# "Hello, World!"

|  |  |
| --- | --- |
| **In Java** | **In C++** |
| // In Greeter.java  public class Greeter  {      public Greeter(String inName) {          name = inName;      }      public String sayHello() {          return "Hello, " + name + "!";      }      private String name;  }  // In GreeterTest.java  public class GreeterTest  {     public static void main(String[] args) {        Greeter greeter = new Greeter("World");        String greeting = greeter.sayHello();        System.out.println(greeting);     }  } | // In Greeter.h  #include <string>  class Greeter  {  public:      Greeter(std::string inName);      std::string SayHello();  private:      std::string name;  };  // In Greeter.cpp  #include <iostream>  Greeter::Greeter(std::string inName) {      name = inName;  }  std::string Greeter::sayHello() {      return "Hello, " + name + "!";  }  // In GreeterTest.cpp  int main() {      Greeter\* greeter = new Greeter("World");      std::string greeting = greeter->sayHello();      std::cout << greeting << std::endl;  } |

Looking at the above example, we can find some major differences between defining a class in Java and C++:

* Each method or field must have its own access modifier.
* You don't use semicolons (;) after the closing brackets in class definitions.
* The main() method is a member of the class - all Java methods must be members of a class.

# Coding Conventions

## Naming Conventions

|  |  |  |
| --- | --- | --- |
| **Types** | **Convetions** | **Examples** |
| Class | Use PascalCase | public class Greeter { ... } |
| Interface | Use PascalCase  Don’t use prefix 'I' | // Good  interface Greeter { ... }  // Bad  interface IGreeter { `} |
| Package | Use all lowercase | package com.example.deepspace; |
| Methods | Use camelCase | displayResult(); |
| Fields,  local variables,  parameters | Use camelCase  Don't use Hungarian  Don't use underscore ( \_ )  Don't use prefix 'm' | // Good  int numberOfPost = 0;  // Bad  int iNumberOfPost = 0;  // Bad  private int \_numberOfPost = 0;  // Bad  private int m\_numberOfPost = 0; |
| Constants | Use all uppercase | static final String SHIPPING\_TYPE = "DropShip"; |
| Tabs or Spaces | Spaces (size of 4) |  |

## Documentation Comments (javadoc)

Java has a standard form for documenting comments that describe methods and classes.

The Java SDK contains a tool, called javadoc, that automatically generates a neat set of HTML pages that document your classes.

Documentation comments are delimited by /\*\* and \*/. For example:

/\*\*

   A class for producing simple greetings.

\*/

class Greeter

{

    /\*\*

        Constructs a Greeter object that can greet a person or entity.

        @param inName the name of person or entity who is addressed in the greetings.

    \*/

    public Greeter(String inName) {

        name = inName;

    }

    /\*\*

        Greet with a "Hello" message.

        @return a message containing "Hello" and name of the greeted person or entity.

    \*/

    public String sayHello() {

        return "Hello, " + name + "!";

    }

    private String name;

}

**Note**: Some programmers like to format documentation comments with a column of asterisks, like this:

/\*\*

\* Greet with a "Hello" message.

\* @return a message containing "Hello" and name of the greeted person or entity.

\*/

It looks pretty, but it **makes it harder to edit the comments**. We don’t recommend that practice!

# Java Software Development Kit (Java SDK)

## What Is Java SDK?

The Java Software Development Kit (SDK) from Sun Microsystems is a **set of command-line tools for compiling, running, and documenting Java programs**.

Two most important tools of Java SDK are:

* Java compiler (javac.exe) to compile Java programs
* Java interpreter (java.exe) to run Java programs.

## How to Use Java SDK?

If you use the Java SDK, follow these instructions:

1. Create a new directory of your choice to hold the program ﬁles. Use a text editor of your choice to prepare the ﬁles Greeter.java and GreeterTest.java. Place them inside the directory you just created.
2. Open a shell window. Use the cd command to change to the directory you just created.
3. Run the compiler with the command: javac GreeterTest.java

If the Java compiler is not on the search path, then you need to use the full path (such as /usr/local/j2sdk1.4/bin/javac or C:\j2sdk1.4\bin\javac) instead of just javac.

