Let's move on to the foundational concepts of the Java programming language. In this chapter, we'll explore into the basic building blocks of Java programs, including how to structure your code, display output, understand different types of data, and perform operations.

What you will learn

In this chapter, you will learn how to:

- Write and run your first Java program.
- Work with variables and data types.
- Use operators and expressions.
- Handle input and output.
- Use constants and wrapper classes.
- Explore the Java API and Math class.
- Practice writing Java with JShell. # 2.1 The Parts of a Java Program Here's where the fun really begins! This sections introduces some basic and necessary parts of a Java program, including:
- **Basic Java program structure**—What's required for a Java program.
- **Print statements**—Writing output to the console to check that your program is running.
- **Comments**—Write notes to yourself and other programmers in the code that the user never sees.
- **Variables**—Powerful data containers that simplify the management of values in your program.

From here on, I encourage you to follow along in your IDE! Copying the code, running it, changing it, and running it again is an important part of learning to program.

Java Program Structure

Every Java program follows a basic structure, often referred to as **boilerplate code**. This provides the necessary framework for your instructions to be executed.

Here's a typical "Hello, World!" program, which is often the first program new developers write.

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

You may be able to understand some of the code, while other aspects may be unclear. Don't worry about memorizing every part of it now. Here's an explanation of each part:

- **`public class HelloWorld`**: This line declares a class named `HelloWorld`. In Java, all code resides within classes. The `public` keyword means this class is accessible from anywhere. The class name should always start with an uppercase letter and typically matches the filename (e.g., `HelloWorld.java`).
- **`public static void main(String[] args)`**: This is the main method, the entry point of every Java application.
  - `public`: Accessible from anywhere.
  - `static`: This method belongs to the `HelloWorld` class itself, not to a specific instance of the class.
  - `void`: This indicates that the method does not return any value.
  - `main`: This is a special name that the Java Virtual Machine (JVM) looks for when starting a program.
  - `String[] args`: This declares a parameter named `args`, which is an array of `String` objects. This allows you to pass command-line arguments to your program, though we won't be using them immediately.
- **`System.out.println("Hello, World!");`**: This is the line that actually performs an action – it prints "Hello, World!" to the console. We'll discuss this more in the next section.
- **`{ }`**: Curly braces define code blocks. They indicate the start and end of classes, methods, and other control structures.

Check Your Understanding (CYU) Questions

1. What is the purpose of the `main` method in a Java program?
2. Why does a Java class name typically match its filename?

Java Compilation to Bytecode with javac

The `HelloWorld.java` program written in the last section is often the first code that Java programmers run because it is a simple, complete Java program.

To run a Java program, you must first compile it into bytecode (a `.class` file), and then the Java Virtual Machine (JVM) can execute the bytecode. When you press the **Run** button on a Java program in an IDE, the IDE automates the two-step process of compiling and executing your code.

This two-step process can be done manually using the `javac` and `java` commands, like so:

```
javac HelloWorld.java
# Result: a file called HelloWorld.class is generated.

java HelloWorld
# Result: Executes HelloWorld.class. Outputs the message:
# Hello, World!
```

The `javac` command is the Java compiler. It takes a `.java` source file and translates the human-readable code into platform-independent bytecode, saving it as a `.class` file. The name of the `.class` file will be the same as the class name in the source file. In this case, `HelloWorld.java` would be compiled into `HelloWorld.class`.

The `java` command is the Java interpreter, which runs the Java Virtual Machine (JVM). It takes the **name of the compiled class file** (without the `.class` extension) as an argument and executes its `main` method. The JVM executes the bytecode contained in `HelloWorld.class`.

When you are working with single-file source code programs (as we are currently), you can simply run the Java interpreter command along with a `.java` file name, like so:

```
java HelloWorld.java
# Hello, World!
```

Using a single command for both compilation and execution creates a `.class` file in memory or a temporary directory and is usually deleted after execution.

The `print` and `println` Methods

Java provides simple ways to display output to the console using the `System.out` object.

`System.out.println()`: This method prints the specified content to the console and then moves the cursor to the next line.

```java
System.out.println("This is the first line.");
System.out.println("This is the second line.");
```

`Output:`

```
This is the first line.
This is the second line.
```

`System.out.print()`: This method prints the specified content to the console but does *not* move the cursor to the next line. Subsequent output will appear on the same line.

```java
System.out.print("This is on the same line. ");
System.out.print("Still on the same line.");
System.out.println(" Now a new line.");
```

Output:

```
This is on the same line. Still on the same line.
Now a new line.
```

Special characters, known as **escape sequences**, can be inserted into a string within a `print()` or `println()` statement in Java. These sequences begin with a backslash \ followed by a character, allowing you to represent characters that are otherwise difficult or impossible to type directly.

