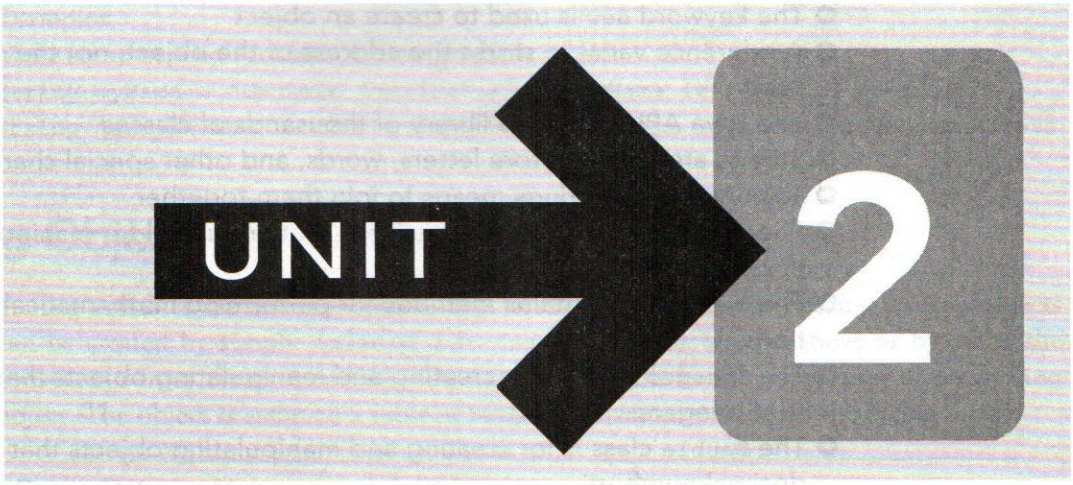


Using Objects

IN THIS UNIT

Summary: This chapter introduces you to the most fundamental object-oriented features of Java. It explains the relationship between an object and a class. You will learn how to use objects created from classes that are already defined in Java. More advanced aspects of classes and objects will be discussed in Unit 5. You will learn how letters, words, and even sentences are stored in variables. `String` objects can store alphanumerical data, and the methods from the `String` class can be used to manipulate this data. The Java API (application programming interface) contains a library of thousands of useful classes that you can use to write all kinds of programs. Because the Java API is huge, the AP Computer Science A Exam only tests your knowledge of a subset of these classes. This unit provides practice working with four of those classes: `String`, `Math`, `Integer`, and `Double`.

Key Ideas

- Java is an object-oriented programming language whose foundation is built on classes and objects.
- A class describes the characteristics of any object that is created from it.
- An object is a virtual entity that is created using a class as a blueprint.
- Objects are created to store and manipulate information in your program.
- A method is an action that an object can perform.

- The keyword `new` is used to create an object.
- A reference variable stores the address of the object, not the object itself.
- The Java API contains a library of thousands of classes.
- Strings are used to store letters, words, and other special characters.
- To concatenate strings means to join them together.
- The `String` class has many methods that are useful for manipulating `String` objects.
- The `Math` class contains methods for performing mathematical calculations.
- The `Integer` class is for creating and manipulating objects that represent integers.
- The `Double` class is for creating and manipulating objects that represent decimal numbers.

The Java API and the AP Computer Science A Exam Subset

Want to make a trivia game that poses random questions? Want to flip a coin 1 million times to test the laws of probability? Want to write a program that uses the quadratic formula to do your math homework? If so, you will want to learn about the Java API.

The Java **API**, or **application programming interface**, describes a set of classes and files for building software in Java. The classes in the APIs and libraries are grouped into packages, such as `java.lang`. The documentation for APIs and libraries are essential to understanding the attributes and behaviors of an object of a class. It shows how all the Java features work and interact. Since the Java API is gigantic, only a subset of the Java API is tested on the AP Computer Science A Exam. All of the classes described in this subset are required. The Appendix contains the entire Java subset that is on the exam. The `String`, `Math`, `Integer`, and `Double` classes are part of the `java.lang` package and are available by default.

The `String` Variable

How does Twitter know when a tweet has more than 280 characters? How does the computer know if your username and password are correct? How are Facebook posts stored? We need a way to store letters, characters, words, or even sentences if we want to make programs that are useful to people. The **String** data type helps solve these problems.

When we wanted to store numbers so that we can retrieve them later, we created either an `int` or `double` variable. Now that we want to store letters or words, we need to create **String variables**. A `String` can be a single character or group of characters like a word or a sentence. **String literals** are characters surrounded by double quotation marks.

General Way to Make a `String` Variable and Assign It a Value

```
String nameOfString = "characters that make up a string literal";
```

Examples

Make some `String` variables and assign them values:

```
String myName = "Captain America";     // myName is "Captain America"
String jennysNumber = "867-5309";      // numbers are treated as characters
```

The `String` Object

`String` variables are actually objects. The `String` class is unique because `String` objects can be created by simply declaring a `String` variable like we did above or by using the constructor from the `String` class. The reference variable knows the address of the `String` object. The object is assigned a value of whatever characters make up the String literal.

