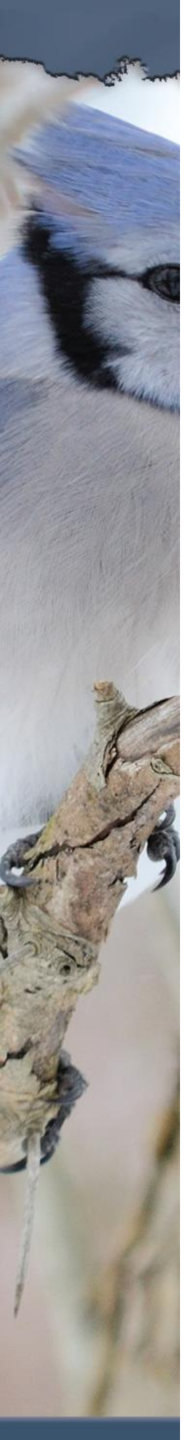
Basic class structure

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

The outer wrapper of TicketMachine

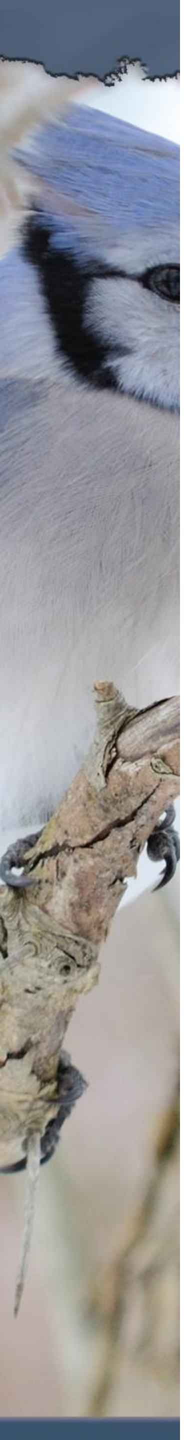
```
public class ClassName
{
      Fields

      Constructors

      Methods
}
```

The inner contents of a class



Keywords

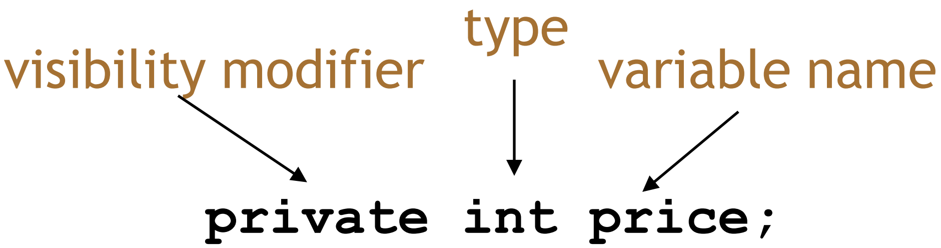
- Words with a special meaning in the language:
  - `public`
  - `class`
  - `private`
  - `int`
- Also known as *reserved words*
- Always entirely lower-case



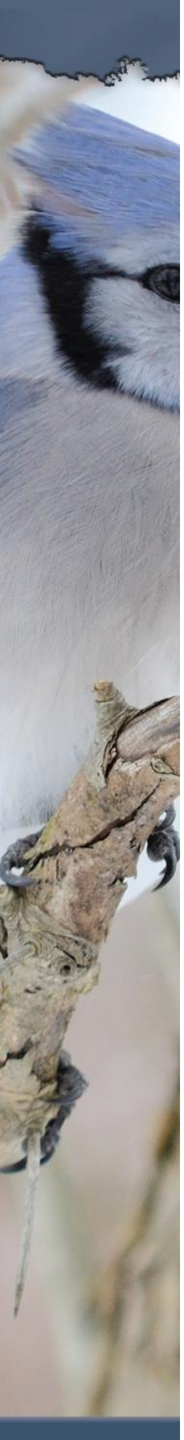
Fields

- Fields store *values* for an object
- They are also known as *instance variables*
- Fields define the *state* of an object
- Use *Inspect* in BlueJ 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.
}
```