Common Escape Sequences:

- `\n`: Newline This moves the cursor to the beginning of the next line, similar to how the println() method works.

```java
System.out.print("First line.\nSecond line.");
```

Output:

```
First line.
Second line.
```

- `\t`: Tab This inserts a horizontal tab, useful for aligning text.

```java
System.out.println("Name\tAge");
System.out.println("Alice\t30");
```

Output:

```
Name    Age
Alice   30
```

- `\"`: Double Quote
  This inserts a literal double quote character ” inside a string that is already enclosed in double quotes.

```java
System.out.println("He said, \"Hello, World!\"");
```

Output:

```
He said, "Hello, World!"
```

- `\'`: Single Quote This inserts a single quote character ‘. While not strictly necessary inside a double-quoted string (you can just type’), it’s a valid escape sequence.

```java
System.out.println("I'm a Java programmer.");
```

Output:

```
I'm a Java programmer.
```

- \\: Backslash
  This inserts a literal backslash character , which is required if you want to print a backslash itself.

```java
System.out.println("C:\\Users\\Guest");
```

Output:

```
C:\Users\Guest
```

By using these escape sequences, you can precisely control the formatting and content of the text you print to the console.

Check Your Understanding: Print

1. What is the key difference between `System.out.print()` and `System.out.println()`?
2. Write a single `System.out.print()` statement that would produce the output: `HelloThere`.

Comments

Comments are non-executable lines of code used to explain what the code does. They are crucial for making your code understandable to yourself and other developers. Java supports two main types of comments:

Single-line comments: Start with `//`.

- Everything from `//` to the end of the line is ignored by the compiler.

```java
// This is a single-line comment
System.out.println("Hello"); // This also works
```

Multi-line comments: Start with `/*` and `*/`.

- Everything between these two markers is ignored.

```java
/*
 * This is a multi-line comment.
 * It can span across several lines.
 */
System.out.println("World");
```

Check Your Understanding: Comments

1. Why are comments important in programming?
2. Which type of comment would you use for a brief explanation of a single line of code?

Variables and Literals

Variables are named storage locations in a computer's memory that can hold data. Literals are constant values that are directly embedded in the code.

A **literal** is the raw data itself, while a **variable** is a named container used to store and reference that data. Think of a literal as the content and a variable as the labeled box you put that content into.

See the following code block for an example:

```
// Print the String literal "Hello" to the console:
System.out.println("Hello");

// Create a name variable (a container) and put a String into it ("Sarah")
String name = "Sarah";
// Print the contents of the variable 'name' to the console:
System.out.println(name);
```

Before you can use a variable, you must declare it by specifying its data type and a name.

```
int score; // Declaration: declares an integer variable named score
score = 100; // Assignment: assigns the literal value 100 to score

// Declaration and initialization in one line
double price = 19.99;
```

Displaying Multiple Items with the + Operator

The `+` operator in `System.out.print` and `System.out.println` can be used to concatenate (join) strings and other data types for output.

```
String name = "Alice";
int age = 30;
System.out.println("Name: " + name + ", Age: " + age);
```

Output:

```
Name: Alice, Age: 30
```

Be Careful with Quotation Marks

String literals are enclosed in double quotation marks (`"`). Characters are enclosed in single quotation marks (`'`). Forgetting or misusing them will lead to compilation errors.

```
String greeting = "Hello"; // Correct: double quotes for String
char initial = 'J';       // Correct: single quotes for char

// int number = "123"; // Error: trying to assign a String literal to an int variable
// char letter = "A"; // Error: trying to assign a String literal to a char variable
```

Identifiers

Identifiers are names given to variables, classes, methods, and other program elements.

- Highly recommended to start with a letter.
- Can start with a dollar sign (`$`) or underscore (`_`), although this is highly discouraged.
- Cannot start with a digit.
- Cannot contain spaces.
- Can contain letters, digits, underscores, or dollar signs.
- Are case-sensitive (`myVariable` is different from `myvariable`).
- Cannot be Java keywords (e.g., `public`, `class`, `int`). ### Java Keywords In Java, a keyword is any one of 68 reserved words that have a predefined meaning in the language. Because of this, you cannot use these words for variables, methods, classes, or as any other identifier.

