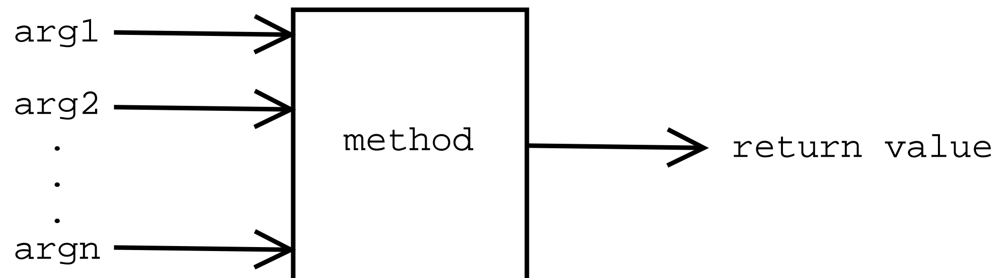
Announcements

- Tutorial today from 2:30pm to 4:00pm at Rutherford Physics (RPHYS) 118
- Midterm, October 20th at 7:00 pm at McConnell 204 (it may change)
- Conflicts:
  - PHAR 300 at 5:45pm at McConnell 103
  - MATH 223 at 8:15pm at McConnell 103

Methods as functions

- Methods can be viewed as a "black box" with inputs and outputs:



- There are three kinds of methods:
  - Mutators: Modify the state of objects,
  - Accessors: Return information about the object,
  - Constructors: Initialize a newly created object.
  - "None of the above": do not change the state or return information, but perform other actions such as sending messages to other objects.

Constructors

- Special methods, whose syntax is given by

```
class_name(list_of_arguments)
{
    statements;
}
```

- For example:

```
public class Student {
    //...
    Student(String n, long i)
    {
        name = n;
        id = i;
    }
    //...
}
```

Constructors (contd.)

- A constructor method gets executed when a new object of the class gets created using the new keyword. Therefore, the general syntax for the expression used to create objects is:

  new *class_name*(*list_of_actual_arguments*);

- For example

```
Student al;
al = new Student(“Alan Turing”, 11001122331);
```

Method calls in context

- There are two forms of method calls:
  - Method call as a statement
  - Method call as an expression
- A method call is a statement if its return type is `void`, otherwise it is an expression.
- If a method call is an expression, it must appear in a context that allows expressions, such as:

  A. the right hand-side of an assignment:

  ```
  long n = dave.get_id();
  String s = dave.get_program();
  ```

  B. ...or, the argument of another method:

  ```
  System.out.println(dave.get_id());
  bert.set_id(dave.get_id());
  ```

- But the types **must** match!

Program Structure

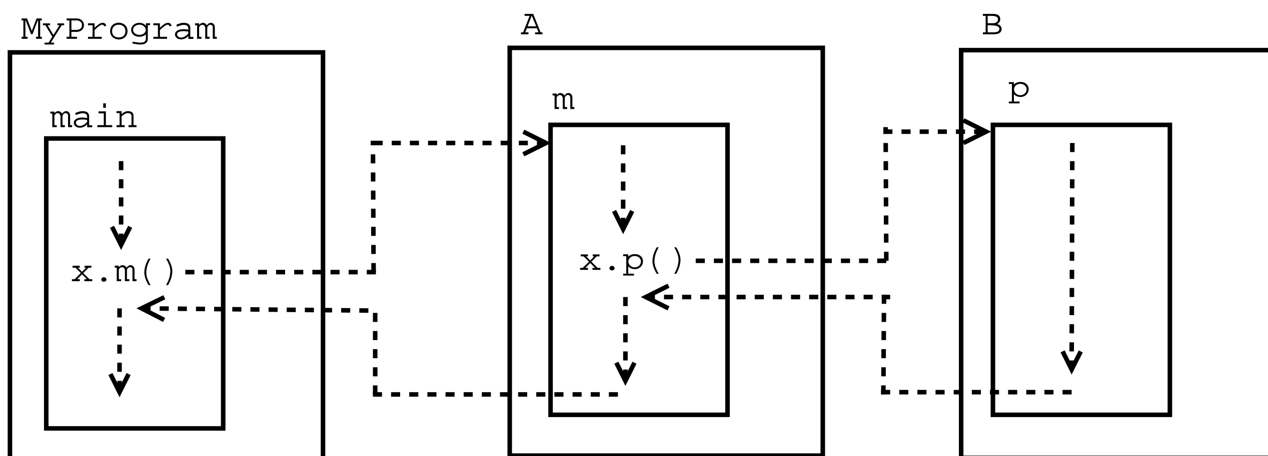
```
public class MyProgram {
  public static void main(String[] args)
  {
    //...
  }
}

public class A {
  //...
}

public class B {
  //...
}
```

McGill

Method invocation: control flow

```
public class MyProgram {
  public static void main(String[] args)
  {
    A x = new A();
    x.m();
    System.out.println(‘‘Main done’’);
  }
}
public class A {
  void m()
  {
    B x = new B();
    x.p();
    System.out.println("m done");
  }
}
public class B {
  void p()
  {
    System.out.println(‘‘Do something’’);
  }
}
```

