Java Programming/Print version

Overview

Preface

> The beautiful thing about learning is nobody can take it away from you.
> —B.B. King (5 October 1997)

Learning a computer programming language is like a toddler's first steps. You stumble, and fall, but when you start walking, programming becomes second nature. And once you start programming, you never cease evolving or picking up new tricks. Learn one programming language, and you will "know" them all — the logic of the world will begin to unravel around you.

Are you new to programming?

This stunning image of the sunset on planet Mars wouldn't have been possible without Java.

If you have chosen Java as your first programming language, be assured that Java is also the first choice for computer science programs in many universities. Its simple and intuitive syntax, or grammar, helps beginners feel at ease with complex programming constructs quickly.

However, Java is not a *basic* programming language. In fact, NASA used Java as the driving force (quite literally) behind its Mars Rover missions. Robots, air traffic control systems and the self-checkout barcode scanners in your favorite supermarkets can all be programmed in Java.

Programming with Java™

By now, you might truly be able to grasp the power of the Java programming language. With Java, there are many possibilities. Yet not every programmer gets to program applications that take unmanned vehicles onto other planets. Software that we encounter in our daily life is somewhat humble in that respect. Software in Java, however, covers a vast area of the computing ecosphere. Here are just a few examples of the ubiquitous nature of Java applications in real-life:

- OpenOffice.org, a desktop office management suite that rivals the Microsoft Office suite has been written in Java.
- The popular building game Minecraft is written in Java.
- Online browser-based games like Runescape, a 3D massively multi-player online role playing game (MMORPG), run on graphics routines, 3D rendering and networking capabilities powered by the Java programming language.
- Two of the world's renowned digital video recorders, TiVo and BSkyB's Sky+ use built-in live television recording software to record, rewind and play your favorite television shows. These applications make extensive use of the Java programming language.

The above mentioned applications illustrate the reach and ubiquity of Java applications. Here's another fact: almost 80% of mobile phone vendors adopt Java as their primary platform for the development of applications. The most widely used mobile-based operating system, Android, uses Java as one of its key application platforms — developers are encouraged to develop applications for Android in the Java programming language.

What can Java *not* do?

Well, to be honest, there is nothing that Java can't do, at least for application programming. Java is a "complete" language; the only limits are programmer imagination and ability. This book aims to get you acquainted with the basics of the language so you can create the software masterpiece of your dreams. The one area where Java can't be

used is for direct interaction with computer hardware. If you want to write an operating system, you will need to look elsewhere!

About This Book

The Java Programming Wikibook is a shared effort in amassing a comprehensive guide of the complete Java platform — from programming advice and tutorials for the desktop computer to programming on mobile phones. The information presented in this book has been conceptualised with the combined efforts of various contributors, and anonymous editors.

The primary purpose of this book is to teach the Java programming language to an audience of beginners, but with its progressive layout of tutorials increasing in complexity, it can be just as helpful for intermediate and experienced programmers. Thus, this book is meant to be used as:

- a collection of tutorials building upon one another in a progressive manner;
- a guidebook for efficient programming with the Java programming language; and,
- a comprehensive manual resource for the advanced programmer.

This book is intended to be used in conjunction with various other online resources, such as:

- the Java platform API documentation (http://download.oracle.com/javase/8/docs/api/overview-summary.html);
- the official Java website (http://www.oracle.com/us/technologies/java/index.html); and,
- active Java communities online, such as Java.net (http://home.java.net) and JavaRanch (http://www.javaranch.com), etc.

Who should read this book?

Everything you would need to know to write computer programs would be explained in this book. By the time you finish reading, you will find yourself proficient enough to tackle just about anything in Java and programs written using it. This book serves as the first few stepping stones of many you would need to cross the unfriendly waters of computer programming. We have put a lot of emphasis in structuring this book in a way that lets you start programming from scratch, with Java as your preferred language of choice. This book is designed for you if any one of the following is true.

- You are relatively new to programming and have heard how easy it is to learn Java.
- You had some BASIC or Pascal in school, and have a grasp of basic programming and logic.
- You already know and have been introduced to programming in earlier versions of Java.
- You are an experienced developer and know how to program in other languages like C++, Visual Basic, Python, Ruby, etc.
- You've heard that Java is great for web applications and web services programming.

Although this book is generally meant to be for readers who are beginning to learn programming, it can be highly beneficial for intermediate and advanced programmers who may have missed out on some vital information. After completing this book you should be able to solve many complicated problems using the Java skills presented in the following chapters. Once you finish, you are also encouraged to undertake ambitious programming projects of your own.

This book assumes that the reader has no prior knowledge of programming in Java, or for that matter, any object-oriented programming language. Practical examples and exercises following each topic and module make it easy to understand the software development methodology. If you are a complete beginner, we suggest that you move slowly through this book and complete each exercise at your own pace.

How to use this book

This book is a reference book of the Java language and its related technologies. Its goal is to give a complete picture of Java and its technologies. While the book can be read from the beginning to end, it is also designed to have individual sections that can be read independently. To help find information quickly, navigation boxes are given in the online version for access to individual topics.

This book is divided into sections. Pages are grouped together into section topics. To make this book expandable in the future via the addition of new sections, the sections navigation-wide are independent from each other. Each section can be considered as a mini book by itself. Pages that belong to the same topic can be navigated by the links on the right hand side.

How can you participate

Content is constantly being updated and enhanced in this book as is the nature of wiki-based content. This book is therefore in a constant state of evolution. Any Wikibooks users can participate in helping this book to a better standard as both a reader, or a contributor.

As a reader

If you are interested in reading the content present in this book, we encourage you to:

- share comments about the technical accuracy, content, or organization of this book by telling the contributors in the **Discussion** section for each page. You can find the link **Discussion** on each page in this book leading you to appropriate sections for discussion. Leave a signature when providing feedback, writing comments, or giving suggestion on the **Discussion** pages. This can be achieved by appending `--` `~~~~` to your messages. Do *not* add your signatures to the **Book** pages, they are only meant for the **Discussion** pages.
- share news about the Java Programming Wikibook with your family and friends and let them know about this comprehensive Java guide online.
- become a contributing author, if you think that you have information that could fill in some missing gaps in this book.

As a contributor

If you are intent on writing content for this book, you need to do the following:

- When writing content for this book, you can always pose as an anonymous contributor, however we recommend you sign-in into the Wikibooks website when doing so. It becomes easier to track and acknowledge changes to certain parts of the book. Furthermore, the opinions and views of logged-in users are given precedence over anonymous users.
- Once you have started contributing content for this book, make sure that you add your name to the contributor list.
- Be bold and try to follow the conventions for this Wikibook. It is important that the conventions for this book be followed to the letter to make content consistent and reliable throughout.

History

On 23 May 1995, John Gage, the director of the Science Office of the Sun Microsystems along with Marc Andreesen, co-founder and executive vice president at Netscape announced to an audience of SunWorld™ that Java technology wasn't a myth and that it was going to be incorporated into Netscape Navigator.[1]

At the time the total number of people working on Java was less than 30.[1] This team would shape the future in the next decade and no one had any idea as to what was in store. From running an unmanned vehicle on Mars to serving as the operating environment of most consumer electronics, e.g. cable set-top boxes, VCRs, toasters and PDAs,[2] Java has come a long way from its inception. Let's see how it all began.

Earlier programming languages

Before Java emerged as a programming language, C++ was the dominant player in the trade. The primary goal of the creators of Java was to create a language that could tackle most of the things that C++ offered while getting rid of some of the more tedious tasks that came with the earlier languages.

Computer hardware went through a performance and price revolution from 1972 to 1991. Better, faster hardware was available at ever lower prices, and the demand for big and complex software exponentially increased. To accommodate the demand, new development technologies were invented.

The C language developed in 1972 by Dennis Ritchie had taken a decade to become the most popular language amongst programmers working on PCs and similar platforms (other languages, like COBOL and FORTRAN, dominated the mainframe market). But, with time programmers found that programming in C became tedious with its structural syntax.[3] Although people attempted to solve this problem, it would be later that a new development philosophy was introduced, one named *Object-Oriented Programming* (OOP). With OOP, one can write code that can be reused later without needing to rewrite the code over and over again. In 1979, Bjarne Stroustrup developed C++, an enhancement to the C language with included OOP fundamentals and features. Sun generated revenue from Java through the selling of licenses for specialized products such as the Java Enterprise System.

The Green team

In December 1990, a project was initiated behind closed doors with the aim to create a programming tool that could render obsolete the C and C++ programming languages. Engineer Patrick Naughton had become extremely frustrated with the state of Sun's C++ and C APIs (Application Programming Interfaces) and tools. While he was considering to move towards NeXT, he was offered a chance to work on new technology and the *Stealth Project* was started, a secret nobody but he knew.


James Gosling, architect and designer of the compiler for the Java technology

This Stealth Project was later named the *Green Project* when James Gosling and Mike Sheridan joined Patrick.[1] As the Green Project teethed, the prospects of the project started becoming clearer to the engineers working on it. No longer did it aim to create a new language far superior to the present ones, but it aimed to target devices other than the computer.

Staffed at 13 people, they began work in a small office on Sand Hill Road in Menlo Park, California. This team came to be called the *Green Team* henceforth in time. The project they underwent was chartered by Sun Microsystems to anticipate and plan for the "next wave" in computing. For the team, this meant at least one significant trend, that of the convergence of digitally controlled consumer devices and computers.[1]

Reshaping thought

The team started thinking of replacing C++ with a better version, a faster version, a responsive version. But the one thing they hadn't thought of, as of yet, was that the language they were aiming for had to be developed for an embedded system with limited resources. An **embedded system** is a computer system scaled to a minimalistic interface demanding only a few functions from its design. For such a system, C++ or any successor would seem too large as all the languages at the time demanded a larger footprint than what was desired. The team thus had to think in a different way to go about solving all these problems.

Co-founder of Sun Microsystems, Bill Joy, envisioned a language combining the power of Mesa and C in a paper named *Further* he wrote for the engineers at Sun. Gathering ideas, Gosling began work on enhancing C++ and named it "C++ ++ --", a pun on the evolutionary structure of the language's name. The ++ and -- meant, *putting in* and *taking out stuff*. He soon abandoned the name and called it **Oak**[1] after the tree that stood outside his office.

Table 1: Who's who of the Java technology[1]
Has worked for GT (Green Team), FP (FirstPerson) and JP (Java Products Group)

Name	GT	FP	JP	Details
Lisa Friendly		✓	✓	FirstPerson employee and member of the Java Products Group
John Gage				Science Office (Director), Sun Microsystems
James Gosling	✓	✓	✓	Lead engineer and key architect of the Java technology
Bill Joy				Co-founder and VP, Sun Microsystems; Principal designer of the UC Berkeley, version of the UNIX® OS
Jonni Kanerva		✓		Java Products Group employee, author of The Java FAQ1
Tim Lindholm		✓	✓	FirstPerson employee and member Java Products Group
Scott McNealy				Chairman, President, and CEO of Sun Microsystems
Patrick Naughton	✓	✓		Green Team member, FirstPerson co-founder
George Paolini				Corporate Marketing (Director), Sun's Java Software Division
Kim Polese		✓		FirstPerson product marketing
Lisa Poulson				Original director of public relations for Java technology (Burson-Marsteller)
Wayne Rosing		✓		FirstPerson President
Eric Schmidt				Former Sun Microsystems Chief Technology Officer
Mike Sheridan	✓			Green Team member

The demise of an idea, birth of another

By now, the work on Oak had been significant but come the year 1993, people saw the demise of set-top boxes, interactive TV and the PDAs. A failure that completely ushered the inventors' thoughts to be reinvented. Only a miracle could make the project a success now. And such a miracle awaited anticipation.

National Center for Supercomputing Applications (NCSA) had just unveiled its new commercial web browser for the internet the previous year. The focus of the team, now diverted towards where they thought the "next-wave" of computing would be — the internet. The team then divulged into the realms of creating the same embeddable technology to be used in the web browser space calling it an **applet** — *a small application*. Keeping all of this in mind, the team created a list of features tackling the C++ problems. In their opinion, the project should ...

- .. be simple and gather tested fundamentals and features from the earlier languages in it,
- .. have standard sets of APIs with basic and advanced features bundled with the language,
- .. get rid of concepts requiring direct manipulation of hardware (in this case, memory) to make the language safe,
- .. be platform independent and may be written for every platform once (giving birth to the WORA idiom),
- .. be able to manipulate network programming out-of-the-box,
- .. be embeddable in web browsers, and ...
- .. have the ability for a single program to multi-task and do multiple things at the same time.

The team now needed a proper identity and they decided on naming the new technology they created **Java** ushering a new generation of products for the internet boom. A by-product of the project was a cartoon named "Duke" created by Joe Parlang which became its identity then.

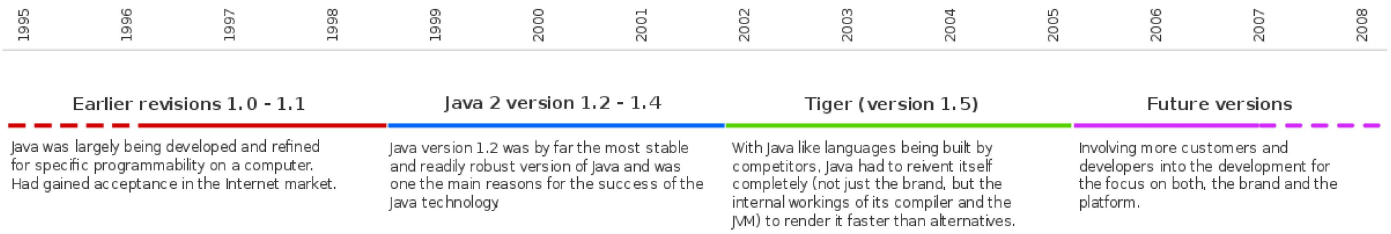
Finally at the SunWorld™ conference, Andreesen unveiled the new technology to the masses. Riding along with the explosion of interest and publicity in the Internet, Java quickly received widespread recognition and expectations grew for it to become the dominant software for browser and consumer applications.[2]

Initially Java was owned by Sun Microsystems, but later it was released to open source; the term Java was a trademark of Sun Microsystems. Sun released the source code for its HotSpot Virtual Machine and compiler in November 2006, and most of the source code of the class library in May 2007. Some parts were missing because they

were owned by third parties, not by Sun Microsystems. The released parts were published under the terms of the GNU General Public License, a free software license.

Versions

Unlike C and C++, Java's growth is pretty recent. Here, we'd quickly go through the development paths that Java took with age.



Development of Java over the years. From version 1.0 to version 1.7, Java has displayed a steady growth.

Initial Release (versions 1.0 and 1.1)

Introduced in 1996 for the Solaris, Windows, Mac OS Classic and Linux, Java was initially released as the Java Development Kit 1.0 (JDK 1.0). This included the Java runtime (the virtual machine and the class libraries), and the development tools (e.g., the Java compiler). Later, Sun also provided a runtime-only package, called the Java Runtime Environment (JRE). The first name stuck, however, so usually people refer to a particular version of Java by its JDK version (e.g., JDK 1.0).

Java 2 (version 1.2)

Introduced in 1998 as a quick fix to the former versions, version 1.2 was the start of a new beginning for Java. The JDKs of version 1.2 and later versions are often called *Java 2* as well. For example, the official name of JDK 1.4 is *The Java(TM) 2 Platform, Standard Edition version 1.4*.

Major changes include:

- Rewrite the event handling (add Event Listeners)
- Change Thread synchronizations
- Introduction of the JIT-Just in time compilers

Kestrel (Java 1.3)

Released on 8 May 2000. The most notable changes were:

- HotSpot JVM included (the HotSpot JVM was first released in April, 1999 for the J2SE 1.2 JVM)
- RMI was modified to support optional compatibility with CORBA
- JavaSound
- Java Naming and Directory Interface (JNDI) included in core libraries (previously available as an extension)
- Java Platform Debugger Architecture (JPDA)
- Synthetic proxy classes

Merlin (Java 1.4)

Released on 6 February 2002, Java 1.4 has improved programmer productivity by expanding language features and available APIs:

- Assertion
- Regular Expression
- XML processing
- Cryptography and Secure Socket Layer (SSL)

- Non-blocking I/O (NIO)
- Logging

Tiger (version 1.5.0; Java SE 5)

Released in September 2004

Major changes include:

- Generics - Provides compile-time type safety for collections :and eliminates the drudgery of casting.
- Autoboxing/unboxing - Eliminates the drudgery of manual conversion between primitive types (such as int) and wrapper types (such as Integer).
- Enhanced for - Shorten the for loop with Collections use.
- Static imports - Lets you import all the static part of a class.
- Annotation/Metadata - Enabling tools to generate code and deployment descriptors from annotations in the source code. This leads to a "declarative" programming style where the programmer says what should be done and tools emit the code to do it. Annotations can be inspected through source parsing or by using the additional reflection APIs added in Java 5.
- JVM Improvements - Most of the run time library is now mapped into memory as a memory image, as opposed to being loaded from a series of class files. Large portion of the runtime libraries will now be shared among multiple JVM instances.

Mustang (version 1.6.0; Java SE 6)

Released on 11 December 2006.

What's New in Java SE 6:

- Web Services - First-class support for writing XML web service client applications.
- Scripting - You can now mix in JavaScript technology source code, useful for prototyping. Also useful when you have teams with a variety of skill sets. More advanced developers can plug in their own scripting engines and mix their favorite scripting language in with Java code as they see fit.
- Database - No more need to find and configure your own JDBC database when developing a database application. Developers will also get the updated JDBC 4.0, a well-used API with many important improvements, such as special support for XML as an SQL datatype and better integration of Binary Large OBjects (BLOBs) and Character Large OBjects (CLOBs) into the APIs.
- More Desktop APIs - GUI developers get a large number of new tricks to play like the ever popular yet newly incorporated SwingWorker utility to help you with threading in GUI apps, JTable sorting and filtering, and a new facility for quick splash screens to quiet impatient users.
- Monitoring and Management - The really big deal here is that you don't need to do anything special to the startup to be able to attach on demand with any of the monitoring and management tools in the Java SE platform.
- Compiler Access - Really aimed at people who create tools for Java development and for frameworks like JavaServer Pages (JSP) or Personal Home Page construction kit (PHP) engines that need to generate a bunch of classes on demand, the compiler API opens up programmatic access to javac for in-process compilation of dynamically generated Java code. The compiler API is not directly intended for the everyday developer, but for those of you deafened by your screaming inner geek, roll up your sleeves and give it a try. And the rest of us will happily benefit from the tools and the improved Java frameworks that use this.
- Pluggable Annotations allows programmer to write annotation processor so that it can analyse your code semantically before javac compiles. For example, you could write an annotation processor that verifies whether your program obeys naming conventions.
- Desktop Deployment - At long last, Java SE 6 unifies the Java Plug-in technology and Java WebStart engines, which just makes sense. Installation of the Java WebStart application got a much needed makeover.
- Security - Java SE 6 has simplified the job of its security administrators by providing various new ways to access platform-native security services, such as native Public Key Infrastructure (PKI) and cryptographic services on Microsoft Windows for secure authentication and communication, Java Generic Security Services (Java GSS) and Kerberos services for authentication, and access to LDAP servers for authenticating users.

- The -lities: Quality, Compatibility, Stability - Bug fixes ...

Dolphin (version 1.7.0; Java SE 7)

Released on 28 July 2011.

Feature additions for Java 7 include:

- JVM support for dynamic languages, following the prototyping work currently done on the Multi Language Virtual Machine
- Compressed 64-bit pointers Available in Java 6 with -XX:+UseCompressedOops (http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html)
- Small language changes (grouped under a project named Coin):
  - Strings in switch
  - Automatic resource management in try-statement
  - Improved type inference for generic instance creation
  - Simplified varargs method declaration
  - Binary integer literals
  - Allowing underscores in numeric literals
  - Catching multiple exception types and rethrowing exceptions with improved type checking
- Concurrency utilities under JSR 166
- New file I/O library to enhance platform independence and add support for metadata and symbolic links. The new packages are java.nio.file and java.nio.file.attribute
- Library-level support for Elliptic curve cryptography algorithms
- An XRender pipeline for Java 2D, which improves handling of features specific to modern GPUs
- New platform APIs for the graphics features originally planned for release in Java version 6u10
- Enhanced library-level support for new network protocols, including SCTP and Sockets Direct Protocol
- Upstream updates to XML and Unicode

Lambda (Java's implementation of lambda functions), Jigsaw (Java's implementation of modules), and part of Coin were dropped from Java 7.

Spider (version 1.8.0; Java SE 8)

Java 8 was released on 18 March 2014, and included some features that were planned for Java 7 but later deferred.

Work on features was organized in terms of JDK Enhancement Proposals (JEPs).

- JSR 335, JEP 126: Language-level support for lambda expressions (officially, lambda expressions; unofficially, closures) under Project Lambda which allow the addition of methods to interfaces without breaking existing implementations. There was an ongoing debate in the Java community on whether to add support for lambda expressions. Supporting lambda expressions also allows the performance of functional-style operations on streams of elements, such as MapReduce-inspired transformations on collections. Default methods allow an author of an API to add new methods to an interface without breaking the old code using it. Although it was not their primary intent, default methods also allow multiple inheritance of behavior (but not state).
- JSR 223, JEP 174: Project Nashorn, a JavaScript runtime which allows developers to embed JavaScript code within applications
- JSR 308, JEP 104: Annotation on Java Types
- Unsigned Integer Arithmetic
- JSR 337, JEP 120: Repeating annotations
- JSR 310, JEP 150: Date and Time API
- JEP 178: Statically-linked JNI libraries
- JEP 153: Launch JavaFX applications (direct launching of JavaFX application JARs)
- JEP 122: Remove the permanent generation

References

1. "Java Technology: The Early Years". Sun Microsystems. http://java.sun.com/features/1998/05/birthday.html. Retrieved 9 May 2008.
2. "History of Java". Lindsey, Clark S.. http://www.particle.kth.se/~lindsey/JavaCourse/Book/Part1/Java/Chapter01/history.html. Retrieved 7 May 2008.
3. Structural syntax is a linear way of writing code. A program is interpreted usually at the first line of the program's code until it reaches the end. One cannot hook a later part of the program to an earlier one. The flow follows a linear top-to-bottom approach.

Java Overview

The new features and upgrades included into Java changed the face of programming environment and gave a new definition to Object Oriented Programming (*OOP* in short). But unlike its predecessors, Java needed to be bundled with standard functionality and be independent of the host platform.

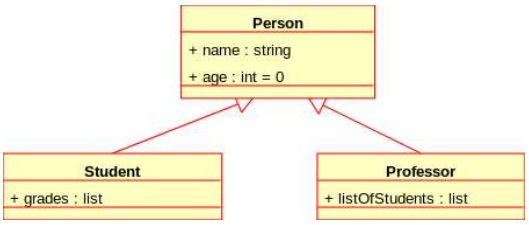
The primary goals in the creation of the Java language:

- It is simple.
- It is object-oriented.
- It is independent of the host platform.
- It contains language facilities and libraries for networking.
- It is designed to execute code from remote sources securely.

The Java language introduces some new features that didn't exist in other languages like C and C++.

Object orientation

Object orientation ("OO") refers to a method of programming and language technique. The main idea of OO is to design software around the "things" (i.e. objects) it manipulates, rather than the actions it performs.



Object oriented programming can be represented with UML diagrams.

As the hardware of the computer advanced, it brought about the need to create better software techniques to be able to create ever increasing complex applications. The intent is to make large software projects easier to manage, thus improving quality and reducing the number of failed projects. Object oriented solution is the latest software technique.

Assembly languages

Software techniques started with the assembly languages, that were close to machine instruction and were easy to convert into executable code. Each hardware had its own assembly language. Assembly language contains low level instructions like move data from memory to hardware registers, do arithmetic operations, and move data back to memory. Programmers had to know the detailed architecture of the computer in order to write programs.

Procedural languages

After the assembly languages, high level languages were developed. Here the language compiler is used to convert the high level program to machine instructions, freeing the programmers from the burden of knowing the computer hardware architecture. To promote the re-use of code and to minimize the use of GOTO instructions, "procedural" techniques were introduced. This simplified the creation and maintenance of software control flow, but left out the organization of data. It became a nightmare to debug and maintain programs having many global variables (i.e. variables that contain data that can be modified anywhere in the application).

Object oriented languages

In OO languages, data is taken seriously with information hiding. Data that is specific to an object can only be accessed by procedures in that object. As a result, objects contain data as well as control flow and a program becomes a series of interactions between objects.

Platform dependence

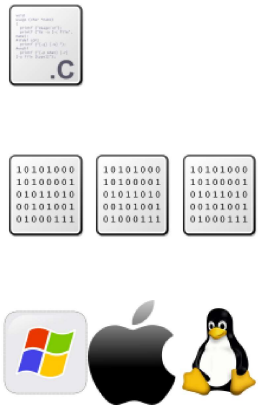
In C or C++ programming, you start to write source code:

... you compile it to a machine code file:

... and then you execute it:

In this situation, the machine code file and its execution are specific to the platform (Windows, Linux, macOS, ...) it was compiled for, that is to say to the *targeted platform*:

... because the compiled file is a machine code file designed to work on a specific platform and hardware. It would have produced different results/output for another platform. So if you want your program to run on several platforms, you have to compile your program several times:



It poses greater vulnerability risks. Note here that when a certain code is compiled into an executable format, the executable cannot be changed dynamically. It would need to be recompiled from the changed code for the changes to be reflected in the finished executable. **Modularity** (dividing code into modules) is *not* present in Java's predecessors. If instead of a single executable, the output application was in the form of modules, one could easily change a single module and review changes in the application. In C/C++ on the other hand, a slight change in code required the whole application to be recompiled.

The idea of Java is to compile the source code into an intermediate language that will be interpreted.

