CHAPTER 3

Primitive Types and Casts

OBJECTIVES

- Learn how primitive types are represented
- Learn how to use casts
- Learn how mixed types are handled in arithmetic expressions
- Learn how to use relational and boolean operators
- Learn more precedence rules

3.1 Representation of Primitive Types in Memory

`int and double` are examples of **primitive types**. A variable with a primitive type corresponds to single box in memory that has a name. Thus, a variable that has a primitive type can hold only one value at a time. We can access this value via the name of the box that holds it. For example, suppose within a method we have the following code:

```
int x;
double y;
x = 3;
y = 5.5;
```

After the two assignment statements are executed, the variables `x` and `y` can be represented with

We show the `y` box bigger than the `x` box because it is, indeed, a bigger box in memory. A `double` value requires more memory space than an `int` value. However, both `x` and `y` correspond to single, named boxes—each of which can contain a single value. Nonprimitive types, like `String`, have more complicated structures associated with them.

At this point we need to digress a little to discuss how memory is organized in a computer. Everything—numbers, strings, instructions—in main memory is represented with **binary numbers**—numbers that contain only 0's and 1's. Each 0 or 1 in a binary number is called a **bit**. A sequence of 8 bits is called a **byte**.

Main memory is an array of individual cells, each of which can hold 1 byte. One cell of main memory—that is, 1 byte—is too small to hold the range of values that a variable of type `int` can have. So Java uses 4 consecutive cells—that is, it uses 4 consecutive bytes for a variable of type `int`. For a `double` variable, it uses 8 consecutive bytes.

What if we want to store an integer in a variable but it is too big to fit into an `int` variable? Recall that the box for an `int` variable is 4 bytes long. With 4 bytes, the values can range roughly from -2 billion to $+2$ billion. However, Java has another primitive type, `long`, that holds integers. A `long` variable uses 8 bytes. Because it uses more bytes than an `int` variable, a `long` variable can hold a much larger range of integers. `long` values range from roughly -10^{19} to 10^{19}. If you need to hold really large integers, use type `long` instead of type `int`.

Java also has two more types that are for integers: `byte` and `short`. `byte` variables use only 1 byte so its range is the least. `short` variables use 2 bytes so its range lies between

	Type	Size	Range
Arithmetic types	`byte`	1 byte	−128 to 127
	`short`	2 bytes	−32768 to 32767
	`int`	4 bytes	approximately −2 billion to 2 billion
	`long`	8 bytes	approximately -10^{19} to 10^{19}
	`float`	4 bytes	approximately -10^{38} to 10^{38}, 8 significant digits
	`double`	8 bytes	approximately -10^{308} to 10^{308}, 16 significant digits
	`char`	2 bytes	can hold any single character
	`boolean`	1 byte	holds either `true` or `false`

Figure 3.1 Primitive Types

that of `byte` and `int`. Fig. 3.1 shows each primitive type in Java, along with its associated size and range.

Java has two types for numbers with a fractional part: `double`, which we have already seen, and `float`. These types differ in the number of bytes used by the corresponding variables. A `double` variable uses 8 bytes of memory; a `float` variable uses only 4 bytes. Because a variable of type `double` uses more bytes than a variable of type `float`, a `double` variable has greater range and can hold values with more precision (i.e., with more significant digits). `double` variables can hold double the significant digits that `float` variables can hold—hence the name `double`. If you perform computations with variables and constants of type `double`, you will, in general, get more accurate and more precise answers than if you use `float`. Thus, you should almost always use `double` rather than `float` to hold fractional numbers. The only circumstance that would make sense using `float` rather than `double` is if your program has to be as small as possible. Because `float` variables occupy less memory than `double` variables, a program with `float` variables would be smaller than the same program with `double` variables.

We distinguish literal constants of type `double` from literal constants of type `float` by appending the letter `f` to the end of `float` constants. For example, `3.33` is a `double` constant and will, therefore, be represented using 8 bytes. `3.33f`, on the other hand, is a `float` constant and will, therefore, be represented using 4 bytes.