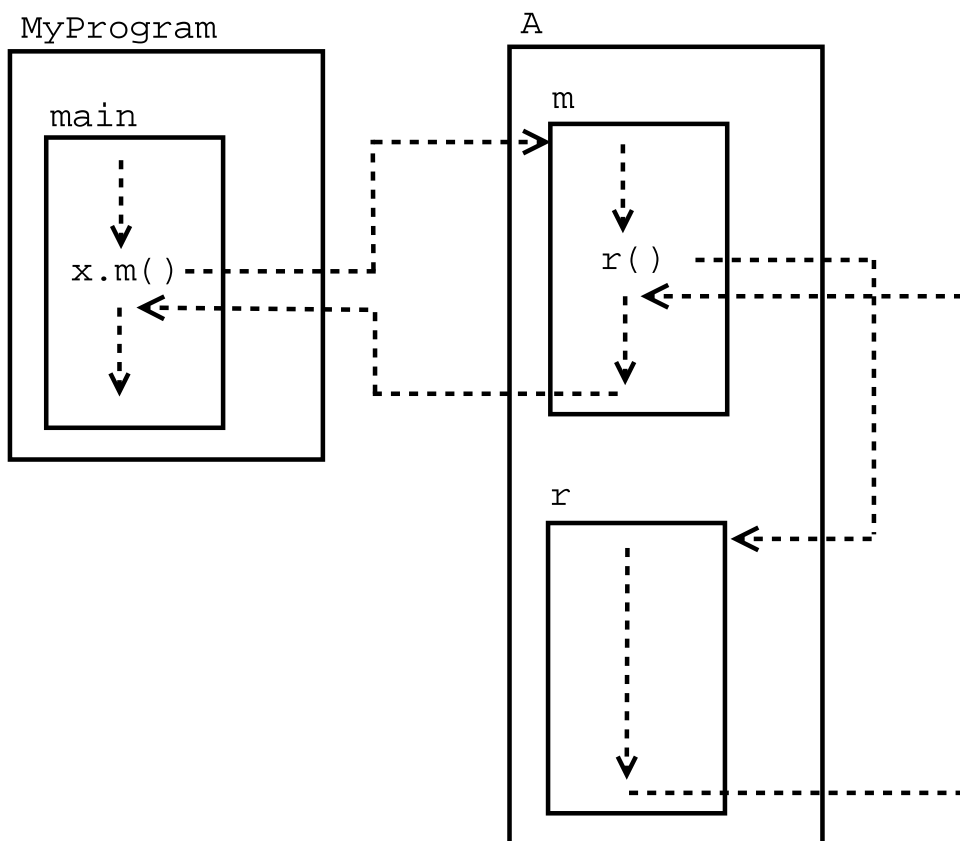
Method invocation: control flow



Method invocation: control flow; "this"

```
public class MyProgram {
  public static void main(String[] args)
  {
    A x = new A();
    x.m();
    System.out.println(‘‘Done’’);
  }
}
public class A {
  void m()
  {
    r();    // Equivalent to this.r();
  }
  void r()
  {
    System.out.println("Do something");
  }
}
```

Method invocation: control flow



Method invocation: control flow

```
class X {
  void f()
  {
    System.out.println(1);
  }
}
class Z {
  public static void main(String[] args)
  {
    X obj1 = new X();
    System.out.println(2);
    obj1.f();
    System.out.println(3);
  }
}
```

Method invocation: control flow

```
class X {
  void f()
  {
    System.out.println(1);
  }
}
class Z {
  public static void main(String[] args)
  {
    X obj1 = new X();
    System.out.println(2);
    obj1.f();
    obj1.f();
    System.out.println(3);
  }
}
```

Method invocation: control flow

```
class X {
  void f()
  {
    System.out.println(1);
  }
  void g()
  {
    System.out.println(4);
  }
}
class Z {
  public static void main(String[] args)
  {
    X obj1 = new X();
    System.out.println(2);
    obj1.f();
    obj1.g();
    System.out.println(3);
  }
}
```

Method invocation: control flow

```
class X {
  void f()
  {
    System.out.println(1);
  }
  void g()
  {
    System.out.println(4);
  }
}
class Z {
  public static void main(String[] args)
  {
    X obj1 = new X();
    System.out.println(2);
    obj1.g();
    obj1.f();
    System.out.println(3);
  }
}
```

Method invocation: control flow

```
class X {
  void g()
  {
    System.out.println(4);
  }
  void f()
  {
    System.out.println(1);
  }
}
class Z {
  public static void main(String[] args)
  {
    X obj1 = new X();
    System.out.println(2);
    obj1.f();
    obj1.g();
    System.out.println(3);
  }
}
```

Method invocation: control flow

```
class X {
  void f()
  {
    System.out.println(1);
  }
}
class Y {
  void g()
  {
    System.out.println(4);
  }
}
```

Method invocation: control flow

```
class Z {
  public static void main(String[] args)
  {
    X obj1 = new X();
    System.out.println(2);
    obj1.f();
    obj1.g();
    System.out.println(3);
  }
}
```

Method invocation: control flow

```
class Z {
  public static void main(String[] args)
  {
    X obj1 = new X();
    Y obj2 = new Y();
    System.out.println(2);
    obj1.f();
    obj2.g();
    System.out.println(3);
  }
}
```

Method invocation: control flow

```
class X {
  void f()
  {
    System.out.println(1);
  }
}
class Y {
  void g()
  {
    X obj3 = new X();
    System.out.println(4);
    obj3.f();
    System.out.println(5);
  }
}
```

Method invocation: control flow

```
class Z {
  public static void main(String[] args)
  {
    X obj1 = new X();
    Y obj2 = new Y();
    System.out.println(2);
    obj1.f();
    obj2.g();
    System.out.println(3);
  }
}
```

Method invocation: control flow

```
class X {
  void f()
  {
    System.out.println(1);
  }
}
class Y {
  void f()
  {
    System.out.println(4);
  }
}
```

Method invocation: control flow

```
class Z {
  public static void main(String[] args)
  {
    X obj1 = new X();
    Y obj2 = new Y();
    System.out.println(2);
    obj1.f();
    obj2.f();
    System.out.println(3);
  }
}
```

Scope

- Different classes can have attributes and methods which have the same names.

