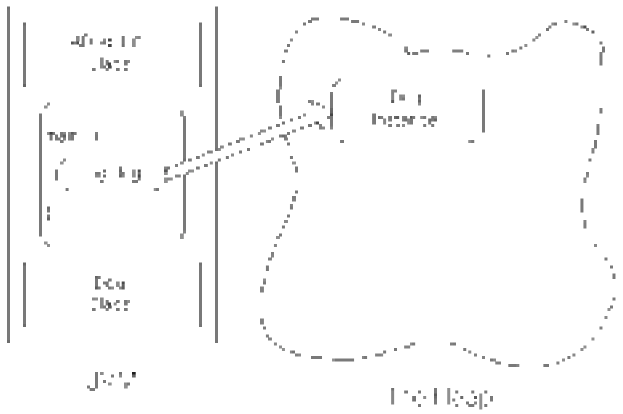
In Java, as with other languages, a program allocates objects dynamically. Java's storage allocation operator is `new`:

Storage Allocation Syntax
`new <data-type>(<arguments>...)`
`<data-type> <variable> = new <data-type>(<arguments>...)`

The `new` operator asks the Java runtime environment to create dynamically (on the fly) an instance of a user-defined data type, for example, "`new Dog()`". You can also associate the instance with a variable for future reference (hence, the term *reference variable*), for example, "`Dog bowwow = new Dog()`". The data type for the reference variable `bowwow` must be specified to the left of the variable name, in this case, "`Dog bowwow`".

Objects receive their storage on/from the heap, which is simply a memory pool area managed by the Java interpreter. The following diagram illustrates the memory allocation for the class files, plus the instance of `Dog` allocated on the heap:



Java Data Types

Java supports a variety of primitive (built-in) data types such as `int` for representing integer data, `float` for representing floating-point values, and others, as well as `class`-defined data types that exist in supporting libraries (Java packages). (All Java primitive data types have lowercase letters.)

The `String` class is defined in `java.lang`, the core package of supplemental class definitions that are fundamental to Java programming. A more complete reference to this class includes the package specification `java.lang.String`. (By convention, class names have mixed-case letters: the first letter of each word is uppercase.)

The Java language has the following primitive types:

Primitive Type	**Description**
`boolean`	`true/false`
`byte`	8 bits
`char`	16 bits (UNICODE)
`short`	16 bits
`int`	32 bits
`long`	64 bits
`float`	32 bits IEEE 754-1985
`double`	64 bits IEEE 754-1985

For an overview of common nonprimitive data types, that is, class definitions provided by the Java environment, see the `java.lang` package within the standard Java distribution documentation, or the documentation supplied with your Java IDE.

Method Overloading

Not all dogs sound alike, however, so to add a little barking variety to the `Dog` implementation, we should define an alternative `bark()` method that accepts a barking sound as a string:

```
class Dog {
  void bark() {
    System.out.println("Woof.");
  }

  void bark(String barkSound) {
    System.out.println(barkSound);
  }
}
```

This version of `Dog` is legal because even though we have two `bark()` methods, the differing signatures allows the Java interpreter to chose the appropriate method invocation. The method definition syntax "`void bark(String barkSound)`" indicates that this variation of `bark()` accepts an argument of type `String`, referred to within the method as `barkSound`.

As another example of method overloading, consider the program `DogChorus`, which creates two dogs and elicits a different barking behavior for each dog:

```
public class DogChorus {
  public static void main(String[] args) {
    Dog fido = new Dog();
    Dog spot = new Dog();
    fido.bark();
    spot.bark("Arf.  Arf.");
    fido.bark("Arf.  Arf.");
    System.exit(0);
  }
}
```

Because `Dog` supports two different barking behaviors, both defined by methods named `bark()`, we can design our program to associate either barking behavior with, say, `fido`. In `DogChorus`, we invoke different barking behaviors for `fido` and `spot`. Note that `fido` changes his bark after hearing `spot`.

Instance Variables

So far, we're defining instances of objects in terms of their behaviors, which in many cases is legitimate, but, in general, user-defined data types incorporate state variables as well. That is, for each instance of `Dog`, it's important to support variability with respect to characteristics such as hair color, weight, and so on. State variables that distinguish one instance of `Dog` from another are called *instance variables*.

