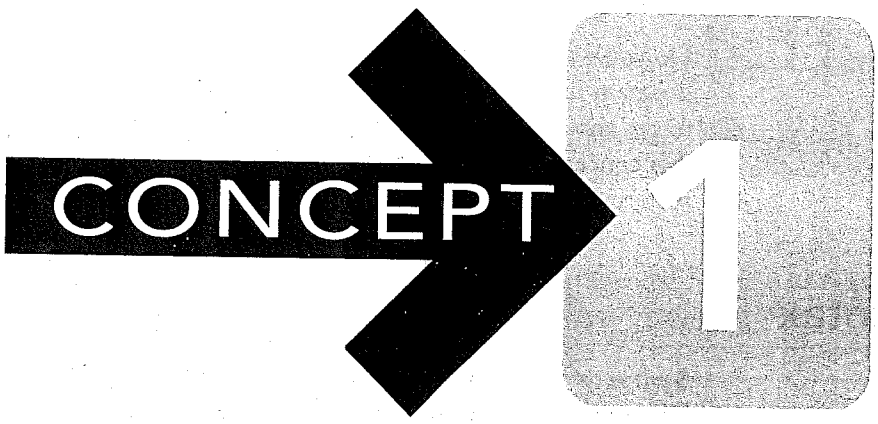


Fundamentals of Programming

IN THIS CONCEPT

Summary: You will learn the fundamental building blocks that all programmers need to write software. The syntax, which is the technical way of writing the code, will be written in Java; however, all of the structures that you will learn in this concept will apply to any language that you come across in the future. To pass the AP Computer Science A Exam, and to be a developer, you must master all of these fundamental concepts.

Key Ideas

- Variables allow you to store information that is used by the program.
- Conditional statements allow you to branch in different directions in your program.
- Looping structures allow you to repeat instructions many times.
- The computer follows the order of operations when performing mathematical calculations.
- Software developers write comments to document their code.
- The console screen is where simple input and output are displayed.
- Good software developers can debug their code and the code of others.

Introduction

Do you want to keep track of a score in a game that you want to write? Do you want your program to make choices based on whatever the user wants? Will you be doing any math to get answers for the user? Do you want an easy way to do something a million times? In this concept you will learn the fundamental building blocks that programmers use to write software to do all these things.

Syntax

Syntax is not a fine you have to pay for doing something you're not supposed to do. The syntax of a programming language describes the correct way to type the code so the program will run. An example of a syntax error is forgetting to put a semicolon after an instruction. Another example is not putting a pair of parentheses in the correct place. You have to fix all syntax errors before your program will run. If your program has any syntax errors, the **compiler** will respond with a **compile-time** error. This type of error prevents the compiler from doing its job of turning your Java code into bytecode. As soon as you have eliminated all syntax errors from your program, your program can be compiled and then **run (executed)**. A list of common syntax errors can be found in the Appendix.

Inline Comment

An inline comment is a way for a programmer to tell someone who is reading the code a secret message. The way to make an inline comment is to make two forward slashes, like this: //.

```
// This is an inline comment.
// The computer ignores everything after the two forward slashes on the same line.
// I will use inline comments as a way to give you secret messages throughout this book.
```

The Console Screen

The **console** screen is the simplest way to input and output information when running a program. The two most common instructions to display information to the screen are **System.out.println()** and **System.out.print()**.

Summary of Console Output

CASE 1: To display some text and move the cursor onto the next line:

```
System.out.println("text goes here");
```

CASE 2: To display some text and NOT move the cursor onto the next line:

```
System.out.print("text goes here");
```

CASE 3: To only move the cursor onto the next line:

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

Example

The difference between the print() and the println() statements:

Predict the Output of This Code:

```
line 1: System.out.print("Players gonna play, ");
line 2: System.out.println("play, play, play, play");
line 3: System.out.print("Haters gonna hate, hate, hate, ");
line 4: System.out.println("hate, hate");
line 5: System.out.println();
line 6: System.out.println("I shake it off, I shake it off");
```

Output on the Console Screen:

```
Players gonna play, play, play, play, play
Haters gonna hate, hate, hate, hate, hate

I shake it off, I shake it off
```

The Semicolon

The semicolon is a special character that signifies the end of an instruction. Every distinct line of code should end with a semicolon. It tells the compiler that an instruction ends here.

```
System.out.println("some text");  // the semicolon separates instructions
```

Primitive Variables

Numbers can be stored and retrieved while a program is running if they are given a home. The way that integers and decimal numbers are stored in the computer is by declaring a **variable**. When you declare a variable, the computer sets aside a space for it in the working memory of the computer (RAM) while the program is running. Once that space is declared, the programmer can assign a value to the variable, change the value, or retrieve the value at any time in the program. In other words, if you want to keep track of something in your program, you have to create a home for it, and declaring a variable does just that.

In Java, the kind of number that the programmer wants to store must be decided ahead of time and is called the **data type**. For the AP Computer Science A Exam, you are required to know two primitive data types that store numbers: the **int** and the **double**.

Variable

A variable is a simple data structure that allows the programmer to store a value in the computer. The programmer can retrieve or modify the variable at any time in the program.

The `int` and `double` Data Types

Numbers that contain decimals can be stored only in the data type called the **double**. Integers can be stored in either the data type called the **int** or the **double**. Remember that this book is designed for the AP Computer Science A Exam. There are several other data types in Java that can store numbers; however, they are not tested on the AP Computer Science A Exam.

When it is time for you to create a variable, you need to know its data type. It's also a common practice to give the variable a starting value if you know what it should be. This process is called *declaring a variable and initializing it.*

The General Form for Declaring a Variable in Java

```
datatype variableName;
```

The General Form for Declaring a Variable and Then Initializing It

```
datatype variableName = value;
```

Examples

Declaring int and double primitive variables in Java:

```
int numberOfPokemon = 25;          // numberOfPokemon gets a value of 25
double health = 112.7;             // health gets the value of 112.7
double gpa = 3;                    // a double can receive an int value
double a = 2.4, b = 5.6, c = 4;    // multiple variables on same line
```

The following graphic is intended to give you a visual of how **primitive variables** are stored in the computer's **RAM (random access memory)**. The value of each variable is stored in the computer's memory at a specific **memory address**. The name of the variable and its value are stored together. Whenever you declare a new primitive variable, think of this picture to imagine what is going on inside the computer. The way this process works inside of a computer is bit more complicated than this, but I hope the diagram helps you imagine how variables are stored in memory.

