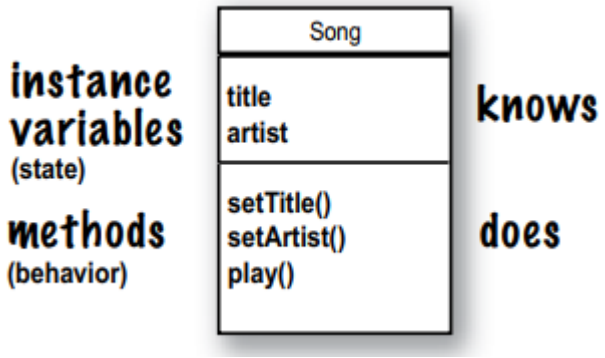
Chapter 4

How Objects Behave

- We already know that each instance of a class (each object of a particular type) can have its own unique values for its instance variables.
- For example, Dog A can have a name “Fido” and a weight of 70 pounds. Dog B is “Killer” and weighs 9 pounds.
- If the Dog class has a method makeNoise(), don’t you think a 70-pound dog barks a bit deeper than the little 9-pounder?
- An object has behavior that acts on its state. In other words, **methods use instance variable values.**

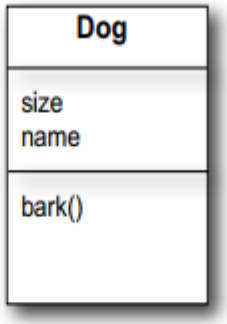
- Recall: a class describes what an object knows and what an object does
- Every object of that type can have different instance variable values. But what about the methods?
- Every instance of a particular class has the same methods, but the methods can ***behave*** differently based on the value of the instance variables
- The play() method plays a song, but the instance you call play() on will play the song represented by the value of the title instance variable for that instance

```
void play() {
    soundPlayer.playSound(title);
}
```



- A small Dog's bark is different from a big Dog's bark.
- The Dog class has an instance variable size, that the bark() method uses to decide what kind of bark sound to make.

```
class Dog {
  int size;
  String name;

  void bark() {
    if (size > 60) {
       System.out.println("Wooof! Wooof!");
    } else if (size > 14) {
       System.out.println("Ruff!  Ruff!");
    } else {
       System.out.println("Yip! Yip!");
    }
  }
}
```



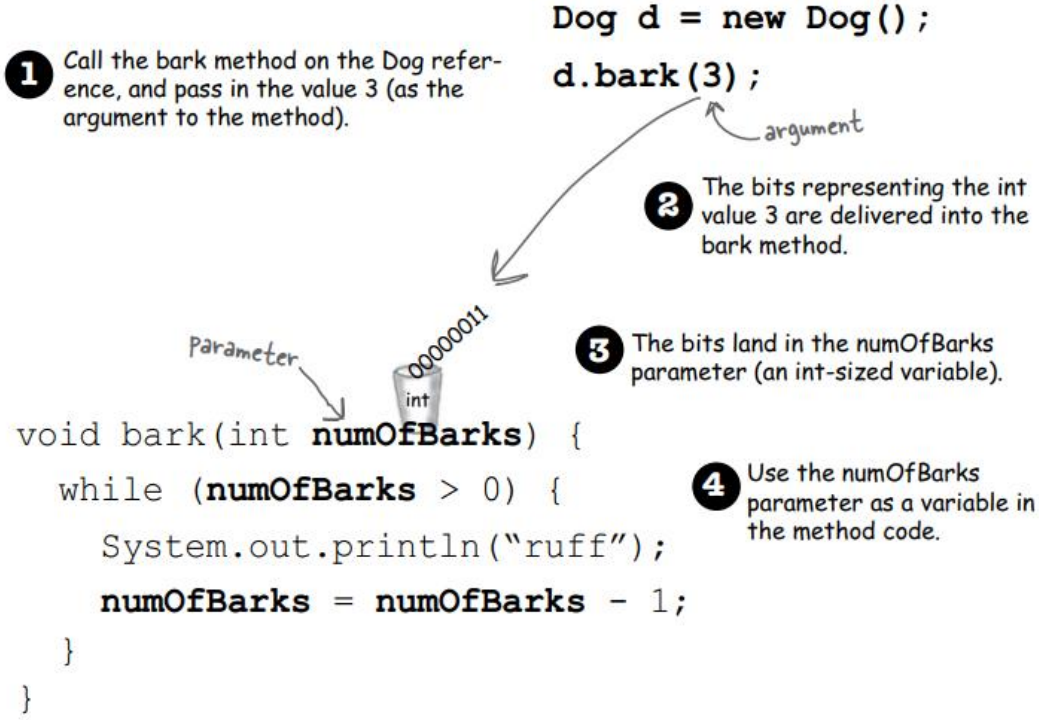

```
class DogTestDrive {

   public static void main (String[] args) {
     Dog one = new Dog();
     one.size = 70;
     Dog two = new Dog();
     two.size = 8;
     Dog three = new Dog();
     three.size = 35;

     one.bark();
     two.bark();
     three.bark();
   }
}
```

```
File Edit Window Help Playdead
%java DogTestDrive
Wooof! Wooof!
Yip! Yip!
Ruff!  Ruff!
```

