Quick Reference

Basic non-object oriented Program Organization:

```
public class ProgramName {

    //This is a comment.  The computer will ignore it.


    public static void main (String args[]) {

        String name = "Dr. Octopus";

        int age = 45;


        System.out.print("Hello ");

        System.out.print(name + ".  You are " + age + " years old.");

    }

}
```

Main Method: This is where the program starts running.

Code Block: You can define a sequence of statements using a code block. Make sure you use the curley braces and that they match! Indent all the statements inside the code block.

```
public class ProgramName2 {

    public static int point1_x, point1_y, point2_x, point2_y;


    public static void main (String args[]) {

        point1_x = 0;   point1_y = 0;

        point2_x = 10;   point2_y = 4;


        double d = distance(point1_x, point1_y, point2_x, point2_y);

        System.out.println("Distance is " + d);

    }


    public static double distance(int x1, int y1, int x2, int y2) {

        int dx = x1-x2;

        int dy = y1-y2;


        return Math.sqrt(dx*dx + dy*dy);

    }

}
```

Class variables: You can declare variables that you can use anywhere in your program (until we get to object-oriented programming later). They are typically declared at the top of the program (outside the main method). The key word *static* designates that there will be one copy of each variable for the whole program.

Helper methods: You can declare new methods like this. See the **methods** section for more detail on creating your own methods, and the difference between *static* and non-static methods.

Variables

Generic pattern for declaring and assigning variables:

```
<Data Type> variable_name = <Value>;
```

e.g.

```
int my_num = 5;
```

```
String name = "foo";
```

```
double area = 3.14*5*5;
```

```
Turtle morris = new Turtle();
```

Naming: You can name your variable anything you like as long as it...

contains only letters, numbers, underscores (_) or dollar signs.

does not start with a number.

is not a reserved word in Java [such as *main* or *int*]

Do not include white space in your variable names!

Primitive Data Types

There are 8 basic data types (called *primitive types*) that Java recognizes.

byte - An integer	An 8-bit integer which stores values from -127 to 128 inclusive. Can be used if you need a large array of integers with small values.
short - An integer	A 16-bit integer which stores values from -32,768 to 32,767 inclusive.
int - An integer	A 32-bit integer which stores values from -2,147,483,648 to 2,147,483,647 inclusive.
long - An integer	A 64-bit integer which stores values from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 inclusive.
float	A 32-bit real number
double	A 64-bit real number
boolean - *true* or *false*	The boolean data type has only two possible values: *true* and *false*. You can use this to keep track of whether or not some condition has occurred (this is called a flag)
char - A single character	A 16-bit Unicode character. Can be treated as an integer in the range 0 to 65,535

Strings

The String data type is used for text. It is *not* a primitive data type--it's an object. But since it's used all the time, I've included it here. You declare and assign Strings thusly:

```
//declaring a String literal
String myName = "Mr. Dobervich";

//comparing two Strings
if (myName.equals("Dr. Octapus")) {
        System.out.println("yes!");
}
```

Numeric Operators

Name	Name	Example
+	Plus	n + 3
-	Minus	45 - n
*	Times	34*n
/	Divide	n / 3
%	Remainder	n % 5 (if n is 25, this equals 4, because 24/5 has remainder 4)
++	Increment (add 1)	n++
--	Decrement (subtract 1)	n--

Order of Operations and Using Parenthesis: Java evaluates expressions using standard mathematical order of operations. It's a good idea to use parenthesis to make statements completely unambiguous:

NO	YES
x + y / 100 // ambiguous	(x + y) / 100 // unambiguous, recommended

Examples:

```
int a = 3; int b = 4; double c = 3.4;
double d = Math.Sqrt(a*a + b*b);          // distance from a to b
a = a + 4;                      // add 4 to a, save back into a
a += 4;                         // short-hand for adding 4 to a
b = a + 4;                      // add 4 to a, save back into b


int n = 25;
boolean is_n_even, is_n_positive;
is_n_positive = (n > 0);        // test if n is positive
is_n_even = (n % 2 == 0);       // tsst if n is even


char my_char = 'd';             // set my_char to 'd'
```

IMPORTANT NOTE: Division with Integers

If you divide two integers, Java will return an integer. For example...

```
        myNumber = 5 / 9;          // BAD:  This returns 0
```

By adding a decimal, you telll the computer to treat it as a *double* which gives you what you want.

```
        myNumber = 5.0 / 9;        // GOOD
```

Type Conversions (Casting)

Automatically converting from one primitive type to another is called *casting*.

```
    variableName = (newType)<expression>;


    Examples:
    float x = 5.67;
    int i = (int) x;    // cast float to int
```

If-Statements (Conditionals)

Here are three "templates" for making an if-statement.