- For example given the following class definition

```
public class C {
    int a;
    //...
}
```

the variables `x.a` and `y.a` are different memory locations in the following client:

```
public class D {
    void m()
    {
        C x = new C();
        C y = new C();
        x.a = 3;
        y.a = 5;
    }
}
```

Scope (contd.)

- This also applies if the attributes are in different classes:

```
public class C {
  int a;
  // ...
}
public class E {
  int a;
  // ...
}
public class D {
  void m()
  {
    C x = new C();
    E y = new E();
    x.a = 3;
    y.a = 5;
  }
}
```

Scope (contd.)

- Parameters and local variables are known only within their method:

```
public class F
{
    void p(int a)
    {
      int b = 2;
      System.out.print(a + ``,'' + b);
    }
    void q(int a)
    {
      int b = 4;
      System.out.print(a + ``,'' + b);
    }
}
```

McGill

Scope (contd.)

- Attributes of a class are shared between its methods:

```
public class F
{
    int n;
    void p()
    {
        n = 3;
        //...
    }
    boolean q(String s)
    {
        if (n < 5 && s.equals(‘‘hello’’))
            return true;
        return false;
    }
}
```

Scope (contd.)

- ...but are different for different objects of the same class:

```
public class H
{
    void w()
    {
        F f1, f2;
        f1 = new F();
        f2 = new F();
        f1.p();
        f2.p();
        boolean a,b;
        a = f1.q(“hello”);
        b = f2.q(“good bye”);
    }
}
```

- In this example, f1.n and f2.n are different variables of the same class, because they belong to different objects of class F.

Scope (contd.)

- The *scope* of a variable, a parameter, an attribute or a method is the part of the program that can access that variable, attribute or method.
- The scope of a parameter of a method is the body of the method.
- Variables declared in the body of a method are called *local variables*, which means that their scope is only the body of the method.
- The direct scope of an attribute of a class or a method is the class itself (e.g. the direct scope of `id` is the `Student` class.)
- However, the indirect scope of an attribute of a class or a method is the rest of the program (e.g. the `id` attribute can be accessed by other clients with the expression `var.id`, where var is of type `Student`.)

Method invocation: parameter passing

- A *frame* is a space in memory which stores a set of variables. It can be viewed as a table containing the memory locations for each variable in the set.

- Suppose that a method is declared as follows:

```
type method(type1 param1, type2 param2,
                ..., typen paramn)
{
     statements;
}
```

- A method call of the form

```
objectreference.method(arg1, arg2, ..., argn)
```

...where *arg1*, *arg2*, ..., *argn* are expressions with type matching the types as appear in the method declaration, is executed by

McGill

Method invocation: parameter passing

First: evaluating each of the arguments *arg1*, *arg2*, ..., *argn* from left to right,

Second: creating a *frame*, reserving space for all the parameters of the method, and local variables declared in the body of the method. The frame also contains a pointer to the object refered to by the *variable*.

Third: in that frame, perform the assignments *param1* = *arg1*; *param2* = *arg2*; ...; *paramn* = *argn*; `this` = *objectreference*;

Fourth: "jumping" to the body of the method and executing the *statements* in order. The calling method is suspended while the called method is executed.

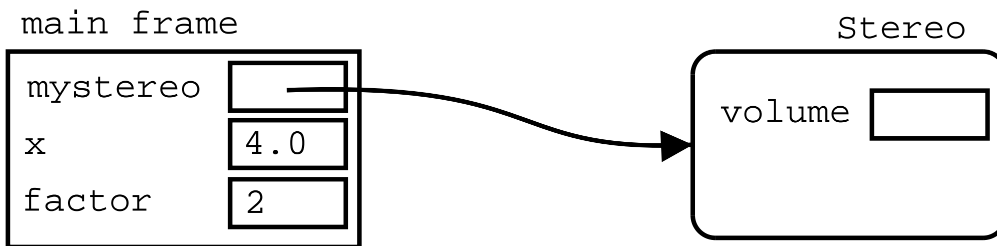
Fifth: when the end of the method is reached, or a `return` statement is reached, stop the method, the frame is discarded, and return to the calling method. The calling method is then resumed in the instruction immediatly after the method call.

Method invocation: Example

```
public class Stereo {
    double volume;
    void set_volume(double v)
    {
        volume = v;
    }
    double get_volume()
    {
        return volume;
    }
}
public class SoundSystem {
    public static void main(String[] args)
    {
        Stereo mystereo = new Stereo();
        double x, factor = 2;
        System.out.println("Testing...");
        x = 4.0;
        mystereo.set_volume(x*factor);
        System.out.println(mystereo.get_volume());
    }
}
```