3.2 Casts

Suppose `b`, `s`, `i`, `l`, `f`, and `d` are variables of type `byte`, `short`, `int`, `long`, `float`, and `double`, respectively. Then the assignment statement

```
i = b;      // i is int, b is byte
```

is legal because the value in `b` will always fit into the variable `i` (recall type `byte` corresponds to 1 byte, and `int` corresponds to 4 bytes). For the same reason, the following statements are also legal:

```
s = b;      // s is short, b is byte
l = b;      // l is long, b is byte
f = b;      // f is float, b is byte
d = b;      // d is double, b is byte
```

For all these statements, a copy of the value in `b` is automatically converted to match the type of the variable on the left. The converted copy is then assigned to the variable on the left. We call this automatic type conversion **type coercion**. For example, if the value in `b` is `3`, a copy of this value is converted to the `double` value `3.0` before it is assigned to `d`.

If we switch the left and right sides in each of the preceding assignment statements, they become illegal. For example, the following statement is illegal because the value in `i` (which occupies 4 bytes) may not fit into `b` (which occupies 1 byte):

```
b = i;      // illegal statement
```

This statement will cause a compile-time error. However, we can modify it to make it legal. We can insert a **cast** to force a type change. A cast consists of the name of a type surrounded by parentheses. For example, in the following statement, we are casting `i` to `byte`:

```
b = (byte)i;          // legal statement because of the cast
```

The cast here does not change the value in `i` or the type of `i`. Instead, it changes the type of a *copy* of the value in `i` to `byte`. The resulting value is then assigned to `b`. To change the value obtained from `i` to type `byte`, the value is **truncated** to 1 byte (i.e., its leftmost 3 bytes are chopped off). Obviously, if a value cannot fit in 1 byte, then its truncation to 1 byte will result in a different value. For example, if the value 257 is truncated to 1 byte, the resulting value is 1. Although the preceding assignment statement is legal, you probably would not want to use it unless you know in advance that `i` will have a value that fits in `b`. The range of values that fit in `b` is −128 to 127.

To assign `d` (which is type `double`) to `b` (which is type `byte`), we must cast `d` to `byte`:

```
b = (byte)d;
```

Here the *integer part* of a copy of the value in `d` is first truncated to 1 byte before it is assigned to `b`. Although the cast makes this statement legal, it does not ensure that the statement will work in a reasonable way when it is executed. If `d` contains 3.12, then the value of its integer part, 3, fits into `b`. Thus, the truncation of its integer part to 1 byte provides the integer part intact. `b` is assigned 3. The values of `b` and `d`, in this case, will differ by only the fractional part of `d`. But if `d` contains 3 billion, then the truncated value of its integer part will be some number between −128 and 127. In this case, the integer value assigned to `b` will be quite different from the integer part of the value in `d`.

The arithmetic types in Figure 3.1 form a hierarchy. As we go from `byte` to `double` in Fig. 3.1, the range of the corresponding variables gets progressively larger. A value of any arithmetic type in the table can be assigned without a cast to a variable of that type or any arithmetic type below it in the table. The simple operative rule is this: *You do not need a cast if the type of the variable to be assigned has a range equal to or greater than the range corresponding to the type of the value to be assigned.* For example, you can assign a `byte` value to an `int` variable without a cast because the range corresponding to `int` is greater than the range corresponding to `byte`.

Casts are not always legal. For example, the following cast is illegal:

```
x = (int)"hello";      // illegal cast
```

where `x` has type `int`. `String` and `int` are **incompatible types**. A string cannot be cast to an `int`, and vice versa.

3.3 Arithmetic Expressions with Mixed Types

A computer does not have circuitry to perform computations on mixed types. For example, it does not have circuitry to add an `int` value to a `double` value. Thus, to add an `int` value and a `double` value, one value must be converted to the type of the other value. Here is the rule for such conversions: *In a mixed-type operation, the value whose type corresponds to a smaller range of values is converted to match the type of the other value.* For example, in

```
System.out.println(2 + 3.5);     // displays 5.5
```

the `2` is automatically converted to `2.0` (i.e., it is converted to type `double`) so its type matches the type of `3.0`. The `2` is converted to `double` rather than the `3.0` to `int` because the type of `2` (which is `int`) has a smaller range than the type of `3.0` (which is `double`).