The source code The intermediate file The interpreter

The intermediate language is the *byte code*. The interpreter is the *Java Virtual Machine* (JVM). The byte code file is universal and the JVM is platform specific:

So a JVM should be coded for each platform. And that's the case. So you just have to generate a unique byte code file (a `.class` file).

The first implementations of the language used an interpreted virtual machine to achieve portability, and many implementations still do. These implementations produce programs that run more slowly than the fully-compiled programs created by the typical C++ compiler, so the language suffered a reputation for producing slow programs. Since Java 1.2, Java VM produces programs that run much faster, using multiple techniques.

The first of these is to simply compile directly into native code like a more traditional compiler, skipping bytecode entirely. This achieves great performance, but at the expense of portability. This is not really used any more.

Another technique, the *just-in-time* (JIT) compiler, compiles the Java bytecode into native code at the time the program is run, and keep the compiled code to be used again and again. More sophisticated VMs even use *dynamic recompilation*, in which the VM can analyze the behavior of the running program and selectively recompile and optimize critical parts of the program. Both of these techniques allow the program to take advantage of the speed of native code without losing portability.

Portability is a technically difficult goal to achieve, and Java's success at that goal is a matter of some controversy. Although it is indeed possible to write programs for the Java platform that behave consistently across many host platforms, the large number of available platforms with small errors or inconsistencies led some to parody Sun's "Write once, run anywhere" slogan as "Write once, debug everywhere".

Standardization

C++ was built atop the C language and as a result divergent ways of doing the same thing manifested around the language. For instance, creating an object could be done in three different ways in C++. Furthermore, C++ did not come with a standard library bundled with its compilers. Instead, it relied on resources created by other programmers; code which rarely fit together.

In Java, standardized libraries are provided to allow access to features of the host machines (such as graphics and networking) in unified ways. The Java language also includes support for multi-threaded programs—a necessity for many networking applications.

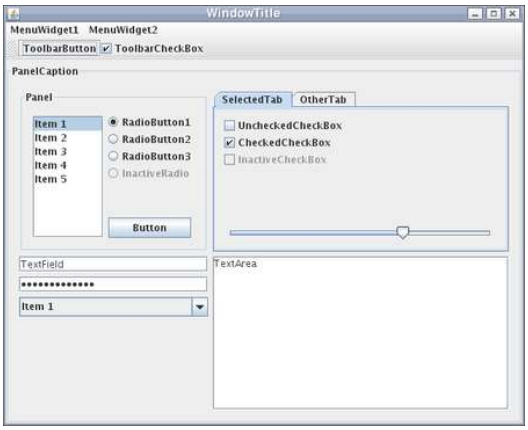
Platform independent Java is, however, very successful with server side applications, such as web services, servlets, or Enterprise JavaBeans.

Java also made progress on the client side: first it had Abstract Window Toolkit (AWT), then Swing, and the most recent client side library is the Standard Widget Toolkit (SWT). It is interesting to see how they tried to handle the two opposing consuming forces. Those are :

Efficient, fast code; port to most popular hardware (write once, test anywhere)
: Use the underlying native subroutine to create a GUI component. This approach was taken by AWT, and SWT.

Portability to any hardware where JVM ported (write once, run anywhere)
: To achieve this to the latter, the Java toolkit should not rely on the underlying native user interface. Swing took this approach.

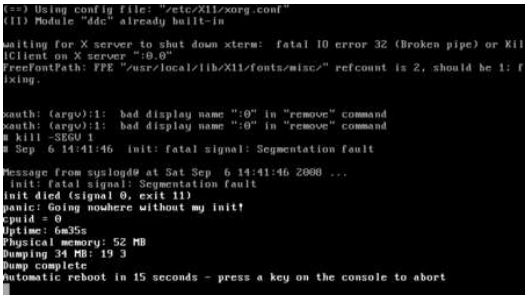
It is interesting to see how the approach was switched back and forth. AWT → Swing → SWT.



Swing does not rely on the underlying native user interface.

Secure execution

With the high-level of control built into the language to manipulate hardware, a C/C++ programmer could access almost any resource, either hardware or software on the system. This was intended to be one of the languages' strong points, but this very flexibility led to confusion and complex programming practices.



The segmentation fault, one of the most recurrent issues in C programming.

Error handling

The old way of error handling was to let each function return an error code then let the caller check what was returned. The problem with this method was that if the return code was full of error-checking codes, this got in the way of the original one that was doing the actual work, which in turn did not make it very readable.

In the new way of error handling, functions/methods do not return error codes. Instead, when there is an error, an exception is thrown. The exceptions can be handled by the **`catch`** keyword at the end of a **`try`** block. This way, the code that is calling the function does not need to be mangled with error checking codes, thus making the code more readable. This new way of error handling is called *Exception handling*.

Exception handling was also added to C++. However, there are two differences between Java and C++ Exception handling:

- In Java, the exception that is thrown is a Java object like any other object in Java. It only has to implement `Throwable` interface.
- In Java, the compiler checks whether an exception may be caught or not. The compiler gives an error if there is no catch block for a thrown exception.

The optional exception handling in the Java predecessors leads the developers not to care about the error handling. As a consequence, unexpected errors often occur. Java forces the developers to handle exceptions. The programmer must handle exception or declare that the user must handle it. Someone must handle it.

Networking capabilities

However powerful, the predecessors of Java lacked a standard feature to network with other computers, and usually relied on the platforms' intricate networking capabilities. With almost all network protocols being standardized, the creators of Java technology wanted this to be a flagship feature of the language while keeping true to the spirit of earlier advances made towards standardizing Remote Procedure Call. Another feature that the Java team focused on was its integration in the World Wide Web and the Internet.

The Java platform was one of the first systems to provide wide support for the execution of code from remote sources. The Java language was designed with network computing in mind.

An applet could run within a user's browser, executing code downloaded from a remote HTTP server. The remote code runs in a highly restricted "sandbox", which protects the user from misbehaving or malicious code; publishers could apply for a certificate that they could use to digitally sign applets as "safe", giving them permission to break out of the sandbox and access the local file system and network, presumably under user control.

Dynamic class loading

In conventional languages like C and C++, all code had to be compiled and linked to one executable program, before execution. In Java, classes are compiled as needed. If a class is not needed during an execution phase, that class is not even compiled into byte code.

This feature comes in handy especially in network programming when we do not know, beforehand, what code will be executed. A running program could load classes from the file system or from a remote server.

Also this feature makes it theoretically possible for a Java program to alter its own code during execution, in order to do some self-learning behavior. It would be more realistic to imagine, however, that a Java program would generate Java code before execution, and then, that code would be executed. With some feedback mechanism, the generated code could improve over time.

Automatic memory garbage collection

In conventional languages like C and C++, the programmer has to make sure that all memory that was allocated is freed. Memory leaks became a regular nuisance in instances where the programmers had to manually allocate the system's memory resources.

Memory resources or buffers have specific modes of operation for optimal performance. Once a buffer is filled with data, it needs to be cleaned up after there is no further use for its content. If a programmer forgets to clean it in their code, the memory is easily overloaded. Programming in C/C++ languages became tedious and unsafe because of these very quirks, and programs built in these languages were prone to memory leakages and sudden system crashes — sometimes even harming the hardware itself. Freeing memory is particularly important in servers, since it has to run without stopping for days. If a piece of memory is not freed after use and the server just keeps allocating memory, that memory leak can take down the server.

In Java, freeing up memory is taken out of the programmers hands; the Java Virtual Machine keeps track of all used memory. When memory is not used any more it is automatically freed up. A separate task is running in the background by the JVM, freeing up unreferenced, unused memory. That task is called the *Garbage Collector*.

The Garbage Collector is always running. This automatic memory garbage collection feature makes it easy to write robust server side programs in Java. The only thing the programmer has to watch for is the speed of object creation. If the application is creating objects faster than the Garbage Collector can free them, it can cause memory problems. Depending on how the JVM is configured, the application either can run out of memory by throwing the `NotEnoughMemoryException`, or can halt to give time for the Garbage Collector to do its job.

Applet

The Java creators created the concept of the *applet*. A Java program can be run in a client browser program. Java was released in 1995; the time when the Internet was becoming more available and familiar to the general public. The promise of Java was in the client browser-side in that code would be downloaded and executed as a Java applet in the client browser program.

See also Java Programming/Applets.

Forbidden bad practices

Over the years, some features in C/C++ programming became abused by the programmers. Although the language allows it, it was known as bad practices. So the creators of Java have excluded them from the language:

- Operator overloading
- Multiple inheritance
- Friend classes (access another object's private members)
- Restrictions of explicit type casting (related to memory management)

Evaluation

In most people's opinions, Java technology delivers reasonably well on all these goals. The language is not, however, without drawbacks. Java tends to be more high-level than similar languages (such as C++), which means that the Java language lacks features such as hardware-specific data types, low-level pointers to arbitrary memory addresses, or programming methods like operator overloading. Although these features are frequently abused or misused by programmers, they are also powerful tools. However, Java technology includes Java Native Interface (JNI), a way to call native code from Java language code. With JNI, it is still possible to use some of these features.

Some programmers also complain about Java's lack of multiple inheritance, a powerful feature of several other object-oriented languages, such as C++. The Java language separates inheritance of type and implementation, allowing inheritance of multiple type definitions through interfaces, but only single inheritance of type implementation via class hierarchies. This allows most of the benefits of multiple inheritance while avoiding many of its dangers. In addition, through the use of concrete classes, abstract classes, as well as interfaces, a Java language programmer has the option of choosing full, partial, or zero implementation for the object type they define, thus ensuring maximum flexibility in application design.

There are some who believe that for certain projects, object orientation makes work harder instead of easier. This particular complaint is not unique to the Java language but applies to other object-oriented languages as well.

The Java Platform

The **Java platform** is the name given to the computing platform from Oracle that helps users to *run* and *develop* Java applications. The platform does not just enable a user to run and develop a Java application, but also features a wide variety of tools that can help developers work efficiently with the Java programming language.

The platform consists of two essential pieces of software:

- the **Java Runtime Environment (JRE)**, which is needed to *run* Java applications and applets; and,
- the **Java Development Kit (JDK)**, which is needed to *develop* those Java applications and applets. If you have installed the JDK, you should know that it comes equipped with a JRE as well. So, for all the purposes of this book, you would only require the JDK.

In this section, we will explore in further detail what these two software components of the Java platform do.

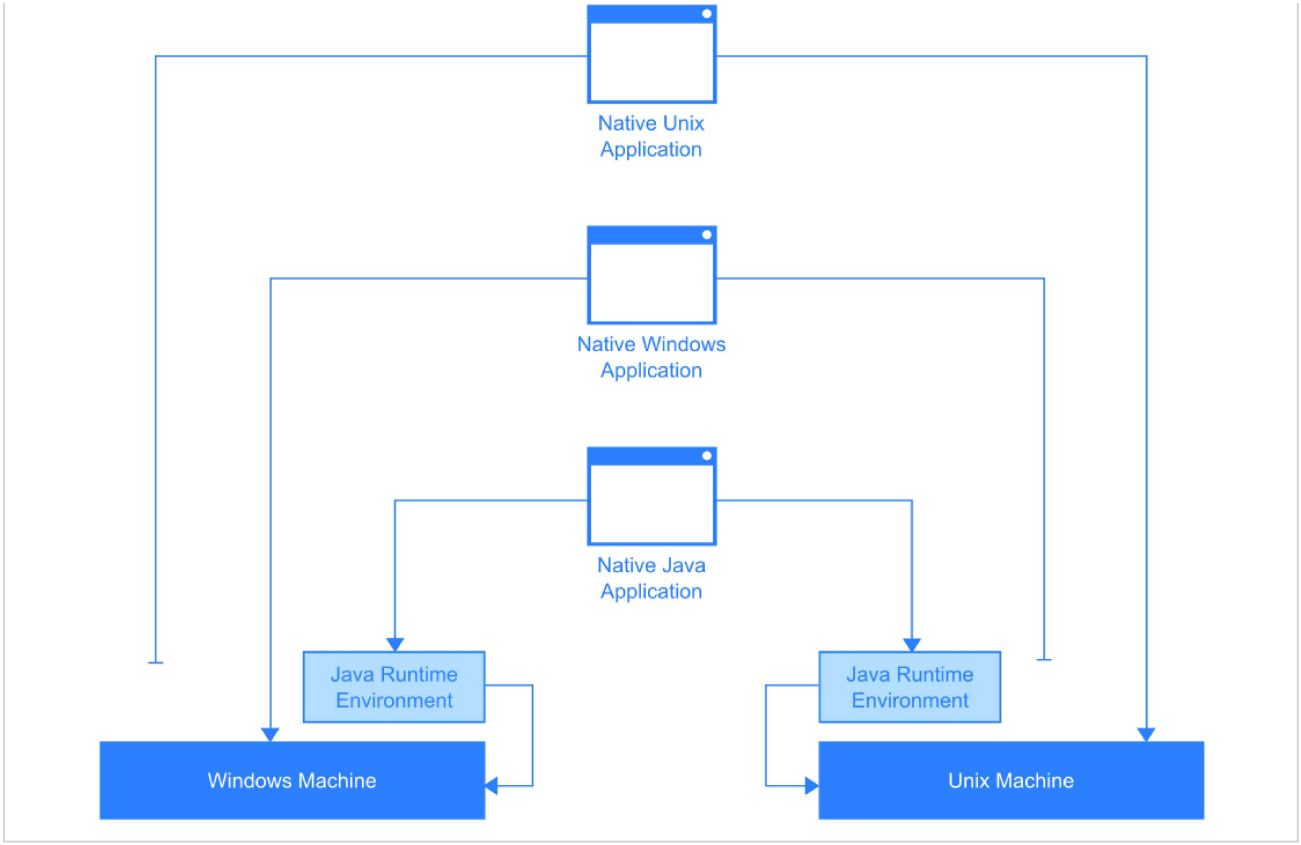
Java Runtime Environment (JRE)

Any piece of code written in the Java programming language can be run on any operating system, platform or architecture — in fact, it can be run on any device that supports the Java platform. Before Java, this amount of ubiquity was very hard to achieve. If a software was written for a Unix-based system, it was impossible to run the same application on a Windows system — in this case, the application was native only to Unix-based systems.

A major milestone in the development of the Java programming language was to develop a special runtime environment that would execute any Java application *independent* of the computer's operating system, platform or architecture.

The **Java Runtime Environment (JRE)** sits on top of the machine's operating system, platform and architecture. If and when a Java application is run, the JRE acts as a liaison between the underlying platform and that application. It interprets the Java application to run in accordance with the underlying platform, such that upon running the application, it looks and behaves like a native application. The part of the JRE that accomplishes this complex liaison agreement is called the **Java Virtual Machine (JVM)**.

Figure 1: Java applications can be *written once* and *run anywhere*. This feature of the Java platform is commonly abbreviated to **WORA** in formal Java texts.



Executing native Java code (or *byte-code*)

Native Java applications are preserved in a special format called the ***byte-code***. Byte-code remains the same, no matter what hardware architecture, operating system, or software platform it is running under. On a file-system, Java byte-code resides in files that have the `.class` (also known as a ***class file***) or the `.jar` (also known as a ***Java archive***) extension. To run byte-code, the JRE comes with a special tool (appropriately named **java**).

Suppose your byte-code is called `SomeApplication.class`. If you want to execute this Java byte-code, you would need to use the following command in Command Prompt (on Windows) or Terminal (on Linux or Mac OS):

Execution

```
$ java SomeApplication
```

If you want to execute a Java byte-code with a `.jar` extension (say, `SomeApplication.jar`), you would need to use the following command in Command Prompt (on Windows) or Terminal (on Linux or Mac OS):

Execution with a jar

```
$ java -jar SomeApplication.jar
```

Not all Java class files or Java archives are executable. Therefore, the **java** tool would only be able to execute files that are executable. Non-executable class files and Java archives are simply called ***class libraries***.

Do you have a JRE?

Most computers come with a pre-installed copy of the JRE. If your computer doesn't have a JRE, then the above commands would not work. You can always check what version of the JRE is installed on the computer by writing the following command in Command Prompt (on Windows) or Terminal (on Linux or Mac OS):

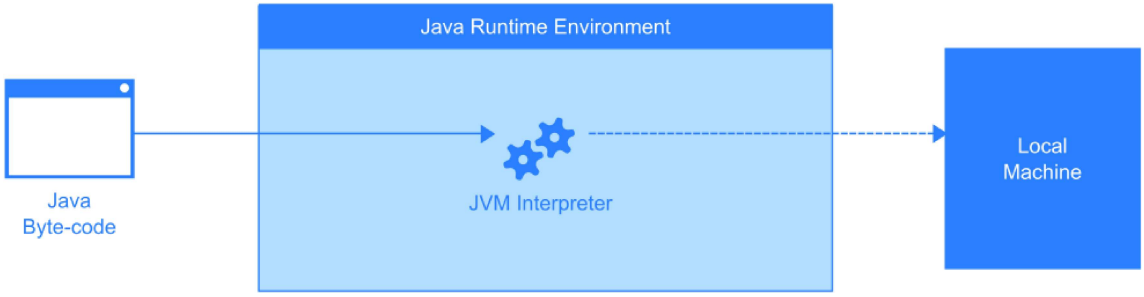
Java version

```
$ java -version
```

Java Virtual Machine (JVM)

Quite possibly, the most important part of the JRE is the **Java Virtual Machine (JVM)**. The JVM acts like a *virtual* processor, enabling Java applications to be run on the local system. Its main purpose is to interpret (*read* translate) the received byte-code and make it appear as native code. The older Java architecture used this process of **interpretation** to execute Java byte-code. Even though the process of interpretation brought the WORA principle to diverse machines, it had a drawback — it consumed a lot of time and clocked the system processor intensively to load an application.

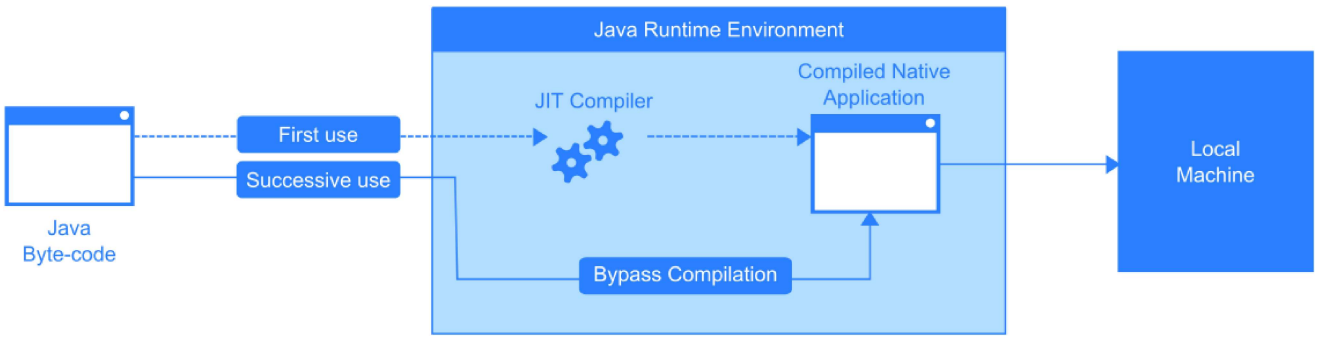
Figure 2: A JVM interpreter *translates* the byte-code line-by-line to make it appear as if a native application is being executed.



Just-in-Time Compilation

Since version 1.2, the JRE features a more robust JVM. Instead of interpreting byte-code, it down-right converts the code straight into equivalent native code for the local system. This process of conversion is called *just-in-time compilation* or *JIT-compilation*. This process only occurs when the byte-code is executed for the first time. Unless the byte-code itself is changed, the JVM uses the compiled version of the byte-code on every successive execution. Doing so saves a lot of time and processor effort, allowing applications to execute much faster at the cost of a *small* delay on first execution.

Figure 3: A just-in-time compiler only compiles the byte-code to equivalent native code at first execution. Upon every successive
execution, the JVM merely uses the already *compiled* native code to optimize performance.



Native optimization

The JVM is an intelligent *virtual* processor. It has the ability to identify areas within the Java code itself that can be optimized for faster and better performance. Based on every successive run of your Java applications, the JVM would optimize it to run even better.

There are portions of Java code that do not require it to be JIT-compiled at runtime, e.g., the Reflection API; therefore, code that uses such functions are not necessarily fully compiled to native code.

Was JVM the first *virtual machine*?

Java was not the first virtual-machine-based platform, though it is by far the most successful and well-known. Previous uses for virtual machine technology primarily involved emulators to aid development for not-yet-developed hardware or operating systems, but the JVM was designed to be implemented entirely in software, while making it easy to efficiently port an implementation to hardware of all kinds.

Java Development Kit (JDK)

The JRE takes care of running the Java code on multiple platforms, however as developers, we are interested in writing pure code in Java which can then be converted into Java byte-code for mass deployment. As developers, we do *not* need to write Java byte-code; rather we write the code in the Java programming language (which is quite similar to writing C or C++ code).

Upon downloading the JDK, a developer ensures that their system has the appropriate JRE and additional tools to help with the development of applications in the Java programming language. Java code can be found in files with the extension **.java**. These files are called ***Java source files***. In order to convert the Java code in these source files to Java byte-code, you need to use the **Java compiler** tool installed with your JDK.

The Java compiler

The **Java compiler** tool (named **javac** in the JDK) is the most important utility found with the JDK. In order to compile a Java source file (say, `SomeApplication.java`) to its respective Java byte-code, you would need to use the following command in Command Prompt (on Windows) or Terminal (on Linux or Mac OS):

Compilation

```
javac SomeApplication.java
```

This command would convert the `SomeApplication.java` source file into its equivalent Java byte-code. The resultant byte-code would exist in a newly created file named `SomeApplication.class`. This process of converting Java source files into their equivalent byte-codes is known as ***compilation***.

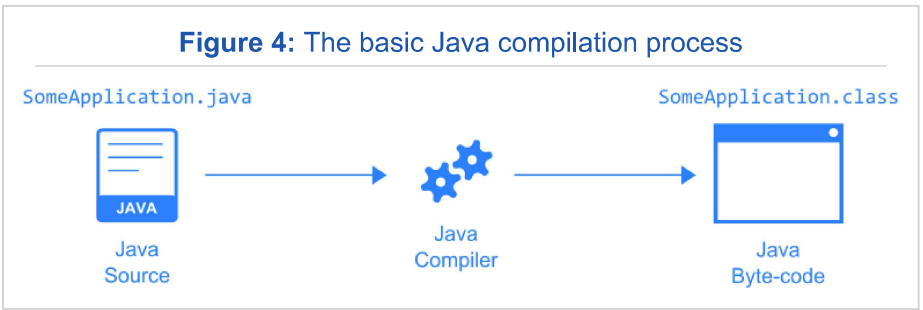


Figure 4: The basic Java compilation process

A list of other JDK tools

There are a huge array of tools available with the JDK that will all be explained in due time as you progress with the book. These tools are briefly listed below in order of their usage:

Applet development

- `appletviewer` — Java applets require a particular environment to execute. Typically, this environment is provided by a browser with a Java plug-in, and a web server serving the applet. However, during development and testing of an applet it might be more convenient to start an applet without the need to fiddle with a browser and a web server. In such a case, Oracle's appletviewer from the JDK can be used to run an applet.

Annotation processing

> *For more about annotation processing, read this (http://docs.oracle.com/javase/1.5.0/docs/guide/apt/GettingStarted.html)*

In Java 1.5 (alias Java 5.0) Oracle added a mechanism called *annotations*. Annotations allow the addition of meta-data to Java source code, and even provide mechanisms to carry that meta-data into compiled `.class` files.

- `apt` — An annotation processing tool which digs through source code, finds annotation statements in the source code and executes actions if it finds known annotations. The most common task is to generate some particular source code. The actions apt performs when finding annotations in the source code are not hard-coded into apt. Instead, one has to code particular annotation handlers (in Java). These handlers are called annotation processors. It can also be described in a simple way without the Oracle terminology: apt can be seen as a source code preprocessor framework, and annotation processors are typically code generators.

Integration of non-Java and Java code

- `javah` — A Java class can call native, or non-Java, code that has been prepared to be called from Java. The details and procedures are specified in the JNI (Java Native Interface). Commonly, native code is written in C (or C++). The JDK tool javah helps to write the necessary C code, by generating C header files and C stub code.

Class library conflicts

- `extcheck` — It can be used prior to the installation of a Java extension into the JDK or JRE environment. It checks if a particular Jar file conflicts with an already installed extension. This tool appeared first with Java 1.5.

Software security and cryptography tools

The JDK comes with a large number of tools related to the security features of Java. Usage of these tools first requires study of the particular security mechanisms. The tools are:

- `keytool` — To manage keys and certificates
- `jarsigner` — To generate and verify digital signatures of JARs (Java ARchives)
- `policytool` — To edit policy files
- `kinit` — To obtain Kerberos v5 tickets
- `klist` — To manage Kerberos credential cache and key table
- `ktab` — To manage entries in a key table

The Java archiver

- `jar` — (short for Java archiver) is a tool for creating Java archives or jar files — a file with `.jar` as the extension. A Java archive is a collection of compiled Java classes and other resources which those classes may require (such as text files, configuration files, images) at runtime. Internally, a jar file is really a .zip file.

The Java debugger

- `jdb` — (short for Java debugger) is a command-line console that provides a debugging environment for Java programs. Although you can use this command line console, IDE's normally provide easier to use debugging environments.

Documenting code with Java

As programs grow large and complex, programmers need ways to track changes and to understand the code better at each step of its evolution. For decades, programmers have been employing the use of special programming constructs called comments — regions that help declare user definitions for a code snippet within the source code. But comments are prone to be verbose and incomprehensible, let alone be difficult to read in applications having hundreds of lines of code.

