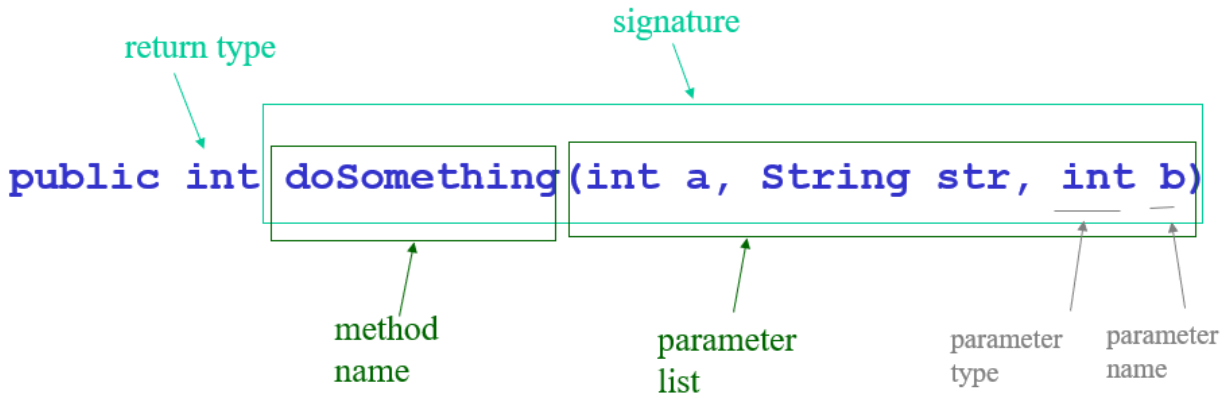
Methods

We now have a number of tools for processing data and making complex decisions. Our programs can now get pretty complicated, which requires a lot of thinking to keep track of how they work. As good computer scientists, we should be starting to feel worried that we need to be lazier, we need tools to help us organize our code into small pieces so that we never have to keep the whole program in mind at once.

Also, we often write a multi-step task that we then need to apply to various different data values. We could do this by copy-pasting the code and putting in different specific values, but any time we have multiple copies of the same code, that means we have to be responsible for keeping track of those copies and updating all of them any time we need to adjust how they work… which we should be too lazy to do.

The solution to both of these problems is to divide our programs up into methods. Methods are also known in other languages as modules, functions, procedures, subroutines, etc.

Method Header



Each method begins with a header that gives it a name, says what values it needs as input from the rest of the program, and says what type of result it will give back to the rest of the program. It should also have an access modifier which will almost always be public.

```
public returnType methodName(type1 param1, type2 param2, …) {
```

Some methods also have the keyword static. Methods that live in the class with main are almost always static, like main. Behavior methods in other classes are usually not static

The method name follows the usual rules for naming and should indicate what the code inside the method body will do.

The parameters are a list of zero or more local variables that the method will use to perform this code. Each parameter will start with a value that it is given by another part of the program when the method runs. We don't know what those values will be before the method runs, so we have to write the method so that it can deal correctly with *any* possible value based on the type.

The return type is the type of a value that the method will give back to another part of the program when it is finished. This can be any valid value for a variable. The return type could also be the special type *void* which means that the method does not return any value at all – it just does a series of steps but does not have a single result.

To start off with, we'll create very simple methods that have zero parameters and type void, so they get no values from the rest of the program, and give back no result.

Method Body

Like other structures, a method can have code inside it, which is called its body. We use curly braces {} to begin and end the method's body.

```
public void printHello5() {
    //body of the method
    for (int i = 0; i < 5; i++) {
        System.out.println("Hello");
     }
} // end of the method
```

The body of a method can include any code we would put anywhere else including conditionals, loops, printing, etc, but we would never create a method inside another method.

Local Variables

Variables created inside a method are local to that method – they only exist within that method ("in that method's scope"), while it is running. They do not persist afterwards. This means that multiple methods could have variables with the same name without interfering with each other.

Consider readability when choosing local variable names; if two methods are doing related things to the same kind of value, it may make more sense for them to use the same variable name. If two methods are doing very different things, it may cut down on confusion if they use different variable names as well.

Calling Methods

The point of a method is that it is a tool that other parts of the code can use. Methods are used by *calling*, that is, using their name. The name of a method always includes the parentheses after it, with values for any parameters inside.