Examples include keywords that define:

- **Primitive data types**: `byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean`, `true`, `false`, `null`, `void`.
- **Control flow**: `if`, `else`, `switch`, `case`, `default`, `for`, `while`, `do`, `break`, `continue`, `return`.
- **Class, method, access, and variable modifiers**: `static`, `final`, `abstract`, `synchronized`, `volatile`, `transient`, `public`, `private`, `protected`.
- **Object-oriented programming**: `class`, `extends`, `implements`, `interface`, `new`, `this`, `super`, `instanceof`.
- **Exception handling**: `try`, `catch`, `finally`, `throw`, `throws`, `assert`.
- **Modules**: `module`, `exports`, `requires`, `transitive`, `opens`, `to`, `uses`, `provides`, `with`, `sealed`, `permits`.
- **Other**: `goto`, `const`, `enum`, `strictfp`, `record`.

Oracle maintains a list of Java Language Keywords.

Class Names

Class names typically follow **PascalCase** (also known as UpperCamelCase), where the first letter of each word is capitalized (e.g., `MyClass`, `HelloWorld`).

Check Your Understanding: Variables

1. What is the difference between declaring a variable and initializing it?
2. How would you combine the `String` literal "My name is" with a `String` variable `userName` and an `int` variable `userAge` to print a complete sentence?
3. Which of the following are valid Java identifiers?
    - `_variable`
    - `1stVar`

- `my-var`
- `public`
- `total$Amount`
- `my variable`

Exercises (Section 2.1)

1. **Greeting Program**: Write a Java program that declares two `String` variables, `firstName` and `lastName`, and initializes them with your first and last name. Then, print a greeting message to the console that says "Hello, [firstName] [lastName]!" using `System.out.println()` and the `+` operator.
2. **Product Information**: Declare variables for a product name (`String`), its price (`double`), and the quantity in stock (`int`). Assign appropriate literal values. Print out each piece of information on a new line, clearly labeled (e.g., "Product: [name]"). # 2.2 Primitive Data Types Java is a **strongly typed** language. When you create a variable in Java, it must have a data type.

Java has eight primitive data types. These are the most basic data types and are built directly into the language. They store simple values directly in memory. You can combine primitive data types to make more complex custom objects.

Data Type	Description	Range / Values	Default Value	Example
`byte`	8-bit integer	-128 to 127	`0`	`byte myAge = 35;`
`short`	16-bit integer	-32,768 to 32,767	`0`	`short year = 2025;`
`int`	32-bit integer	Approximately -2 billion to 2 billion	`0`	`int population = 150000;`
`long`	64-bit integer	Used for very large whole numbers	`0L`	`long worldPopulation = 8000000000L;`
`float`	32-bit floating-point number	Single-precision decimal values	`0.0f`	`float pi = 3.14f;`
`double`	64-bit floating-point number	Double-precision decimal values	`0.0d`	`double price = 19.99;`
`boolean`	Logical data type	Stores `true` or `false`	`false`	`boolean isAdult = true;`
`char`	16-bit Unicode character	Stores a single character	`\u0000`(null character)	`char initial = 'J';`

Notice how the default value for these primitives is **0**. This is because, in computer science, counting often begins with 0, a convention known as **zero-based numbering**. This is especially true for indexing, such as with strings or arrays, where the first element is at index 0.

We will primarily focus on `int`, `double`, `char`, and `boolean`.

Integer Data Types

`byte`, `short`, `int`, and `long` are used to store whole numbers. The choice depends on the range of values you need to store.

`int` is generally the default choice unless you have specific memory or range requirements.

```
byte smallNum = 120;
int count = 1000000;
long veryBigNum = 9876543210L; // 'L' suffix indicates a long literal
```

Integer data types cannot contain decimal points.

Floating-Point Data Types

`float` and `double` are used to store numbers with decimal points.

- `double`: In most cases, you should use `double` because it provides higher precision and is the default type for decimal literals.

```
double pi = 3.14159;
double scientific = 4.728197e4; // Represents 4.728197 * 10^4, which is 47281.97
```

- `float`: Uses half the memory of `double` and can be slightly faster for certain operations, but offers less precision. If you use a `float` literal, you must append `F` or `f`.

```
float lessPrecisePi = 3.14F;
```

Floating-point literals cannot contain embedded currency symbols or commas. However, for ease of reading, you can embed underscores into numbers to keep track of commas. For example:

```
double pay;
pay = 1000000.25; // pay is 1000000.25
pay = 1_000_000.25;
// pay is still 1000000.25, but the number is clearer to the programmer.
```

The `boolean` Data Type

The `boolean` data type can only hold one of two values: `true` or `false`. It's fundamental for decision-making in programs.

```java
boolean isJavaFun = true;
boolean hasPermission = false;
```

The `char` Data Type

The `char` data type stores a single character. Character literals are enclosed in single quotes.

```java
char grade = 'A'; // Single quote for char
char symbol = '$';
```

Java's `char` type supports Unicode, meaning it can represent characters from almost any language. You can use Unicode escape sequences (e.g., `\u4F60` for ‘ ’), though it's generally recommended to type the character directly if your editor supports it.