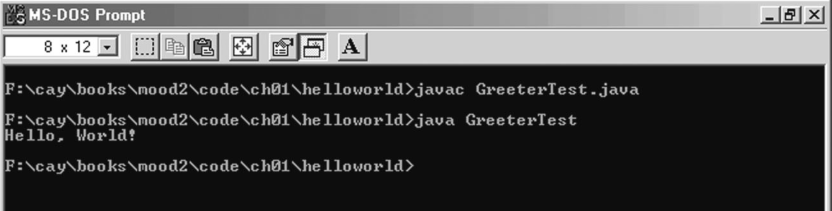
Note that the Greeter.java ﬁle is automatically compiled as well since the GreeterTest class requires the Greeter class.

If any compilation errors are reported, then make a note of the ﬁle and line numbers and ﬁx them.

1. Have a look at the ﬁles in the current directory. Verify that the compiler has generated two class ﬁles: Greeter.class and GreeterTest.class.
2. Start the Java interpreter with the command: java GreeterTest.

Note that do not specify the .class extension. The Java interpreter expects only class files, so it will produce an error if you explicitly specify the .class extension.

1. Now you will see a message "Hello, World!" in the shell window:



**Tip**: For easy development, we can use an IDE to quickly compile, run, debug and test Java programs. The Java community prefers **Eclipse** IDE over any other. Of course, we must make sure the Java SDK is installed first.

# Java Virtual Machine (JVM)

## How Computers Understand and Run a Program?

Computers cannot understand high-level languages -- they can only understand low-level machine languages. A machine language consists of a sequence of binary instructions that can be directly executed on a computer's processor. For this reason, programs written in high-level languages must be translated into machine language programs, which are called *executables*, before they can be run on a computer.

Two methods available for translating a high-level programming language into machine language are compilation and interpretation.

* *Compilation* involves translating an entire high-level program into a whole machine language program, which can then be executed in its entirety.
* *Interpretation* involves translating a high-level program into machine instructions line-by-line; one line is translated and executed before the next line is reached.

## What Makes Java Different from C/C++?

Different computers use different machine languages. An executable that runs on one machine/platform will not run on another machine/platform that uses a different machine language. **In order to run on different computers, C/C++ code must be recompiled using the correct compiler for each machine/platform**.

This is assuming that you can use the same code base without modification for each compiler. But very often, you'll need to modify your code to get it to compile, because you will probably use different APIs for each platform. In other words, you might have to write three different versions of your source code, one for each target platform.

### Java Runs Everywhere

Like C/C++ programs, Java programs must be compiled to be able to run. **Unlike C/C++ programs, Java programs are not compiled down to a platform-specific machine language, but to a platform-independent language called *bytecode***. Bytecode is similar to machine language, but bytecode is not designed to run on any real, physical computer. Instead, it’s designed to be run by a program, called a *Java Virtual Machine* (JVM), which simulates a real machine. That’s why a **Java program can be run on any system that has a JVM installed**.

Simply put, the JVM is an interpreter that translates Java bytecode into real machine language instructions that are executed on the underlying, physical machine. More specifically, the term JVM is used generically to refer to any program that executes Java class files.

**The Java interpreter program is a specific JVM implementation**. The Java Runtime Environment (JRE) is another example of a JVM implementation.

### And It Handles Run-Time Errors Easier

In any language, run-time errors are more difficult to debug than compile-time errors. But because Java programs are executed in a JVM, **run-time errors can be handled in a graceful way**. Whereas C/C++ programs may simply crash, the Java interpreter will at least report the run-time error that caused program execution to halt.

### But C/C++ Is Faster

In Java, bytecode needs to be translated to machine language during run-time. But in C/C++, machine language is already available after built-time. That’s why compiled programs tend to execute faster than interpreted programs.

