The Joy of Coding

> The Java™ programming platform consists of the Java programming language, a set of standard libraries, and the virtual machine in which Java programs run. The syntax of the Java programming language closely resembles C, but has features such as a built-in boolean type, support for international character sets, and automatic memory management.

The Java Programming Language

- What is Java?
- The Java Programming Language
- Tools for compiling and running Java

 Last updated April 20, 2025.

What is Java?

"A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multi-threaded, and dynamic language."
– Original Java White Paper

"C++ without the guns, knives or clubs."
– James Gosling

Java is a programming platform for developing portable, hardware-independent applications and libraries.

Java provides a standard application programming interface (API) for dealing with common data structures (e.g. linked lists and hash tables), file I/O, etc.

Java was designed with modern computing in mind and thus contains facilities for networking, graphics, and security.

Java is rapidly becoming the language of choice for application and enterprise-level computing

Java development tools can be downloaded from:

`https://openjdk.org/projects/jdk`

Alphabet Soup

Over the years, there have been a number of acronyms associated with Java*

- JDK: The Java Development Kit was the original name of the Java toolset provided by Sun
  - Compiler, runtime environment, standard libraries
- J2SE: The JDK was renamed to the Java 2 Standard Edition
  - The "2" was supposed to signify the maturity of the Java platform
- J2EE: Java 2 Enterprise Edition is a suite of Java-based enterprise technologies
  - Database access, middle tier data modeling and business logic, web front end
- Java 5: J2SE 1.5 became know as "Java 5" to once again show the maturity of the platform
- JSE and JEE: With the advent of "Java 5", the J2 got confusing
  - "Java Platform Standard Edition"

*Ugh. Marketing.

The Java Programming Language

Developed in 1995 by Ken Arnold and James Gosling at Sun Microsystems

Java programs are built from *classes* that have *methods* consisting of *statements* that perform work.

Every class is in a *package*. Packages provide a naming context for classes.

Program execution begins with a `main` method whose sole argument is an array of `String`s that are the command line arguments.

The infamous Hello World program:

```
package edu.pdx.cs.joy.lang;

public class Hello {   // Hello is a class
  // main is a method
  public static void main(String[] args) {
    System.out.println("Hello World!");
  }
}
```

This program prints the string `Hello World` to standard output.

Java's Execution Environment

Java was designed to be platform independent

"Write once, run anywhere"

Java programs (classes) are compiled into binary *class files* that contain machine instructions called *bytecodes*.

The bytecodes are executed by a Java Virtual Machine (JVM).

JVMs are platform-dependent and are usually implemented in a language like C.

Compiling a Java Program

A Java class is described in a text file that contains its source code

The name of the file corresponds to the name of the class

It is a good idea to place source files in a directory whose name corresponds to the class's package

For instance, our Hello World class's source file is:

`edu/pdx/cs/joy/lang/Hello.java`

These conventions help keep your source code organized

The `javac` tool compiles Java source code into bytecode that is placed in a *class file*

```
$ cd edu/pdx/cs/joy/lang
$ javac -d ~/classes Hello.java
```

The `-d` option specifies where the class file is written

`~/classes/edu/pdx/cs/joy/lang/Hello.class`

Executing a Java Program

The `java` command invokes the Java Virtual Machine

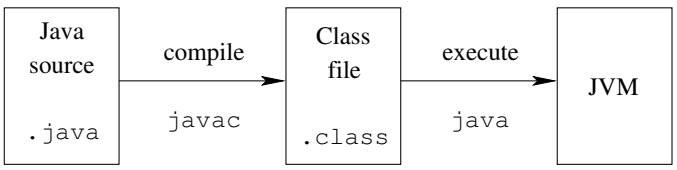
Execution begins with the `main` method of the given class. (Note that `java` takes the name of a **class**, not the name of a class file!)