`char` also supports hexadecimal values (e.g., `0x6e` for ‘n’). Some hexadecimal values have special meanings: - `0x07` represents the “BEL” character (bell sound). - `0x20` represents an explicit space character.

Variable Assignment and Initialization

You can declare a variable and then assign a value to it in separate steps, or you can initialize it (assign a value) at the time of declaration.

```java
// Declare then assign
int quantity;
quantity = 10;

// Declare and initialize
double temperature = 25.5;
```

Variables in Java *should* always have a type. However, Java 10 introduced the `var` keyword, which allows the compiler to infer the data type based on the initial value. While convenient, it can sometimes hide the underlying type, potentially leading to compiler errors if you're not careful about the data types involved. It's generally not encouraged for beginners as Java is a **strongly typed language**.

```java
// Using var (not generally encouraged for beginners)
// var str = "Hello";  // Compiler infers String
// var num = 123;    // Compiler infers int
// var dbl = 45.67;  // Compiler infers double
```

Variables Hold Only One Value at a Time

A variable can only hold one value of its declared type at any given moment. When you assign a new value to a variable, the old value is overwritten.

```java
int x = 5;
System.out.println(x); // Output: 5
x = 10;                // The old value (5) is replaced by the new value (10)
System.out.println(x); // Output: 10
```

Check Your Understanding: Primitives

1. List the four primitive data types we are focusing on in this chapter.
2. When should you use a `double` instead of a `float`?
3. What is the purpose of the `L` suffix on a `long` literal?
4. What character is represented by `\u4F60`?
5. Explain why using the `var` keyword might be discouraged for new Java programmers.

Extra Exercise (Section 2.2)

1. **Student Profile**: Declare variables for a student's ID number (`long`), their average grade (`double`), whether they are currently enrolled (`boolean`), and their favorite initial character (`char`). Assign appropriate values and print them clearly labeled.
2. **Memory Management**: Imagine you have an `int` variable `count` initialized to `50`. Then you assign `count = 100;`. What is the value of `count` after these operations? Write a small program to verify your answer.
3. **Scientific Notation**: Declare a `double` variable and assign it the value `1.234e-3`. What number does this represent? Print the variable to confirm. # 2.3 Operators and Operands An **operator** is a symbol or keyword that performs an action on one or more values, while an **operand** is a value that the operator acts upon.

For example, in the expression `2 + 2`: - `+` is the operator. - `2` and `2` are the operands.

Java has binary operators (operators that require two operands), a unary operator (one operand), and a ternary operator (three operands).

Arithmetic Operators

Arithmetic operators perform mathematical calculations.

- **Binary Operators**: These operators require two operands (values) to perform an operation.
  - `+` (addition)
  - `-` (subtraction)
  - `*` (multiplication)
  - `/` (division)
  - `%` (modulus - returns the remainder of a division)

```java
int a = 10;
int b = 3;

System.out.println(a + b); // 13
System.out.println(a - b); // 7
System.out.println(a * b); // 30
System.out.println(a / b); // 3 (Integer division rounds down!)
System.out.println(a % b); // 1 (10 divided by 3 is 3 with a remainder of 1)
```

Integer Division

Be careful with integer division. When both operands are integers, Java performs integer division, which truncates (rounds down) the result to a whole number, discarding any fractional part.

```java
int result1 = 7 / 3; // result1 will be 2
double result2 = 7.0 / 3; // result2 will be 2.3333333333333335 (one operand is double)
double result3 = 7 / 3.0; // result3 will be 2.3333333333333335 (one operand is double)
```

Operator Precedence

Java follows the standard order of operations (PEMDAS/BODMAS): 1. Parentheses `()` 2. Exponents (not a direct operator in Java, typically handled by `Math.pow()`) 3. Multiplication `*`, Division `/`, Modulus `%` (from left to right) 4. Addition `+`, Subtraction `-` (from left to right)

```java
int calculation = 6 / 2 * (1 + 2);
// 1. (1 + 2) = 3
// 2. 6 / 2 = 3
// 3. 3 * 3 = 9
System.out.println(calculation); // Output: 9
```

Check Your Understanding: Arithmetic Operators

1. What is an "operand"? Give an example.
2. What is the result of `10 / 4` in Java? Why?
3. Evaluate the following expression step-by-step: `5 + 3 * 2 - 8 / 4`.