- Just as you expect from any programming language, you can pass values into your methods

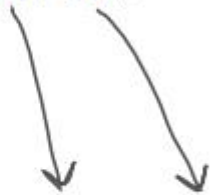
```
d.bark(3)
```

- A method uses parameters. A caller passes arguments
- **Important: If a method takes a parameter, you must pass it something** (you can't set default values for parameters in Java)



- Methods can have multiple parameters. Separate them with commas when you declare them, and separate the arguments with commas when you pass them.
- If a method has parameters, you must pass arguments of **the right type and order**

```
void go() {
   TestStuff t = new TestStuff();
   t.takeTwo(12, 34);
}

void takeTwo(int x, int y) {
   int z = x + y;
   System.out.println("Total is " + z);
}
```



The arguments you pass land in the same order you passed them. First argument lands in the first parameter, second argument in the second parameter, and so on.

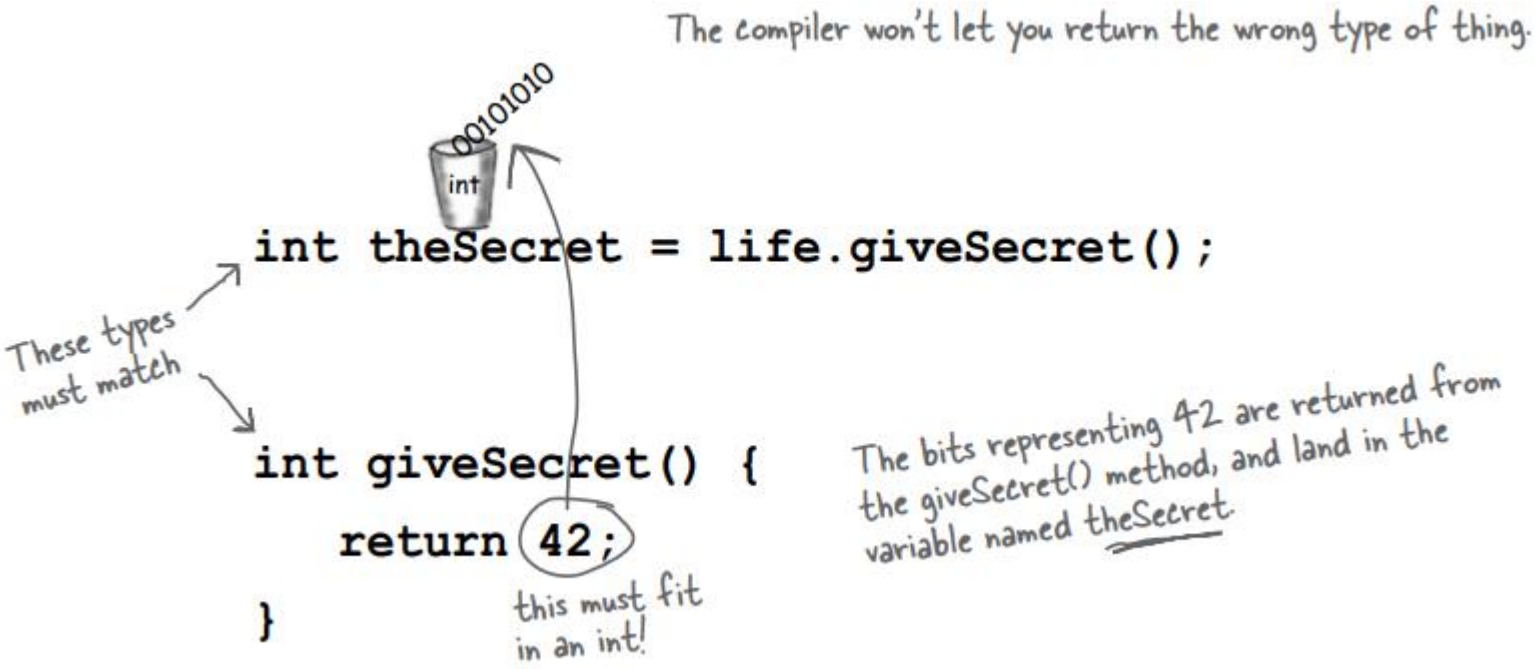
- You can pass variables into a method, as long as the variable type matches the parameter type.

```
void go() {
   int foo = 7;
   int bar = 3;
   t.takeTwo(foo, bar);
}

void takeTwo(int x, int y) {
   int z = x + y;
   System.out.println("Total is " + z);
}
```