General Way to Make a `String` Object Using the Constructor from the `String` Class

```
String nameOfString = new String("characters that make a string");
```

Example 1

Make a `String` object and assign it a value:

```
String myName = new String("Captain America"); // myName is Captain America
```

Example 2

Create a `String` object, but don't give it an initial value (use the empty constructor from the `String` class). The `String` object contains an **empty string**, which means that the value exists (it is not null); however, there are no characters in the String literal and it has a length of 0 characters.

```
String yourName = new String();    // yourName contains an empty string
```

Example 3

Create a String reference variable but don't create a `String` object for it. The String reference variable is assigned a value of **null** because there isn't an address of a `String` object for it to hold.

```
String theirName;    // theirName is null
```

`null` `String` Versus Empty `String`

A **null** string is a string that has no value and no length. It does not hold an address of a `String` object.

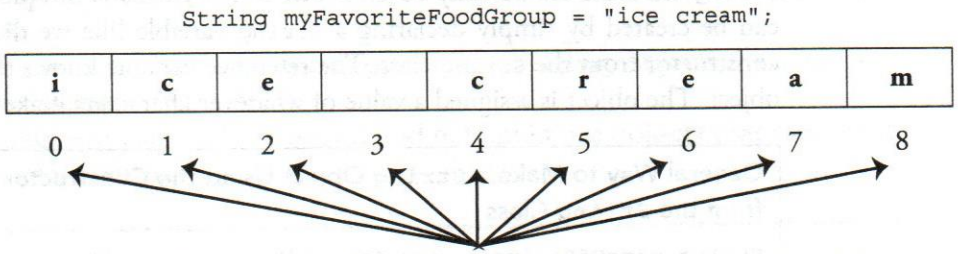
An **empty** string is a string that has a value of "" and its length is zero. There is no space inside of the double quotation marks, because the string is empty.

It may seem like they are the same thing, but they are not. The difference is subtle. The empty string has a value and the null string does not. Furthermore, you can perform a method on an empty string, but *you cannot perform a method on a null string.*

Any attempt to execute a method on a null string will result in a run-time error called a **NullPointerException**.

A Visual Representation of a `String` Object

This is a visual representation of what a string looks like in memory. The numbers below the characters are called the **indices** (plural of **index**) of the string. Each index represents the location of where each of the characters lives within the string. Notice that the first index is zero. An example of how we read this is: the character at index 4 of myFavoriteFoodGroup is the character "c". Also, *spaces* count as characters. You can see that the character at index 3 is the space character.



These numbers represent the index of each character in the `String` object.

`String` Concatenation

Now that you know how to create a string and give it a value, let's make lots of them and string them together (Ha!). The process of joining strings together is called **concatenation**. The + and += signs are used to concatenate strings.

Example 1

Concatenate two strings using +.

```
String firstName = "Harry";
String lastName = "Potter";
String entireName = firstName + lastName;  // entireName is "HarryPotter"
```

Example 2

Concatenate two strings using +=.

```
String firstName = "Harry";
String lastName = "Potter";
lastName += firstName;                     // lastName is "PotterHarry"
```

Example 3

Concatenate a number and a string using +.

```
String firstName = "Harry";
int swords = 8;
String result = firstName + swords;              // result is "Harry8"
```

Example 4

Incorrectly concatenate a string and the sum of two numbers using +. Notice that the sum is not computed since the + operator follows a string variable and is treated as concatenation.

```
String firstName = "Harry";
int swords = 8;
int brooms = 4;
String result = firstName + swords + brooms;       // result is "Harry84"
```

Example 5

Correctly concatenate a string and the sum of two numbers using +. Notice the () are evaluated first which means the + is treated as addition.

```
String firstName = "Harry";
int swords = 8;
int brooms = 4;
String result = firstName + (swords + brooms);      // result is "Harry12"
```

Example 6

Concatenate the sum of two numbers and a string in a different order. Notice that when the numbers come first in the concatenation, the sum is computed.

```
String firstName = "Harry";
int swords = 8;
int brooms = 4;
String result = swords + brooms + firstName;        // result is "12Harry"
```

Concatenation

Strings can be joined together to form a new string using concatenation. The + sign or the += operation may be used to join strings.

You can even join an `int` or a `double` along with a string. These numbers will be rendered useless for any mathematical purpose because they are turned into strings.

The Correct Way to Compare Two `String` Objects

Everyone reading this book has had to log in to a computer at some time. How does the computer know if the user typed the username and password correctly? The answer is that the software uses an operation to compare the input with the information in a database.

In Java, we can determine if two strings are equal to each other. This means that the two strings have exactly the same characters in the same order.

