Chapter 1

The Java architecture

Compiling, Interpreting, and debugging

Major topics:

- The Java Standard Edition Platform
- The Java Virtual Machine
- The Java Compiler
- Inspecting source and byte code
- Activity: Quiz

1.1 Compiling and Interpreting

A Central Processor Unit (CPU) is part of the machine processor that executes simple bits operations:

⊙ **Example:** (R-Type operation execute by a CPU) Adding register 1 and 2 and placing to register 6. “funct” describes the operation:

op	rs	rt	rd	shamt	funct	
0	1	2	6	0	32	decimal
000000	00001	00010	00110	00000	100000	binary

A machine code or assembly code is a set of instructions on the bits operations that can be executed by a CPU. Instructions are patterns of bits that by physical design correspond to different commands to the machine. Writing directly machine code is hard and time expensive. Higher level languages allow to write code that humans can better understand. Every processor or processor family has its own machine code instruction set.

Compiling code translates a whole piece of code into machine code and then executes the machine code in the CPU. The computation is therefore delegated to HW. Systems may also differ in other details, such as memory arrangement, operating systems, or peripheral devices. Because a program normally relies on such factors, different systems will typically not run the same machine code, even when the same type of processor is used. A good compiler performs a set of optimisations: for instance while it looks up for a variable in the main memory of a machine, it performs other CPU operations on the rest of the machine code until the variable has been retrieved. This is possible because the whole code as been translated into machine code.

Interpreting code compiles and executes a piece of code line by line. On the one hand, interpreting a code cannot perform much optimisation as it knows only the instructions for part of the code until the line that has been interpreted. On the other hand, to interpret a code in a specific language, we need only to have an environment that compiles and executes (both!) the code line by line. Once we have it we do not need any CPU. Thanks to its environment the language is portable. An example are PHP or Pynton and many scripting languages (Java script). When we call "php -f file.php" from a shell command, we are invoking such environment. The good aspects of such environment is that they can be easily used in web browser.

Programming languages can be compiled and interpreted. Java architecture is a good example to see how compilation and interpretation work.

1.1.1 Code Translations in Java

In Java, we can simply write the source code in *plain text* files ending with the **.java** extension. With the help of a compiler, source files are then compiled into **class** files ending with the **.class** extension by the javac compiler. A class file does not contain code that is native to a specific processor; it instead contains *bytecode* written in the language understandable by the Java Virtual Machine (JVM). The byte code has a hardware- and operating system-independent binary format.

A class loader is responsible of dynamically load class files at run-time and passing the byte code to the JVM. The java loader runs the file with an instance of the Java VM, specific for each CPU and operating system. Before passing the class file loaded in the run-time environment, the Java verifier checks the fields for security.