`java` requires the fully-qualified (including package) name of the `main` class

You also need to tell the JVM where to look for class files

- The `-classpath` (or `-cp`) option specifies a directory path to search for classes
- Alternatively, you can set the `CLASSPATH` environment variable

```
$ java -cp ~/classes edu.pdx.cs.joy.lang.Hello
Hello World
```



Types in the Java Programming Language

In addition to class types, Java has eight *primitive types*:

`boolean`	`true` or `false`
`char`	16-bit Unicode 1.1 character
`byte`	8-bit signed integer
`short`	16-bit signed integer
`int`	32-bit signed integer
`long`	64-bit signed integer
`float`	32-bit IEEE 754-1985 floating point
`double`	64-bit IEEE 754-1985 floating point

Example using primitive types:

```
package edu.pdx.cs.joy.lang;

public class Add {
  public static void main(String[] args) {
    int sum = 4 + 5;
    System.out.println(sum);
  }
}
```

Literals

`int` literals: `42`, `052`, `0x2a`, `0X2A`

`long` literals: An `int` literal with `L` or `l` appended

- Starting in Java 7, binary notation can be used for integral values (`byte`, `short`, `int`, and `long`)
- `byte someByte = (byte) 0b00101010;`

Floating point literals: `3.2`, `3.2e1`, `.32E2`

A floating point literal is by default a `double`. Appending an `f` or `F` to a floating point literal makes it a `float`.

Also in Java 7, an underscore can appear anywhere in a numeric literal as a separator

- `long marker = (long) 0xCAFE_BABE;`
- Underscore can't appear at the beginning or end of a literal or adjacent to a decimal point

`char` literals (surrounded by single quotes): `\n`, `\t`, `\\`, `\'`, `\"`, etc.

`char` literals may also be specified using their octal or hexadecimal values: `\ddd` and `\uXXXX`

`String` literals (surrounded by double quotes) cannot contain newlines. If you want a line break, use `\n`.

Arrays

An array is an ordered collection of values that all have the same type.

An array containing a given number of elements is created with the `new` operator (kind of like `calloc`):

```
int[] arr = new int[5];
```

The length of an array is fixed and can be accessed using:

```
int size = arr.length;
```

You can also have arrays of arrays:

```
int rows = 4; int columns = 5;
float[][] matrix = new float[rows][columns];
```

Arrays can be initialized using:

```
String[] workdays = {"Monday", "Tuesday",
              "Wednesday", "Thursday", "Friday"};
```

Arrays are zero-indexed:

```
System.out.println(workdays[2]);
Wednesday
```


Comments in Java Programs

Java allows both C* and C++ style comments.

```
package edu.pdx.cs.joy.lang;

/**
 * This program prints out the product of two
 * <code>double</code>s.
 */
public class Product {
  public static void main(String[] args) {
    /* Multiply two doubles... */
    double product = 3.5 * 1.8;
    System.out.println(product);  // Print product
  }
}
```

Comments between `/**` and `*/` are called *documentation comments*.

Doc comments describe the class or method directly following them.

The `javadoc` tool uses these comments to generate HTML documentation for classes.

*Java comments do not nest!

Operators

Binary arithmetic operators: `+ - * / %`

Unary arithmetic operators: `- +`

The binary + operator is *overloaded* to concatenate two `String`s:

```
String shaft = "Shaft!";
shaft += " Can you dig it";
System.out.println(shaft + "?");
```

The + operator can also be used to concatenate primitive types to a `String`:

```
System.out.println("2 plus 2 is " + (2 + 2));
```

Increment and decrement operators: `++ --`

```
int i = 16;
System.out.println(++i + " " + i++ + " " + i);
```

Prints `17 17 18`

Operators

Relational/equality operators: `> >= < <= == != && ||`

These operators have `boolean` results

```
void f(boolean a, boolean b) {
  // This is illegal
  boolean c = ((a && b) == 0);
}
```

Bitwise operators: `& | ^ ~`

Shift operators:

<<	Shift left, fill with zeroes
>>	Shift right, fill with highest (sign) bit
>>>	Shift right, fill with zeroes

The conditional operator: `?:`

Assignment operators: `+= -= *=` %= etc.

