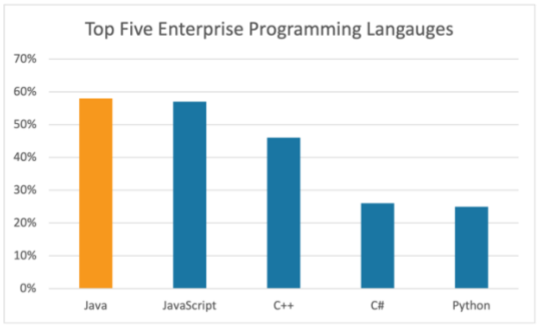
Java Basics

A Brief Introduction for Product Managers, Technical Writers, and Other Non-Programmers in Tech

If you work in the software industry, you have an excellent chance of running into Java at some point in your career. According to a 2018 study by the Cloud Foundry Foundation, 58% of enterprise application developers surveyed reported that their organizations used Java, making it the number one programming language among those surveyed. JavaScript took second place with 57%, but it's important to note that JavaScript is required for most web applications and is frequently used alongside Java or another language.

Figure 1. Percentage of Enterprise Developers Using the Top Five Programming Languages



Source: Adapted from Cloud Foundry Foundation, 2018

If you're not a programmer, Java can be intimidating. If you have experience programming in a language like JavaScript that isn't object-oriented or strongly typed, Java can be confusing. The goal of this article is to remove some of the fear and confusion so that you can understand the shape, if not the mechanics, of Java. It won't teach you how to write code, but it will help you understand some basic concepts and vocabulary so that you can understand what people mean when they talk about working in Java.

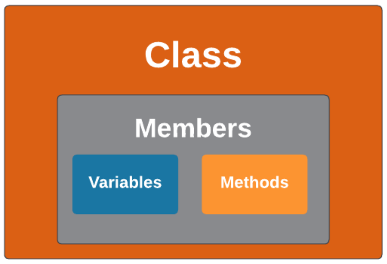
Understanding Object-Oriented Programming

Java belongs to a category of programming languages known as object-oriented programming languages. All object-oriented programming languages share three traits:

- **Encapsulation.** Data and actions are grouped together into units called *classes*.
- **Inheritance.** Classes can be extended, and child classes inherit data and actions from their parents.
- **Polymorphism.** Actions can be used in multiple ways depending on the data they act on.

These concepts are easier to understand when applied to a real-world example. Let's imagine we're designing a software application to keep track of the books in a library.

Figure 2. The Class is Java's Unit of Encapsulation



The Class

Encapsulation means that data and actions are packaged together into organized units. In Java, we call the units *classes.* We call the data *variables*, the actions *methods*, and together, they are known as the *members* of a class.

Let's imagine a Book class as an example.

Declaring Variables

Think of the characteristics of a book that you might want to include in an electronic record. There's the title, the author, and the number of pages. We can create a variable to hold each characteristic. Because Java is what's known as a *strongly typed* language, we must define what type of data each variable can contain when we declare it.

Java has many data types, including six just for numbers. For now, let's keep it simple and stick to *int*, or integer, and *Strings,* which are strings of text enclosed in quotation marks. We'll say that the number of pages will always be an integer, while the title and author will always be strings.

Figure 3. George Boole

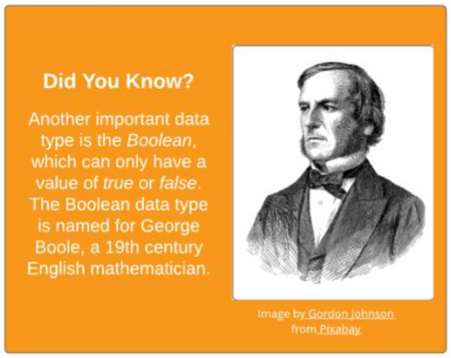
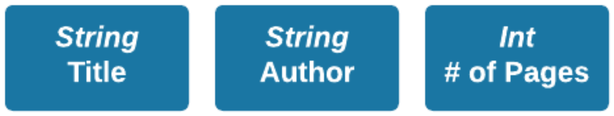