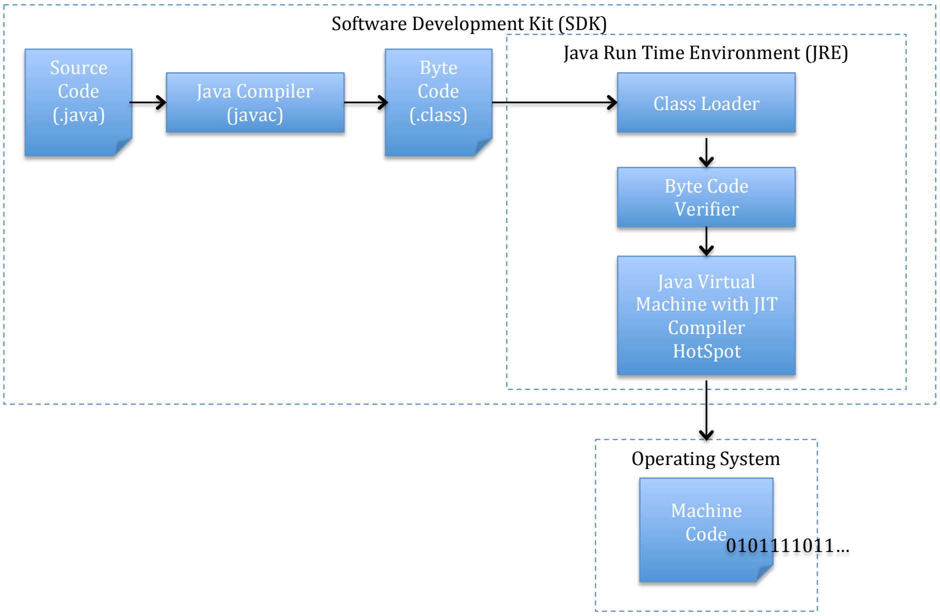

Figure 1.1: The key components of a Java Platform

1.2 The Java Standard Edition Platform (JSE) and The Standard Development Kit (SDK)

The Java Standard Edition Platform (JSE) is the software environment in which a program is compiled and runs and consists of several components. Each component executes in a container. To interoperate with various containers, these components require deployment descriptor files, configuration files, property files, and/or metadata files, and other configuration files. All these files describe the components and how they will interact with other components and their container environment. Every implementation of JSE must adhere to the following specifications:

- Development Tools: to compile, run, monitor, debug, and document an application. The fundamental tools are the javac compiler, the java launcher, and the javadoc documentation tool.
- Application Programming Interface (API): The API provides the core functionality of the Java programming language.
- Deployments tools: Software tools with mechanisms for deploying an

application such as the Java Plug-In software.

- User Interface Toolkits: These are tools for Graphical User Interfaces (GUIs) such as JavaFX, Swing, and Java 2D toolkits.
- Integration Libraries: Integration libraries enable database access and manipulation of remote objects. Examples are JDBC API, Java Naming and Directory Interface (JNDI) API, Java Remote Method Invocation (JRMI).