1. Single Statement If (uncommon)	2. If "block" (common)	3. If...Else (common)

`if` (test)
 	
    statement;
	`if` (test)
	`{`
	
	    statement;
	
	    statement;
	
	    ...
	
	`}`
		`if` (test)
		`{`
		
		    statement1;
		
		    ...
		
		    statement10;
		`} else {`
		
		    statement11;
		
		    ....
		
		    statement20;
		
		`}`

"If test is true, then do this one statement"	"If test is true, then execute these statements in order"	"If test is true, do statements 1 to 10. OTHERWSISE (if test is false) do statements 11 to 20"
Examples		

`if (a < 0)`
 	
   `a = 0;`
	`if (a < 0) {`
	   `a = 0;`
	   `c++;`
	`}`
		`if (a < b) {`
		   `c = b;    // a is smaller`
		`} else {`
		   `c = a;    // a not smaller`
		`}`
		
		`// now c equals whichever of a`
		`or b was smallest`

Comparison Operators

These are used to compare values, often in If-statements.

Symbol	Name	Example
<	less than	`if (i < 100) { ... }`
<=	less than or equal to	`if (i <= 100) { ... }`
>	greater than	`if (i > 100) { ... }`
>=	greater than or equal to	`if (i >= 100) { ... }`
==	equal to	`if (i == 100) { ... }`
!=	not equal to	`if (i != 100) { ... }`

Compound Statements

You may use these symbols for creating multiple conditions in your if-statements. Note that each statement has to occur inside a pair of parenthesis. You may string together as many as you wish.

Symbol	Name	Example
&&	AND	`if ( (i > 10) && (i < 20) ) { ... }` English: "BOTH i is greater than ten AND i is less than 20"
||	OR	`if ( (i < 10) || (i > 20) ) { ... }` English: "i is less than 10 or i is larger than 20 (or both)"
!	NOT	`if ( !(i > 10) ) { ... }` English: "It is not the case that i is greater than 10"

Examples:

```
if ((i == 0) && (j == 0))
{
    System.out.println("both are 0");
    ...
}


if ((i == 0) || (j == 0))
{
    System.out.println("i or j or both are 0");
    ...
}


if (i > 10)
{
    System.out.println("i is > 10");
} else {
    System.out.println("too small!");
    i++;
}


if ((a > 10) && (a < 20))
{
System.out.println("a is between 10 and 20");
}
```

You can also test to see if something is true... for example

```
boolean testResults = runOurTest();
if (testResults) {
        System.out.println("It worked!");
} else {
        System.out.println("It didn't work!");
}
```

COMMON MISTAKES!   Here are three common mistakes using if-statements.

1). Difference between = and ==

A single = sign is for assignment. It sets a variable on the left to the value on the right.

A double == sign is for testing equality. It returns true or false.

WRONG	Right

if (a = 5) {
    ... // this sets a to be 5.
}
	if (a == 5) {
	    .... // this tests to see if a equals 5
	}

2). Comparing primitive types and comparing Objects (e.g. Strings)

You can use == to compare any of the primitive data types (int, char, etc.)