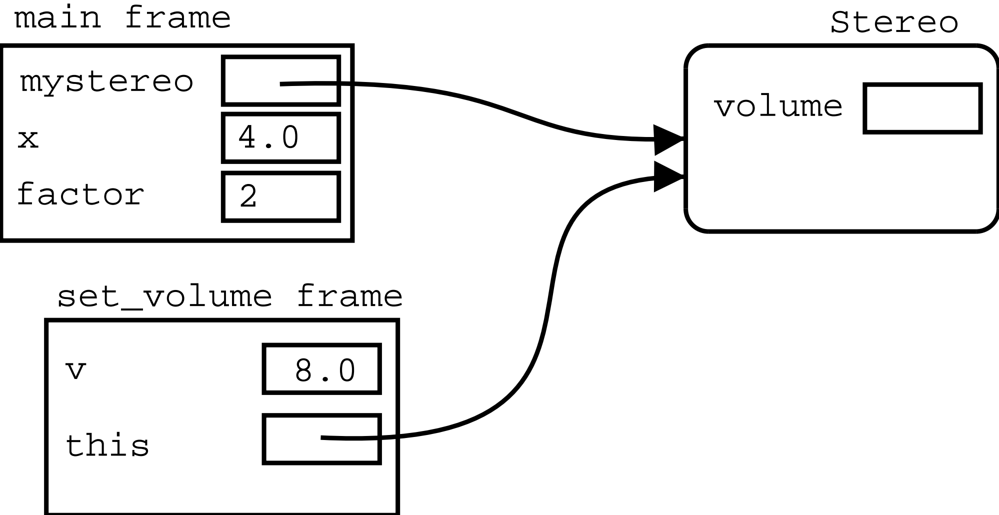
Method invocation: Memory structure

Before calling `mystereo.set_volume(x*factor)`



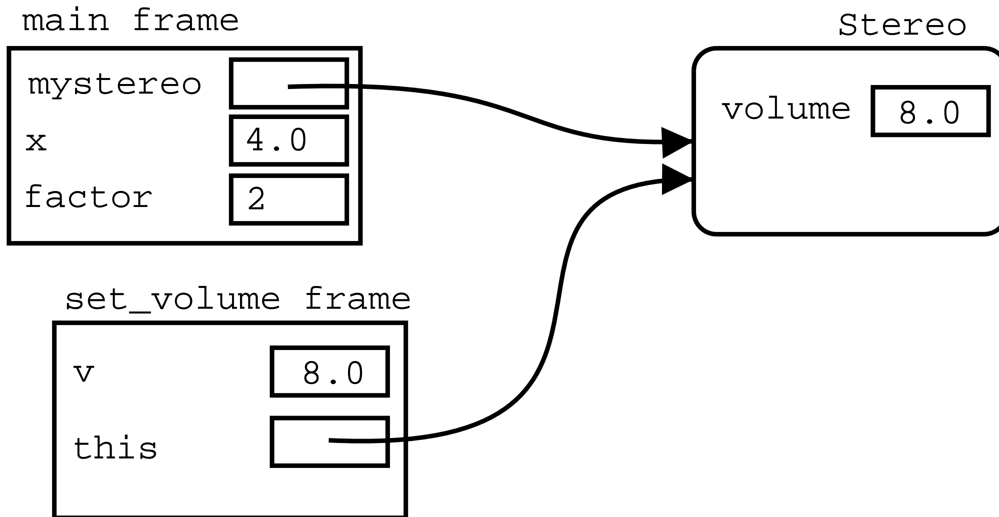
First its arguments (`x*factor`) are evaluated:
Evaluating `x*factor` in the main frame results in `8.0`

Method invocation: Memory structure



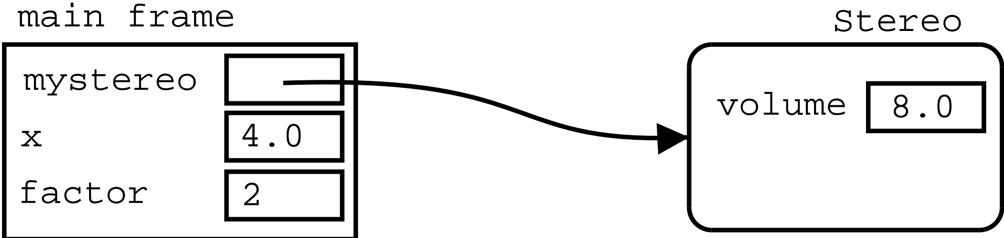
Method invocation: Memory structure

The current method (main) is suspended, and
the body of the called method (set_volume) is executed
in the context of the current frame (the set_volume frame):



Method invocation: Memory structure

Finally the called method frame is discarded, and computation of the calling method (main) is resumed in the instruction immediately after the method call.



Method invocation

```
public class Spy {
  int id;
  String name;
  Spy(String n, int i) {
    id = i;
    name = n;
  }
  String perform_mission(String description,
                         String target) {
    String info;
    this.getInsideTarget(target);
    info = this.getInformation(description);
    return info;
  }
  void getInsideTarget(String target) {
    System.out.println(id + " reporting.");
    System.out.println("Inside: " + target);
  }
  String getInformation(String messge) {
    return "Secret of " + message;
  }
}
```

Method invocation

```
public class MI6Sim {
  public static void main(String[] args)
  {
    Spy bond;
    String secret;

    bond = new Spy(``James Bond'', 007);

    secret = bond.perform_mission(``bake a pie'',
                                  ``kitchen'');
  }
}
```

Method invocation

main frame

bond

secret

Method invocation

main frame

bond

secret

Spy constructor frame

n

i

this

Method invocation

main frame

bond

secret

Spy constructor frame

n

i

this

Spy

id

name

Method invocation

main frame

bond

secret

Spy constructor frame

n James Bond

i 007

this

Spy

id

name

McGill

Method invocation

main frame

bond

secret

Spy constructor frame

n James Bond

i 007

this

Spy

id 007

name

Method invocation

main frame

bond	
secret	

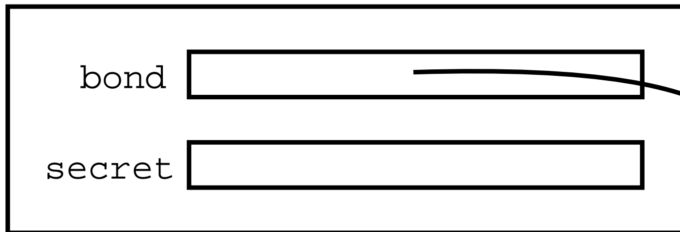
Spy constructor frame

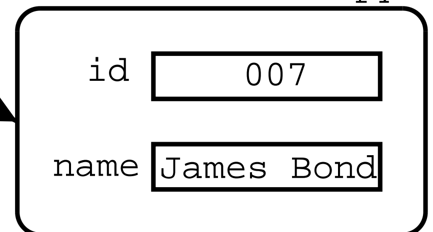
n	James Bond
i	007
this	(reference to Spy)