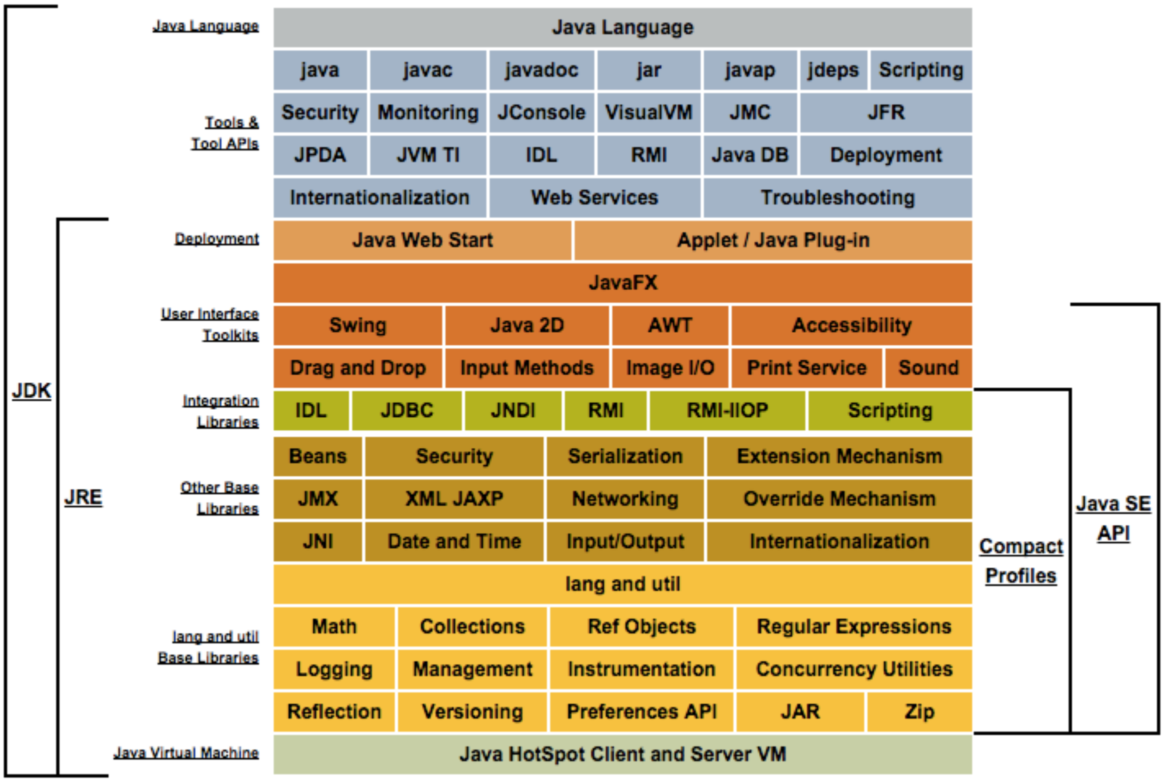

Figure 1.2: ORACLE implementation of JSE. source: http://docs.oracle.com

Figure 1.2 illustrates the ORACLE implementation of JSE. It consists in the Java Software Development Kit (JDK) and Java Runtime Environment (JRE). The JDK is a superset of JRE. The JDK includes all the tools to compile, run, monitor, and debug a program. The JRE is responsible of the execution of the program and includes the Java Virtual Machine (JVM), Figure 1.1. This JSE includes in addition all the tools and API needed to interface the JDK with the java language.

1.2.1 The Java Virtual Machine (JVM)

A JVM is an abstract concept that is defined through its specifications that read a class and perform the operations specified therein.

⊙ **Note:** The JVM is a "virtual machine" that is it aims at emulating the machine (CPU)

It is implemented in a concrete program and is instantiated into an executable program compiled into byte code. The SUN JVM is written in Java (`http://hg.openjdk.java.net/jdk8`). The implementation of a JVM performs the following actions:

- Load. Loading is the process of finding the binary representation of a class or interface type with a particular name and creating a class or interface file from that binary representation. If such class file is not found the the loading throws the exception: *ClassNotFoundException*. A class file consists of a stream of 8-bit bytes.
- Link. Linking verifies and prepares the correctness of a type so that it can be executed.
  - Prepare. Preparing allocates memory for the static field of classes and interfaces and initialise them to their default values.
  - Verify. Verification ensures that the binary representation of a class is structurally correct.
- Resolve. Resolution associate references to run-time values.
- Initialise classes and interfaces. Initialisation defines the values of class or interface variables.

1.2.2 Just In Time Compilation

Historically, interpreters have been considered slow, but because the Java interpreter runs compiled byte-code, Java is a relatively fast language.

As the JVM is available on many operating systems, the same .class can be translated into machine code on Microsoft Windows, the Solaris Operating System, Linux, or Mac OS. In addition, Java can compile byte-code to native machine code on the fly also called *Just-In-Time (JIT) compilation.* The JIT translation consists of having a compiler which generates code for an application (or class library) during execution of the application itself.

The JIT HotSpot VM.

HotSpot `http://openjdk.java.net/groups/hotspot/` is a JVM that uses the JIT translation and performs additional steps at runtime to increase performance. It efficiently manages the Java heap using garbage collectors.

HotSpot implements the JVM Specification, it is written in C/C++ and is delivered as a shared library in JRE.

HotSpot uses *adaptive compilation* to optimise translation. Programs spend almost all of their time executing a relatively small part of the code again and again. The chunk of code that is executed repeatedly may only be a small percent of the total program, but its behaviour determines the program's overall performance. The adaptive compilation first profiles the code while it is executing, to see what parts are being executed repeatedly. It allows the interpreter time to "warm up" Java methods, by executing them thousands of times. It uses a configurable invocation-count threshold to decide which methods to compile. This warm-up period allows a compiler to make better optimisation decisions, because it can observe (after initial class loading) a more complete class hierarchy. Once it know the frequent executed parts, it translates only these into machine code. Since it only compiles a small portion of the bytecode, it can afford to take the time necessary to optimise those portions. The rest of the program may not need to be compiled at all - just interpreted - saving memory and time.

HotSpot can also configure the platform: it will select a compiler, Java heap configuration, and garbage collector that produce good to excellent performance for most applications. Under special circumstances, however, specific tuning may be required to get the best possible performance.

⊙ **Note:** Sun/Oracle claims that with just-in-time compilation, Java code can execute nearly as fast as native compiled code and maintain its transportability and security. There is only one true performance hit that compiled Java code will always suffer for the sake of security: array bounds checking.