If you want to compare Strings you have to use a special method called **.equals()** as shown below.
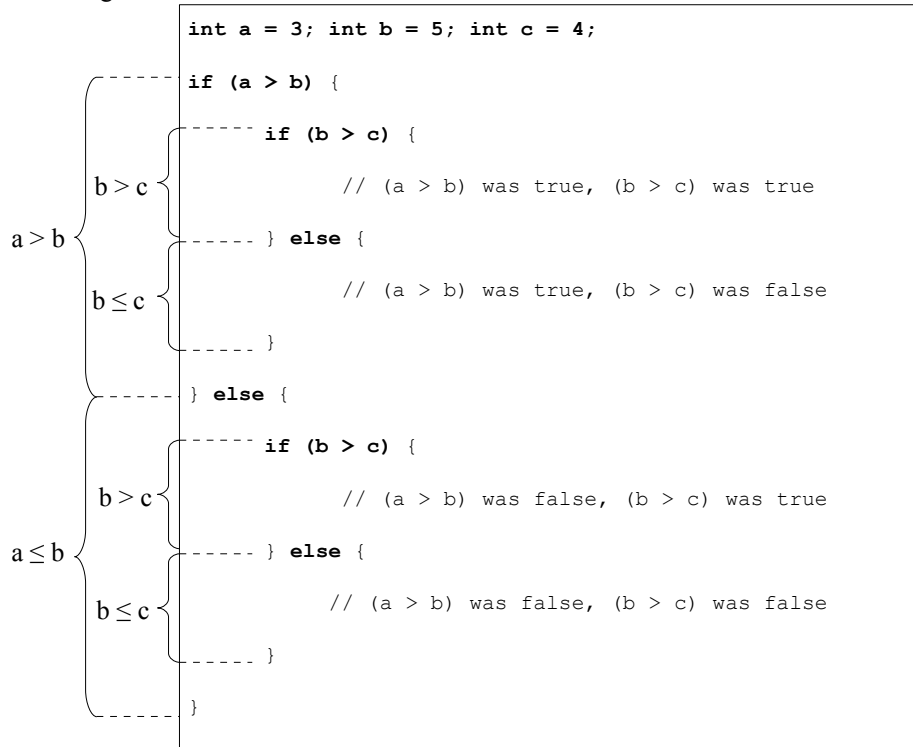
WRONG	Right

if (name == "phil") {
 	
}
	if (name.equals("phil")) {
	}

3). Comparing floats and doubles

if (my_double == 3.14) {
 	
}
	This is dangerous! doubles and floats are subject to rounding errors and so might not be exactly what you expect them to be!

Nested if-statements

You can place as many if-statements as you wish inside each other. Here is a diagram illustrating the idea.

```
int a = 3; int b = 5; int c = 4;

if (a > b) {

    if (b > c) {

            // (a > b) was true, (b > c) was true

    } else {

            // (a > b) was true, (b > c) was false

    }

} else {

    if (b > c) {

            // (a > b) was false, (b > c) was true

    } else {

            // (a > b) was false, (b > c) was false

    }

}
```

(diagram annotations: $a > b$, $b > c$, $b \leq c$, $a \leq b$, $b > c$, $b \leq c$)

Chained If-Statements

Another common pattern is a series of if statements. Notice that the last statement is an *else*. This is like the default value if none of the other if-statements are true.

```
int testscore = 76;

char grade;


if (testscore >= 90) {

    grade = 'A';

} else if (testscore >= 80) {

    grade = 'B';

} else if (testscore >= 70) {

    grade = 'C';

} else if (testscore >= 60) {

    grade = 'D';

} else {

    grade = 'F';

}

System.out.println("Grade = " + grade);
```

There is a short-hand way to do this using something called the **switch statement**. It's not a huge short-cut, however, and I won't detail it here.

How to Loop and Repeat in your program

While Loop

The while-loop tests whether *expression* is true. If it is, the loop will execute all the *statements* and loop back to the top to test *expression* again. It will continue to loop as long as *expression* is true.