- `javadoc` — Java provides the user with a way to easily publish documentation about the code using a special commenting system and the javadoc tool. The javadoc tool generates documentation about the Application Programming Interface (API) of a set of user-created Java classes. javadoc reads source file comments from the `.java` source files and generates HTML documents that are easier to read and understand without looking at the code itself.
- `javap` — Where Javadoc provide a detailed view into the API and documentation of a Java class, the javap tool prints information regarding members (constructors, methods and variables) in a class. In other words, it lists the class' API and/or the compiled instructions of the class. javap is a formatting disassembler for Java bytecode.

The native2ascii tool

`native2ascii` is an important, though underappreciated, tool for writing properties files — files containing configuration data — or resource bundles — files containing language translations of text.

Such files can contain only ASCII and Latin-1 characters, but international programmers need a full range of character sets. Text using these characters can appear in properties files and resource bundles only if the non-ASCII and non-Latin-^1 characters are converted into Unicode escape sequences (\uXXXX notation).

The task of writing such escape sequences is handled by `native2ascii`. You can write the international text in an editor using the appropriate character encoding, then use `native2ascii` to generate the necessary ASCII text with embedded Unicode escape sequences. Despite the name, `native2ascii` can also convert from ASCII to native, so it is useful for converting an existing properties file or resource bundle back to some other encoding.

`native2ascii` makes most sense when integrated into a build system to automate the conversion.

Remote Method Invocation (RMI) tools

Java IDL and RMI-IIOP Tools

Deployment & Web Start Tools

Browser Plug-In Tools

Monitoring and Management Tools / Troubleshooting Tools

With Java 1.5 a set of monitoring and management tools have been added to the JDK, in addition to a set of troubleshooting tools.

The monitoring and management tools are intended for monitoring and managing the virtual machine and the execution environment. They allow, for example, monitoring memory usage during the execution of a Java program.

The troubleshooting tools provide rather esoteric insight into aspects of the virtual machine. (Interestingly, the Java debugger is not categorized as a troubleshooting tool.)

All the monitoring and management and troubleshooting tools are currently marked as "experimental" (which does not affect jdb). So they might disappear in future JDKs.

Java class libraries (JCL)

In most modern operating systems, a large body of reusable code is provided to simplify the programmer's job. This code is typically provided as a set of dynamically loadable libraries that applications can call at runtime. Because the Java platform is not dependent on any specific operating system, applications cannot rely on any of the existing libraries. Instead, the Java platform provides a comprehensive set of standard class libraries, containing much of the same reusable functions commonly found in modern operating systems.

The Java class libraries serve three purposes within the Java platform. Like other standard code libraries, they provide the programmer with a well-known set of functions to perform common tasks, such as maintaining lists of items or performing complex string parsing. In addition, the class libraries provide an abstract interface to tasks that would normally depend heavily on the hardware and operating system. Tasks such as network access and file access are often heavily dependent on the native capabilities of the platform. The Java java.net and java.io libraries implement the required native code internally, then provide a standard interface for the Java applications to perform those tasks. Finally, some underlying platforms may not support all of the features a Java application expects. In these cases, the class libraries can either emulate those features using whatever is available, or provide a consistent way to check for the presence of a specific feature.

Similar concepts

The success of the Java platform and the concepts of the write once, run anywhere principle has led to the development of similar frameworks and platforms. Most notable of these is the Microsoft's .NET framework and its open-source equivalent Mono.

The .NET framework

The .NET framework borrows many of the concepts and innovations of Java — their alternative for the JVM is called the Common Language Runtime (CLR), while their alternative for the byte-code is the Common Intermediate Language (CIL). In fact, the .NET platform had an implementation of a Java-like language called Visual J# (formerly known as J++).

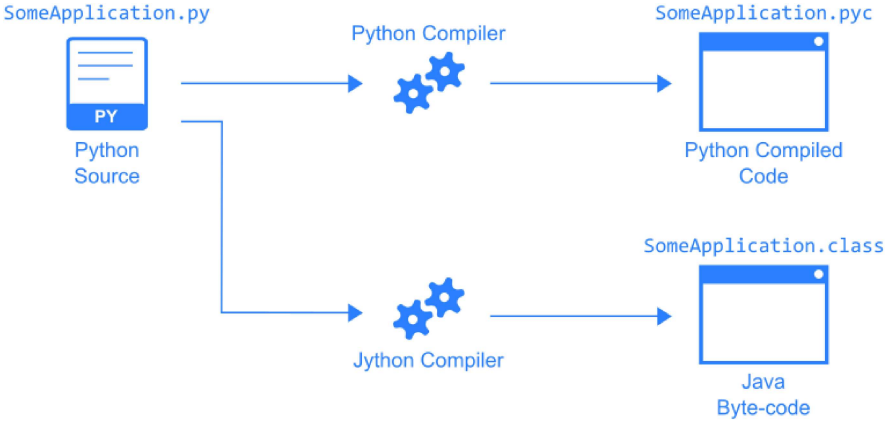
J# is normally not supported with the JVM because instead of compiling it in Java byte-code, the .NET platform compiles the code into CIL, thus making J# different from the Java programming language. Furthermore, because J# implements the .NET Base Class Libraries (BCL) instead of the Java Class Libraries, J# is nothing more than a non-standard extension of the Java programming language. Due to the lack of interest from developers, Microsoft had to withdraw their support for J#, and focused on a similar programming language: C#.

Third-party compilers targeting the JVM

The word Java, by itself, usually refers to the Java programming language which was designed for use with the Java platform. Programming languages are typically outside of the scope of the phrase "platform". However, Oracle does not encourage the use of any other languages with the platform, and lists the Java programming language as a core part of the Java 2 platform. The language and runtime are therefore commonly considered a single unit.

There are cases where you might want to program using a different language (say, Python) and yet be able to generate Java byte-code (instead of the Python compiled code) to be run with the JVM. Many third-party programming language vendors provide compilers that can compile code written in their language to Java byte-code. For instance, Python developers can use Jython compilers to compile Python code to the Java byte-code format (*as illustrated below*).

Figure 5: Third-party JVM-targeted compilation for non-Java source compilation to Java byte-code. Illustrated example
shows Python source being compiled to both Python compiled code and Java byte-code.



Of late, JVM-targeted third-party programming and scripting languages have seen tremendous growth. Some of these languages are also used to extend the functionalities of the Java language itself. A few examples include the following:

- Groovy
- Pizza
- GJ (Generic Java) – later officially incorporated into Java SE 5.
- NetREXX

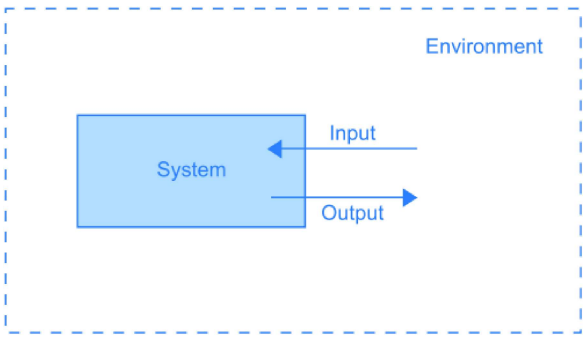
Getting started

Understanding systems

We conceptualize the world around us in terms of systems. A **system** is a web of interconnected objects working together in tandem. In the systems theory, a system is set out as a single entity within a world surrounded by an environment. A system interacts with its surrounding environment using messages of two distinct types:

- **inputs:** messages received from the surrounding environment; and,
- **outputs:** messages given back to the surrounding environment.

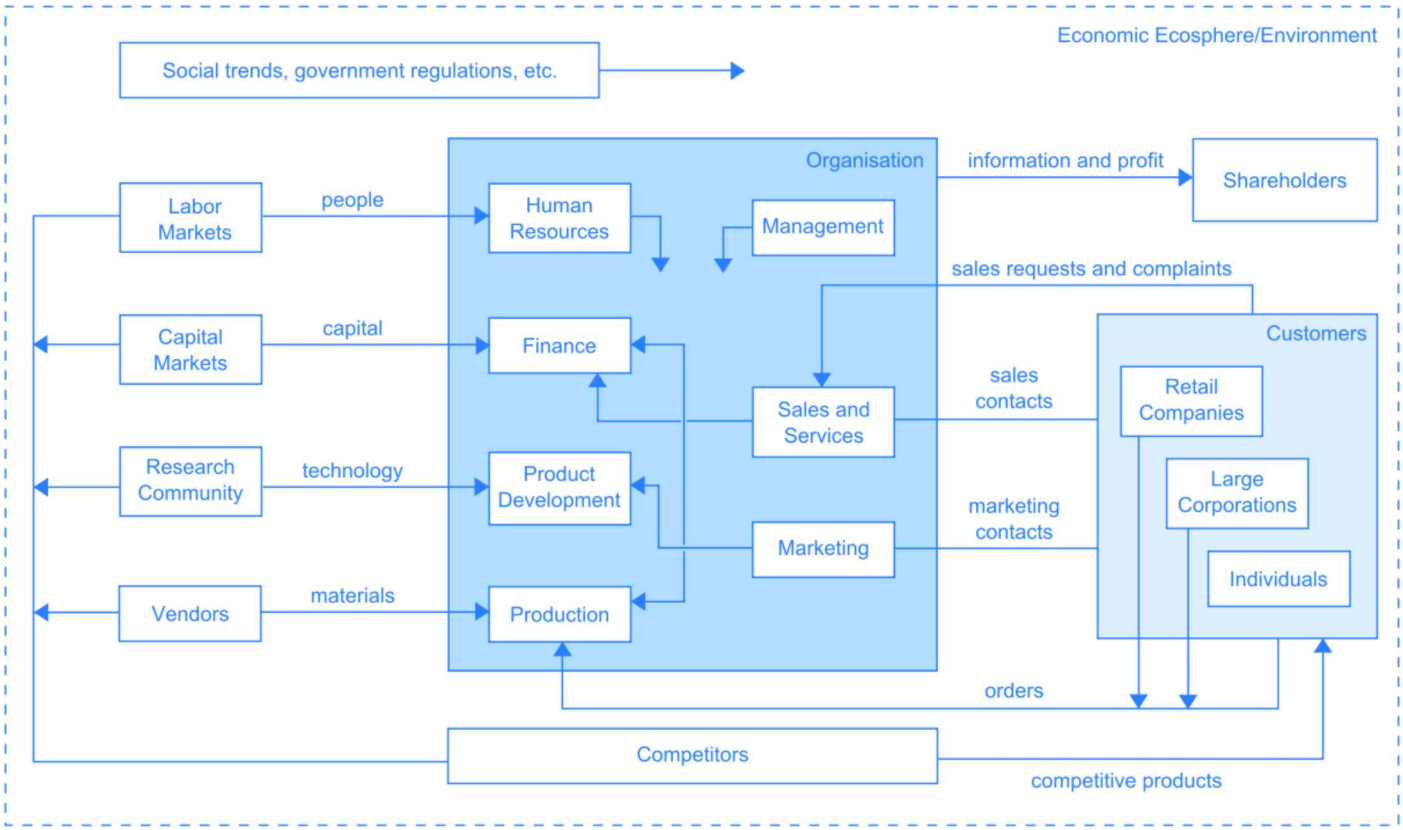
Figure 1: A simple system interacting with its environment using input and output messages.



Life is a complicated mess of interconnected objects sending signals and messages. See the illustration below in figure 2 demonstrating a complex system for an economic ecosphere for a single company. Imagine what this system diagram would be like if you were to add a few more companies and their sub-systems. Computer software systems in

general are a complex web of further interconnected **sub-systems** – where each sub-systems may or may not be divided into further sub-systems. Each sub-system communicates with others using **feedback** messages – that is, *inputs* and *outputs*.

Figure 2: Example of a complex system with multiple sub-systems and interactions



The process of abstraction

Programming is essentially thinking of solutions to problems in real life as a system. With any programming language, you need to know how to address real-life problems into something that could be accurately represented within a computer system. In order to begin programming with the Java programming language (or in fact, with any programming language), a programmer must first understand the basics of abstraction.

Abstraction is the process of *representing* real-life problems and objects into your programs.

Suppose a novelist, a painter and a programmer were asked to *abstract* (i.e., *represent*) a real-life object in their work. Suppose, the real-life object that needs to be abstracted is *an animal*. Abstraction for a novelist would include writing the description of the animal whilst the painter would draw a picture of the animal – but what about a computer programmer?

The Java programming language uses a programming paradigm called object-oriented programming (OOP), which shows you exactly what a programmer needs to be doing. According to OOP, every object or problem in real-life can be translated into a virtual object within your computer system.

Thinking in objects

In OOP, every abstraction of a real-life object is simply called an **object** within your code. An object is essentially the most basic representation of a real-life object as part of a computer system. With Java being an object-oriented language, everything within Java is represented as an object. To demonstrate this effect, if you were to define an abstraction of an animal in your code, you would write the following lines of code (as you would for any other abstraction):

```
class Animal { }
```

The code above creates a space within your code where you can start *defining* an object; this space is called a **class (*or* type) definition**. All objects need to be defined using a class definition in order for them to be used in your program. Notice the curly brackets – anything you write within these brackets would serve as a definition or specification for your object. In the case of the example above, we created a class definition called `Animal` for objects that could serve as an abstract representation of any animal in real-life. The way that a Java environment evaluates this code to be a class definition is by looking at the prefix word we used to begin our class definition (i.e., `class`). Such predefined words in the Java language are known as **keywords** and make up the grammar for the language (known as **programming syntax**).

Understanding class definitions and types

Aristotle was perhaps the first person to think of abstract types or typologies of objects. He started calling them classes – e.g., classes of birds, classes of mammals. Class definitions therefore serve the purpose well in defining the common characteristics or types of objects you would be creating. Upon declaring a class definition, you can create objects based on that definition. In order to do so however, you need to write a special syntax that goes like this:

```
Animal dog = new Animal();
```

The code above effectively creates an object called `dog` based on the class definition for `Animal`. In non-programmer parlance, the code above would translate into something akin to saying, "Create a new object `dog` of type `Animal`." A single class definition enables you to create multiple objects as the code below indicates:

```
Animal dog = new Animal();
Animal cat = new Animal();
Animal camel = new Animal();
```

Basically, you just have to write the code for your class or type definition once, and then use it to create countless numbers of objects based on that specification. Although you might not grasp the importance of doing so, this little exercise saves you a lot of time (a luxury that was not readily available to programmers in the pre-Java days).

Expanding your class definitions

Although each object you create from a class definition is essentially the same, there has to be a way of differentiating those objects in your code. Object fields (or simply **fields**) are what makes your objects unique from other objects. Let's take our present abstraction for instance. An animal could be a dog, cat, camel or a duck but since this abstraction is of a very generic kind, you need to define fields that are common to all of these animals and yet makes the animals stand apart. For instance, you can have two fields: **name** (a common name given to any one of these animals) and **legs** (the number of limbs any one of these animals would require to walk). As you start defining your objects, they start to look like this:

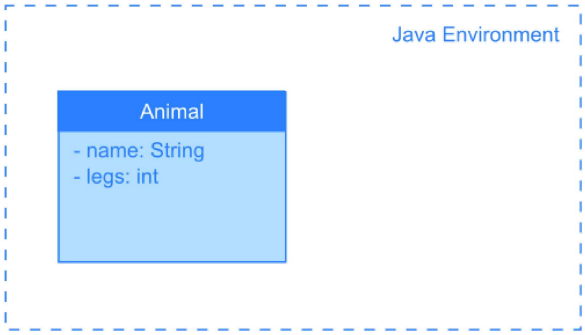
```
class Animal {

  String name;
  int legs;
}
```

In the code above you defined two object fields:

- a field called `name` of type `String`; and,
- a field called `legs` of type `int`.

These special pre-defined types are called **data types**. The `String` data type is used for fields that can hold *textual* values like names, while the `int` (integer) data type is used for fields that can hold *numeric* values

Figure 3: In order to denote the `Animal` object as a system within the Java Environment, you present it as such. Note how fields are presented.



In order to demonstrate how fields work, we will go ahead and create objects from this amended version of our class definition as such:

```
Animal animal1 = new Animal();
Animal animal2 = new Animal();

animal1.name = "dog";
animal1.legs = 4;

animal2.name = "duck";
animal2.legs = 2;
```

You can access the fields of your created objects by using the `.` (dot) or **membership operator**. In the example above, we created two objects: `animal1` and `animal2` of type `Animal`. And since, we had established that each `Animal` has two fields namely `name` and `legs`, we accessed and modified these fields for each of our objects using the membership operator to set the two apart. By declaring different values for different objects, we can manipulate their current **state**. So, for instance:

- the `animal1` object is a `"dog"` with `4` legs to walk with; while,
- the `animal2` object is a `"duck"` with `2` legs to walk with.

What sets the two objects apart is their current state. Both the objects have different states and thus stand out as two different objects even though they were created from the same template or class definition.

Adding behavior to objects

At this point, your objects do nothing more than declare a bunch of fields. Being a system, your objects should have the ability to interact with their environment and other systems as well. To add this capability for interaction, you need to add interactive behavior to your object class definitions as well. Such behavior is added to class definitions using a programming construct called **method**.

In the case of the `Animal`, you require your virtual representation of an animal to be able to move through its environment. Let's say, as an analogy, you want your `Animal` object to be able to walk in its environment. Thus, you need to add a method named `walk` to our object. To do so, we need to write the following code:

```
class Animal {

  String name;
  int legs;

  void walk() { }
}
```

As you write this code, one thing becomes immediately apparent. Just like the class description, a method has curly brackets as well. Generally, curly brackets are used to define an area (or **scope**) within your object. So the first set of curly brackets defined a scope for your class definition called the **class-level scope**. This new set of curly brackets alongside a method defines a scope for the further definition of your method called the **method-level scope**.

In this instance, the name of our method is `walk`. Notice however that the name of our method also features a set of round brackets as well. More than just being visual identifiers for methods, these round brackets are used to provide our methods with additional input information called **arguments**.

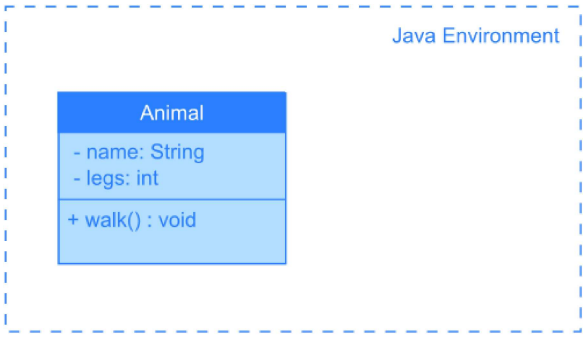
A method therefore enables an object to:

1. Accept input: Receive some argument(s);
2. Process information: work on the received argument(s) within its curly brackets; and,
3. Generate ouput: *occasionally*, return something back.

In essence, methods are what makes an object behave more like a system.

Notice the keyword `void` before the name of the method – this tells us that the method `walk` returns *nothing*. You can set a method to return any data type – it can be a **String** or an **int** as well.

Figure 4: The `Animal` object can now be denoted as having an interaction behavior within the Java Environment
as illustrated here. Note the difference between the presentation of fields and methods.



The process of encapsulation

By now, we thoroughly understand that any object can interact with its environment and in turn be influenced by it. In our example, the `Animal` object exposed certain fields – `name` and `legs`, and a method – `walk()` to be used by the environment to manipulate the object. This form of exposure is implicit. Using the Java programming language, a programmer has the power to define the level of access other objects and the environment have on a certain object.

Using access modifiers

Alongside declaring and defining objects, their fields and methods, a programmer also has the ability to define the levels of access on those elements. This is done using keywords known as **access modifiers**.

Let's modify our example to demonstrate this effect:

```
class Animal {

  public String name;
  public int legs;

  public void walk() { }
}
```

By declaring all fields and methods `public`, we have ensured that they can be used outside the scope of the `Animal` class. This means that any other object (other than `Animal`) has access to these member elements. However, to restrict access to certain member elements of a class, we can always use the `private` access modifier (as demonstrated below).

```
class Animal {

  private String name;
  private int legs;

  public void walk() { }
}
```

In this example, the fields `name` and `legs` can only be accessed within the scope of the `Animal` class. No object outside the scope of this class can access these two fields. However, since the `walk()` method still has `public` access, it can be manipulated by actors and objects outside the scope of this class. Access modifiers are not just limited to fields or methods, they can be used for class definitions as well (as is demonstrated below).

```
public class Animal {

  private String name;
  private int legs;

  public void walk() { }
}
```

The following list of keywords show the valid access modifiers that can be used with a Java program:

keyword	description
`public`	Opens access to a certain field or method to be used outside the scope of the class.
`private`	Restricts access to a certain field or method to only be used within the scope of the class.
`protected`	Access to certain field or methods is reserved for classes that inherit the current class. More on this would be discussed in the section on *inheritance*.

Installation

In order to make use of the content in this book, you would need to follow along each and every tutorial rather than simply reading through the book. But to do so, you would need access to a computer with the **Java platform** installed on it — the Java platform is the basic prerequisite for *running* and *developing* Java code, thus it is divided into two essential pieces of software:

- the **Java Runtime Environment (JRE)**, which is needed to *run* Java applications and applets;
- the **Java Development Kit (JDK)**, which is needed to *develop* those Java applications and applets.

However as a developer, you would only require the JDK which comes equipped with a JRE as well.

As Java is just a programming language that allows you to program the computer, it has multiple implementations available. The most popular implementation of JDK and JRE are the "Oracle Java SE" (formerly known as Sun JDK), maintained by Oracle as a commercial release. However another similarly popular implementation is the OpenJDK, with the benefit of being free software that could distribute freely under GPL v2 without the requirement of accepting the "Oracle Binary Code License Agreement for the Java SE Platform Products and JavaFX". The third option - the GCJ, as part of the GNU Compiler Collection, would also supply the JDK and JRE.

Given below are installation instruction for the Oracle Java SE JDK for various operating systems:

Installation instructions for Windows

Availability check for JRE

The Java Runtime Environment (JRE) is necessary to execute Java programs. To check which version of Java Runtime Environment (JRE) you have, follow the steps below.

1. For Windows Vista or Windows 7, click **Start** › **Control Panel** › **System and Maintenance** › **System**.
   For Windows XP, click **Start** › **Control Panel** › **System**.
   For Windows 2000, click **Start** › **Settings** › **Control Panel** › **System**.
   Alternatively, you can also press ⊞ Win + R to open the **Run** dialog. With the dialog open, type `cmd` at the prompt:

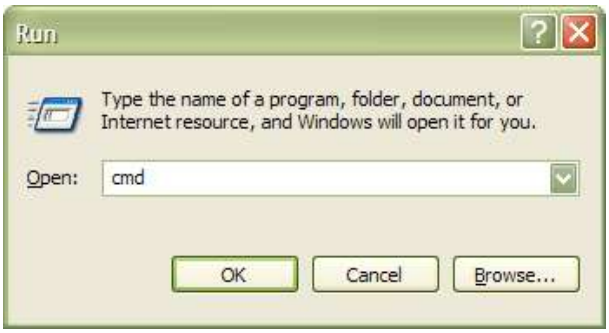


Figure 1.1: Run dialog

2. In the command window with black background graced with white text, type the following command:

JRE availability check

```
java -version
```

If you get an error, such as:

Other output error

```
Bad command or file name
```

..then the JDK may not be installed or it may not be in your path.

To learn more about the Command Prompt syntax, take a look at this MS-DOS tutorial (http://tnd.com/camosun/elex130/dostutor1.html).

You may have other versions of Java installed; this command will only show the first in your PATH. You will be made familiar with the PATH environment variable later in this text. For now, if you have no idea what this is all about. Read through towards the end and we will provide you with a step-by-step guide on how to set your own environment variables.

You can use your system's file search utilities to see if there is a `javac.exe` executable installed. If it is, and it is a recent enough version (Java 1.4.2 or Java 1.5, for example), you should put the `bin` directory that contains `javac` in your system path. The Java runtime, `java`, is often in the same `bin` directory.

If the installed version is older (i.e. it is Java 1.3.1 or Java 1.4.2 and you wish to use the more recent Java 5 release), you should proceed below with downloading and installing a JDK.

It is possible that you have the Java runtime (JRE), but not the JDK. In that case the `javac` program won't be found, but the `java -version` will print the JRE version number.

Availability check for JDK

Some Windows based systems come built-in with the JRE, however for the purposes of writing Java code by following the tutorials in this book, you would require the JDK nevertheless. The Java Development Kit (JDK) is necessary to build Java programs. First, check to see if a JDK is already installed on your system. To do so, first open a command window and execute the command below.

Availability check

```
javac -version
```

If the JDK is installed and on your executable path, you should see some output which tells you the command line options. The output will vary depending on which version is installed and which vendor provided the Java installation.

Advanced availability check options on Windows platform

On a machine using the Windows operating system, one can invoke the Registry Editor utility by typing `REGEDIT` in the Run dialog. In the window that opens subsequently, if you traverse through the hierarchy `HKEY_LOCAL_MACHINE > SOFTWARE > JavaSoft > Java Development Kit` on the left-hand.

The resultant would be similar to figure 1.2, with the only exception being the version entries for the Java Development Kit. At the time of writing this manuscript, the latest version for the Java Development Kit available from the Internet was 1.7 as seen in the Registry entry. If you see a resultant window that resembles the one presented above, it would prove that you have Java installed on your system, otherwise it is not.