⊙ **Quiz:** What does it meant that Java HotSpot compiles just the most executed byte code and the rest is interpreted?

⊙ **Solution:** It translates into machine code the frequent code as a whole and the rest line by line. Then it executes it.

1.3 The Java compiler

Javac is the ORACLE/SUN compiler for java. It translates the java source code into byte code. It reads class and interfaces declarations and translate them into byte codes.

1.4 Java command-line instructions

We can run, compile, and debug a Java file with command-line instructions. To get the options for any of the commands, one types '-help' soon after the command.

1.4.1 Getting started

Open the shell environment in Mac run Terminal.app in Windows run cmd.exe

To get where is the JDK in your file system from anywhere:

```
> which java
/usr/bin/java
```

To get the version of the JDK from anywhere:

```
>  java -version
java version "1.8.0_25"
Java(TM) SE Runtime Environment (build 1.8.0_25-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.25-b02, mixed mode)
```

In Mac, Linux, Solaris:
To display the environment variables

```
> env
```

To see the variables in the PATH
In Mac:

```
> echo $PATH
```

In Windows:

```
> PATH
```

To set your PATH to include the JDK sub-folder named java.
In Mac:

```
export PATH=$PATH:/usr/java/jdk1.6.0_10/bin
```

In Windows: For bash, edit the startup file (/.bashrc):

```
PATH=/usr/local/jdk1.7.0/bin:$PATH
```

```
export PATH
```

See also
https://docs.oracle.com/javase/tutorial/essential/environment/paths.html

⊙ **Note:** If you do not set the PATH variable, you need to specify the full path to the executable every time you run it, such as:

```
C:\Java\jdk1.7.0\bin\javac MyClass.java
```

1.4.2 Compilation: javac

The Java Compiler translates programs written in the Java into bytecode.

⊙ **Example:** with the file 'MyClass.java':

```
javac MyClass.java
```

If the system cannot find javac, check the set path command. If javac runs but it returns errors, check the Java text. If the program compiles, but you get an exception, check the spelling and capitalisation in the file name and the class name as Java is case-sensitive.

⊙ **Example:** Where are my .class files? java creates files in the same folder of the .java files. To specify the path javac uses to look up classes needed to run or being referenced by other classes, invoke:

```
> javac -classpath .;C:/users/dac/classes;C:/tools/java/classes ...
```

1.4.3 Execution: java

The java command executes Java class files created by a Java compiler (e.g., java).

⊙ **Example:** Example with the file 'MyClass.class':

```
> java MyClass
```

1.4.4 Disassembler: javap

One can get the bytecode information contained in a .class file with the command line “javap” as in the following foo.java file:

```
class foo{
    void for99(){
        for(int i=0; i<99; i++){}
    }
    void while99(){
        int i =0;
        while(i<99){
            i++;
        }
    }
}
```

```
> javac foo.java
> javap foo.class
```

```
Compiled from "foo.java"
class foo {
  foo();
  void for99();
  void while99();
}
```

or more verbose:

```
> javap -c Analyzer.class
```

```
Compiled from "foo.java"
class foo {
  foo();
    Code:
       0: aload_0
       1: invokespecial #1 // Method java/lang/Object."<init>":()V
       4: return

  void for99();
    Code:
       0: iconst_0
       1: istore_1
       2: iload_1
       3: bipush      99
       5: if_icmpge   14
       8: iinc        1, 1
      11: goto        2
      14: return

  void while99();
    Code:
       0: iconst_0
       1: istore_1
       2: iload_1
       3: bipush      99
       5: if_icmpge   14
       8: iinc        1, 1
      11: goto        2
      14: return
}
```

