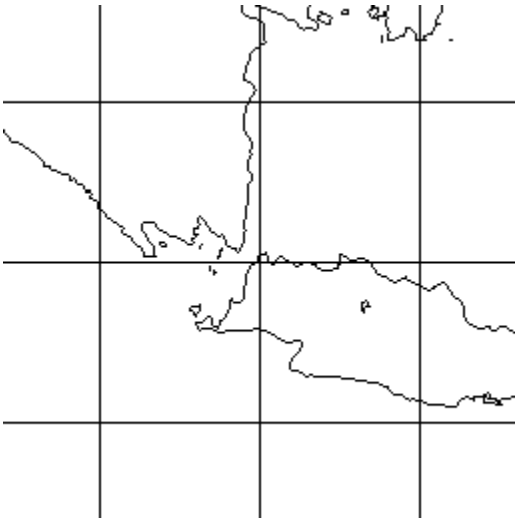
JAVA FOR ABSOLUTE BEGINNERS



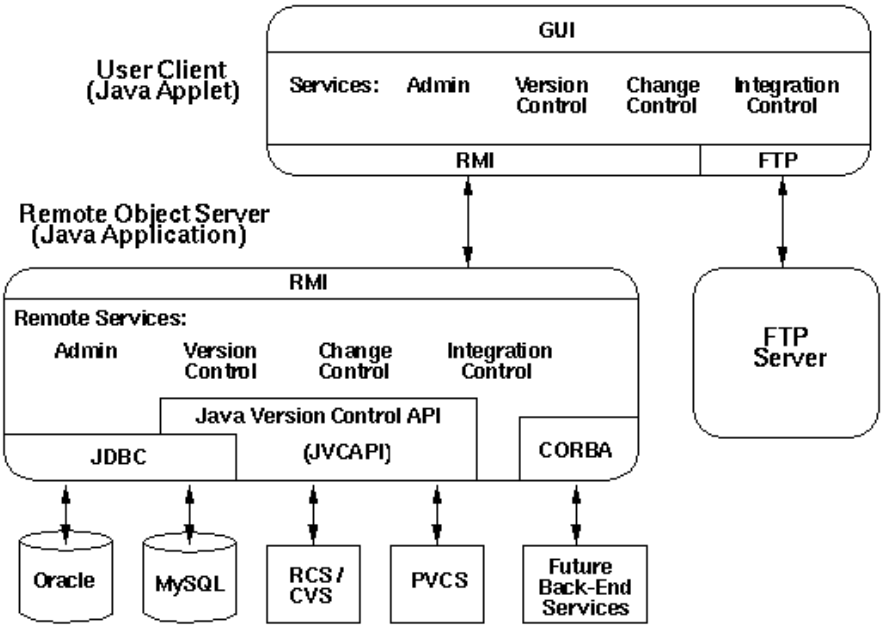


TABLE OF CONTENTS

PREFACE

Programming in Java is simple. The coherent architecture of it paves way to do wonders even for very beginners to the language.

Despite the simplicity of Java, the beginners always find it difficult to learn. Even very simple programs written in Java should follow the object oriented style of programming and understanding the basic concepts behind the scene always put them in trouble.

This handbook serves as a supplementary to the series of discussions “Java For Absolute Beginners”. The prime target is to give the beginners a thorough understanding of the basics of programming in Java and the basics of object oriented programming being the current programming paradigm and an absolute essential for programming in Java.

Although the real taste of Java goes with the ease of graphical user interface design and the power of client side web programming using applets, neither the discussions nor the handbook includes enough information related to them. As I said, this is the foundation to explore such technologies, may be of your own.

My sincere thanks goes to Dr K.M. Liyanage, the Director of the Computing Center, Faculty of Engineering, University of Peradeniya and his staff for their support in making the effort a success.

Kamal Sisira Kumara

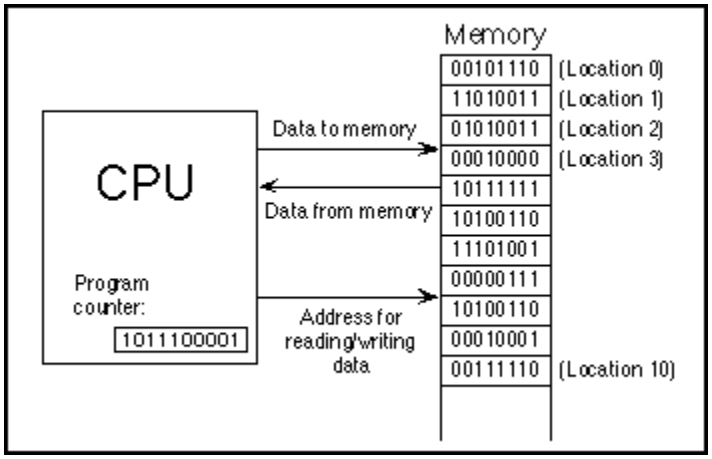
CHAPTER 1: BASICS OF COMPUTER PROGRAMMING

HOW DOES THAT HORRIBLE BOX RUN MY PROGRAM?

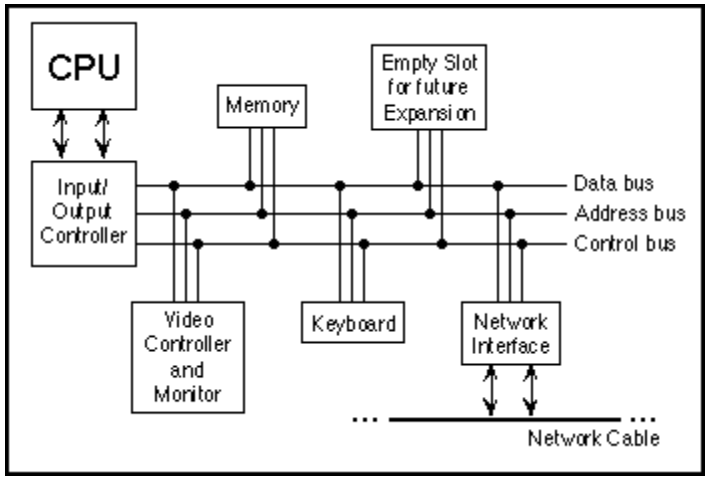
A **program** is simply a list of **instructions** to be followed by a computer. While running a program, these instructions are first loaded into the **main memory** (also known as **RAM** or **random access memory**) of the computer with **data** that is to be acted upon by them. The **loading** of the instructions and data to the primary memory may take place from various places (from a hard drive, key board, mouse, over the network, etc).

For the purpose of referencing these instructions and data, the **locations** of the primary memory are numbered in a sequential manner. Each number is known as an **address** since it points to a unique place in the main memory containing binary digits. Usually one location of the main memory holds an eight bit binary word.

The **central processing unit** (**CPU**) being the brain of a computer **fetches** these instructions from the main memory and **executes** them. The CPU contains **registers** each of them having the capability of holding a single machine language instruction (or a single piece of data). The **program counter** (**PC**) being such a register keeps track of the current position of the program. It holds the address of the next instruction to be fetched from the main memory to the CPU. Once a fetching is completed, the CPU executes it and another fetching is performed from the location of the main memory having the address given by the content of the program counter. This cyclic process is known as the **fetch-and-execute cycle** and takes place until the program ends.



While such a program is in execution, the **peripherals** of the computer may **interrupt** (a mouse click, key press, etc) the main program and transfer control to some other programs temporarily. For example a printer may request the CPU to work in favor of it for a while. In such a case, the CPU may execute what is in the **device driver** (software which says how to handle the interrupt generated by the particular device) and return to what it was doing when the interrupt occurred.



Further more, modern computers are expected to be **multitasking**. We prefer to browse the web while listening to music. At the same time, a file may be downloaded to the hard drive. This may involve running several programs simultaneously. On the other hand, one single program may also tackle several tasks at the same time. This is known as **multi-threading**.

Whatever happens, this sort of complexity arises the necessity of having a master program known as the **operating system** (**OS**) to share the resources of a computer system among the programs that request them. The operating system handles all the interrupts, communication with the user and the hardware and handles the issues of running several concurrent programs on top of it.

HOW CAN YOU AND ME WRITE PROGRAMS?

The CPU can only understand **machine language programs**. This implies that despite the way we use to write the programs, they should somehow be **translated** into machine language programs for the CPU to execute them.