Operators and Precedence

Operators in Java from highest to lowest precedence

postfix operators	`[] .` *expr*`++` *expr*`--` `(`*params*`)`
unary operators	`++`*expr* `--`*expr* `-`*expr* `+`*expr* `! ~`
creation or cast	`new (`*type*`)` *expr*
multiplicative	`* / %`
additive	`+ -`
shift	`<< >> >>>`
relational	`< > >= <= instanceof`
equality	`== !=`
bitwise AND	`&`
bitwise exclusive XOR	`^`
bitwise inclusive OR	`|`
logical AND	`&&`
logical OR	`||`
conditional	`?:`
assignment	`= += -= *= /= %= >>= <<=` etc.

Binary operators are *left-associative*. Assignment is *right-associative*: `a=b=c` is `a=(b=c)`

Control flow Statements

`if-else` statement:

```
public String whoIsBigger(int a, int b) {
  if (a > b) {
    return a + " is bigger";
  } else if (b > a) {
    return b + " is bigger";
  } else {
    return "The values are the same";
  }
}
```

`switch` and `case:`

```
public String getEnding(int n) {
  switch (n) {
    case 1: return "st";
    case 2: return "nd";
    case 3: return "rd";
    default: return "th";
  }
}
```

Prior to Java 7, only primitive types and `enum` types could be used in `case` statements. Since Java 7, `String`s can be used in `case` statements. Note that the comparison is made is made using the `String`'s `equals` method which is case-sensitive.

Control Flow Statements

`while` and `do-while`:

```
public String readLine() {
  String s = "";
  char c = readChar();
  while (c != '\n') {
    s += c;
    c = readChar();
  }
  return s;
}
```

`for` statement:

```
public String readLine() {
  String s = "";
  for (char c = readChar(); c != '\n';
      c = readChar()) {
    s += c;
  }
  return s;
}
```

Labels, `break` and `continue`

Statements may be preceded by a *label*

You can use labels in conjunction with a `break` or `continue` to exit from an outer loop:

```
boolean searchFor(double[][] matrix, double d) {
  boolean found = false;
 OUTER:
  for (int i = 0; i < matrix.length; i++) {
    for (int j = 0; j < matrix[i].length; j++) {
      if (matrix[i][j] == d) {
        found = true;
        break OUTER;
      }
    }
  }
  return found;
}
```

There is no `goto` in Java. There are other, more clean, mechanisms in Java to transfer control flow.

Enhanced `for` loop

J2SE 1.5 introduced an enhanced `for` loop syntax that doesn't require an index variable:

```
int[] array = ...
int sum = 0;
for (int i : array) {
  sum += i;
}
```

This syntax can be read as:

> "For each `int`, `i`, in array, `array`, do..."

Variable Scoping

Variable scoping is about the same as in C++

- The variable declared in the initializer of a `for` loop is not visible outside of the loop
- Variables declared inside an `if` block, etc. are not visible outside
- Variable redeclaration is illegal

```
void scoping(int one) {
  for (int i = 0; i < one; i++) {
    // i can be used here
  }

  // i cannot be used here

  if (one < 5) {
    int two = 7;
    // two can be used here
  }

  // two cannot be used here
}
```

Note that the names of variables, methods, etc. are **case-sensitive**

Exceptions: When Bad things happen to Good Objects

Errors of varying severity may occur during program execution

- Dividing by zero
- Trying to read a non-existent file
- Indexing beyond the end of an array

If your program tried to accommodate every potential error case, it would become extremely messy.

Java uses *exceptions* to signal errors without cluttering code

When an error condition occurs, an exception is *thrown*. Methods declare that they might throw an exception.

An exception is *caught* by encompassing code and handled appropriately (e.g. printing an error message or prompting the user to enter another file name).

Kinds of exceptions

Checked exceptions are unexpected, but not surprising (e.g. a file cannot be found).

If a method may throw a checked exception, it must be declared in the method's `throws` clause.

Unchecked exceptions are more rare and usually signal a serious problem with your program (e.g. a divide by zero, there is no more memory left).