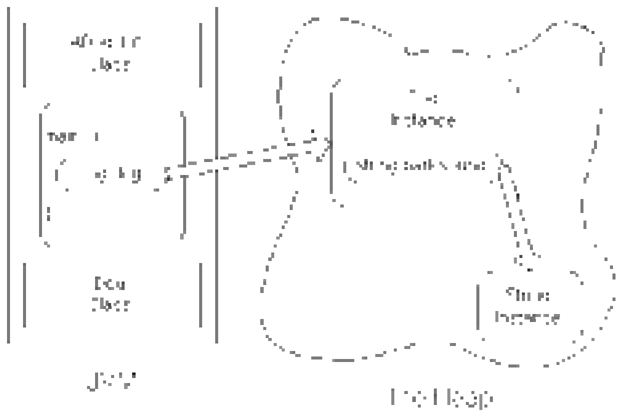
Now suppose we add an instance variable to reflect a particular dog's barking sound; a `String` instance can represent each dog's bark:

```
class Dog {
  String barkSound = new String("Woof.");

  void bark() {
    System.out.println(barkSound);
  }

  void bark(String barkSound) {
    System.out.println(barkSound);
  }
}
```

The definition of `Dog` now includes the instance variable `barkSound`. Each time a new instance of `Dog` is created, that instance will include a reference variable for an instance of `String` representing that particular dog's bark. This *instance* variable is initialized to the default value of `"Woof."`. Consider the line of code

```
String barkSound = new String("Woof.");
```

This statement allocates an instance of `String`, initializes it with the value `"Woof."` (provided in the parentheses following the `String` class name), and stores this data under the reference variable `barkSound`. Note that the reference variable `barkSound` is part of each instance of `Dog`, but that it references an instance of `String` that's allocated on the heap, as is the instance of `Dog`:



Now that the default barking behavior is represented by an instance variable, we can remove the `"Woof."` data from the original `bark()` method, replacing it with a reference to the current value of `barkSound`:

```
void bark() {
  System.out.println(barkSound);
}
```

That is, we've transferred unconditional state data from a method definition into an instance variable that can vary from one dog to the next, and perhaps more importantly, for a particular dog, its value can change dynamically.

Access Methods

In order for the value of an instance variable to vary over time, we must supply a method to change its value; such a method is typically referred to as an *access method*. By convention, a method that's provided simply to affect a change to an instance variable's value begins with the word "set":

```
  void setBark(String barkSound) {
    this.barkSound = barkSound;
  }
```

This method is interesting because it uses two different variables with the same name, `barkSound`. First, the `barkSound` defined as an parameter is the new barking sound. Any unqualified reference to `barkSound` within this method refers to this data passed as an argument. We also have, however, a `barkSound` instance variable for each dog that is instantiated. With Java, we can use the special "instance handle" `this` to refer to the *current* instance of `Dog`. Hence,

```
    this.barkSound = barkSound;
```

replaces the current value of the instance variable (`this.barkSound` with the new value passed as an argument (`barkSound`) to `setBark()`.

To put the `this` variable in perspective, suppose we create an instance of `Dog` referred to as `fido`. Then, if we execute `setBark()` with respect to `fido`, namely,

```
    fido.setBark("Ruff.");
```

the `this` instance in `setBark()` is `fido` and, in particular, `this.barkSound` is the `barkSound` instance variable for the `fido` object.

In the following version of `DogChorus`, we create an object, `fido`, change its barking characteristic from the default `"Woof."` to `"Ruff."`, and then invoke the barking behavior:

```
public class DogChorus {
  public static void main(String[] args) {
    Dog fido = new Dog();
    fido.setBark("Ruff.");
    fido.bark();
    System.exit(0);
  }
}
```

With this modification, the characteristics of an object such as `fido` are reflected by both the current values of instance/state variables *and* the available behaviors defined by the methods in `Dog`.

Instance Methods

The type of methods we're designing at present are called *instance methods* because they are invoked relative to a particular instance of a class. For this reason, an instance method can reference an instance variable directly, without the `this` qualifier, as long as there is no variable name conflict, for example,

```
  void bark() {
    System.out.println(barkSound);
  }
```

In this case, the no-argument version of `bark()` references the instance variable `barkSound` directly. As implied by the `setBark()` definition, however, we could also write `bark()` as follows:

```
  void bark() {
    System.out.println(this.barkSound);
  }
```

Here, there are no other variables within (local to) `bark()` named `barkSound`, so these implementations are equivalent.

Conditional Execution

So far, within each method we've used sequential execution only, executing one statement after another. Like other languages, Java provides language constructs for conditional execution, specifically, `if`, `switch`, and the conditional operator `?`.