Combined Assignment Operators

These operators combine an arithmetic operation with an assignment. They are frequently used in loops or when accumulating values.

- `+=` (add and assign)
- `-=` (subtract and assign)
- `*=` (multiply and assign)
- `/=` (divide and assign)
- `%=` (modulus and assign)

```
int x = 10;
x += 5; // Equivalent to x = x + 5; x is now 15

int y = 20;
y -= 7; // Equivalent to y = y - 7; y is now 13

int z = 5;
z *= 2; // Equivalent to z = z * 2; z is now 10
```

Check Your Understanding: Combined Operators

1. Rewrite `total = total - amount;` using a combined assignment operator.
2. If `value` is `7`, what is `value` after `value %= 3;` ?

Conversion between Primitive Data Types

Java allows you to convert values from one primitive data type to another. This is known as "casting."

Implicit Casting (Widening Conversion)

Implicit casting occurs automatically when you convert a smaller data type to a larger data type. This is safe because there's no risk of losing information. The order of widening conversions is:

`byte` → `short` → `char` → `int` → `long` → `float` → `double`.

```
int num = 100;
double result = num; // Implicitly converts int to double
System.out.println(result); // Output: 100.0
```

Explicit Casting (Narrowing Conversion)

Explicit casting (or narrowing conversion) is required when converting a larger data type to a smaller data type because there's a risk of losing data or precision. You must manually specify the target type in parentheses before the value you want to convert.

```
double decimal = 100.99;
int decCast = (int) decimal; // Explicitly casts double to int
System.out.println(decCast); // Output: 100 (the decimal part is truncated)
```

Mixed Integer Operations

When an operation involves different integer types, the result is typically promoted to the largest type involved to prevent loss of data.

```java
byte b = 10;
int i = 5;
int sum = b + i; // byte 'b' is promoted to int before addition
```

Other Mixed Mathematical Expressions

Similar to mixed integer operations, when a `double` is involved in an arithmetic expression with an `int` or `float`, the result will be a `double`.

```java
int intVal = 5;
double doubleVal = 2.5;
double mixedResult = intVal * doubleVal; // intVal is promoted to double, result is double
System.out.println(mixedResult); // Output: 12.5
```

Check Your Understanding: Data Conversions

1. Explain the difference between widening and narrowing conversion.
2. What happens if you cast a `double` value of `15.75` to an `int`?
3. Why is an explicit cast required when converting a `long` to an `int`?

Extra Exercise (Section 2.3)

1. **Temperature Conversion**: The formula to convert Celsius to Fahrenheit is `F = C * 9/5 + 32`. Write a program that declares a `double` variable for Celsius temperature (e.g., `25.0`), calculates the Fahrenheit equivalent, and prints the result. Be careful with integer division if you use integer literals for `9` and `5`.
2. **Money Calculation**: You have `15.99` dollars and want to calculate how many full dollars you have. Declare a `double` variable for your money and use explicit casting to store the whole dollar amount in an `int` variable. Print both values.
3. **Remainder Practice**: Write a program that takes an integer number of minutes (e.g., `150`) and calculates how many full hours and remaining minutes that represents using integer division and the modulus operator. Print the result (e.g., "150 minutes is 2 hours and 30 minutes"). # 2.4 Non-primitive Data Types While primitive data types store simple values, Java lets you use and create complex data types called **Classes** or **Objects**.

The most common Java Object is the `String` class, a non-primitive data type used to store sequences of characters (text).

Objects Are Created from Classes

In Java, objects are instances of classes. Think of a class as a blueprint, and an object is a concrete realization of that blueprint. Classes allow you to combine multiple data types together to define objects however you see fit.

For example, imagine if there was a Java class called **Song**. The file would be called `Song.java`.

Most recorded songs share some common attributes: they have a title, a songwriter, an album, and a year they were released. In Java, these would be called **fields**. You could imagine that a song title, songwriter, and album could be Strings and the release year could be an integer. These four elements (`String`, `String`, `String`, and `int`) would come together to form a Song object. Classes can be made up of fields of Java primitives and non-primitive objects.

You could also choose to capture other aspects of a song, depending on what is important to you: length, genre, band-name, producer name, location. The possibilities are endless with Java classes.