Did You Know?
Another important data type is the *Boolean*, which can only have a value of *true* or *false*. The Boolean data type is named for George Boole, a 19th century English mathematician.
Image by Gordon Johnson from Pixabay

Figure 4. Variables of the Book Class

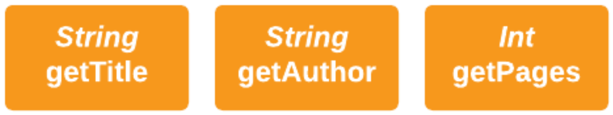


Declaring Methods

Variables define the data that belongs to a class, but it takes methods to interact with that data. What might a library application need to do with information about a book?

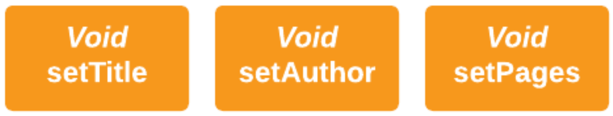
A library application should be able to look up information about the books in our database, so we declare a method for each of our variables to get, or *return*, the value. Java requires us to include the data type a method will return in the declaration. For example, a method to get the number of pages in a book will always return an integer.

Figure 5. Getter Methods



A library application should also be able to add information about new books to the database, so we'll need methods to set these values. These methods will have a return type of *void*. This means that the method doesn't return any data.

Figure 6. Setter Methods



These types of methods are known as *accessors* and *mutators*, or more colloquially as *getters* and *setters*. They're among the most common types of methods used in programming.

The book class now has a set of data and methods to act upon that data, but we still can't make it do anything. That's because a class represents a template for a thing and not the thing itself. To represent an actual thing, we must declare an *object*.

The Object

An object represents a real item that exists in the physical world, such as a book on a library shelf. Each object we create from a class is called an *instance* of that class. There's no limit to the number of book objects we can create from the book class.

Creating an object is sometimes called *instantiation* and requires two pieces. The first is a special method in the class called a *constructor*. The return type of a constructor is the name of the class. That's because, in Java, each class is also a data type.

The second piece required for instantiation is the word *new*. When we want to create an object, we simply declare a *new* instance of a class.

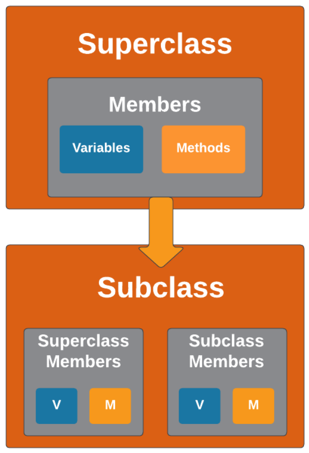
```
Book theHobbit = new Book();
```

This calls the constructor method from the book class to create a new object called theHobbit. This object represents an actual book, *The Hobbit* by J.R.R. Tolkien.

Extending Classes

We've defined a general book class, but what if we want to create categories for more specific types of books? In Java, we can simply extend our book class to create a new class called novel. The novel class inherits all of the members of the book class. We can also add more variables and methods to the novel class, like a string called plot for holding a brief plot summary.

Figure 7. Inheritance



We can extend the novel class to create a romance novel class, a sci-fi novel class, and a young adult novel class. Each of these classes will inherit the members of both the book and novel classes.

When we extend a class, we call the parent class a *superclass* and the child class a *subclass*. Since there's no limit on how many times a class can be extended, it's possible for a class to be both a superclass and a subclass. In the example above, novel would be a subclass to book, but a superclass to romance novel.

Interfaces & Packages

An i*nterface* is similar to a class, but it only defines what a class has to do, not how it will do it. This is useful in situations where we want to apply the same action differently depending on the object on which it is performed.

For example, imagine our library also has DVDs available for checkout, and our library application includes a DVD class. Books can be checked out for four weeks at a time, while DVDs can only be checked out for three days at a time.