Automatic type conversion occurs only when the two operands for an operator have different types. For example, in

```
System.out.println(1.0 + 7/2);              // displays 4.0
```

the first operation performed is the division (because division has higher precedence than addition). Because both operands in this division (`7` and `2`) have type `int`, automatic type conversion does *not* occur. Because both operands have type `int`, the division is performed by the integer division circuitry in the computer. The result of the division is 3. Next, the `double` `1.0` and the `int` `3` are added. Because of the mixed type, the `3` is converted to the `double` `3.0`. The addition then yields the result `4.0`. If we want the entire expression to be evaluated using `double` values, we have to change at least one of the operands of the division to type `double`. That will, in turn, force the other operand to type `double`. For example, we can rewrite `7` as a `double` constant:

```
System.out.println(1.0 + 7.0/2);            // displays 4.5
```

or we can cast `7` to `double`:

```
System.out.println(1.0 + (double)7/2);      // displays 4.5
```

Casts have higher precedence than all the arithmetic operators. Thus, in the preceding statement, the cast is applied to `7` *before* the division. The cast causes `7` to be converted to `double`, which, in turn, causes `2` to be converted to `double`. The division of these two `double` values results in `3.5`. If the cast had lower precedence than division, then the division would be performed *before* the cast, in which case both operands in the division would be type `int`. If this were the case, the division would result in `3` rather than `3.5`.

3.4 Type char

Another primitive type in Java is `char`. A `char` variable can hold a single character. A literal `char` constant consists of a single character enclosed by *single* quotes. For example, `'a'`, `'A'`, `'5'`, and `'&'` are all `char` constants. Note that `'a'` and `"a"` are not the same. `'a'` (with single quotes) is a `char` constant. `"a"` (with double quotes) is a `String` constant.

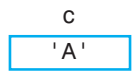
Suppose we execute

```
c = 'A';
```

where `c` is declared with

```
char c;
```

We can then represent the variable c and its contents with



But what is actually inside the c box? Recall that all data in computer memory is in binary. Thus, the c box actually contains a binary number—the binary number that represents the uppercase letter A. Every character has a corresponding number that represents it. For example, the number that represents the character 'A' is 65 decimal. Thus, the c box shown actually contains the binary equivalent of the decimal number 65. The specific encoding of characters that Java uses is called Unicode.

To get the literal char constant for the symbol $, we simply surround $ with single quotes to get '$'. The left quote starts the char constant; the right quote ends the char constant. But how do we represent the char constant that is the single quote? If we write ''', the middle single quote ends the constant. The right quote then appears as garbage and will trigger a compile-time error. To remedy this problem, we simply backslash the middle quote to get '\''. The backslash makes the middle quote into an ordinary character—ordinary in the sense that it does not end the char constant. Sequences that start with the backslash are called **escape sequences**.

We can also use escape sequences to represent some special char constants. For example, '\n' represents the newline character (this is the character at the end of every line of a text file). Fig. 3.2 shows the common escape sequences that we can use in Java programs.

'\n' (the newline character), '\r' (the carriage return character), '\t' (the tab character), and the space character are called **whitespace** characters. These characters do not produce any marking on the display or on a paper when printed—hence, the name "whitespace."

\'	ordinary single quote (i.e., a quote that does not end a char constant)
\"	ordinary double quote (i.e., a quote that does not end a String constant)
\\	ordinary backslash
\n	newline character
\r	carriage return character
\t	tab character

Figure 3.2

We can use escape sequences individually in `char` constants or together with other characters in string constants. For example, in

```
char c;
c = '\'';
```

we are using the escape sequence `\'` in a `char` constant. In

```
System.out.println("I read \"War and Peace\" in 10 minutes.");
```

we are using the escape sequence `\"` twice within a string constant. This statement displays

```
I read "War and Peace" in 10 minutes.
```

3.5 Type `boolean`

There are many values of type `int`. They range from about −2 billion to 2 billion. However, there are only two values of type `boolean`: `true` and `false`. A variable whose type is `boolean` can be assigned only the values `true` or `false`. For example, the following code assigns `true` to `boo1` and then assigns the value in `boo1` to `boo2`:

```
boolean boo1, boo2;
boo1 = true;       // boo1 contains true
boo2 = boo1;       // contents of boo1 (true) assigned to boo2
```

Be sure *not* to use quotes around `true` and `false` because then you would get string constants. For example, if `boo1` is a `boolean` variable, the following statement is illegal:

```
boo1 = "true";     // illegal because "true" is type String not boolean
```

3.6 Relational Operators

The relational operators are "<" (less than), "<=" (less than or equal), ">" (greater than), ">=" (greater than or equal), "==" (equal), and "`!=`" (not equal). The relational operators perform comparisons. They yield a `boolean` (i.e., `true` or `false`) result. For example, when we execute the following statement, `true` is displayed because `2` is less than `3`:

```
System.out.println(2 < 3);              // displays true
```