We know that Java programs automatically begin with the first line of the main method. If main calls another method, that method is run at that point in main. If main does not call a method, that method does not run, even if it is present in the code.

```
1   public static void square() {
2       Scanner scanner = new Scanner(System.in);
3       System.out.println("your number? ");
4       int userNum = scanner.nextInt();
5       System.out.println("square is " + userNum * userNum);
6   }
7
8   public static void cube() {
9       Scanner scanner = new Scanner(System.in);
10      System.out.print("your number? ");
11      int userNum = scanner.nextInt();
12      System.out.println("cube is "
13              + userNum * userNum * userNum);
14  }
15
16  public static void main(String[] args) {
17      Scanner scanner = new Scanner(System.in);
18     // decide whether to call the cube or square method
19      System.out.print("1 for cube, 2 for square ");
20      int user = scanner.nextInt();
21      if (user == 1) {
22          cube();
23      } else {
24          square();
25      }
26
27      System.out.println("Goodbye");
28  }
```

In this code, the first thing that happens is that the user is prompted to choose between cube or square in main. If the user chooses 1, the cube() method runs. Once the cube method finishes, we are back in main, and we print “Goodbye” and in that case, the

square() method never runs. if the user chooses anything else, the square method runs, then we print “Goodbye” but in that case the cube() method never runs.

In this example, I put square() and cube() above main(). In Java, the order of whole methods does not matter; however, the code *inside* each method is still run from top to bottom. In this example I put main at the bottom, but the program will start with the first line inside the main, no matter where main is located in relation to other methods.

The Stack

When a method is called, a chunk of main memory space for the program is taken up by the method, including space for any local variables it has. The part of program memory used for methods is called the *stack* and each method’s chunk of space is called a *stack frame*.

When the method ends, the stack frame is “given back” that is, that chunk of main memory space is marked as no longer occupied, it is open for use to store a new stack frame for another method. If one method calls another which calls another, then we could end up with multiple stackframes which take a while before they come back off the stack.

Parameters

To make methods really useful, we need a way to get input from where the method is called into the method, so we can do the same code multiple times but with different starting values. We do this by passing values into the method’s parameters.

Parameter Lists

In its parentheses, each method is defined with a list of zero or more parameters. These are local variables in the method, but unlike variables created in the body of the method, they are automatically given values before the method starts. When writing the method, you don’t have to give them values, but you do have to think about how to *write the method so that it will work properly no matter what value the parameters start with*.

Like any other variable, each parameter needs a type, and any type valid for variables can be used for parameter types, including class types.

If you need the variable to start with one specific value, then it should not be a parameter, it should be a local variable defined in the body of the method. If you need input from the user, the variable for that should not be a parameter, it should be a local variable defined in the body of the method. The point of a parameter is that it is a way for *another part of the program* to feed a value to the method.

When we talk about method parameters, we use words like “takes” “needs” “starts with” or “given” while if we were reading the value of a local variable as user input, we would say “takes as input” “asks the user for” “takes as input” “given by the user”.

Calling Methods with Parameters

Parameters to a method are listed in a specific order. When the method is called, it must be given values in the same order, in its parentheses. Those values might be literals, or stored in variables, or creating by evaluating an expression. We might talk about calling a method with given values in its parentheses as calling it “of” “on” “with” those values or “applied to” those values.

In this example, two methods each have two parameters.

```
1   // either do or don't print the given string
2   public static void printDo(boolean printIt, String word) {
3       if (printIt) {
4           System.out.println(word);
5       }
6   }
7
8   // print the given string a number of times
9   public void printTimes(String word, int howMany) {
10      for (int i = 0; i < howMany; i++) {
11          System.out.println(word);
12      }
13  }
14
15  public static void main(String[] args) {
16      printTimes("hello", 3);
17      printDo(true, "hi");
18
19      int times = 2;
20      printTimes("goodbye", times);
21      printDo(true, "bye");
22
23      String str = "this is awkward";
24      printTimes(str, times - 3);
25      printDo(false, "huh");
26
27      System.out.println("END");
28  }
```

Here is the output we would see when the above code runs annotated with the lines causing the output

```
hello               line 16 / line 11
hello               line 16 / line 11
hello               line 16 / line 11
hi                  line 17 / line 4
goodbye             line 20 / line 11
goodbye             line 20 / line 11
bye                 line 21 / line 4
END                      line 27
```