Exceptions are handled with a `try-catch-finally` block:

```
try {
  // Code that may throw an exception or two
  readFromFile(file);
} catch (FileNotFoundException ex) {
  // Executed if the exception was throw
  printOutErrorMessage("File not found!");
} catch (EndOfFileException ex) {
  // Some exceptions can be "ignored"
} finally {
  // Executed regardless of whether or not an
  // exception was thrown ("clean up code")
  closeFileStream(file);
}
```


Exceptions example

```
package edu.pdx.cs.joy.lang;

public class AddTwo {
  /** Prints the sum of two numbers from the
   * command line. */
  public static void main(String[] args) {
    int anInt = 0, anotherInt = 0;

    try {
      anInt = Integer.parseInt(args[0]);
    } catch (NumberFormatException ex) {
      System.err.println("Invalid integer: " +
                         args[0]);
      System.exit(1);
    }

    try {
      anotherInt = Integer.parseInt(args[1]);  // 25
    } catch (NumberFormatException ex) {
      System.err.println("Invalid integer: " +
                         args[1]);
      System.exit(1);
    }

    System.out.println(anInt + " + " + anotherInt +
                       " = " + (anInt + anotherInt));
  }
}
```


Working with our exceptions example

```
$ java -cp ~/classes edu.---.AddTwo 1 2
1 + 2 = 3

$ java -cp ~/classes edu.---.AddTwo Fred 2
Invalid integer: Fred

$ java -cp ~/classes edu.---.AddTwo 1 Bob
Invalid integer: Bob

$ java -cp ~/classes edu.---.AddTwo 1
Exception in thread "main"
    java.lang.ArrayIndexOutOfBoundsException: 1
at edu.---.AddTwo.main(AddTwo.java:25)
```

The last example shows what happens when you don't catch an exception

There was only one element in the `args` array, attempting to access element `1` caused an `ArrayIndexOutOfBoundsException` to be thrown

The JVM prints out a *stack trace* that includes the line number in the source code at which the exception was thrown

Catching multiple types of exceptions

Java 7 introduced a language syntax for catching multiple types of exceptions in a try/catch block

```
try {
  queryDataBase();
} catch (IOException | SQLException ex) {
  printOutErrorMessage("Error while querying", ex);
}
```

You can throw your own exceptions

```
package edu.pdx.cs.joy.lang;

public class DivTwo {
  public static void main(String[] args) {
    double d1 = 0.0;
    double d2 = 0.0;

    if (args.length < 2) {
      System.err.println("Not enough arguments");
      System.exit(1);
    }

    try {
      d1 = Double.parseDouble(args[0]);
      d2 = Double.parseDouble(args[1]);
    } catch (NumberFormatException ex) {
      System.err.println("Not a double: " + ex);
      System.exit(1);
    }

    if (d2 == 0.0) {
      String m = "Denominator can't be zero!";
      throw new IllegalArgumentException(m);
    }
    System.out.println(d1 + " / " + d2 + " = " +
                       (d1/d2));
  }
}
```


Working with our example

```
$ java -cp ~/classes edu.---.DivTwo 5.0 2.0
5.0 / 2.0 = 2.5

$ java -cp ~/classes edu.---.DivTwo 5.0 -4
5.0 / -4.0 = -1.25

$ java -cp ~/classes edu.---.DivTwo 5.0 Fred
Not a double: java.lang.NumberFormatException: Fred

$ java -cp ~/classes edu.---.DivTwo 42.6
Not enough arguments

$ java -cp ~/classes edu.---.DivTwo 5.0 0
Exception in thread "main"
  java.lang.IllegalArgumentException:
  Denominator can't be zero!
at edu.---.DivTwo.main(DivTwo.java:34)
```


Packages

Naming conflicts often arise when code is reused

- Multiple companies have a `User` class

Java uses packages to distinguish between classes

- Every class is in a package
- Classes in the same package have related functionality (e.g. `java.net`)
- Packages are hierarchical in nature
  - The standard Java packages begin with `java`
  - The classes for this course begin with `edu.pdx.cs.joy`
  - All of your classes must begin with `package edu.pdx.cs.joy.`*userid*
- A class's package is specified by the `package` declaration at the top of its source file
  - If no `package` declaration is specified the class is placed in the *default package*

Packages

When a class references another class that is outside of its package, it must be *fully-qualified* or `imported`

Classes in the `java.lang` package are implicitly imported

```
package edu.pdx.cs.joy.lang;

import java.io.File;  // Just File from java.io

public class Packages {
  public static void main(String[] args) {
    File file = new File(args[0]);
    java.net.URL url = file.toURL();
    String name = url.toString();
    System.out.println(name);
  }
}
```

You can import all of the classes in a package with `import java.io.*;`

Note that the `*` is not recursive!