⊙ **Note:** It is useful to inspect the output of java. For example, we can see that the two loops are the same in the byte code!

```
> javap Analyzer.class
Compiled from "Analyzer.java"
public abstract class org.apache.lucene.analysis.Analyzer {
  protected boolean overridesTokenStreamMethod;
  static java.lang.Class class$java$lang$String;
  static java.lang.Class class$java$io$Reader;
  public org.apache.lucene.analysis.Analyzer();
  public abstract org.apache.lucene.analysis.TokenStream
      tokenStream(java.lang.String, java.io.Reader);
  public org.apache.lucene.analysis.TokenStream
      reusableTokenStream(java.lang.String, java.io.Reader) throws
      java.io.IOException;
  protected java.lang.Object getPreviousTokenStream();
  protected void setPreviousTokenStream(java.lang.Object);
  protected void setOverridesTokenStreamMethod(java.lang.Class);
  public int getPositionIncrementGap(java.lang.String);
  public int getOffsetGap(org.apache.lucene.document.Fieldable);
  public void close();
  static java.lang.Class class$(java.lang.String);
}
```

The javap command disassembles a class file. Its output depends on the options used. If no options are used, javap prints out the public fields and methods of the classes passed to it and prints its output to stdout (standard output).

With the command

```
javap -c Analyzer
```

one gets the full description of the class file by method:

```
public abstract class org.apache.lucene.analysis.Analyzer
    implements java.io.Closeable {
  static final boolean $assertionsDisabled;

  protected org.apache.lucene.analysis.Analyzer();
    Code:
       0: aload_0
       1: invokespecial #1 // Method java/lang/Object."<init>":()V
       4: aload_0
       5: new           #2   // class
          org/apache/lucene/util/CloseableThreadLocal
       8: dup
       9: invokespecial #3  // Method
          org/apache/lucene/util/CloseableThreadLocal."<init>":()V
      12: putfield      #4  // Field
         tokenStreams:Lorg/apache/lucene/util/CloseableThreadLocal;
      15: getstatic     #5  // Field $assertionsDisabled:Z
      18: ifne          36
      21: aload_0
      22: invokespecial #6 // Method assertFinal:()Z
      25: ifne          36
      28: new           #7  // class java/lang/AssertionError
      31: dup
      32: invokespecial #8  // Method
         java/lang/AssertionError."<init>":()V
      35: athrow
      36: return

  public abstract org.apache.lucene.analysis.TokenStream
      tokenStream(java.lang.String, java.io.Reader);

  public org.apache.lucene.analysis.TokenStream
      reusableTokenStream(java.lang.String, java.io.Reader) throws
      java.io.IOException;
    Code:
       0: aload_0
       1: aload_1
       2: aload_2
       3: invokevirtual #24 // Method tokenStream:...;
       6: areturn

  protected java.lang.Object getPreviousTokenStream();
    Code:
       0: aload_0
```

```
   1: getfield    #4   // Field
       tokenStreams:Lorg/apache/lucene/util/CloseableThreadLocal;
   4: invokevirtual #25 // Method
       org/apache/lucene/util/CloseableThreadLocal.get:()Ljava/lang/Object;
   7: areturn
   8: astore_1
   9: aload_0
  10: getfield    #4  // Field
      tokenStreams:Lorg/apache/lucene/util/CloseableThreadLocal;
  13: ifnonnull   26
  16: new         #27  // class
      org/apache/lucene/store/AlreadyClosedException
  19: dup
  20: ldc         #28   // String this Analyzer is closed
  22: invokespecial #29 // Method
      org/apache/lucene/store/AlreadyClosedException."<init>":(Ljava/lang/String;)V
  25: athrow
  26: aload_1
  27: athrow
 Exception table:
   from   to  target type
      0    7    8   Class java/lang/NullPointerException

protected void setPreviousTokenStream(java.lang.Object);
 Code:
   0: aload_0
   1: getfield    #4   // Field
       tokenStreams:Lorg/apache/lucene/util/CloseableThreadLocal;
   4: aload_1
   5: invokevirtual #30 // Method
       org/apache/lucene/util/CloseableThreadLocal.set:(Ljava/lang/Object;)V
   8: goto        31
  11: astore_2
  12: aload_0
  13: getfield    #4   // Field
      tokenStreams:Lorg/apache/lucene/util/CloseableThreadLocal;
  16: ifnonnull   29
  19: new         #27   // class
      org/apache/lucene/store/AlreadyClosedException
  22: dup
  23: ldc         #28  // String this Analyzer is closed
  25: invokespecial #29 // Method
      org/apache/lucene/store/AlreadyClosedException."<init>":(Ljava/lang/String;)V
  28: athrow
  29: aload_2
```

```
    30: athrow
    31: return
  Exception table:
     from    to  target type
         0     8    11   Class java/lang/NullPointerException

public int getPositionIncrementGap(java.lang.String);
  Code:
     0: iconst_0
     1: ireturn

public int getOffsetGap(org.apache.lucene.document.Fieldable);
  Code:
     0: aload_1
     1: invokeinterface #31, 1 // InterfaceMethod
        org/apache/lucene/document/Fieldable.isTokenized:()Z
     6: ifeq          11
     9: iconst_1
    10: ireturn
    11: iconst_0
    12: ireturn

public void close();
  Code:
     0: aload_0
     1: getfield      #4  // Field
        tokenStreams:Lorg/apache/lucene/util/CloseableThreadLocal;
     4: invokevirtual #32 // Method
        org/apache/lucene/util/CloseableThreadLocal.close:()V
     7: aload_0
     8: aconst_null
     9: putfield      #4  // Field
        tokenStreams:Lorg/apache/lucene/util/CloseableThreadLocal;
    12: return

static {};
  Code:
     0: ldc_w         #33  // class
        org/apache/lucene/analysis/Analyzer
     3: invokevirtual #10 // Method
        java/lang/Class.desiredAssertionStatus:()Z
     6: ifne          13
     9: iconst_1
    10: goto          14
    13: iconst_0
```

```
      14: putstatic     #5   // Field $assertionsDisabled:Z
      17: return
}
```