Spy

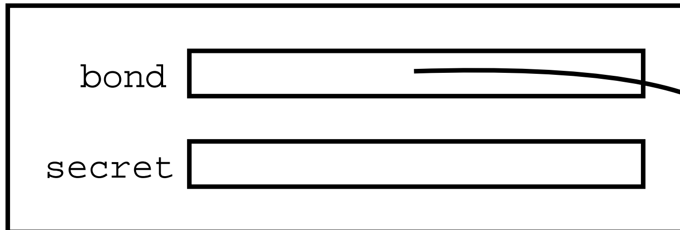
id	007
name	James Bond

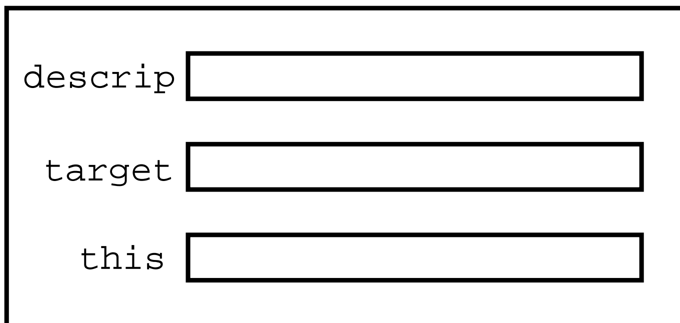
Method invocation

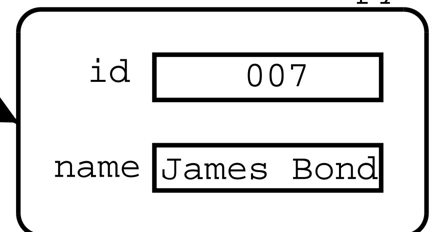




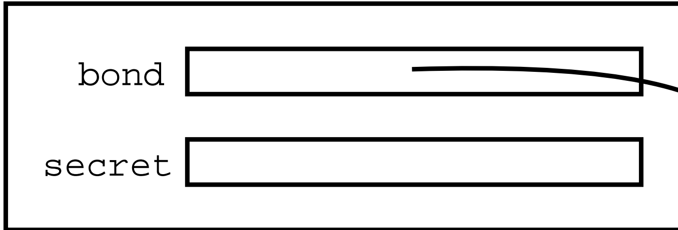
Method invocation

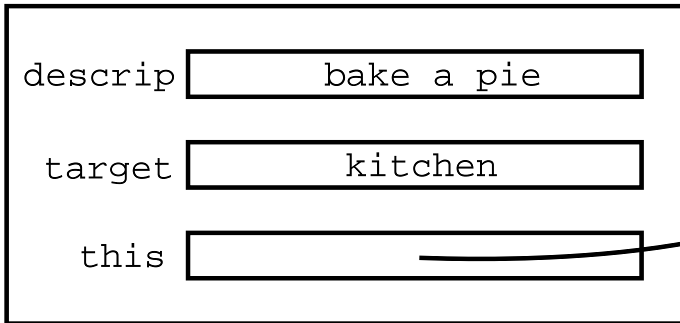


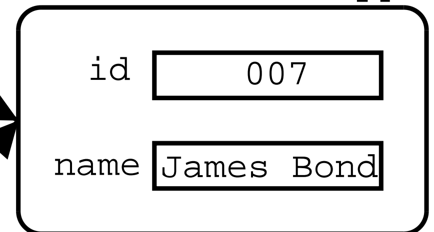




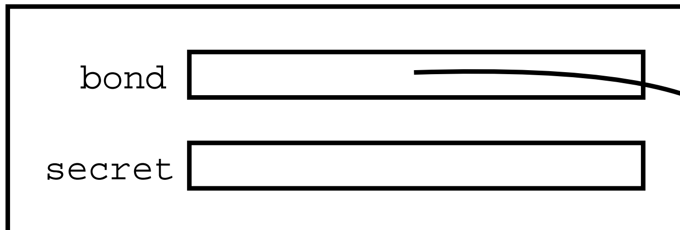
Method invocation

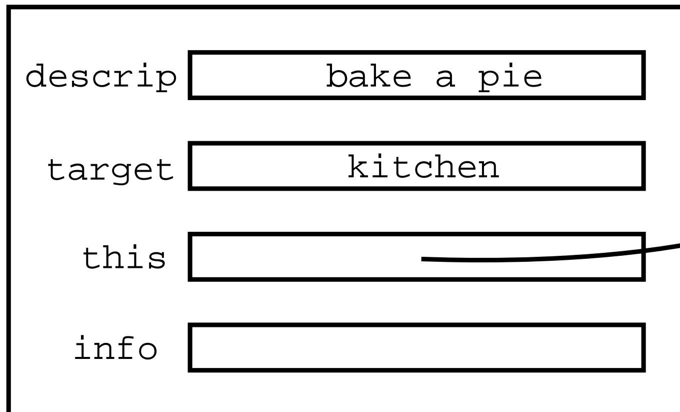


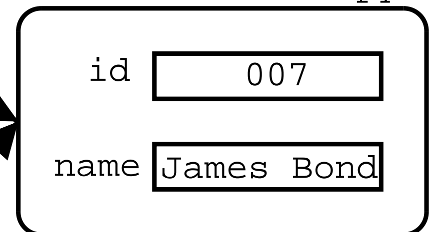