The Correct Way to Compare Two `String` Objects

Use the **equals** method or the **compareTo** method to determine if two `String` variables contain the exact same character sequence.

Important `String` Methods

The `equals` Method

The **equals** method returns a boolean answer of true if the two strings are identical and it returns an answer of false if the two strings differ in any way. The `equals` method is **case-sensitive**, so, for example, the uppercase "A" is different than the lowercase "a". The ! operator can be used in combination with the `equals` method to compare if two strings are not equal.

Example 1

Determine if two strings are equal:

```
String username1 = "Cat Lady";
String username2 = "cat lady";                    // notice the lowercase letters
System.out.println(username1.equals("Cat Lady")); // prints true
System.out.println(username2.equals("Cat Lady")); // prints false
```

Example 2

Determine if two strings are not equal:

```
String pet1 = "Cat";
String pet2 = "Dog";
System.out.println(!pet1.equals(pet2));      // prints true
```

Never Use == to Compare Two `String` Objects

We used the double equals (==) to compare if two numbers were the same. However, == does not correctly tell you if two strings are the same. What's worse is that comparing two strings using == does not throw an error; it simply compares whether or not the two String references point to the same object. Even worse is that sometimes it appears to work correctly. Just don't do it!

```
String username = "CatLady";
String first = "Cat";
String last = "Lady";
if (username == first + last)
{
   // Comparison will be false.
   // NEVER USE == TO COMPARE TWO STRINGS!
}
```

The `compareTo` Method

The second way to compare two strings is to use the **compareTo** method. Instead of returning an answer that is a boolean like the `equals` method did, the `compareTo` method returns an integer. The result of the `compareTo` method is a positive number, a negative number, or zero. This integer describes how the two strings are ordered **lexicographically**; a case-sensitive ordering of words similar to a dictionary but using the **UNICODE** codes of each character.

The `compareTo` Method

The `compareTo` method returns an integer that describes how the two strings compare.

```
int result = firstString.compareTo(secondString);
```

The answer is zero if the two strings are exactly the same.
The answer is negative if firstString comes before secondString (lexicographically).
The answer is positive if firstString comes after secondString (lexicographically).

Example 1

The answer is zero if the two strings are exactly the same:

```
String string1 = "Hello";
String string2 = "Hello";
int result = string1.compareTo(string2);        // result is 0
```

Example 2

The answer is negative when the first string comes before the second string (lexicographically):

```
String string1 = "Hello";
String string2 = "World";
int result = string1.compareTo(string2);        // result is negative
```

Example 3

The answer is positive when the first string comes after the second string (lexicographically):

```
String string1 = "Hello";
String string2 = "World";
int result = string2.compareTo(string1);        // result is positive
```

Example 4

Uppercase letters come before their lowercase counterparts. "Hello" comes before "hello" when comparing them using lexicographical order:

```
String string1 = "Hello";
String string2 = "hello";
int result = string1.compareTo(string2);        // result is negative
```

Fun Fact: *Unicode is a worldwide computing standard that assigns a unique number to nearly every possible character (including symbols) for every language. There are more than 120,000 characters listed in the latest version of Unicode. For example, the uppercase letter "A" is 41 (hexadecimal), the lowercase letter "a" is 61 (hexadecimal), and "ñ" is F1 (hexadecimal).*

The `length` Method

The **length** method returns the number of characters in the specified string (including spaces).

Example 1

Find the length of a string:

```
String tweet = "I'm going to rock this AP CS exam.";
int result = tweet.length();                    // result is 34
```

Example 2

Find the length of a string that is empty, but not null (it has "" as its value):

```
String pokemonCard = new String();
int result = pokemonCard.length();          // result is 0
```

Example 3

Attempt to find the length of a string that is null:

```
String pokemonCard;                    // no String created
int result = pokemonCard.length();     // error: NullPointerException
```

The `NullPointerException`

Attempting to use a method on a string that has not been initialized will result in a run-time error called a `NullPointerException`. The compiler is basically telling you that there isn't an object to work with, so how can it possibly perform any actions with it?

The `indexOf` Method

To find if a string contains a specific character or group of characters, use the **indexOf** method. It searches the string and if it locates what you are looking for, it returns the index of its location. Remember that the first index of a string is zero and the last index is length -1. If the `indexOf` method doesn't find the string that you are looking for, it returns -1.

Example 1

Find the index of a specific character in a string.

```
String sport = "Soccer";
int result = sport.indexOf("e");          // result is 4
```

Example 2

Find the index of a character in a string when it appears more than once.

```
String sport = "Soccer";
int result = sport.indexOf("c");          // result is 2 (finds only
                                          // the first "c")
```

Example 3

Attempt to find the index when the character is not in the string.

```
String sport = "Team";
int result = sport.indexOf("i");          // result is -1
                                          //  there is no "i" in Team (ha)
```

Example 4

Find the index of a string inside a string.

```
String sport = "Soccer";
String findMe = "ce";
int result = sport.indexOf(findMe);       // result is 3 because "ce" begins
                                          // at position 3
```

The `substring` Method

The **substring** method is used to extract a specific character or group of characters from a string. All you have to do is tell the `substring` method where to start extracting and where to stop.