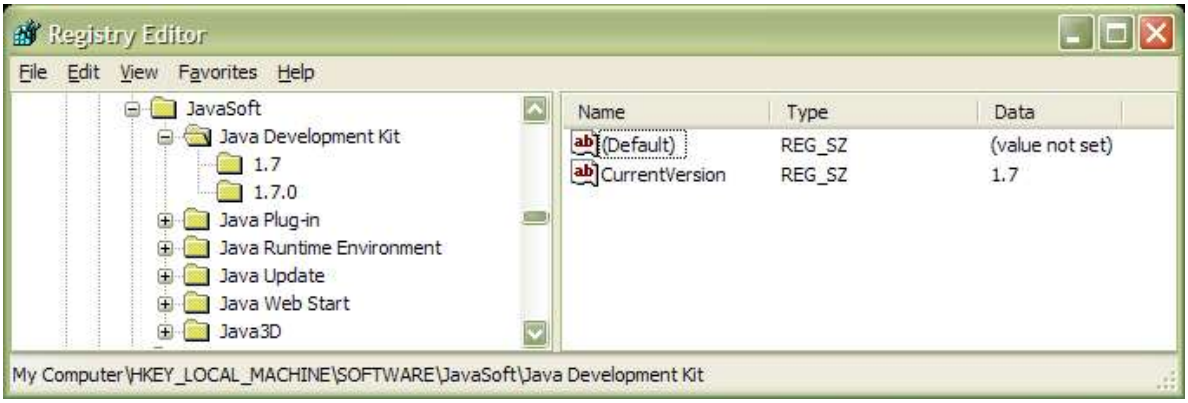


Figure 1.2: Registry Editor

Caution should be exercised when traversing through the Registry Editor. Any changes to the keys and other entries may change the way your Windows operating system normally works. Even minor changes may result into catastrophic failures of the normal working of your machine. Better that you don't modify or tend to modify anything whilst you are in the Registry Editor.

Download instructions

To acquire the latest JDK (version 7), you can manually download the Java software (http://www.oracle.com/technetwork/java/javase/downloads/index.html) from the Oracle website.

For the convenience of our readers, the following table presents direct links to the latest JDK for the Windows operating system.

Operating system	Setup Installer	License
Windows x86	Download (http://download.oracle.com/otn-pub/java/jdk/7u1-b08/jdk-7u1-windows-i586.exe)	Oracle Binary Code License Agreement (http://www.oracle.com/technetwork/java/javasebusiness/documentation/java-se-bcl-license-430205.html)
Windows x64	Download (http://download.oracle.com/otn-pub/java/jdk/7u1-b08/jdk-7u1-windows-x64.exe)	Oracle Binary Code License Agreement (http://www.oracle.com/technetwork/java/javasebusiness/documentation/java-se-bcl-license-430205.html)

You must follow the instructions for the setup installer wizard step-by-step with the default settings to ensure that Java is properly installed on your system. Once the setup is completed, it is highly recommended to restart your Windows operating system.

If you kept the default settings for the setup installer wizard, your JDK should now be installed at **C:\Program Files\Java\jdk1.7.0_01**. You would require the location to your **bin** folder at a later time — this is located at **C:\Program Files\Java\jdk1.7.0_01\bin** It may be a hidden file, but no matter. Just don't use Program Files (x86)\ by mistake unless that's where the files were installed by Java.

Updating environment variables

In order for you to start using the JDK compiler utility with the Command Prompt, you would need to set the environment variables that points to the **bin** folder of your recently installed JDK. To set permanently your environment variables, follow the steps below.

1. To open **System Properties** dialog box use, the Control Panel or type the following command in the command window:

```
System properties
```

```
rundll32 shell32.dll,Control_RunDLL sysdm.cpl
```

2. Navigate to the **Advanced** tab on the top, and select **Environment Variables...**
3. Under **System variables**, select the variable named **Path** and click **Edit...**
4. In the **Edit System Variable** dialog, go to the **Variable value** field. This field is a list of directory paths separated by semi-colons (;).
5. To add a new path, append the location of your JDK **bin** folder separated by a semi-colon (;).
6. Click **OK** on every opened dialog to save changes and get past to where you started.

Start writing code

Once you have successfully installed the JDK on your system, you are ready to program code in the Java programming language. However, to write code, you would need a decent text editor. Windows comes with a default text editor by default — **Notepad**. In order to use notepad to write code in Java, you need to follow the steps below:

1. Click **Start** › **All Programs** › **Accessories** › **Notepad** to invoke the application.
   Alternatively, you can also press ⊞ Win + R to open the **Run** dialog. With the dialog open, type the following command at the prompt:

```
Notepad launching
```

```
notepad
```

2. Once the **Notepad** application has fired up, you can use the editor to write code for the Java programming language.

Installation instructions for GNU/Linux

Availability check for JRE

The Java Runtime Environment (JRE) is necessary to execute Java programs. To check which version of JRE you have, follow the steps below.

1. Open the **Terminal** window.
2. Type the following command:

JRE availability check

```
java -version
```

If you get something like this:

Output on a particular Kubuntu 12.10 installation, with OpenJDK as the provider of JDK and JRE

```
java version "1.7.0_09"
OpenJDK Runtime Environment (IcedTea7 2.3.3) (7u9-2.3.3-0ubuntu1~12.10.1)
OpenJDK Client VM (build 23.2-b09, mixed mode, sharing)
```

... then a JRE is installed. If you get an error, such as:

Output error

```
java: command not found
```

... then the JDK may not be installed or it may not be in your PATH.

You may have other versions of Java installed; this command will only show the first in your PATH. You will be made familiar with the PATH environment variable later in this text. For now, if you have no idea what this is all about, read through towards the end and we will provide you with a step-by-step guide on how to set your own environment variables.

You can use your system's file search utilities to see if there is a `javac` executable installed. If it is, and it is a recent enough version, you should put the `bin` directory that contains `javac` in your system PATH. The Java runtime, `java`, is often in the same `bin` directory.

If the installed version is older (i.e. it is Java 5 and you wish to use the more recent Java 7 release), you should proceed below with downloading and installing a JDK.

It is possible that you have the Java runtime (JRE), but not the JDK. In that case the `javac` program won't be found, but the `java -version` will print the JRE version number.

Availability check for JDK

The Java Development Kit (JDK) is necessary to build Java programs. For our purposes, you must use a JDK. First, check to see if a JDK is already installed on your system. To do so, first open a terminal window and execute the command below.

Availability check

```
javac -version
```

If the JDK is installed and on your executable path, you should see some output which tells you the command line options. The output will vary depending on which version is installed and which vendor provided the Java installation.

Installation using Terminal

Downloading and installing the Java platform on Linux machines is very easy and straightforward. You have two choices to install the Java platforms: using a package manager such as DPKG/APT, YUM/RPM etc., or directly install them using the binary package. To use the terminal to download and install the Oracle Java SE platform, follow the instructions below.

1. Open the **Terminal** window.
2. At the prompt, type in the line followed by the name of your package manager as shown below:

Retrieving the java packages

```
# APT - Ubuntu, Linux Mint
$ sudo add-apt-repository ppa:webupd8team/java
$ sudo apt update
$ sudo apt-get install oracle-java8-installer
# APT - Debian, etc.
$ echo "deb http://ppa.launchpad.net/webupd8team/java/ubuntu xenial main" | sudo tee
/etc/apt/sources.list.d/webupd8team-java.list
$ echo "deb-src http://ppa.launchpad.net/webupd8team/java/ubuntu xenial main" | sudo tee -a
/etc/apt/sources.list.d/webupd8team-java.list
$ sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys EEA14886
$ sudo apt-get update
$ sudo apt-get install oracle-java8-installer
# Portage - Gentoo
# You need to accept the license and fetch the source code manually from
http://www.oracle.com/technetwork/java/javase/downloads/index.html
#   and save it to /usr/portage/distfiles
$ echo "dev-java/oracle-jdk-bin Oracle-BCLA-JavaSE" | sudo tee -a /etc/portage/package.license # Accept the Oracle
License
$ emerge oracle-jdk-bin
# Pacman - Arch
$ sudo pacman -S jdk8-openjdk
```

3. All Java softwares should be installed and instantly available now.

Download instructions

Alternatively, you can manually download the Java software (http://www.oracle.com/technetwork/java/javase/downloads/index.html) from the Oracle website.

For the convenience of our readers, the following table presents direct links to the latest JDK for the Linux operating system.

Operating system	RPM	Tarball	License
Linux x86	Download (http://download.oracle.com/otn-pub/java/jdk/7u7-b10/jdk-7u7-linux-i586.rpm)	Download (http://download.oracle.com/otn-pub/java/jdk/7u1-b08/jdk-7u1-linux-i586.tar.gz)	Oracle Binary Code License Agreement (http://www.oracle.com/technetwork/java/javasebusiness/documentation/java-se-bcl-license-430205.html)
Linux x64	Download (http://download.oracle.com/otn-pub/java/jdk/7u1-b08/jdk-7u1-linux-x64.rpm)	Download (http://download.oracle.com/otn-pub/java/jdk/7u1-b08/jdk-7u1-linux-x64.tar.gz)	Oracle Binary Code License Agreement (http://www.oracle.com/technetwork/java/javasebusiness/documentation/java-se-bcl-license-430205.html)

Start writing code

The most widely available text editor on GNOME desktops is **Gedit**, while on the KDE desktops, one can find **Kate**. Both these editors support syntax highlighting and code completion and therefore are sufficient for our purposes.

However, if you require a robust and standalone text-editor like the Notepad++ editor on Windows, you would require the use of the minimalistic editor loaded with features – **SciTE**. Follow the instructions below if you wish to install **SciTE**:

1. Open the **Terminal** window.
2. At the prompt, write the following:

Retrieving the java packages

```
$ sudo apt-get install scite
```

3. You should now be able to use SciTE for your programming needs. You may also want to try Geany. Installation instructions are similar to those for SciTE.

Installation instructions for Mac OS

On Mac OS, both the JRE and the JDK are already installed. However, the version installed was the latest version when the computer was purchased, so you may want to update it.

Updating Java for Mac OS

1. Go to the Java download page (http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html).
2. Mechanically accept Oracle's license agreement.
3. Click on the link for Mac OS X.
4. Run the installer package.

Availability check for JDK

The Java Development Kit (JDK) is necessary to build Java programs. For our purposes, you must use a JDK. First, check to see if a JDK is already installed on your system. To do so, first open a terminal window and execute the command below.

Availability check

```
java -version
```

If the JDK is installed and on your executable path, you should see some output which tells you the command line options. The output will vary depending on which version is installed and which vendor provided the Java installation.

Installation instructions for Solaris

No Install Option for Programming Online

If you already have the JRE installed, you can use the Java Wiki Integrated Development Environment (JavaWIDE) to code directly in your browser, no account or special software required.

Click here to visit the JavaWIDE Sandbox to get started. (http://sandbox.javawide.org)

For more information, click here to visit the JavaWIDE site. (http://www.javawide.org)

Compilation

In Java, programs are not compiled into executable files; they are compiled into bytecode (as discussed earlier), which the JVM (Java Virtual Machine) then executes at runtime. Java source code is compiled into bytecode when we use the `javac` compiler. The bytecode gets saved on the disk with the file extension `.class`. When the program is to be run, the bytecode is converted, using the just-in-time (JIT) compiler. The result is machine code which is then fed to the memory and is executed.

Java code needs to be compiled twice in order to be executed:

1. Java programs need to be compiled to bytecode.
2. When the bytecode is run, it needs to be converted to machine code.

The Java classes/bytecode are compiled to machine code and loaded into memory by the JVM when needed the first time. This is different from other languages like C/C++ where programs are to be compiled to machine code and linked to create an executable file before it can be executed.

Quick compilation procedure

To execute your first Java program, follow the instructions below:

1. Proceed only if you have successfully installed and configured your system for Java as discussed here.
2. Open your preferred text editor — this is the editor you set while installing the Java platform.
   For example, **Notepad** or **Notepad++** on Windows; **Gedit**, **Kate** or **SciTE** on Linux; or, **XCode** on Mac OS, etc.
3. Write the following lines of code in a new text document:

Code listing 2.5: HelloWorld.java

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

4. Save the file as **HelloWorld.java** — the name of your file should be the same as the name of your class definition and followed by the `.java` extension. This name is *case-sensitive*, which means you need to capitalize the precise letters that were capitalized in the name for the class definition.
5. Next, open your preferred command-line application.
   For example, **Command Prompt** on Windows; and, **Terminal** on Linux and Mac OS.
6. In your command-line application, navigate to the directory where you just created your file. If you do not know how to do this, consider reading through our crash courses for command-line applications for Windows or Linux.
7. Compile the Java source file using the following command which you can copy and paste in if you want:

Compilation

```
javac HelloWorld.java
```

If you obtain an error message like `error: cannot read: HelloWorld.java 1 error`, your file is not in the current folder or it is badly spelled. Did you navigate to the program's location in the command prompt using the cd (*c*hange *d*irectory) command?

If you obtain another message ending by `1 error` or `... errors`, there may be a mistake in your code. Are you sure all words are spelled correctly and with the exact case as shown? Are there semicolons and brackets in the appropriate spot? Are you missing a quote? Usually, modern IDEs would try coloring the entire source as a quote in this case.

If your computer emits beeps, then you may have illegal characters in your `HelloWorld.java`.

If no `HelloWorld.class` file has been created in the same folder, then you've got an error. Are you launching the `javac` program correctly?

8. Once the compiler returns to the prompt, run the application using the following command:

Execution

```
java HelloWorld
```

If you obtain an error message like `Exception in thread "main" java.lang.NoClassDefFoundError: HelloWorld`, the `HelloWorld.class` file is not in the current folder or it is badly spelled.

If you obtain an error message like `Exception in thread "main" java.lang.NoSuchMethodError: main`, your source file may have been badly written.

9. The above command should result in your command-line application displaying the following result:

Output

```
Hello World!
```

Ask for help if the program did not execute properly in the Discussion page for this chapter.

Automatic Compilation of Dependent Classes

In Java, if you have used any reference to any other java object, then the class for that object will be automatically compiled, if that was not compiled already. These automatic compilations are nested, and this continues until all classes are compiled that are needed to run the program. So it is usually enough to compile only the high level class, since all the dependent classes will be automatically compiled.

Main class compilation

```
javac ... MainClass.java
```

However, you can't rely on this feature if your program is using reflection to create objects, or you are compiling for servlets or for a "jar", package. In these cases you should list these classes for explicit compilation.

Main class compilation

```
javac ... MainClass.java ServletOne.java ...
```

Packages, Subdirectories, and Resources

Each Java top level class belongs to a package (covered in the chapter about Packages). This may be declared in a `package` statement at the beginning of the file; if that is missing, the class belongs to the unnamed package.

For compilation, the file must be in the right directory structure. A file containing a class in the unnamed package must be in the current/root directory; if the class belongs to a package, it must be in a directory with the same name as the package.

The convention is that package names and directory names corresponding to the package consist of only lower case letters.

Top level package

A class with this package declaration

Code section 2.1: Package declaration

```
package example;
```

has to be in a directory named `example`.

Subpackages

A class with this package declaration

Code section 2.2: Package declaration with sub-packages

```
package org.wikibooks.en;
```

has to be in a directory named `en` which has to be a sub-directory of `wikibooks` which in turn has to be a sub-directory of `org` resulting in `org/wikibooks/en` on Linux or `org\wikibooks\en` on Windows.

Java programs often contain non-code files such as images and properties files. These are referred to generally as *resources* and stored in directories local to the classes in which they're used. For example, if the class `com.example.ExampleApp` uses the `icon.png` file, this file could be stored as `/com/example/resources/icon.png`. These resources present a problem when a program is compiled, because `javac` does not copy them to wherever the `.class` files are being compiled to (see above); it is up to the programmer to move the resource files and directories.

Filename Case

The Java source file name must be the same as the public class name that the file contains. There can be only one public class defined per file. The Java class name is case sensitive, as is the source file name.

The naming convention for the class name is for it to start with a capital letter.

Compiler Options

Debugging and Symbolic Information

To debug into Java system classes such as String and ArrayList, you need a special version of the JRE which is compiled with "debug information". The JRE included inside the JDK provides this info, but the regular JRE does not. Regular JRE does not include this info to ensure better performance.

```
   Modern compilers do a pretty good job converting your high-level code, with its nicely indented and nested control structures and
arbitrarily typed variables into a big pile of bits called machine code (or bytecode in case of Java), the sole purpose of which is to
run as fast as possible on the target CPU (virtual CPU of your JVM). Java code gets converted into several machine code instructions.
Variables are shoved all over the place – into the stack, into registers, or completely optimized away. Structures and objects don't even
exist in the resulting code – they're merely an abstraction that gets translated to hard-coded offsets into memory buffers.
```

```
   So how does a debugger know where to stop when you ask it to break at the entry to some function? How does it manage to find what to
show you when you ask it for the value of a variable? The answer is – debugging information.
```

```
   Debugging information is generated by the compiler together with the machine code. It is a representation of the relationship between
the executable program and the original source code. This information is encoded into a pre-defined format and stored alongside the
machine code. Many such formats were invented over the years for different platforms and executable files.
```

Symbolic Information : Symbolic resolution is done at class loading time at linking resolution step. It is the process of replacing symbolic references from the type with direct references. It is done by searching into method area to locate the referenced entity

The JIT compiler

The Just-In-Time (JIT) compiler is the compiler that converts the byte-code to machine code. It compiles byte-code once and the compiled machine code is re-used again and again, to speed up execution. Early Java compilers compiled the byte-code to machine code each time it was used, but more modern compilers cache this machine code for reuse on the machine. Even then, java's JIT compiling was still faster than an "interpreter-language", where code is compiled from **high level language**, instead of from byte-code each time it was used.

The standard JIT compiler runs *on demand*. When a method is called repeatedly, the JIT compiler analyzes the bytecode and produces highly efficient machine code, which runs very fast. The JIT compiler is smart enough to recognize when the code has already been compiled, so as the application runs, compilation happens only as needed.

As Java applications run, they tend to become faster and faster, because the JIT can perform runtime profiling and optimization to the code to meet the execution environment. Methods or code blocks which do not run often receive less optimization; those which run often (so called *hotspots*) receive more profiling and optimization.

Execution

There are various ways in which Java code can be executed. A complex Java application usually uses third party APIs or services. In this section we list the most popular ways a piece of Java code may be packed together and/or executed.

JSE code execution

Java language first edition came out in the client-server era. Thick clients were developed with rich GUI interfaces. Java first edition, JSE (Java Standard Edition) had/has the following in its belt:

- GUI capabilities (AWT, Swing)
- Network computing capabilities (RMI)
- Multi-tasking capabilities (Threads)

With JSE the following Java code executions are possible:

Stand alone Java application

(Figure 1) Stand alone application refers to a Java program where both the user interface and business modules are running on the same computer. The application may or may not use a database to persist data. The user interface could be either AWT or Swing.
The application would start with a `main()` method of a Class. The application stops when the `main()` method exits, or if an exception is thrown from the application to the JVM. Classes are loaded to memory and compiled as needed, either from the file system or from a *.jar file, by the JVM.
Invocation of Java programs distributed in this manner requires usage of the command line. Once the user has all the class files, he needs to launch the application by the following command line (where Main is the name of the class containing the main() method.)

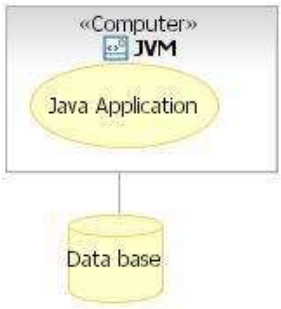


Figure 1: Stand alone execution

Execution of class

```
java Main
```

Java 'jar' class libraries

Utility classes, framework classes, and/or third party classes are usually packaged and distributed in Java '*.jar' files. These 'jar' files need to be put in the CLASSPATH of the java program from which these classes are going to be used.
If a jar file is executable, it can be run from the command line:

Execution of archive

```
java -jar Application.jar
```

Client Server applications

The client server applications consist of a front-end, and a back-end part, each running on a separate computer. The idea is that the business logic would be on the back-end part of the program, which would be reused by all the clients. Here the challenge is to achieve a separation between front-end user interface code, and the back-end business logic code.
The communication between the front-end and the back-end can be achieved by two ways.

- One way is to define a data communication protocol between the two tiers. The back-end part would listen for an incoming request. Based on the protocol it interprets the request and sends back the result in data form.
- The other way is to use Java Remote Invocation (RMI). With the use of RMI, a remote object can be created and used by the client. In this case Java objects are transmitted across the network.

More information can be found about client-server programming, with sample code, at the Client Server Chapter in this book.

Web Applications

For applications needed by lots of client installations, the client-server model did not work. Maintaining and upgrading the hundreds or thousands of clients caused a problem. It was not practical. The solution to this problem was to create a unified, standard client, for all applications, and that is the *Browser*.
Having a standard client, it makes sense to create a unified, standard back-end service as well, and that is the *Application Server*.
Web Application is an application that is running in the Application Server, and it can be accessed and used by the Browser client.
There are three main area of interest in Web Applications, those are:

- The Web Browser. This is the container of rendering HTML text, and running client scripts
- The HTTP protocol. Text data are sent back and forth between Browser and the Server
- The Web server to serve static content, Application server to serve dynamic content and host EJBs.

Wikipedia also has an article about Web application.

J2EE code execution

As the focus was shifting from reaching GUI clients to thin client applications, with Java version 2, Sun introduced J2EE (Java 2 Extended Edition). J2EE added :

- Components Base Architecture, (Servlet, JSP, EJB Containers)

With J2EE the following Java component executions are possible:

Java Servlet code

(Figure 2) Java got its popularity with server side programming, more specifically with J2EE servlets. Servlets are running in a simple J2EE framework to handle client HTTP requests. They are meant to replace CGI programming for web pages rendering dynamic content.
The servlet is running in a so called servlet-container/web container.
The servlet's responsibility is to:

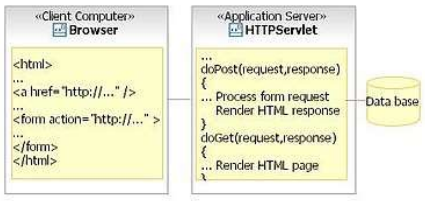


Figure 2: Servlet Execution

- Handle the request by doing the business logic computation,
- Connecting to a database if needed,
- Create HTML to present to the user through the browser

The HTML output represents both the presention logic and the results of the business computations. This represents a huge problem, and there is no real application relying only on servlets to handle the presention part of the responsibility. There are two main solutions to this:

- Use a template tool (Store the presentation part in an HTML file, marking the areas that need to be replaced after business logic computations).
- Use JSP (See next section)

Wikipedia also has an article about Servlets.

Java Server Pages (JSP) code

(Figure 3) JSP is an HTML file with embedded Java code inside. The first time the JSP is accessed, the JSP is converted to a Java Servlet. This servlet outputs HTML which has inside the result of the business logic computation. There are special JSP tags that helps to add data dynamically to the HTML. Also JSP technology allows to create custom tags.
Using the JSP technology correctly, business logic computations should not be in the embedded Java part of the JSP. JSP should be used to render the presentation of the static and dynamic data.
Depending on the complexity of the data, 100% separation is not easy to achieve. Using custom tags, however may help to get closer to 100%. This is advocated also in MVC architecture (see below).

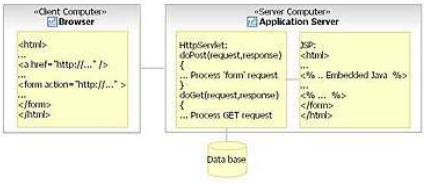


Figure 3: Jsp Execution

EJB code

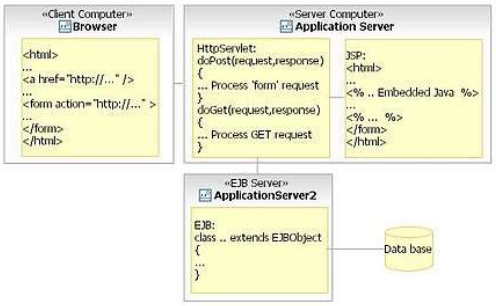


Figure 4: EJB Execution

(Figure 4) In the 1990s, with the client server computing, a trend started, that is to move away from Mainframe computing. That resulted in many small separate applications in a Company/Enterprise. Many times the same data was used in different applications. A new philosophy, "Enterprise Computing", was created to address these issues. The idea was to create components that can be reused throughout the Enterprise. The Enterprise Java Beans (EJBs) were supposed to address this.

An **EJB** is an application component that runs in an EJB container. The client accesses the EJB modules through the container, never directly. The container manages the life cycle of the EJB modules, and handles all the issues that arise from network/enterpise computing. Some of those are security/access control, object pooling, transaction management, ...

EJBs have the same problems as any reusable code: they need to be generic enough to be able to be reused and the changes or maintenance of EJBs can affect existing clients. Many times EJBs are used unnecessarily when they are not really needed. An EJB should be designed as a separate application in the enterprise, fulfilling one function.

Combine J2EE components to create an MVC architecture

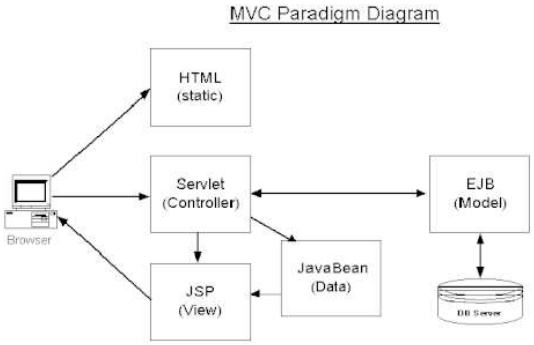


Figure 5: MVC Execution

This leads us to the three layers/tiers as shown in (Figure 5).

In modern web applications, with lots of static data and nice graphics, how the data is presented to the user became very important and usually needs the help of a graphic artist.

To help programmers and graphic artists to work together, the separation between data, code, and how it is presented became crucial.

- The **view** (User Interface Logic) contains the logic that is necessary to construct the presentation. This could be handled by JSP technology.
- The servlet acts as the **controller** and contains the logic that is necessary to process user events and to select an appropriate response.
- The business logic (**model**) actually accomplishes the goal of the interaction. This might be a query or an update to a database. This could be handled by EJB technology.

For more information about MVC, please see MVC.

