Methods

CSE 142, Summer 2002

Computer Programming 1

http://www.cs.washington.edu/education/courses/142/02su/

Readings and References

- Reading (reminder)
  - Chapter 5, *An Introduction to Programming and Object Oriented Design using Java*, by Niño and Hosch
  - Chapter 5, *Introduction to Programming in Java*, Dugan
- Other References

State and Behavior

- State
  - generally held in one or more “instance variables” for the object
  - there is a set of instance variables for *each* object
  - For example

    If class Person defines **`height`** as a property of the class then each object of this type (each Person object) will have a **`height`** variable and an associated value
- Behavior
  - defined by the methods that are implemented for the class
  - the same methods are shared by all the objects created from a particular class template

Dog.java

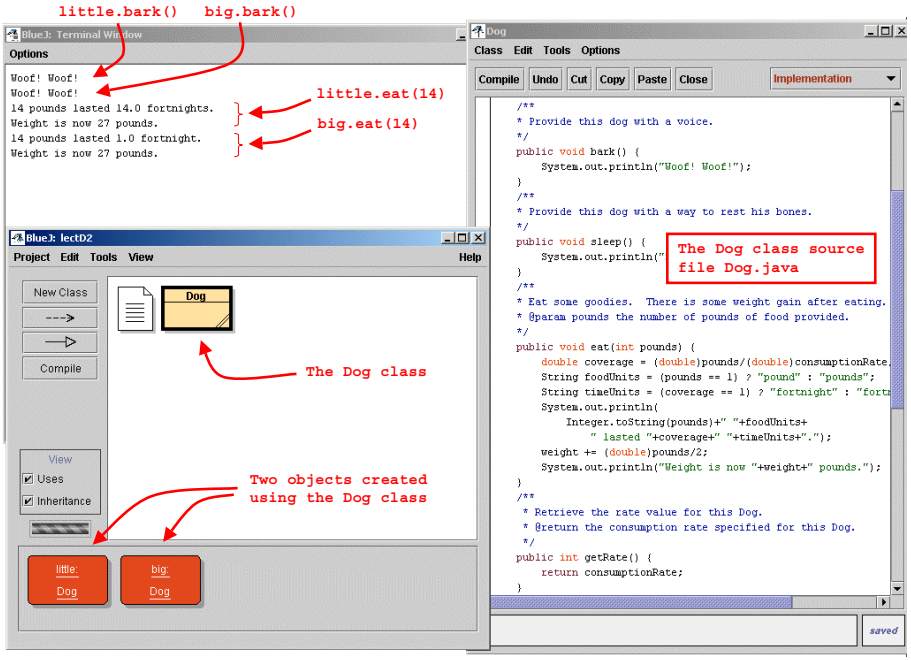
```
/**
 * Sample class for demonstrating class structure.
 */
public class Dog {
    /**
    * Create a new Dog.  This constructor supplies a default weight of 20 pounds.
    * @param rate the rate at which this dog eats, specified in pounds/fortnight.
    */
    public Dog(int rate) {
        consumptionRate = rate;
        weight = 20;
    }
    /**
     * Provide this dog with a voice.
     */
     public void bark() {
         System.out.println("Woof! Woof!");
     }
    /**
    * Provide this dog with a way to rest his bones.
    */
    public void sleep() {
        System.out.println("Snrf ... woof ... snrf ...");
    }
```

This is the **`bark()`** method of class Dog

Method

- A method is a block of statements that can be invoked to perform a particular action
  - implementing and then calling methods are the way we specify what an object does
  - invoking a method for an object is sometimes called “sending a message to the object”
- The collection of all the methods defined for a class defines what objects of that class can do
  - For example, if we define methods **`bark, sleep, eat,`** and **`getRate`** in the Dog class, then all Dog objects created from that class can do all those things.

`Dog.java`



Parameters

- Some methods know how to implement a little bit of behavior without needing any more information

```
public void bark() {
    System.out.println("Woof! Woof!");
}
```

  - A Dog implemented this way will bark exactly the same way every time this method is called
- But many methods need to know something additional in order to actually perform their task

```
/**
 * Eat some goodies.  There is some weight gain after eating.
 * @param pounds the number of pounds of food provided.
 */
public void eat(int pounds) {
    double coverage = (double)pounds/(double)consumptionRate;
    …
```

  - We use parameters (arguments) to provide this additional information

Specifying the required parameters

- The method header declares the type and name for each required parameter
- method **`eat`** has one parameter of type **`int`** named **`pounds`**

```
/**
 * Eat some goodies.  There is some weight gain after eating.
 * @param pounds the number of pounds of food provided.
 */
public void eat(int pounds) {
    double coverage = (double)pounds/(double)consumptionRate;
    …
```