Conditional Constructs

```
if (<boolean-expression>)
  <statement>...
else
  <statement>...
```

```
switch (<expression>) {
  case <const-expression>:
    <statements>...
    break;

  more-case-statement-break-groups...

  default:
    <statements>...
}
```

```
(<boolean-expression>) ? <if-true-expression>
: <if-false-expression>
```

The more general construct, `if` has the syntax:

```
if (<boolean-expression>)
  <statement>...
```

where <*statement*>... can be one statement, for example,

```
x = 4;
```

or multiple statements grouped within curly brackets (a statement group), for example,

```
{
  x = 4;
  y = 6;
}
```

and <*boolean-expression*> is any expression that evaluates to a `boolean` value, for example,

Boolean Expression	**Interpretation**
`x < 3`	`x` is less than `3`
`x == y`	`x` is equal to `y`
`x >= y`	`x` is greater than or equal to `y`
`x != 10`	`x` is not equal to `10`
<*`variable`*>	*`variable`* is `true`

If the Boolean expression evaluates to `true`, the statement (or statement group) following the `if` clause is executed.

Java also supports an optional `else` clause; the syntax is:

```
if (<boolean-expression>)
  <statements>...
else
  <statements>...
```

If the Boolean expression evaluates to `true`, the statement (or statement group) following the `if` clause is executed; otherwise, the statement (or statement group) following the `else` clause is executed.

Boolean expressions often include one or more Java comparison operators, which are listed in the following table:

Comparison	**Interpretation**

Operator	
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
!=	not equal to

Returning to our user-defined type `Dog`, we can add additional state variables for more flexible representation of real-world objects. Suppose we add instance variables `gentle` and `obedienceTrained`, which can be `true` or `false`:

```
class Dog {
  String barkSound = new String("Woof.");
  boolean gentle = true;
  boolean obedienceTrained = false;
  ...
```

In Java, Boolean values are literals (case is significant), and `boolean` variables accept either value. For `gentle` and `obedienceTrained` there are no `new` operators because we're not creating objects--we're creating primitive variables and assigning them the default values `true` and `false`.

Access methods provide the flexibility for modifying these instance variables on a dog-by-dog basis:

```
  void setGentle(boolean gentle) {
    this.gentle = gentle;
  }

  void setObedienceTrained(boolean trained) {
    obedienceTrained = trained;
  }
```

Note that the reference to `obedienceTrained` in `setObedienceTrained()` does not require qualification with `this` because there is no local variable by the same name.

Methods that Return Values

With these new variables, we can provide convenience methods such as

```
boolean isGoodWithChildren() {
  if (gentle == true && obedienceTrained == true)
    return true;
  else
    return false;
}
```

This method introduces additional Java syntax. First, instead of the modifier `void`, `isGoodWithChildren()` provides a return type, in this case, `boolean`. With the replacement of `void` by either a primitive, system-, or user-defined data type, the method must provide a `return` statement for every execution path possible through the method--the Java compiler enforces this "contract."

A return statement has the syntax

```
return <value>;
```

where <*value*> is any expression that evaluates to the appropriate return data type.

For `isGoodWithChildren()`, if the `if` statement's Boolean expression evaluates to `true`, the first `return` executes, which terminates the method execution and returns the result of the evaluation, which in this case is simply the literal value `true`. If the Boolean expression evaluates to `false`, the code block following the `else` clause is evaluated, in this case, a single return statement that returns `false`.

In this example, the Boolean expression is compound--it includes two expressions, each with the `==` comparison operator. The comparative expressions are linked with the *logical and* operator `&&`; hence, the complete expression evaluates to `true` only if both subexpressions evaluate to `true`.

Boolean expressions often include one or more Java logical operators, which are listed in the following table:

Logical Operator	**Interpretation**
`&&`	and ("short-circuit" version)
`&`	and ("full-evaluation" version)

`||`	or ("short-circuit" version)
`|`	or ("full-evaluation" version)

With the "short-circuit" versions, evaluation of subsequent subexpressions is abandoned as soon as a subexpression evaluates to `false` (in the case of `&&`) or `true` (in the case of `||`).