Compilers allow us to do this sort of translation. First the program is written in an **abstract way** without thinking about the specific platform (for example the type of processor) on which it is supposed to run. Such written programs are said to be in **source code** format and written in a human understandable way. The compiler translates this into a machine language program and

the translated can then be run. Another way is to use an **interpreter** program where the input to it being the source code. An interpreter evaluates the source code at runtime and executes the **high level instructions** in it dealing with the CPU.

Example:

```
A high level language written statement in a program may
look like this:

                i = j + 1;

After the translation, it may look something like this:

                1100 0011
                1101 0010
                0010 1010
                1101 0010
                I am fed up, fill some more lines please.
```

Programs written in Java are both compiled and interpreted. More on this will be discussed later.

WORKING WITH HIGH LEVEL LANGUAGES

Computer programming involves manipulation of **data** using **instructions**.

In a high level programming language, data is handled using **variables**. A variable is just a name given to a memory location (or a set of adjacent memory locations considered as a single unit) for ease handling of data. Staying at the high level language platform, we use variables just as we use variables in day-to-day mathematics. Handling memory is taken care by the compiler and it is not our concern. On the other hand a variable may contain different types of data. The type of data may integers, floating point numbers (i.e. numbers with decimal places), single characters, strings (i.e. collection of characters forming one or more words), etc.

In a high level programming language, instructions include **control structures** and **subroutines**.

Control structures allow controlling the flow of the program. They include **decision-making**, **looping**, **branching** and Java in particular has an **exception handling** structure which gives provision for programmers a convenient way to tackle errors that may occur at runtime.

- Decision making structures allow a program to flow differently depending on whether a condition is satisfied or not.

```
if ( i == 1 ) {
     doThis();
} else {
     doThat();
}
```

- Looping involves doing some work repeatedly (until a governing condition becomes false).

```
while ( i < 10 ) {
     doThis();
}
```

- Branching allows immediate transfer of flow control from one place to another. For example a loop can be terminated with the execution of the statement "break;".

- Exception handling also performs a jump from the place where the error occurred to another place. At this jumped location, programmer can provide code for the **termination** or the **resumption** of the program.

Subroutines allow the main program to be broken into smaller manageable pieces. Consider drawing a house. Here, the main program is drawing the house. We can write a subroutine to draw a window of the house. In the main program we can **call** this subroutine each time we want to draw a window. In each case we may have to send some **parameters** to the subroutine telling the dimensions and the position of the window.

```
Here is the main program to draw the house:
main() {
     drawWindow(15,30,20,60);
     drawDoor(…,…,…,…);
     drawWall(…,…,…);
     drawWindow(…,…,…,…);
}

Here is the subroutine to draw the window:
drawWindow(…,…,…,…) {
     J = k + 1;
     drawThisLine(j, k);
     drawThatLine(…,…);
     ……;
     ……;
}
```

Repetitive calling of the subroutine from elsewhere of the program can be done and hence they minimize typing of the source code. On the other hand subroutines let the programmer to tackle one small piece of the whole program at one time thereby reducing the thinking effort needed to create the programs.

No horse gets anywhere till he is harnessed.

No steam or gas ever drives anything until it is confined.

No Niagara is ever turned into light and power until it is tunneled.

No life ever grows great until it is focused, dedicated and disciplined.

CHAPTER 2: OBJECT ORIENTED PROGRAMMING

EVOLUTION OF PROGRAMMING STYLES UPTO OBJECT ORIENTED PROGRAMMING

The first and second generations of programming languages being the machine and **assembly languages**, the first successful high level programming languages (third generation) were FORTRAN and BASIC. Their abstraction level was high enough for programmers to write general programs in a human understandable way without strictly thinking about how memory is managed and the machine language instructions supported by the processors. FORTRAN and BASIC had limited flow control and data structures. C and PASCAL formed the second layer of the third generation languages having more flow control and data structures.

The above stated third generation programming languages formed a style of programming called **structured programming**. Structured programming is heavily based on writing **functions** to solve a given problem. Writing functions involved the following scheme: To solve a large problem, break the problem into several pieces and work on each piece separately; to solve each piece, treat it as a new problem which can itself be broken down into smaller problems; eventually, you will work your way down to problems that can be solved directly, without further decomposition. This approach is called **top-down programming**.

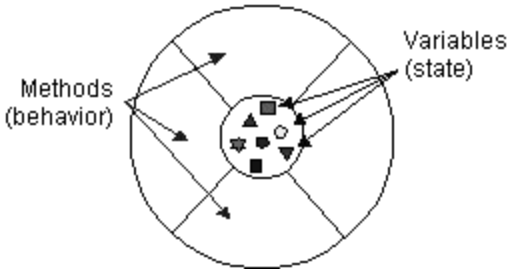
This approach tends to produce code unique to a given problem. Reusing the same code for another program usually involves heavy modifications. Writing instructions (or functions) to perform the task is the prime concern. The way of thinking is "**What has to be done, and what data is needed to get it done?**".

Reusing the code necessarily cuts down the cost of producing software and this lead the programmers to write components of a large program as **modular** as possible. A **module** is a component of a larger system that interacts with the rest of the system in a simple, well-defined, straightforward manner.

Now, using these modules at the bottom level, designing programs in the reverse direction (to the top) was possible up to some extent. This approach is called **bottom-up design**.

In practice top-down and bottom-up approaches are combined.

Further evolution of the idea of modularizing the components of a large system paved way to **object oriented programming**. In this approach, all the components are treated as objects having their own **internal states** (data stored) and **methods** to work on these data including mere setting and retrieval of them as well as methods to manipulate them. An object can be visualized as follows:



Now, a program written in an **OOP** language becomes a collection of objects. Objects react with each other by sending **messages**.

This exactly matches how the real world works. For example an "object man" has his own internal states (hair color, height, weight, volume, etc.) and methods to work with them (tell his hair color to someone, tell his density (#$!?) to someone by dividing the weight by volume and sending the result).

The level of abstraction was so high that programmers need not mould the problems into something understandable to the computer. Problems could be solved thinking about how real world works and direct mapping of the real world into the computer domain was possible.

Java is said to be a 3.3-generation language which came after the second generation Pascal and C high level programming languages and is pure object oriented. C++ also supports OOP, but provision for backward compatibility allows C++ programmers to incorporate non-OOP code with OOP code.

KEY CONCEPTS OF OBJECT ORIENTED PROGRAMMING

A language which supports mere creation of objects is not recognized as a sufficient programming tool. Code reusage, ease of maintenance, efficient ways of handling data and behavior of multiple objects, efficient usage of system resources, etc are also factors to be considered while defining a programming language. These factors formed the four **basic concepts of OOP** given below.

1. **Data Encapsulation**
2. **Data Abstraction**
3. **Inheritance**
4. **Polymorphism**

A programming language is said to be object oriented if it supports all these concepts. They defined the way of thinking in terms of objects. Good object oriented programming is necessarily followed by thorough understanding of these basic concepts and for the time being laymen definitions are given which you are not supposed to understand at the first place. They will be

discussed in depth after you get familiarized with writing small Java programs that do not necessarily require a rich knowledge of these concepts.

DATA ENCAPSULATION:

This is the process of localizing data definitions, hiding data within an object, and allowing access to it only through special functions known as member functions, methods or messages.

DATA ABSTRACTION:

This is the process of grouping related pieces of data together into logical units.

INHERITANCE:

This is the process of subclassing, children receiving data and methods from their parents.

POLYMORPHISM:

Different entities (objects) respond differently to the same message.

Wisdom comes not from extensive reading, but from inward inspection and reflection.

CHAPTER 3: INTRODUCTION TO JAVA TECHNOLOGY

Applications of Java Technology have already become very diverse. Some are given below:

Screen Phone
with PersonalJava™
and Web Browser

Mobile Phone
with PersonalJava
or EmbeddedJava™

Desktop Computer
with Java-enabled
Web Browser

Desktop Computer with Java
technology-based application

Network
Computer

Server with
Java technology-based applications
or servlets

Java technology is both a programming language and a platform

THE JAVA PROGRAMMING LANGUAGE

The Java programming language is a high-level language that can be characterized by all of the following buzzwords:

- Simple
- Architecture-neutral
- Object-oriented
- Portable
- Distributed
- High-performance
- Interpreted
- Multithreaded
- Robust
- Dynamic
- Secure

Advantages of the language are:

- **Get started quickly**: Although the Java programming language is a powerful object-oriented language, it's easy to learn, especially for programmers already familiar with C or C++.

- **Write less code**: Comparisons of program metrics (class counts, method counts, and so on) suggest that a program written in the Java programming language can be four times smaller than the same program in C++.

