Java Syntax and Core Concepts

## Pure Object-Oriented Languages

A pure object-oriented language is a programming language in which:

* Everything stored in memory is an object. All data values are instances of classes. This means that there are no primitive types outside of the object-oriented system. Even basic data types like integers, characters and booleans are objects.
* All methods are encapsulated within classes.
* Static methods are not allowed.

Java, like C#, is for the most part an object-oriented language, due to its use of classes and objects, encapsulation inheritance and polymorphism. However, it is **not** a pure object-oriented language because:

* It includes primitive data types (not based on classes).
* It allows static methods.

In Java, all code should be written inside classes, similar to C#. However, unlike C++, Java does not support standalone procedural programming. All executable code in Java must be encapsulated within methods that belong to classes.

## Entry Point of a Java Application

The entry point of a Java application is the `main` method, and this can be put in any class (as a static method). By convention, the class that will contain this method will typically have the same name as the project, for example `HelloWorld`.

For a `main` method to be treated as the entry point of a program, it must have the following signature:

public static void main(String[] args)

If there are multiple classes containing this method signature, the one to use as the entry point of the program may be specified in the IDE. However, it is not typically recommended to have more than one `main` method.

## Primitive Data Types

In Java, there are eight primitive data types:

**`byte`** - 8-bit signed integer

**`short`** - 16-bit signed integer

**`int`** - 32-bit signed integer

**`long`** - 64-bit signed integer

**`float`** - 32-bit floating point (single precision)

**`double`** - 64-bit floating point (double precision)

**`char`** - 16-bit Unicode character (uses single quotes, e.g., 'A')

**`boolean`** - represents one bit of information

**Wrapper Classes**

In Java, primitive data types are not objects and are treated differently from classes and objects.

Java provides a set of wrapper classes in the `java.lang` package to encapsulate these primitive types, allowing them to be used in contexts that require objects, such as collections.

For example, primitive data types in Java do not have methods, however, their wrapper classes do.

This differs from C#, where primitive data types are aliases for structs defined in the `System` namespace of the .NET framework. That is, `int` and `System.Int32` are the same thing. However, in Java, `int` and its wrapper class `java.lang.Integer` are **not** the same thing.

## Java Standard Library

*In C#, the `System` namespace is included in practically all source code files, as it contains the structs for primitive data types, collection classes, classes for system functionality (such as `Console` and `Math`), the `Exception` base class, `Stream` and other classes for handling I/O operations and more. Java doesn't have a direct equivalent to the C# `System` namespace, but it provides similar core functionalities through its own set of packages and classes within the Java Standard Library.*

The Java Standard Library, also known as the Java API (Application Programming Interface), is a collection of pre-written **classes**, **interfaces**, and **packages** provided by Oracle Corporation. These classes and packages provide a wide range of functionality for building Java applications, from basic language features to advanced tools for networking, security, and graphical user interfaces.

**Packages**

A Java package is a namespace that organises a group of related classes and interfaces. It is similar to a C# namespace or Python module.

Some of the core Java packages include:

**`java.lang`**  
Contains fundamental classes and interfaces closely tied to the language itself. These include:

* **`Object`**: the root class of the class hierarchy.
* **`String`**: represents sequences of characters.
* **`System`**: provides system-level functionalities, such as standard input, output and error streams.
* **`Math`**: provides mathematical functions and constants.
* Wrapper classes for primitive data types.

**`java.util`**  
Contains utility classes, including collections framework, date and time facilities, random number generation and more.

* **Collections**: `ArrayList`, `HashMap`, `HashSet` etc.
* **Date and Time**: `Date`, `Calendar`, `TimeZone`.
* **Other Utilities**: `Random`, `Scanner`, `Properties`.

**`java.io`**

Provides classes for system input and output through data streams, serialisation and the file system.

* **Streams**: `InputStream`, `OutputStream`, `Reader`, `Writer`
* **File Handling**: `File`, `FileInputStream`, `FileOutputStream`

**`java.net`**

Contains classes for networking applications.

* **Networking**: `Socket`, `ServerSocket`, `URL`, `URLConnection`.

**`javafx`**

A set of graphics and media packages that enables developers to design, create, test, debug, and deploy rich client applications.

The Java Standard Library (Java API) is part of the JDK.

***\*Most Java packages that you want to include in a source code file need to be explicitly imported, except for the `java.lang` package, which is imported implicitly by default.***

## Memory Management

Java does not use explicitly memory management like C++. Instead, it uses automatic garbage collection, like C#.