The values of foo and bar land in the x and y parameters. So now the bits in x are identical to the bits in foo (the bit pattern for the integer '7') and the bits in y are identical to the bits in bar.

What's the value of z? It's the same result you'd get if you added foo + bar at the time you passed them into the takeTwo method

- Of course, you can get things back from a method
- We can declare a method to give a specific type of value back to the caller
- If you declare a method to return a value, you **must** return a value of the declared type

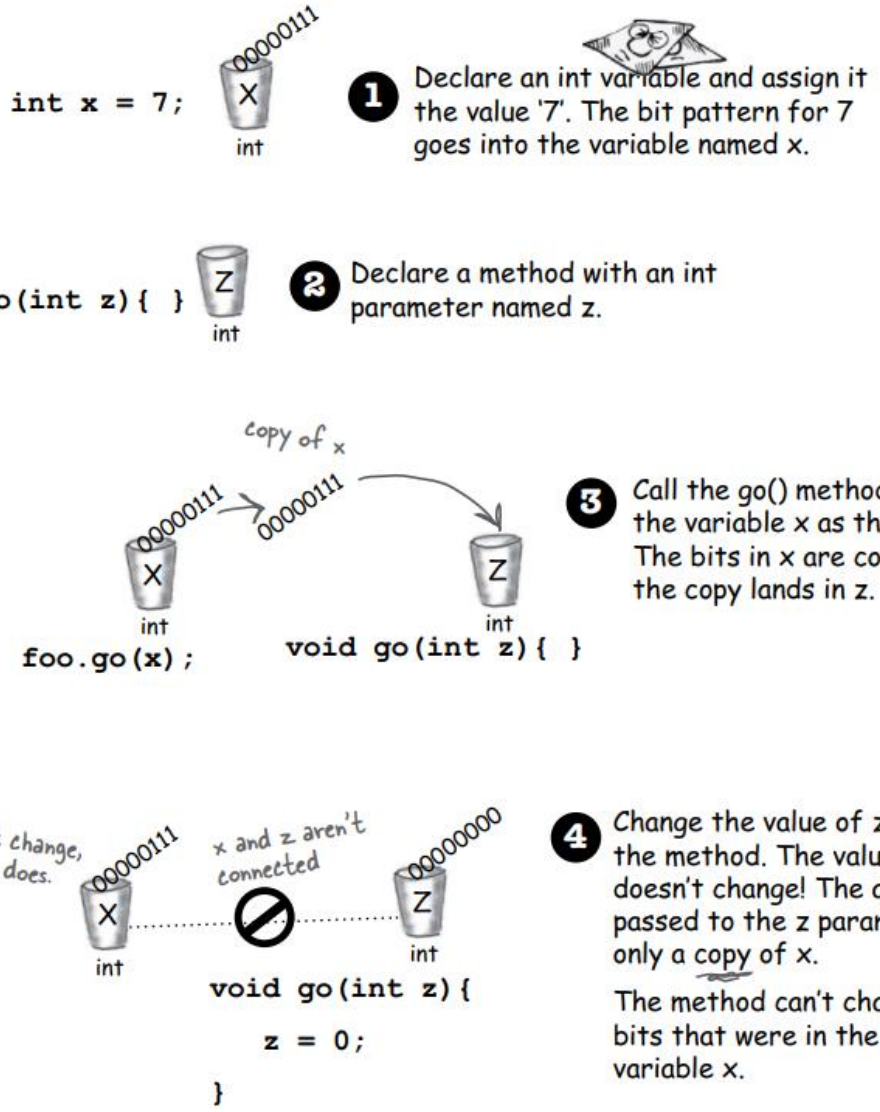


- Quiz: given the method below, which of the method calls are legal?

```
int calcArea(int height, int width) {
    return height * width;
}
```

```
1  int a = calcArea(7, 12);
2  short c = 7;
3  calcArea(c,15);
4  int d = calcArea(57);
5  calcArea(2,3);
6  long t = 42;
7  int f = calcArea(t,17);
8  int g = calcArea();
9  calcArea();
10 byte h = calcArea(4,20);
11 int j = calcArea(2,3,5);
```