- **Write better code**: The Java programming language encourages good coding practices, and its garbage collection helps you avoid memory leaks. Its object orientation, its JavaBeans component architecture, and its wide-ranging, easily extendible API let you reuse other people's tested code and introduce fewer bugs.

- **Develop programs faster**: Your development time may be as much as twice as fast versus writing the same program in C++. Why? You write fewer lines of code and it is a simpler programming language than C++.

- **Avoid platform dependencies with 100% Pure Java**: You can keep your program portable by following the purity tips mentioned throughout this tutorial and avoiding the use of libraries written in other languages.

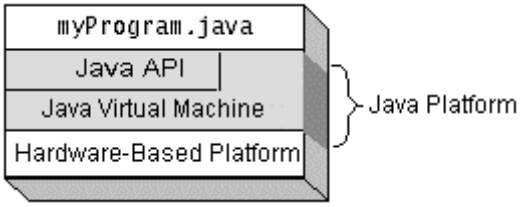
- **Write once, run anywhere**: Because 100% Pure Java programs are compiled into machine-independent bytecodes, they run consistently on any Java platform.

- **Distribute software more easily**: You can upgrade applets easily from a central server. Applets take advantage of the feature of allowing new classes to be loaded "on the fly," without recompiling the entire program.

THE JAVA PLATFORM

A **platform** is the hardware or software environment in which a program runs. The Java platform has two components:

- **The Java Virtual Machine (Java VM)**
- **The Java Application Programming Interface (Java API)**



CHAPTER 4: YOUR FIRST PROGRAM IN JAVA

Here is the conventional “Hello World!” program written in Java:

```
/**
 * This application simply displays “Hello World!”
 * to the standard output
 */

class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Follow these steps to create the program:

1. Write the source code in a file called "**HelloWorldApp.java**"

2. Compile the file using the java compiler:

   javac HelloWorldApp.java

   The compilation process produces a file called "HelloWorld.class" and this is the file that can be run in Java Virtual Machine (JVM).

3. Run the program invoking JVM:

   java HelloWorld

   This will display "Hello World!" at the standard output of the computer.

Important points of this exercise are:

- Provision to include extra text for your purposes is allowed using **comments**. **All text inside the comments are discarded by the compiler**.

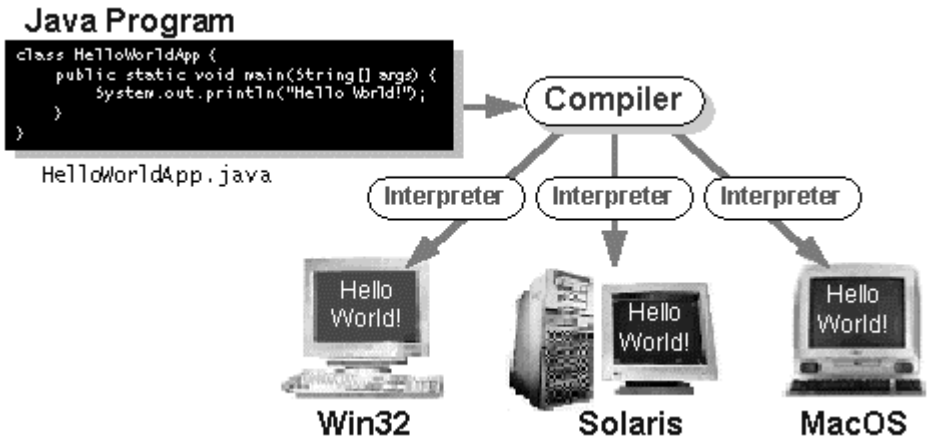
- **Java is a case sensitive language**. Upper case and lower case letters are distinct and writing "Class" for "class" will produce an error message at compile time.

- **The name of the output of the compiling process is the name of the class you specified inside your source code file**. It is not the name of the source code file. A program written in Java is a collection of ".class" files.

- **The main() method is the starting point of the execution of the program**. Inside the main method, you can write statements to be executed.

- System.out.println("Hello World!") is the statement which prints the string "Hello World!" to the standard output.

Here is a graphical view of what should happen:



The Java language supports three kinds of comments:

1. /* *text* */

   The compiler ignores everything from /* to */.

2. /** *documentation* */

   This indicates a documentation comment (*doc comment*, for short). The compiler ignores this kind of comment, just like it ignores comments that use /* and */. The JDK javadoc tool uses doc comments when preparing automatically generated documentation.

3. // *text*

   The compiler ignores everything from // to the end of the line.

CHAPTER 5: DATA TYPES IN JAVA

The data types of the Java programming language are divided into two categories: **primitive** and **reference** types.

PRIMITIVE DATA TYPES

A primitive type is predefined by the Java programming language and named by its reserved keyword.

Keyword	Description	Size/Format
(integers)		
byte	Byte-length integer	8-bit two's complement
short	Short integer	16-bit two's complement
int	Integer	32-bit two's complement
long	Long integer	64-bit two's complement
(real numbers)		
float	Single-precision floating point	32-bit IEEE 754
double	Double-precision floating point	64-bit IEEE 754
(other types)		
char	A single character	16-bit Unicode character
boolean	A boolean value (true or false)	true or false

REFERENCE TYPES

Distinct to primitive types, reference types contain **references to objects**. The pointed may be of type class, interface or array.

Arrays are of prime importance in any high level language despite whether they support OOP of not. An array is a collection of data items, all of the same type (primitive or reference), in which each item's position is uniquely designated by an integer. They are defined and used with the

square-bracket indexing operator [ ]. If you look at the very first program we wrote, you see an array of type "String" declared in the main method:

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

The args array contains the **arguments** that were given at the **command line**. Say you started a program called Copy by typing the following at the command line:

```
java Copy file1 file2
```

Within the main method, the names of the file can be referenced using args. args[0] contains the string (a string is a collection of characters) "file1" and args[1] contains "file2".

The following lines describe some syntax used to declare and/or initialize arrays.

1. Just declare an array of integers called a1:

   ```
   int[] a1;
   ```

2. Declare an array of integers called a1 and set the size of it to be 20:

   ```
   int[] a1 = new int[20];
   ```

3. First declare and then initiate:

   ```
   int[] a1 = new int[3];
   a1[0] = 1;
   a1[1] = 2;
   a1[2] = -1;
   ```

4. Declare and initiate an array at the same time:

   ```
   int[] a1 = {1,2,3,4,5};

   or

   int[] a1 = {1,2,3,4,5,};
   ```

Tenderness is the chief gift of great men.

CHAPTER 6: OPERATORS IN JAVA

Operators are the most basic works, acts or simply the primitive verbs defined in the language.

ARITHMATIC OPERATORS

Operator	Use	Description
+	op1 + op2	Adds op1 and op2
-	op1 - op2	Subtracts op2 from op1
*	op1 * op2	Multiplies op1 by op2
/	op1 / op2	Divides op1 by op2
%	op1 % op2	Computes the remainder of dividing op1 by op2

Operator	Use	Description
+	+op	Promotes op to int if it's a byte, short, or char
-	-op	Arithmetically negates op

Operator	Use	Description
++	op++	Increments op by 1; evaluates to the value of op before it was incremented
++	++op	Increments op by 1; evaluates to the value of op after it was incremented
--	op--	Decrements op by 1; evaluates to the value of op before it was decremented
--	--op	Decrements op by 1; evaluates to the value of op after it was decremented

RELATIONAL AND CONDITIONAL OPERATORS

Operator	Use	Returns true if
>	op1 > op2	op1 is greater than op2
>=	op1 >= op2	op1 is greater than or equal to op2
<	op1 < op2	op1 is less than op2
<=	op1 <= op2	op1 is less than or equal to op2
==	op1 == op2	op1 and op2 are equal
!=	op1 != op2	op1 and op2 are not equal

Operator	Use	Returns true if
&&	op1 && op2	op1 and op2 are both true, conditionally evaluates op2
||	op1 || op2	Either op1 or op2 is true, conditionally evaluates op2
!	! op	op is false
&	op1 & op2	op1 and op2 are both true, always evaluates op1 and op2
|	op1 | op2	Either op1 or op2 is true, always evaluates op1 and op2
^	op1 ^ op2	if op1 and op2 are different--that is if one or the other of the operands is true but not both

SHIFT AND LOGICAL OPERATORS

Operator	Use	Operation
>>	op1 >> op2	shift bits of op1 right by distance op2
<<	op1 << op2	shift bits of op1 left by distance op2
>>>	op1 >>> op2	shift bits of op1 right by distance op2 (unsigned)

Operator	Use	Operation
&	op1 & op2	bitwise and
|	op1 | op2	bitwise or
^	op1 ^ op2	bitwise xor
~	~op2	bitwise complement

There are two ways of spreading light: to be the candle or the mirror that reflects it.

ASSIGNMENT OPERATORS

Operator	Use	Equivalent to
+=	op1 += op2	op1 = op1 + op2
-=	op1 -= op2	op1 = op1 - op2
*=	op1 *= op2	op1 = op1 * op2
/=	op1 /= op2	op1 = op1 / op2
%=	op1 %= op2	op1 = op1 % op2
&=	op1 &= op2	op1 = op1 & op2
|=	op1 |= op2	op1 = op1 | op2
^=	op1 ^= op2	op1 = op1 ^ op2
<<=	op1 <<= op2	op1 = op1 << op2
>>=	op1 >>= op2	op1 = op1 >> op2
>>>=	op1 >>>= op2	op1 = op1 >>> op2

OTHER OPERATORS

Operator	Description
?:	Shortcut if-else statement
[]	Used to declare arrays, create arrays, and access array elements
.	Used to form qualified names
(*params*)	Delimits a comma-separated list of parameters
(*type*)	Casts (converts) a value to the specified type
new	Creates a new object or a new array
instanceof	Determines whether its first operand is an instance of its second operand

CHAPTER 7: CONTROLLING THE FLOW OF YOUR PROGRAM

The following table lists the flow controlling mechanisms present in Java:

Statement Type	Keyword
looping	`while, do-while , for`
decision making	`if-else, switch-case`
exception handling	`try-catch-finally, throw`
branching	`break, continue, label:, return`

LOOPS

Use the **while** statement to loop over a block of statements while a boolean expression remains true. The expression is evaluated at the top of the loop.