## Exception Handling

Java uses `try`, `catch`, `finally`, and `throw`, like C#.

try {

int[] numbers = {1, 2, 3};

System.out.println(numbers[10]);

} catch (ArrayIndexOutOfBoundsException e) {

System.out.println("Array index out of bounds.");

} finally {

System.out.println("The 'try catch' is finished.");

}

## Inheritance and Interfaces

Java does not support multiple inheritance (when a class inherits from more than one parent class). Similar to C#, Java uses single inheritance (one superclass), but allows multiple interfaces.

interface Animal {

void animalSound();

}

class Pig implements Animal {

public void animalSound() {

System.out.println("The pig says: wee wee");

}

}

## Generics

Java uses generics to ensure type safety.

Java uses `<T>` for generics.

public class MyClass<T> {

T value;

public MyClass(T value) {

this.value = value;

}

public T getValue() {

return value;

}

}

## Concurrency

Java has a build-in concurrency model using threads and the `java.util.concurrent` package.

public class MyThread extends Thread {

public void run() {

System.out.println("This code is running in a thread");

}

public static void main(String[] args) {

MyThread myThread = new MyThread();

myThread.start();

}

}

## Casing Conventions

* Classes and Interfaces: PascalCase
* Methods and Variables: camelCase
* Constants: UPPER\_CASE\_WITH\_UNDERSCORES

## Nullable Types

All objects in Java can be null. Primitive data types cannot be null directly, but you can use their wrapper classes which can be null.

## Wrapper Classes for Primitive Data Types

|  |  |
| --- | --- |
| **Primitive Data Type** | **Corresponding Wrapper Class** |
| `byte` | `Byte` |
| `short` | `Short` |
| `int` | `Integer` |
| `long` | `Long` |
| `float` | `Float` |
| `double` | `Double` |
| `char` | `Character` |
| `boolean` | `Boolean` |

## Value and Reference Types

**Primitive data types** are stored as value types in memory directly.

**Objects and arrays** contain a reference (address) to a memory location where the actual object or array is stored. The **reference** is stored in the **stack**, while the actual **data** is stored in the **heap**.

## `System` Class

The `System` class is defined in the `java.lang` package`. It provides various useful methods and fields. Due to containing a private constructor, it cannot be instantiated.

It contains several static fields and methods that provide access to system resources and configuration. These include:

**`out`** - an object of the `java.io.PrintStream` class, which has methods like the following:

**`.print()`** - Prints text without a newline.

**`.println()`** - Prints text followed by a newline.

**`.printf()`** - Formats and prints text.

## Virtual and Abstract Methods

All methods in Java are virtual (able to be overridden) by default, and do not need to be explicitly labelled as such. Abstract methods need to be labelled using the `abstract` keyword. Abstract classes (those containing one or more abstract methods) also need to be labelled as `abstract`.

## Formatted Strings / String Interpolation

Java uses the `String.format()` method for formatted strings / string interpolation. This method uses a slightly different syntax from Python or C#.

This is a text example I made using string interpolation in Java:

protected Animal(String name){  
  
 this.name = name;  
 System.out.println(String.format("An animal named %s was created!", name));  
}

The `%` sign is used followed by a character representing the type of data being interpolated. The variables to use in each instance of interpolation should be specified as arguments to the method. The characters to use include:

|  |  |
| --- | --- |
| %s | Strings |
| %d | Integers |
| %f | Floating-point numbers |
| %x | Hexadecimal numbers |
| %n | Newline |

However, in Java, string interpolation is not typically recommended to be used in these kinds of situations, and string concatenation is preferred (primarily due to performance and readability considerations). String interpolation is more commonly used for more complex formatting requirements, such as text alignment and decimal formatting.

## Extending Constructors in Derived Classes

In the constructor of a derived class, you can call the constructor of the superclass using the `super()` method. This is done implicitly if the constructor takes no arguments, however if it does expect arguments (or you want to use an overload that requires arguments), this needs to be done implicitly.

public Dog(String name, String breed) {

super(name); // Calls Animal(String name)

this.breed = breed;

System.out.println(String.format("A dog of breed %s was created!", breed));

}

## Java Equivalent of C# `foreach` Loop or C++ Range-Based For Loop

Java uses `for-each` loops, which have a similar syntax to C# `foreach` loops.

for (Type element : collection) {

// Use the element

}