```
private int price;
```

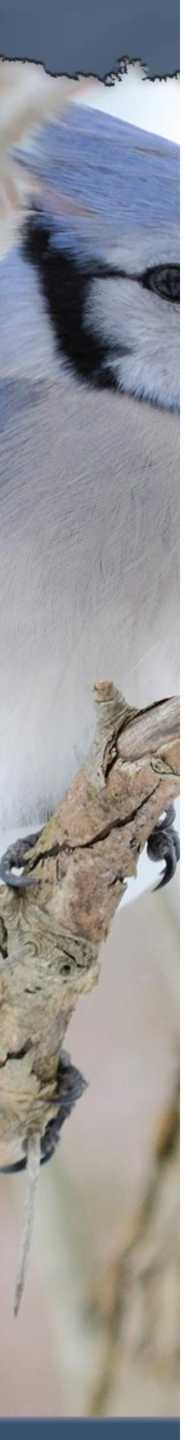


Visibility

- Private members
  - Can be accessed only by instances of same class
  - Provide concrete implementation / representation

- Public members
  - Can be accessed by any object
  - Provide abstract view (client-side)

- Protected members
  - Can be accessed by instances of the same class and its subclasses



Declaration with an access modifier

- Each class declaration that begins with the access modifier public must be stored in a file that has **exactly the same name** as the class and ends with the **.java** file-name extension.

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:
  - Initial values stored into the fields
  - Parameter values often used for these

Constructors (cont.)

- A constructor is a procedure for creating objects of the class.
- Keyword new requests memory from the system to store an object, then calls the corresponding class’s constructor to initialize the object.
- A constructor often initializes an object’s fields.
- Constructors do not have a return type (not even void) and they do not return a value.
- All constructors in a class have the same name — **the name of the class**.
- Constructors may take parameters.

Constructors (cont.)

- If a class has more than one constructor, they must have different numbers and/or types of parameters.
- Programmers often provide a “no-args” constructor that takes no parameters (a.k.a. *arguments*).
- If a programmer does not define any constructors, Java provides one default (no-args) constructor, which allocates memory and sets fields to the default values.

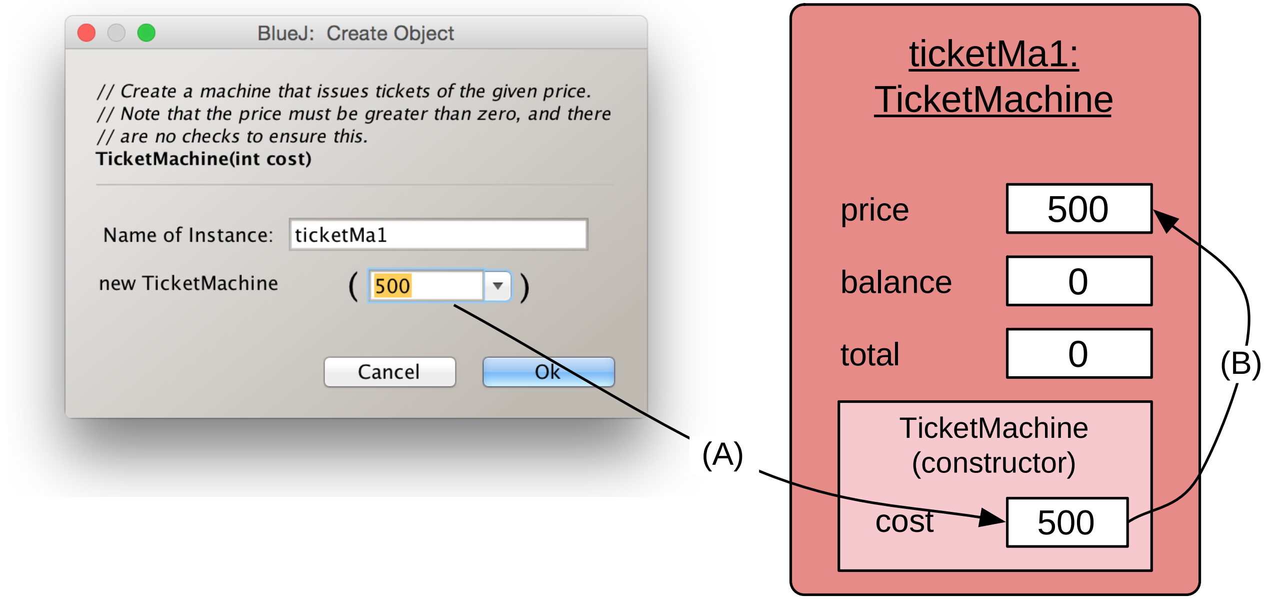
Constructors (cont.)

A nasty bug:

```
public class MyClass
 {
   ...
   // Constructor:
   public  void  MyClass (…)
   {
       ...
   }
   ...
```

Compiles fine, but the compiler thinks this is a method and uses **MyClass**’s default no-args constructor instead.

Passing data via parameters



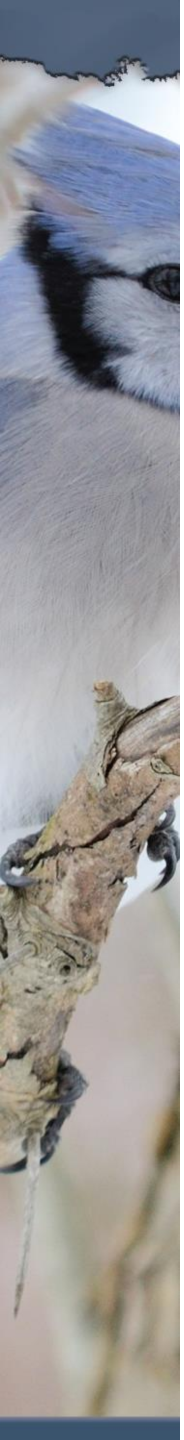
BlueJ: Create Object

```
// 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

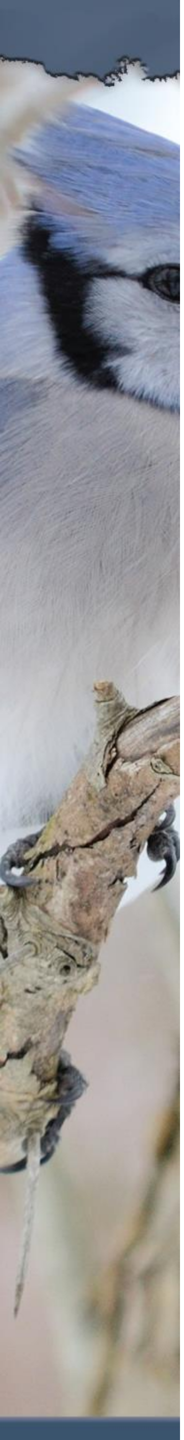
new TicketMachine (500)

Parameters **are another sort of variable**



Choosing variable names

- There is a lot of freedom over choice of names … so 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`**