- Java is **pass-by-value**



- Q: What happens if the argument you want to pass is an **object** instead of a primitive?

```
public class TestVar
{
    public static void main(String[] args)
    {
        Dog aDog = new Dog();
        aDog.name = "Pluto";
        sayName(aDog);
        System.out.println("my name is " + aDog.name);
    }

    public static void sayName(Dog d)
    {
        System.out.println("my name is" +d.name);
        d.name = "Lucky";
    }
}
```

- What's the output of the following program?

```
public class Test2
{
    public static void main(String[] args)
    {
        Dog aDog = new Dog();
        aDog.name = "Pluto";
        sayName(aDog);
        System.out.println("my name is " + aDog.name);
    }

    public static void sayName(Dog d)
    {
        System.out.println("my name is " +d.name);
        d = new Dog();
        d.name = "Lucky";
    }
}
```

- Quiz
- What is output of the following code segment?

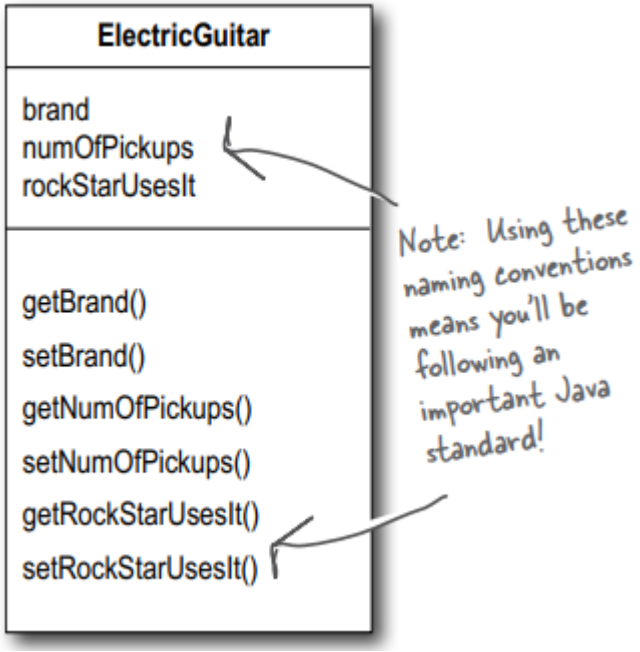
```
Dog[] d1 = {new Dog(10, "Lucky"), new Dog(6, "Mary")};
Dog[] d2 = new Dog[d1.length];
for(int i = 0; i < d1.length; i++) {
    d2[i] = d1[i];
}
d1[0].age = 12;
System.out.println(d2[0].getAge());
```

Dog
int age; String name;
getName(); getAge();

- Q: Can a method declare multiple return values? Or is there some way to return more than one value?
- A: Sort of. A method can declare only one return value. BUT… if you want to return, say, three int values, then the declared return type can be an int **array**
- Q: Do I have to return the exact type I declared?
- A: You can return anything that can be *implicitly* promoted to that type. So, you can pass a byte where an int is expected. You must use an *explicit* cast when the declared type is smaller than what you're trying to return
- Q: Do I have to do something with the return value of a method? Can I just ignore it?
- A: In Java, you don't have to assign or use the return value

- Summary
- You can pass more than one parameters into a method.
- The number and type of values you pass in must match the order and type of parameters declared by the method.
- Values passed in and out of the methods can be implicitly promoted to a larger type or explicitly cast to a smaller type.
- A method must declare a return type (otherwise the compiler complains). A void return type means you don't have to return anything.

- Getters and Setters
- Getters and Setters let you, well, get and set instance variable values, usually.
- A Getter's sole purpose in life is to send back, as a return value, the value of whatever it is that particular Getter is supposed to be Getting.
- A Setter lives and breathes for the chance to take an argument value and use it to set the value of an instance variable.