Memory Address: 0000 numberOfPokemon = 25	*Memory Address: 0001* health = 112.7	*Memory Address: 0002* gpa = 3.0
Memory Address: 0003 a = 2.4	*Memory Address: 0004* b = 5.6	*Memory Address: 0005* c = 4.0
Memory Address: 0006 <*empty*>	*Memory Address: 0007* <*empty*>	*Memory Address: 0008* <*empty*>
Memory Address: 0009 <*empty*>	*Memory Address: 000A* <*empty*>	*Memory Address: 000B* <*empty*>

Naming Convention and Camel Case

The manner in which you choose a variable name should follow the rules of a **naming convention**. A naming convention makes it easy for programmers to recognize and recall variable names.

Obviously, you are familiar with uppercase and lowercase, but let me introduce you to **camel case**, the technique used to create **identifiers** in Java. All primitive variable names start with a lowercase letter; then for each new word in the variable name, the first letter of the new word is assigned an uppercase letter.

All variable names should be descriptive, yet concise. The name should describe exactly what the variable holds and be short enough to not become a pain to write every time you use it. Single letter variable names should be avoided, as they are not descriptive.

Examples

Mistakes when declaring int and double primitive variables in Java:

```
double 5dollars = 900.0        // Error: name cannot begin with a number
int #tickets = 5;              // Error: name cannot start with a # symbol
int theNumberOfPokemonThatAshCaughtDuringHisLifetime = 151;
// the name is valid, but is a terrible choice because it is way too long
```

Summary of Variable Names in Java

- Variable names should be meaningful, descriptive, and concise.
- Variable names cannot begin with a number or symbol (the dollar sign and underscore are exceptions to this rule).
- By naming convention, variable names use camel case.

The `boolean` Data Type

If you want to store a value that is not a number, but rather a **true** or **false** value, then you should choose a **boolean** variable. The boolean data type is stored in the computer memory in the same way as ints and doubles.

Examples

Declaring boolean variables in Java:

```
boolean userHasWon = false;        // userHasWon is false
boolean unicornIsVisible = true;   // unicornIsVisible is true
boolean bananaCount = 5;           // Error: boolean cannot store a number
```

One-Letter Variable Names

On the free-response section of the AP Computer Science A Exam, you should name your variables in a meaningful way so the reader thinks you know what you're doing. Declaring all your variable names with single letters that don't explain what is stored is considered bad programming practice.

Keywords in Java

A **keyword** is a reserved word in Java. They are words that the compiler looks for when translating the Java code into bytecode. Examples of keywords are **int**, **double**, **public**, and **boolean**. Keywords can never be used as the names of variables, as it confuses the compiler and therefore causes a syntax error. A full list of Java keywords appears in the Appendix.

Example

The mistake of using a keyword as a variable name:

```
int public = 6;        // Syntax Error: public is a Java keyword
```

Mathematical Operations

Order of Operations

Java handles all mathematical calculations using the **order of operations** that you learned in math class (your teacher may have called it PEMDAS or GEMA). The order of operations does exactly that: it tells you the order to evaluate any expression that may contain parentheses, multiplication, division, addition, subtraction, and so on. The computer will figure out the answers in exactly the same way that you learned in math class.

Operator	Priority
Parentheses	Top priority
Multiplication, Division, Modulo	Done in order from left to right (equal precedence)
Addition, Subtraction	Done in order from left to right (equal precedence)

Modulo

The **modulus** (or **mod**) operator appears often on the AP Computer Science A Exam. While it is not deliberately taught in most math classes, it is something that you are familiar with.

The mod operator uses the percent symbol, **%**, and produces the remainder after doing a division. Back in grade school, you may have been taught that the answer to 14 divided by 3 was 4 R 2, where the *R* stood for *remainder*. If this is how you were taught, then mod will be simple for you. The result of 14 % 3 is 2 because there are 2 left over after dividing 14 by 3.

Modulo Example	Answer	Explanation
20 % 7	6	20 divided by 7 is 2 with a remainder of 6
100 % 20	0	100 divided by 20 is 5 with a remainder of 0
41 % 12	5	41 divided by 12 is 3 with a remainder of 5
0 % 2	0	0 divided by 2 is 0 with a remainder of 0

Using Modulo to Find Even or Odd Numbers

The mod operator is great for determining if a number is even or odd. If someNumber % 2 = 0, then someNumber is even. Or, if someNumber % 2 = 1, then someNumber is odd.

Furthermore, the mod operator can be used to find a multiple of *any* number. For example, if someNumber % 13 = 0, then someNumber is a multiple of 13.

Division with Integers

When you divide an int by an int in Java, the result is an int. This is referred to as **integer division**. The decimal portion of the answer is ignored. The technical way to say it is that the result is **truncated**, which means that the decimal portion of the answer is dropped.

Arithmetic Operator	Java Symbol	Example	Result	Justification
Addition	+	7 + 3	10	Simple addition
Subtraction	−	7 − 3	4	Simple subtraction
Multiplication	*	7 * 3	21	Simple multiplication
Division	/	7 / 3	2	7 divided by 3 is 2 (remainder is dropped)
Modulo	%	7 % 3	1	7 divided by 3 is 2 with a remainder of 1

Examples

The examples show how division works with int and double values in Java. If either number is a double, you get a double result.

- A double divided by a double is a double. // example: 10.0 / 4.0 = 2.5
- A double divided by an int is a double. // example: 10.0 / 4 = 2.5
- An int divided by a double is a double. // example: 10 / 4.0 = 2.5
- An int divided by an int is an int. // example: 10 / 4 = 2

`ArithmeticException`

Any attempt to divide by zero or perform someNumber % 0 will produce a **run-time error** called an **ArithmeticException**. The program crashes when it tries to do this math operation.

Examples

Getting an ArithmeticException error by doing incorrect math:

```
int num1 = 5, num2 = 0;
int num3 = num1 / num2;          // ArithmeticException: / by zero
int num4 = num1 % num2;          // ArithmeticException: / by zero
```

`ArithmeticException`

Dividing by zero or performing a mod by zero will produce an error called an **ArithmeticException**.