While line 16 runs, memory would look like this (note the counter I counting its way up to howMany)

variable	*MM*	*value*
[stackframe: main (args)]		
args	50	null
	51	
	…	
[stackframe:printTimes(“hello”, 3)]		
word	60	“hello”
howMany	61	3
i	62	~~0~~ 1 ~~2~~ 3
	…	

After this, the stackframe for printTimes would be given back, so the next method call could re-use the same space, so while line 17 runs, memory could look like

variable	*MM*	*value*
[stackframe: main (args)]		
args	50	null
	51	
	…	
[stackframe:printDo(true, “hi”)]		
printIt	60	true
word	61	“hi”
	…	

While line 20 runs (notice that howMany gets the value of the times variable, 2:

variable	*MM*	*value*
[stackframe: main (args)]		
args	50	null
times	51	2
	…	
[stackframe:printTimes(“goodbye”, times)]		
word	60	“goodbye”
howMany	61	2
i	62	~~0~~ 1 2
	…	

While line 24 runs (this time word gets the value of the variable str and howMany gets the value of the expression times – 3

variable	*MM*	*value*
[stackframe: main (args)]		
args	50	null
times	51	2
str	52	“this is awkward”
	…	
[stackframe:printTimes(str, times - 3)]		
word	60	“this is awkward
howMany	61	-1
i	62	0
	…	

Main calls each method three times with different values for the parameters, called *arguments*. (I will only talk about the stack for the first call, all the others are similar).

- printTimes(“hello”, 3)
    - just before printTimes begins, a stack frame is placed on the stack and in it are space for the parameter variable word, which is initialized to the value “hello” and the parameter variable howMany, which is initialized to the value 3
    - the method runs and “hello” is printed three times
    - the method ends, and we give back the stack frame, so this space in main memory can be re-used for other methods. Now we are back in main()
- printDo(true, “hi”)
    - just before printDo begins, the parameter variable printIt is initialized to the value true and the parameter variable howMany is initialized to the value “hi”
    - the method runs, printing “hi”
    - the method ends and we are back in main()
- printTimes(“goodbye”, times)
    - just before printTimes begins, the parameter variable word is initialized to the value “goodbye” and the parameter variable howMany is initialized to the value 2 (the values of the times variable)
        - printTimes does *not* have access to the times variable itself; it would *never* make sense to try to use the times variale in printTimes. We would never write printTimes to be dependent on the name of a variable from main!
    - the method runs and “goodbye” is printed two times
    - the method ends, and we are back in main()

- printDo(true, “bye”)
    - just before printDo begins, the parameter variable printIt is initialized to the value true and the parameter variable howMany is initialized to the value “bye”
    - the method runs, printing “bye”
    - the method ends and we are back in main()
- printTimes(str, times - 3)
    - just before printTimes begins, the parameter variable word is initialized to the value “this is awkward” and the parameter variable howMany is initialized to the value -1 (the result of subtracting 3 from the value of the times variable)
    - the method runs and since the for loop starts at 0 and goes only as long as it is < howMany, which is -1, nothing is printed
    - the method ends, and we are back in main()
- printDo(false, “huh”)
    - just before printDo begins, the parameter variable printIt is initialized to the value false and the parameter variable howMany is initialized to the value “huh”
    - the method runs, but since it checks printIt, which is false, it doesn’t print anything
    - the method ends and we are back in main()

Notice that we were able to pass in literal values, values stored in variables, and values from expressions as our arguments.

In general, the reason we are passing values into a method is so that the method can use those values, so in most cases we wouldn’t be giving a parameter variable a new value inside the method body, but we might *adjust* the value of a parameter in the course of the task the method does, so it is worth considering what happens when we pass a variable in as argument to a parameter that is changed by its method.

```
1   public void countDownFrom(int top) {
2        // since top is already initialized
3        // there is no initialization step
4        // in this for loop
5        for (          ; top > 0; top = top - 1) {
6            System.out.println(top);
7        }
8        System.out.println("YAY");
9    }
10
11   public static void main(String[] args) {
```

```
12          countDownFrom(10);
13
14          int k = 5;
15          countDownFrom(k);
16
17          System.out.println("k is now " + k);
18      }
```

While line 12 runs, memory might look like this:

variable	*MM*	*value*
[stackframe: main (args)]		
args	50	null
	...	
[stackframe:countDownFrom(10)]		
top	60	~~10 9 8 7 6 5 4 3 2 1~~ 0
	...	