- `import java.*` will not import all Java classes

Differences Between Java and C++

Java does not have `goto`

Exceptions in Java must be caught

No “pass by reference” (no pointers, either)

No << operator for I/O

Java operators cannot be overridden or overloaded by the programmer

No preprocessor (`#define` and friends)

The `!` operator can only be used with `boolean`s

- Illegal code:

```
void wrong(int bad) {
  if (!bad) {
    // ...
  }
}
```


The `jar` Tool

`jar` is a tool that allows you to combine multiple files (usually class files) into a “Java Archive”

Let’s you put related classes (e.g. all of the classes in your library or application) into a single file

- Makes downloading easier
- Jar files can be “signed” for security

`jar` looks a lot like the UNIX `tar` tool:

```
$ cd ~/classes
$ jar cf classes.jar .
$ jar tf classes.jar
META-INF/
META-INF/MANIFEST.MF
edu/
edu/pdx/
edu/pdx/cs/joy/
edu/pdx/cs/joy/lang/
edu/pdx/cs/joy/lang/Hello.class
```


The `jar` Tool

The JVM knows how to read classes from a jar file

```
$ java -cp classes.jar edu.pdx.cs.joy.lang.Hello
Hello World
```

We will be using a number of jar files:

`projects.jar`	Classes for your project
`examples.jar`	Example code for the course
`family.jar`	The family tree application
`grader.jar`	Code for submitting your projects

Jar files may have their contents compressed and may contain files such as pictures and sounds that may be needed by an application

A jar file contains a special entry called the *manifest* that describes the jar file

- Version, creator, digital signature information, etc.

Classes in the `java.util.jar` package can be used to manipulate jar files

Documenting Your Code with `javadoc`

`javadoc` is a utility that generates HTML documentation for Java classes

The HTML contains links between classes for easy navigation

`javadoc` uses documentation comments that describe classes, methods, and exceptions

Documentation is placed between `/**` and `*/` comments that precede the class, field, or method being described

Special comment tags used with `javadoc`:

`@author`	Who wrote it
`@param`	Description of a parameter to a method
`@return`	Description of value returned by a method
`@throws`	An exception thrown by the method
`@see`	References another class or method

A class with `javadoc` comments

```
package edu.pdx.cs.joy.lang;

/**
 * This class demonstrates Javadoc comments
 *
 * @author David Whitlock
 * @version 1.0
 */
public class Javadoc {

  /**
   * Returns the inverse of a <code>double</code>
   * @param d
   *        The <code>double</code> to invert
   * @return The inverse of a <code>double</code>
   * @throws IllegalArgumentException
   *         The <code>double</code> is zero
   */
  public double invert(double d)
   throws IllegalArgumentException {
    if (d == 0.0) {
      String s = d + " can't be zero!";
      throw new IllegalArgumentException(s);
    } else {
      return 1.0 / d;
    }
  }
}
```


Using the `javadoc` tool

`javadoc` needs to know both the location of your source files and your class files

`-sourcepath` dir	Where to find Java source files
`-classpath` dir	Where to find class files
`-d` dir	Destination directory, where html files are placed

To generate HTML file for classes in the `edu.pdx.cs.joy.lang` package (\ is the UNIX command line continuation character):

```
$ javadoc -classpath ~/classes -sourcepath src \
  -d ~/docs edu.pdx.cs.joy.lang
```

The HTML files will be placed in `~/docs`

Javadoc offers a powerful and standard means for documenting Java classes

Summary

Java programs are written with classes that contain methods. Execution starts with the `main` method.

Java programs are compiled into a intermediate binary form called bytecode

The bytecode is executed by a virtual machine thus making Java programs platform-independent

Java has many of the same primitive data types and control flow structures as C

To encourage good error handling, Java has built-in exceptions

Java classes are organized into packages

The `jar` tool is used to bundle class files together and the `javadoc` tool creates HTML documentation from Java source code