# Primitive Data Types

|  |  |  |
| --- | --- | --- |
| **Type** | **Size** | **Range** |
| int | 4 bytes | –2,147,483,648 … 2,147,483,647 |
| long | 8 bytes | –9,223,372,036,854,775,808L … 9,223,372,036,854,775,807L |
| short | 2 bytes | -32768 … 32767 |
| byte | 1 byte | -128 … 127 |
| char | 2 bytes | '\u0000' - '\uFFFF' |
| boolean | 1 byte | true, false |
| double | 8 bytes | Approximately ±1.79769313486231570E+308 |
| float | 4 bytes | Approximately ±3.40282347E+38F |

**Notes:**

* long constants have a sufﬁx L (such as 10000000000L) and float constants have a sufﬁx F (such as 3.1415927F).
* Characters are encoded in **Unicode** (that's why a char's size is 2 bytes).
* Common escape character sequences (such as \n, \t, \r, \b, \', \", \\) in Java are same as in C. However, Java doesn’t not support following escapes: alert (\a), question mark (?), hexadecimal (\xhhh) and octal number (\ooo).
* Java use the keyword boolean for boolean type, not bool as C/C++.
* Same as C, conversions in Java that don’t incur information loss (such as short to int or float to double) are always legal. Values of type char can be converted to int. All integer types can be converted to float or double, even if it means a loss of precision.

But all other conversions require a cast such as:

double x = 10.0 / 3.0; // sets x to 3.3333333333333335

int n = (int)x; // sets n to 3

float f = (float)x; // sets f to 3.3333333

* Most aspects of C/C++ fundamental types are defined by the compiler. Different compilers often represent fundamental types in different ways, especially in terms of size and range (e.g., an int in C/C++ can be 4 bytes in 32-bit machine, or 8 bytes in 64-bit machine). By contrast, **Java guarantees the size, range, and behavior of its primitive types no matter what compiler** you use.

# OOP

## Example

|  |  |
| --- | --- |
| **In Java** | **In C++** |
| // In Shape.java  abstract public class Shape  {      protected int height;      protected int width;        Shape(int w, int h) {          this.width = w;          this.height = h;      }        public void area() {          System.out.println("Parent class!");      }  }  // In Rectangle.java  public class Rectangle extends Shape  {      Rectangle(int w, int h) {          super(w, h);      }      public void area() {          int area = super.width \* super.height;          System.out.print("Rectangle area: ");          System.out.println(area);      }  }  // In Triangle.java  public class Triangle extends Shape  {      Triangle(int w, int h) {          super(w, h);      }      public void area() {          int area = super.width \* super.height / 2;          System.out.print("Triangle area: ");          System.out.println(area);      }  }  // In Main.java  public class Main  {      public static void main(String[] args) {          Shape rec = new Rectangle(1, 2);          rec.area();            Shape tri = new Triangle(2, 3);          tri.area();      }  } | // In Shape.cpp  #include <stdio.h>  class Shape {      protected:          int width;          int height;        public:          Shape(int w, int h) {              width = w;              height = h;          }          virtual void area() {              printf("Parent class!\n";          }  };  // In Rectangle.cpp  #include <stdio.h>  class Rectangle : public Shape {      public:          Rectangle(int w, int h) : Shape(w, h) { }            void area() {              int area = width \* height;              printf("Rectangle area: %d\n", area);          }  };  // In Triangle.cpp  class Triangle : public Shape {      public:          Triangle(int w, int h) : Shape(w, h) { }            void area() {              int area = width \* height / 2;              printf("Triangle area: %d\n", area);          }  };  // In Main.cpp  int main() {      Shape\* rect = new Rectangle(1, 2);      rect->area();        Shape\* tri = new Triangle(2, 3);      tri->area();        return 0;  } |

## Constructor

Like C++, Java doesn’t allow to inherit constructors. A superclass constructor must be called implicitly or explicitly by child constructors.

In the above example, the line super(w, h) is required to explicitly call the superclass constructor.

**Notes:**

* **Java compiler will not create a default constructor for you**. You have to create it yourself.

## Abstract Class

You can define an abstract class by using the abstract keyword.

In the subclass, we use the keyword extends to specify which superclass the subclass inherits from. Like C++, once you extend a class from an abstract, in addition to overriding existing methods from the superclass, you can add new fields and methods to it.