Jini

After J2EE Sun had a vision about the next step of network computing. That is Jini. The main idea is that in a network environment, there would be many independent services and consumers. Jini would allow these services/consumers to interact dynamically with each other in a robust way. The basic features of Jini are:

- **No user intervention** is needed when services are brought on or offline. (In contrast to EJBs where the client program has to know the server and port number where the EJB is deployed, in Jini the client is *supposed to find*, to discover, the service in the network.)
- **Self healing** by adapting when services (consumers of services) come and go. (Services periodically need to renew a lease to indicate that they are still available.)
- Consumers of JINI services do not need prior knowledge of the service's implementation. The **implementation is downloaded dynamically** and run on the consumer JVM, without configuration and user intervention. (For example, the end user may be presented with a slightly different user interface depending upon which service is being used at the time. The implementation of the user interface code would be provided by the service being used.)

A minimal Jini network environment consists of:

- One or more **services**
- A **lookup-service** keeping a list of registered services

- One or more **consumers**

Jini is not widely used at the current writing (2006). There are two possible reasons for it. One is Jini a bit complicated to understand and to set it up. The other reason is that Microsoft pulled out from Java, which caused the industry to turn to the use of proprietary solutions.

Understanding a Java Program

This article presents a small Java program which can be run from the console. It computes the distance between two points on a plane. You do not need to understand the structure and meaning of the program just yet; we will get to that soon. Also, because the program is intended as a simple introduction, it has some room for improvement, and later in the module we will show some of these improvements. But let's not get too far ahead of ourselves!

The Distance Class: Intent, Source, and Use

This class is named *Distance*, so using your favorite editor or Java IDE, first create a file named `Distance.java`, then copy the source below, paste it into the file and save the file.

Code listing 2.1: Distance.java

```
1  public class Distance {
2    private java.awt.Point point0, point1;
3
4    public Distance(int x0, int y0, int x1, int y1) {
5      point0 = new java.awt.Point(x0, y0);
6      point1 = new java.awt.Point(x1, y1);
7    }
8
9    public void printDistance() {
10     System.out.println("Distance between " + point0 + " and " + point1
11                        + " is " + point0.distance(point1));
12   }
13
14   public static void main(String[] args) {
15     Distance dist = new Distance(
16              intValue(args[0]), intValue(args[1]),
17              intValue(args[2]), intValue(args[3]));
18     dist.printDistance();
19   }
20
21   private static int intValue(String data) {
22     return Integer.parseInt(data);
23   }
24 }
```

At this point, you may wish to review the source to see how much you might be able to understand. While perhaps not being the most literate of programming languages, someone with understanding of other procedural languages such as C, or other object oriented languages such as C++ or C#, will be able to understand most if not all of the sample program.

Once you save the file, compile the program:

Compilation command

```
$ javac Distance.java
```

(If the `javac` command fails, review the installation instructions.)

To run the program, you supply it with the x and y coordinates of two points on a plane separated by a space. For this version of Distance, only integer points are supported. The command sequence is *`java Distance <x0> <y0> <x1> <y1>`* to compute the distance between the points (x_0, y_0) and (x_1, y_1).

If you get a `java.lang.NumberFormatException` exception, some arguments are not a number. If you get a `java.lang.ArrayIndexOutOfBoundsException` exception, you did not provide enough numbers.

Here are two examples:

Output for the distance between the points (0, 3) and (4, 0)

```
$ java Distance 0 3 4 0
Distance between java.awt.Point[x=0,y=3] and java.awt.Point[x=4,y=0] is 5.0
```

Output for the distance between the points (-4, 5) and (11, 19)

```
$ java Distance -4 5 11 19
Distance between java.awt.Point[x=-4,y=5] and java.awt.Point[x=11,y=19] is 20.518284528683193
```

We'll explain this strange looking output, and also show how to improve it, later.

Detailed Program Structure and Overview

As promised, we will now provide a detailed description of this Java program. We will discuss the syntax and structure of the program and the meaning of that structure.

Introduction to Java Syntax

```
public class Distance {
   private java.awt.Point point0, point1;

   public Distance(int x0, int y0, int x1, int y1) {
     point0 = new java.awt.Point(x0, y0);
     point1 = new java.awt.Point(x1, y1);
   }

   public void printDistance() {
     System.out.println("Distance between " + point0 + " and " + point1
                     + " is " + point0.distance(point1));
   }

   public static void main(String[] args) {
     Distance dist = new Distance(
              intValue(args[0]), intValue(args[1]),
              intValue(args[2]), intValue(args[3]));
     dist.printDistance();
   }

   private static int intValue(String data) {
     return Integer.parseInt(data);
   }
}
```

Figure 2.1: Basic Java syntax.

For a further treatment of the syntax elements of Java, see also Syntax.

The *syntax* of a Java class is the characters, symbols and their structure used to code the class. Java programs consist of a sequence of tokens. There are different kinds of tokens. For example, there are word tokens such as **`class`** and **`public`** which represent ***keywords* (in purple above)** — special words with reserved meaning in Java. Other words such as `Distance`, `point0`, `x1`, and `printDistance` are not keywords but *identifiers* (in grey). Identifiers have many different uses in Java but primarily they are used as names. Java also has tokens to represent numbers, such as `1` and `3`; these are known as *literals* (in orange). *String literals* (in blue), such as `"Distance between "`, consist of zero or more characters embedded in double quotes, and *operators* (in red) such as `+` and `=` are used to express basic computation such as addition or String concatenation or assignment. There are also left and right braces (`{` and `}`) which enclose *blocks*. The body of a class is one such block. Some tokens are punctuation, such as periods `.` and commas `,` and semicolons `;`. You use *whitespace* such as spaces, tabs, and newlines, to separate tokens. For example, whitespace is required between keywords and identifiers: `publicstatic` is a single identifier with twelve characters, not two Java keywords.

Declarations and Definitions

```
public class Distance {

  private java.awt.Point point0, point1;


  public Distance(int x0, int y0, int x1, int y1) {
    point0 = new java.awt.Point(x0, y0);
    point1 = new java.awt.Point(x1, y1);
  }


  public void printDistance() {
    System.out.println("Distance between " + point0 + " and " + point1
      + " is " + point0.distance(point1));
  }


  public static void main(String[] args) {
    Distance dist = new Distance(
      intValue(args[0]), intValue(args[1]),
      intValue(args[2]), intValue(args[3]));
    dist.printDistance();
  }


  private static int intValue(String data) {
    return Integer.parseInt(data);
  }


}
```

Figure 2.2: Declarations and Definitions.

Sequences of tokens are used to construct the next building blocks of Java classes as shown above: declarations and definitions. A class declaration provides the name and visibility of a class. In our example, `public class Distance` is the class declaration. It consists (in this case) of two keywords, **`public`** and **`class`** followed by the identifier `Distance`.

This means that we are defining a class named `Distance`. Other classes, or in our case, the command line, can refer to the class by this name. The `public` keyword is an access modifier which declares that this class and its members may be accessed from other classes. The `class` keyword, obviously, identifies this declaration as a class. Java also allows declarations of *interfaces* and *annotations*.

The class declaration is then followed by a block (surrounded by curly braces) which provides the class's definition (in blue in figure 2.2). The definition is the implementation of the class – the declaration and definitions of the class's members. This class contains exactly six members, which we will explain in turn.

1. Two field declarations, named `point0` and `point1` (in green)
2. A constructor declaration (in orange)
3. Three method declarations (in red)

Example: Instance Fields

The declaration

Code section 2.1: Declaration.

```
1  private java.awt.Point point0, point1;
```

...declares two *instance fields*. Instance fields represent named values that are allocated whenever an instance of the class is constructed. When a Java program creates a `Distance` instance, that instance will contain space for `point0` and `point1`. When another `Distance` object is created, it will contain space for its *own* `point0` and `point1` values. The value of `point0` in the first `Distance` object can vary independently of the value of `point0` in the second `Distance` object.

This declaration consists of:

1. The `private` access modifier,
   which means these instance fields are not visible to other classes.
2. The type of the instance fields. In this case, the type is `java.awt.Point`.
   This is the class `Point` in the `java.awt` package.
3. The names of the instance fields in a comma separated list.

These two fields could also have been declared with two separate but more verbose declarations,

Code section 2.2: Verbose declarations.

```
1  private java.awt.Point point0;
2  private java.awt.Point point1;
```

Since the type of these fields is a reference type (i.e. a field that *refers to* or can hold a *reference to* an object value), Java will implicitly initialize the values of `point0` and `point1` to null when a `Distance` instance is created. The null value means that a reference value does not refer to an object. The special Java literal **`null`** is used to represent the null value in a program. While you can explicitly assign null values in a declaration, as in

Code section 2.3: Declarations and assignments.

```
1  private java.awt.Point point0 = null;
2  private java.awt.Point point1 = null;
```

It is not necessary and most programmers omit such default assignments.

Example: Constructor

A *constructor* is a special method in a class which is used to construct an instance of the class. The constructor can perform initialization for the object, beyond that which the Java VM does automatically. For example, Java will automatically initialize the fields `point0` and `point1` to null.

Code section 2.4: The constructor for the class

```
1  public Distance(int x0, int y0, int x1, int y1) {
2     point0 = new java.awt.Point(x0, y0);
3     point1 = new java.awt.Point(x1, y1);
4  }
```

The constructor above consists of five parts:

1. The optional access modifier(s).
   In this case, the constructor is declared `public`
2. The constructor name, which must match the class name exactly: `Distance` in this case.
3. The constructor parameters.
   The parameter list is required. Even if a constructor does not have any parameters, you must specify the empty list `()`. The parameter list declares the type and name of each of the method's parameters.
4. An optional **`throws`** clause which declares the exceptions that the constructor may throw. This constructor does not declare any exceptions.
5. The constructor body, which is a Java block (enclosed in `{}`). This constructor's body contains two statements.

This constructor accepts four parameters, named `x0, y0, x1` and `y1`. Each parameter requires a parameter type declaration, which in this example is **`int`** for all four parameters. The parameters in the parameter list are separated by commas.

The two assignments in this constructor use Java's ***`new`*** *operator* to allocate two `java.awt.Point` objects. The first allocates an object representing the first point, `(x0, y0)`, and assigns it to the `point0` instance variable (replacing the null value that the instance variable was initialized to). The second statement allocates a second `java.awt.Point` instance with `(x1, y1)` and assigns it to the `point1` instance variable.

This is the constructor for the Distance class. Distance implicitly extends from `java.lang.Object`. Java inserts a call to the super constructor as the first executable statement of the constructor if there is not one explicitly coded. The above constructor body is equivalent to the following body with the explicit super constructor call:

Code section 2.5: Super constructor.

```
{
   super();
   point0 = new java.awt.Point(x0, y0);
   point1 = new java.awt.Point(x1, y1);
}
```

While it is true that this class could be implemented in other ways, such as simply storing the coordinates of the two points and computing the distance as $\sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}$, this class instead uses the existing `java.awt.Point` class. This choice matches the abstract definition of this class: to print the distance between two points on the plane. We take advantage of existing behavior already implemented in the Java platform rather than implementing it again. We will see later how to make the program more flexible without adding much complexity, because we choose to use object abstractions here. However, the key point is that this class uses information hiding. That is, *how* the class stores its state or how it computes the distance is hidden. We can change this implementation without altering how clients use and invoke the class.

Example: Methods

Methods are the third and most important type of class member. This class contains three *methods* in which the behavior of the `Distance` class is defined: `printDistance()`, `main()`, and `intValue()`

The printDistance() method

The `printDistance()` method prints the distance between the two points to the standard output (normally the console).

Code section 2.6: `printDistance()` method.

```
public void printDistance() {
  System.out.println("Distance between " + point0
    + " and " + point1
    + " is " + point0.distance(point1));
}
```

This *instance method* executes within the context of an implicit `Distance` object. The instance field references, `point0` and `point1`, refer to instance fields of that implicit object. You can also use the special variable `this` to explicitly reference the current object. Within an instance method, Java binds the name `this` to the object on which the method is executing, and the type of `this` is that of the current class. The body of the `printDistance` method could also be coded as

Code section 2.7: Explicit instance of the current class.

```
1    System.out.println("Distance between " + this.point0
2      + " and " + this.point1
3      + " is " + this.point0.distance(this.point1));
```

to make the instance field references more explicit.

This method both computes the distance and prints it in one statement. The distance is computed with `point0.distance(point1);` `distance()` is an instance method of the `java.awt.Point` class (of which `point0` and `point1` are instances). The method operates on `point0` (binding `this` to the object that `point0` refers to during the execution of the method) and accepting another Point as a parameter. Actually, it is slightly more complicated than that, but we'll explain later. The result of the `distance()` method is a double precision floating point number.

This method uses the syntax

Code section 2.8: String concatenation.

```
1   "Distance between " + this.point0
2   + " and " + this.point1
3   + " is " + this.point0.distance(this.point1)
```

to construct a String to pass to the `System.out.println()`. This expression is a series of *String concatenation* methods which concatenates Strings or the String representation of primitive types (such as doubles) or objects, and returns a long string. For example, the result of this expression for the points (0,3) and (4,0) is the String

Output

```
"Distance between java.awt.Point[x=0,y=3] and java.awt.Point[x=4,y=0] is 5.0"
```

which the method then prints to `System.out`.

In order to print, we invoke the `println()`. This is an instance method from `java.io.PrintStream`, which is the type of the static field `out` in the class `java.lang.System`. The Java VM binds `System.out` to the standard output stream when it starts a program.

The main() method

The `main()` method is the main entry point which Java invokes when you start a Java program from the command line. The command

Output

```
java Distance 0 3 4 0
```

instructs Java to locate the Distance class, put the four command line arguments into an array of String values, then pass those arguments to the `public static main(String[])` method of the class. We will introduce arrays shortly. Any Java class that you want to invoke from the command line or desktop shortcut must have a main method with this signature or the following signature: `public static main(String...)`.

Code section 2.9: `main()` method.

```
public static void main(String[] args) {
   Distance dist = new Distance(
         intValue(args[0]), intValue(args[1]),
         intValue(args[2]), intValue(args[3]));
   dist.printDistance();
}
```

The `main()` method invokes the final method, `intValue()`, four times. The `intValue()` takes a single string parameter and returns the integer value represented in the string. For example, `intValue("3")` will return the integer 3.

People who do test-first programming or perform regression testing write a main() method in every Java class, and a main() function in every Python module, to run automated tests. When a person executes the file directly, the main() method executes and runs the automated tests for that file. When a person executes some other Java file that in turn imports many other Java classes, only one main() method is executed -- the main() method of the directly-executed file.

The `intValue()` method

The `intValue()` method delegates its job to the `Integer.parseInt()` method. The main method could have called `Integer.parseInt()` directly; the `intValue()` method simply makes the `main()` method slightly more readable.

Code section 2.10: `intValue()` method.

```
private static int intValue(String data) {
   return Integer.parseInt(data);
}
```

This method is **`private`** since, like the fields `point0` and `point1`, it is part of the internal implementation of the class and is not part of the external programming interface of the `Distance` class.

Static vs. Instance Methods

Both the `main()` and `intValue()` methods are *static methods*. The **`static`** keyword tells the compiler to create a single memory space associated with the class. Each individual object instantiated has its own private state variables and methods but use the same **static** methods and members common to the single class object created by the compiler when the first class object is instantiated or created. This means that the method executes in a static or non-object context — there is no implicit separate instance available when the static methods run from various objects, and the special variable `this` is not available. As such, static methods cannot access instance methods or instance fields (such as `printDistance()`) or `point0`) directly. The `main()` method can only invoke the instance method `printDistance()` method via an instance reference such as `dist`.

Data Types

Most declarations have a data type. Java has several categories of data types: reference types, primitive types, array types, and a special type, void.

Primitive Types

The *primitive types* are used to represent boolean, character, and numeric values. This program uses only one primitive type explicitly, ***`int`***, which represents 32 bit signed integer values. The program also implicitly uses ***`double`***, which is the return type of the `distance()` method of `java.awt.Point`. `double` values are 64 bit IEEE floating point values. The `main()` method uses integer values 0, 1, 2, and 3 to access elements of the command line arguments. The `Distance()` constructor's four parameters also have the type `int`. Also, the `intValue()` method has a return type of `int`. This means a call to that method, such as `intValue(args[0])`, is an expression of type `int`. This helps explain why the main method cannot call:

Code section 2.11: Wrong type.

```
new Distance(args[0], args[1], args[2], args[3]) // This is an error
```

Since the type of the `args` array element is String, and our constructor's parameters must be `int`, such a call would result in an error because Java will not automatically convert values of type String into `int` values.

Java's primitive types are ***`boolean`***, ***`byte`***, ***`char`***, ***`short`***, ***`int`***, ***`long`***, ***`float`*** and ***`double`***. Each of which are also Java language keywords.

Reference Types

In addition to primitive types, Java supports *reference type*. A reference type is a Java data type which is defined by a Java class or interface. Reference types derive this name because such values *refer to* an object or contain a *reference to* an object. The idea is similar to pointers in other languages like C.

Java represents sequences of character data, or *String*, with the reference type `java.lang.String` which is most commonly referred to as `String`. *String literals*, such as `"Distance between "` are constants whose type is String.

This program uses three separate reference types:

1. java.lang.String (or simply String)
2. Distance
3. java.awt.Point

> *For more information see chapter: Java Programming/Classes, Objects and Types.*

Array Types

Java supports *arrays*, which are aggregate types which have a fixed element type (which can be any Java type) and an integral size. This program uses only one array, `String[] args`. This indicates that `args` has an array type and that the element type is `String`. The Java VM constructs and initializes the array that is passed to the `main` method. See *arrays* for more details on how to create arrays and access their size.

The elements of arrays are accessed with integer indices. The first element of an array is always element 0. This program accesses the first four elements of the `args` array explicitly with the indices 0, 1, 2, and 3. This program does *not* perform any input validation, such as verifying that the user passed at least four arguments to the program. We will fix that later.

void

`void` is not a type in Java; it represents the absence of a type. Methods which do not return values are declared as *void methods*.

This class defines two void methods:

Code section 2.12: Void methods

```
1  public static void main(String[] args) { ... }
2  public void printDistance()  { ... }
```

Whitespace

Whitespace in Java is used to separate the tokens in a Java source file. Whitespace is required in some places, such as between access modifiers, type names and Identifiers, and is used to improve readability elsewhere.

Wherever whitespace is required in Java, one or more whitespace characters may be used. Wherever whitespace is optional in Java, zero or more whitespace characters may be used.

Java whitespace consists of the

- space character ' ' (0x20),
- the tab character (hex 0x09),

- the form feed character (hex 0x0c),
- the line separators characters newline (hex 0x0a) or carriage return (hex 0x0d) characters.

Line separators are special whitespace characters in that they also terminate line comments, whereas normal whitespace does not.

Other Unicode space characters, including vertical tab, are not allowed as whitespace in Java.

Required Whitespace

Look at the **`static`** method `intValue`:

Code section 2.13: Method declaration

```
1  private static int intValue(String data) {
2    return Integer.parseInt(data);
3  }
```

Whitespace is required between **`private`** and **`static`**, between **`static`** and **`int`**, between **`int`** and `intValue`, and between `String` and `data`.

If the code is written like this:

Code section 2.14: Collapsed code

```
1  privatestaticint intValue(String data) {
2    return Integer.parseInt(data);
3  }
```

...it means something completely different: it declares a method which has the return type `privatestaticint` It is unlikely that this type exists and the method is no longer static, so the above would result in a semantic error.

Indentation

Java ignores all whitespace in front of a statement. As this, these two code snippets are identical for the compiler:

Code section 2.15: Indented code

```
1  public static void main(String[] args) {
2    Distance dist = new Distance(
3              intValue(args[0]), intValue(args[1]),
4              intValue(args[2]), intValue(args[3]));
5    dist.printDistance();
6  }
7
8  private static int intValue(String data) {
9    return Integer.parseInt(data);
10 }
```

Code section 2.16: Not indented code

```
1  public static void main(String[] args) {
2  Distance dist = new Distance(
3  intValue(args[0]), intValue(args[1]),
4  intValue(args[2]), intValue(args[3]));
5  dist.printDistance();
6  }
7
8  private static int intValue(String data) {
9  return Integer.parseInt(data);
10 }
```

However, the first one's style (with whitespace) is preferred, as the readability is higher. The method body is easier to distinguish from the head, even at a higher reading speed.

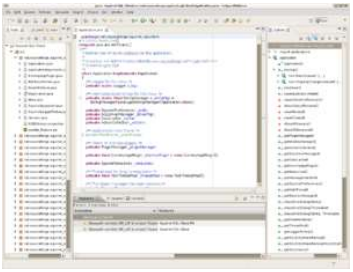
Java IDEs

What is a Java IDE?

A Java IDE (for *Integrated Development Environment*) is a software application which enables users to more easily write and debug Java programs. Many IDEs provide features like syntax highlighting and code completion, which help the user to code more easily.

Eclipse

Eclipse is a Free and Open Source IDE, plus a developer tool framework that can be extended for a particular development need. IBM was behind its development, and it replaced IBM VisualAge tool. The idea was to create a standard look and feel that can be extended via plugins. The extensibility distinguishes Eclipse from other IDEs. Eclipse was also meant to compete with Microsoft Visual Studio tools. Microsoft tools give a standard way of developing code in the Microsoft world. Eclipse gives a similar standard way of developing code in the Java world, with a big success so far. With the online error checking only, coding can be sped up by at least 50% (coding does not include programming).



Eclipse on Ubuntu

The goals for Eclipse are twofold:

1. Give a standard IDE for developing code
2. Give a starting point, and the same look and feel for all other more sophisticated tools built on Eclipse

IBM's WSAD, and later IBM Rational Software Development Platform, are built on Eclipse.

Standard Eclipse features:

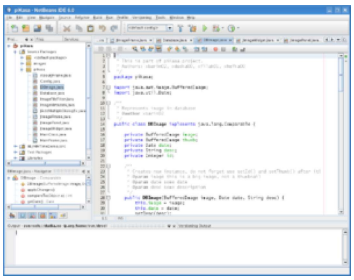
- Standard window management (perspectives, views, browsers, explorers, ...)
- Error checking as you type (immediate error indications, ...)
- Help window as you type (type ., or `<ctrl> space`, ...)
- Automatic build (changes in source code are automatically compiled, ...)
- Built-in debugger (full featured GUI debugger)
- Source code generation (getters and setters, ...)
- Searches (for implementation, for references, ...)
- Code refactoring (global reference update, ...)
- Plugin-based architecture (ability to build tools that integrate seamlessly with the environment, and some other tools)
- ...

More info: Eclipse (http://www.eclipse.org/) and Plugincentral (http://www.eclipseplugincentral.com/).

NetBeans

The NetBeans IDE is a Free and Open Source IDE for software developers. The IDE runs on many platforms including Windows, GNU/Linux, Solaris and Mac OS X. It is easy to install and use straight out of the box. You can easily create Java applications for mobile devices using Mobility Pack in NetBeans. With Netbeans 6.0, the IDE has become one of the most preferred development tools, whether it be designing a Swing UI, building a mobile application, an enterprise application or using it as a platform for creating your own IDE.

More info: netbeans.org (http://www.netbeans.org/products/ide/)



NetBeans on GNU/Linux

JCreator

JCreator is a simple and lightweight JAVA IDE from XINOX Software. It runs only on Windows platforms. It is very easy to install and starts quickly, as it is a native application. This is a good choice for beginners.

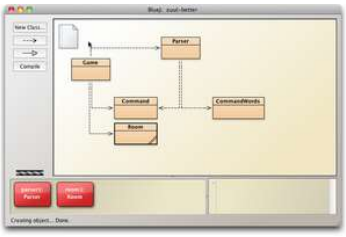
More info: http://www.apcomputerscience.com/ide/jcreator/index.htm or JCreator (http://www.jcreator.com/)

Processing

Processing is an **enhanced** IDE. It adds some extra commands and a simplified programming model. This makes it much easier for beginners to start programming in Java. It was designed to help graphic artists learn a bit of programming without struggling too much. Processing runs on Windows, GNU/Linux and Mac OS X platforms.

More info: Processing (http://www.processing.org).

BlueJ



BlueJ on Mac OS X

BlueJ is an IDE that includes templates and will compile and run the applications for you. BlueJ is often used by classes because it is not necessary to set classpaths. BlueJ has its own sets of libraries and you can add your own under preferences. That sets the classpath for all compilations that come out of it to include those you have added and the BlueJ libraries.

BlueJ offers an interesting GUI for creation of packages and programs. Classes are represented as boxes with arrows running between them to represent inheritance/implementation or if on is constructed in another. The source code is generated by the UML diagram or vice-versa. BlueJ adds all those classes (the project) into the classpath at compile time.

By default it doesn't display the line numbers, so this should be ticked into Options\Preferences...

More info: BlueJ Homesite (http://www.bluej.org)

Kawa

Kawa is basically a Java editor developed by Tek-Tools. It does not include wizards and GUI tools, best suited to experienced Java programmers in small and midsized development teams. It looks that there is no new development for Kawa.

See also a javaworld article (http://www.javaworld.com/javaworld/jw-06-2000/jw-0602-iw-kawa.html)

JBuilder

JBuilder is an IDE with proprietary source code, sold by Embarcadero Technologies. One of the advantages is the integration with Together, a modeling tool.

More info: Embarcadero (http://www.embarcadero.com/).

DrJava

DrJava is an IDE developed by the JavaPLT group at Rice University. It is designed for students.

For more information see DrJava (http://www.drjava.org).

Other IDEs

- Geany
- IntelliJ IDEA
- JDeveloper
- jGRASP
- jEdit
- MyEclipse
- Visual Café
- Gel (http://www.gexperts.com/products/gel/download.php)
- JIPE (http://jipe.sourceforge.net/)
- Zeus (http://www.zeusedit.com/java.html)
- Setu Eye Saving Lightweight(fast)C,C++,JAVA IDE (http://www.setuide.byethost14.com/SetuIDEeyesaving.htm)

Language Fundamentals

The previous chapter *"Getting started"* was a primer course in the basics of understanding how Java programming works. Throughout the chapter, we tackled a variety of concepts that included:

- Objects and class definitions;
- Abstract and data types;
- Properties;
- Methods;
- Class-level and method-level scopes;
- Keywords; and,
- Access modifiers, etc.

