CHAPTER 1

Introductory Java Language Features

Fifty loops shalt thou make . . .
—Exodus 26:5

Chapter Goals

- Packages and classes
- Types and identifiers
- Operators
- Input/output
- Storage of numbers
- Binary and hexadecimal numbers
- Control structures
- Errors and exceptions

The AP Computer Science course includes algorithm analysis, data structures, and the techniques and methods of modern programming, specifically, object-oriented programming. A high-level programming language is used to explore these concepts. Java is the language currently in use on the AP exam.

Java was developed by James Gosling and a team at Sun Microsystems in California; it continues to evolve. The AP exam covers a clearly defined subset of Java language features that are presented throughout this book. The College Board website, *http://www.collegeboard.com/student/testing/ap/subjects.html*, contains a complete listing of this subset.

Java provides basic control structures such as the `if-else` statement, `for` loop, for-each loop, and `while` loop, as well as fundamental built-in data types. But the power of the language lies in the manipulation of user-defined types called objects, many of which can interact in a single program.

PACKAGES AND CLASSES

A typical Java program has user-defined classes whose objects interact with those from Java class libraries. In Java, related classes are grouped into *packages*, many of which are provided with the compiler. You can put your own classes into a package—this facilitates their use in other programs.

2. Typically, the class that contains the `main` method does not contain many additional methods.
3. The words `class`, `public`, `static`, `void`, and `main` are *reserved words*, also called *keywords*.
4. The keyword `public` signals that the class or method is usable outside of the class, whereas `private` data members or methods (see Chapter 2) are not.
5. The keyword `static` is used for methods that will not access any objects of a class, such as the methods in the `FirstProg` class in the example on the previous page. This is typically true for all methods in a source file that contains no *instance variables* (see Chapter 2). Most methods in Java do operate on objects and are not static. The `main` method, however, must always be static.
6. The program shown on the previous page is a Java *application*. This is not to be confused with a Java *applet*, a program that runs inside a web browser or applet viewer. Applets are not part of the AP subset.

Javadoc Comments

The Javadoc comments `@param`, `@return`, and `@throws` are part of the AP Java subset. Here is an example.

```
/** Puts obj at location loc in this grid, and returns
 *  the object previously at this location.
 *  Returns null if loc was previously unoccupied.
 *  Precondition: obj is not null, and loc is valid in this grid.
 *  @param loc the location where the object will be placed
 *  @param obj the object to be placed
 *  @return the object previously at the specified location
 *  @throws NullPointerException if the object is null
 */
public E put(Location loc, E obj)
```

This will produce the following Javadoc output:

---

put

`public E` **put** `(Location loc, E obj)`

Puts `obj` at location `loc` in this grid, and returns
the object previously at this location.
Returns `null` if `loc` was previously unoccupied.
Precondition: `obj` is not `null`, and `loc` is valid in this grid.

Parameters:

`loc` - the location where the object will be placed
`obj` - the object to be placed

Returns:

the object previously at the specified location

Throws:

`NullPointerException` - if the object is `null`

```
int num = 5;
double realNum = num;     //num is cast to double
```

Assigning a double to an int without a cast, however, causes a compile-time error. For example,

```
double x = 6.79;
int intNum = x;      //Error. Need an explicit cast to int
```

Note that casting a floating-point (real) number to an integer simply truncates the number. For example,

```
double cost = 10.95;
int numDollars = (int) cost;     //sets numDollars to 10
```

If your intent was to round cost to the nearest dollar, you needed to write

```
int numDollars = (int) (cost + 0.5);  //numDollars has value 11
```

To round a negative number to the nearest integer:

```
double negAmount = -4.8;
int roundNeg = (int) (negAmount - 0.5);   //roundNeg has value -5
```

The strategy of adding or subtracting 0.5 before casting correctly rounds in all cases.

Storage of Numbers

INTEGERS

Integer values in Java are stored exactly, as a string of bits (binary digits). One of the bits stores the sign of the integer, 0 for positive, 1 for negative.

The Java built-in integral type, byte, uses one byte (eight bits) of storage.