Modifying Number Variables

The Assignment Operator

The equal sign, =, is called an **assignment operator** because you use it when you want to assign a value to a variable. It acts differently from the equal sign in mathematics since the equal sign in math shows that the two sides have the same value. The assignment operator gives the left side the value of whatever is on the right side. The right side of the equal sign is computed first and then the answer is assigned to the variable on the left side.

Assignment statements use the **equal sign**, =, and are processed from RIGHT to LEFT. The RIGHT side is computed first; then the answer is stored in the variable on the LEFT.

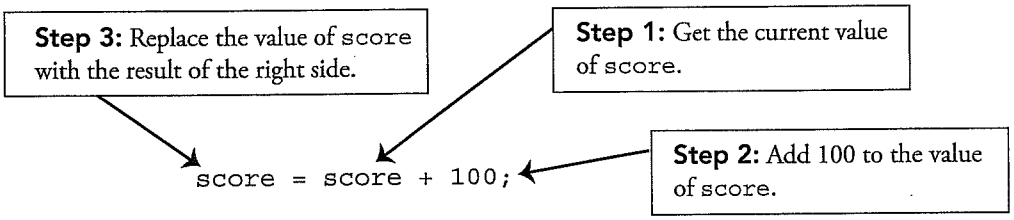
Accumulating

I'm pretty sure that if you were writing some kind of game, you would want to keep track of a score. You may also want a timer. In order to do this, you will need to know how to count and accumulate. When giving a variable a value, the right side of the assignment statement is computed first, and the result is given to the variable on the left side of the statement (even if it is the same variable!). The left side of the assignment statement is replaced by the right side. So, to modify a number variable, you must change it and then store it back on itself.

Example

How to accumulate in Java using the assignment operator:

```
int score = 0;        // create a score variable and initialize it to zero
score = score + 100; // add 100 to the variable score
```



Short-Cut Operators

Java provides another way to modify the value of a variable called a **short-cut operator**. There is a short-cut operator for addition, subtraction, multiplication, division, and modulo. It's called a short-cut operator because you only have to write the variable name one time rather than twice like in the previous explanation.

Examples

How to modify a variable using short-cut operators:

```
int score = 90;
score += 10;          // adds 10 to score: score is now 100
score -= 25;          // subtracts 25 from score: score is now 75
score *= 10;          // multiplies score by 10: score is now 750
score /= 5;           // divides score by 5: score is now 150
score %= 3;           // score mod 3: score is now 0
```

Incrementing and Decrementing a Variable

To **increment** a variable means to *add* to the value stored in the variable. This is how we can *count up* in Java. There are three common ways to write code to add *one* to a variable. You may see any of these techniques for incrementing by one on the AP Computer Science A Exam.

```
i++;                    // increments i by one
i = i + 1;              // increments i by one
i+=1;                   // increments i by one
```

To **decrement** a variable means to *subtract* from its value. This is how we can *count down* in Java. There are three common ways to subtract *one* from a variable in Java.

```
i--;                    // decrements i by one
i = i - 1;              // decrements i by one
i-=1;                   // decrements i by one
```

Casting a Variable

On the AP Computer Science A Exam, you will be tested on the correct way to **cast** variables. Casting is a way to tell a variable to temporarily become a different data type for the sake of performing some action, such as division. The next example demonstrates why casting is important using division with integers. Observe how the cast uses parentheses.

The Most Common Type of Cast

The most common types of casts are from an int to a double or a double to an int.

Example 1

Here is the *correct way* to cast an int to a double: Make the variable pretend that it is a double *before* doing the division.

Hey hits, I know you are an int, but could you just temporarily pretend that you are a double so that I can give this softball player her correct batting average? If you don't, her average will be 0.0.

```
int atBats = 5;
int hits = 2;
double battingAverage = (double)hits / atBats;      // battingAverage is 0.4
```

Example 2

Here is the *wrong way* to cast an int to a double. This example does not produce the result we want, because we are casting the *result* of the division between the integers. Remember that 2 divided by 5 is 0 (using integer division).

```
int atBats = 5;
int hits = 2;
double battingAverage = (double)(hits / atBats); // battingAverage is 0.0
```

Manually Rounding a `double`

On the AP Computer Science A Exam, you will have to know how to round a double to its nearest whole number. This requires a cast from a double to an int.

Example 1

Manually rounding a positive decimal number to the nearest integer:

```
double roundMe = 53.6;
int result = (int)(roundMe + 0.5);                  // result is 54
```

Example 2

Manually rounding a negative decimal number to the nearest integer:

```
double roundMe = -53.6;
int result = (int)(roundMe - 0.5);                    // result is -54
```

Relational Operators

Double Equals Versus Single Equals

Java compares numbers the same way that we do in math class, using **relational operators** (like greater than or less than). A **condition** is a comparison of two values using a relational operator. To decide if two numbers are equal in Java, we compare them using the **double equals**, ==. This is different from the **single equals**, =, which is called an **assignment operator** and is used for assigning values to a variable.

The not Operator: !

To compare if two numbers are **not equal** in Java, we use the **not operator**. It uses the exclamation point, **!**, and represents the opposite of something. It is also referred to as a **negation** operator.

Table of Relational Operators

The following table shows the symbols for comparing numbers using relational operators.

Comparison	Java Symbol
Greater than	>
Less than	<
Greater than or equal to	>=
Less than or equal to	<=
Equal to	==
Not equal to	!=

Examples

Demonstrating relational operators within conditional statements:

```
int a = 6;
System.out.println(a == 10);  // false is printed;  6 is not equal to 10
System.out.println(a < 10);   // true is printed;  6 is less than 10
System.out.println(a >= 10);  // false is printed;  6 is less than 10
System.out.println(a != 10);  // true is printed;  6 is not equal to 10
```

The ! as a Toggle Switch

The not operator, !, can be used in a clever way to reverse the value of a boolean.

Example

Using the not operator to toggle player turns for a two-player game:

```
boolean playerOneTurn = true;          // playerOneTurn is true
playerOneTurn = !playerOneTurn;        // playerOneTurn is false
playerOneTurn = !playerOneTurn;        // playerOneTurn is true
```

Logical Operations

Compound Conditionals: Using AND and OR

AND and **OR** are **logical operators**. In Java, the AND operator is typed using two ampersands (**&&**) while the OR operator uses two vertical bars (||). These two logical operators are used to form **compound conditional statements**.