If the *Expression* is false the first time, Java will skip all the *statements* inside the loop.
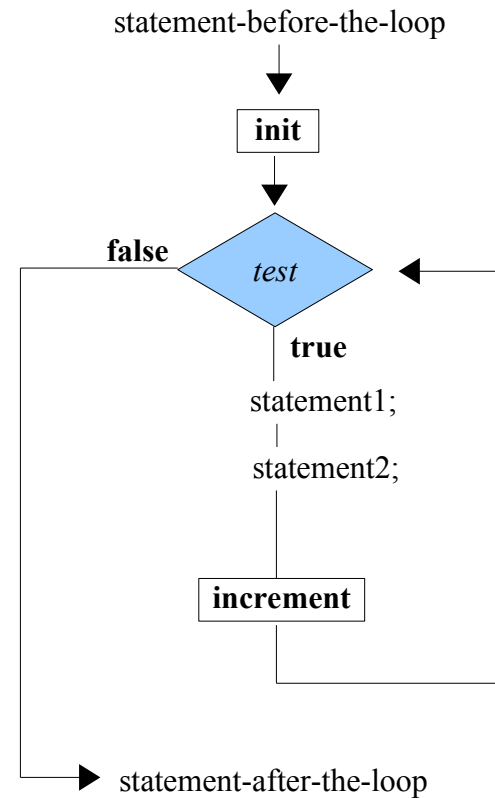
```
statement-before-the-loop;

while (expression)

{

    statement1;

    statement2;

    ...

}

statement-after-the-loop;
```
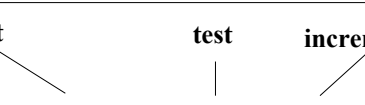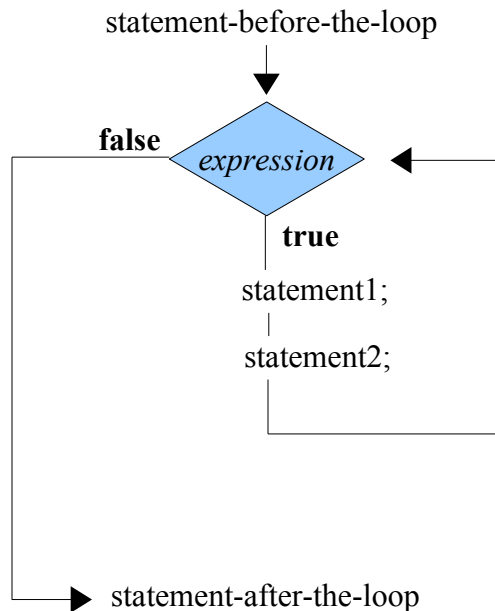
For-Loop

The for-loop initializes a variable first, then enters a loop in which it checks if *expression* is true, then executes *statements*, and finally executes **update** before returning to the top to check *expression* again.

For-loop	Same as the following while-loop

```for (init; test; increment)```
 ``` { ```
 ```   statement1; ```
 ```   statement2; ```
 ```   ... ```
 ``` } ```
	```init;```
	 ```while (test) {```
	 ``` statement1;```
	 ``` statement2;```
	 ``` ...```
	 ``` ...```
	 ``` increment;```
	 ```}```

statement-before-the-loop

init

false test

true

statement1;

statement2;

increment

statement-after-the-loop

statement-before-the-loop

false

expression

true

statement1;

statement2;

statement-after-the-loop

An example of identical for- and while- loops. Both of these loops iterate i from 0 to 9 and then exit.

For-loop	Same as the following while-loop

init   **test**   **increment**
 	
`for (int i = 0; i < 10; i++)`
`{`
 	
    `statement;`
 	
    `statement;`
 	
    `...`
 	
`}`
	`int i = 0;`
	`while (i < 10) {`
	    `statement;`
	    `statement;`
	    `...`
	    `...`
	    `i++;`
	`}`

When to use for-loops, when to use while-loops

For-loops: Usually you use a for-loop if you want to loop a specific number of times that you know in advance. For example, if you want to loop 10 times.

```
loop n times pattern:
for (int i = 0; i < n; i++) {

    System.out.println(i);   // statements to repeat

}
```

While-loops: Usually you use a while loop if you're not sure how many times it will need to loop. For example, if you want to keep asking the user to enter a number until the number is positive.

```
Repeat while something-is-true pattern:
while (myTurtle.clearInFront()) {

    myTurtle.moveForward();

}
```

```
while (true) {

    statements;

    if (test) {

        break;

    }

}
```

Infinite loop since *true* is always true

break will exit the while-loop immediately

do....while() loop

For and While loops both have a test at the beginning to see if they should run their code block even once.