The operands in the expression `2 < 3` are type `int`, but the result is `boolean`. Thus, if we want to assign the value of this expression to a variable, that variable should have the type

boolean. For example, the following sequence assigns the value of 2 < 3 (which is true) to boo and then displays the value in boo:

```
boolean boo;
boo = 2 < 3;                          // true is assigned to boo
System.out.println(boo);              // displays true
```

As in arithmetic expressions, we can use variables in relational expressions. For example, if x and y are int variables, then x < y is a valid relational expression whose true/false value depends on the values of x and y. For example, in the following code, x < y is true:

```
int x, y;
x = 2;
y = 3;
System.out.println(x < y);            // displays true
```

Now consider the following code:

```
int x;
x = 2;
System.out.println(1 < x < 3);        // illegal!!!
```

x is equal to 2, which lies between 1 and 3. So you would expect the println statement here to display true. However, the expression 1 < x < 3 is, in fact, illegal in Java. You will get a compile-time error. Let's see why this happens. This expression has two operators. They are both "<" so they, of course, have the same precedence. Because "<" is left associative, the computer will perform the two "<" operations in left-to-right order. That is, it first evaluates 1 < x, which yields the value true because 1 is less than x. The value true then becomes the left operand for the second "<" operator. That is, the second operation is, in effect, true < 3, which makes no sense. You cannot compare true with 3. We want to compare x—not true—with 3. We will learn how to correctly test if x is between 1 and 3 in Section 3.7.

Two warnings about the relational operators: First, the two-character operators, "<=" and ">=", must not have any embedded spaces. For example, you cannot write

```
System.out.println(2 < = 3);
                   space here is illegal
```

Second, the equality operator uses two equal signs, not one. To test if x is equal to y use

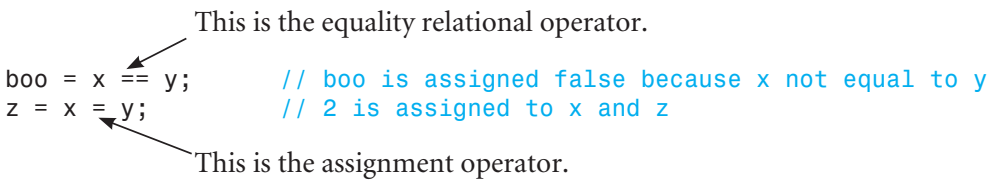
```
x == y      // testing for equality
```

not

```
x = y        // assignment of y to x
```

For example, in the first assignment statement that follows, we test if x is equal to y and assign the result to boo. In the second assignment statement, we assign the value in y (which is 2) to x and then to z:

```
boolean boo;
int x = 1, y = 2, z;
```



Multiple assignments in a single statement are performed right to left. Thus, in

```
z = x = y;
```

the value in y is first assigned to x and then to z. The assignment operator is **right associative**—that is, multiple assignments in a statement are performed right to left.

3.7 Boolean Operators

The boolean operators are "&&" (AND), "||" (OR), and "!" (NOT). Their operands must be boolean, and they yield boolean values. For example, suppose we declare p, q, and r with

```
boolean p = true, q = false, r;
```

and then execute

```
r = p && q;
```

The boolean operator "&&" yields true only if *both* p and q are true. In this example, q is false. Thus, the value of the right side is false. This value is assigned to r.

We can represent how the "&&" operator works with a truth table (see Fig. 3.3). A **truth table** shows the true/false value that an operator yields for every possible combination of values for its operands.

p	q	p && q
false	false	false
false	true	false
true	false	false
true	true	true

Figure 3.3

For example, the first line in Fig. 3.3 indicates that if `p` is `false` and `q` is `false`, then `p && q` is also `false`. The only combination of operands for which the result is `true` is when both operands are `true`.

Fig. 3.4 shows the truth table for the `boolean` OR operator "`||`":

The "`||`" operator yields `true` if `p` or `q` or both are `true`.

The boolean NOT operator "`!`" takes only one operand so its truth table is simpler (see Fig. 3.5). This operator precedes the boolean value, variable, or expression on which it

p	q	p || q
false	false	false
false	true	true
true	false	true
true	true	true

Figure 3.4

p	!p
false	true
true	false

Figure 3.5

operates. For example, to apply the NOT operator to the boolean variable p, we put "!" in front of p to get !p.