While line 15 runs, memory might look like this:

variable	*MM*	*value*
[stackframe: main (args)]		
args	50	null
k	51	5
	...	
[stackframe:countDownFrom(k)]		
top	60	~~5 4 3 2 1~~ 0
	...	

In this code, the countDownFrom method uses the top parameter as its counter, counting down from whatever value top was initialized to until top is down to 0, decreasing each time.

When we call countDownFrom(10), the top parameter variable is initialized to 10, and then it counts down from 10, changing the top variable each time through the loop. There was no variable in main, so there is nothing to change.

When we call countDownFrom(k), the top parameter variable is initialized to 5, which is the value of k from main. The code counts down from 5, each time through the loop changing the value of top. But does this also change the value of k?

Pass by Value

No, it doesn't. Java's approach to arguments is called *pass by value*: a parameter is initialized with a *copy of the value* that is given when the method is called. Since it is just a copy, any change made inside the method would not affect any variable that was passed in for that parameter.

So, no change made to a parameter's value inside the called method can change the value of a variable in the calling method. In our example above, top would be initialized to 5, a copy of the value of k, but the changes made to top after that would not affect k, and k would still be 5 after the countDownFrom method finished for the second time.

Returning from Methods

Just as we use parameters to get values from the caller into a method, a method can give a value back to a caller by returning it.

Return Type

Any type that is valid for a variable is also valid for a method's return type, including class types. We also have the special type `void` to indicate that a method does not return anything. If a method specifies a return type, the method must return a value of that type by the end of the method. A method can return any value of its type, including values like zero or null. Only a single value can be returned.

Return

The keyword return is followed by the value (of the return type) that this method will evaluate to. It could return any expression that evaluates to the correct type, including a literal value, a variable, or the result of doing operations

```
public double sumDouble(double a, double b) {
    double sd = (a + b) * 2;
    // return statement
    return sd;
}
```

In the example above, we did the math in the expression and put the result into a variable, then returned the variable. It is somewhat more common, in a case like this, just to put the expression after the keyword return:

```
public double sumDouble(double a, double b) {
   return (a + b) * 2;
}
```

A return statement ends the method, so a method can't have any statements after a return. For this reason, we should never have a return inside a loop unless it is also in a conditional that means it only happens under certain circumstances (which mean the method should end early at that point).

Many people feel that code is easier to read and debug if there is only one return statement per method, so they would use a variable to store the eventual value to return, change that variable as needed with conditionals, and then return that variable at the end. We previously talked about situations like this in the only method we've written with a return so far, the toString.

Even if we write a method using multiple return statements, only one of them can happen.

When we talk about methods that return, we would say things like it "returns" "gives back" or "results in" the value, while to say that it prints instead, we would say it "reports" "tells" the user" or "shows".

Calling Methods that Return

When we call a method that returns, that method call evaluates to the returned value – imagine the method call turning into the return value. Most of the time, if we called a method that returns, it is because we want to do something with that value. So although it is legal to just call a method returns as a statement of its own, this is usually a bad idea. We should be storing the return value in a variable or else using it in another expression, maybe passing it to another method (say, to print it).

If we fail to do this, we sometimes say that we "dropped the return value on the floor" imagining that the method tossed its return value back to our code and we failed to catch it.

```
public double sumDouble(double a, double b) {
    return (a + b) * 2;
}

public static void main(String[] args) {
    // useless, we drop 14 on the floor
    sumDouble(5.1, 1.9);

    // catching 14 in a variable
    double result = sumDouble(5.1, 1.9);

    // printing 14
```

```
    System.out.println("your result is " + sumDouble(5.1,
   1.9));

    // using return values as arguments
    double bigMath =
        sumDouble(sumDouble(2.2, 3.3), sumDouble(1.1, 5.5));
        // evaluated as:
        // sumDouble(11.0, sumDouble(1.1, 5.5))
        // sumDouble(11.0, 13.2)
        // 48.8
    System.out.println("result of big math was " + bigMath);
}
```

So at the end, we print "result of big math was 48.8"

Overloading

Choosing good names for methods is an important programming skill; we want the name to clearly communicate what the method does so that when we read a single statement that calls the method, we have good expectations for that many lines of code inside the method that this may stand for.

Sometimes we need to write several methods that do the same task, but the parameters are different in number or type. For readability, it may actually be better to give all these methods the same name. When we have multiple names with the same name but different parameters, this is called *overloading* a method.