If you want to run some code, then test to see if you should loop it again, you can use the do....while() loop as follows.

```
do {

    // statements to loop

} while (condition);
```

This is a good loop to use to enforce user input: This code will keep looping as long as the number the user enters is less than or equal to 0. (In other words, it will only continue on once the user enters a positive number).

```
String num; int n;

do {

    num = JOptionPane.showInputDialog("Enter a positive number");

    n = Integer.parseInt(num);

} while (n <= 0);
```

Methods

This complicated looking diagram is supposed to show what happens when you call a method. On line 7, the program calls the **distance** method. Each of the 4 values in the parentheses on line 7 get assigned to the 4 variables declared on line 11. The program then jumps to the first line of the **distance** method on line 12. When it gets to line 15, the **return** keyword will return whatever value follows it. Execution jumps *back* to line 7, which called **distance** in the first place. The method-call to the right of the = sign gets replaced by the number that was returned. So, at the end, d will get assigned whatever number **distance** returns.

```
1    public class ProgramName2 {

2        public static int point1_x, point1_y, point2_x, point2_y;

3

4        public static void main (String args[]) {

5            point1_x = 0;   point1_y = 0;

6

7            double d = distance(point1_x, point1_y, 10, 4);

8            System.out.println("Distance is " + d);

9        }

10

11       public static double distance(int x1, int y1, int x2, int y2) {

12           int dx = x1-x2;

13           int dy = y1-y2;

14

15           return Math.sqrt(dx*dx + dy*dy);

16       }

17   }
```

Declaring your own methods

Here is a typical method declaration with the different parts labeled:

```
public static double calcAnswer(double wingSpan, int numberOfEngines) {

        //do the calculation here


}
```

Argument list. These are the local variables which will hold the values that get passed to the method.

Method name

Data type for return value (or *null* for no return value)

Static modifier (or blank)

public or *private*

Common Mistake: "Missing Return Statement"

If your method has an error which reads "Missing Return Statement" you might have put your return statement inside an if-statement. Think: What happens if the if-statement is false? You never tell the method what it should return in this case.

```
public static boolean is_even(int n) {
    if (n % 2 == 0) {
        return true;
    }
}
```

To fix the problem, add another return statement at the end which will return what you want if the if-statement is false.
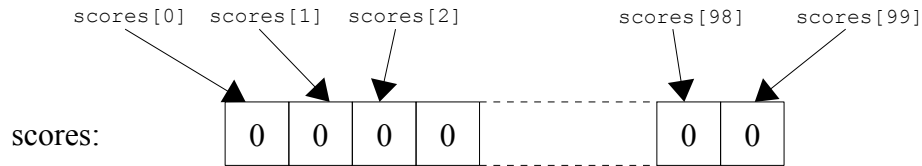
Arrays

An array is a list of variables that you can refer to by their place in the list.

Declaring an array

General pattern: `<data-type>[] name = new <data-type>[length];`
Example: `int[] scores = new int[100];`

This creates a list named *scores* of 100 int variables. The list is automatically initialized to all 0's (because the default value for an int is 0).

scores[0] scores[1] scores[2]   scores[98]   scores[99]

scores:

0	0	0	0		0	0

NOTE: The first array element is numbered 0. The second element is numbered 1, and so on. If your list is 100 ints long, the last int will be numbered 99.

Here are some examples of things you can do with arrays.

```
int[] raw_scores = new int[100];
boolean[] passed = new boolean[100];
String[] names = new String[100];
```
Declaring arrays

```
names[0] = "babar";
passed[1] = false;
names[100] = "pepe";       // error! invalid index.
raw_scores[99] = 23;
```
Assigning values

```
for (int i = 0; i < raw_scores.length; i++) {
  raw_scores[i] = (i*i+1)%7;
}
```
Loop through the array, assign a value to every element

```
// Test an array-element in an if-statment.

if (raw_scores[3] > 5) {
  System.out.println("Fourth score is: " + raw_scores[3]);
}
```

Length of an array: If the array is named my_array you can access how long it is with the variable: `my_array.length`

Note: `my_array.length` tells you the number of array elements, NOT the index of the last array element.

Another example: This method tests whether the array passed to it has the same value twice in a row.

```
public static boolean double_value(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        if (arr[i] == arr[i+1])
            return true;
    }
    return false;
}
```

Reading User Input

Here is an example of how you can ask the user for an integer:

```
1       String response;
2       int number;
3       response = JOptionPane.showInputDialog("Enter an integer");
4       number = Integer.parseInt(response);
```

Lines 1 and 2 declare the variables you'll need.

Line 3 displays the message "Enter an integer" and stores the user's response in a String variable.

Line 4 asks the Integer class to interpret the String as an int and save it into *number*.

