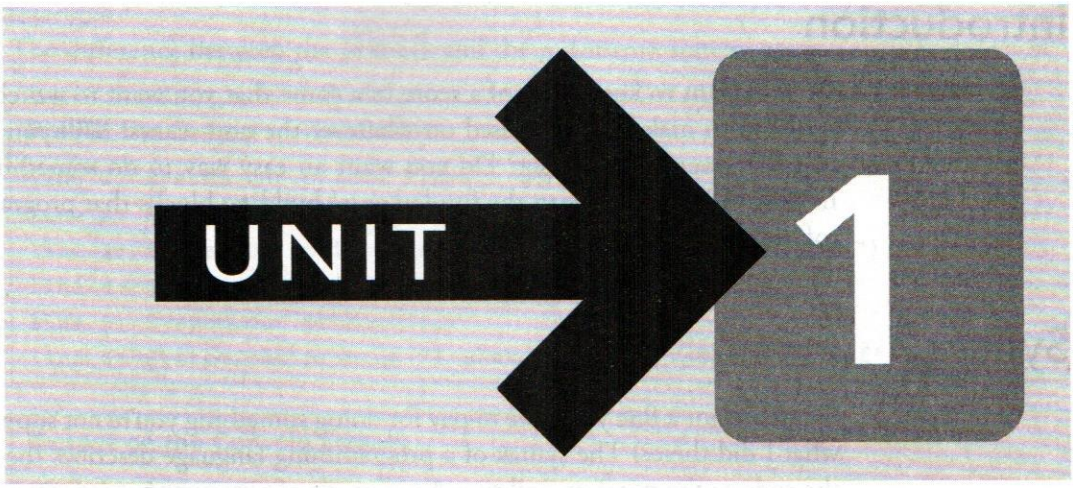


Primitive Types

IN THIS UNIT

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, you must master all of these fundamental concepts.

Key Ideas

- Variables allow you to store information that is used by the program.
- The computer follows the order of operations when performing mathematical calculations.
- 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 unit 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. (Ha, see what I did there?) 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()**. A **string literal** is a sequence of characters enclosed in double quotation marks, such as "text goes here". You must enclose the contents of the string literal in quotation marks so the compiler knows you want to *literally* print the string "text goes here".

Summary of Console Output

CASE 1: To display some text and then 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 ends the instruction
```

Primitive Variables

Numbers can be stored and retrieved while a program is running if they are given a home. The way that integer 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 **double**. Integers can be stored in either the data type called **int** or **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 a 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
```

Constants

A constant is a name for a variable whose value doesn't change during the run of a program. Any attempt to change the value of the constant will generate a compiler error. The Java naming convention is to capitalize each letter of the constant name and to include underscores ("_") when two or more words are combined. This is known as snake case.

The General Form for Declaring a Constant

final datatype VARIABLE_NAME = value;

Examples

Declaring constants in Java:

```
final int INCHES_IN_FOOT = 12;    // INCHES_IN_FOOT gets the value of 12 and
                                  // will remain 12
final double TAX_RATE = 0.06;     // TAX_RATE gets the value of 0.06 and will
                                  // remain 0.06
```

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.
- By naming convention, constant names use snake 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 `int` and `double`.

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 can understand 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
```

Primitive Data Types

- int, double, boolean
- memory associated with a variable holds its actual value

Mathematical Operations

Order of Operations

Java handles all mathematical calculations using the **order of operations** that you were taught in math class (your teacher may have called it PEMDAS). 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 were taught 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: division by zero
int num4 = num1 % num2;      // ArithmeticException: division 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.

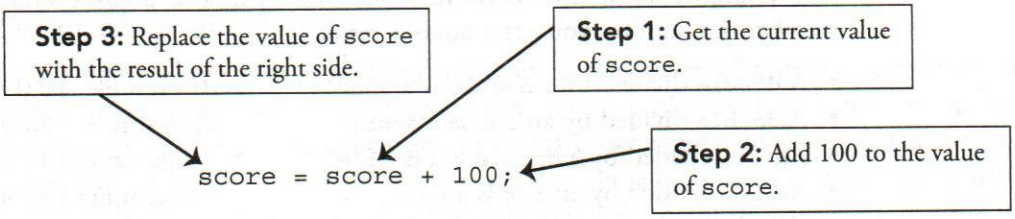
Accumulating

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 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
```