However, abstract class **cannot be instantiated**. In the above example, we can Shape rec = new Rectangle(1, 2) but we cannot Shape rec = new Shape(1, 2).

## Abstract Method

You can define an abstract method by using the abstract keyword. It’s a kind of method which is declared, but not implemented. It’s same as pure virtual method in C++. So,

// In Java

abstract public void area();

// In C++

public:

virtual void area() = 0;

## Interface

In C++, we define an interface by declaring all methods as pure virtual and using no fields. In Java, we can define an interface by using the interface keyword (instead of the class keyword).

An interface is like an abstract class, with the following exceptions:

* All methods of an interface are implicitly abstract, so you don't need to use the abstract keyword for each method.
* All members of an interface are implicitly public.
* All fields of an interface are implicitly static and final.
* An interface cannot be instantiated.

In the subclass, we use the keyword implements to specify which interface the subclass inherits from. Like C++, when a class implements an interface, it must implement every method defined in that interface.

## Multiple Inheritance

Unlike C++, Java doesn’t support multiple inheritance. So, Java classes can only extend one class.

## The 'Object' Class

Unlike C++, Java supports a centrally rooted class hierarchy, and the Object class is the root class of that hierarchy. All Java objects ultimately inherit from the Object class, which defines the basic functionality that every Java object is guaranteed to have.

If you do not explicitly extend a class, the Java compiler will automatically extend the class Object. Some commonly-used methods in Object are toString(), wait().

The other implication of having Object is that all objects can be cast down to Object objects. In C++, you can use templates to define data structures that take objects of different types. In Java, you can define data structures that take objects of class Object, and these data structures can hold any Java object.

## The 'super' Keyword

We use the super keyword to explicitly refer to superclass members from a subclass.

Like the this keyword, the use of the super keyword is optional. In the above example, we can int area = width \* height instead of int area = super.width \* super.height.

## Inner Classes and Interfaces

Like C++, you generally use an inner class or inner interface to represent a class or interface that you will only need to use inside the class or interface in which it is declared.

Let's say you are defining a LinkedList class. You will probably use a Node class to represent the individual nodes that are chained together to create the linked list. Since this Node class will only be used internally in the LinkedList class, it’s appropriate to declare it as a private inner class.

class LinkedList

{

    private Node head;

    public LinkedList() {

        head = null;

    }

    public boolean isEmpty() {

        return (head == null);

    }

    public void add(int data) {

        Node node = new Node(data);

        node.next = head;

        head = node;

    }

    public void remove() {

        head = head.next;

    }

    public void display() {

        Node currentNode = head;

        while(currentNode != null) {

            System.out.println(currentNode.toString());

            currentNode = currentNode.next;

        }

    }

    private class Node

    {

        public int data;

        public Node next;

        public Node(int data) {

            this.data = data;

        }

        public String toString() {

            return "(" + data + ")";

        }

    }

    public static void main(String[] args) {

        LinkedList list = new LinkedList();

        list.add(8);

        list.add(6);

        list.add(5);

        list.add(3);

        list.remove();

        list.display();

    }

}

# Other Differences Between C/C++ and Java

## Operator Overloading

C++ allows operator overloading. For example, if you write a class to represent a matrix, you can overload the + operator so that it can perform matrix addition correctly on two matrix objects.

However, **Java does not allow programmers to overload any operator**.

## Parameterless Functions

C/C++ allows to define a function that takes no parameters in one of two ways: by using empty parentheses or by using the void keyword in between parentheses.

However, Java does not allow you to use the void in this way; any **Java method that accepts no parameters must be declared with empty parentheses**.

## Variable-Length Parameter Function

C/C++ allows to define functions that take a variable-length parameter list, by using the (...) notation.

However, Java does not provide this facility, nor is there a replacement for this feature. In general, **passing in an array parameter** can serve a similar purpose.