From this point on, we will be looking into the above mentioned concepts and many more in finer detail with a deeper and richer understanding of how each one of them works. This chapter on **Language fundamentals** introduces the fundamental elements of the Java programming language in detail. The discussions in this chapter will use the concepts we have already gathered from our previous discussions and build upon them in a progressive manner.

The Java programming syntax

In linguistics, the word **syntax** (which comes from Ancient Greek *σύνταξις* where *σύν* [syn] means "together", and *τάξις* [táxis] means "an ordering") refers to "the process of arranging things". It defines the principles and rules for constructing phrases and sentences in natural languages.

When learning a new language, the first step one must take is to learn its **programming syntax**. *Programming syntax* is to programming languages what *grammar* is to spoken languages. Therefore, in order to create effective code in the Java programming language, we need to learn its syntax — its principles and rules for constructing valid code statements and expressions.

Java uses a syntax similar to the C programming language and therefore if one learns the Java programming syntax, they automatically would be able to read and write programs in similar languages — C, C++ and C#

The next step one must take when learning a new language is to learn its keywords; by combining the knowledge of keywords with an understanding of syntax rules, one can create statements, Programming Blocks, Classes, Interfaces, et al.

Use packages to avoid name collisions. To hide as much information as possible use the access modifiers properly.

Create methods that do one and if possible only one thing/task. If possible have separate method that changes the object state.

In an object oriented language, programs are run with objects; however, for ease of use and for historic reasons, Java has primitive types. Primitive Data Types only store values and have no methods. Primitive Types may be thought of as Raw Data and are usually embedded attributes inside objects or used as local variables in methods. Because primitive types are not subclasses of the object superclass, each type has a Wrapper Class which is a subclass of Object, and can thus be stored in a collection or returned as an object.

Java is a strong type checking language. There are two concepts regarding types and objects. One is the object type and the other the template/class the object was created from. When an object is created, the template/class is assigned to that object which can not be changed. Types of an object however can be changed by type casting. Types of an object is associated with the object reference that referencing the object and determines what operation can be performed on the object through that object reference. Assigning the value of one object reference to a different type of object reference is called type casting.

The most often used data structure in any language is a character string. For this reason java defines a special object that is String.

To aggregate same type java objects to an array, java has a special array object for that. Both java objects and primitive types can be aggregated to arrays.

Statements

Now that we have the Java platform on our systems and have run the first program successfully, we are geared towards understanding how programs are actually made. As we have already discussed, a program is a set of instructions, which are tasks provided to a computer. These instructions are called **statements** in Java. Statements can be anything from a single line of code to a complex mathematical equation. Consider the following line:

Code section 3.1: A simple assignment statement.

```
1  int age = 24;
```

This line is a simple instruction that tells the system to initialize a variable and set its value as 24. If the above statement was the only one in the program, it would look similar to this:

Code listing 3.1: A statement in a simple class.

```
1  public class MyProgram
2   {
3      public static void main(String[] args)
4      {
5          int age = 24;
6      }
7  }
```

Java places its statements within a class declaration and, in the class declaration, the statements are usually placed in a method declaration, as above.

Variable declaration statement

The simplest statement is a variable declaration:

Code section 3.2: A simple declaration statement.

```
1  int age;
```

It defines a variable that can be used to store values for later use. The first token is the data type of the variable (which type of values this variable can store). The second token is the name of the variable, by which you will be referring to it. Then each declaration statement is ended by a semicolon (`;`).

Assignment statements

Up until now, we've assumed the creation of variables as a single statement. In essence, we assign a value to those variables, and that's just what it is called. When you assign a value to a variable in a statement, that statement is called an **assignment statement** (also called an initialization statement). Did you notice one more thing? It's the semicolon (`;`), which is at the end of each statement. A clear indicator that a line of code is a statement is its termination with an ending semicolon. If one was to write multiple statements, it is usually done with each statement on a separate line ending with a semicolon. Consider the example below:

Code section 3.3: Multiple assignment statements.

```
1  int a = 10;
2  int b = 20;
3  int c = 30;
```

You do not necessarily have to use a new line to write each statement. Just like English, you can begin writing the next statement where you ended the first one as depicted below:

Code section 3.4: Multiple assignment statements on the same line.

```
1  int a = 10; int b = 20; int c = 30;
```

However, the only problem with putting multiple statements on one line is, it's very difficult to read it. It doesn't look that intimidating at first, but once you've got a significant amount of code, it's usually better to organize it in a way that makes sense. It would look more complex and incomprehensible written as it is in Listing 3.4.

Now that we have looked into the anatomy of a simple assignment statement, we can look back at what we've achieved. We know that...

- A statement is a unit of code in programming.
- If we are assigning a variable a value, the statement is called an assignment statement.
- An assignment statement includes three parts: a data type, the variable name (also called the identifier) and the value of a variable. We will look more into the nature of identifiers and values in the section *Variables* later.

Now, before we move on to the next topic, you need to try and understand what the code below does.

Code section 3.5: Multiple assignment statements with expressions.

```
1  int firstNumber = 10;
2  int secondNumber = 20;
3  int result = firstNumber + secondNumber;
4  System.out.println(result);
5  secondNumber = 30;
6  System.out.println(result);
```

The first two statements are pretty much similar to those in Section 3.3 but with different variable names. The third however is a bit interesting. We've already talked of variables as being similar to gift boxes. Think of your computer's memory as a shelf where you put all those boxes. Whenever you need a box (or variable), you call its identifier (that's the name of the variable). So calling the variable identifier `firstNumber` gives you the number `10`, calling `secondNumber` would give you `20` hence when you add the two up, the answer should be `30`. That's what the value of the last variable `result` would be. The part of the third statement where you add the numbers, i.e., `firstNumber +`

`secondNumber` is called an **expression** and the expression is what decides what the value is to be. If it's just a plain value, like in the first two statements, then it's called a **literal** (the value is *literally* the value, hence the name *literal*).

Note that after the assignment to `result` its value will not be changed if we assign different values to `firstNumber` or `secondNumber`, like in line 5.

With the information you have just attained, you can actually write a decent Java program that can sum up values.

Assertion

An assertion checks if a condition is true:

Code section 3.6: A return statement.

```
1      public int getAge()
2      {
3          assert age >= 0;
4          return age;
5      }
```

Each `assert` statement is ended by a semi-colon (`;`). However, assertions are disabled by default, so you must run the program with the *-ea* argument in order for assertions to be enabled (`java -ea [name of compiled program]`).

Program Control Flow

Statements are evaluated in the order as they occur. The execution of flow begins at the top most statement and proceed downwards till the last statement is encountered. A statement can be substituted by a statement block. There are special statements that can redirect the execution flow based on a condition, those statements are called *branching* statements, described in detail in a later section.

Statement Blocks

A bunch of statements can be placed in braces to be executed as a single block. Such a block of statements can be named or be provided with a condition for execution. Below is how you'd place a series of statements in a block.

Code section 3.7: A statement block.

```
1  {
2      int a = 10;
3      int b = 20;
4      int result = a + b;
5  }
```

Branching Statements

Program flow can be affected using function/method calls, loops and iterations. Of various types of branching constructs, we can easily pick out two generic branching methods.

- Unconditional Branching
- Conditional Branching

Unconditional Branching Statements

If you look closely at a method, you'll see that a method is a named statement block that is executed by calling that particular name. An unconditional branch is created either by invoking the method or by calling `break`, `continue`, `return` or `throw`, all of which are described below.

When the name of another method is encountered in a flow, it stops execution in the current method and branches to the newly called method. After returning a value from the called method, execution picks up at the original method on the line below the method call.

Code listing 3.8: UnconditionalBranching.java

```
1  public class UnconditionalBranching {
2      public static void main(String[]
   args) {
3          System.out.println("Inside main
   method! Invoking aMethod!");
4          aMethod();
5          System.out.println("Back in main
   method!");
6      }
7
8      public static void aMethod() {
9          System.out.println("Inside
   aMethod!");
10     }
11 }
```

Output provided with the screen of information running the above code.

```
Inside main method! Invoking aMethod!
Inside aMethod!
Back in main method!
```

The program flow begins in the `main` method. Just as `aMethod` is invoked, the flow travels to the called method. At this very point, the flow branches to the other method. Once the method is completed, the flow is returned to the point it left off and resumes at the next statement after the call to the method.

Return statement

A `return` statement exits from a block, so it is often the last statement of a method:

Code section 3.9: A return statement.

```
1      public int getAge() {
2          int age = 24;
3          return age;
4      }
```

A return statement can return the content of a variable or nothing. Beware not to write statements after a return statement which would not be executed! Each `return` statement is ended by a semi-colon (`;`).

Conditional Branching Statements

Conditional branching is attained with the help of the `if...else` and `switch` statements. A conditional branch occurs only if a certain condition expression evaluates to true.

Conditional Statements

Also referred to as ***if*** *statements*, these allow a program to perform a test and then take action based on the result of that test.

The form of the **`if`** statement:

```
if (condition) {
  do statements here if condition is true
} else {
  do statements here if condition is false
}
```

The *condition* is a boolean expression which can be either **`true`** or **`false`**. The actions performed will depend on the value of the condition.

Example:

Code section 3.10: An if statement.

```
1  if (i > 0) {
2     System.out.println("value stored in i is greater than zero");
3  } else {
4     System.out.println("value stored is not greater than zero");
5  }
```

If statements can also be made more complex using the else if combination:

```
if (condition 1) {
   do statements here if condition 1 is true
} else if (condition 2) {
   do statements here if condition 1 is false and condition 2 is true
} else {
  do statements here if neither condition 1 nor condition 2 is true
}
```

Example:

Code section 3.11: An if/else if/else statement.

```
1  if (i > 0) {
2     System.out.println("value stored in i is greater than zero");
3  } else if (i < 0) {
4     System.out.println("value stored in i is less than zero");
5  } else {
6     System.out.println("value stored is equal to 0");
7  }
```

If there is only one statement to be executed after the condition, as in the above example, it is possible to omit the curly braces, however Oracle's Java Code Conventions (http://www.oracle.com/technetwork/java/index.html#449) explicitly state that the braces should always be used.

There is no looping involved in an if statement so once the condition has been evaluated the program will continue with the next instruction after the statement.

If...else statements

The **`if`** ... **`else`** statement is used to conditionally execute one of two blocks of statements, depending on the result of a boolean condition.

Example:

Code section 3.12: An if/else statement.

```
1  if (list == null) {
2    // This block of statements executes if the condition is true.
3  } else {
4    // This block of statements executes if the condition is false.
5  }
```

Oracle's Java Code Conventions (http://www.oracle.com/technetwork/java/index.html#449) recommend that the braces should always be used.

An **`if`** statement has two forms:

```
if (boolean-condition)
   statement1
```

and

```
if (boolean-condition)
   statement1
else
   statement2
```

Use the second form if you have different statements to execute if the *boolean-condition* is true or if it is false. Use the first if you only wish to execute $statement_1$ if the condition is true and you do not wish to execute alternate statements if the condition is false.

The code section 3.13 calls two **int** methods, `f()` and `y()`, stores the results, then uses an **if** statement to test if `x` is less than y and if it is, the $statement_1$ body will swap the values. The end result is x always contains the larger result and y always contains the smaller result.

Code section 3.13: Value swap.

```
1  int x = f();
2  int y = y();
3  if (x < y) {
4    int z = x;
5    x = y;
6    y = z;
7  }
```

if...**else** statements also allow for the use of another statement, **else if**. This statement is used to provide another **if** statement to the conditional that can only be executed if the others are not true. For example:

Code section 3.14: Multiple branching.

```
1  if (x == 2)
2    x = 4;
3  else if (x == 3)
4    x = 6;
5  else
6    x = -1;
```

The **else if** statement is useful in this case because if one of the conditionals is true, the other must be false. Keep in mind that if one is true, the other *will not* execute. For example, if the statement at line `2` contained in the first conditional were changed to `x = 3;`, the second conditional, the **else if**, would still not execute. However, when dealing with primitive types in conditional statements, it is more desirable to use `switch statements` rather than multiple **else if** statements.

Switch statements

The **switch** conditional statement is basically a shorthand version of writing many **if**...**else** statements. The syntax for **switch** statements is as follows:

```
switch(<variable>) {
  case <result>: <statements>; break;
  case <result>: <statements>; break;
  default: <statements>; break;
}
```

This means that if the variable included equals one of the case results, the statements following that case, until the word `break` will run. The `default` case executes if none of the others are true. **Note:** the only types that can be analysed through **switch** statements are **char**, **byte**, **short**, or **int** primitive types. This means that `Object` variables can not by analyzed through **switch** statements. However, as of the JDK 7 release, you can use a String object in the expression of a switch statement.

Code section 3.15: A switch.

```
1  int n = 2, x;
2  switch (n) {
3    case 1: x = 2;
4      break;
5    case 2: x = 4;
6      break;
7    case 3: x = 6;
8      break;
9    case 4: x = 8;
10     break;
11
```

```
    }
12  return x;
```

In this example, since the integer variable n is equal to 2, `case 2` will execute, make x equal to 4. Thus, 4 is returned by the method.

Iteration Statements

Iteration Statements are statements that are used to iterate a block of statements. Such statements are often referred to as loops. Java offers four kinds of iterative statements.

- The `while` loop
- The `do...while` loop
- The `for` loop
- The `foreach` loop

The while loop

The `while` loop iterates a block of code while the condition it specifies is `true`.

The syntax for the loop is:

```
while (condition) {
   statement;
 }
```

Here the condition is an expression. An expression as discussed earlier is any statement that returns a value. `While` condition statements evaluate to a boolean value, that is, either `true` or `false`. As long as the condition is `true`, the loop will iterate the block of code over and over and again. Once the condition evaluates to `false`, the loop exits to the next statement outside the loop.

The do...while loop

The do-while loop is functionally similar to the while loop, except the condition is evaluated AFTER the statement executes

```
do {
   statement;
 } while (condition);
```

The for loop

The for loop is a specialized while loop whose syntax is designed for easy iteration through a sequence of numbers. Example:

Code section 3.16: A for loop.

```
1  for (int i = 0; i < 100;
   i++) {
2    System.out.println(i +
   "\t" + i * i);
3  }
```

Output for code listing 3.16 if you compile and run the statement above.

```
0     0
1     1
2     4
3     9
...
99    9801
```

The program prints the numbers 0 to 99 and their squares.

The same statement in a while loop:

Code section 3.17: An alternative version.

```
int i = 0;
while (i < 100) {
   System.out.println(i + "\t" + i * i);
   i++;
}
```

The foreach loop

The foreach statement allows you to iterate through all the items in a collection, examining each item in turn while still preserving its type. The syntax for the foreach statement is:

```
for (type item : collection) statement;
```

For an example, we'll take an array of `Strings` denoting days in a week and traverse through the collection, examining one item at a time.

Code section 3.18: A foreach loop.

```
String[] days = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
"Saturday", "Sunday"};

for (String day : days) {
  System.out.println(day);
}
```

Output for code listing 3.18

```
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
```

Notice that the loop automatically exits after the last item in the collection has been examined in the statement block.

Although the enhanced for loop can make code much clearer, it can't be used in some common situations.

- **Only access.** Elements can not be assigned to, eg, not to increment each element in a collection.
- **Only single structure.** It's not possible to traverse two structures at once, eg, to compare two arrays.
- **Only single element.** Use only for single element access, eg, not to compare successive elements.
- **Only forward.** It's possible to iterate only forward by single steps.
- **At least Java 5.** Don't use it if you need compatibility with versions before Java 5.

The continue and break statements

At times, you would like to re-iterate a loop without executing the remaining statement within the loop. The **`continue`** statement causes the loop to re-iterate and start over from the top most statement inside the loop.

Where there is an ability to re-iterate the loop, there is an ability to exit the loop when required. At any given moment, if you'd like to exit a loop and end all further work within the loop, the **`break`** ought to be used.

The **`continue`** and **`break`** statements can be used with a label like follows:

Code section 3.19: Using a label.

```
String s = "A test string for the switch!\nLine two of test
string...";
outer: for (int i = 0; i < s.length(); i++) {
  switch (s.charAt(i)) {
    case '\n': break outer;
    case ' ': break;
    default: System.out.print(s.charAt(i));
  }
}
```

Output for code listing 3.19

```
AteststringfortheswitchLine two of test
string...
```

Throw statement

A `throw` statement exits from a method and so on and so on or it is caught by a `try/catch` block. It does not return a variable but an exception:

Code section 3.20: A return statement.

```
    public int getAge() {
        throw new NullPointerException();
    }
```

Beware not to write statements after a `throw` statement which would not be executed too! Each `throw` statement is ended by a semi-colon (`;`).

try/catch

A `try/catch` must at least contain the `try` block and the `catch` block:

Code section 3.21: try/catch block.

```
try {
  // Some code
} catch (Exception e) {
  // Optional exception handling
} finally {
  // This code is executed no matter what
}
```

Test your knowledge

Question 3.1: How many statements are there in this class?

Code listing 3.2: AProgram.java

```
public class AProgram {

    private int age = 24;

    public static void main(String[] args) {
        int daysInAYear = 365;int ageInDay = 100000;
        int localAge = ageInDay / daysInAYear;
    }

    public int getAge() {
        return age;
    }
}
```

Answer

5
One statement at line 3, two statements at line 6, one statement at line 7 and one statement at line 11.

Conditional blocks

Conditional blocks allow a program to take a different path depending on some condition(s). These allow a program to perform a test and then take action based on the result of that test. In the code sections, the actually executed code lines will be highlighted.

If

The **`if`** block executes only if the boolean expression associated with it is true. The structure of an `if` block is as follows:

if (***boolean expression*$_1$**) {

***statement*$_1$**
***statement*$_2$**
...
***statement*$_n$**

}

Here is a double example to illustrate what happens if the condition is true and if the condition is false:

Code section 3.22: Two `if` blocks.

```
1  int age = 6;
2  System.out.println("Hello!");
3
4  if (age < 13) {
5    System.out.println("I'm a child.");
6  }
7
8  if (age > 20) {
9    System.out.println("I'm an adult.");
10 }
11
12 System.out.println("Bye!");
```

`Output for Code section 3.22`

```
Hello!
I'm a child
Bye!
```

If only one statement is to be executed after an `if` block, it does not have to be enclosed in curly braces. For example, `if (i == 0) i = 1;` is a perfectly valid portion of Java code. This works for most control structures, such as `else` and `while`. However Oracle's Java Code Conventions (http://www.oracle.com/technetwork/java/index.html#449) explicitly state that the braces should always be used.

If/else

The `if` block may optionally be followed by an **`else`** block which will execute if that boolean expression is false. The structure of an `if` block is as follows:

if (***boolean expression*$_1$**) {

***statement*$_1$**
***statement*$_2$**
...
***statement*$_n$**

} else {

***statement*$_{1bis}$**
***statement*$_{2bis}$**
...
***statement*$_{nbis}$**

}

If/else-if/else

An `else-if` block may be used when multiple conditions need to be checked. `else-if` statements come after the `if` block, but before the `else` block. The structure of an `if` block is as follows:

if (***boolean expression*$_1$**) {

***statement*$_{1.1}$**
***statement*$_{1.2}$**
...
***statement*$_n$**

} else if (***boolean expression*$_2$**) {

***statement*$_{2.1}$**
***statement*$_{2.2}$**
...
***statement*$_{2.n}$**

} else {

***statement*$_{3.1}$**
***statement*$_{3.2}$**
...
***statement*$_{3.n}$**

}

Here is an example to illustrate:

Code listing 3.3: MyConditionalProgram.java

```
1  public class MyConditionalProgram {
2      public static void main (String[] args) {
3        int a = 5;
4        if (a > 0) {
5            // a is greater than 0, so this statement will execute
6            System.out.println("a is positive");
7        } else if (a >= 0) {
8            // a case has already executed, so this statement will NOT
   execute
9            System.out.println("a is positive or zero");
10       } else {
11           // a case has already executed, so this statement will NOT
   execute
12           System.out.println("a is negative");
13       }
14     }
15 }
```

Output for code listing 3.3

```
a is positive
```

Keep in mind that *only a single block* will execute, and it will be the first true condition.

All the conditions are evaluated when `if` is reached, no matter what the result of the condition is, after the execution of the `if` block:

Code section 3.23: A new value for the variable a.

```
1  int a = 5;
2  if (a > 0) {
3      // a is greater than 0, so this statement will execute
4      System.out.println("a is positive");
5      a = -5;
6  } else if (a < 0) {
7      // a WAS greater than 0, so this statement will not execute
8      System.out.println("a is negative");
9  } else {
10     // a does not equal 0, so this statement will not execute
11     System.out.println("a is zero");
12 }
```

Output for code section 3.23

```
a is positive
```

Conditional expressions

Conditional expressions use the compound `?:` operator. Syntax:

$$\textit{boolean expression}_1 \ ? \ \textit{expression}_1 \ : \ \textit{expression}_2$$

This evaluates `boolean expression`$_1$, and if it is `true` then the conditional expression has the value of `expression`$_1$; otherwise the conditional expression has the value of `expression`$_2$.

Example:

Code section 3.24: Conditional expressions.

```
1  String answer = (p < 0.05)? "reject" : "keep";
```

This is equivalent to the following code fragment:

Code section 3.25: Equivalent code.

```
1  String answer;
2  if (p < 0.05) {
3      answer = "reject";
4  } else {
5      answer = "keep";
6  }
```

Switch

The **`switch`** conditional statement is basically a shorthand version of writing many **`if`**...**`else`** statements. The **`switch`** block evaluates a **`char`**, **`byte`**, **`short`**, or **`int`** (or **`enum`**, starting in J2SE 5.0; or `String`, starting in J2SE 7.0), and, based on the value provided, jumps to a specific **`case`** within the switch block and executes code until the **`break`** command is encountered or the end of the block. If the switch value does not match any of the case values, execution will jump to the optional **`default`** case.

The structure of a `switch` statement is as follows:

switch (***int$_1$ or char$_1$ or short$_1$ or byte$_1$ or enum$_1$ or String value$_1$***) {

case ***case value$_1$***:

statement$_{1.1}$
...
statement$_{1.n}$
break;

case ***case value$_2$***:

statement$_{2.1}$
...
statement$_{2.n}$
break;

default:

statement$_{n.1}$
...
statement$_{n.n}$

}

Here is an example to illustrate:

Code section 3.26: A switch block.

```
1  int i = 3;
2  switch(i) {
3      case 1:
4          // i doesn't equal 1, so this code won't execute
5          System.out.println("i equals 1");
6          break;
7      case 2:
8          // i doesn't equal 2, so this code won't execute
9          System.out.println("i equals 2");
10         break;
11     default:
12         // i has not been handled so far, so this code will execute
13         System.out.println("i equals something other than 1 or 2");
14 }
```

Output for code section 3.26

```
i equals something other than 1 or 2
```

If a case does not end with the `break` statement, then the next case will be checked, otherwise the execution will jump to the end of the `switch` statement.

Look at this example to see how it's done:

Code section 3.27: A switch block containing a case without break.

```
1  int i = -1;
2  switch(i) {
3      case -1:
4      case 1:
5          // i is -1, so it will fall through to this case and execute this
   code
6          System.out.println("i is 1 or -1");
7          break;
8      case 0:
9          // The break command is used before this case, so if i is 1 or -1,
   this will not execute
10         System.out.println("i is 0");
11 }
```

Output for code section 3.27

```
i is 1 or -1
```

Starting in J2SE 5.0, the `switch` statement can also be used with an **`enum`** value instead of an integer.

Though `enum`s have not been covered yet, here is an example so you can see how it's done (note that the enum constants in the cases do not need to be qualified with the type:

Code section 3.28: A switch block with an enum type.

```
1  Day day = Day.MONDAY; // Day is a fictional enum type containing the days
   of the week
2  switch(day) {
3      case MONDAY:
4          // Since day == Day.MONDAY, this statement will execute
5          System.out.println("Mondays are the worst!");
6          break;
7      case TUESDAY:
8      case WEDNESDAY:
9      case THURSDAY:
10         System.out.println("Weekdays are so-so.");
11         break;
12     case FRIDAY:
13     case SATURDAY:
14     case SUNDAY:
15         System.out.println("Weekends are the best!");
16         break;
17 }
```

Output for code section 3.28

```
Mondays are the worst!
```

Starting in J2SE 7.0, the `switch` statement can also be used with an `String` value instead of an integer.

Code section 3.29: A switch block with a String type.

```
1  String day = "Monday";
2  switch(day) {
3      case "Monday":
```

Output for code section 3.29

```
Mondays are the worst!
```

```
        // Since day == "Monday", this statement will execute
        System.out.println("Mondays are the worst!");
        break;
    case "Tuesday":
    case "Wednesday":
    case "Thursday":
        System.out.println("Weekdays are so-so.");
        break;
    case "Friday":
    case "Saturday":
    case "Sunday":
        System.out.println("Weekends are the best!");
        break;
    default:
        throw new IllegalArgumentException("Invalid day of the week: "
+ day);
}
```

Loop blocks

Loops are a handy tool that enables programmers to do repetitive tasks with minimal effort. Say we want a program that can count from 1 to 10, we could write the following program.