Methods and Memory

Remember that the pass by value rule means that changes made to a parameter will not affect any variable passed in for that parameter. However, the value of a reference variable is the memory address of an object, so if we use the dot operator to make changes to instance variables, these changes will still be there after the method ends.

```
// Dog class
public class Dog {
    String name;
    int age;
    Dog bestFriend;
}
```

```
public static void main(String[] args){
     Dog fluffy = new Dog();
     fluffy.name = "Fluffy";
     fluffy.age = 3;

     // pass fluffy's address
     // to method
     playDate(fluffy);

     // Mr Friendly is sticking around
     // after end of method
     System.out.println(
          "still friends with " +
          fluffy.bestFriend.name);
}

 public static void playDate(Dog visitor) {
   // it was a long playdate
   visitor.age = visitor.age + 1;

   // create a new Dog
   // on the heap
   Dog playMate = new Dog();
   playMate.name = "Mr. Friendly";

   // put the address of the new dog
   // into visitor's bestfriend
   // variable - permanent change
   visitor.bestFriend = playMate;

   System.out.println(visitor.name + " made
    friends with " + playMate.name;
 }
```

variable	*MM*	*value*
[stackframe: main (args)]		
args	50	null
fluffy	51	MM100
	...	
[stackframe: playDate(fluffy)]		
visitor	60	MM100
playMate	61	MM200
	...	
[heap]		
	100	name: "Fluffy" age: ~~3~~ 4 bestFriend ~~null~~ MM200 ...
	...	
	200	name: "Mr. Friendly" age: 1 bestFriend null ...

The playDate method takes a Dog as its visitor parameter, this means it takes the value of a Dog variable, which is the memory address of a Dog object on the heap. So when we call the method as playDate(fluffy), as the method begins, the value of visitor starts as the value of fluffy, which is the memory address MM100.

So changes during playDate to the visitor's values affect the same object as original Dog fluffy, since they are just two variables pointing to the same object. The change to visitor's age is a change to fluffy's age. The change to Visitor's bestFriend is a change to fluffy's bestFriend.

This means that when we give visitor a new bestFriend, that Dog is still reachable (on the heap) when we get back to main, by going through fluffy.bestFriend. So, we can take advantage of putting data in objects on the heap to get more back from methods than the single value we could just by returning.

Memory Management

Suppose we create a Dog object using new, but then changed the variable that was holding that Dog's address by using new again, so that it pointed to a different Dog. They we would have lost access to the first Dog object. The object still existed in main memory, but our program no longer had a way to access it.

Space for any variables in a method's stackframe is automatically given back when the method ends, but space for objects on the heap is not automatically given back.

In some languages, it is the responsibility of the programmer to manage how their program uses main memory space. In these languages if we use new to create an object on the heap and then lose access to it, we can now no longer reuse this memory space to store something else – it is still marked as being in use – but we cannot actually use the data there. This is called a *memory leak*. In these languages, the programmer must explicitly give back the space for an object on the heap so that space can be re-used. They have to think carefully about when to do this. Too early would mean destroying data while we still need it, too late could lead to running out of memory space.

We're glad we work in Java where we don't have to deal with that!

In Java, the program's memory space automatically undergoes a process called garbage collection where objects on the heap are tested to see whether they are still reachable by following pointers starting from variables still on the stack. If we can no longer access an object on the heap in any way from the stack, that means the code can no longer use that object, and it is automatically marked as space to be re-used. At the end of the program, all space on stack and heap is given back and can be used for other programs.

In the following update to the previous code, both main and playDate create some extra Dogs. The stackframe for playDate is given back when the method ends, and it is greyed out to indicate this. Suppose that right before the final println, garbage collection ran. The Dog object at MM 100 is still reachable from the stack through the leash variable.

The Dog at MM200 is reachable because we can go through leash.bestFriend – so we're not just checking what we have a local variable in main for. If we can reach an object through several uses of the dot operator, it is still usable by the program.

The Dog at MM300 is no longer reachable, since the only variable that stored it was in the playDate stackframe, which has been given back. The Dog at MM400 is no longer reachable; the fluffy variable used to hold that address, but it has been replaced. The Dog at MM500 is reachable through the fluffy variable.