Logical Operator	Java Symbol	Compound Conditional	Result with Explanation
AND	&&	*condition1* && *condition2*	True only if both conditions are true. False if either condition is false.
OR	||	*condition1* || *condition2*	True if either condition is true. False only if both conditions are false.

The Truth Table

A **truth table** describes how AND and OR work.

First Operand (A)	Second Operand (B)	A && B	A || B
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

Example 1

Suppose you are writing a game that allows the user to play as long as their score is less than the winning score and they still have time left on the clock. You would want to allow them to play the game as long as *both* of these two conditions are true.

```
boolean continuePlaying = (score < winningScore && timeLeft > 0);
```

Example 2

Computing the results of compound conditionals:

```
int a = 3, b = 4, c = 5;
System.out.println(a <= 3 && b != 4);          // false is printed
System.out.println(b % 2 == 1 || c / 2 == 1);  // false is printed
System.out.println(a > 2 && (b > 5 || c < 6)); // true is printed
```

Explanation: `(3 <= 3) && (4 != 4)` → `true && false` → `false`
Explanation: `(4 % 2 == 1) || (5 / 2 == 1)` → `false || false` → `false`
Explanation: `(3 > 2) && (4 > 5 || 5 < 6)` → `true && (false || true)` → `true && true` → `true`

Fun Fact: *The boolean variable is named in honor of George Boole who founded the field of algebraic logic. His work is at the heart of all computing.*

Short-Circuit Evaluation

Java uses **short-circuit evaluation** to speed up the evaluation of compound conditionals. As soon as the result of the final evaluation is known, then the result is returned and the remaining evaluations are not even performed.

Find the result of: `(1 < 3 || (a >= c - b % a + (b + a / 7) - (a % b +c)))`

In a *split second,* you can determine that the result is true *because 1 is less than 3.* End of story. The rest is never even evaluated. Short-circuit evaluation is useful when the second half of a compound conditional might cause an error. This is tested on the AP Computer Science A Exam.

Example

Demonstrate how short-circuit evaluation can prevent a division by zero error.

If the count is equal to zero, the second half of the compound conditional is never evaluated, thereby avoiding a division by zero error.

```
int count = 0;
int total = 0;
boolean result = (count != 0 && total/count > 0);  // result is false
```

DeMorgan's Law

On the AP Computer Science A Exam, you must be able to evaluate complex compound conditionals. **DeMorgan's Law** can help you decipher ones that fit certain criteria.

Compound Conditional	Applying DeMorgan's Law
`!(a && b)`	`!a || !b`
`!(a || b)`	`!a && !b`

An easy way to remember how to apply DeMorgan's Law is to think of the distributive property from algebra, but with a twist. Distribute the ! to both of the conditionals and also change the logical operator to its opposite. Note the use of the parentheses and remember that the law can be applied both forward and backward.

Example

Computing the results of complex logical expressions using DeMorgan's Law:

```
int a = 2, b = 3;
boolean result1 = !(b == 3 && a < 1);      // result1 is true
boolean result2 = !(a != 2 || b <= 4);     // result2 is false
```

Explanation for `result1`: DeMorgan's Law says that you should *distribute* the negation operator. Therefore, the negation of b == 3 is b != 3 and the negation of a < 1 is a >= 1:

```
!(b == 3 && a < 1) → (b != 3) || (a >= 1) → (3 != 3) || (2 >= 1)
                                          → false || true → true
```

Explanation for `result2`: The negation of a != 2 is a == 2 and the negation of b <= 4 is b > 4:

```
!(a != 2 || b <= 4) → (a == 2) && (b > 4) → (2 == 2) && (3 > 4)
                                          → true && false → false
```

The Negation of a Relational Operator

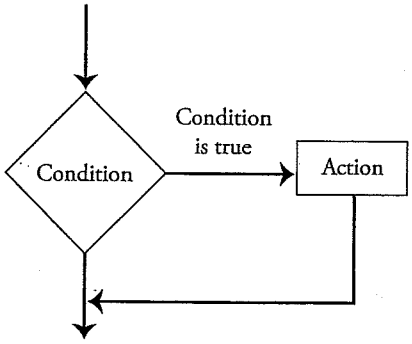
The negation of *greater than* (>) is *less than or equal to* (<=) and vice versa.
The negation of *less than* (<) is *greater than or equal to* (>=) and vice versa.
The negation of *equal to* (==) is *not equal to* (!=) and vice versa.

Conditional Statements

If you want your program to branch out into different directions based on the input from the user or the value of a variable, then you will want to have a **conditional** statement in your program.

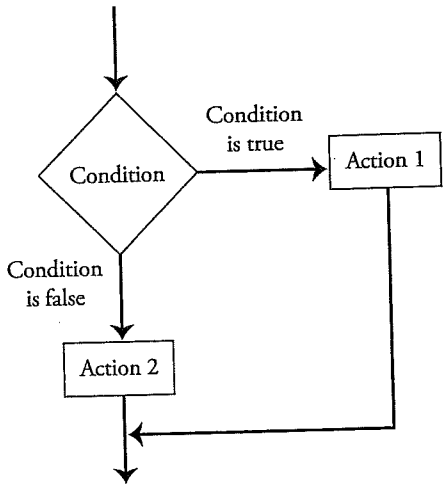
The `if` Statement

The **if** statement is a conditional statement. In its simplest form, it allows the program to execute a specific set of instructions if some certain condition is met. If the condition is not met, then the program skips over those instructions.



The `if-else` Statement

The **if-else** statement allows the program to execute a specific set of instructions if some certain condition is met and a different set of instructions if the condition is not met. I will sometimes refer to the code that is executed when the condition is true as the *if clause* and the code that follows the else as the *else clause*.



The syntax for if and if-else statements can be tricky. Pay attention to the use of the curly braces.