Code listing 3.4: Count.java

```
class Count {
    public static void main(String[] args) {
        System.out.println("1 ");
        System.out.println("2 ");
        System.out.println("3 ");
        System.out.println("4 ");
        System.out.println("5 ");
        System.out.println("6 ");
        System.out.println("7 ");
        System.out.println("8 ");
        System.out.println("9 ");
        System.out.println("10 ");
    }
}
```

Output for code listing 3.4

```
1
2
3
4
5
6
7
8
9
10
```

The task will be completed just fine, the numbers 1 to 10 will be printed in the output, but there are a few problems with this solution:

- **Flexibility**: what if we wanted to change the start number or end number? We would have to go through and change them, adding extra lines of code where they're needed.
- **Scalability**: 10 repeats are trivial, but what if we wanted 100 or even 1000 repeats? The number of lines of code needed would be overwhelming for a large number of iterations.
- **Maintenance**: where there is a large amount of code, one is more likely to make a mistake.
- **Feature**: the number of tasks is fixed and doesn't change at each execution.

Using loops we can solve all these problems. Once you get you head around them they will be invaluable to solving many problems in programming.

Open up your editing program and create a new file saved as `Loop.java`. Now type or copy the following code:

Code listing 3.5: Loop.java

```
class Loop {
    public static void main(String[] args) {
        int i;
        for (i = 1; i <= 10; i++) {
            System.out.println(i + " ");
        }
    }
}
```

Output for code listing 3.5

```
1
2
3
4
5
6
7
8
```

```
9
10
```

If we run the program, the same result is produced, but looking at the code, we immediately see the advantages of loops. Instead of executing ten different lines of code, line 5 executes ten times. Ten lines of code have been reduced to just four. Furthermore, we may change the number 10 to any number we like. Try it yourself, replace the 10 with your own number.

While

`while` loops are the simplest form of loop. The `while` loop repeats a block of code while the specified condition is true. Here is the structure of a `while` loop:

while (***boolean expression*$_1$**) {

***statement*$_1$**
***statement*$_2$**
...
***statement*$_n$**

}

The loop's condition is checked before each iteration of the loop. If the condition is false at the start of the loop, the loop will not be executed at all. The code section 3.28 sets in `squareHigherThan200` the smallest integer whose square exceeds 200.

Code section 3.28: The smallest integer whose square exceeds 200.

```
1  int squareHigherThan200 = 0;
2
3  while (squareHigherThan200 * squareHigherThan200 < 200) {
4    squareHigherThan200 = squareHigherThan200 + 1;
5  }
```

If a loop's condition will never become false, such as if the **`true`** constant is used for the condition, said loop is known as an *infinite loop*. Such a loop will repeat indefinitely unless it is broken out of. Infinite loops can be used to perform tasks that need to be repeated over and over again without a definite stopping point, such as updating a graphics display.

Do... while

The **`do-while`** loop is functionally similar to the `while` loop, except the condition is evaluated AFTER the statement executes It is useful when we try to find a data that does the job by randomly browsing an amount of data.

do {

***statement*$_1$**
***statement*$_2$**
...
***statement*$_n$**

} while (***boolean expression*$_1$**);

For

The **`for`** loop is a specialized `while` loop whose syntax is designed for easy iteration through a sequence of numbers. It consists of the keyword `for` followed by three extra statements enclosed in parentheses. The first statement is the variable declaration statement, which allows you to declare one or more integer variables. The second is the

condition, which is checked the same way as the `while` loop. Last is the iteration statement, which is used to increment or decrement variables, though any statement is allowed.

This is the structure of a `for` loop:

for (***variable declarations***; ***condition***; ***iteration statement***) {

***statement*$_1$**
***statement*$_2$**
...
***statement*$_n$**

}

To clarify how a for loop is used, here is an example:

Code section 3.29: A `for` loop.

```
1  for (int i = 1; i <= 10; i++) {
2      System.out.println(i);
3  }
```

Output for code listing 3.29

```
1
2
3
4
5
6
7
8
9
10
```

The `for` loop is like a template version of the `while` loop. The alternative code using a `while` loop would be as follows:

Code section 3.30: An iteration using a `while` loop.

```
1  int i = 1;
2  while (i <= 10) {
3    System.out.println(i);
4    i++;
5  }
```

The code section 3.31 shows how to iterate with the `for` loop using multiple variables and the code section 3.32 shows how any of the parameters of a `for` loop can be skipped. Skip them all, and you have an infinitely repeating loop.

Code section 3.31: The `for` loop using multiple variables.

```
1  for (int i = 1, j = 10; i <= 10; i++, j--) {
2    System.out.print(i + " ");
3    System.out.println(j);
4  }
```

Code section 3.32: The `for` loop without parameter.

```
1  for (;;) {
2    // Some code
3  }
```

For-each

Arrays haven't been covered yet, but you'll want to know how to use the enhanced for loop, called the `for-each` loop. The `for-each` loop automatically iterates through a list or array and assigns the value of each index to a variable.

To understand the structure of a `for-each` loop, look at the following example:

Code section 3.33: A `for-each` loop.

```
1  String[] sentence = {"I", "am", "a", "Java", "program."};
2  for (String word : sentence) {
3
```

Output for code section 3.33

```
I am a Java program.
```

```
        System.out.print(word + " ");
    }
```

The example iterates through an array of words and prints them out like a sentence. What the loop does is iterate through `sentence` and assign the value of each index to `word`, then execute the code block.

Here is the general contract of the `for-each` loop:

for (***variable declaration*** : ***array or list***) {

statement$_1$
statement$_2$
...
statement$_n$

}

Make sure that the type of the array or list is assignable to the declared variable, or you will get a compilation error. Notice that the loop automatically exits after the last item in the collection has been examined in the statement block.

Although the enhanced for loop can make code much clearer, it can't be used in some common situations.

- **Only access.** Elements can not be assigned to, eg, not to increment each element in a collection.
- **Only single structure.** It's not possible to traverse two structures at once, eg, to compare two arrays.
- **Only single element.** Use only for single element access, eg, not to compare successive elements.
- **Only forward.** It's possible to iterate only forward by single steps.
- **At least Java 5.** Don't use it if you need compatibility with versions before Java 5.

Break and continue keywords

The **`break`** keyword exits a flow control loop, such as a for loop. It basically *breaks* the loop.

In the code section 3.34, the loop would print out all the numbers from 1 to 10, but we have a check for when `i` equals 5. When the loop reaches its fifth iteration, it will be cut short by the `break` statement, at which point it will exit the loop.

Code section 3.34: An interrupted `for` loop.

```
for (int i = 1; i <= 10; i++) {
    System.out.println(i);
    if (i == 5) {
       System.out.println("STOP!");
       break;
    }
}
```

Output for code section 3.34

```
1
2
3
4
5
STOP!
```

The **`continue`** keyword jumps straight to the next iteration of a loop and evaluates the boolean expression controlling the loop. The code section 3.35 is an example of the `continue` statement in action:

Code section 3.35: A `for` loop with a skipped iteration.

```
for (int i = 1; i <= 10; i++) {
    if (i == 5) {
        System.out.println("Caught i == 5");
        continue;
    }
    System.out.println(i);
}
```

Output for code section 3.35

```
1
2
3
4
Caught i == 5
6
7
8
9
10
```

As the `break` and `continue` statements reduce the readability of the code, it is recommended to reduce their use or replace them with the use of `if` and `while` blocks. Some IDE refactoring operations will fail because of such statements.

Test your knowledge

Question 3.2: Consider the following code:

Question 3.2: Loops and conditions.

```
int numberOfItems = 5;
int currentItems = 0;
int currentCandidate = 1;

while (currentItems < numberOfItems) {
  currentCandidate = currentCandidate + 1;
  System.out.println("Test with integer: " + currentCandidate);

  boolean found = true;
  for (int i = currentCandidate - 1; i > 1; i--) {

    // Test if i is a divisor of currentCandidate
    if ((currentCandidate % i) == 0) {
      System.out.println("Not matching...");
      found = false;
      break;
    }

  }

  if (found) {
    System.out.println("Matching!");
    currentItems = currentItems + 1;
  }
}

System.out.println("Find the value: " + currentCandidate);
```

What will be printed in the standard output?

Answer

Output for Question 3.2

```
Test with integer: 2
Matching!
Test with integer: 3
Matching!
Test with integer: 4
Not matching...
Test with integer: 5
Matching!
Test with integer: 6
Not matching...
Test with integer: 7
Matching!
Test with integer: 8
Not matching...
Test with integer: 9
Not matching...
Test with integer: 10
Not matching...
Test with integer: 11
Matching!
Find the value: 11
```

The snippet is searching the 5^{th} prime number, that is to say: 11. It iterates on each positive integer from 2 (2, 3, 4, 5, 6, 7, 8, 9, 10, 11...), among them, it counts the prime numbers (2, 3, 5, 7, 11) and it stops at the 5^{th} one.

So the snippet first iterates on each positive integer from 2 using the `while` loop:

Answer 3.2.1: `while` loop.

```
int numberOfItems = 5;
int currentItems = 0;
int currentCandidate = 1;
```

```
4
5  while (currentItems < numberOfItems) {
6    currentCandidate = currentCandidate + 1;
7    System.out.println("Test with integer: " + currentCandidate);
8
9    boolean found = true;
10   for (int i = currentCandidate - 1; i > 1; i--) {
11
12     // Test if i is a divisor of currentCandidate
13     if ((currentCandidate % i) == 0) {
14       System.out.println("Not matching...");
15       found = false;
16       break;
17     }
18
19   }
20
21   if (found) {
22     System.out.println("Matching!");
23     currentItems = currentItems + 1;
24   }
25 }
26
27 System.out.println("Find the value: " + currentCandidate);
```

For each iteration, the current number is either a prime number or not. If it is a prime number, the code at the left will be executed. If it is not a prime number, the code at the right will be executed.

Answer 3.2.2: A prime number.

```
1  int numberOfItems = 5;
2  int currentItems = 0;
3  int currentCandidate = 1;
4
5  while (currentItems < numberOfItems) {
6    currentCandidate = currentCandidate + 1;
7    System.out.println("Test with integer: " +
   currentCandidate);
8
9    boolean found = true;
10   for (int i = currentCandidate - 1; i > 1; i-
   -) {
11
12     // Test if i is a divisor of
   currentCandidate
13     if ((currentCandidate % i) == 0) {
14       System.out.println("Not matching...");
15       found = false;
16       break;
17     }
18
19   }
20
21   if (found) {
22     System.out.println("Matching!");
23     currentItems = currentItems + 1;
24   }
25 }
26
27 System.out.println("Find the value: " +
   currentCandidate);
```

Answer 3.2.3: Not a prime number.

```
1  int numberOfItems = 5;
2  int currentItems = 0;
3  int currentCandidate = 1;
4
5  while (currentItems < numberOfItems) {
6    currentCandidate = currentCandidate + 1;
7    System.out.println("Test with integer: " +
   currentCandidate);
8
9    boolean found = true;
10   for (int i = currentCandidate - 1; i > 1; i--)
   {
11
12     // Test if i is a divisor of
   currentCandidate
13     if ((currentCandidate % i) == 0) {
14       System.out.println("Not matching...");
15       found = false;
16       break;
17     }
18
19   }
20
21   if (found) {
22     System.out.println("Matching!");
23     currentItems = currentItems + 1;
24   }
25 }
26
27 System.out.println("Find the value: " +
   currentCandidate);
```

The prime numbers are counted using `currentItems`. When `currentItems` is equal to `numberOfItems` (5), the program go out of the `while` loop. `currentCandidate` contains the last number, that is to say the 5^{th} prime number:

Answer 3.2.4: End of the program.

```
1  int numberOfItems = 5;
2  int currentItems = 0;
3  int currentCandidate = 1;
4
5  while (currentItems < numberOfItems) {
6    currentCandidate = currentCandidate + 1;
7    System.out.println("Test with integer: " + currentCandidate);
8
9    boolean found = true;
10   for (int i = currentCandidate - 1; i > 1; i--) {
11
12     // Test if i is a divisor of currentCandidate
13     if ((currentCandidate % i) == 0) {
```

```
      System.out.println("Not matching...");
      found = false;
      break;
    }

  }

  if (found) {
    System.out.println("Matching!");
    currentItems = currentItems + 1;
  }
}

System.out.println("Find the value: " + currentCandidate);
```

Labels

Labels can be used to give a name to a loop. The reason to do this is so we can break out of or continue with upper-level loops from a nested loop.

Here is how to label a loop:

label name*:*loop

To break out of or continue with a loop, use the `break` or `continue` keyword followed by the name of the loop.

For example:

Code section 3.36: A double `for` loop.

```
int i, j;
int[][] nums = {
    {1, 2, 5},
    {6, 9, 7},
    {8, 3, 4}
};

Outer:
for (i = 0; i < nums.length; i++) {
    for (j = 0; j < nums[i].length; j++) {
        if (nums[i][j] == 9) {
            System.out.println("Found number 9 at (" + i + ", " + j + ")");
            break Outer;
        }
    }
}
```

`Output for code section 3.36`

```
Found number 9 at (1, 1)
```

You needn't worry if you don't understand all the code, but look at how the label is used to break out of the outer loop from the inner loop. However, as such a code is hard to read and maintain, it is highly recommended not to use labels.

Try... catch blocks

See also Throwing and Catching Exceptions.

The **`try-catch`** blocks are used to catch any exceptions or other throwable objects within the code.

Here's what `try-catch` blocks looks like:

try {

***statement*$_{1.1}$**
***statement*$_{1.2}$**
...
***statement*$_{1.n}$**

} catch (***exception*$_1$**) {

$statement_{2.1}$
...
$statement_{2.n}$

}

The code listing 3.6 tries to print all the arguments that have been passed to the program. However, if there not enough arguments, it will throw an exception.

Code listing 3.6: Attempt.java

```
1  public class Attempt {
2    public static void main(String[] args) {
3      try {
4        System.out.println(args[0]);
5        System.out.println(args[1]);
6        System.out.println(args[2]);
7        System.out.println(args[3]);
8      } catch (ArrayIndexOutOfBoundsException e) {
9        System.out.println("Not enough arguments");
10     }
11   }
12 }
```

In addition to the try and catch blocks, a **`finally`** block may be present. The finally block is always executed, even if an exception is thrown. It may appear with or without a catch block, but always with a try block.

Here is what a finally block looks like:

try {

$statement_{1.1}$
$statement_{1.2}$
...
$statement_{1.n}$

} catch ($exception_1$) {

$statement_{2.1}$
...
$statement_{2.n}$

} finally {

$statement_{3.1}$
...
$statement_{3.n}$

}

Examples

The code listing 3.7 receives a number as parameter and print its binary representation.

Code listing 3.7: GetBinary.java

```
1  public class GetBinary {
2      public static void main(String[] args) {
3          if (args.length == 0) {
4              // Print usage
5              System.out.println("Usage: java GetBinary <decimal integer>");
6              System.exit(0);
7          } else {
8              // Print arguments
9              System.out.println("Received " + args.length + " arguments.");
10             System.out.println("The arguments are:");
```

```
            for (String arg : args) {
                System.out.println("\t" + arg);
            }
        }

        int number = 0;
        String binary = "";

        // Get the input number
        try {
            number = Integer.parseInt(args[0]);
        } catch (NumberFormatException ex) {
            System.out.println("Error: argument must be a base-10 integer.");
            System.exit(0);
        }

        // Convert to a binary string
        do {
            switch (number % 2) {
                case 0: binary = '0' + binary; break;
                case 1: binary = '1' + binary; break;
            }
            number >>= 1;
        } while (number > 0);

        System.out.println("The binary representation of " + args[0] + " is " + binary);
    }
}
```

The code listing 3.8 is a simulation of playing a game called Lucky Sevens. It is a dice game where the player rolls two dice. If the numbers on the dice add up to seven, he wins $4. If they do not, he loses $1. The game shows how to use control flow in a program as well as the fruitlessness of gambling.

Code listing 3.8: LuckySevens.java

```
import java.util.*;

public class LuckySevens {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        Random random = new Random();
        String input;
        int startingCash, cash, maxCash, rolls, roll;

        // Loop until "quit" is input
        while (true) {
            System.out.print("Enter the amount of cash to start with (or \"quit\" to quit): ");

            input = in.nextLine();

            // Check if user wants to exit
            if (input.toLowerCase().equals("quit")) {
                System.out.println("\tGoodbye.");
                System.exit(0);
            }

            // Get number
            try {
                startingCash = Integer.parseInt(input);
            } catch (NumberFormatException ex) {
                System.out.println("\tPlease enter a positive integer greater than 0.");
                continue;
            }

            // You have to start with some money!
            if (startingCash <= 0) {
                System.out.println("\tPlease enter a positive integer greater than 0.");
                continue;
            }

            cash = startingCash;
            maxCash = cash;
            rolls = 0;
            roll = 0;

            // Here is the game loop
            for (; cash > 0; rolls++) {
               roll = random.nextInt(6) + 1;
               roll += random.nextInt(6) + 1;

                if (roll == 7)
                    cash += 4;
                else
                    cash -= 1;

                if (cash > maxCash)
                    maxCash = cash;
```

```
            }

            System.out.println("\tYou start with $" + startingCash + ".\n"
                    + "\tYou peak at $" + maxCash + ".\n"
      + "\tAfter " + rolls + " rolls, you run out of cash.");
        }
    }
}
```

Boolean expressions

Boolean values are values that evaluate to either **`true`** or **`false`**, and are represented by the **`boolean`** data type. Boolean expressions are very similar to mathematical expressions, but instead of using mathematical operators such as "+" or "-", you use comparative or boolean operators such as "==" or "!".

Comparative operators

Java has several operators that can be used to compare variables. For example, how would you tell if one variable has a greater value than another? The answer: use the "greater-than" operator.

Here is a list of the comparative operators in Java:

- > : Greater than
- < : Less than
- >= : Greater than or equal to
- <= : Less than or equal to
- == : Equal to
- != : Not equal to

To see how these operators are used, look at this example:

Code section 3.37: Comparisons.

```
int a = 5, b = 3;
System.out.println(a > b); // Value is true because a is greater than b
System.out.println(a == b); // Value is false because a does not equal b
System.out.println(a != b); // Value is true because a does not equal b
System.out.println(b <= a); // Value is true because b is less than a
```

Output for code section 3.37

```
true
false
true
true
```

Comparative operators can be used on any primitive types (except `boolean`), but only the "equals" and "does not equal" operators work on objects. This is because the less-than/greater-than operators cannot be applied to objects, but the equivalency operators can.

Specifically, the == and != operators test whether both variables point to the same object. Objects will be covered later in the tutorial, in the "Classes, Objects, and Types" module.

Boolean operators

The Java boolean operators are based on the operations of the boolean algebra. The boolean operators operate directly on boolean values.

Here is a list of four common boolean operators in Java:

- ! : Boolean NOT

- && : Boolean AND
- || : Boolean inclusive OR
- ^ : Boolean exclusive XOR

The boolean NOT operator ("!") inverts the value of a boolean expression. The boolean AND operator ("&&") will result in true if and only if the values on both sides of the operator are true. The boolean inclusive OR operator ("||") will result in true if either or both of the values on the sides of the operator is true. The boolean exclusive XOR operator ("^") will result in true if one and only one of the values on the sides of the operator is true.

To show how these operators are used, here is an example:

Code section 3.38: Operands.

```
1  boolean iMTrue = true;
2  boolean iMTrueToo = true;
3  boolean iMFalse = false;
4  boolean iMFalseToo = false;
5
6  System.out.println("NOT operand:");
7  System.out.println(!iMTrue);
8  System.out.println(!iMFalse);
9  System.out.println(!(4 < 5));
10 System.out.println("AND operand:");
11 System.out.println(iMTrue && iMTrueToo);
12 System.out.println(iMFalse && iMFalseToo);
13 System.out.println(iMTrue && iMFalse);
14 System.out.println(iMTrue && !iMFalse);
15 System.out.println("OR operand:");
16 System.out.println(iMTrue || iMTrueToo);
17 System.out.println(iMFalse || iMFalseToo);
18 System.out.println(iMTrue || iMFalse);
19 System.out.println(iMFalse || !iMTrue);
20 System.out.println("XOR operand:");
21 System.out.println(iMTrue ^ iMTrueToo);
22 System.out.println(iMFalse ^ iMFalseToo);
23 System.out.println(iMTrue ^ iMFalse);
24 System.out.println(iMFalse ^ !iMTrue);
```

Output for code section 3.38

```
NOT operand:
false
true
false
AND operand:
true
false
false
true
OR operand:
true
false
true
false
XOR operand:
false
false
true
false
```

Here are the truth tables for the boolean operators:

a	!a
true	false
false	true

a	b	a && b	a || b	a ^ b
true	true	true	true	false
true	false	false	true	true
false	true	false	true	true
false	false	false	false	false

For help on simplifying complex logic, see De Morgan's laws.

In Java, boolean logic has a useful property called *short circuiting*. This means that expressions will only be evaluated as far as necessary. In the expression `(a && b)`, if a is false, then b will not be evaluated because the expression will be false no matter what. Here is an example that shows that the second expression is not automatically checked:

Code section 3.39: Short circuiting.

```
1  System.out.println((4 < 5) || ((10 / 0) == 2));
```

Output for code section 3.39

```
true
```

To disable this property, you can use & instead of && and | instead of || but it's not recommended.

For the bitwise operations on & and |, see Arithmetic expressions.

Variables

In the Java programming language, the words **field** and **variable** are both one and the same thing. Variables are devices that are used to store data, such as a number, or a string of character data.

Variables in Java programming

Java is considered as a **strongly typed** programming language. Thus all variables in the Java programming language ought to have a particular **data type**. This is either declared or inferred and the Java language only allows programs to run if they adhere to type constraints.

If you present a numeric type with data that is not numeric, say textual content, then such declarations would violate Java's type system. This gives Java the ability of **type safety**. Java checks if an expression or data is encountered with an incorrect type or none at all. It then automatically flags this occurrence as an error at compile time. Most type-related errors are caught by the Java compiler, hence making a program more secure and safe once compiled completely and successfully. Some languages (such as C) define an interpretation of such a statement and use that interpretation without any warning; others (such as PL/I) define a conversion for almost all such statements and perform the conversion to complete the assignment. Some type errors can still occur at runtime because Java supports a cast operation which is a way of changing the type of one expression to another. However, Java performs run time type checking when doing such casts, so an incorrect type cast will cause a runtime exception rather than succeeding silently and allowing data corruption.

On the other hand, Java is also known as a **hybrid language**. While supporting object oriented programming (OOP), Java is not a pure OO language like Smalltalk or Ruby. Instead, Java offers both object types and primitive types. Primitive types are used for boolean, character, and numeric values and operations. This allows relatively good performance when manipulating numeric data, at the expense of flexibility. For example, you cannot subclass the primitive types and add new operations to them.

Kinds of variables

In the Java programming language, there are four kinds of variables.

Code listing 3.9: ClassWithVariables.java

```
1  public class ClassWithVariables {
2      public int id = 0;
3      public static boolean isClassUsed;
4
5      public void processData(String parameter) {
6        Object currentValue = null;
7      }
8  }
```

In the code listing 3.9, are examples of all four kinds of variables.

- **Instance variables**: These are variables that are used to store the state of an object (for example, `id`). Every object created from a class definition would have its own copy of the variable. It is valid for and occupies storage for as long as the corresponding object is in memory.
- **Class variables**: These variables are explicitly defined within the class-level scope with a **`static`** modifier (for example, `isClassUsed`). No other variables can have a `static` modifier attached to them. Because these variables are defined with the `static` modifier, there would always be a single copy of these variables no matter how many times the class has been instantiated. They live as long as the class is loaded in memory.
- **Parameters** or **Arguments**: These are variables passed into a method signature (for example, `parameter`). Recall the usage of the `args` variable in the main method. They are not attached to modifiers (i.e. `public`, `private`, `protected` or `static`) and they can be used everywhere in the method. They are in memory during the execution of the method and can't be used after the method returns.

- **Local variables**: These variables are defined and used specifically within the method-level scope (for example, `currentValue`) but not in the method signature. They do not have any modifiers attached to it. They no longer exist after the method has returned.

Test your knowledge

Question 3.5: Consider the following code:

Question 3.5: SomeClass.java

```
1  public class SomeClass {
2    public static int c = 1;
3    public int a = c;
4    private int b;
5
6    public void someMethod(int d) {
7      d = c;
8      int e;
9    }
10 }
```

In the example above, we created five variables: `a`, `b`, `c`, `d` and `e`. All these variables have the same data type `int` (integer). However, can you tell what kind of variable each one is?

Answer

- a and b are **instance variables**;
- c is a **class variable**;
- d is a **parameter** or **argument**; and,
- e is a **local variable**.

Creating variables

Variables and all the information they store are kept in the computer's memory for access. Think of a computer's memory as a table of data — where each cell corresponds to a variable.

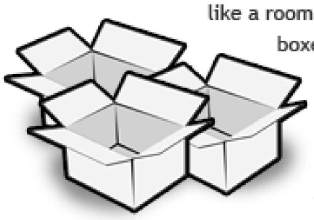
Upon creating a variable, we basically create a new address space and give it a unique name. Java goes one step further and lets you define what you can place within the variable — in Java parlance you call this a *data type*. So, you essentially have to do two things in order to create a variable:

- Create a variable by giving it a unique name; and,
- Define a data type for the variable.

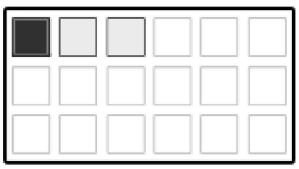
The following code demonstrates how a simple variable can be created. This process is known as *variable declaration*.

Code section 3.40: A simple variable declaration.

```
1 int a;
```



A computer's memory is much like a room to stuff in stacks of boxes to be filled with things. But, a more direct and linear way of looking at memory would be to envision it as a table of inter-connected cells. These cells can be either allocated to be filled at a later time, or might contain a value within. Rest of the memory stays unused.



A graphical representation of computer memory

Assigning values to variables

Because we have provided a data type for the variable, we have a hint as to what the variable can and cannot hold. We know that `int` (integer) data type supports numbers that are either positive or negative integers. Therefore once a variable is created, we can provide it with any integer value using the following syntax. This process is called an *assignment operation*.