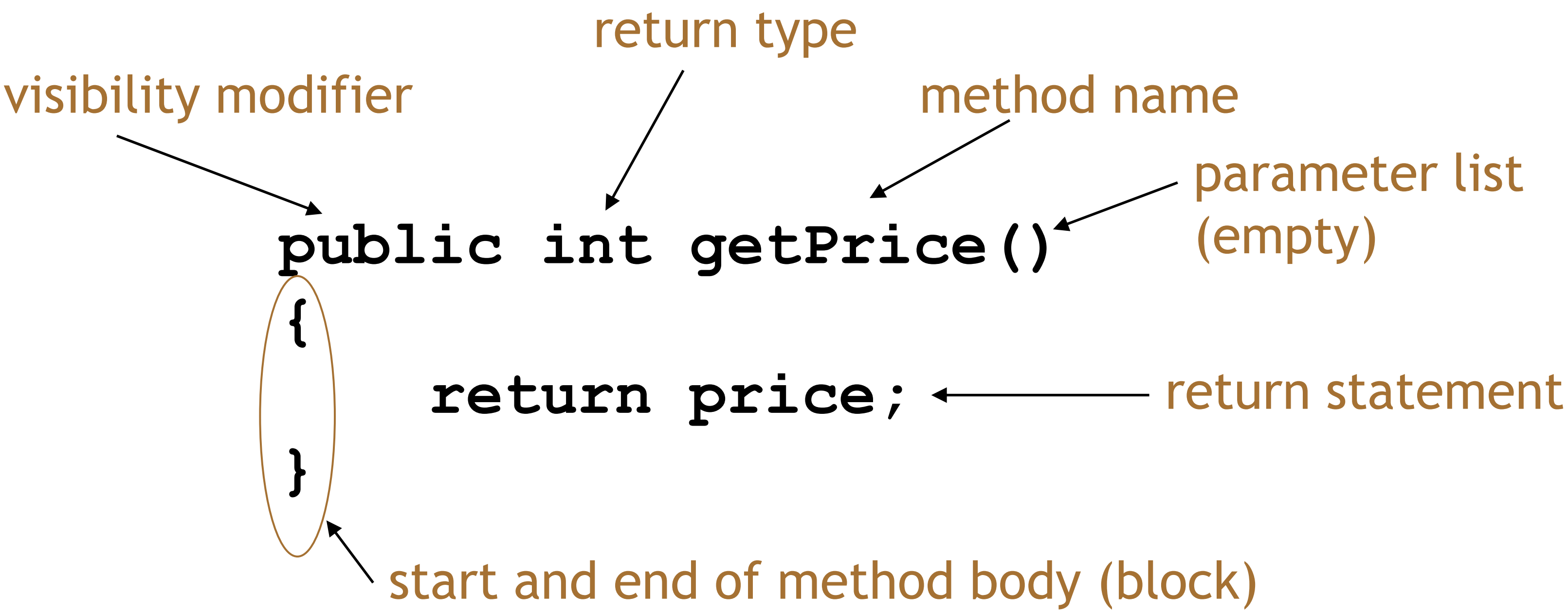
Methods

- 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 (e.g. Print methods)

Method structure

- The header provides the method’s *signature*:
  - **`public int getPrice()`**
- The header tells us:
  - the *visibility* to objects of other classes (e.g. public, private or protected)
  - whether the method *returns a result*
  - the *name* of the method
  - whether the method takes *parameters*
- The body encloses the method’ s *statements* within curly braces { }

Accessor (`get`) methods



return type

visibility modifier

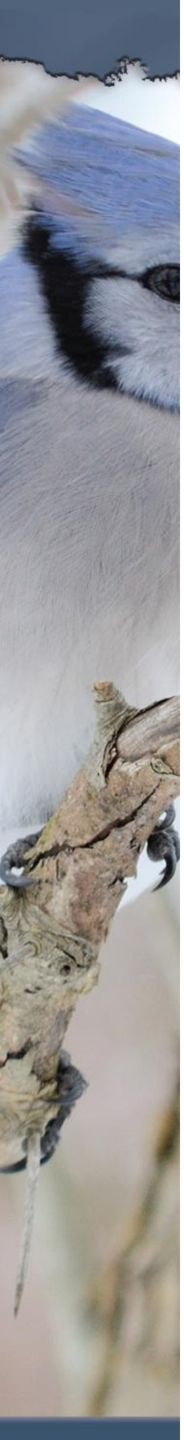
method name

parameter list (empty)

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

return statement

start and end of method body (block)



Test

```
public class CokeMachine
{
   private price;

   public CokeMachine()
   {
      price = 300
   }

   public int getPrice
   {
      return Price;
   }
```

- What is wrong here?