You can use this same pattern for any primitive data type:

```
String response;

int i;
double d;
boolean b;

// INTEGER
response = JOptionPane.showInputDialog("Enter an integer");
i = Integer.parseInt(response);

// DOUBLE
response = JOptionPane.showInputDialog("Enter a real number");
d = Double.parseDouble(response);

// BOOLEAN
response = JOptionPane.showInputDialog("Enter 'true' or 'false'");
b = Boolean.parseBoolean(response);
```

Object Oriented Programming Concepts

Procedural vs. Object Oriented Programming

In a procedural programming style you break your task into sub-tasks which you can write methods for, but the organization of your program is still a step-by-step process.

In an object oriented style you break your task down into simple interactions between different kinds of objects. Objects are models for the things you want to interact. Objects have certain facts associated with them, and are able to perform certain actions. Often your software objects model real-world objects. You might have objects to represent customers and bank accounts, for example. Other times the objects represent more abstract things like numbers.

Here is a quick explanation of the different concepts involved in the Object Oriented style.

What is an Object?

Objects are key to understanding *object-oriented* technology. Consider some familiar real-world objects: your dog, your desk, your television set, your bicycle. Real-world objects share two characteristics: They all have *state* and *behavior*. *State* is the name for facts that describe the present state of your object. You can think of them as the variables that constitute your object. *Behavior* is the things your object can do.

For example, Dogs have state (name, color, breed, hungry (or not)) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes).
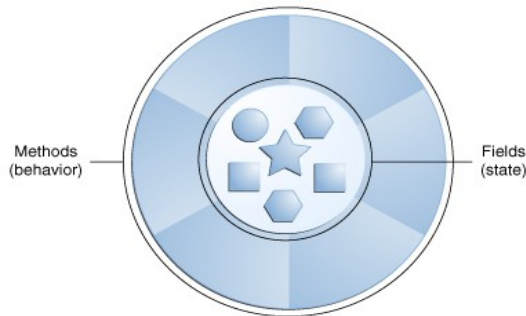

Software objects model real-world objects by modeling their state and behavior. An object stores its state in *fields* (which is just a new name for variables that describe and object's state) and exposes its behavior through *methods*. An objects methods usually do something with its internal state. They may let you change the state of an object, or ask about the state of an object so you can decide what you want to do with it.

Hiding internal state information and requiring all interaction to be performed through an object's methods is known as ***data encapsulation***—a fundamental principle of object-oriented programming.

Consider a bicycle, for example:
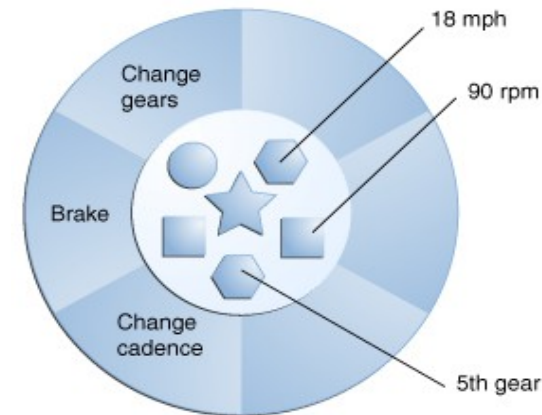
Data encapsulation (hiding an object's state and forcing all interactions to happen through methods) means the object remains in control of how the outside world is allowed to use it. For example, if the bicycle only has 6 gears, a method to change gears could reject any value that is less than 1 or greater than 6. Sometimes you can think of an object's methods as request handlers. When another piece of code calls a method on an object, it's making a request for the object to do something. Your objects method gets to decide how it will respond to that particular request.

There are four major benefits to organizing code in this way:
1. **Modularity**: The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.

2. **Information-hiding**: By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.

3. **Code re-use**: If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.

4. **Pluggability and debugging ease**: If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace *it*, not the entire machine.

What is a Class?

In the real world, you'll often find many individual objects all of the same kind. There may be thousands of other bicycles in existence, all of the same make and model. Each bicycle was built from the same set of blueprints and therefore contains the same components. In object-oriented terms, we say that your bicycle is an *instance* of the *class of objects* known as bicycles. A *class* is the blueprint from which individual objects are created.