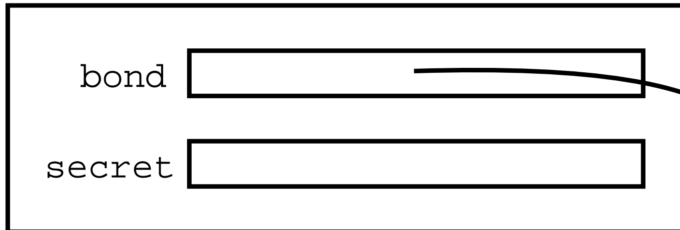
Method invocation

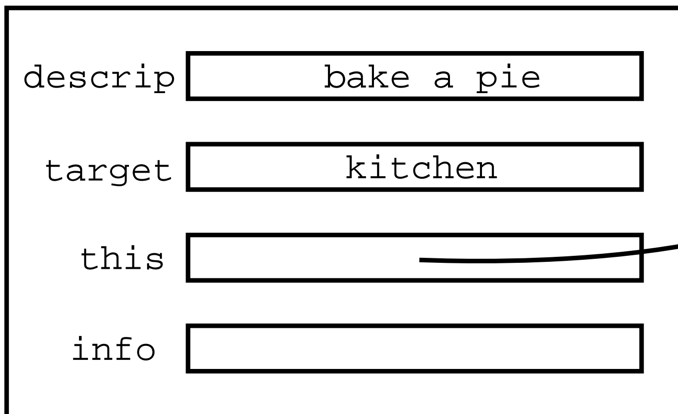


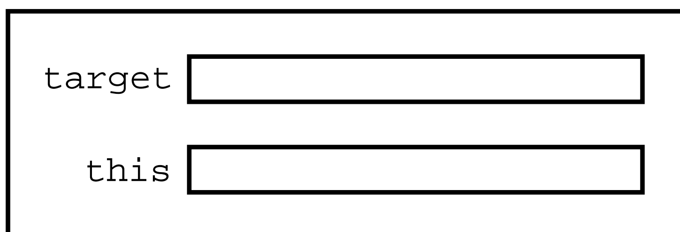


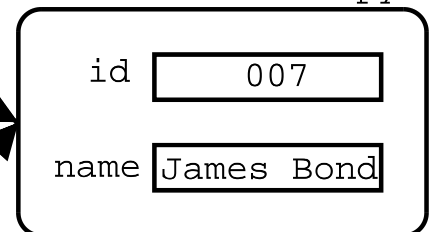


Method invocation

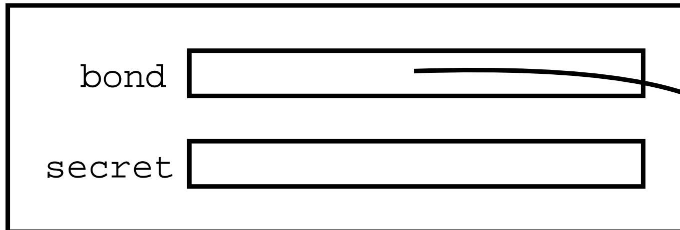


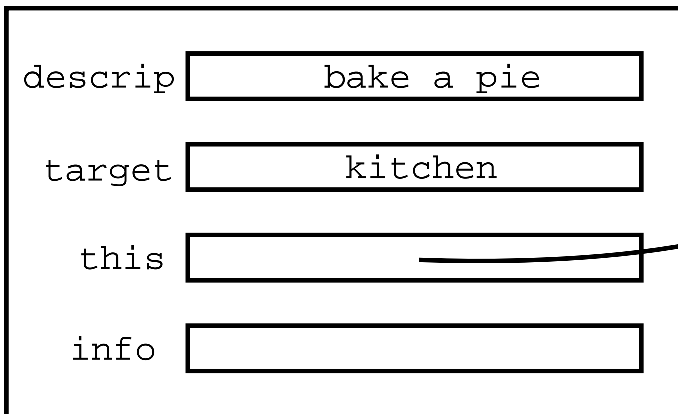


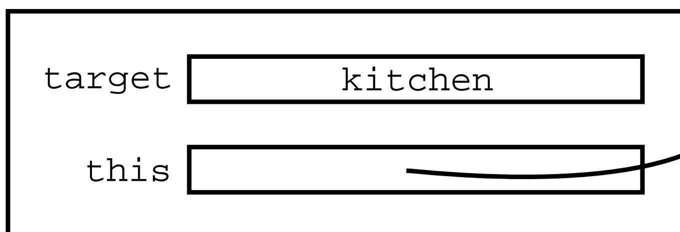


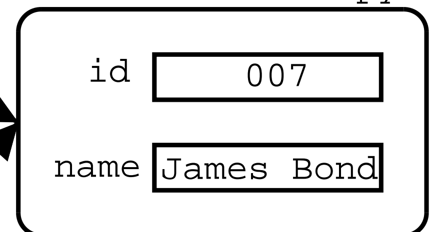


Method invocation

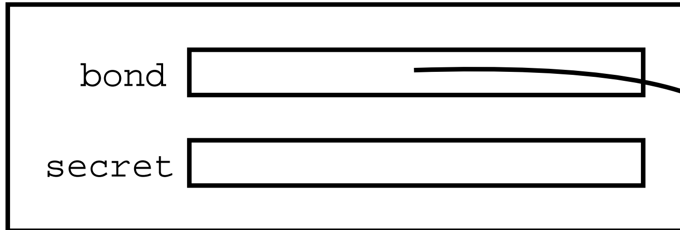


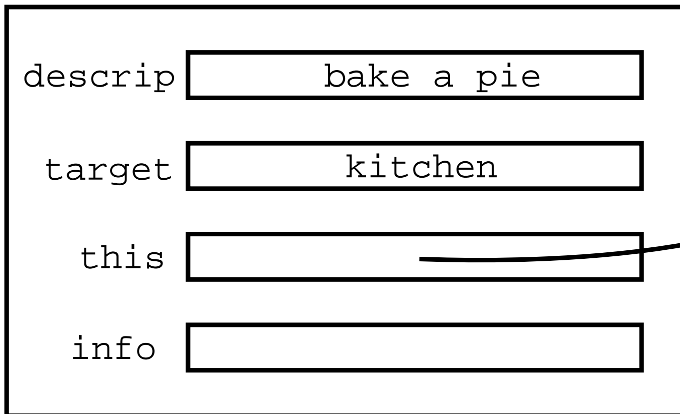