1.4.5 Debugger: jdb

The Java Debugger helps find and fix bugs in Java programs. There are two major categories of errors: compiler errors and execution (or run-time) errors and then there are failures. In the following we will see compiler errors and failures. In the following, we learn how to discover them.

⊙ **Example:** The following file DatesBuggy.java contains some syntax errors[1]:

---

[1] Thank to prof. Mary K. Vernon http://pages.cs.wisc.edu/ vernon/cs367/tutorials/jdb.tutorial.html

```
import java.io.*;
class DatesBuggy {
public static int daysInMonth (int month) {
    if (month == 9) || (month == 4) || (month == 6) || (month == 11)) {
        return 30;
    }
    else if (month == 2)
         return 28;
    else return 31;
}
public static void main (String[] args) {
    int someMonth, someDay;
    int laterMonth, laterDay;
    int aMonth;
    someMonth = Integer.parseInt(args[0]);
    someDay = Integer.parseInt(args[1]);
    laterMonth = Integer.parseInt(args[2]);
    laterDay = Integer.parseInt(args[3]);
    /* Used to record what day in the year the first day */
    /* of someMonth and laterMonth are. */
    int someDayInYear = 0;
    int laterDayInYear = 0;
    for (aMonth = 0, aMonth < someMonth; aMonth = aMonth + 1) {
        someDayInYear = someDayInYear + daysInMonth(aMonth);
    }
    for (aMonth = 1; aMonth < laterMonth; aMonth = aMonth + 1) {
        laterDayInYear = laterDayInYear + daysInMonth(aMonth);
    }
    /* The answer */
    int daysBetween = 0;
    System.out.println("The difference in days between " +
                       someMonth + "/" + someDay + " and " +
                       laterMonth + "/" + laterDay + " is: ");
    daysBetween = laterDayInYear - someDayInYear;
    daysBetween = daysBetween + laterDay - someDay;
    System.out.println(daysBetween);
}
}
```

Compile the file with javac:

```
> javac DatesBuggy.java
DatesBuggy.java:6: error: illegal start of expression
    if (month == 9) || (month == 4) || (month == 6) || (month ==
        11)) {
                  ^

DatesBuggy.java:6: error: not a statement
    if (month == 9) || (month == 4) || (month == 6) || (month ==
        11)) {
                                              ^

DatesBuggy.java:6: error: ';' expected
    if (month == 9) || (month == 4) || (month == 6) || (month ==
        11)) {
                                                            ^

DatesBuggy.java:9: error: 'else' without 'if'
    else if (month == 2)
    ^
DatesBuggy.java:29: error: not a statement
    for (aMonth = 0, aMonth < someMonth; aMonth = aMonth + 1) {
                         ^
DatesBuggy.java:29: error: ';' expected
    for (aMonth = 0, aMonth < someMonth; aMonth = aMonth + 1) {
                                                     ^

6 errors
```

Look at the code and notice that:
Compiler Error1: The code lacks a parenthesis in the first if condition.
Compiler Error2: The code lacks a semicolon in the first for condition.