The `substring` method is **overloaded**. This means that there is more than one version of the method.

Overloaded Method	Description
`substring(int index)`	Starts extracting at `index` and stops extracting when it reaches the last character in the string
`substring(int firstIndex, int secondIndex)`	Starts extracting at `firstIndex` and stops extracting at `secondIndex - 1`

Example 1

Extract every character starting at a specified index from a string.

```
String motto = "May the force be with you.";
String result = motto.substring(14);          // result is "be with you."
```

Example 2

Extract one character from a string.

```
String motto = "May the force be with you.";
String result = motto.substring(2,3);         // result is "y"
```

Example 3

Extract two characters from a string.

```
String motto = "May the force be with you.";
String result = motto.substring(5,7);         // result is "he", the letters
                                              // at positions 5 and 6
```

Example 4

Extract the word "force" from a string:

```
String motto = "May the force be with you.";
String result = motto.substring(8,13);     // result is "force"
                                           // the letters at positions 8-12
```

Example 5

Use numbers that don't make any sense:

```
String motto = "May the force be with you.";
String result = motto.substring(72);  //error: StringIndexOutOfBoundsException
```

The `StringIndexOutOfBoundsException`

If you use an index that doesn't make any sense, you will get a run-time error called a **StringIndexOutOfBoundsException**. Using an index that is greater than or equal to the length of the string or an index that is negative will produce this error.

Example 6

Extract the word "the" from the following string by finding and using the first and second spaces in the string:

```
String motto = "May the force be with you.";
int firstSpace = motto.indexOf(" ");
int secondSpace = motto.indexOf(" ",firstSpace + 1);
String word = motto.substring(firstSpace + 1, secondSpace); //word is "the"
```

Helpful Hint for the `substring` Method

To find out how many characters to extract, subtract the first number from the second number.

```
String myString = "I solemnly swear that I am up to no good.";
String result = myString.substring(5,13);       // result is "emnly sw"
```

Since 13 – 5 = 8, the result contains a string that is a total of eight characters.

Note: When the start index and end index are the same number, the `substring` method does not return a character of the string; it returns the empty string `""`.

You will receive a Java Quick Reference sheet to use on the Multiple-choice and Free-response sections, which lists the `String` class methods that may be included in the exam. Make sure you are familiar with it before you take the exam.

Class Constructors and Methods	Explanation
String Class	
`String(String str)`	Constructs a new `String` object that represents the same sequence of characters as `str`
`int length()`	Returns the number of characters in a `String` object
`String substring(int from, int to)`	Returns the substring beginning at index `from` and ending at index `to -1`
`String substring(int from)`	Returns `substring(from, length())`
`int indexOf(String str)`	Returns the index of the first occurrence of `str; returns -1` if not found
`boolean equals(String other)`	Returns `true` if `this` is equal to `other`; returns `false` otherwise
`int compareTo(String other)`	Returns a value <0 if `this` is less than `other`; returns zero if `this` is equal to `other`; returns a value >0 if `this` is greater than `other`

A `String` Is Immutable

An immutable object is an object whose state cannot be modified after it is created. String objects are immutable, meaning that the String methods do not change the String object. `String` variables are actually references to `String` objects, but they don't operate the same way as normal object reference variables.

Every time you make a change to a `String` variable, Java actually tosses out the old object and creates a new one. You are never actually making changes to the original `String` object and *resaving* the same object with the changes. Java makes the changes to the object and then creates a brand-new `String` object and that is what you end up referencing.

Example

Demonstrate how a `String` variable is immutable through concatenation. Even though it seems like you are modifying the `String` variable, you are not. A completely brand-new `String` object is created and the variable is now referencing it:

```
String str = "CS";     // str references a String object "CS"
str = str + " Rules"; // str now references a new String object "CS Rules"
```

Escape Sequences

How would you put a double quotation mark inside of a string? An error occurs if you try because the double quotes are used to define the start and end of a String literal. The solution is to use an **escape sequence**.

Escape sequences are special codes that are embedded in String literals that are sent to the computer that say, "Hey, watch out, something different is coming." They use a backslash followed by a special character. Escape sequences are commonly used when concatenating strings to create a single string. Here are the most common escape sequences (`\t` is not tested on the AP Computer Science A exam but is useful for displaying output in columns).