Although `isGoodWithChildren()` provides a full illustration of `if`, including the optional `else` clause, this method can be coded more succinctly. Java, like C, is a syntactically powerful language. First, we can actually remove the `if` because the return values correspond to the Boolean expression, that is, if the Boolean expression evaluates to `true`, return `true`; otherwise, return `false`. The more concise implementation is:

```
boolean isGoodWithChildren() {
  return (gentle == true && obedienceTrained == true);
}
```

One more reduction is possible. Note that each subexpression involves a `boolean` variable compared to the `boolean` literal `true`. In this case, each subexpression can be reduced to the `boolean` variable itself:

```
boolean isGoodWithChildren() {
  return (gentle && obedienceTrained);
}
```

Access Methods Revisited

`Dog` provides `set`-style access methods for modifying an instance variable. At times, it's necessary to retrieve an instance variable's value. Typically, if a class has instance variables that support set operations, they support get operations as well. For each of our set methods, we should code a corresponding get method, for example,

```
boolean getObedienceTrained() {
  return obedienceTrained;
}
```

Note that in the case of `boolean` instance variables such as `obedienceTrained`, some programmers prefer the `is`-style naming convention over the `get`-style and some programmers like to provide both:

```
boolean isObedienceTrained() {
  return obedienceTrained;
  }
```

Note that `isGoodWithChildren()` from the previous section is not really an access method--it does not return (report) an instance variable's value. Instead, it combines higher level, meaningful information with respect to *an instance of* the class `Dog`.

Iterative Execution

Java provides the `while`, `do-while`, and `for` language constructs for iterating over a statement (or statement group) multiple times. `while` is the more general iterative construct; `for` is the more syntactically powerful.

Iterative Constructs

`while (<boolean-expression>)`
`  <statements>...`

`do`
`  <statements>...`
`while (<boolean-expression>)`

`for (<init-stmts>...; <boolean-expression>; <exprs>...)`
`  <statements>...`

With iteration, we can (to the dismay of our neighbors) make barking behavior repetitive:

```
  void bark(int times) {
    while (times > 0) {
      System.out.println(barkSound);
      times = times - 1;
    }
  }
```

Thus, with yet another `bark()` method, we support the object-oriented task of sending a `Dog` instance the `bark` *message*, accompanied with a *message request* (method argument) for *n* barks, represented in the method definition by the parameter `times`.

`DogChorus` now begins to reflect its name:

```
public class DogChorus {
  public static void main(String[] args) {
    Dog fido = new Dog();
    Dog spot = new Dog();
    spot.setBark("Arf.  Arf.");
    fido.bark();
    spot.bark();
    fido.bark(4);
    spot.bark(3);
    new Dog().bark(4); // unknown dog
    System.exit(0);
  }
}
```

`DogChorus` now displays the following:

```
Woof.
Arf.  Arf.
Woof.
Woof.
Woof.
Woof.
Arf.  Arf.
Arf.  Arf.
Arf.  Arf.
Woof.
Woof.
Woof.
Woof.
```

Note the line in the source code with the comment "`// unknown dog`". As we mentioned, Java is a dynamic language, another example of which we illustrate here. An "unnamed" `Dog` is instantiated on the fly (appears suddenly from down the street), joins in the chorus, and then disappears.

That is, with Java we can create an instance of any class on the fly, without assigning it to a reference variable for future use (assuming we need it only once), and use it directly. Furthermore, the Java syntax and order of evaluation for "`new` *`<data-type>`*`()`" is designed so that we can do this without having to group the `new` operation in parentheses.

Java's MultiFunction Operators

We've mentioned that Java is syntactically powerful, like C. Java supports several powerful, multifunction operators described in the following table:

Multifunction Operator	**Interpretation**
`++`	increment (by `1`)
`--`	decrement (by `1`)

`+=`	ncrement (by specified value)
`-=`	lecrement (by specified value)
`*=`	nultiply (by specified value)
`/=`	livide (by specified value)
`&=`	itwise and (with specified value)
`|=`	itwise inclusive or (with specified value)
`^=`	itwise exclusive or (with specified value)
`%=`	nteger remainder (by specified value)

These operators (actually operator combinations) are multifunction operators in the sense that they combine multiple operations: expression evaluation followed by variable assignment. For example, `x++` first evaluates `x`, increments the resulting value by `1`, assigns the result back to `x`, and "produces" the initial value of `x` as the *ultimate evaluation*. In contrast, `++x` first evaluates `x`, increments the resulting value by `1`, assigns the result back to `x`, and produces the updated value of `x` as the ultimate evaluation.

Note that `x++` and `++x` are equivalent in standalone contexts where the only task is to increment a variable by one, that is, contexts where the ultimate evaluation is ignored:

```
int x = 4;
x++; // same effect as ++x
System.out.println("x = " + x);
```

This code produces the output:

```
x = 5
```

In the call to `println()`, the argument is a concatenation of the string `"x = "` and `x` after its conversion to a string. String operations, including the use of `+` for string concatenation, are discussed in the section Strings.