- Why we need Getter and Setter?
- In OO design, we **hate** exposing our data
- Exposed means reachable with the dot operator, as in:

```
theDog.height = 27;
```

- Make a direct change to the Dog object's instance variable is quite dangerous

```
theDog.height = 0;
```

- This would be a **Bad Thing**. We need to build setter methods for all the instance variables, and find a way to force other code to call the setters rather than access the data directly.

```
public void setHeight(int ht) {
    if (ht > 9) {
        height = ht;
    }
}
```

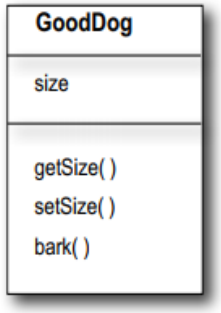
We put in checks to guarantee a minimum cat height.

- How exactly do you hide the data? With the public and private access modifiers.
- Mark your instance variables **private** and provide **public** getters and setters for access control
- This idea is called ***encapsulation***

Encapsulating the GoodDog class

Make the instance variable private.

Make the getter and setter methods public.




```
class GoodDog {

   private int size;

   public int getSize() {
     return size;
   }

   public void setSize(int s) {
     size = s;
   }

   void bark() {
     if (size > 60) {
        System.out.println("Wooof! Wooof!");
     } else if (size > 14) {
        System.out.println("Ruff!  Ruff!");
     } else {
        System.out.println("Yip! Yip!");
     }
   }
}

class GoodDogTestDrive {

   public static void main (String[] args) {
     GoodDog one = new GoodDog();
     one.setSize(70);
     GoodDog two = new GoodDog();
     two.setSize(8);
     System.out.println("Dog one: " + one.getSize());
     System.out.println("Dog two: " + two.getSize());
     one.bark();
     two.bark();
   }
}
```

Even though the methods don't really add new functionality, the cool thing is that you can change your mind later. you can come back and make a method safer, faster, better.

Any place where a particular value can be used, a *method call that returns that type can be used.*

instead of:

```
int x = 3 + 24;
```

you can say:

```
int x = 3 + one.getSize();
```

- Java has four access levels and three access modifiers. These three modifiers are: public, protected, and private. The four access levels are:
- public - any code anywhere can access the public “thing”
- protected - protected works just like default except it also allows subclasses outside the package to inherit the protected thing
- default - only code within the same **package** as the class with the default thing can access the default thing
- private - only code within the same class can access the private thing
- When you do not use any access modifier, you are in the default level.
- Most of the time you’ll use only public and private access levels

- **public**
- Use public for classes, constants (static final variables), and methods that you're exposing to other code (for example getters and setters) and most constructors.
- **private**
- Use private for virtually all instance variables, and for methods that you don't want outside code to call
- default
- Why we want to restrict access to code within the same package? Typically, packages are designed as a group of classes that work together as a related set. So it might make sense that classes within the same package need to access one another's code, while as a package, only a small number of classes and methods are exposed to the outside world.

- Declaring and initializing instance variables
- A variable declaration needs at least a name and a type

```
int size;
String name;
```

- You can initialize (assign a value) to the variable at the same time

```
int size = 420;
String name = "Donny";
```

- What happens when you call a getter method *without* initializing the instance variable?

```
class PoorDog {

   private int size;
   private String name;

   public int getSize() {
      return size;
   }
   public String getName() {
      return name;
   }

}
```

declare two instance variables, but don't assign a value

What will these return??

- Instance variables always get a default value. If you don't explicitly assign a value to an instance variable, or you don't call a setter method, the instance variable still has a value.

Type	Default value
Int	0
float	0.0
boolean	false
references	null

- Quiz: how about ***local variables***? Do they have default values?

- What's the difference between instance and local variables?
- Instance variables are declared ***inside a class*** but not within a method

```
class Horse {
   private double height = 15.2;
   private String breed;
   // more code...
}
```

- Local variables are declared ***within a method***

```
class AddThing {
   int a;
   int b = 12;

   public int add() {
      int total = a + b;
      return total;
   }
}
```

- Local variables MUST be initialized before use (otherwise the compiler complains!)

```
class Foo {
   public void go() {
      int x;
      int z = x + 3;
   }
}
```

Won't compile!! You can declare x without a value, but as soon as you try to USE it, the compiler freaks out.