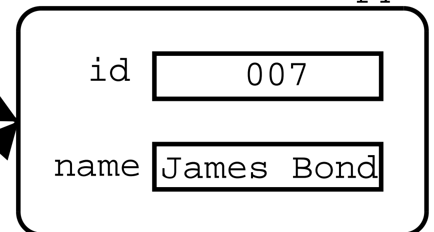




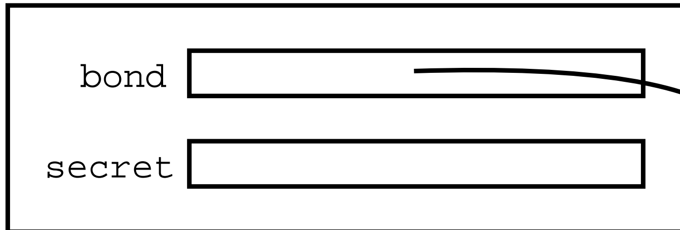
Method invocation

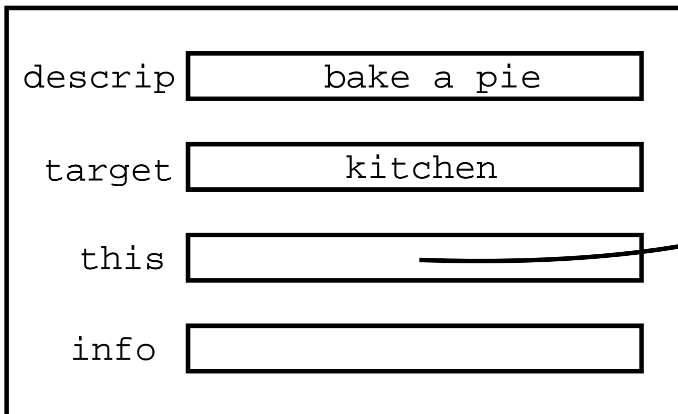


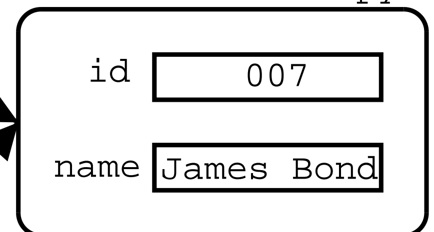


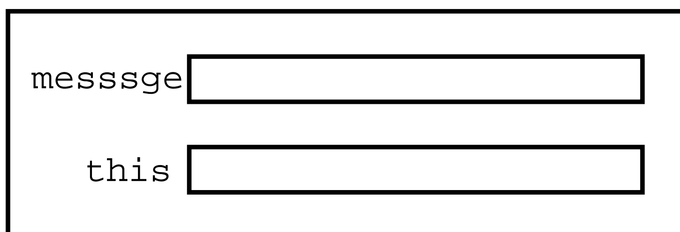


Method invocation

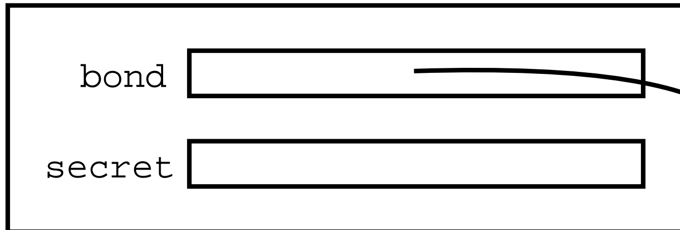


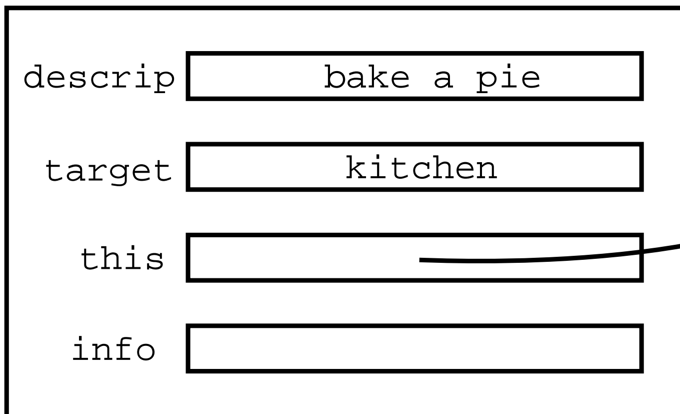


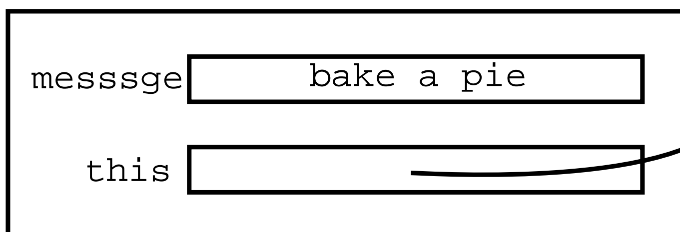


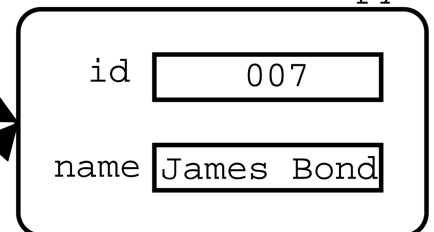


Method invocation

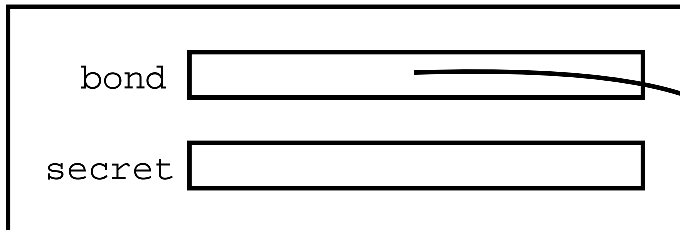