The picture represents the largest positive integer that can be stored using type byte: $2^7 - 1$.

Type int in Java uses four bytes (32 bits). Taking one bit for a sign, the largest possible integer stored is $2^{31} - 1$. In general, an n-bit integer uses $n/8$ bytes of storage, and stores integers from -2^{n-1} to $2^{n-1} - 1$. (Note that the extra value on the negative side comes from not having to store -0.) There are two Java constants that you should know. Integer.MAX_VALUE holds the maximum value an int can hold, $2^{31} - 1$. Integer.MIN_VALUE holds the minimum value an int can hold, -2^{31}.

Built-in types in Java are byte (one byte), short (two bytes), int (four bytes), and long (eight bytes). Of these, only int is in the AP Java subset.

FLOATING-POINT NUMBERS

There are two built-in types in Java that store real numbers: float, which uses four bytes, and double, which uses eight bytes. A *floating-point number* is stored in two parts: a *mantissa*, which specifies the digits of the number, and an exponent. The JVM (Java Virtual Machine) represents the number using scientific notation:

$$\text{sign} * \text{mantissa} * 2^{\text{exponent}}$$

Final Variables

A *final variable* or *user-defined constant*, identified by the keyword `final`, is used to name a quantity whose value will not change. Here are some examples of `final` declarations:

```
final double TAX_RATE = 0.08;
final int CLASS_SIZE = 35;
```

NOTE

1. Constant identifiers are, by convention, capitalized.
2. A `final` variable can be declared without initializing it immediately. For example,

```
final double TAX_RATE;
if (< some condition >)
    TAX_RATE = 0.08;
else
    TAX_RATE = 0.0;
// TAX_RATE can be given a value just once: its value is final!
```

3. A common use for a constant is as an array bound. For example,

```
final int MAXSTUDENTS = 25;
int[] classList = new int[MAXSTUDENTS];
```

4. Using constants makes it easier to revise code. Just a single change in the `final` declaration need be made, rather than having to change every occurrence of a value.

OPERATORS

Arithmetic Operators

Operator	Meaning	Example
+	addition	`3 + x`
-	subtraction	`p - q`
*	multiplication	`6 * i`
/	division	`10 / 4  //returns 2, not 2.5!`
%	mod (remainder)	`11 % 8  //returns 3`

NOTE

1. These operators can be applied to types `int` and `double`, even if both types occur in the same expression. For an operation involving a `double` and an `int`, the `int` is promoted to `double`, and the result is a `double`.
2. The mod operator %, as in the expression `a % b`, gives the remainder when `a` is divided by `b`. Thus `10 % 3` evaluates to `1`, whereas `4.2 % 2.0` evaluates to `0.2`.
3. Integer division `a/b` where both `a` and `b` are of type `int` returns the integer quotient only (i.e., the answer is truncated). Thus, `22/6` gives `3`, and `3/4` gives `0`. If at least one of the operands is of type `double`, then the operation becomes

Optional topic

Comparing Floating-Point Numbers

Because of round-off errors in floating-point numbers, you can't rely on using the == or != operators to compare two double values for equality. They may differ in their last significant digit or two because of round-off error. Instead, you should test that the magnitude of the difference between the numbers is less than some number about the size of the machine precision. The machine precision is usually denoted ϵ and is typically about 10^{-16} for double precision (i.e., about 16 decimal digits). So you would like to test something like $|x - y| \leq \epsilon$. But this is no good if x and y are very large. For example, suppose $x = 1234567890.123456$ and $y = 1234567890.123457$. These numbers are essentially equal to machine precision, since they differ only in the 16th significant digit. But $|x - y| = 10^{-6}$, not 10^{-16}. So in general you should check the *relative* difference:

$$\frac{|x - y|}{\max(|x|, |y|)} \leq \epsilon$$

To avoid problems with dividing by zero, code this as

$$|x - y| \leq \epsilon \max(|x|, |y|)$$

An example of code that uses a correct comparison of real numbers can be found in the Shape class on p. 146.