```
while (boolean expression) {
     statement(s)
}
```

Use the **do-while** statement to loop over a block of statements while a boolean expression remains true. The expression is evaluated at the bottom of the loop, so the statements within the do-while block execute at least once:

```
do {
     statement(s)
} while (expression);
```

The **for** statement loops over a block of statements and includes an initialization expression, a termination condition expression, and an increment expression.

```
for (initialization ; termination ; increment) {
     statement(s)
}
```

Leisure is the time for doing something useful.

DECISION-MAKING STATEMENTS

The Java programming language has two decision-making statements: **if-else** and **switch**. The more general-purpose statement is if; use switch to make multiple-choice decisions based on a single integer value. The following is the most basic if statement whose single statement block is executed if the boolean expression is true:

```
if (boolean expression) {
    statement(s)
}
```

Here's an if statement with a companion else statement. The if statement executes the first block if the boolean expression is true; otherwise, it executes the second block:

```
if (boolean expression) {
    statement(s)
} else {
    statement(s)
}
```

You can use else if to construct compound if statements:

```
if (boolean expression) {
    statement(s)
} else if (boolean expression) {
    statement(s)
} else if (boolean expression) {
    statement(s)
} else {
    statement(s)
}
```

The switch statement evaluates an integer expression and executes the appropriate case statement.

```
switch (integer expression) {
    case integer expression:
         statement(s)
         break;
    ...
    default:
         statement(s)
         break;
}
```

EXCEPTION-HANDLING STATEMENTS

Use the try, catch, and finally statements to handle exceptions.

```
try {
    statement(s)
} catch (exceptiontype name) {
    statement(s)
} catch (exceptiontype name) {
    statement(s)
} finally {
    statement(s)
}
```

BRANCHING STATEMENTS

Some branching statements change the flow of control in a program to a labeled statement. You label a statement by placing a legal identifier (the label) followed by a colon (:) before the statement:

```
statementName: someJavaStatement;
```

Use the unlabeled form of the break statement to terminate the innermost switch, for, while, or do-while statement.

```
break;
```

Use the labeled form of the break statement to terminate an outer switch, for, while, or do-while statement with the given label:

```
break label;
```

A continue statement terminates the current iteration of the innermost loop and evaluates the boolean expression that controls the loop.

```
continue;
```

The labeled form of the continue statement terminates the current iteration of the loop with the given label:

```
continue label;
```

Use return to terminate the current method.

```
return;
```

You can return a value to the method's caller, by using the form of return that takes a value.

```
return value;
```

CHAPTER 8: BREAKING DOWN THE PROGRAM INTO MANAGEABLE PIECES

WRITING SIMPLE MATHODS

Methods allow you to break down a big program into smaller pieces. The idea was introduced in the first chapter where drawing a house was broken down into smaller methods of drawing windows, doors, etc. A method may return or may not return some data upon the completion of it. For example the method System.out.println("Hello World!") we used in our first program is a method which prints the string "Hello World!" to the stdout and returns nothing.

Below is the syntax used to define a method which takes two arguments (you may use more):

```
return-type method-name(arg1-type arg1, arg2-type arg2) {

      statements;

}
```

Here is a complete program which use two simple programmer defined methods to calculate the factorial of a number (return type is an integer) and print it in a formatted manner (returns nothing or "**void**").

```
/* Factorial is a stand alone program which accepts a number as the first
*  argument at the command line and prints the factorial of it to the
*  stdout.
*/

public class Factorial {
/* This is the main method
*/
    public static void main(String[] args) {
      int i = 0;  // i is the input number
      int j = 0;  // j is to store the result or the factorial of i

/* If an argument is not given at the command line, accessing args[0] is
*  invalid and this may throw an exception of type
*  ArrayIndexOutOfBoundsException.
*
*  If the first argument is not a number, Integer.parseInt(String) method
*  throws a NumberFormatException.
*/
      try {
          i = Integer.parseInt(args[0]); // i is assigned the number
      } catch (ArrayIndexOutOfBoundsException aioobe) {
          // Send an error message to stderr and exit the program
          System.err.println("Please enter an argument to the program");
          System.exit(1);
      } catch (NumberFormatException nfe) {
```

```
            // Send an error message to stderr and exit the program
            System.err.println("First argument to the program is not a " +
                               "number");
            System.exit(1);
        }

/* j is assigned the factorial of i returned by the method factorial(int).
*  If i is a negative number, factorial(int) method is written to throw an
*  exception. This is caught in this place and the program is stopped.
*/
        try {
            j = factorial(i);   // calling the method factorial(int)
        } catch (Exception e) {
            // Send an error message to stderr and exit the program
            System.err.println("Factorial of a negative number is not " +
                               "defined");
            System.exit(2);
        }

/* Call the method formatPrint(int, int) to print the output
*/
        formatPrint(i, j);
      } // main() ends here

/* Below is the definition of the factorial(int) method. It throws an
*  exception (namely the exception "Exception") if the input to the method is
*  a negative number.
*
*  "throws Exception" says that the method has the ability to throw an
*   Exception
*/
      static int factorial(int number) throws Exception {
        // If the number is negative throw Excption
        if (number < 0) {
            throw new Exception();
        } else if (number == 0) {  // If the number is 0, return 1
            return 1;
        } else {    // number if > 0 and return n*((n-1)!)
            return number*factorial(number -1);
        }
      } // factorial() ends here