Purpose	Sequence	Example	`System.out.print(result)`
include a double quote	`\"`	`String result = "\"OMG\"";`	`"OMG"`
tab	`\t`	`String result = "a\tb\tc";`	`a b c`

new line	`\n`	`String result = "a\nb\nc";`	`a`
			`b`
			`c`

backslash	`\\`	`String result = "I am a backslash: \\";`	`I am a backslash: \`

The `Math` Class

The **Math class** was created to help programmers solve problems that require mathematical computations. Most of these methods can be found on a calculator that you would use in math class. In fact, if you think about how you use your calculator, it will make it easier to understand how the `Math` class methods work. The `Math` class also includes a random number generator to help programmers solve problems that require random events.

Compared to the other classes in the AP Computer Science A subset, the `Math` class is special in that it contains only **static** methods. This means that you don't need to create an object from the class in order to use the methods in the class. All you have to do is use the class name followed by the dot operator and then the method name.

The `Math` Class Doesn't Need an Object

The methods and fields of the `Math` class are accessed by using the class name followed by the method name.

```
Math.methodName();
```

Important `Math` Class Methods

The `Math` class contains dozens of awesome methods for doing mathematical work. The AP Computer Science A exam only requires you to know a small handful of them.

Mathematical Operation	Method Name in Java
Absolute Value	`abs`
Square Root	`sqrt`
Base Raised to a Power	`pow`
Random Number Generator	`random`

Example 1

Demonstrate how to use the absolute value method on an integer:

```
int result = Math.abs(-8);              // result is 8
```

Example 2

Demonstrate how to use the absolute value method on a double:

```
double result = Math.abs(-8.4);         // result is 8.4
```

Example 3

Demonstrate how to use the square root method. The sqrt method will compute the square root of any number that is greater than or equal to 0. The result is always a double, even if the input is a perfect square number.

```
double result = Math.sqrt(49);                  // result is 7.0
```

Example 4

Demonstrate how to use the power method. The pow method always returns a double.

```
double result = Math.pow(5,3); // result is 125.0 (5 raised to the 3rd power)
```

Example 5

Demonstrate how to use the random method.

```
double result = Math.random(); // result is a double in the interval [0.0, 1)
```

Example 6

Demonstrate how to access the built-in value of pi from the `Math` class.

```
double result = Math.PI;        // PI is not a method, it is a public field
```

Random Number Generator in the `Math` Class

The random number generator from the `Math` class returns a double value in the range from 0.0 to 1.0, including 0.0, but not including 1.0. Borrowing some notation from math, we say that the method returns a value in the interval [0.0, 1.0).

The word *inclusive* means to include the endpoints of an interval. For example, the interval [3,10] means all of the numbers from 3 to 10, including 3 and 10. Written mathematically, it's the same as $3 \leq x \leq 10$. Finally, you could say you are describing all the numbers between 3 and 10 (inclusive).

Using `Math.random()` to Generate a Random Integer

The `random` method is great for generating a random double, but what if you want to generate a random integer? The secret lies in multiplying the random number by the total number of choices and then casting the result to an integer.

General Form for Generating a Random Integer Between 0 and Some Number

```
int result = (int)(Math.random() * (total # of choices));
```

General Form for Generating a Random Integer Between Two Positive Numbers

```
int high = /*  some number greater than zero  */
int low = /*  some number greater than zero and less than high  */

int result = (int)(Math.random() * (high - low + 1)) + low;
```

Example 1

Generate a random integer between 0 and 12 (inclusive):

```
int result = (int)(Math.random() * 13);          // 13 possible answers
```

Example 2

Generate a random integer between 4 and 20 (inclusive):

```
int result = (int)(Math.random() * 17) + 4;      // 17 possible answers
```

Example 3

Pick a random character from a string. Use the length of the string in the calculation.

```
String quote = "And may the odds be ever in your favor.";
int index = (int)(Math.random() * quote.length());
String result = quote.substring(index, index + 1); //random character
```

Math Class	
`static int abs(int x)`	Returns the absolute value of an `int` value
`static double abs(double x)`	Returns the absolute value of a `double` value
`static double pow(double base, double exponent)`	Returns the value of the first parameter raised to the power of the second parameter
`static double sqrt(double x)`	Returns the positive square root of a `double` value
`static double random()`	Returns a `double` value greater than or equal to 0.0 and less than 1.0

The `Integer` Class

The **Integer** class has the power to turn a primitive `int` into an object. This class is called a **wrapper class** because the class *wraps* the primitive `int` into an object. A major purpose for the `Integer` class will be revealed in Unit 7. Java can convert an `Integer` object back into an `int` using the `intValue` method.