## Arrays

Java arrays are objects, so they are declared using the new operator. Also, the bracket characters ([]) that are used to indicate arrays are bound to the array type, not the array name.

|  |  |
| --- | --- |
| **In Java** | **In C/C++** |
| int[] scores = new int[100];  char[] grades = {'A', 'B', 'C'};  int[][] table = {{1, 2}, {3, 4}}; | int scores[100];  char grades[] = {'A', 'B', 'C'};  int table[][] = {{1, 2}, {3, 4}}; |

## Passing Arguments to the main() Method

Command-line arguments are passed into Java programs as parameters to the main() method, just as they are in C/C++. However, instead of coming in as a char array like argv, arguments come into Java programs as an array of String objects. Because you can determine the length of a Java array, there is **no need for an int parameter such as C's argc**.

In addition, **unlike C/C++, the first element in the array (at index 0) is the first argument, not the name of the program**.

## Default Parameter

Unlike C++, Java doesn't support default parameters.

## Pointers and References

**Java does not include pointers**. Instead, Java references to objects serve a similar purpose as pointers in C/C++.

Of course, you can't use Java references in the same way that you can use C/C++ pointers. Java references cannot be incremented or decremented, you can't convert references to or from primitive types, and **there are no addressing operators (such as &), or dereferencing operators (such as -> or \*) in Java**.

## Global Variables

Unlike C/C++, **Java offers no way to declare global variables (or methods)**.

## Preprocessors and Macros

**Java does not have a preprocessor**, so no directives or macros.

## struct, union, typedef

**Java does not offer these concepts**.

## enum

Unlike C++98, **enum in Java is internally a class** (like enum class or scoped-enum in C++11).

More details: <https://www.geeksforgeeks.org/enum-in-java/>

## Garbage Collection

Java uses a garbage collector (GC) to automatically reclaim memory by recycling objects when they are no longer referenced. Therefore, we don’t need to deallocate memory manually as in C/C++.

For how GC works: <https://www.dynatrace.com/resources/ebooks/javabook/how-garbage-collection-works/>

## Templates

Unlike C++, Java doesn't support templates. For how to simulate templates in Java, check <https://stackoverflow.com/a/2159346>.

# Collections

# Packages

## What Are Packages?

**Java classes can be grouped into packages**. Package names are dot-separated sequences of identiﬁers, such as

* java.util
* javax.swing
* com.sun.misc
* edu.cs.alice

To guarantee uniqueness of package names, Sun recommends that you start a package name with a domain name (such as com.sun or edu.cs) because domain names are guaranteed to be unique.

## How to Add Classes to Packages?

You place a class inside a package by adding a package statement to the top of the source ﬁle:

package edu.cs.alice;

public class Greeter

{

...

}

The full name of a class consists of the package name followed by the class name, such as edu.cs.alice.Greeter.

**Note**: Large programs consist of many classes in multiple packages. **The class ﬁles must be located in sub-directories that match the package name**. For example, the class ﬁle Greeter.class for the class edu.cs.alice.Greeter must be in a sub-directory edu/cs/alice of the project’s base directory. The base directory is the directory that contains all package directories as well as any classes that are contained in the default package (that is, the package without a name).

--- edu

|--- cs

|--- alice

|--- Greeter.java

## What Is Default Package?

Any class without a package statement is in the "default package" with no package name.

## How to Import Classes from Packages?

It is tedious to use these full names in your code. So, Java provides the import statement to use the shorter class name instead. For example, after you place the following statement into your source ﬁle, you can refer to the class simply as ArrayList:

import java.util.ArrayList;

You can also import all classes from a package:

import java.util.\*;

**Note**: If you simultaneously use two classes with the same name (such as java.util.Date and java.sql.Date), you are stuck – you must use the full name.

**Note:**

You **NEVER need to import the classes in the java.lang package**, such as String or Math. They’re imported in every project by default.

# Annotations

# OSGi

## What Is OSGi?

OSGi, standing for *Open Services Gateway Initiative* (also known as *Dynamic Module System for Java*) defines **an architecture for developing and deploying modular applications**. To simply put, it allows you to break your application into multiple modules and thus more easily manage cross-dependencies between them.

Developing on the OSGi platform means building your app using OSGi APIs (so that different modules can interact with each other), then deploying it in an OSGi container.