(there are five errors!)

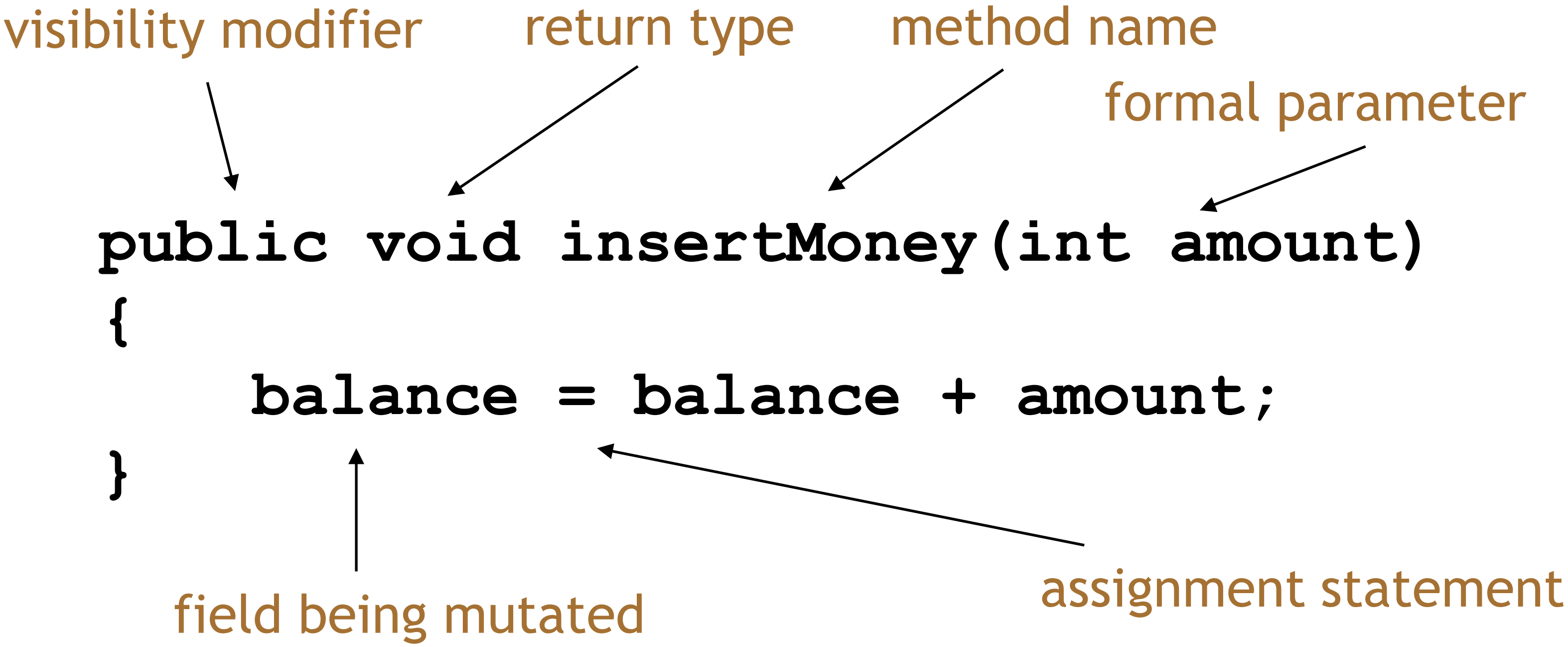
Test

```
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



visibility modifier

return type

method name

formal parameter

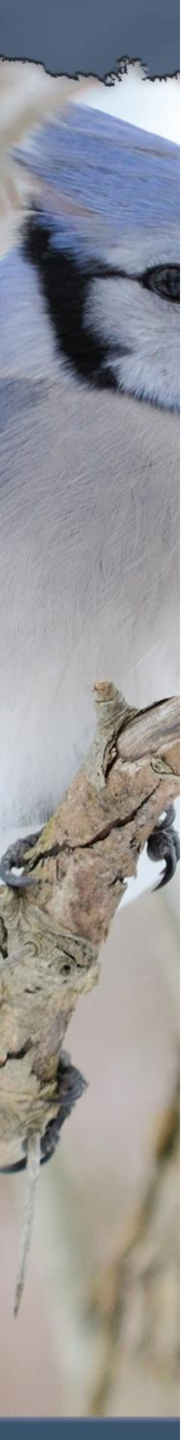
```
public void insertMoney(int amount)
{
    balance = balance + amount;
}
```

field being mutated

assignment statement

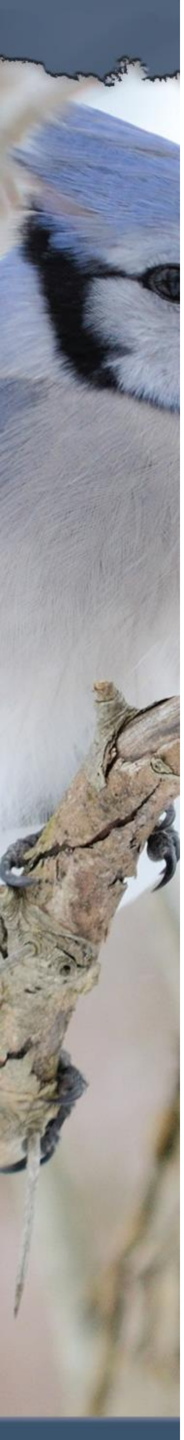
Compound assignment operators (e.g. +=, -=, *=, /=)