- Q: What about method parameters? How do the rules about local variables apply to them?
- A: parameters are treated as ***local variables***. However, they always have a value before you use them in the method. Why?

- Argues about getter and setter
- Why we need getter and setter?
- https://stackoverflow.com/a/1568230
- The point of encapsulation is not that you should not be able to know or to change the object's state from outside the object, but that you should have a reasonable **policy** for doing it.
- Remember: not every instance variable should have a getter and a setter
- Getter and setter does not necessarily come in pairs

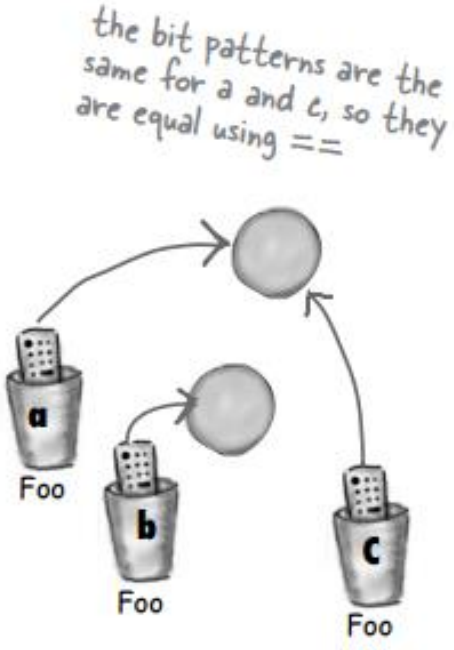
- Comparing variables (primitives or references)
- You want to know if two primitives are the same → use the == operator
- Quiz: is this true?

```
int a = 3;
byte b = 3;
if (a == b)
```

- The == operator can be used to compare two variables of any kind, and it simply compares the bits

- What does that mean when you use == on two object references?
- the == operator returns true if two reference variables refer to the same object

```
Foo a = new Foo();
Foo b = new Foo();
Foo c = a;
if (a == b) { // false }
if (a == c) { // true }
if (b == c) { // false }
```



- The == operator tests for reference equality (whether they are the same object)
- If you want to test for value equality (whether they are 'logically' equal), you have to use {OBJECT}.equals()

```
String str = "cycu";
System.out.println(str== new String("cycu"));   //false
System.out.println(str.equals("cycu"));         //true

/**
   Strings may have been "interned" by the compiler (or by manual)
   In that case, the == operator returns true because they
   refer to the same object
 */

System.out.println(str== "cycu");         //sometimes be true
```

- When you write a statement like this
  `String str = "cycu";`
- This is called **string literal**
- When you declare a string in this way, you are actually calling a special function in String class and it searches a string constant pool.
- If there already exists a string value “cycu” in this pool, then str will store the reference of that string and no new String object will be created.

* The String object will always take more time to execute than string literal

* The string constant pool is also located in heap.

- The == operator cannot compare primitives with reference variables. For example, the following code won't work:

```
String num = "2";

int x = 2;

if (x == num)  //  horrible explosion!
```

- We have to make the String "2" into the int 2. Built into the Java class library there is a class called **Integer** (that's right, an Integer class, not the int primitive), and one of its jobs is to take Strings that represent numbers and convert them into actual numbers.

- Can they compile?

A

```
class XCopy {

  public static void main(String [] args) {

    int orig = 42;

    XCopy x = new XCopy();

    int y = x.go(orig);

    System.out.println(orig + " " + y);
  }

  int go(int arg) {

    arg = arg * 2;

    return arg;
  }
}
```

B

```
class Clock {
  String time;

  void setTime(String t) {
    time = t;
  }

  void getTime() {
    return time;
  }
}

class ClockTestDrive {
  public static void main(String [] args) {

    Clock c = new Clock();

    c.setTime("1245");
    String tod = c.getTime();
    System.out.println("time: " + tod);
  }
}
```

- Extension: In Java 5.0, it has included a feature that simplifies the creation of methods that need to take a variable number of arguments.
- This feature is called varargs and it is short-form for variable-length arguments.
- A variable-length argument is specified by three periods(...)

```
void bark(int ... paras) {
    System.out.println(paras.length);
}
```

- This syntax tells the compiler that fun( ) can be called with zero or more arguments.
- As a result, here *paras* is implicitly declared as an array of type int[]
- Note: there can be only one variable argument in a method.
- Note: variable argument (varargs) must be the last argument.