Example 1

Create an `Integer` object from an `int`.

```
int num = 5;
Integer myInteger = new Integer(num); // myInteger is an object, value 5
```

Example 2

Obtain the value of an `Integer` object using the `intValue()` method.

```
Integer myInteger = new Integer(8);    // myInteger is an object, value 8
int result = myInteger.intValue();     // result is a primitive, value 8
```

Example 3

Attempt to create an `Integer` object using a `double` value.

```
double num = 5.67;
Integer myInteger = new Integer(num);    // compile-time error
```

Example 4

Attempt to create an `Integer` object without using a value.

```
Integer myInteger = new Integer();       // compile-time error
```

Fields of the `Integer` Class

MAX_VALUE and **MIN_VALUE** are public **fields** of the `Integer` class. A field is a property of a class (like an instance variable). These two are called **constant** fields and their values cannot be changed during run-time. By naming convention, constants are typed using only uppercase and underscores. Also, they don't have a pair of parentheses, because they are not methods.

Why is there a maximum or minimum integer? Well, Java sets aside 32 bits for every `int`. So, the largest integer that can be stored in 32 bits would look like this: 01111111 11111111 11111111 11111111. Notice that the *leading* bit is 0. That's because the first bit is used to assign the sign (positive or negative value) of the integer. If you do the conversion to decimal, this binary number is equal to 2,147,483,647; the largest integer that can be stored in an `Integer` object. It's also equal to one less than 2 to the 31st power.

If you add one to this maximum number, the result causes an **arithmetic overflow** and the new number is pretty much meaningless. Worse yet, the overflow *does not* cause a run-time error, so your program doesn't crash. Moral of the story: Be careful when you are using extremely large positive or small negative integers.

Example 1

Obtain the maximum value represented by an int or Integer.

```
int result = Integer.MAX_VALUE;      // result is 2147483647 (2^31 - 1)
```

Example 2

Obtain the minimum value represented by an int or Integer.

```
int result = Integer.MIN_VALUE;      // result is -2147483648 (-2^31)
```

`MAX_VALUE` and `MIN_VALUE` Are Constants

A **constant** in Java is a value that is defined by a programmer and cannot be changed during the running of the program. The two values, `MAX_VALUE` and `MIN_VALUE`, are constants of the `Integer` class that were defined by the creators of Java. By naming convention, constants are always typed in uppercase letters and underscores.

Fun Fact: *Java has different data types for storing numbers. The long data type uses 64 bits and the largest number it can store is 9,223,372,036,854,775,807. This is not on the AP Computer Science A exam.*

The `Double` Class

The **Double** class is a wrapper class for primitive doubles. The `Double` class is used to turn a `double` into an object.

Example 1

Create a `Double` object from a `double`.

```
double num = 5.67;
Double myDouble = new Double(num);        // myDouble is an object, value 5.67
```

Example 2

Obtain the value of a `Double` object by using the `doubleValue()` method.

```
Double myDouble = new Double(6.125);
double result = myDouble.doubleValue();   // result is an object, value 6.125
```

Example 3

Create a `Double` object using an `int` value.

```
int num = 9;
Double myDouble = new Double(num);  // myDouble is an object, value is 9.0
```

Example 4

Attempt to create a `Double` object without using a value.

```
Double myDouble = new Double();           // compile-time error
```

Fun Fact: *The Double class also has a MAX_VALUE and MIN_VALUE, but they are not tested on the AP Computer Science A exam.*

Autoboxing and Unboxing

Conversion between primitive types (`int`, `double`, `boolean`) and the corresponding object wrapper class (`Integer`, `Double`, `Boolean`) is performed automatically by the compiler. This process is called "autoboxing." Conversely, "unboxing" is the automatic conversion from the wrapper class to the primitive type. The reason we need to know this conversion will be relevant in Unit 7.

Example 1

Autobox a `Double` object from a `double`.

```
double num = 7.28;
Double myNum = num;          // myNum is an object, value 7.28
```

Example 2

Unbox an `Integer` object to an integer.

```
int value = new Integer(27);            // value is an integer, value 27
```

Example 3

Multiply the value of a `Double` object by 3 and store its value into a `double`.

```
Double myValue = new Double (2.02);
double triple = 3 * myValue;  // automatically unboxes the object into a
                              // primitive
```

Example 4

Multiply the value of a `Double` object by 3 and store its value into a `double` without using the automatic unboxing feature.

```
Double myValue = new Double (2.02);
double triple = 3 * myValue.doubleValue();
```

Summary of the `Integer` and `Double` Classes

Integer Class	
`Integer(int value)`	Constructs a new `Integer` object that represents the specified `int` value
`Integer.MIN_VALUE`	The minimum value represented by an `int` or `Integer`
`Integer.MAX_VALUE`	The maximum value represented by an `int` or `Integer`
`int intValue()`	Returns the value of this `Integer` as an `int`
Double Class	
`Double(double value)`	Constructs a new `Double` object that represents the specified `double` value
`double doubleValue()`	Returns the value of this `Double` as a `double`

You will receive a Java Quick Reference sheet to use on the Multiple-choice and Free-response sections, which lists the `Math`, `Integer`, and `Double` class methods that may be included in the exam. Make sure you are familiar with it before you take the exam.