The class is the generic type that defines what the object is while the object is the specific instance of the class that you create in a Java program.

Just like how we use primitive data types to create variables (for example, `int num = 34`), we also use non-primitive data types to create variables. Instead of writing the primitive data type, we write the classname of whatever object we want to create.

Here are some examples of creating variables with the Song data type:

```
Song petersFave = new Song("Jokerman", "Bob Dylan", "Infidels", 1983)
Song sarahsSong = new Song("dragon eyes", "Adrianne Lenker", "songs", 2020)
Song classicNico = new Song("I'll Keep It With Mine", "Nico", "Chelsea Girl", 1967)
Song classicSam = new Song("Having a Party", "Sam Cooke", "The Man Who Invented Soul", 2000
```

We'll explore Java Objects more later, but they are a defining feature of Java I wanted to introduce early.

Primitive Type Variables and Class Type Variables

- **Primitive type variables**: Store the actual value directly in memory. For example, an `int` variable directly holds the integer value.
- **Class type variables (Objects)**: Store a *reference* (memory address) to where the actual data is stored in memory.

For a `String` object, the variable holds a reference to a memory location that contains information about the string's characters and its length. The characters themselves are stored as primitive `char` types in a different memory location that the `String` object references.

String Class

The **Java API (Application Programming Interface)** provides a large collection of pre-written classes that provide a standardized way for developers to interact with various functionalities.

Keep Exploring! The Java API offers functionalities for a wide range of tasks, from basic operations like mathematical functions to complex features for database connectivity, web development, networking, security, and more. You can see the actual source code definitions of every class in the Java API on the OpenJDK GitHub. A slightly more user-friendly way of exploring the classes provided by the Java API is through the Oracle JDK 21 API docs.

String is one such class provided by the Java API, meaning it's readily available for you to use. In fact, the source code definition of a Java String is available on GitHub at String.java and in the Oracle documentation at Class String.

String is a special non-primitive data type because it can be initialized using a *string literal* (enclosed in double quotes).

```
String greeting = "Hello World"; // String literal assignment
```

All other non-primitive data types (objects) must be created using the `new` keyword. While it's less common for `String`, you *can* also create a `String` object this way:

```
String message = new String("Welcome to Java"); // Using the new keyword
```

Object Methods

Objects have methods that perform actions on that object.

Methods are invoked using dot notation: `object.method()`. If a method requires parameters (additional information to perform its action), they are passed inside the parentheses.

Example `String` methods:

- `.length()`: Returns the number of characters in the string.
- `.replace(char oldChar, char newChar)`: Returns a new string with all occurrences of `oldChar` replaced by `newChar`.
- `.charAt(int index)`: Returns the `char` value at the specified index.
- `.equalsIgnoreCase(String anotherString)`: Compares this `String` to another `String`, ignoring case considerations.
- `.toLowerCase()` and `.toUpperCase()`: Converts all of the characters in this `String` to lower or upper case.

Here's some code examples of String methods:

```
String s = "Sarah";

int len = s.length(); // len will be 5
System.out.println(len);

String newString = s.replace('a', 'b'); // newString will be "Sbrbh"
System.out.println(newString);
```

```java
System.out.println(s.charAt(0)); // returns 'S'
System.out.println(s.charAt(1)); // returns 'A'

System.out.println(s.equalsIgnoreCase("SARAH")); // returns true
System.out.println(s.equalsIgnoreCase("SAMANTHA")); // returns false

String uppercaseName = s.toUpperCase(); // uppercaseName will be "SARAH"
System.out.println(uppercaseName);

System.out.println(s.toLowerCase()); // prints "sarah".
```

You can also perform String methods on String literals (not only variables). For example:

```java
System.out.println("This is a String literal".length()); // prints 24

System.out.println("This is a String literal".replace("literal", "figurative"));

System.out.println("This is a String literal".toUpperCase());
```

You can also chain methods together:

```java
String str = "Welcome to Java String methods. It's a strange foreign world!";
String newStr = str.replace("strange", "whole").replace("foreign", "new").toUpperCase();
System.out.println(newStr);
```

Check Your Understanding: Object Methods

1. How do primitive type variables differ from class type variables (objects) in terms of how they store data in memory?
2. What is the unique way `String` objects can be initialized compared to other Java objects?
3. If you have a `String` variable `word = "banana";`, what would `word.length()` return?

The Scanner Class: Reading Keyboard Input

The next Java class we want to work with is the **Scanner**. The Scanner class is a powerful tool for reading input from various sources, including the keyboard (console). While `System.out` is for printing, `System.in` is for reading input.