Compound Assignment Operators

Java provides another way to modify the value of a variable called a **compound assignment operator**. There is a compound assignment operator for addition, subtraction, multiplication, division, and modulo. It's called a compound assignment 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 compound assignment 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
```

Arithmetic Overflow

In Java and most other programming languages, integers have a limited range of values. An integer takes up 4 bytes of memory and has a range from `Integer.MIN_VALUE` to `Integer.MAX_VALUE` inclusive. If you attempt to do an assignment or calculation outside of these bounds, arithmetic overflow will occur. The program will still run, but unexpected results will occur.

Example

```
int max = Integer.MAX_VALUE;     // assigns the largest integer that can be stored:
                                 // 2,147,483,647
max++;                           // since this value exceeds the largest value, over-
                                 // flow occurs
System.out.println(max);         // doesn't cause an error, but an unexpected value
                                 // is printed (-2,147,483,648)
```

Fun Fact: *The value of Integer.MAX_VALUE represents the maximum positive value for a 32-bit signed binary number, which is 2^{31}-1. In 2014 Psy's music video "Gangnam Style" was on the verge of exceeding the 32-bit integer limit for YouTube video counts. Engineers saw this coming and updated the system to a 64-bit counter.*

The Correct Way to Compare If Two `double` Values Are Equal

It is never a good idea to compare if two `double` values are equal using the == sign. Calculations with a double can contain rounding errors. You may have seen this when printing the result of some kind of division and you get something like this as a result: `7.00000000002`.

The correct way to compare if two doubles are equal is to use a *tolerance*. If the difference between the two numbers is less than or equal to the tolerance, then we say that the two numbers are *close enough* to be considered equal. Since we might not know which of the two numbers is bigger, subtract the two numbers and find the absolute value of the result. This way, the difference between the two numbers is always positive.

General Form for Comparing If Two `double` Values Are Equal

```
double a =  /*  Some double  */
double b =  /*  Some double  */
double tolerance = 0.00001;  // The 0.00001 varies with the application

boolean closeEnough = Math.abs(a - b) <= tolerance;
```

Example

Determine if two doubles are "equal" using a very small tolerance:

```
double num1 = 3.99999;
double num2 = 4.0;
double tolerance = 0.0001;
boolean result = Math.abs(num1 - num2) <= tolerance;     // result is true
```

Avoid Using the == Sign When Comparing `double` Values

Rounding errors cause problems when trying to determine if two `double` values are equal. Never use == to compare if two doubles are equal. Instead, determine if the difference between the two doubles is close enough for the two doubles to be considered equal.

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 (curly braces must have partners)

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 unit)
- `NullPointerException` (explained in Unit 2: Using Objects)
- `IndexOutOfBoundsException` (explained in Unit 2: Using Objects)
- `ArrayIndexOutOfBoundsException` (explained in Unit 6: Array)
- `ConcurrentModificationException` (explained in Unit 7: ArrayList)

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.

› 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.
- A string literal is enclosed in double quotes.
- 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 `int` or `public`.
- A variable name cannot be the same as a keyword.
- 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.
- When a variable is declared final, its value cannot be changed once it is initialized.

Math

- 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 `%=`.
- "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.
- Avoid comparing double values using the `==` sign. Instead, use a tolerance.

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.
- 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.

› Review Questions

1. Consider the following code segment.

```
int var = 12;
var = var % 7;
var--;
System.out.println(var);
```

What is printed as a result of executing the code segment?

(A) `0`
(B) `1`
(C) `2`
(D) `4`
(E) `5`

2. Consider the following code segment.

```
int count = 5;
double multiplier = 2.5;
int answer = (int)(count * multiplier);
answer = (answer * count) % 10;
System.out.println(answer);
```

What is printed as a result of executing the code segment?

(A) `0`
(B) `2.5`
(C) `6`
(D) `12.5`
(E) `60`

3. Consider the following code segment.

```
double x = 3.6;
double y = 2;
int a = 5, b = 13;
double z = x + y * b / a;
```

What is the value of z after the code segment is executed?

(A) 4.6
(B) 8.6
(C) 8.8
(D) 7.6
(E) 14.56

4. Consider the task of finding the average score in a game.

```
int sum =          // a number representing the sum of all of the scores
int count =        // a number representing the number of games played
```

Which statement will correctly calculate the average score of all the games?

(A) `double average = sum / count;`
(B) `double average = double (sum / count);`
(C) `double average = (double) (sum / count);`
(D) `double average = (double) sum / count;`
(E) `double average = sum / double (count);`

5. Consider the code segment.

```
int a = 16, b = 0;
double c = a/b;
```

The segment compiles but has a run-time error. Which error is generated?

(A) `ArithmeticException`
(B) `DivisionByZeroException`
(C) `IndexOutOfBoundsException`
(D) `NullPointerException`
(E) `TypeMismatchException`