› Rapid Review

Strings

- The `String` class is used to store letters, numbers, or other symbols.
- A String literal is a sequence of characters contained within double quotation marks.
- You can create a `String` variable by declaring it and directly assigning it a value.
- You can create a `String` variable using the keyword `new` along with the String constructor.
- Strings are objects from the `String` class and can utilize methods from the `String` class.
- Strings can be concatenated by using the + or += operators.
- Compare two `String` variables using the `equals` method or the `compareTo` method.
- Strings should never be compared using the == comparator.
- Every character in a string is assigned an index. The first index is zero. The last index is one less than the length of the string.
- Escape sequences are special codes that are embedded in String literals to perform such tasks as issuing a new line or a tab, or printing a double quote or a backslash.
- A null string does not contain any value. It does not reference any object.
- If a string is null, then no methods can be performed on it.
- A `NullPointerException` is thrown if there is any attempt to perform a method on a null string.

- An empty string is not the same as a null string.
- An empty string has a value of "" and a length of zero.

`String` Methods

- The `equals(String other)` method returns true if the two strings that are being compared contain exactly the same characters.
- The `compareTo(String other)` method compares two strings lexicographically.
- Lexicographical order is a way of ordering words based on their Unicode values.
- The `length()` method returns the number of characters in the string.
- The `indexOf(String str)` method finds and returns the index value of the first occurrence of str within the `String` object that called the method. The value of -1 is returned if str is not found.
- The `substring(int index)` method returns a new `String` object that begins with the character at index and extends to the end of the string.
- The `substring(int beginIndex, int endIndex)` method returns a new `String` object that begins at the `beginIndex` and extends to the character at `endIndex - 1`.
- Strings are immutable. A new `String` object is created each time changes are made to the value of a string.
- A `StringIndexOutOfBoundsException` error is thrown for any attempt to access an index that is not in the bounds of the String literal.

`Math` Class

- The `Math` class has static methods, which means you don't create a `Math` object in order to use the methods or fields from the class.
- The dot operator is used to call the methods from the `Math` class: `Math.methodName()`.
- The `Math.abs(int x)` method returns an `int` that is the absolute value of x.
- The `Math.abs(double x)` method returns a `double` that is the absolute value of x.
- The `Math.sqrt(double x)` method returns a `double` that is the square root of x.
- The `Math.pow(double base, double exponent)` method returns a `double` that is the value of base raised to the power of exponent.
- The `Math.random()` method returns a `double` that is randomly chosen from the interval [0.0, 1.0).
- By casting the result of a `Math.random()` statement, you can generate a random integer.

`Integer` and `Double` Classes

- The `Integer` and `Double` classes are called wrapper classes, since they *wrap* a primitive int or `double` value into an object.
- An `Integer` object contains a single field whose type is `int`.
- The `Integer(int value)` constructor constructs an `Integer` object using the `int` value.
- The `intValue()` method returns an `int` that is the value of the `Integer` object.
- The `Integer.MAX_VALUE` constant is the largest possible `int` in Java.
- The `Integer.MIN_VALUE` constant is the smallest possible `int` in Java.
- The `Double(double value)` constructor constructs a `Double` object using the `double` value.
- The `doubleValue()` method returns a `double` value that is the value of the `Double` object.

› Review Questions

Basic Level

1. The relationship between a class and an object can be described as:
   (A) The terms *class* and *object* mean the same thing.
   (B) A class is a program, while an object is data.
   (C) A class is the blueprint for an object and objects are instantiated from it.
   (D) An object is the blueprint for a class and classes are instantiated from it.
   (E) A class can be written by anyone, but Java provides all objects.

2. Consider the following code segment.

```
String newString = "Welcome";
String anotherString = "Home";
newString = anotherString + "Hello";
System.out.println(newString);
```

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

   (A) `HomeHello`
   (B) `HelloHello`
   (C) `WelcomeHello`
   (D) `Welcome Hello`
   (E) `Home Hello`

3. What is printed as a result of executing the following statement?

```
System.out.println(7 + 8 + (7 + 8) + "Hello" + 7 + 8 + (7 + 8));
```

   (A) `7878Hello7878`
   (B) `7815Hello7815`
   (C) `30Hello30`
   (D) `30Hello7815`
   (E) `7815Hello30`

4. Consider the following code segment.

```
String str1 = "presto chango";
String str2 = "abracadabra";

int i = str1.indexOf("a");
System.out.println(str2.substring(i, i + 1) + i);
```

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

   (A) `p0`
   (B) `r9`
   (C) `r10`
   (D) `ra9`
   (E) `ra10`

5. Consider the following code segment.

```
String s1 = "avocado";
String s2 = "banana";
System.out.println(s1.compareTo(s2) + " " + s2.compareTo(s1));
```

Which of these could be the result of executing the code segment?