In the following context, the placement of the increment operator is important:

```
int x = 4;
int y = x++;
int z = ++x;
System.out.println(
  "x = " + x + " y = " + y + " z = " + z);
```

This code produces the output:

```
x = 6 y = 4 z = 6
```

The following table includes examples and interpretations:

Multifunction Operator	**Example**	**Pedestrian Equivalent**
`++`	`x++, ++x`	`x = x + 1`
`--`	`x--, --x`	`x = x - 1`
`+=`	`x += y`	`x = x + y`
`-=`	`x -= y`	`x = x - y`
`*=`	`x *= y`	`x = x * y`
`/=`	`x /= y`	`x = x / y`
`&=`	`x &= y`	`x = x & y`
`|=`	`x |= y`	`x = x | y`
`^=`	`x ^= y`	`x = x ^ y`
`%=`	`x %= y`	`x = x % y`

Note that Java restricts bitwise operations with `&`, `|`, and `^` to integer values, which makes sense. Further information on binary operations is available in many introductory computer science texts.

Using the decrement operator we can rewrite the iterative operation in `bark()` somewhat more succinctly as follows:

```
  void bark(int times) {
    while (times > 0) {
      System.out.println(barkSound);
      times--;
    }
  }
```

Further code reduction is possible:

```
  void bark(int times) {
    while (times-- > 0)
      System.out.println(barkSound);
  }
```

In this case, we use the ultimate evaluation of `times` in the `while` construct's Boolean expression that controls iteration (loop continuation). In particular, `times--` decrements the variable but "produces" the initial value of `times` for the expression evaluation proceeding the greater-than comparison operation.

Strings

As mentioned, `String` is a system-defined class--not a primitive--defined in `java.lang`, the core package of supplemental class definitions included with all Java distributions. The `lang` package is considered so essential that no steps are neccessary by the programmer to use its classes. A quick examination of `java.lang.String` shows an extensive number of methods for string manipulation.

The `lang` package also provides a complementary class, `java.lang.StringBuffer`. `String` instances are immutable; that is, they cannot be modified. To use equivalent terminology, `String` operations are *nondestructive*. A programmer simply creates strings, uses them, and when there is no further reference to them, the Java interpreter's garbage collection facility (Java Garbage Collection) recovers the storage space. Most of the string-oriented tasks necessary for normal programming can be accomplished with instances of `String` (which is quite efficient), for example, creating string constants, concatenating strings, and so on.

`StringBuffer`, on the other hand, is more powerful. It includes many methods for destructive string operations, for example, substring and character-level manipulation of strings such as splicing one or more characters into the middle of a string. A good rule of thumb is to use `String` wherever possible, and consider `StringBuffer` only when the functionality provided by `String` is inadequate.

In an earlier section, we performed a common display operation involving strings, namely:

```
System.out.println("x = " + x);
```

This simple line of code demonstrates several string-related issues. First, note that `println()` accepts one argument, which is satisfied by the result of the expression evaluation that includes `+`. In this *context*, `+` performs a string concatenation.

Because + is recognized by the Java compiler as a string concatenation operator, the compiler will automatically generate the code to convert any non-`String` operands to `String` instances. In this case, if `x` is an `int` with the value `5`, its value will be converted, generating the string constant `"5"`. The latter is concatenated with `"x = "` producing `"x = 5"`, the single argument to `println()`.

Thus, in the earlier example, we had the code

```
int x = 4;
x++; // same effect as ++x
System.out.println("x = " + x);
```

which produces the output:

```
x = 5
```

Note that you can use this automatic conversion and concatenation anywhere, not just as an argument to a method such as `println()`. This feature is incredibly powerful and convenient, demonstrating once again Java's syntactic power:

```
String waterCoolerGreeting;
if (employee.getAge() > 40) {
  waterCoolerGreeting =
    employee.getFirstName() +
    "!  Wow, what's it like to be " +
    employee.getAge() + "?";
}
else
  waterCoolerGreeting =
    "Hi, " + employee.getFirstName() + "!"
```

Another issue is the use of a double quote-delimited character sequence directly, for example, `"x = "`. Because `String` is a class, the general way to create a string instance is:

```
String prompt = new String("x = ");
```

Note that we have to provide a string in order to create a string! As a convenience for the programmer, Java always recognizes a sequence of characters between double quotes as a string constant; hence, we can use the following short-cut to create a `String` instance and assign it to the reference variable `prompt`:

```
String prompt = "x = ";
String barkSound = "Woof.";
```

One final issue is that automatic conversion to a `String` instance works for