Example 1

The if statement for one or more lines of code:

```
if (condition)
{
   // one or more instructions to be performed when condition is true
}
```

Example 2

The if-else statement for one or more lines of code for each result:

```
if (condition)
{
   // instructions to be performed when condition is true
}
else
{
   // instructions to be performed when condition is false
}
```

Example 3

The if-else statement using a compound conditional:

```
if (condition1 && condition2)
{
   // condition1 and condition2 are both true
}
else
{
   // either condition1 or condition2 is false (or both are false)
}
```

Example 4

Nested if-else statements for one or more lines of code (watch the curly braces!):

```
if (condition1)
{
   // condition1 is true
   if (condition2)
   {
      // condition1 is true and condition2 is true
   }
   else
   {
      // condition1 is true and condition2 is false
   }
}
else
{
   // condition1 is false
   if (condition3)
   {
      // condition1 is false and condition3 is true
   }
   else
   {
      // condition1 is false and condition3 is false
   }
}
```

Example 5

When you want to execute only one line of code when something is true, you don't actually need the curly braces. Java will execute the first line of code after the if statement if the result is true. This is not recommended but it is possible that you will see this on the AP Computer Science A Exam. * Note: In books, authors often leave out the curly braces to save space.

```
if (condition)
   // single instruction to be performed when condition is true
```

Example 6

The same goes for an if-else statement. If you want to execute only one line of code for each value of the condition, then no curly braces are required.

```
if (condition)
   // single instruction to be performed when condition is true
else
   // single instruction to be performed when condition is false
```

Can You Spot the Error in This Program?

```
if (condition);
{
   // instructions to be performed when condition is true
}
```

Answer: Never put a semicolon on the same line as the condition. It ends the if-statement right there. In general, *never put a semicolon before a curly brace.*

The Dangling `else`

If you don't use curly braces properly within if-else statements, you may get a **dangling else**. A dangling else attaches itself to the nearest if statement and not necessarily to the one that you may have intended. Use curly braces to avoid a dangling else and control which instructions you want executed. The dangling else does not cause a compile-time error, but rather it causes a **logic error**. The program doesn't execute in the way that you expected.

Example

The second else statement attaches itself to the nearest preceding if statement:

```
if (condition1)
   // instruction performed when condition1 is true
   if (condition2)
      // instruction performed when condition1 is false and condition2 is true
   else
      // instruction performed when condition1 is false and condition2 is false
```

Scope of a Variable

The word **scope** refers to the code that knows that a variable exists. Local variables, like the ones we have been declaring, are only known within the block of code in which they are defined. This can be confusing for beginning programmers. Another way to say it is: a variable that is declared inside a pair of curly braces is only known inside that set of curly braces.

Curly Braces and Blocks of Code

Curly braces come in pairs and the code they surround is called a **block of code**.

```
{
   // A group of instructions inside a pair of curly braces
   // is called a "block of code". Variables declared inside a block
   // of code are only known inside that block of code.
}
```

Looping Statements

The `for` Loop

Suppose your very strict music teacher catches you chewing gum in class and tells you to write out "I will not chew gum while playing the flute" 100 times. You decide to use your computer to do the work for you. How can you write a program to repeat something as many times as you want? The answer is a **for loop**.

General Form for Creating a `for` Loop

```
for (initialize loop control variable(LCV);condition using LCV; modify LCV)
{
   // instructions to be repeated
}
```

The for loop uses a **loop control variable** to repeat its instructions. This loop control variable is typically an int and is allowed to have a one-letter name like i, j, k, and so on. This breaks the rule of having meaningful variable names. Yes, the loop control variable is a rebel.

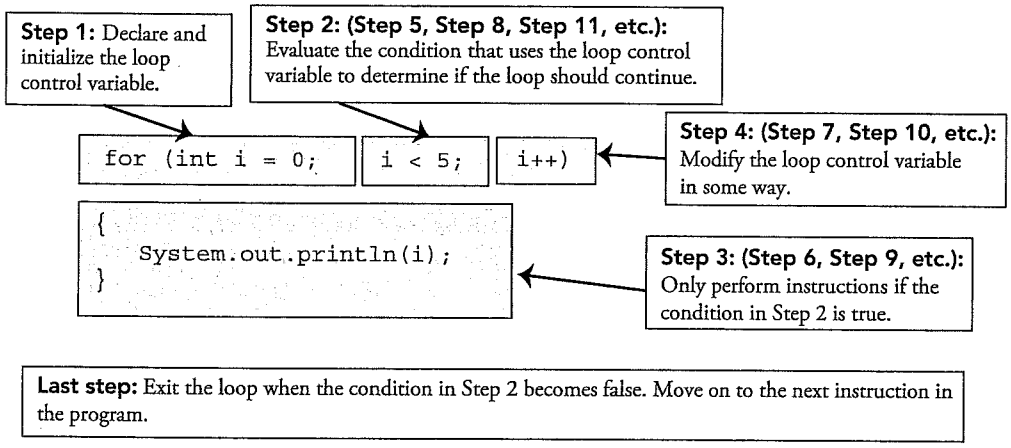
Explanation 1: The `for` loop for those who like to read paragraphs . . .

When a for loop starts, its loop control variable is declared and initialized. Then, a comparison is made using the loop control variable. If the result of the comparison is true, the instructions that are to be repeated are executed one time. The loop control variable is then modified in some way (usually incremented or decremented, but not always). Next, the loop control variable is compared again using the same comparison as before. If the result of this comparison is true, the loop performs another **iteration**. The loop control variable is again modified (in the same way that it was before), and this process continues until the comparison of the loop control variable is false. When this happens, the loop ends and we **exit (or terminate) the loop**. The total number of times that the loop repeats the instructions is called the number of iterations.

Explanation 2: The `for` loop for those who like a list of steps . . .

Step 1: Declare and initialize the loop control variable.
Step 2: Compare the loop control variable in some way.
Step 3: If the result of the comparison is true, then execute the instructions one time.
If the result of the comparison is false, then skip to Step 6.
Step 4: Modify the loop control variable in some way (usually increment or decrement, but not always).
Step 5: Go to Step 2.
Step 6: Exit the loop and move on to the next line of code in the program.

Explanation 3: The `for` loop for those who like pictures . . .



The Loop Control Variable

The most common approach when using a for loop is to declare the loop control variable in the for loop. If this is the case, then the loop control variable is only known inside the for loop. Any attempt to use the loop control variable in an instruction after the loop will get a compile-time error (cannot be resolved to a variable).

However, it is possible to declare a loop control variable prior to the for loop code. If this is the case, then the loop control variable is known before the loop, during the loop, and after the loop.