So the Dogs at MM300 and MM400 would be garbage collected at this point

```
// Dog class
public class Dog {
    String name;
    int age;
    Dog bestFriend;
}
```

```
public static void main(String[] args){
    Dog fluffy = new Dog();
    fluffy.name = "Fluffy";
    fluffy.age = 3;

    // pass fluffy's address
    // to method
    playDate(fluffy);

    // Mr Friendly is sticking around
    // after end of method
    System.out.println(
        "still friends with " +
        fluffy.bestFriend.name);

  // second variable, same dog
  Dog leash = fluffy;
  // same as fluffy.age = 10
  leash.age = 10;

  // another dog
  fluffy = new Dog();
  fluffy.name = "New Dog";

  // yet another dog
  fluffy = new Dog();
```

variable	*MM*	*value*
[stackframe: main (args)]		
args	50	null
fluffy	51	~~MM100~~ ~~MM400~~ MM500
leash	52	MM100
	...	
[stackframe: playDate(fluffy)]		
visitor	60	MM100
playMate	61	MM200
thirdDog	62	MM300
[heap]		
	100	name: "Fluffy" age: ~~3~~ ~~4~~ 10 bestFriend: ~~null~~ MM200 ...
	...	
	200	name: "Mr. Friendly" age: 1 bestFriend: null ...
	...	
	300	name: "Stranger" age: 0 bestFriend: null ...
	...	
	400	name: "New Dog" age: 0

```
    fluffy.name = "Final Dog";

    System.out.println("END");
}

 public static void playDate(Dog visitor) {
   // it was a long playdate
   visitor.age = visitor.age + 1;

   // create a new Dog
   // on the heap
   Dog playMate = new Dog();
   playMate.name = "Mr. Friendly";

   Dog thirdDog = new Dog();
   thirdDog.name = "Stranger";

   // put the address of the new dog
   // into visitor's bestfriend
   // variable - permanent change
   visitor.bestFriend = playMate;

   System.out.println(visitor.name + " made
    friends with " + playMate.name + " and
    " + thirdDog.name;
 }
```

	bestFriend: null ...
...	
500	name: "Final Dog" age: 0 bestFriend: null ...
...	

Garbage collection is much more convenient for programmers, and means that at most we can run low on space for a little while until the next time garbage collection runs during the run of our program. On the other hand, we can't control when garbage collection happens, so it could potentially happen that in a program we create a lot of objects at a point when garbage collection hasn't run for a while, and our program would suddenly slow down as garbage collection runs and tries to keep up. In modern versions of Java though, a little bit of garbage collection happens periodically throughout a program's run to try to keep the speed consistent.

Methods in Classes

(Of course, all Java methods live in classes, but here we are talking about behavior methods in classes rather than static methods that live with main.)

Methods can have local variables, including parameters, but methods in a class with instance variables also have access to all of those instance variables. There is no reason to pass a class' instance variable in to one of its methods as a parameter.

Remember that adding instance variables is an important design decision for a class. You should never add extra instance variables just because a particular method needs a variable – that's what local variables are for. Adding an instance variable means that you are changing the class' design because the original design had a problem. If there is a local variable in one method in a class that we need to get to another method in order to do a task, we should use parameters and return to do this.

Now let's look at a fundamental difference between a method outside and inside a class

```
public class Cat{
   public String name;

   // inside the Cat class
   // a Cat is doing this behavior,
   // so if we pass in another Cat
   // there are two Cats in this method
   public void sayHi(Cat othercat) {
         System.out.println("Cat " + name +
         " says hi to Cat " + othercat.name);
   }
}

public class Tester {

   // outside the Cat class
   // have to pass in two Cats to have
   // two Cats in the method
   public static void sayBye(Cat c1, Cat c2) {
         System.out.println("Cat " + c1.name +
              " says bye to Cat " + c2.name);
   }

   public static void main(String[] args) {
    Cat cat1 = new Cat();
    Cat cat2 = new Cat();
    cat1.name = "Fluffy";
    cat2.name = "Tux";
    sayBye(cat1, cat2); //"Cat Fluffy says bye to Cat Tux"
    sayBye(cat2, cat1); //"Cat Tux says bye to Cat Fluffy"
    cat1.sayHi(cat2);   //"Cat Fluffy says hi to Cat Tux"
```

```
     cat2.sayHi(cat1);    //"Cat Tux says hi to Cat Fluffy"

    }
}
```

The method sayBye is outside the Cat class. It has two Cat parameters, called c1 and c2 passed in so that it can print the first Cat saying bye to the other Cat.