```
balance += amount;
```



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*

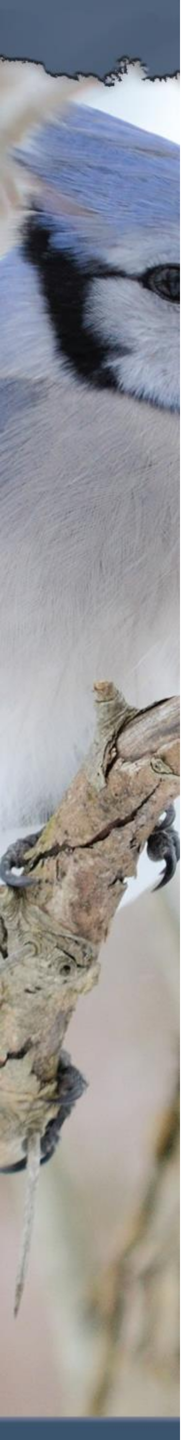


Printing from methods

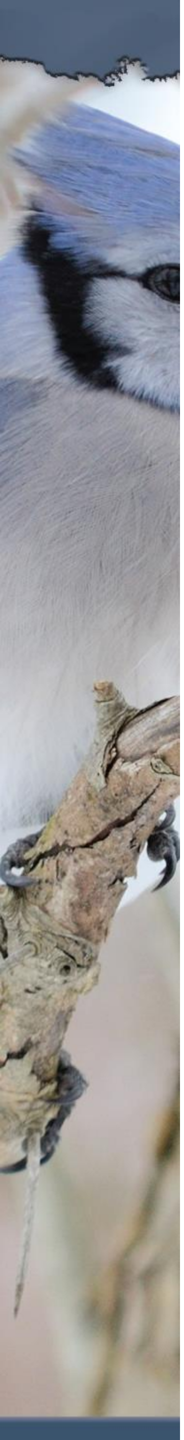
```
public void printTicket()
{
    // Simulate the printing of a ticket.
    System.out.println("##################");
    System.out.println("# The BlueJ Line");
    System.out.println("# Ticket");
    System.out.println("# " + price + " cents.");
    System.out.println("##################");
    System.out.println();