Example 1

A simple example of a for loop:

```
for (int i = 0; i < 3; i++)
{
    System.out.println("hello");
}
```

```
OUTPUT
hello
hello
hello
```

Example 2

The loop control variable is often used inside the loop as shown here:

```
for (int j = 10; j > 6; j--)
{
    System.out.print(j + "      ");
    System.out.println(10 - j);
}
```

```
OUTPUT
10      0
 9      1
 8      2
 7      3
```

Example 3

In this example, an error occurs when an attempt is made to print the loop control variable after the loop has finished. The variable was declared inside the loop and is not known after the loop exits:

```
for (int k = 1; k <= 4; k++)
{
    System.out.println(k);
}
System.out.println(k*10);
```

```
Compile-time error: k cannot be resolved to a variable
```

Example 4

In this example, the loop control variable is declared in a separate instruction before the loop. This allows the variable to be known before, during, and after the loop:

```
int i;
for (i = 1; i < 5; i++)
{
    System.out.println(i);
}
System.out.println(i);
```

```
OUTPUT
1
2
3
4
5
```

Fun Fact: *In an early programming language called Fortran, the letters i, j, and k were reserved for storing integers. To this day programmers still use these letters for integer variables even though they can be used to store other data types.*

The Nested `for` Loop

A for loop that is inside of another for loop is called a **nested for loop**. The **outer loop control variable** (OLCV) is used to control the outside loop. The **inner loop control variable** (ILCV) is used to control the inside loop. There is no limit to the number of times you can nest a series of for loops.

General Form for a Nested `for` Loop

```
for (initialize OLCV; condition using OLCV; modify OLCV)
{
   for (initialize ILCV; condition using ILCV; modify ILCV)
   {
      // instructions to be repeated
   }
}
```

Execution Count Within a Nested `for` Loop

When a nested for loop is executed, the **inner loop** is performed in its entirety for every iteration of the **outer loop**. That means that the number of times that the instructions inside the inner loop are performed is equal to the product of the number of times the outer loop is performed and the number of times the inner loop is performed.

Example

To demonstrate execution count for a nested for loop: The outer loop repeats a total of three times (once for i = 0, then for i = 1, and then for i = 2). The inner loop repeats four times (once for j = 1, then for j = 2, then for j = 3, and then for j = 4). The inner loop repeats (starts over) for every iteration of the outer loop. Therefore, the System.out.println statement executes a total of 12 times:

```
for (int i = 0; i < 3; i++)                  // the outer loop repeats 3 times
{
   for (int j = 1; j < 5; j++)               // the inner loop repeats 4 times
   {
      System.out.println(i + "  " + j);      // total of 12 iterations (3 * 4)
   }
}
```

```
OUTPUT
0    1
0    2
0    3
0    4
1    1
1    2
1    3
1    4
2    1
2    2
2    3
2    4
```

Execution Count for a Nested `for` Loop

To find the total number of times that the code inside the inner for loop is executed, multiply the number of iterations of the outer loop by the number of iterations of the inner loop.

```
for (int x = 4; x > 0; x--)          // the outer loop repeats 4 times
{
   for (int y = 9; y <= 15; y++)     // the inner loop repeats 7 times
   {
      // some instruction            // total of 28 iterations (4 * 7)
   }
}
```

The `while` Loop

Consider the software that is used to check out customers at a store. The clerk drags each item over the scanner until there aren't any more items to be scanned. *Viewing this through the eyes of a programmer*, I can suggest that a **while loop** is used to determine the final cost of all the items. The process repeats until there aren't any more items. Now, in contrast, if the clerk asked you how many items you had in your cart before he started scanning, then I would suggest that a for loop is being used to determine the final cost of all the items.

General Form for a `while` Loop

Recommended form: Repeat instructions (using curly braces):

```
while (condition)
{
   // one or more instructions to be repeated
}
```

Legal (but not recommended) form: Repeat an instruction (without using curly braces):

```
while (condition)
   // single instruction to be repeated
```

The while loop repeats instructions just like the for loop, but it does it differently. Instead of having a loop control variable that is a number like the for loop, the while loop can use any kind of data type to control when it is to perform another iteration of the loop. I like to use the phrase *as long as* as a replacement for the word *while* when reading a while loop. In other words: *As long as the condition is true, I will continue to repeat the instructions.*

Explanation 1: The `while` loop for those who like to read paragraphs . . .

Declare and initialize some variable of any data type *before* the loop begins. Then compare this variable at the start of the loop. If the result of the comparison is true, the instructions inside the loop get executed one time. Then, modify the variable in some way. Next, compare the variable again in the same way as before. If the result is true, the instructions get executed one more time. This process continues until the result of the comparison is false. At this point, the loop ends and we exit the loop. You must make certain that the variable that is being compared is changed inside of the loop. If you forget to modify this variable, you may end up with a loop that never ends!

Explanation 2: The `while` loop for those who like a list of steps . . .

Step 1: Declare and initialize some variable of any data type.
Step 2: Compare this variable in some way.
Step 3: If the result of the comparison is true, then execute the instructions one time.
If the result of the comparison is false, then skip to Step 6.
Step 4: Modify the variable that is to be compared.
Step 5: Go to Step 2.
Step 6: Exit the loop and move on to the next line of code in the program.

Example 1

A simple example of a while loop:

```
int count = 0;
while (count < 5)
{
   System.out.println(count);
   count++;
}
```

```
OUTPUT
0
1
2
3
4
```

Example 2

Use a while loop to require a user to enter the secret passcode for a vault that contains a million dollars. Do not allow the user to exit the loop until they get it right. If they get it right the first time, do not enter the loop (do not execute any of the instructions inside the loop).

```
System.out.print("Enter the secret passcode: ");
int secretPasscode = /* int provided by user */
while (secretPasscode != 1234)
{
   System.out.println("Sorry that is incorrect. Try again: ");
   secretPasscode = /* int provided by user */
}
System.out.println("Congratulations. You have access to the vault.");
```

```
OUTPUT
Enter the secret passcode: 9999
Sorry that is incorrect. Try again: 8888
Sorry that is incorrect. Try again:  1234
Congratulations. You have access to the vault.
```

A Flag in Programming

A **flag** in programming is a virtual way to signal that something has happened. Think of it as, "I'm waving a flag" to tell you that something important just occurred. Normally, boolean variables are chosen for flags. They are originally set to false, then when something exciting happens, they are set to true.

