One of the most powerful features of a programming language is the ability to define and manipulate**variables**. A variable is a named location that stores a value. Values may be numbers, text, images, sounds, and other types of data. To store a value, you first have to declare a variable.

String Messgage;

This statement is a declaration, because it declares that the variable named message has the type String. Each variable has a type that determines what kind of values it can store.For example, the inttype can store integers, and the char type can store characters.

Some types begin with a capital letter and some with lowercase.

To declare an integer variable named x, you simply type:

intx;

Note that x is an arbitrary name for the variable. In general, you should use names that indicate what the variables mean. For example, if you saw these declarations, you could probably guess what values would be stored:

String firstName;

String lastName;

inthour, minute;

int age;

This example declares two variables with type String and two with type

int. When a variable name contains more than one word, like firstName, it is conventional to capitalize the first letter of each word except the first. Variable names are case-sensitive, so firstNameis not the same as firstname or FirstName.

This example also demonstrates the syntax for declaring multiple variables with the same type on one line: hour and minute are both integers. Note that each declaration statement ends with a semicolon ( ;).

You can use any name you want for a variable. However there are about 50 reserved words, called **keywords**, which you are not allowed to use as variable names. These words include public, class, static, void, and int, which are used by the compiler to analyze the structure of the program.

|  |  |  |  |
| --- | --- | --- | --- |
| abstract | assert | boolean | break |
| byte | case | catch | char |
| class | const | continue | default |
| do | double | else | enum |
| extends | final | finally | float |
| for | goto | if | implements |
| import | instanceof | int | interface |
| long | native | new | package |
| private | protected | public | return |

|  |  |  |  |
| --- | --- | --- | --- |
| short | static | strictfp | super |
| switch | synchronized | this | throw |
| throws | transient | try | void |
| volatile | | while | |

Assignment

Now that we have declared variables, we want to use them to store values. We do that with an assignment statement.

message = "Hello!"; // give message the value "Hello!"

hour = 11; // assign the value 11 to hour

minute = 59; // set minute to 59

This example shows three assignments, and the comments illustrate different ways people sometimes talk about assignment statements. The vocabulary can be confusing here, but the idea is straightforward:

* When you declare a variable, you create a *named storage location*.
* When you make an assignment to a variable, you update its value.

As a general rule, a variable has to have the same type as the value you assign to it. For example, you cannot store a string in minute or an integer in message.

A common source of confusion is that some strings look like integers, but they are not. For example, message can contain the string "123", which is made up of the characters '1', '2', and '3'. But that is not the same thing as the integer 123.

message = "123"; // legal

message = 123; // not legal

Variables must be initialized (assigned for the first time) before they can be used. You can declare a variable and then assign a value later, as in the previous example. You can also declare and initialize on the same line:

String message = "Hello!";

inthour = 11;

intminute = 59;

Because Java uses the**=**symbol for assignment, it is tempting to interpret the statement a = b as a statement of equality. It is not!

Equality is commutative, and assignment is not. For example, in mathematics

ifa= 7 then 7 = a. In Java a = 7; is a legal assignment statement, but 7 = a; is not. The left side of an assignment statement has to be a variable name (storage location)

Also, in mathematics, a statement of equality is true for all time. If a = b now, a is always equal to b. In Java, an assignment statement can make two

variables equal, but they don't have to stay that way.

inta = 5;

intb = a; // a and b are now equal

a = 3; // a and b are no longer equal

The third line changes the value of a, but it does not change the value of b, so they are no longer equal.

Printing variables

Variables can be displayed using printor println. The following statements declare a variable named firstLine, assign it the value

"Hello, again!", and display that value.

String firstLine = "Hello, again!";

System.out.println(firstLine);

When we talk about displaying a variable, we generally mean the value of the variable.To display the name of a variable, you have to put it in quotes.

System.out.print("The value of firstLine is ");

System.out.println(firstLine);

Conveniently, the syntax for displaying a variable is the same regardless of its

type. For example:

inthour = 11;

intminute = 59;

System.out.print("The current time is ");

System.out.print(hour);

System.out.print(":") ;

System.out.print(minute);

System.out.println(".");

The output of this program is:

The current time is 11:59.

To output multiple values on the same line, it's common to use several print statements followed by printlnat the end. But don't forget the println!On many computers, the output from print is stored without being displayeduntil printlnis run; then the entire line is displayed at once.

Arithmetic operators

Operators are symbols that represent simple computations. For example, the addition operator is +, subtraction is -, multiplication is \*, and division is /. The following program converts a time of day to minutes:

inthour = 11;

intminute = 59;

System.out.print("Number of minutes since midnight: ");

System.out.println(hour \* 60 + minute):

In this program, hour \* 60 + minute is an expression, which represents a single value to be computed. When the program runs, each variable is replaced by its current value, and then the operators are applied. The values operators work with are called operands.

The result of the previous example is:

Number of minutes since midnight: 719

Expressions are generally a combination of numbers, variables, and operators.

When complied and executed, they become a single value.

For example, the expression 1 + 1 has the value 2. In the expression hour - 1,

Java replaces the variable with its value, yielding 11 - 1, which has the value

10. In the expression hour \* 60 + minute, both variables get replaced, yielding 11 \* 60 + 59. The multiplication happens first, yielding 660 + 59. Then the addition yields 719.

Addition, subtraction, and multiplication all do what you expect, but you might be surprised by division.