The method sayHi is inside the Cat class. All methods in the Cat class are behaviors that a Cat can do. So there is automatically a Cat in the method, the cat who is *doing* the method. This method also has a Cat parameter, which adds a *second* Cat to the method, so it can print the Cat who is doing the behavior saying hi to the other cat.

In the behavior method sayHi, to talk about the name of the Cat doing the behavior we just say `name`, but to talk about the other Cat's name we have to say `othercat.name`. It would be nice if for the cat doing the behavior we had something to put in front of the dot operator to make it more obvious that there are two Cats here, but we don't have a variable for that. The Cat doing the behavior didn't come in as a parameter variable, and we certainly couldn't use the variables from main! [1]

this

To make it more obvious in a behavior method that we always have an object that is doing the behavior, we can use the keyword `this` (sometimes pronounced "t'hiss") to represent the Object that was in front of the dot operator when the method was called. So with `this` we could re-write sayHi as

```
    public void sayHi(Cat othercat) {
       System.out.println("Cat " + this.name +
        " says hi to Cat " + othercat.name);
    }
This code does exactly the same thing, but instead of the bare
    instance variable name, we used this.name to emphasize whose
    name we are talking about: the name of the cat who is the
    one saying hi.
```

So if we called cat1.sayHi(cat2) then `this` is the same as cat1 and othercat is cat2, but if we call cat2.sayHi(cat1) then `this` is the same as cat2 and othercat is cat1. You can think of `this` as being like a variable holding the object's own memory address, but it is important to understand that it isn't a variable in the sense that we can't change it.

---

[1] For so many reasons! They are in a different method's scope, in a whole different class! They could have been created by a different programmer long after we wrote the Cat class! We don't want to limit the sayHi method to only ever work for those specific Cat variables, it should be a behavior that works for all Cats ever!

If we think of behavior methods in terms of the object doing them, we could think of `this` as "me", "myself" "the one doing the behavior. So we might pronounce `this.name` as "my name" or "my own name"

So far `this` has helped to make code a little easier to read when there are multiple objects involved. It can also be used to avoid the shadowing problem; if we are in the habit of always saying `this.instanceVar` instead of just `instanceVar`, we're less likely to accidentally create a new local variable with the same name when what we intended was to use an instance variable.

The Power of this

Because `this` holds the memory address of the object doing the behavior, it also allows some things we could not do before, such as setting a variable to point to the object doing the behavior, or passing that object to another method.

Here's an updated Cat class

```
public class Cat{
   public String name;
   public Cat friend;

   public void sayHi(Cat othercat) {
         System.out.println("Cat " + this.name +
         " says hi to Cat " + othercat.name);
   }
   public void makeFriends(Cat othercat) {
         // my friend is the other cat
         this.friend = othercat;
         // the other cat's friend is ME
         othercat.friend = this;

         System.out.println(this + " made a new friend");
   }
}
```

```
public static void main(String[] args) {
    Cat cat1 = new Cat();
    Cat cat2 = new Cat();
    cat1.name = "Fluffy";
    cat2.name = "Tux";
    cat1.sayHi(cat2);
    cat2.sayHi(cat1);
    cat1.makeFriends(cat2);
```

```
}
```

In the makeFriends method, the friend instance variables of two Cats are set to point to each other, using `this` so that the Cat who is carrying out this task can set the other cat's friend variable to store the acting Cat's memory address.

Let's think about how main memory looks when this code runs (instead of drawing the memory multiple times, I'll just show all versions of the stackframes, even though really the later ones would replace the earlier ones). When we call a behavior method with a dot operator, whatever object was in front of the dot operator is `this`, so its memory address was stored in the this keyword.

variable	*MM*	*value*
[stackframe: main (args)]		
args	50	null
cat1	51	MM100
cat2	52	MM200
	…	
[stackframe: cat1.sayHi(cat2)]		
this	60	MM100
othercat	61	MM200
	…	
[stackframe: cat2.sayHi(cat1)]		
this	60	MM200
othercat	61	MM100
	…	
[stackframe: cat1.makeFriends(cat2)]		
this	60	MM100
othercat	61	MM200
	…	
[heap]		
	100	name: "Fluffy" friend: ~~null~~ MM200 …
	…	
	200	name: "Tux" friend: ~~null~~ MM100 …
	…	

I didn’t show the stackframe for println for the above code. Note that passing `this` to println is a way for an object to print itself (this triggers its own toString (you can’t see the toString for Cat, to make the examples shorter, but we know it must have one!)).