Example 3

Use a while loop to continue playing a game as long as the game is not over:

```
boolean gameOver = false;
while (!gameOver)
{
   // play the game
   // When a player wins, set gameOver = true
}
```

Strength of the `while` Loop

A strength of the while loop is that you don't have to know how many times you will execute the instructions before you begin looping. You just have to know when you want it to end.

The Infinite Loop

If you make a mistake in a looping statement such that the loop never ends, the result is an **infinite loop.**

Example 1

Accidentally getting an infinite loop when using a for loop:

```
for (int i = 0; i < 10; i--)
{
   i++;
   // i will always revert back to 0 causing an infinite loop
}
```

Example 2

Accidentally getting an infinite loop when using a while loop:

```
int i = 0;
while (i < 10)
{
   // do something forever because you forgot to increment i
}
```

Boundary Testing

A very common error when using a for loop or a while loop is called a **boundary error.** It means that you accidently went one number past or one number short of the right amount. *The loop didn't perform the precise number of iterations.* This concept is tested many times on the AP Computer Science A Exam. The only real way to prevent it is to hand-trace your code very carefully to make sure that your loop is executing the correct number of times.

Goldilocks and the Three Looping Structures

Always test that your for loops, nested for loops and while loops are executing the correct number of times—not too many, but not too few. Just the right number of times. Hint: If you use:

```
for (int i = 0; i < maximum; i++)
```

then the loop will execute maximum number of times.

Bases Other Than Decimal

Binary, Octal, and Hexadecimal

As humans, we are familiar with base 10, or **decimal notation**. However, computers don't use base 10, they use **base 2**, **base 8**, or **base 16**. The AP Computer Science A Exam requires that you understand how to convert between these different number base systems and also understand the underpinnings of **place value** systems other than decimal.

System	Base	Digits	Number of Digits
Decimal	10	0,1,2,3,4,5,6,7,8,9	10
Binary	2	0,1	2
Octal	8	0,1,2,3,4,5,6,7	8
Hexadecimal	16	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F	16

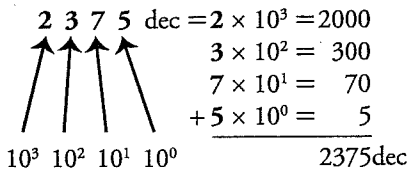
Each of these number systems uses place value and is controlled by its base. Think back to when you were in grade school learning about really big numbers. The number 2375 in decimal means two thousands, three hundreds, seven tens, and five ones. This is because each of the digits resides in a place that represents a power of 10. This system is called *place value*, because, yes, you guessed it, each place has a value.

Notice that we ran out of digits in hexadecimal and started to use the first six letters of the English alphabet. This is the secret code of the hexadecimal. Just note that A = 10, B = 11, C = 12, D = 13, E = 14, and F = 15.

Fun Fact: *The word* **bit** *comes from the words* binary digit. *A bit is represented by either a 1 or a 0, which represents on or off. The symbol for the power button is actually a combination of a 1 and a 0.*

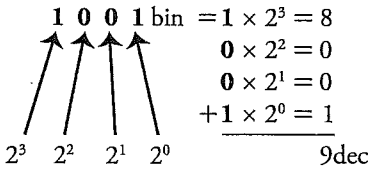
Decimal Numbers (Base 10)

This graphic explains what a number in base 10 means. Now, let's apply this same idea to convert numbers from binary, octal, and hexadecimal to base 10.



Example 1

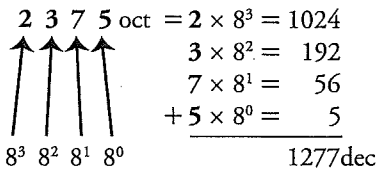
Converting a binary (base 2) number to decimal:



Therefore, $1001_{bin} = 9_{dec}$

Example 2

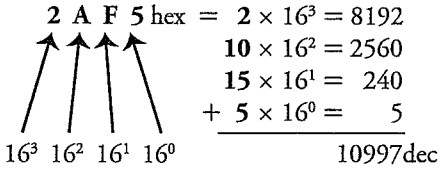
Converting an octal (base 8) number to decimal:



Therefore, $2375_{oct} = 1277_{dec}$

Example 3

Converting a hexadecimal (base 16) number to decimal (recall: A = 10 and F = 15):



Therefore, $2AF5_{hex} = 10997_{dec}$

Question: Why do computer scientists confuse Halloween with Christmas?

Answer: Because $31_{OCT} = 25_{DEC}$

Commenting Your Code

Inline Comments

I've been using inline comments throughout this concept and will continue to use them in the rest of the book. They begin with two forward slashes, //. All text that comes after the slashes (on the same line) is ignored by the compiler.

```
// This is an example of an inline comment.
```

Multiple-Line Comments

When your comment takes longer than one line, you will want to use a multiple-line comment. These comments begin with /* and end with */. The compiler ignores everything in between these two special character sequences.

```
/*
  This is how to write a multiple-line comment. Everything typed here is
  ignored by the compiler; even mi mysteaks is speling, so be particularly
  careful of your spelling when you are typing multiple-line comments!
*/
```

Javadoc Comments

You will see **Javadoc comments** (also called **documentation comments**) on the AP Computer Science A Exam, especially in the FRQ section. These comments begin with /** and end with */. The compiler ignores everything in between the two character sequences.

```
/**
 *  This is an example of a Javadoc comment.
 *  You will see these "documentation" comments on the AP CS exam.
 */
```

Types of Errors

Compile-Time Errors

When you don't use the correct **syntax** in Java, the compiler yells at you. Well, it doesn't actually yell at you, it just won't compile your program and gives you a **compile-time error**. A list of the most common compile-time errors is located in the Appendix.

A brief list of compile-time errors:

- Forgetting a semicolon at the end of an instruction
- Forgetting to put a data type for a variable
- Using a keyword as a variable name
- Forgetting to initialize a variable
- Forgetting a curly brace (a curly brace doesn't have a partner)

Run-Time Exceptions

If your program crashes while it is running, then you have a **run-time error**. The compiler will display the error and it may have the word **exception** in it.