A more general solution is to use floating-point numbers, which can represent fractions as well as integers. In Java, the default floating-point type is called double, which is short for double-precision. You can create double variables and assign values to them using the same syntax we used for the other types:

doublepi;

pi = 3.14159;

Java performs **floating-point division**" when one or more operands are double values. So we can solve the problem we saw in the previous section:

doubleminute = 59.0;

System.out.print("Fraction of the hour”);

**Composition**

So far we have looked at the elements of a programming language variables, expressions, and statements in isolation, without talking about how to put them together. One of the most useful features of programming languages is their ability to take small building blocks and compose them. For example, we know how to multiply numbers and we know how to display values. We can combine these operations into a single statement:

System.out.println (17 \* 3);

Any arithmetic expression can be used inside a print statement. We've already seen one example:

System.out.println (hour \* 60 + minute);

You can also put arbitrary expressions on the right side of an assignment:

intpercentage;

percentage = (minute \* 100) / 60;

The left side of an assignment must be a variable name, not an expression.

That's because the left side indicates where the result will be stored, and expressions do not represent storage locations.

hour = minute + 1; // correct

minute + 1 = hour; // compiler error

The ability to compose operations may not seem impressive now, but we will see examples later on that allow us to write complex computations neatly and concisely. But don't get too carried away. Large, complex expressions can be hard to read and debug.

Types of errors

Three kinds of errors can occur in a program: **compile-time errors**, **run-time errors**, and **logic errors**. It is useful to distinguish among them in order to track them down more quickly.

**Compile-time errors** occur when you violate the syntax rules of the Java language. For example, parentheses and braces have to come in matching pairs. So (1 + 2) is legal, but 8) is not. In the latter case, the program cannot be compiled, and the compiler displays an error.

Error messages from the compiler usually indicate where in the program the error occurred, and sometimes they can tell you exactly what the error is.

public class Hello {

public static void main(String[] args) {

// generate some simple output

System.out.println ("Hello, World!");

}

}

If you forget the semicolon at the end of the print statement, you might get an error message like this:

File: Hello.java [line: 5]

Error: ';' expected

That's pretty good: the location of the error is correct, and the error message tells you what's wrong.

But error messages are not always easy to understand. Sometimes the compiler reports the place in the program where the error was detected, not where it actually occurred. And sometimes the description of the problem is more confusing than helpful.

For example, if you leave out the closing brace at the end of main (line 6), youmight get a message like this:

File: Hello.java [line: 7]

Error: reached end of file while parsing

There are two problems here. First, the error message is written from the compiler's point of view, not yours. **Parsing** is the process of reading a program before translating; if the compiler gets to the end of the \_le while still parsing that means something was omitted. But the compiler doesn't know what. It also doesn't know where. The compiler discovers the error at the end of the program (line 7), but the missing brace should be on the previous line.

Error messages contain useful information, so you should make an effort to read and understand them. But don't take them too literally.

During the first few weeks of your programming career, you will probably spend a lot of time tracking down compile-time errors. But as you gain experience, you will make fewer mistakes and find them more quickly.

The second type of error is a **run-time error**, so-called because it does not appear until after the program has started running. In Java, these errors occur while the interpreter is executing byte code and something goes wrong. These errors are also called “exceptions" because they usually indicate that something exceptional (and bad) has happened.

For example, if you accidentally divide by zero you will get a message like this:

Exception in thread "main" java.lang.ArithmeticException: / by zero

atHello.main(Hello.java:5)

Error messages sometimes contain additional information that won't make sense yet. So one of the challenges is to figure out where to find the useful parts without being overwhelmed by extraneous information. Also, keep in find that the line where the program crashed may not be the line that needs to be corrected.

The third type of error is the logic error. If your program has a logic error, it will compile and run without generating error messages, but it will not do the right thing. Instead, it will do exactly what you told it to do. For example, here is a version of the hello world program with a logic error:

public class Hello {

public static void main(String[] args) {

System.out.println("Hello, ");

System.out.println("World!");

}

}

Identifying logic errors can be hard because you have to work backwards, looking at the output of the program, trying to figure out why it is doing the wrong thing, and how to make it do the right thing. Usually the compiler and the interpreter can't help you, since they don't know what the right thing is.

**Definitions**

* **Variable**: A named storage location for values. All variables have a type, which is declared when the variable is created.
* **Value**: A number, string, or other data that can be stored in a variable. Every value belongs to a type (for example, intor String).
* **Declaration**: A statement that creates a new variable and specifies its type.
* **Type**: Mathematically speaking, a set of values. The type of a variable deter- mines which values it can have.
* **Syntax**: The structure of a program; the arrangement of the words and symbols it contains.
* **Keyword**: A reserved word used by the compiler to analyze programs. You cannot use keywords (like public, class, and void) as variable names.
* **Assignment**: A statement that gives a value to a variable.
* **Initialize**: To assign a variable for the first time.
* **State**: The variables in a program and their current values.
* **State Diagram**: A graphical representation of the state of a program at a point in time.
* **Operator**: A symbol that represents a computation like addition, multiplication, or string concatenation.
* **Operand**: One of the values on which an operator operates. Most operators in Java require two operands.
* **Expression**: A combination of variables, operators, and values that represents a single value. Expressions also have types, as determined by their operators and operands.
* **Floating-point**: A data type that represents numbers with an integer part and a fractional part. In Java, the default floating-point type is double.
* **Rounding error**: The difference between the numbers we want to represent and the nearest floating-point number.
* **Concatenate**: To join two values, often strings, end-to-end. The rules that determine in what order operations are evaluated.
* **Composition**: The ability to combine simple expressions and statements into compound expressions and statements.
* **Compile-time error**: An error in the source code that makes it impossible to compile. Also called a \syntax error".
* **Parse**: To analyze the structure of a program; what the compiler does first.
* **Run-time error**: An error in a program that makes it impossible to run to completion. Also called an exception".
* **Logic error**: An error in a program that makes it do something other than what the programmer intended.