## What're Advantages of OSGi?

* You can **install, uninstall, start (activate), and stop (de-activate) different modules of your app dynamically** without having to restart the container.
* Your app can have more than one version of a particular module running at the same time.
* OSGi provides very good infrastructure for developing service-oriented apps, as well as embedded, mobile, and rich internet apps.

## What Is OSGi Container?

As long as your app is compliant with the OSGi specification, it should be able to run in any OSGi-compliant container. Currently, there are three popular open source OSGi containers:

* **Equinox** is the reference implementation for the framework portion of the OSGi Service Platform Release 4. It is the modular Java runtime at the heart of the Eclipse IDE, and implements all of the mandatory and most of the optional features of the OSGi R4 specification.
* **Knopflerfish** is an implementation of the OSGi R3 and OSGi R4 specifications. It implements all the mandatory features and some optional features defined in the R4 specification.
* **Apache Felix** is an OSGi container from the Apache Software Foundation. At the time of writing, this container is not fully compliant with the OSGI R4 specification.

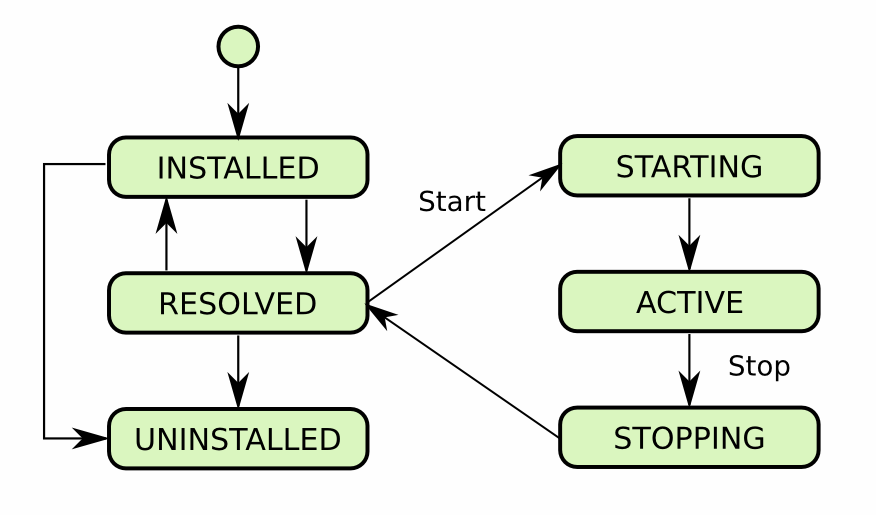
## Life Cycle of Bundles in OSGi

After installed, each bundle is persisted in a local bundle cache. The OSGi runtime then tries to resolve its dependencies.

If all required dependencies are resolved, the bundle is in the RESOLVED status. Otherwise, it stays in the INSTALLED status.

In case several bundles exist, the one with the highest valid version is used. If the versions are the same, the bundle with the lowest unique identifier (ID) is used. Every bundle gets this ID assigned by the framework during the installation.

This life cycle is depicted in the following picture:



## OSGi in Eclipse

Eclipse IDE offers the **best support for developing OSGi bundles**. Not only does it provide wizards for creating OSGi bundles, it also has an embedded Equinox OSGi container that you can use to execute and debug OSGi plugins.

**In this tutorial, we will use Equinox as our OSGi container**.

*You might not know!*

- Every Eclipse plug-in is essentially an OSGi bundle with some additional Eclipse-specific code. Eclipse also allows you to build standard-compliant OSGi bundles without code specific to Eclipse.

- The Eclipse programming model typically calls them **plug-in** but these terms are interchangeable.

## How to Develop OSGi Bundles in Eclipse?

<https://www.eclipse.org/forums/index.php/t/273044/>

<https://www.javaworld.com/article/2077837/java-se-hello-osgi-part-1-bundles-for-beginners.html?page=2>

<http://tech-ragtime.seesaa.net/article/233880180.html> (it is convenient to use the add required bundle button)

<https://www.vogella.com/tutorials/OSGi/article.html>