The following Bicycle class is one possible implementation of a bicycle:

```
                        ─── Class name.  Must be Capitalized!

   class Bicycle {

        private int cadence = 0;          The fields that
        private int speed = 0;            describe the
        private int gear = 1;             object's state


        public void changeCadence(int newValue) {
            cadence = newValue;
        }

        public void changeGear(int newValue) {
            gear = newValue;
        }

        public void speedUp(int increment) {      The methods that
            speed = speed + increment;            describe the
        }                                         object's behavior

        public void applyBrakes(int decrement) {
            speed = speed - decrement;
        }

        public void printState() {
            System.out.println("cadence: " +
                cadence + " speed: " + speed +
                " gear: " + gear);
        }
   }
```

No Main Method! This class doesn't have a main method because it's not a complete application; it's just the blueprint for bicycles that might be *used* in an application. The responsibility of creating and using new Bicycle objects belongs to some other class in your application.

For a more detailed explanation of the parts of a class, see the next section on classes.

Here's a BicycleDemo class that creates two separate Bicycle objects and invokes their methods:

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

        // Create two different Bicycle objects
        Bicycle bike1 = new Bicycle();
        Bicycle bike2 = new Bicycle();

        // Invoke methods on those objects
        bike1.changeCadence(50);
        bike1.speedUp(10);
        bike1.changeGear(2);
        bike1.printStates();

        bike2.changeCadence(50);
        bike2.speedUp(10);
        bike2.changeGear(2);
        bike2.changeCadence(40);
        bike2.speedUp(10);
        bike2.changeGear(3);
        bike2.printStates();
    }
}
```

The output of this test prints the ending pedal cadence, speed, and gear for the two bicycles:

```
cadence:50 speed:10 gear:2
cadence:40 speed:20 gear:3
```

What Is Inheritance?

Different kinds of objects often have a certain amount in common with each other. Mountain bikes, road bikes, and tandem bikes, for example, all share the characteristics of bicycles (current speed, current pedal cadence, current gear). Yet each also defines additional features that make them different (tandem bicycles have two seats and two sets of handlebars; road bikes have drop handlebars, etc).

Object-oriented programming allows classes to *inherit* commonly used state and behavior from other classes. For example, you might create classes to define several different types of Bicycle, called MountainBike, RoadBike, and TandemBike. These all *extend* the Bicycle class, which means they *inherit* all the fields and methods of the Bicycle class. The Bicycle class is called their *superclass*, and they called *subclasses* of Bicycle. In Java, each class is allowed to have one direct superclass, and each superclass has the potential for an unlimited number of *subclasses*. (Other languages, such as C++ allow something called *multiple inheritance* in which one object can inherit

from more than one superclass. Java uses something called *interfaces* to achieve a similar result).

If you want to create a TandemBike class that inherits from the Bicycle class, simply add the words *extends Bicycle* to the declation of TandemBike. You don't have to add any more code for your TandemBike to have all the variables and methods that Bicycle objects have.
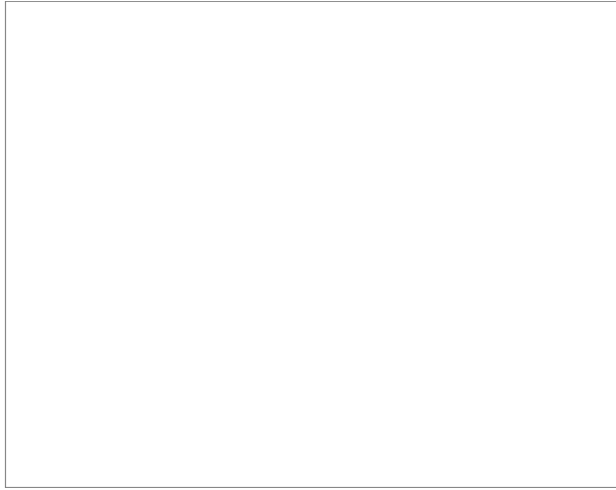
You may add additional variables and methods that will be unique to TandemBike objects however. For example:

```
class MountainBike extends Bicycle {
        private int numberOfRiders;

        public void changeNumberOfRiders(int n) {
                numberOfRiders = n;
        }
}
```

What is an Interface?

As you've already learned, objects define their interaction with the outside world through the methods that they expose to external code. Methods form the object's *interface* with the outside world; the buttons on the front of your television set, for example, are the interface between you and the electrical wiring on the other side of its plastic casing. You press the "power" button to turn the television on and off.

The purpose of an interface is to define a set of behaviors for a certain kind of thing. For example, some behaviors that define a bank might be *deposit*, *withdraw*, and

getBalance. Any objects that can perform these behaviors in the agreed-upon way can function as a bank. That object might be Bank of America, or your mom, or your cousin, or a box under your bed.

The whole point of an interface is that it's like a contract. If an object implements the behaviors defined by an interface, then you can treat that object as a certain type of thing. E.g. if my Box object implements the Bank interface, then I can treat it as a Bank.

Practically speaking, you declare an interface as a set of method declarations with no bodies.

```
interface Bank {

        public void deposit(double amount);

        public void withdraw(double amount);

        public double getBalance();
}
```

To create an object that implements this interface, add the words *implements Bank* to the class declaration, and then make sure you add all the methods the interface declares.

```
class MyBank implements Bank {
    // add fields you need

    // add the methods deposit, withdraw, and getBalance
}
```

Interfaces form a contract between the class and the outside world, and this contract is enforced at build time by the compiler. If your class claims to implement an interface, all methods defined by that interface must appear in its source code before the class will successfully compile.

What is a Package?

A package is a namespace that organizes a set of related classes and interfaces. Conceptually you can think of packages as being similar to different folders on your computer. Because software written in the Java programming language can be composed of hundreds or *thousands* of individual classes, it makes sense to keep things organized by placing related classes and interfaces into packages.

Declaring New Classes

Class name. Must be Capitalized!

```java
class Bicycle {

    private int speed;
    private int gear;


    public Bicycle(int s, int g) {
        this.speed = s;
        this.gear = g;
    }

    public Bicycle() {
        this(3, 5);
    }

    public Bicycle(int s) {
        this(s, 5);
    }

    public void changeGear(int newValue) {
        gear = newValue;
    }

    public void speedUp(int increment) {
        speed = speed + increment;
    }

    public void applyBrakes(int decrement) {
        speed = speed - decrement;
    }

    public int getSpeed() {
        return speed;
    }

    public int getGear() {
        return gear;
    }

    public void printState() {
        System.out.println("cadence: " +
          cadence + " speed: " + speed +
          " gear: " + gear);
    }
}
```

*The **fields** that describe the object's **state***

Constructors – these are special methods to initialize new objects when they're created

mutator methods that change field values

accessor methods that let external code request the object's field values

Note: "accessor" and "mutator" methods are sometimes informally called "getter" and "setter" methods because they are for getting the values of fields in the object, and setting the field values.

private, public, and protected: These modifies determine what other code is able to access. If a field or method is marked **private** then only the current class has access. **public** means any other classes have access, and **protected** means that only subclasses of the current class have access.

Constructor: The constructor is a special function that is called when you want to actually create an instance of the object. For example, to create a Bicycle object, you could say:

```java
Bicycle my_bike = new Bicycle(1, 2);
```

Data type

Variable name

Constructor

Arguments: 1 is assigned to s and 2 is assigned to g in the constructor. These are then assigned to the new object's fields.

Generally, constructors initialize the instance variables (fields) in an object.

NOTE: The constructor MUST have the same name as the class. It must also have no return type (i.e. the return type is left blank).

Multiple constructors: As with any method, you can **overload** the constructor by declaring more than one constructor with different arguments. This is generally done if you want to give the option of creating your object with different information. In my example I've given a constructor with no arguments that uses default values of 3 an 5 for the speed and gear. I've also given a constructor where the code creating the object can give only the speed, or give the speed and gear.

Notice that my other constructors have the code:

```java
this(3, 5);
```

This is how you can call one constructor from inside another. In this example, I am calling the Bicycle constructor with two integer arguments (the first one I declared).

Common "Mistakes"

These aren't mistakes that will cause your code not to work. These are things new programmers commonly do that experienced programmers have short-cuts for.

New programmer way	Experienced programmer way
i = i + 1;	i++;
n = n + 5;	n += 5;

if (my_test == true) {
    statement;
}
	if (my_test) {
	    statement;
	}

boolean check = false;
if (n > 0) {
    check = true;
}
	boolean check;
	
	check = (n > 0);