These are the run-time errors that you are required to understand on the AP Computer Science A Exam:

- ArithmeticException (explained in this Concept)
- NullPointerException (explained in Concept 3: The String Class)
- IndexOutOfBoundsException (explained in Concept 5: Data Structures)
- ArrayIndexOutOfBoundsException (explained in Concept 5: Data Structures)
- IllegalArgumentException (explained in Concept 7: Classes and Objects)

Logic Errors

When your program compiles and runs without crashing, but it doesn't do what you expected it to do, then you have a **logic error**. A logic error is the most challenging type of error to fix because you have to figure out where the problem is in your program. Is your math correct? Are your if-statements comparing correctly? Does your loop actually do what it's supposed to do? Are any statements out of order or are you missing something? Logic errors require you to read your code very carefully to determine the source of the error. My advice is to help other people fix their errors so you can ask for help from them when you need it.

Logic Errors on the AP Computer Science A Exam

You will have to analyze code in the multiple-choice section of the exam and find hidden logic errors.

Debugging

The process of removing the errors in your program (compile-time, run-time, and logic) is called **debugging** your program. On the AP Computer Science A Exam, you will be asked to find errors in code.

Fun Fact: *Grace Murray Hopper documented the first actual computer bug on September 9, 1947. It was a moth that got caught in Relay #70 in Panel F of the Harvard Mark II computer.*

`System.out.println` as a Debugging Device

A common way to debug a computer program is to peek inside the computer while it is running and display the current values of variables on the console screen. We aren't actually opening up the computer. We are just displaying the current values of the important variables. By printing the values of the variables at precise moments, you can determine what is going on during the running of the program and hopefully figure out the error.

Example

Figure out what this loop is doing by writing an output statement to the console screen. Printing the values of a, b, and i can help you figure out what the program is doing.

```
int a = 1;
for (int i = 1; i < 10; i++)
{
   a += i;
   int b = a % i;
   System.out.println(i + "  " + a + "  " + b);       // Debugging trick
}
```

```
OUTPUT
1   2   0
2   4   0
3   7   1
4   11   3
5   16   1
6   22   4
7   29   1
8   37   5
9   46   1
```

› Rapid Review

Variables

- Variables store data and must be declared with a data type.
- The int, double, and boolean types are called primitive data types.
- Variables may be initialized with a value or be assigned one later in the program.
- Variable names may begin with a letter, dollar sign, or underscore. They cannot begin with a number.
- Camel case is used for all variable names starting with a lowercase letter.
- Choose meaningful names for variables.
- A keyword is a word that has special meaning to the compiler such as *while* or *public*.
- Keywords cannot be used as variable names.
- The int data type is used to store integer data.
- The double data type is used to store decimal data.
- To cast a variable means to temporarily change its data type.
- The most common type of cast is to cast an int to a double.
- The boolean data type is used to store either a true or false value.

Math and Logic

- The arithmetic operators are +, −, *, /, and %.
- The modulo operator, %, returns the remainder of a division between two integers.
- Java evaluates all mathematical expressions using the order of operations.
- The precedence order for all mathematical calculations is parentheses first, then *, / and % equally from the left to right, then + and − equally from left to right.
- Java uses integer division when dividing an int by an int. The result is a truncated int.
- "Truncating" means dropping (not rounding) the decimal portion of a number.
- The equal sign, =, is called the assignment operator.
- To accumulate means to add (or subtract) a value from a variable.
- Short-cuts for performing mathematical operations are +=, −=, *=, /=, and %=.
- The relational operators in Java are >, >=, <, <=, ==, and !=.
- "To increment" means to add one to the value of a number variable.
- "To decrement" means to subtract one from the value of a number variable.
- A condition is an expression that evaluates to either true or false.
- The logical operator AND is coded using two ampersands, &&.
- The && is true only when both conditions are true.
- The logical operator OR is coded using two vertical bars, ||.

- The || is true when either condition or both are true.
- Software developers use logical operators to write compound conditionals.
- The NOT operator (negation operator) is coded using the exclamation point, !.
- The ! can be used to flip-flop a boolean.
- DeMorgan's Law states: !(A && B) = !A || !B and also !(A || B) = !A && !B.
- When evaluating conditionals, the computer uses short-circuit evaluation.

Programming Statements

- The if and the if-else are called conditional statements since the flow of the program changes based upon the evaluation of a condition.
- Curly braces come in pairs and the code they contain is called a block of code.
- Curly braces need to be used when more than one instruction is to be executed for if, if-else, for, and while statements.
- The scope of a variable refers to the code that knows that the variable exists.
- Variables declared within a conditional are only known within that conditional.
- The for loop is used to repeat one or more instructions a specific number of times.
- The for loop is a good choice when you know exactly how many times you want to repeat a set of instructions.
- The for loop uses a numeric loop control variable that is compared and also modified during the execution of the loop. When the comparison that includes this variable evaluates to false, the loop exits.
- Variables declared within a for loop are only known within that for loop.
- A nested for loop is a for loop inside of a for loop.
- The number of iterations of a nested for loop is the product of the number of iterations for each loop.
- The while loop does not require a numeric loop control variable.
- The while statement must contain a conditional that compares a variable that is modified inside the loop.
- Choose a while loop when you don't know how many times you need to repeat a set of instructions but know that you want to stop when some condition is met.
- An infinite loop is a loop that never ends.
- Forgetting to modify the loop control variable or having a condition that will never be satisfied are typical ways to cause infinite loops.
- Variables declared within a while loop are only known within the while loop.
- A boundary error occurs when you don't execute a loop the correct number of times.

Miscellaneous

- The System.out.println() statement displays information to the console and moves the cursor to the next line.
- The System.out.print() statement displays information to the console and does not move the cursor to the next line.
- Computer scientists use bases other than decimal such as binary, octal, and hexadecimal.
- Programs are documented using comments. There are three different types of comments: inline, multiple line, and Javadoc.
- There are three main types of errors: compile-time, run-time, and logic.
- Compile-time errors are caused by syntax errors.
- *Exception* is another name for *error*.
- There are many types of run-time exceptions and they are based on the type of error.

- Logic errors are difficult to fix because you have to read the code very carefully to find out what is going wrong.
- Debugging is the process of removing errors from a program.
- Printing variables to the console screen can help you debug your program.