Fix the compiler errors, compile and run it again:

```
>  javac DatesBuggy.java
>  java DatesBuggy 1 12 3 4
The difference in days between 1/12 and 3/4 is:
20
```

Now you do not get and error but wrong answer!
Failure1: number of days is wrong. (Correct output is 51. Where is the error in the code?)
Recompile the program with the '-g' option to tell the compiler to provide information that jdb can use to display local (stack) variables:

```
>javac -g DatesBuggy.java
```

Run your program by having jdb start the Java interpreter:

```
>jdb DatesBuggy 1 12 3 4
```

At this point, jdb has invoked the Java interpreter, the DatesBuggy class is loaded, and the interpreter stops before entering main().
Give the command ‘stop in DatesBuggy.main’ and then ‘run’. The interpreter will continue executing for a very short time until just after it enters main().

```
> jdb DatesBuggy 1 12 3 4
Initializing jdb ...
> stop in DatesBuggy.main
Deferring breakpoint DatesBuggy.main.
It will be set after the class is loaded.
> run
run DatesBuggy 1 12 3 4
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint DatesBuggy.main

Breakpoint hit: "thread=main", DatesBuggy.main(), line=18 bci=0
18        someMonth = Integer.parseInt(args[0]);
main[1]
```

Type list to see the source code for the instructions that are about to execute, or you can type print args to see the value of the variable called ”args”

```
main[1] list
14        int someMonth, someDay;
15        int laterMonth, laterDay;
16
17        int aMonth;
18 =>     someMonth = Integer.parseInt(args[0]);
19        someDay = Integer.parseInt(args[1]);
20
21        laterMonth = Integer.parseInt(args[2]);
22        laterDay = Integer.parseInt(args[3]);
23
```

One tricky point about jdb is that if you use the step command to execute one instruction at a time, you will step into the instructions for the method called Integer.parseInt. The source code for predefined Java classes is not available to jdb, so jdb cannot list the lines of code or print the values of any variables in that method. Thus, you should set a breakpoint after the arguments are parsed, using the command stop at DatesBuggy:30 and then type ‘cont’ to let the interpreter continue executing until it again reaches a breakpoint (i.e., until it is about to execute line 24). You will get null after the command locals or print below if you did not run the compiler java with ”-g”.

```
> jdb DatesBuggy 1 12 3 4
Initializing jdb ...
> stop at DatesBuggy:24
Deferring breakpoint DatesBuggy:241.
It will be set after the class is loaded.
> run
run DatesBuggy 1 12 3 4
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint DatesBuggy:24

Breakpoint hit: "thread=main", DatesBuggy.main(), line=24 bci=44
24          someDayInYear = someDayInYear + daysInMonth(aMonth);

main[1] print laterMonth
laterMonth = 3
main[1] locals
Method arguments:
args = instance of java.lang.String[4] (id=439)
Local variables:
someMonth = 1
someDay = 12
laterMonth = 3
laterDay = 4
someDayInYear = 0
laterDayInYear = 0
aMonth = 0
main[1]
```

One should continue to examine the program’s behaviour as it executes by setting further breakpoints, or using **step** to execute one instruction at a time. At each breakpoint, use the print or locals command to examine the values of program variables, until you isolate the error. Note that when the method called daysInMonth is called, jdb can stop at a breakpoint in that method (e.g., if you say stop in DatesBuggy.daysInMonth) and it can list the code or print variable values in that method. The error in DatesBuggy can be corrected by changing only ONE line.

When you think you have found the error: copy the file to another file in case you need to use it later, correct the error, and recompile and execute it to see if the problem is solved. Type exit to exit the debugging.