Logical Operators

Operator	Meaning	Example
!	NOT	`if (!found)`
&&	AND	`if (x < 3 && y > 4)`
||	OR	`if (age < 2 || height < 4)`

NOTE

1. Logical operators are applied to boolean expressions to form *compound boolean expressions* that evaluate to `true` or `false`.
2. Values of `true` or `false` are assigned according to the truth tables for the logical operators.

&&	T	F
T	T	F
F	F	F

||	T	F
T	T	T
F	T	F

!	
T	F
F	T

For example, F && T evaluates to F, while T || F evaluates to T.

3. *Short-circuit evaluation.* The subexpressions in a compound boolean expression are evaluated from left to right, and evaluation automatically stops as

Operator Precedence

highest precedence	→	(1)	`!, ++, --`
		(2)	`*, /, %`
		(3)	`+, -`
		(4)	`<, >, <=, >=`
		(5)	`==, !=`
		(6)	`&&`
		(7)	`||`
lowest precedence	→	(8)	`=, +=, -=, *=, /=, %=`

Here operators on the same line have equal precedence. The evaluation of the operators with equal precedence is from left to right, except for rows (1) and (8) where the order is right to left. It is easy to remember: The only "backward" order is for the unary operators (row 1) and for the various assignment operators (row 8).

Example

What will be output by the following statement?

```
System.out.println(5 + 3 < 6 - 1);
```

Since + and - have precedence over <, `5 + 3` and `6 - 1` will be evaluated before evaluating the boolean expression. Since the value of the expression is false, the statement will output `false`.

INPUT/OUTPUT

Input

Since there are so many ways to provide input to a program, user input is not a part of the AP Java subset. If reading input is a necessary part of a question on the AP exam, it will be indicated something like this:

```
double x = call to a method that reads a floating-point number
```

or

```
double x = IO.readDouble();     //read user input
```

NOTE

The `Scanner` class (since Java 5.0) simplifies both console and file input. It will not, however, be tested on the AP exam.

Output

Testing of output will be restricted to `System.out.print` and `System.out.println`. Formatted output will not be tested.

`System.out` is an object in the `System` class that allows output to be displayed on the screen. The `println` method outputs an item and then goes to a new line. The `print` method outputs an item without going to a new line afterward. An item to be printed can be a string, or a number, or the value of a boolean expression (`true` or `false`). Here are some examples:

CONTROL STRUCTURES

Control structures are the mechanism by which you make the statements of a program run in a nonsequential order. There are two general types: decision making and iteration.

Decision-Making Control Structures

These include the `if`, `if...else`, and `switch` statements. They are all selection control structures that introduce a decision-making ability into a program. Based on the truth value of a boolean expression, the computer will decide which path to follow. The `switch` statement is not part of the AP Java subset.

THE `if` STATEMENT

```
if (boolean expression)
{
    statements
}
```

Here the *statements* will be executed only if the *boolean expression* is `true`. If it is `false`, control passes immediately to the first statement following the `if` statement.

THE `if...else` STATEMENT

```
if (boolean expression)
{
    statements
}
else
{
    statements
}
```

Here, if the *boolean expression* is `true`, only the *statements* immediately following the test will be executed. If the *boolean expression* is `false`, only the *statements* following the `else` will be executed.

NESTED `if` STATEMENT

If the statement part of an `if` statement is itself an `if` statement, the result is a *nested* `if` *statement.*

Example 1

```
if (boolean expr1)
    if (boolean expr2)
        statement;
```

This is equivalent to

```
if (boolean expr1 && boolean expr2)
    statement;
```

Iteration

Java has three different control structures that allow the computer to perform iterative tasks: the `for` loop, `while` loop, and `do...while` loop. The `do...while` loop is not in the AP Java subset.

THE for LOOP

The general form of the `for` loop is

```
for (initialization; termination condition; update statement)
{
    statements          //body of loop
}
```

The termination condition is tested at the top of the loop; the update statement is performed at the bottom.