We can create a function that will retrieve the due date of either a book or a DVD by creating an interface called DueDate. The DueDate interface declares the name (getDueDate) and return type (*date)* of one method, but it does not include any logic for how the application should retrieve the due date.

Interfaces are *implemented* by classes. The book class and the DVD class can both implement the DueDate interface. Each class that implements the DueDate interface must define logic for the getDueDate method, but the logic does not need to be the same. In this case, the getDueDate method in the book class adds four weeks to the checkout date to find the due date, while the getDueDate method in the DVD class adds only three days. The library application can call the getDueDate method on books and DVDs, and Java will know which logic to use on each one.

This is an example of *polymorphism*, meaning that one method can take many forms.

A *package* is a way of bundling related classes and interfaces together, much like a folder in a computer's file system. Other packages can *import* entire or partial packages. While packages serve many functions, the important thing to understand as a beginner is that they keep code organized, control access, and prevent *name conflicts*.

Conclusion

Java can be overwhelming at first, but if you remember nothing more than the three principles of encapsulation, inheritance, and polymorphism, you'll find the mechanics of Java easier to absorb.

While this article attempts to provide a broad overview of Java in simple terms, we've barely scratched the surface of how Java works. As you continue to learn, remember that there are many resources available to you. If you have a question, someone else is likely to have already asked it on a question-and-answer forum like Stack Overflow. If you'd like to try writing your own code in Java, you can find simple, free coding tutorials for beginners at Codecademy. Finally, don't overlook the developers in your organization. Most developers are happy to answer questions and explain how their code works.

Glossary of Java Terms

Accessor: A method to retrieve the value of a variable. Also known as a getter.
Boolean: A data type that can have a value of either *true* or *false*.
Class: Java's unit of encapsulation. Functions as a template for objects.
Constructor: A special type of method that creates an instance of its class.
Data Type: The kind of data, such as an integer, string, date, or class.
Declaration: A statement that defines a variable, method, or class.
Encapsulation: Data and actions are grouped together into units called *classes*.
Implement: The act of creating a class that defines logic for the methods specified in an interface.
Import: To retrieve code from one package within another.
Instantiate: To create an object.
Inheritance: Classes can be extended, and child classes inherit data and actions from their parents.
Interface: A piece of code that defines what a class does, but not how it will do it.
JVM: The Java Virtual Machine turns the human-readable code written by developers into bytecode, which can be read by a variety of operating systems.
Keyword: A word that has a value defined by Java.
Main: The first method Java executes in an application.
Method: Java's name for a subroutine or function. Logic defining actions to interact with data in a class.
Member: Collective name for the variables and methods of a class.
Mutator: A method to set or update the value of a variable. Also known as a setter.
Name conflict: Occurs when two classes, variables, or methods have the same name.
New: A keyword used to call the constructor method when creating an object.
Object: An instance of a class. Represents a real-world item.
Package: A group of related classes and interfaces.
Polymorphism: Actions can be used in multiple ways depending on the data they act on.
Private: A keyword that forbids a member to be accessed by code outside of its own class.
Public: A keyword that allows a member to be accessed by code outside of its own class.
Return: The act of retrieving a value.
Static: A keyword that allows a member of a class to be used before any objects of that class are created when added to that member's declaration.
Strongly Typed: A category of programming languages that requires programmers to indicate the type of data for all variables and methods.
Subclass: Indicates a class that extends another class.
Superclass: Indicates a class that is extended to create another class.
Variable: Represents a piece of data.
Void: A return type used to indicate that a method does not return any data.

References

Schildt, Herbert. 2018. *Java: A Beginner's Guide*. Oracle Press.

Cloud Foundry Foundation. “These Are the Top Languages for Enterprise Application Development.” *Cloud Foundry*, August 2018. Accessed December 13, 2022. https://www.cloudfoundry.org/wp-content/uploads/Developer-Language-Report_FINAL.pdf.