Scanner also comes from the Java API. However, it is derived from a different **package** than String.

> **Keep Exploring!** If you look at the Java source code, you'll see that `String.java` is located in a folder called `src/java.base/share/classes/java/lang`. Just focus on the

> part after `classes`, which indicates the classes that are available in Java.
>
> The `java/lang` folder is called the `java.lang` package in Java. This package defines the necessary components for the Java language, so it is automatically included in every Java program by default. The full reference to the String class would be: `java.lang.String`.

Other Java packages are not included by default. The **Scanner** class is located inside of the **util** package (GitHub source code and Oracle documentation), so we will need to **import** it before using it.

To use `Scanner`:

1. **Import**: You must import the `Scanner` class at the top of your Java file.

```
import java.util.Scanner;
```

2. **Initialize**: Create a `Scanner` object using the `new` keyword, passing `System.in` as a parameter to indicate you're reading from the keyboard.

```
Scanner keyboard = new Scanner(System.in);
```

3. **Read Input**: Use `Scanner` methods to read different types of input:
   - `nextLine()`: Reads an entire line of text until the user presses Enter.
   - `nextInt()`: Reads the next integer value.
   - `nextDouble()`: Reads the next double value.

Important! If the `Scanner` method expects a specific data type (e.g., `nextInt()`) and the user enters something that cannot be converted to that type (e.g., text), your program will crash.

```
import java.util.Scanner; // Required for Scanner

public class UserInput {
    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in); // Initialize Scanner

        System.out.print("Enter your name: ");
        String name = keyboard.nextLine(); // Read a line of text

        System.out.print("Enter your age: ");
        int age = keyboard.nextInt(); // Read an integer

        System.out.print("Enter your GPA: ");
        double gpa = keyboard.nextDouble(); // Read a double

        System.out.println("Hello, " + name + "! You are " + age + " years old with a GPA o

        keyboard.close(); // It's good practice to close the scanner when done
    }
}
```

Mixing Calls to nextLine with Calls to Other Scanner Methods

A common pitfall when mixing `nextLine()` with `nextInt()`, `nextDouble()`, or `next()` is that `nextInt()`, `nextDouble()`, and `next()` do not consume the newline character (`\n`) left in the input buffer after the user presses Enter. When `nextLine()` is called immediately after, it consumes this leftover newline, resulting in an empty string.

To prevent this, you can add an extra `keyboard.nextLine()` call to consume the remaining newline character after reading a number.

```java
import java.util.Scanner;

public class ScannerMixExample {
    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);

        System.out.print("Enter an integer: ");
        int number = keyboard.nextInt();
        // Reads the integer, leaves newline in buffer

        keyboard.nextLine(); // Consume the leftover newline

        System.out.print("Enter a line of text: ");
        String text = keyboard.nextLine(); // Now reads the actual text

        System.out.println("You entered: " + number + " and \"" + text + "\"");

        keyboard.close();
    }
}
```

Check Your Understanding: Scanner

1. What is the purpose of `import java.util.Scanner;`?
2. If you use `keyboard.nextInt()` to read a number, what common issue might arise if `keyboard.nextLine()` is called immediately afterwards, and how do you fix it?
3. What happens if a user types "abc" when a `Scanner` is expecting `nextInt()`? ## Creating Named Constants with `final` The `final` keyword is used to declare a constant, which is a variable whose value cannot be changed after it is initialized. Constants are useful for values that should remain fixed throughout the program, such as mathematical constants (e.g., PI) or fixed rates.

By convention, constant names are written in **ALL CAPS** with underscores separating words.

```java
final double PI = 3.14159; // PI is a constant
final int DAYS_IN_WEEK = 7;
final double HOURLY_RATE = 25.50; // Example for assignment 2a

// PI = 3.14; // This would cause a compilation error
```

Check Your Understanding: Constants

1. What is the purpose of the `final` keyword?
2. What is the naming convention for Java constants?

Extra Exercise (Section 2.4)

1. **String Manipulation**:
    - Declare a `String` variable `sentence` and initialize it with "Java Programming is Fun!".
    - Print the length of the string.
    - Create a new `String` called `modifiedSentence` by replacing all occurrences of the letter 'o' with '0' (zero). Print `modifiedSentence`.
2. **Simple Calculator Input**: Write a program that prompts the user to enter two whole numbers. Read these numbers using the `Scanner` object. Then, print their sum, difference, product, and quotient, clearly labeling each result.
3. **Area of a Circle with Constant**: Declare a `final double` constant for PI (you can use `3.14159`). Prompt the user to enter the radius of a circle (a `double`). Calculate and print the area of the circle using the formula `Area = PI * radius * radius`.