Example 1

```
//outputs 1 2 3 4
for (i = 1; i < 5; i++)
   System.out.print(i + " ");
```

Here's how it works. The *loop variable* `i` is initialized to `1`, and the termination condition `i < 5` is evaluated. If it is `true`, the body of the loop is executed, and then the loop variable `i` is incremented according to the update statement. As soon as the termination condition is `false` (i.e., `i >= 5`), control passes to the first statement following the loop.

Example 2

```
//outputs 20 19 18 17 16 15
for (k = 20; k >= 15; k--)
    System.out.print(k + " ");
```

Example 3

```
//outputs 2 4 6 8 10
for (j = 2; j <= 10; j += 2)
    System.out.print(j + " ");
```

NOTE

1. The loop variable should not have its value changed inside the loop body.
2. The initializing and update statements can use any valid constants, variables, or expressions.
3. The scope (see p. 100) of the loop variable can be restricted to the loop body by combining the loop variable declaration with the initialization. For example,

```
for (int i = 0; i < 3; i++)
{
    ...
}
```

4. The following loop is syntactically valid:

NOTE

1. It is possible for the body of a `while` loop never to be executed. This will happen if the test evaluates to `false` the first time.
2. Disaster will strike in the form of an infinite loop if the test can never be false. Don't forget to change the loop variable in the body of the loop in a way that leads to termination!

The body of a `while` loop must contain a statement that leads to termination.

Example 2

```
int power2 = 1;
while (power2 != 20)
{
    System.out.println(power2);
    power2 *= 2;
}
```

Since `power2` will never exactly equal 20, the loop will grind merrily along eventually causing an integer overflow.

Example 3

```
/* Screen out bad data.
 * The loop won't allow execution to continue until a valid
 * integer is entered.
 */
System.out.println("Enter a positive integer from 1 to 100");
int num = IO.readInt();        //read user input
while (num < 1 || num > 100)
{
    System.out.println("Number must be from 1 to 100.");
    System.out.println("Please reenter");
    num = IO.readInt();
}
```

Example 4

```
/* Uses a sentinel to terminate data entered at the keyboard.
 * The sentinel is a value that cannot be part of the data.
 * It signals the end of the list.
 */
final int SENTINEL = -999;
System.out.println("Enter list of positive integers," +
    " end list with " + SENTINEL);
int value = IO.readInt();       //read user input
while (value != SENTINEL)
{
    process the value
    value = IO.readInt();       //read another value
}
```

An *unchecked exception* is one where you don't provide code to deal with the error. Such exceptions are automatically handled by Java's standard exception-handling methods, which terminate execution. You now need to fix your code!

A *checked exception* is one where you provide code to handle the exception, either a `try/catch/finally` statement, or an explicit `throw new ...Exception` clause. These exceptions are not necessarily caused by an error in the code. For example, an unexpected end-of-file could be due to a broken network connection. Checked exceptions are not part of the AP Java subset.

The following exceptions are in the AP Java subset:

Exception	Discussed on page
`ArithmeticException`	on the previous page
`NullPointerException`	103
`ClassCastException`	142
`ArrayIndexOutOfBoundsException`	233
`IndexOutOfBoundsException`	244
`IllegalArgumentException`	this page

See also `NoSuchElementException` (pp. 247, 248) and `IllegalStateException` (pp. 247, 249), which refer to iterators, an optional topic.

Java allows you to write code that throws a standard unchecked exception. Here are typical examples:

Example 1

```
if (numScores == 0)
    throw new ArithmeticException("Cannot divide by zero");
else
    findAverageScore();
```

Example 2

```
public void setRadius(int newRadius)
{
    if (newRadius < 0)
        throw new IllegalArgumentException
                ("Radius cannot be negative");
    else
        radius = newRadius;
}
```

NOTE

1. `throw` and `new` are both reserved words.
2. The error message is optional: The line in Example 1 could have read

   ```
   throw new ArithmeticException();
   ```

   The message, however, is useful, since it tells the person running the program what went wrong.
3. An `IllegalArgumentException` is thrown to indicate that a parameter does not satisfy a method's precondition.