  - note that there is a javadoc comment describing the purpose of the parameter

Parameter declaration

- Declaring the parameter in the parameter list is just like declaring it in the body of the code
  - The variable **`pounds`** has a type (**`int`**) and it can be used in expressions exactly the way any other variable in the method is used
- You can declare several parameters in the formal parameter list of a method
  - but try to keep the number down
  - if there are too many, the users of this method (you and other programmers) will have a hard time keeping straight just which parameter is which

Some examples from class java.lang.String

- **`toLowerCase()`**

  Converts all of the characters in this String to lower case using the rules of the default locale

- **`startsWith(String prefix)`**

  Tests if this string starts with the specified prefix

- **`substring(int beginIndex, int endIndex)`**

  Returns a new string that is a substring of this string

- **`regionMatches(int toffset, String other, int ooffset, int len)`**

  Tests if two string regions are equal

Parameter variables used in body of method

```
public String substring(int beginIndex, int endIndex) {
  if (beginIndex < 0) {
    throw new StringIndexOutOfBoundsException(beginIndex);
  }
  if (endIndex > count) {
        throw new StringIndexOutOfBoundsException(endIndex);
  }
  if (beginIndex > endIndex) {
        throw new StringIndexOutOfBoundsException(endIndex-beginIndex);
  }
  return ((beginIndex == 0) && (endIndex == count)) ? this :
        new String(offset+beginIndex, endIndex-beginIndex, value);
}
```


What about the values of these parameters?

- The values (if any required) must be supplied by the caller
  - the compiler checks that the type of the actual values provided matches the type of the parameters that were specified by the method and will not compile the code if they are incompatible

```
rover.bark();
rover.eat(2);
rover.bark();
rover.eat(14);
```

- The method can vary its behavior in whatever way is appropriate based on the actual values of the parameters

Supplying an actual value

- The actual values don't have to be literals like 2 or 14
- You can supply a variable name in the call, and the current value of the variable will be provided to the method

```
int currentFoodAmount = 4;
Dog jack = new Dog(2);
jack.eat(currentFoodAmount);
currentFoodAmount = 20;
jack.eat(currentFoodAmount);
```

- In this example, the method **`eat`** executes twice, once with **`pounds`** equal to 4, and then again with **`pounds`** equal to 20
- Notice that the method always associates the value with the name **`pounds`**, even though the caller might be using something else

The actual arguments can be expressions

- You can calculate the value to be passed right in the call to the method if that is appropriate
  - Recall: **`substring(int beginIndex, int endIndex)`**

```
int beginIndex = 0;
String myName = "Doug Johnson";
String twoChar = myName.substring(beginIndex, beginIndex+2);
```

  - **`twoChar` is now a reference to a String containing "Do"**
- If necessary and possible, the compiler will convert the value provided by the caller to the type of the value that was requested by the method in the formal parameter list

Returning a value to the caller

- A method can also return a value to its caller
- For example, recall the "accessor" methods that allow you to ask an object what some part of its current state is

```
public int getX()
public int getWidth()
```

- The word **`int`** in the above examples specifies the type of value that the method returns

```
/**
 * Get current X value.
 * @return the X coordinate
 */
public int getX() {
    return x;
}
```

SlideX.java

Documentation for methods

- Short, useful description of the purpose of the method.
  - javadoc takes the first sentence of this description and uses it in the summary part of the documentation page
  - If there is important background information on how to use the method, it should follow the initial sentence.
- All parameters
  - use an @param entry for each parameter
- The return value, if any
  - use an @return tag if appropriate
- Error exceptions (we will discuss these late in the quarter)
  - use a @throws tag if appropriate

SlideX.java - generate javadoc

Constructors

- A constructor is used to create a new object of a particular class
- Constructors are special methods that get called with the **new** operator

```
Dog rover = new Dog(10);
```

  - we can't use the rover.bark() pattern to call the Dog() constructor because rover doesn't have a value until *after* we call the constructor
- The name of a constructor is the same as the name of the class
  - in this case **Dog(int rate)** is a constructor for the class **Dog**
- You can think of the constructor as a method that initializes everything according to what the caller has specified, using whatever default values might be appropriate

Multiple Constructors

- There are often several constructors for any one class
- They all have the same name (the name of the class)
- They must differ in their parameter lists
  - the compiler can tell which constructor you mean by looking at the list of arguments you supply when you call the constructor

```
Rectangle deadTree;
Rectangle liveTrunk;

deadTree=new Rectangle(150,150,10,50);
liveTrunk=new Rectangle(200,210,10,50,Color.orange,true);
```

- There is no return value specified for any constructor, because a constructor always fills in the values in a new object

Cat.java