Wrapper Classes and Parse Methods

Primitive data types do not have methods. However, Java provides "wrapper classes" for each primitive type, allowing them to be treated as objects and thus have methods. These wrapper classes start with a capital letter and generally use the full word for the primitive type (e.g., `Integer` for `int`, `Double` for `double`, `Character` for `char`, `Boolean` for `boolean`).

These wrapper classes are contained in the `java.lang` package, so they do not need to be imported.

One common use of these wrapper classes is for the `parse` methods, which are used to convert `String` representations of numbers into their respective numeric primitive types.

- `Integer.parseInt(String s)`: Converts a `String` to an `int`.
- `Double.parseDouble(String s)`: Converts a `String` to a `double`.

```java
String str1 = "123";
int number = Integer.parseInt(str1); // Converts "123" to the integer 123
System.out.println(number + 1); // Output: 124

String str2 = "99.95";
double dbl = Double.parseDouble(str2); // Converts "99.95" to the double 99.95
System.out.println(dbl * 2); // Output: 199.9
```

You can also convert a numeric primitive type to a `String` using `String.valueOf()`:

```java
int num = 100;
String strNum = String.valueOf(num); // Converts 100 to the String "100"
System.out.println(strNum + " apples"); // Output: 100 apples
```

These `parse` methods and `String.valueOf()` are examples of **static methods**, which can be called directly on the class name (e.g., `Integer.parseInt`) without needing to create an instance of the class. We will explore static methods in more detail in later chapters.

Check Your Understanding: Wrapper Classes

1. What is the purpose of a wrapper class for a primitive data type?
2. How do the names of wrapper classes typically differ from their corresponding primitive types?
3. If you have the `String` "45.7", which parse method would you use to convert it to a `double`?

Math Class

The `Math` class in Java provides methods for performing common mathematical functions. Like the wrapper classes, you don't need to create an instance of the `Math` class to use its methods; its methods are `static`.

The Math class is contained in the `java.lang` package, so it does not need to be imported.

Some useful `Math` methods include:

- `Math.sqrt(double a)`: Returns the square root of a number.
- `Math.pow(double base, double exponent)`: Returns the value of the `base` raised to the power of the `exponent`.
- `Math.abs(int a)` (and overloads for other types): Returns the absolute value.
- `Math.random()`: Returns a random double value between 0.0 (inclusive) and 1.0 (exclusive).

```java
double squareRoot = Math.sqrt(64); // squareRoot will be 8.0
System.out.println("Square root of 64: " + squareRoot);
```

```java
double powerResult = Math.pow(2, 8); // powerResult will be 256.0
System.out.println("2 to the power of 8: " + powerResult);
```

For a comprehensive list of `Math` methods, you can refer to the Java documentation or resources like W3Schools.

Check Your Understanding: Math Class

1. Do you need to create an object of the `Math` class to use its methods? Why or why not?
2. Which `Math` method would you use to calculate 5 cubed (5 to the power of 3)?

Extra Exercise (Section 2.4 Continued)

1. **String to Number Conversion**: Prompt the user to enter their favorite number as text (e.g., "75"). Read this input as a `String`, then convert it to an `int` and print the number multiplied by 2.
2. **Advanced Calculation**: Use `Math` class methods to calculate and print the result of `sqrt(25) + pow(3, 4)`. # 2.5 JShell JShell is an interactive Java console that comes with the Java Development Kit (JDK). It's useful for quickly testing snippets of Java code, experimenting with syntax, declaring variables, and running calculations without needing to create a full Java program file.

To open JShell, simply type `jshell` in your terminal (macOS/Linux) or PowerShell/Command Prompt (Windows). The terminal will display `jshell>` and you can type Java inline, for example:

```
jshell> int x = 10;
x ==> 10

jshell> x * 2
$2 ==> 20

jshell> String name = "Java";
name ==> "Java"

jshell> name.length()
$4 ==> 4
```

Check Your Understanding (CYU) Questions:

1. What is JShell used for?
2. How do you start JShell from your command line?

Extra Exercise (Section 2.5)

1. **JShell Exploration**: Open JShell and try the following:
   - Declare a `double` variable and assign it `123.45`.
   - Calculate the absolute value of `-15` using `Math.abs()`.
   - Convert the `String` "2024" to an `int` using `Integer.parseInt()`.
   - Experiment with `System.out.print()` and `System.out.println()` in JShell.