    // Update the total collected with the balance.
    total = total + balance;

    // Clear the balance.
    balance = 0;
}
```

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

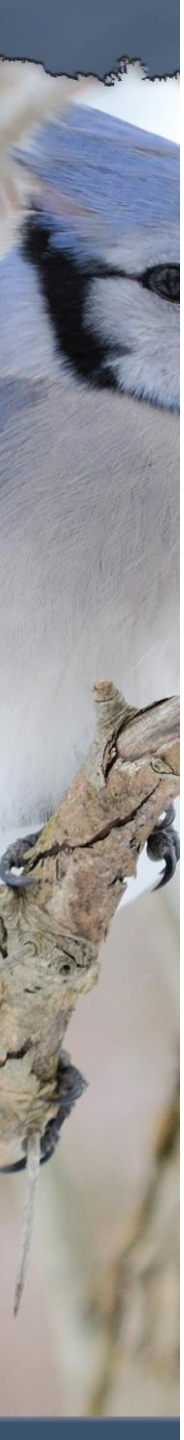


How do we write a method to ‘refund’ an excess balance?



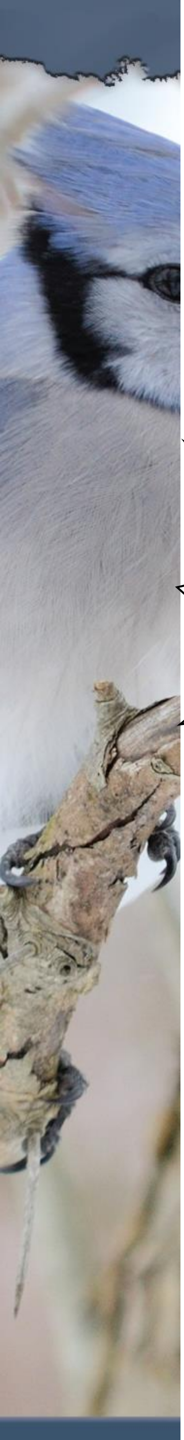
Variables – a recap

- 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 just like parameters
  - But MUST be *declared* within the method first
  - Unlike parameters which receives external values, the method MUST *set* their values
  - Used for temporary calculation and storage
  - Exist only as long as method is being executed
  - ONLY accessible from within declared code block
  - ONLY defined within a particular *scope*
  - Storage and values will DISAPPEAR after the method call is completed
  - May NOT be accessed outside of the method



Local variables

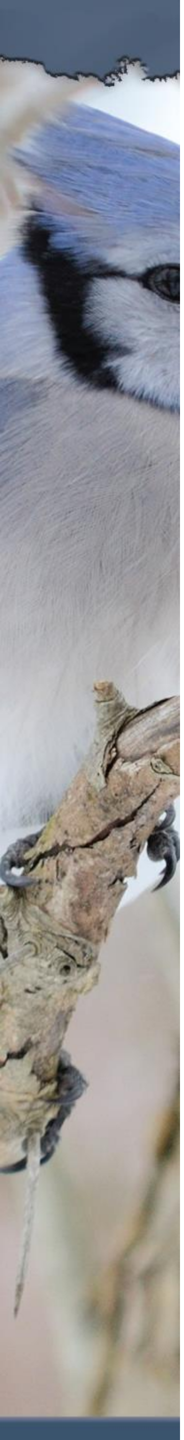
A local variable

No visibility modifier

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

Replace declaration & assignment with:

```
int amountToRefund = balance;
```



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)

Scope and lifetime of variables

- Fields
  - Scope: the entire *class* in which it was defined
  - Lifetime: existence time of its containing object
- Parameters
  - Scope: *method/constructor* which it is declared
  - Lifetime: execution time of *method/constructor* in which it was declared/passed into
- Local variables
  - Scope: the *code block* in which it was declared
  - Lifetime: the execution time of the *code block* in which it was declared and initialized in