(A) `0 0`
(B) `-1 -1`
(C) `-1 1`
(D) `1 -1`
(E) `1 1`

6. Which of the following returns the last character of the string `str`?

(A) `str.substring(0);`
(B) `str.substring(0, str.length());`
(C) `str.substring(length(str));`
(D) `str.substring(str.length() - 1);`
(E) `str.substring(str.length());`

7. Which statement assigns a random integer from 13 to 27 inclusive to the variable `rand`?

(A) `int rand = (int)(Math.random() * 13) + 27;`
(B) `int rand = (int)(Math.random() * 27) + 13;`
(C) `int rand = (int)(Math.random() * 15) + 13;`
(D) `int rand = (int)(Math.random() * 14) + 27;`
(E) `int rand = (int)(Math.random() * 14) + 13;`

8. After executing the code segment, what are the possible values of the variable `var`?

```
int var = (int) Math.random() * 1 + 10;
```

(A) All integers from 1 to 10
(B) All real numbers from 1 to 10 (not including 10)
(C) 10 and 11
(D) 10
(E) No value. Compile-time error.

9. Which of the following statements generates a random integer between -23 and 8 (inclusive)?

(A) `int rand = (int)(Math.random() * 32 - 23);`
(B) `int rand = (int)(Math.random() * 32 + 8);`
(C) `int rand = (int)(Math.random() * 31 - 8);`
(D) `int rand = (int)(Math.random() * 31 - 23);`
(E) `int rand = (int)(Math.random() * 15 + 23);`

10. Consider the following code segment.

```
int r = 100;
int answer = (int)(Math.PI * Math.pow(r, 2));
System.out.println(answer);
```

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

(A) `30000`
(B) `31415`
(C) `31416`
(D) `31415.9265359`
(E) Nothing will print. The type conversion will cause an error.

Advanced Level

11. Consider the following code segment.

```
String ex1 = "example";
String ex2 = "elpmaxe";
ex1 = ex1.substring(1, ex1.indexOf("p"));
ex2 = ex2 + ex2.substring(3, ex1.length()) + ex2.substring(3, ex2.length());
String ex3 = ex1.substring(1) + ex2.substring(ex2.indexOf("x"), ex2.length() - 2);
System.out.println(ex3);
```

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

(A) `amxem`
(B) `amxema`
(C) `amxepma`
(D) `ampxepm`
(E) The program will terminate with a `StringIndexOutOfBoundsException`.

12. Which print statement will produce the following output?

```
/\what    time
"is it?"//\\
```

(A) `System.out.print("/\\what\ttime\n\"is it?\"//\\\\");`
(B) `System.out.print("//\\what\ttime\n\"is it?\"////\\\\");`
(C) `System.out.print("/\\what\time\n\"is it?\"//\\\\");`
(D) `System.out.print("/\\what\ttime\n"is it?"//\\\\");`
(E) `System.out.print("//\\what\time + \"is it?\"////\\\\");`

13. Consider the following code segment.

```
int rand = (int)(Math.random() * 5 + 3);
rand = (int)(Math.pow(rand, 2));
```

After executing the code segment, what are the possible values for the variable `rand`?

(A) 3, 4, 5, 6, 7
(B) 9, 16, 25, 36, 49
(C) 5, 6, 7
(D) 25, 36, 49
(E) 9

14. Consider the following code segment.

```
int val1 = 10;
int val2 = 7;
val1 = (val1 + val2) % (val1 - val2);
val2 = 3 + val1 * val2;
int val3 = (int)(Math.random() * val2 + val1);
```

After executing the code segment, what is the possible range of values for `val3`?

(A) All integers from 2 to 17
(B) All integers from 2 to 75
(C) All integers from 75 to 76
(D) All integers from 2 to 74
(E) All integers from 2 to 18

15. Assume `double dnum` has been correctly declared and initialized with a valid value. What is the correct way to find its absolute value, rounded to the nearest integer?

(A) `(int) Math.abs(dnum - 0.5);`
(B) `(int) Math.abs(dnum + 0.5);`
(C) `(int) (Math.abs(dnum) + 0.5);`
(D) `Math.abs((int)dnum - 0.5);`
(E) `Math.abs((int)dnum) - 0.5;`

16. Which of these expressions correctly computes the positive root of a quadratic equation assuming coefficients `a`, `b`, `c` have been correctly declared and initialized with valid values? Remember, the equation is:

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

(A) `(-b + Math.sqrt (Math.pow(b, 2)) - 4ac) / 2a`
(B) `(-b + Math.sqrt (Math.pow(b, 2) - 4ac)) / 2a`
(C) `(-b + Math.sqrt (Math.pow(b, 2) - 4 * a * c)) / 2 * a`
(D) `(-b + Math.sqrt (Math.pow(b, 2)) - 4 * a * c) / (2 * a)`
(E) `(-b + Math.sqrt (Math.pow(b, 2) - 4 * a * c)) / (2 * a)`