/* Below is the method to print the output of the program. This is rather
*  a simple method and you could have print this without writing a special
*  method.
*
*  The method returns nothing and hence void is used as the return type
*/
      static void formatPrint(int number, int result) {
        System.out.println("You gave the number : " + number);
        System.out.println("I calculated the factorial of it : " + result);
        System.out.println("Happy now ?");
      } // formatPrint() ends here
} // class Factorial ends here
```

Here are some "Things-to-Think":

1. Where do you find the definition of Integer.parseInt(String) method in your code? (Tip: Did you use System.out.println(String) method earlier?)
2. What are the diffences between System.out.println(String) and System.err.println(String)?
3. Why two ways of exit: System.exit(1) and System.exit(2)? How about System.exit(0)?
4. What if I don't put the keyword "static" in the definition of the methods? Compile and see this.
5. What is the largest number you can give as an argument to the program? Can you explain the reason for the output of the program if you input 100?
6. Say you are watching your award winning obsolete film "The Square" for the 999 time simply for the sake of your burning desire to watch the favorite scene in it. Your favorite actress (with whom you opened a hardware shop [Ana Kadaya] later) is a news reader and she reads news:

   "The handsome prince (of cause the author of this handbook being me the great Kamal Sisira Kumara) is reported to have kissed her again…."

   You like the scene for two reasons. To be specific I am listing them below:

   1. She was, oops, still is your sweet little thing.
   2. The way you arranged the scene:

      She is reading news to the left side of the screen and you project what you get from the camera just in front of her to the right side of the screen.

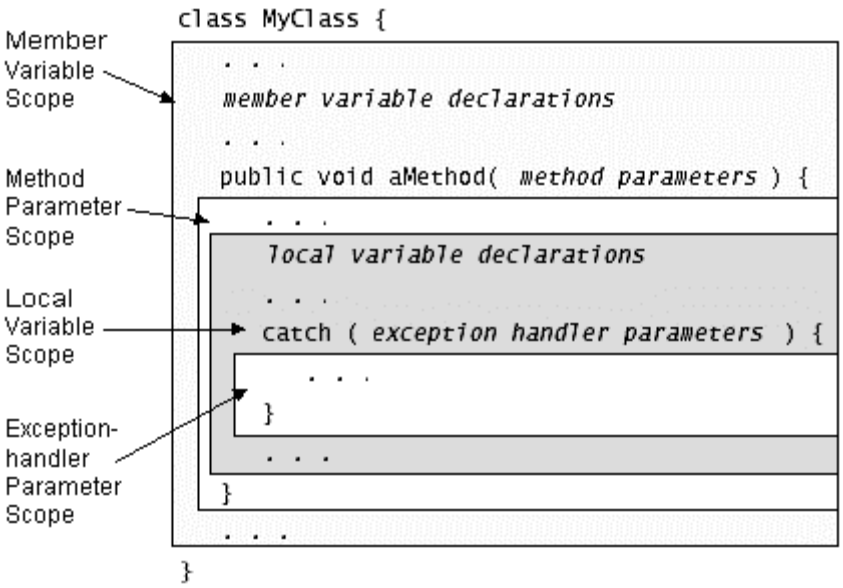
      Here is the big deal: On that projected portion she reads news and again to the right side another projected image is seen. In that projected screen she is shown for the third time reading news and to the right side of her……

      Go inside one step further, you have another step to go and see her. You feel like reaching the infinity. No end……

Will you accept if I say n! is n*((n-1)!)? May I humbly call this **recursion**?

SCOPE OF YOUR VARIABLES

Declaration of a variable somewhere in the source file does not imply that it can be referenced from anywhere in that file. A variable has a **scope** which is the region of a program from where the variable can be referred to by its simple name. The scope of a variable is determined by the location where it is declared. Depending on this location, variables can be classified into four groups and their scopes are shown in the following figure.



CHAPTER 9: OBJECT ORIENTED PROGRAMMING- PART 2

The programs written so far do not use the real power of OOP. The following pages discuss how to use real objects in your programs.

CLASSES AND OBJECTS

The key to OOP is the concept of an **object** containing both data and methods that manipulate the data.

A **class** defines the implementation of a particular kind of object. It describes the attributes (data) and attitudes or behavior (methods) of objects.

JVM can read the classes and create objects for you. Or else I can think my classes become objects when they manifest in the run-time environment.

If that is the case, here is a simple question:

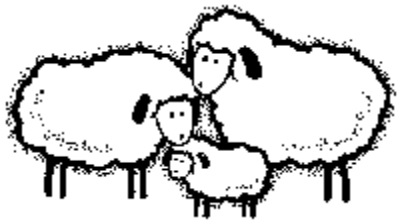
I want three objects similar to each other, but containing different data values. Types of data are the same. How many classes do I have to use?

On the other hand, what is a class anyway?

WRITING A SIMPLE CLASS

Keep in mind the key concept: **Data Abstraction** (i.e. do not write a class called "Male" and write a method to check the pregnancy).

Here is a simple class called "Sheep":



Kindness consists in loving people more than they deserve.

```
class Sheep {
    // The following 3 lines describe the attributes of the class
    String name = null;
    String color = null;
    int age;
    float weight;
    float volume;

    // The following methods describe the functionality of the class

    // An access method to set the name of the sheep
    void setName(String nameOfSheep) {
      name = nameOfSheep;
    }

    // An access method to set the color of the sheep
    void setColor(String color) {
      this.color = color;
    }


    // Following three are also access methods to set the data members
    // of the class
    void setAge(int age) {
      this.age = age;
    }

    void setWeigth(float weigth) {
      this.weight = weigth;
    }

    void setVolume(float volume) {
      this.volume = volume;
    }

    // Here is a method with no return type that manipulate data
    // This method describes how a sheep talks.
    void talk() {
      System.out.println("I am " + name + ".");
      System.out.println("My skin is " + color + ".");
      System.out.println("I am " + age + " years old");
      System.out.println("How about you?");
    }

    // Here is another method that has a return type
    float getDensith() {
      return weight/volume;
    }
} // class Sheep ends here
```

CREATE OBJECTS USING YOUR CLASS

In your source code, you can use the operator "**new**" to state that you want to create (**instantiate**) an object. After the creation, you can sent and receive messages.

Here is a program which use our "Sheep" class:

```
public class SheepTest {
    public static void main(String[] args) {
     Sheep sheep1 = new Sheep(); // This line says to create a
                                 // Sheep object
     sheep1.setName("Baba");    // Call the method setName
                                // and set the name of the sheep
     sheep1.setColor("black");
     sheep1.setAge(2);
     sheep1.weigth = 30;  // How about direct accessing of data
                          // members without using methods
     sheep1.volume = 0.5;
     sheep1.talk(); // Ask the sheep to talk
     System.out.println("This mad sheep's density: " +
                                        sheep1.getDensity());
    } // main ends here
} // class SheepTest ends here
```

NEW OBJECTS AND SET DATA AT THE SAME TIME: USING CONSTRUCTORS

Want to create a sheep having a name, color and age in one shot? Define the class like this:

```
class Sheep {
    String name, color;
    int age;
    float weight, volume;

    // Here is the constructor, a special method which create
    // objects
    Sheep(String name, String color, int age) {
     this.name = name;
     this.color = color;
     this.age = age;
    }

    // define other methods……
}
```

Here are the contents of the modified main method of the SheepTest class:

```
Sheep sheep2 = new Sheep("Anabella", "white", 1);
sheep2.talk();
sheep2.weigth = 31;
shepp2.volume = 0.5;
System.out.println("This mad sheep's density: " +
                                    sheep2.getDensity());
```

- The **constructor** is a special method that is used with the keyword "new" to instantiate an object. The name of the constructor should match the class name.

OVERLOADING THE CONSTRUCTOR

For the same class, several constructors with different **signatures** can be used. For example you can write a second constructor to create a sheep in our earlier example which sets all the data members in one shot.

Depending on the requirements, you can use a preferred constructor while instantiating an object.

An example:

> Say we know everything about Baba and we do not know weight and volume of Anabella. We also have written two constructors in our Sheep class one which accepts only the name, color and age and another one which accepts all the previous and also weight and volume.

Call the constructors as follows to create Baba and Anabella:

```
new Sheep("Baba", "black", 2, 30, 0.5);
new Sheep("Anabella", "white", 1);
```

On top of my head, I remember at least two instances where the operator "new" is not used to create some special objects. Writers of Java programming language have made this possible due to the very frequent usage of these special objects. Can you figure out what those objects are and syntax used to make them?

ENCAPSULATION AND ACCESS SPECIFIERS

If you are very much concerned with data encapsulation (or hiding data), you may declare all the data members within your class private and allow manipulation of them using methods.

On the other hand, you may specify who (objects of particular classes) can access the data and call the methods defined in your class using the **access specifiers** being some keywords in Java.

The following lists the keywords used to control the access to data and methods:

Specifier	class	subclass	package	world
Private	X			
protected	X	X*	X	
Public	X	X	X	X
package	X		X	

MEANING OF "static"

If a method of a data member of a class is declared static, JVM allocates memory only for a single copy of that method or data member in favor of all the objects of that class. All objects of the class share this single copy as it belongs to the object which access it at a given time. This implies that changing the value of a **static data member** by one object will be reflected to all the objects.

System.out.println(String) is a **static method** which prints a string to stdout. If it were not, we could have created an object called System and proceed.

USING PREWRITTEN LIBRARIES

AS COMPUTERS AND THEIR USER INTERFACES have become easier to use, they have also become more complex for programmers to deal with. You can write programs for a simple console-style user interface using just a few subroutines that write output to the console and read the user's typed replies. A modern graphical user interface, with windows, buttons, scroll bars, menus, text-input boxes, and so on, might make things easier for the user. But it forces the programmer to cope with a hugely expanded array of possibilities. The programmer sees this increased complexity in the form of great numbers of subroutines that are provided for managing the user interface, as well as for other purposes.

Prewritten libraries overcome this problem. A programmer can at any time use these libraries in his applications. The Java programming language is supplemented by such a large library known as the standard **API** (**Application Programming Interface**). You've seen part of this API already, in the form of the String data type and its associated routines, and the System.out.print() routines. The standard Java API includes routines for working with graphical user interfaces, for

network communication, for reading and writing files, and more. It's tempting to think of these routines as being built into the Java language, but they are technically subroutines that have been written and made available for use in Java programs.

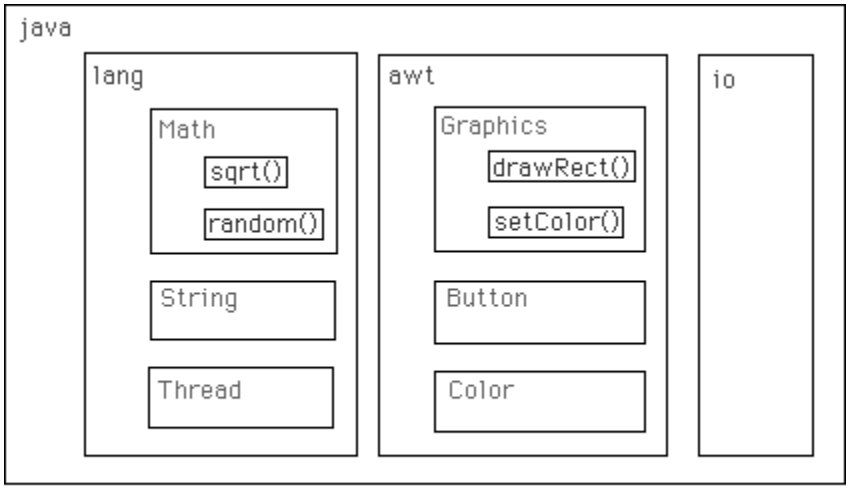
Java is platform-independent. That is, the same program can run on platforms as diverse as Macintosh, Windows, UNIX, and others. The same Java API must work on all these platforms. But notice that it is the **interface** that is platform-independent; the **implementation** varies from one platform to another. A Java system on a particular computer includes implementations of all the standard API routines. A Java program includes only **calls** to those routines. When the Java interpreter executes a program and encounters a call to one of the standard routines, it will pull up and execute the implementation of that routine which is appropriate for the particular platform on which it is running. This is a very powerful idea. It means that you only need to learn one API to program for a wide variety of platforms.

Like all subroutines in Java, the routines in the standard API are grouped into classes. To provide larger-scale organization, classes in Java can be grouped into **packages**. You can have even higher levels of grouping, since packages can also contain other packages. In fact, the entire standard Java API is implemented as one large package, which is named "java". The java package, in turn, is made up of several other packages, and each of those packages contains a number of classes.

One of the sub-packages of java, for example, is called "awt". Since awt is contained within java, its full name is actually java.awt. This is the package that contains classes related to graphical user interfaces, such as the Button class which represents push-buttons on the screen, and the Graphics class which provides routines for drawing on the screen. Since these classes are contained in the package java.awt, their full names are actually java.awt.Button and java.awt.Graphics.

The java package includes several other sub-packages, such as java.io, which provides facilities for input/output, java.net, which deals with network communication, and java.applet, which implements the basic functionality of applets. The most basic package is called java.lang, which includes fundamental classes such as String and Math.

It might be helpful to look at a graphical representation of the levels of nesting in the java package, its sub-packages, the classes in those sub-packages, and the subroutines in those classes. This is not a complete picture, since it shows only a few of the many items in each element:



Subroutines nested in classes nested in two layers of packages.
The full name of sqrt() is java.lang.Math.sqrt()

Let's say that you want to use the class java.awt.Button in a program that you are writing. One way to do this is to use the full name of the class. For example, you could say

```
java.awt.Button  stopBttn;
```

to declare a variable named stopBttn whose type is java.awt.Button. Of course, this can get tiresome, so Java makes it possible to avoid using the full names of classes. If you put

```
import java.awt.Button;
```

at the beginning of your program, before you start writing any class, then you can abbreviate the full name java.awt.Button to just the name of the class, Button. This would allow you to say just

```
Button  stopBttn;
```

to declare the variable stopBttn. (The only effect of the import statement is to allow you to use simple class names instead of full "package.class" names; you aren't really importing anything substantial. If you leave out the import statement, you can still access the class -- you just have to use its full name.) There is a shortcut for importing all the classes from a given package. You can import all the classes from java.awt by saying

```
import java.awt.*;
```

In fact, any Java program that uses a graphical user interface is likely to begin with this line. A program might also include lines such as "import java.net.*;" or "import java.io.*;" to get easy access to networking and input/output classes.

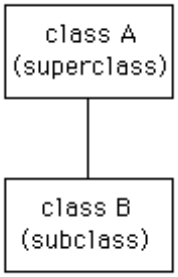
Because the package java.lang is so fundamental, all the classes in java.lang are **automatically** imported into every program. It's as if every program began with the statement "import java.lang.*;".

Programmers can create new packages. Suppose that you want some classes that you are writing to be in a package named utilities. Then the source code file that defines those classes must begin with the line "package utilities;". Any program that uses the classes should include the directive "import utilities.*;" to obtain access to all the classes in the utilities package.

In projects that define large numbers of classes, it makes sense to organize those classes into one or more packages. It also makes sense for programmers to create new packages as toolboxes that provide functionality and API's for dealing with areas not covered in the standard Java API.

Although we won't be creating packages explicitly in our programs, **every** class is actually part of a package. If a class is not specifically placed in a package, then it is put in something called the **default package**, which has no name.

LET ME TEACH YOU HOW TO PRODUCE CHILDREN: INHERITANCE EXPLAINED HERE

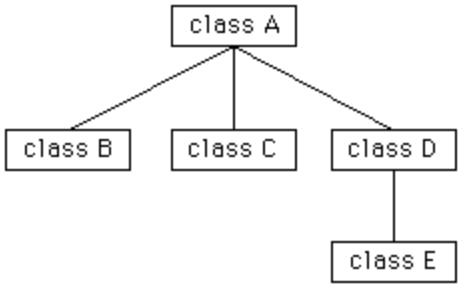


The term **inheritance** refers to the fact that one class can inherit part or all of its structure and behavior from another class. The class that does the inheriting is said to be a **subclass** of the class from which it inherits. If class B is a subclass of class A, we also say that class A is a **superclass** of class B. (Sometimes the terms **derived class** and **base class** are used instead of subclass and superclass.)

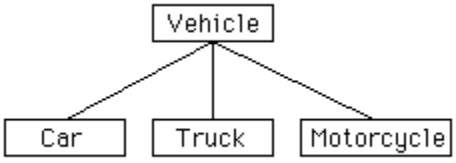
In Java, when you create a new class, you can declare that it is a subclass of an existing class. If you are defining a class named "B" and you want it to be a subclass of a class named "A", you would write

```
class B extends A {
    .
    .  // additions to, and modifications of,
    .  // stuff inherited from class A
    .
}
```

Several classes can be declared as subclasses of the same superclass. This is shown in the diagram, where class E is a subclass of class D which is itself a subclass of class A. In this case, class E is considered to be a subclass of class A, even though it is not a direct subclass.



Suppose that a program has to deal with motor vehicles, including cars, trucks, and motorcycles.



The program could use a class named Vehicle to represent all types of vehicles. The Vehicle class could include instance variables such as registrationNumber and owner and instance methods such as transferOwnership(). These are variables and methods common to all vehicles. Three subclasses of Vehicle -- Car, Truck, and Motorcycle -- could then be used to hold variables and methods specific to particular types of vehicles. The Car class might add an instance variable numberOfDoors, the Truck class might have numberOfAxels, and the Motorcycle class could have a boolean variable hasSidecar. The declarations of these classes in Java program would look, in outline, like this:

```
class Vehicle {
   int registrationNumber;
   Person owner;// (assuming that a Person class has been
                // defined)
   void transferOwnership(Person newOwner) {
       . . .
```

```
    }
    . . .
}
class Car extends Vehicle {
    int numberOfDoors;
    . . .
}
class Truck extends Vehicle {
    int numberOfAxels;
    . . .
}
class Motorcycle extends Vehicle {
    boolean hasSidecar;
    . . .
}
```

Suppose that myCar is a variable of type Car that has been declared and initialized with the statement