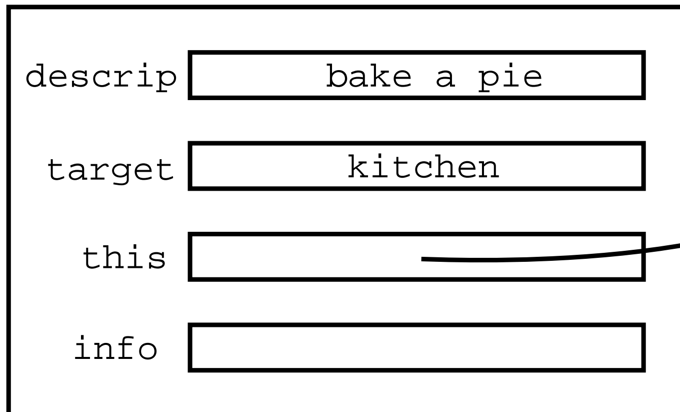


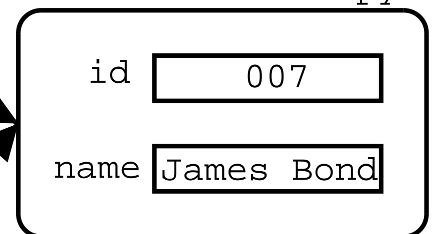




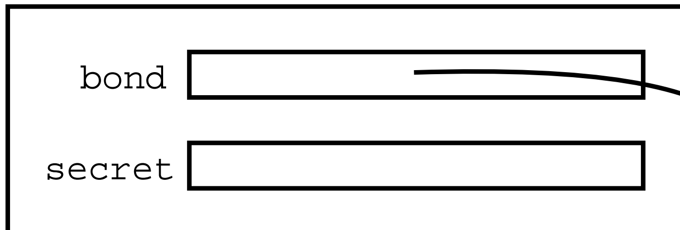
Method invocation

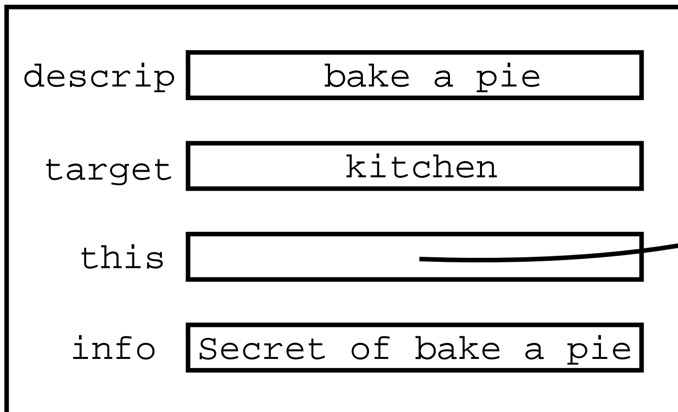


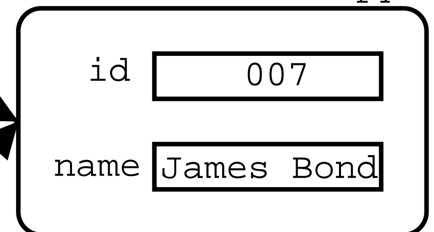




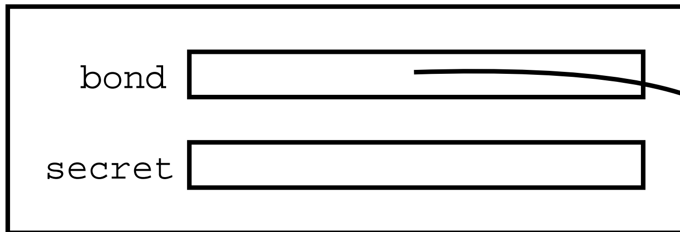
Method invocation

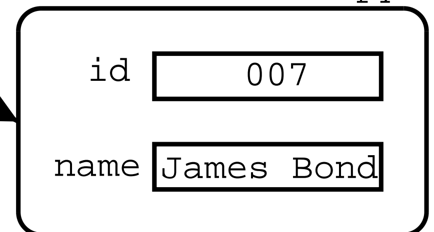




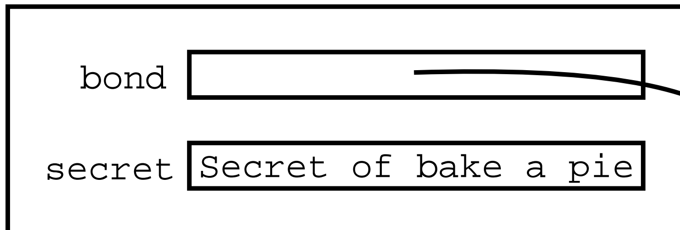


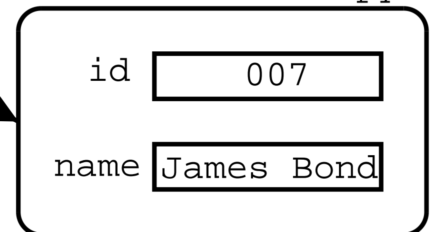
Method invocation





Method invocation





The end