If p is false, then !p is true; if p is true, then !p is false. The "!" operator "flips" the truth value of its operand. "!" is an example of a unary operator. A **unary operator** takes only one operand. **Binary operators**, like "+" and "&&", take two operands.

The boolean operators are often used in conjunction with the relational operators. For example, consider the statement

```
boo = x > 1 && x < 3;
```

where boo has type boolean and x has type int. In this statement, x > 1 and x < 3 are evaluated first (because ">" and "<" have higher precedence than "&&"). The true/false values of these two relational expressions are then ANDed by the "&&" operator. Thus, the value of the entire right side is true only if both x is greater than 1 and x is less than 3. In mathematical notation, we would indicate that x is greater than 1 and less than 3 with

```
1 < x < 3
```

As we have already pointed out, this expression is illegal in Java. To capture this relationship in Java, we must use two relational expressions, x > 1 and x < 3, joined by the "&&" operator.

3.8 Sample Program

Let's now examine the program in Fig. 3.6. It uses a variety of primitive types.

The values of the right sides of the assignment statements on Lines 8 and 9 are boolean—that is, they are true or false. Thus, the type of the variables to which these values are assigned should also be boolean. The println statements on Lines 10, 11, and 12 display the true/false values of the boolean expressions they contain. On Line 16, the value in s is truncated to 1 byte, resulting in a value of 1. 1 is then assigned to b. On Line 19, both constants have type float because they are suffixed with the letter f. On Line 20, both constants have type double. Thus, the computation on Line 20 is performed with greater precision (i.e., more significant digits). We can see this from the output produced by Lines 22 and 23. The value displayed for d has twice the significant digits as the value

```
 1 class PrimitiveTypes
 2 {
 3    public static void main(String[] args)
 4    {
 5        int x = 1, y = 2;
 6        System.out.println(x < y);        // displays true
 7        boolean boo1, boo2;
 8        boo1 = x < y;                     // boo1 assigned true
 9        boo2 = false;                     // boo2 assigned false
10        System.out.println(boo1 && boo2); // displays false
11        System.out.println(boo1 || boo2); // displays true
12        System.out.println(!boo2);        // displays true
13
14        byte b;
15        short s = 257;
16        b = (byte)s;              // truncated value assigned
17        System.out.println(b);    // displays 1
18
19        float f = 1.0f/3.0f;    // f suffix means float constant
20        double d = 1.0/3.0;     // no suffix means double constant
21
22        System.out.println(f);    // displays 0.33333334
23        System.out.println(d);    // displays 0.3333333333333333
24
25        d = 3.99999999;
26        long lg;
27        lg = (long)d;             // fractional part truncated
28        System.out.println(lg);   // displays 3
29    }
```

Figure 3.6

displayed for `f`. When `d` is cast to `long` in Line 27, its fractional part is truncated, leaving the integer `3`, which is assigned to `lg`.

3.9 More on Operator Precedence

In the statement,

```
b = x > 2 && x < 4;
```

we do not have to use any parentheses because the relational operators, "<" and ">", have a higher precedence than the boolean operator, "&&". Thus, the relational subexpressions, `x > 2` and `x < 4`, are evaluated first (which is the order we want). Their truth values are then ANDed by the "&&" operator.

Fig. 3.7 lists all the operators we have seen so far, from highest to lowest precedence.

Operator	Description	
`!, ++, --,+, -`	not, increment, decrement, unary plus, unary minus	`Highest precedence`
`new, (type)`	`new` operator, cast	
`*, /, %`	multiplication, division, remainder	
`+, -`	addition, subtraction	
`<, > <=, >=`	relational operators except equality/inequality	
`==, !=`	equality/inequality relational operators	
`&&`	and	
`||`	or	
`=`	assignment	
		`Lowest precedence`

Figure 3.7

LABORATORY

Laboratory 3 Prep

1. What values are assigned to `b1`, `b2`, `b3`, and `b4` by the following code? Run the code to check your answers.

```
byte b1, b2, b3, b4;
int i = 3, j = 258;
double x = 4.999, y = 4.0E2;
b1 = (byte)i;
b2 = (byte)j;
b3 = (byte)x;
b4 = (byte)y;
```

2. Incorporate the following code in a program and compile. Why is there a compile-time error?

```
int x;
x = 999999999999;
```