```
Car myCar = new Car();
```

(Note that, as with any variable, it is OK to declare a variable and initialize it in a single statement. This is equivalent to the declaration "Car myCar;" followed by the assignment statement "myCar = new Car();".) Given this declaration, a program could to refer to myCar.numberOfDoors, since numberOfDoors is an instance variable in the class Car. But since class Car extends class Vehicle, a car also has all the structure and behavior of a vehicle. This means that myCar.registrationNumber, myCar.owner, and myCar.transferOwnership() also exist.

Now, in the real world, cars, trucks, and motorcycles are in fact vehicles. The same is true in a program. That is, an object of type Car or Truck or Motorcycle is automatically an object of type Vehicle. This brings us to the following Important Fact:

A variable that can hold a reference
to an object of class A can also hold a reference
to an object belonging to any subclass of A.

The practical effect of this in our example is that an object of type Car can be assigned to a variable of type Vehicle. That is, it would be legal to say

```
Vehicle myVehicle = myCar;
```

or even

```
Vehicle myVehicle = new Car();
```

After either of these statements, the variable myVehicle holds a reference to a Vehicle object that happens to be an instance of the subclass, Car. The object "remembers" that it is in fact a Car,

and not **just** a Vehicle. Information about the actual class of an object is stored as part of that object. It is even possible to test whether a given object belongs to a given class, using the instanceof operator. The test:

```
if (myVehicle instanceof Car) ...
```

determines whether the object referred to by myVehicle is in fact a car.

On the other hand, if myVehicle is a variable of type Vehicle the assignment statement

```
myCar = myVehicle;
```

would be illegal because myVehicle could potentially refer to other types of vehicles that are not cars. The computer will not allow you to assign an int value to a variable of type short, because not every int is a short. Similarly, it will not allow you to assign a value of type Vehicle to a variable of type Car because not every vehicle is a car. As in the case of ints and shorts, the solution here is to use type-casting. If, for some reason, you happen to know that myVehicle does in fact refer to a Car, you can use the type cast (Car)myVehicle to tell the computer to treat myVehicle as if it were actually of type Car. So, you could say

```
myCar = (Car)myVehicle;
```

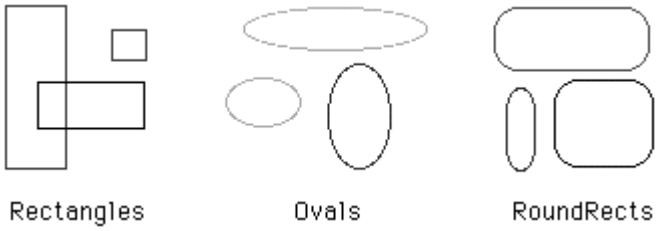
and you could even refer to ((Car)myVehicle).numberOfDoors. As an example of how this could be used in a program, suppose that you want to print out relevant data about a vehicle. You could say:

```
System.out.println("Vehicle Data:");
System.out.println("Registration number:  "
                          + myVehicle.registrationNumber);
    if (myVehicle instanceof Car) {
        System.out.println("Type of vehicle:  Car");
        Car c;
        c = (Car)myVehicle;
       System.out.println("Number of doors:  " +
                         c.numberOfDoors);
      }
      else if (myVehicle instanceof Truck) {
         System.out.println("Type of vehicle:  Truck");
         Truck t;
         t = (Truck)myVehicle;
         System.out.println("Number of axels:  " + t.numberOfAxels);
      }
      else if (myVehicle instanceof Motorcycle) {
         System.out.println("Type of vehicle:  Motorcycle");
         Motorcycle m;
         m = (Motorcycle)myVehicle;
         System.out.println("Has a sidecar:    " + m.hasSidecar);
      }
```

Note that for object types, when the computer executes a program, it checks whether type-casts are valid. So, for example, if myVehicle refers to an object of type Truck, then the type cast (Car)myVehicle will produce an error.

POLYMORPHERIC BEHAVIOUR OF OBJECTS

As another example, consider a program that deals with shapes drawn on the screen. Let's say that the shapes include rectangles, ovals, and roundrects of various colors.



Three classes, Rectangle, Oval, and RoundRect, could be used to represent the three types of shapes. These three classes would have a common superclass, Shape, to represent features that all three shapes have in common. The Shape class could include instance variables to represent the color, position, and size of a shape. It could include instance methods for changing the color, position, and size of a shape. Changing the color, for example, might involve changing the value of an instance variable, and then redrawing the shape in its new color:

```
class Shape {

    Color color;   // Color of the shape.  (Recall that class Color
                   // is defined in package java.awt.  Assume
                   // that this class has been imported.)

    void setColor(Color newColor) {
          // Method to change the color of the shape.
       color = newColor; // change value of instance variable
       redraw(); // redraw shape, which will appear in new color
    }

    void redraw() {
          // method for drawing the shape
       ? ? ?  // what commands should go here?
    }

    . . .          // more instance variables and methods

} // end of class Shape
```

Now, you might see a problem here with the method redraw(). The problem is that each different type of shape is drawn differently. The method setColor() can be called for any type of shape. How does the computer know which shape to draw when it executes the redraw()? Informally, we can answer the question like this: The computer executes redraw() by asking the shape to redraw **itself**. Every shape object knows what it has to do to redraw itself.

In practice, this means that each of the specific shape classes has its own redraw() method:

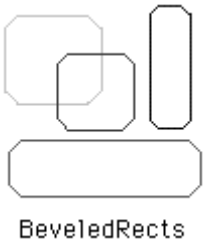
```
class Rectangle extends Shape {
   void redraw() {
      . . .  // commands for drawing a rectangle
   }
   . . . // possibly, more methods and variables
}
class Oval extends Shape {
   void redraw() {
      . . .  // commands for drawing an oval
   }
   . . . // possibly, more methods and variables
}
class RoundRect extends Shape {
   void redraw() {
      . . .  // commands for drawing a rounded rectangle
   }
   . . . // possibly, more methods and variables
}
```

If oneShape is a variable of type Shape, it could refer to an object of any of the types, Rectangle, Oval, or RoundRect. As a program executes, and the value of oneShape changes, it could even refer to objects of different types at different times! Whenever the statement

```
oneShape.redraw();
```

is executed, the redraw method that is actually called is the one appropriate for the type of object to which oneShape actually refers. There may be no way of telling, from looking at the text of the program, what shape this statement will draw, since it depends on the value that oneShape happens to have when the program is executed. Even more is true. Suppose the statement is in a loop and gets executed many times. If the value of oneShape changes as the loop is executed, it is possible that the very same statement "oneShape.redraw();" will call different methods and draw different shapes as it is executed over and over. We say that the redraw() method is **polymorphic**. A method is polymorphic if the action performed by the method depends on the actual type of the object to which the method is applied. Polymorphism is one of the major distinguishing features of object-oriented programming.

Perhaps this becomes more understandable if we change our terminology a bit: In object-oriented programming, calling a method is often referred to as sending a message to an object. The object responds to the message by executing the appropriate method. The statement "oneShape.redraw();" is a message to the object referred to by oneShape. Since that object knows what type of object it is, it knows how it should respond to the message. From this point of view, the computer always executes "oneShape.redraw();" in the same way: by sending a message. The response to the message depends, naturally, on who receives it. From this point of view, objects are active entities that send and receive messages, and polymorphism is a natural, even necessary, part of this view. Polymorphism just means that different objects can respond to the same message in different ways.



One of the most beautiful things about polymorphism is that it lets code that you write do things that you didn't even conceive of, at the time you wrote it. If for some reason, I decide that I want to add beveled rectangles to the types of shapes my program can deal with, I can write a new subclass, BeveledRect, of class Shape and give it its own redraw() method. Automatically, code that I wrote previously -- such as the statement oneShape.redraw() -- can now suddenly start drawing beveled rectangles, even though the beveled rectangle class didn't exist when I wrote the statement!

In the statement "oneShape.redraw();", the redraw message is sent to the object oneShape. Look back at the method from the Shape class for changing the color of a shape:

```
void setColor(Color newColor) {
   color = newColor; // change value of instance variable
   redraw(); // redraw shape, which will appear in new color
}
```

A redraw message is sent here, but which object is it sent to? Well, the setColor method is itself a message that was sent to some object. The answer is that the redraw message is sent to that same object, the one that received the setColor message. If that object is a rectangle, then it is the redraw() method from the Rectangle class that is executed. If the object is an oval, then it is the redraw() method from the Oval class. This is what you should expect, but it means that the redraw(); statement in the setColor() method does **not** necessarily call the redraw() method in the Shape class! The redraw() method that is executed could be in any subclass of Shape.

Again, this is not a real surprise if you think about it in the right way. Remember that an instance method is always contained in an object. The class only contains the source code for the method. When a Rectangle object is created, it contains a redraw() method. The source code for that method is in the Rectangle class. The object also contains a setColor() method. Since the Rectangle class does not define a setColor() method, the source code for the rectangle's setColor() method comes from the superclass, Shape. But even though the source codes for the two methods are in different classes, the methods themselves are part of the same object. When the rectangle's setColor() method is executed and calls redraw(), the redraw() method that is executed is the one in the same object.

ABSTRACT CLASSES

Whenever a Rectangle, Oval, or RoundRect object has to draw itself, it is the redraw() method in the appropriate class that is executed. This leaves open the question, What does the redraw() method in the Shape class do? How should it be defined?

The answer may be surprising: We should leave it blank! The fact is that the class Shape represents the abstract idea of a shape, and there is no way to draw such a thing. Only particular, concrete shapes like rectangles and ovals can be drawn. So, why should there even be a redraw() method in the Shape class? Well, it has to be there, or it would be illegal to call it in the setColor() method of the Shape class, and it would be illegal to write "oneShape.redraw();", where oneShape is a variable of type Shape. The computer would say, oneShape is a variable of type Shape and there's no redraw() method in the Shape class.

Nevertheless the version of redraw() in the Shape class will never be called. In fact, if you think about it, there can never be any reason to construct an actual object of type Shape! You can have **variables** of type Shape, but the objects they refer to will always belong to one of the subclasses of Shape. We say that Shape is an **abstract class**. An abstract class is one that is not used to construct objects, but only as a basis for making subclasses. An abstract class exists **only** to express the common properties of all its subclasses.

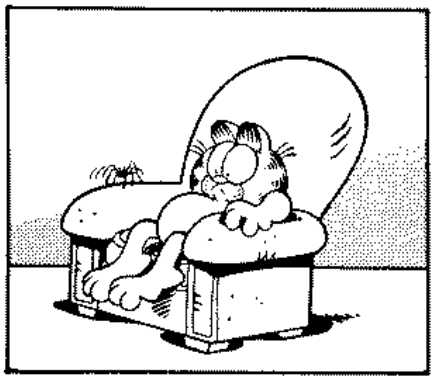
Similarly, we could say that the redraw() method in class Shape is an **abstract method**, since it is never meant to be called. In fact, there is nothing for it to do -- any actual redrawing is done by redraw() methods in the subclasses of Shape. The redraw() method in Shape has to be there. But it is there only to tell the computer that all Shapes understand the redraw message. As an abstract method, it exists merely to specify the common interface of all the actual, concrete versions of redraw() in the subclasses of Shape. There is no reason for the abstract redraw() in class Shape to contain any code at all.

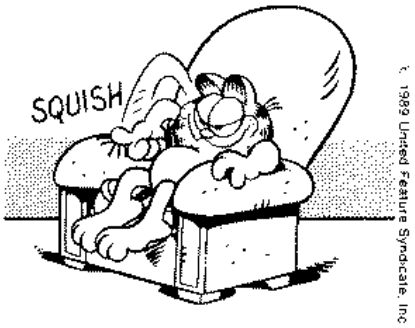
Shape and its redraw() method are semantically abstract. You can also tell the computer, syntactically, that they are abstract by adding the modifier "abstract" to their definitions. For an abstract method, the block of code that gives the implementation of an ordinary method is replaced by a semicolon. An implementation must be provided for the abstract method in any concrete subclass of the abstract class. Here's what the Shape class would look like as an abstract class:

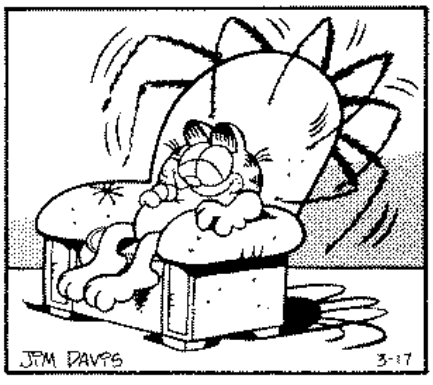
```
abstract class Shape {

    Color color;   // color of shape.

    void setColor(Color newColor) {
          // method to change the color of the shape
       color = newColor; // change value of instance variable
       redraw(); // redraw shape, which will appear in new color
    }
```

```
    abstract void redraw();
          // abstract method -- must be defined in
          // concrete subclasses
    . . .          // more instance variables and methods

} // end of class Shape
```

Once you have done this, it becomes illegal to try to create actual objects of type Shape, and the computer will report an error if you try to do so.







CHAPTER 10: EXCEPTION HANDLING

When a program violates the semantic constraints of the Java programming language, the Java virtual machine signals this error to the program as an **exception**. An example of such a violation is an attempt to index outside the bounds of an array. Some programming languages and their implementations react to such errors by peremptorily terminating the program; other programming languages allow an implementation to react in an arbitrary or unpredictable way. Neither of these approaches is compatible with the design goals of the Java platform: to provide portability and robustness. Instead, the Java programming language specifies that an exception will be thrown when semantic constraints are violated and will cause a non-local transfer of control from the point where the exception occurred to a point that can be specified by the programmer. An exception is said to be **thrown** from the point where it occurred and is said to be **caught** at the point to which control is transferred.

Programs can also throw exceptions explicitly, using **throw** statements.

Every exception is represented by an instance of the class **Throwable or one of its subclasses**; such an object can be used to carry information from the point at which an exception occurs to the handler that catches it.

When an exception is thrown, control is transferred from the code that caused the exception to the nearest dynamically-enclosing **catch** clause of a **try** statement that handles the exception.

A statement or expression is *dynamically enclosed* by a catch clause if it appears within the try block of the try statement of which the catch clause is a part, or if the caller of the statement or expression is dynamically enclosed by the catch clause.

Consider the following example:

```
class TestException extends Exception {
    TestException() { super(); }
    TestException(String s) { super(s); }
}

class Test {
    public static void main(String[] args) {
        for (int i = 0; i < args.length; i++) {
            try {
                thrower(args[i]);
                System.out.println("Test \"" + args[i]
                    + "\" didn't throw an exception");
            } catch (Exception e) {
                System.out.println("Test \"" + args[i]
                    + "\" threw a " + e.getClass()
                    + "\n        with message: "
                    + e.getMessage());
```

```
            }
        }
    }

static int thrower(String s) throws TestException {
    try {
        if (s.equals("divide")) {
            int i = 0;
            return i/i;
        }
        if (s.equals("null")) {
            s = null;
            return s.length();
        }
        if (s.equals("test"))
            throw new TestException("Test message");
            return 0;
    } finally {
        System.out.println("[thrower(\"" + s +
            "\") done]");
    }
  }
}
```

If we execute the test program, passing it the arguments:

```
divide null not test
```

it produces the output:

```
[thrower("divide") done]
Test "divide" threw a class
java.lang.ArithmeticException
        with message: / by zero
[thrower("null") done]
Test "null" threw a class
java.lang.NullPointerException
        with message: null
[thrower("not") done]
Test "not" didn't throw an exception
[thrower("test") done]
Test "test" threw a class TestException
        with message: Test message
```

This example declares an exception class TestException. The main method of class Test invokes the thrower method four times, causing exceptions to be thrown three of the four times. The try statement in method main catches each exception that the thrower throws. Whether the

invocation of thrower completes normally or abruptly, a message is printed describing what happened.

The possible exceptions in a program are organized in a hierarchy of classes, rooted at class **Throwable**, a direct subclass of Object. The classes **Exception** and **Error** are direct subclasses of Throwable. The class **RuntimeException** is a direct subclass of Exception.

Programs can use the pre-existing exception classes in throw statements, or define additional exception classes, as subclasses of Throwable or of any of its subclasses, as appropriate. To take advantage of the Java platform's compile-time checking for exception handlers, it is typical to define most new exception classes as **checked exception classes**, specifically as subclasses of Exception that are not subclasses of RuntimeException.

The class Exception is the superclass of all the exceptions that ordinary programs may wish to recover from. The class RuntimeException is a subclass of class Exception. The subclasses of RuntimeException are **unchecked exception classes**. The subclasses of Exception other than RuntimeException are all checked exception classes.

The class Error and its subclasses are exceptions from which ordinary programs are not ordinarily expected to recover.