3. Incorporate the following code in a program and compile. Why is there a compile-time error?

```
float x;
x = 2.0;
```

4. Why do the first and third `println` statements that follow display different values? Why do the second and fourth `println` statements display different values?

```
int x = 1;
System.out.println(x == 2);
System.out.println(x);
System.out.println(x = 2);
System.out.println(x);
```

5. What is displayed by the following statement:

```
System.out.println(100.0 + 50/4);
```

Laboratory 3

1. Run a program with the following code to determine if it is legal to concatenate two `char` constants, or to concatenate a `char` constant to a string.

```
System.out.println('a' + 'b');            // is this legal?
System.out.println('a' + "bcd");          // is this legal?
```

LABORATORY

2. Write a program that initializes the `int` variables `x`, `y`, and `z` to `1`, `2`, and `3`, respectively, and then executes

   ```
   x = y = z;
   ```

   From the values assigned to `x`, `y`, and `z`, determine the order in which the assignments occur. Left to right, or right to left? Is the assignment operator left or right associative?

3. Is it legal to assign a `char` value to an `int` variable with a cast? For example, is the following code legal?

   ```
   int i;
   i = (int)'A';       // is this legal?
   ```

   What value is assigned to `i`? Is the cast required? Is it legal to assign an `int` value to a `char` variable with a cast? Is a cast required? Run a test program to check your answers.

4. What is the effect of the following code:

   ```
   char c = 'N';
   c = (char)((int)'a' – (int)'A' + (int)c);
   System.out.println(c);
   ```

   Run a test program to check your answer. The code for every lowercase letter is 32 more than the code for the corresponding uppercase letter. Thus, `(int)'a' – (int)'A'` is equal to 32. If you add 32 to the character '`N`', what character do you get?

5. Is it legal to cast a `boolean` constant to `int`. For example, is the following statement legal:

   ```
   int i;
   i = (int)true;
   ```

   Run a test program to check your answer. If the code is legal, display the value assigned to `i`, and then repeat using the `boolean` constant `false` in place of `true`.

6. Write a program that determines if the value of `x` is less than 1 or greater than 10. If it is, your program should display `true`; otherwise, it should display `false`. Test your program for `x` equal to `-3`, `1`, `5`, `10`, and `20`.

7. Write a program that assigns `1.2345` to `x`, and then extracts from `x` and separately displays its integer part and its fractional part. `x` should have type `double`.

LABORATORY

Homework 3

1. Write a program that computes and displays (with an appropriate label) the exact average of `1`, `2`, `3`, and `4` (the correct answer is `2.5`).
2. Write a program that outputs the truth table for the "`&&`" operator. Use the "`&&`" operator in your program to determine the values of `p && q` that appear in your table. For example, to display the value of "`&&`" when both operands are `false`, use

   ```
   System.out.println("false    false    " + false && false);
   ```

   Do *not* use

   ```
   System.out.println("false    false    false");
   ```

   Your output should look like this:

p	q	p && q
false	false	false
false	true	false
true	false	false
true	true	true

3. Write a program that computes and displays the truth table for `!(p && q)`. Use the technique described in Homework Exercise 2.
4. Write a program that computes and displays the truth table for `!p || !q`. Use the technique described in Homework Exercise 2. Compare your results with those from Homework Exercise 3. The equivalence of `!(p && q)` and `!p || !q` is one of **DeMorgan's Laws** (see Homework Exercise 6 for the other DeMorgan's Law).
5. Write a program that computes and displays the truth table for `!(p || q)`. Use the technique described in Homework Exercise 2.
6. Write a program that computes and displays the truth table for `!p && !q`. Use the technique described in Homework Exercise 2. Compare your results with those from Homework Exercise 5. The equivalence of `!(p || q)` and `!p && !q` is the one of **DeMorgan's Laws**.
7. Same as Homework Exercise 2, but use the "`^`" (exclusive OR) operator.
8. `charAt` is a method in a `String` object that returns the character at the specified position. For example, if `c` is type `char` and `s` is a reference to a `String` object, then the

LABORATORY

following assignment statement assigns `c` the first character in the `String` object to which `s` points:

```
c = s.charAt(0);
```

Write a program in which you assign `"AB3cd$"` to `s`. Display each character in this string on a separate line. Display all the letters in uppercase. Thus, your program should display AB3CD$. Your program should work for any five-character string. Do not use the `toUpperCase` method. (*Hint:* See Lab Exercise 4.)