Code section 3.41: Variable declaration and assignment operation (on different lines).

```
int a;
a = 10;
```

Java provides programmers with a simpler way of combining both variable declaration and assignment operation in one line. Consider the following code:

Code section 3.42: Variable declaration and assignment operation (on the same line).

```
int a = 10;
```

Grouping variable declarations and assignment operations

Consider the following code:

Code section 3.43: Ungrouped declarations.

```
int a;
int b;
String c;
a = 10;
b = 20;
c = "some text";
```

There are various ways by which you can streamline the writing of this code. You can group the declarations of similar data types in one statement, for instance:

Code section 3.44: Grouped declarations.

```
int a, b;
String c;
a = 10;
b = 20;
c = "some text";
```

Alternatively, you can further reduce the syntax by doing group declarations and assignments together, as such:

Code section 3.45: Grouped declarations and assignments.

```
int a = 10, b = 20;
String c = "some text";
```

Identifiers

Although memory spaces have their own addresses — usually a hash number such as `0xCAD3`, etc. — it is much easier to remember a variable's location in the memory if we can give it a recognizable name. **Identifiers** are the names we give to our variables. You can name your variable anything like `aVariable`, `someVariable`, `age`, `someonesImportantData`, etcetera. But notice: none of the names we described here has a space within it. Hence, it is pretty obvious that spaces aren't allowed in variable names. In fact, there are a lot of other things that are not allowed in variable names. The things that *are* allowed are:

- Characters `A` to `Z` and their lower-case counterparts `a` to `z`.
- Numbers `0` to `9`. However, numbers should not come at the beginning of a variable's name.
- And finally, special characters that include only `$` (dollar sign) and `_` (underscore).

Test your knowledge

Question 3.6: Which of the ones below are proper variable identifiers?

1. `f_name`

2. `lastname`
3. `someones name`
4. `$SomeoneElsesName`
5. `7days`
6. `TheAnswerIs42`

Answer

I can tell you that 3 and 5 are not the right way to do things around here, the rest are proper identifiers.

Any valid variable names might be correct but they are not always what you should be naming your variables for a few reasons as listed below:

- The name of the variable should reflect the value within them.
- The identifier should be named following the naming guidelines or conventions for doing so. We will explain that in a bit.
- The identifier shouldn't be a nonsense name like `lname`, you should always name it properly: `lastName` is the best way of naming a variable.

Naming conventions for identifiers

When naming identifiers, you need to use the following guidelines which ensure that your variables are named accurately. As we discussed earlier, we should always name our variables in a way that tells us what they hold. Consider this example:

Code section 3.46: Unknown process.

```
1  int a = 24;
2  int b = 365;
3  int c = a * b;
```

Do you know what this program does? Well, it multiplies two values. That much you guessed right. But, do you know what those values are? Exactly, you don't. Now consider this code:

Code section 3.47: Time conversion.

```
1  int age = 24;
2  int daysInYear = 365;
3  int ageInDays = age * daysInYear;
```

Now you can tell what's happening, can't you? However, before we continue, notice the *case* of the variables. If a word contains CAPITAL LETTERS, it is in **UPPER CASE**. If a word has small letters, it is in **lower case**. Both cases in a word renders it as **mIxEd CaSe**.

The variables we studied so far had a mixed case. When there are two or more words making up the names of a variable, you need to use a special case called the *camel-case*. Just like the humps of a camel, your words need to stand out. Using this technique, the words `first` and `name` could be written as either `firstName` or `FirstName`.

The first instance, `firstName` is what we use as the names of variables. Remember though, `firstName` is not the same as `FirstName` because Java is **case-sensitive**. Case-sensitive basically implies that the case in which you wrote one word is the case you have to call that word in when using them later on. Anything other than that is not the same as you intended. You'll know more as you progress. You can hopefully tell now why the variables you were asked to identify weren't proper.

Literals (values)

Now that we know how variables should be named, let us look at the values of those variables. Simple values like numbers are called *literals*. This section shows you what literals are and how to use them. Consider the following code:

Code section 3.48: Literals.

```
int age = 24;
long bankBalance = 20000005L;
```

By now, we've only seen how numbers work in assignment statements. Let's look at data types other than numbers. Characters are basically letters of the English alphabet. When writing a single character, we use single quotes to encapsulate them. Take a look at the code below:

Code section 3.49: Character.

```
char c = 'a';
```

Why, you ask? Well, the explanation is simple. If written without quotes, the system would think it's a variable identifier. That's the very distinction you have to make when differentiating between variables and their literal values. Character data types are a bit unusual. First, they can only hold a single character. What if you had to store a complete name within them, say *John*, would you write something like:

Code section 3.50: Character list.

```
char firstChar = 'J';
char secondChar = 'o';
char thirdChar = 'h';
char fourthChar = 'n';
```

Now, that's pathetic. Thankfully, there's a data type that handles large number of characters, it's called a `String`. A string can be initialized as follows:

Code section 3.51: String.

```
String name = "John";
```

Notice, the use of double quotation marks instead of single quotation marks. That's the only thing you need to worry about.

Primitive Types

Primitive types are the most basic data types available within the Java language. There are 8: **`boolean`**, **`byte`**, **`char`**, **`short`**, **`int`**, **`long`**, **`float`** and **`double`**. These types serve as the building blocks of data manipulation in Java. Such types serve only one purpose — containing pure, simple values of a kind. Because these data types are defined into the Java type system by default, they come with a number of operations predefined. You can not define a new operation for such primitive types. In the Java type system, there are three further categories of primitives:

- Numeric primitives: **`short`**, **`int`**, **`long`**, **`float`** and **`double`**. These primitive data types hold only numeric data. Operations associated with such data types are those of simple arithmetic (addition, subtraction, etc.) or of comparisons (is greater than, is equal to, etc.)
- Textual primitives: **`byte`** and **`char`**. These primitive data types hold characters (that can be Unicode alphabets or even numbers). Operations associated with such types are those of textual manipulation (comparing two words, joining characters to make words, etc.). However, **`byte`** and **`char`** can also support arithmetic operations.
- Boolean and null primitives: **`boolean`** and **`null`**.

All the primitive types have a fixed size. Thus, the primitive types are limited to a range of values. A smaller primitive type (**`byte`**) can contain less values than a bigger one (**`long`**).

Category	Types	Size (bits)	Minimum Value	Maximum Value	Precision	Example
Integer	`byte`	8	-128	127	From +127 to -128	`byte b = 65;`
	`char`	16	0	2^{16}-1	All Unicode characters[1]	`char c = 'A';` `char c = 65;`
	`short`	16	-2^{15}	2^{15}-1	From +32,767 to -32,768	`short s = 65;`
	`int`	32	-2^{31}	2^{31}-1	From +2,147,483,647 to -2,147,483,648	`int i = 65;`
	`long`	64	-2^{63}	2^{63}-1	From +9,223,372,036,854,775,807 to -9,223,372,036,854,775,808	`long l = 65L;`
Floating-point	`float`	32	2^{-149}	$(2-2^{-23})\cdot 2^{127}$	From 3.402,823,5 E+38 to 1.4 E-45	`float f = 65f;`
	`double`	64	2^{-1074}	$(2-2^{-52})\cdot 2^{1023}$	From 1.797,693,134,862,315,7 E+308 to 4.9 E-324	`double d = 65.55;`
Other	`boolean`	--	--	--	false, true	`boolean b = true;`
	`void`	--	--	--	--	--

Integer primitive types silently overflow:

Code section 3.52: Several operators.

```
1  int i = Integer.MAX_VALUE;
2  System.out.println(i);
3  i = i + 1;
4  System.out.println(i);
5  System.out.println(Integer.MIN_VALUE);
```

Console for Code section 3.52

```
2147483647
-2147483648
-2147483648
```

As Java is strongly typed, you can't assign a floating point number (a number with a decimal point) to an integer variable:

Code section 3.53: Setting a floating point number as a value to an `int` (integer) type.

```
1  int age;
2  age = 10.5;
```

A primitive type should be set by an appropriate value. The primitive types can be initialized with a literal. Most of the literals are primitive type values, except String Literals, which are instance of the `String` class.

Numbers in computer science

Programming may not be as trivial or boring as just crunching huge numbers any more. However, huge chunks of code written in any programming language today, let alone Java, obsessively deal with numbers, be it churning out huge prime numbers,[2] or just calculating a cost of emission from your scooter. In 1965, Gemini V space mission escaped a near-fatal accident caused by a programming error.[3] Again in 1979, a computer program overestimated the ability of five nuclear reactors to withstand earthquakes; the plants shut down temporarily.[4] There is one thing common to both these programming errors: the subject data, being computed at the time the errors occurred, was numeric. Out of past experience, Java came bundled with revised type checking for numeric data and put significant emphasis on correctly identifying different types of it. You must recognise the importance of numeric data when it comes to programming.

Numbers are stored in memory using a binary system. The memory is like a grid of cells:

Each cell can contain a *binary digit* (shortened to *bit*), that is to say, zero or one:

0	1	1	0	0	1	0	1

Actually, each cell **does** contain a binary digit, as one bit is roughly equivalent to `1` and an empty cell in the memory signifies `0`. A single binary digit can only hold two possible values: a zero or a one.

Memory state								*Gives*
							`0`	→ 0
							`1`	→ 1

Multiple bits held together can hold multiple permutations — 2 bits can hold 4 possible values, 3 can hold 8, and so on. For instance, the maximum number 8 bits can hold (`11111111` in binary) is `255` in the decimal system. So, the numbers from 0 to 255 can fit within 8 bits.

Memory state								*Gives*
`0`	`0`	`0`	`0`	`0`	`0`	`0`	`0`	→ 0
`0`	`0`	`0`	`0`	`0`	`0`	`0`	`1`	→ 1
`0`	`0`	`0`	`0`	`0`	`0`	`1`	`0`	→ 2
`0`	`0`	`0`	`0`	`0`	`0`	`1`	`1`	→ 3
...								...
`1`	`1`	`1`	`1`	`1`	`1`	`1`	`1`	→ 255

It is all good, but this way, we can only host positive numbers (or unsigned integers). They are called *unsigned integers*. Unsigned integers are whole number values that are all positive and do not attribute to negative values. For this very reason, we would ask one of the 8 bits to hold information about the sign of the number (positive or negative). This leaves us with just 7 bits to actually count out a number. The maximum number that these 7 bits can hold (`1111111`) is `127` in the decimal system.

Positive numbers

Memory state								*Gives*
`0`	`0`	`0`	`0`	`0`	`0`	`0`	`0`	→ 0
`0`	`0`	`0`	`0`	`0`	`0`	`0`	`1`	→ 1
`0`	`0`	`0`	`0`	`0`	`0`	`1`	`0`	→ 2
`0`	`0`	`0`	`0`	`0`	`0`	`1`	`1`	→ 3
...	...							...
`0`	`1`	`1`	`1`	`1`	`1`	`1`	`1`	→ 127

Negative numbers

Memory state								*Gives*
`1`	`0`	`0`	`0`	`0`	`0`	`0`	`0`	→ -128
`1`	`0`	`0`	`0`	`0`	`0`	`0`	`1`	→ -127
`1`	`0`	`0`	`0`	`0`	`0`	`1`	`0`	→ -126
`1`	`0`	`0`	`0`	`0`	`0`	`1`	`1`	→ -125
...	...							...
`1`	`1`	`1`	`1`	`1`	`1`	`1`	`1`	→ -1

Altogether, using this method, 8 bits can hold numbers ranging from `-128` to `127` (including zero) — a total of 256 numbers. Not a bad pay-off one might presume. The opposite to an unsigned integer is a *signed integer* that have the capability of holding both positive and negative values.

But, what about larger numbers. You would need significantly more bits to hold larger numbers. That's where Java's numeric types come into play. Java has multiple numeric types — their size dependent on the number of bits that are at play.

In Java, numbers are dealt with using data types specially formulated to host numeric data. But before we dive into these types, we must first set some concepts in stone. Just like you did in high school (or even primary school), numbers in Java are placed in clearly distinct groups and systems. As you'd already know by now, number systems includes groups like the *integer* numbers (0, 1, 2 ... ∞); *negative integers* (0, -1, -2 ... -∞) or even *real* and *rational* numbers (value of Pi, ¾, 0.333~, etcetera). Java simply tends to place these numbers in two distinct groups, **integers** (-∞ ... 0 ... ∞) and **floating point** numbers (any number with decimal points or fractional representation). For the moment, we would only look into integer values as they are easier to understand and work with.

Integer types in Java

With what we have learned so far, we will identify the different types of signed integer values that can be created and manipulated in Java. Following is a table of the most basic numeric types: integers. As we have discussed earlier, the data types in Java for integers caters to both positive and negative values and hence are **signed numeric types**. The size in bits for a numeric type determines what its minimum and maximum value would be. If in doubt, one can always calculate these values.

Lets see how this new found knowledge of the basic integer types in Java fits into the picture. Say, you want to numerically manipulate the days in a year — all 365 days. What type would you use? Since the data type `byte` only goes up to 127, would you risk giving it a value greater than its allowed maximum. Such decisions might save you from dreaded errors that might occur out of the programmed code. A much more sensible choice for such a numeric operation might be a `short`. Now, why couldn't they make just one data type to hold all kinds of numbers? Let's explore why.

When you tell a program you need to use an integer, say even a `byte`, the Java program allocates a space in the memory. It allocates whole 8 bits of memory. Where it wouldn't seem to matter for today's memory modules that have place for almost a dozen trillion such bits, it matters in other cases. Once allocated that part of the memory gets used and can only be claimed back after the operation is finished. Consider a complicated Java program where the only data type you'd be using would be `long` integers. What happens when there's no space for more memory allocation jobs? Ever heard of the Stack Overflow errors. That's exactly what happens — your memory gets completely used up and fast. So, choose your data types with extreme caution.

Enough talk, let's see how you can create a numeric type. A numeric type begins with the type's name (`short`, `int`, etc.) and then provides with a name for the allocated space in the memory. Following is how it's done. Say, we need to create a variable to hold the number of days in a year.

Code section 3.54: Days in a year.

```
1  short daysInYear = 365;
```

Here, `daysInYear` is the name of the variable that holds `365` as its value, while `short` is the data type for that particular value. Other uses of integer data types in Java might see you write code such as this given below:

Code section 3.55: Integer data types in Java.

```
1  byte maxByte = 127;
2  short maxShort = 32767;
3  int maxInt = 2147483647;
4  long maxLong = 9223372036854775807L;
```

Integer numbers and floating point numbers

The data types that one can use for integer numbers are `byte`, `short`, `int` and `long` but when it comes to floating point numbers, we use `float` or `double`. Now that we know that, we can modify the code in the code section 3.53 as:

Code section 3.56: Correct floating point declaration and assignment.

```
1  double age = 10.5;
```

Why not `float`, you say? If we'd used a `float`, we would have to append the number with a `f` as a suffix, so `10.5` should be `10.5f` as in:

Code section 3.57: The correct way to define floating point numbers of type `float`.

```
1  float age = 10.5f;
```

Floating-point math never throws exceptions. Dividing a non-zero value by `0` equals `infinity`. Dividing a non-infinite value by `infinity` equals `0`.

Test your knowledge

Question 3.7: Consider the following code:

Question 3.7: Primitive type assignments.

```
...

a = false;
b = 3.2;
c = 35;
d = -93485L;
e = 'q';
```

These are five variables. There are a `long`, a `byte`, a `char`, a `double` and a `boolean`. Retrieve the type of each one.

Answer

Answer 3.7: Primitive type assignments and declarations.

```
boolean a;
double b;
byte c;
long d;
char e;

a = false;
b = 3.2;
c = 35;
d = -93485L;
e = 'q';
```

- a can only be the `boolean` because only a boolean can handle boolean values.
- e can only be the `char` because only a char can contain a character.
- b can only be the `double` because only a double can contain a decimal number here.
- d is the `long` because a `byte` can not contain such a low value.
- c is the remaining one so it is the `byte`.

Data conversion (casting)

Data conversion (casting) can happen between two primitive types. There are two kinds of casting:

- Implicit: casting operation is not required; the magnitude of the numeric value is always preserved. However, *precision* may be lost when converting from integer to floating point types
- Explicit: casting operation required; the magnitude of the numeric value may not be preserved

Code section 3.58: Implicit casting (int is converted to long, casting **is not needed).**

```
int  i = 65;
long l = i;
```

Code section 3.59: Explicit casting (long is converted to int, casting **is needed).**

```
long l = 656666L;
int  i = (int) l;
```

The following table shows the conversions between primitive types, it shows the casting operation for explicit conversions:

	from byte	from char	from short	from int	from long	from float	from double	from boolean
to byte	-	(byte)	(byte)	(byte)	(byte)	(byte)	(byte)	N/A
to char		-	(char)	(char)	(char)	(char)	(char)	N/A
to short		(short)	-	(short)	(short)	(short)	(short)	N/A
to int				-	(int)	(int)	(int)	N/A
to long					-	(long)	(long)	N/A
to float						-	(float)	N/A
to double							-	N/A
to boolean	N/A	N/A	N/A	N/A	N/A	N/A	N/A	-

Unlike C, C++ and similar languages, Java can't represent `false` as `0` or `null` and can't represent `true` as non-zero. Java can't cast from boolean to a non-boolean primitive data type, or vice versa.

For non primitive types:

	to Integer	to Float	to Double	to String	to Array
Integer	-	(float)x	(double)x x.doubleValue()	x.toString() Float.toString(x)	new int[] {x}
Float	java.text.DecimalFormat("#").format(x)	-	(double)x	x.toString()	new float[] {x}
Double	java.text.DecimalFormat("#").format(x)	java.text.DecimalFormat("#").format(x)	-	x.toString()	new double[] {x}
String	Integer.parseInt(x)	Float.parseFloat(x)	Double.parseDouble(x)	-	new String[] {x}
Array	x[0]	x[0]	x[0]	Arrays.toString(x)	-

Notes

1. According to "STR01-J. Do not assume that a Java char fully represents a Unicode code point". Carnegie Mellon University - Software Engineering Institute. https://wiki.sei.cmu.edu/confluence/display/java/STR01-J.+Do+not+assume+that+a+Java+char+fully+represents+a+Unicode+code+point. Retrieved 27 Nov 2018., not all Unicode characters fit into a 16 bit representation
2. As of edit (11 December 2013), the Great Internet Mersenne Prime Search project has so far identified the largest prime number as being 17,425,170 digits long. Prime numbers are valuable to cryptologists as the bigger the number, the securer they can make their data encryption logic using that particular number.
3. Gemini 5 landed 130 kilometers short of its planned Pacific Ocean landing point due to a software error. The Earth's rotation rate had been programmed as one revolution per solar day instead of the correct value, one revolution per sidereal day.
4. A program used in their design used an arithmetic sum of variables when it should have used the sum of their absolute values. (Evars Witt, "*The Little Computer and the Big Problem*", AP Newswire, 16 March 1979. See also Peter Neumann, "*An Editorial on Software Correctness and the Social Process*" Software Engineering Notes, Volume 4(2), April 1979, page 3)

Arithmetic expressions

In order to do arithmetic in Java, one must first declare at least one variable. Typically one declares a variable and assigns it a value before any arithmetic is done. Here's an example of declaring an integer variable:

Code section 3.59: Variable assignation.

```
1  int x = 5;
```

After creating a variable, one can manipulate its value by using Java's operators: + (addition), - (subtraction), * (multiplication), / (integer division), % (modulo or remainder), ++ (pre- & postincrement by one), -- (pre- & postdecrement by one).

Code listing 3.10: Operators.java

```
1  public class Operators {
2    public static void main(String[] args) {
3      int x = 5;
4      System.out.println("x = " + x);
5      System.out.println();
6
7      System.out.println("--- Addition             ---");
8      x = 5;
9      System.out.println("x + 2 = " + (x + 2));
10     System.out.println("x = " + x);
11     System.out.println();
12
13     System.out.println("--- Subtraction          ---");
14     x = 5;
15     System.out.println("x - 4 = " + (x - 4));
16     System.out.println("x = " + x);
17     System.out.println();
18
19     System.out.println("--- Multiplication       ---");
20     x = 5;
21     System.out.println("x * 3 = " + (x * 3));
22     System.out.println("x = " + x);
23     System.out.println();
24
25     System.out.println("--- (Integer) Division   ---");
26     x = 5;
27     System.out.println("x / 2 = " + (x / 2));
28     System.out.println("x = " + x);
29     System.out.println();
30
31     System.out.println("--- Modulo (Remainder)   ---");
32     x = 5;
33     System.out.println("x % 2 = " + (x % 2));
34     System.out.println("x = " + x);
35     System.out.println();
36
37     System.out.println("--- Preincrement by one  ---");
38     x = 5;
39     System.out.println("++x   = " + (++x  ));
40     System.out.println("x = " + x);
41     System.out.println();
42
43     System.out.println("--- Predecrement by one  ---");
44     x = 5;
45     System.out.println("--x   = " + (--x  ));
46     System.out.println("x = " + x);
47     System.out.println();
48
49     System.out.println("--- Postincrement by one ---");
50     x = 5;
51     System.out.println("x++   = " + (x++  ));
52     System.out.println("x = " + x);
53     System.out.println();
54
55     System.out.println("--- Postdecrement by one ---");
56     x = 5;
57     System.out.println("x--   = " + (x--  ));
58     System.out.println("x = " + x);
59     System.out.println();
60   }
61 }
```

Console for Code listing 3.10

```
x = 5

--- Addition             ---
x + 2 = 7
x = 5

--- Subtraction          ---
x - 4 = 1
x = 5

--- Multiplication       ---
x * 3 = 15
x = 5

--- (Integer) Division   ---
x / 2 = 2
x = 5

--- Modulo (Remainder)   ---
x % 2 = 1
x = 5

--- Preincrement by one  ---
++x   = 6
x = 6

--- Predecrement by one  ---
--x   = 4
x = 4

--- Postincrement by one ---
x++   = 5
x = 6

--- Postdecrement by one ---
x--   = 5
x = 4
```

The division operator rounds towards zero: `5/2` is 2, and `-5/2` is -2. The remainder operator has the same sign as the left operand; it is defined such that `((a/b)*b) + (a%b)` is always equal to a. The preincrement, predecrement, postincrement, and postdecrement operators are special: they also change the value of the variable, by adding or subtracting one. The only difference is that preincrement/decrement returns the new value of the variable; postincrement returns the original value of the variable.

Test your knowledge

Question 3.8: Consider the following code:

Question 3.8: Question8.java

```
1  public class Question8 {
2    public static void main(String[] args) {
3      int x = 10;
4      x = x + 10;
5      x = 2 * x;
6      x = x - 19;
7      x = x / 3;
8      System.out.println(x);
9    }
10 }
```

What will be printed in the standard output?

Answer

Output for Question 3.8

```
7
```

```
int x = 10; => 10
x = x + 10; => 20
x = 2 * x; => 40
x = x - 19; => 21
x = x / 3; => 7
```

When using several operators in the same expression, one must consider Java's order of precedence. Java uses the standard PEMDAS (Parenthesis, Exponents, Multiplication and Division, Addition and Subtraction) order. When there are multiple instances of the same precedence, Java reads from left to right. Consider what the output of the following code would be:

Code section 3.60: Several operators.

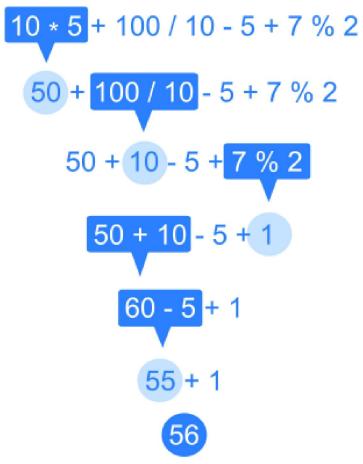
```
1  System.out.println(10*5 + 100/10 - 5 + 7%2);
```

Console for Code section 3.60

```
56
```

The following chart shows how Java would compute this expression:

Figure 3.1: Computation of an arithmetic expression in the Java programming language



Besides performing mathematical functions, there are also operators to assign numbers to variables (each example again uses the variable initialized as `x = 5`):

Code listing 3.11: Assignments.java

```
1  public class Assignments {
2    public static void main(String[] args) {
3      int x = 5;
4      x = 3;
5      System.out.println("Assignment
   (x = 3) : " + x);
6
7      x = 5;
8      x += 5;
9      System.out.println("Assign x plus another integer to itself
   (x += 5): " + x);
10
11     x = 5;
12     x -= 4;
13     System.out.println("Assign x minus another integer to
   itself         (x -= 4): " + x);
14
15     x = 5;
16     x *= 6;
17     System.out.println("Assign x multiplied by another integer
   to itself (x *= 6): " + x);
18
19     x = 5;
20     x /= 5;
21     System.out.println("Assign x divided by another integer to
   itself    (x /= 5): " + x);
22   }
23 }
```

Console for Code listing 3.11

```
Assignment
(x = 3) : 3
Assign x plus another integer to itself
(x += 5): 10
Assign x minus another integer to itself
(x -= 4): 1
Assign x multiplied by another integer to
itself (x *= 6): 30
Assign x divided by another integer to
itself    (x /= 5): 1
```

Using bitwise operators within Java

Java has besides arithmetic operators a set of bit operators to manipulate the bits in a number, and a set of logical operators. The bitwise logical operators are

Operator	Function	Value of x before	Example input	Example output	Value of x after
&	Bitwise AND	7	x&27	3	7
|	Bitwise OR	7	x|27	31	7
^	Bitwise XOR	7	x^27	28	7
~	Bitwise inversion	7	~x	-8	7

Besides these logical bitwise functions, there are also operators to assign numbers to variables (`x = -5`):

Operator	Function	Example input	Example output
&=	Assign x bitwisely ANDed with another value to itself	`x &= 3`	3
|=	Assign x bitwisely ORed with another value to itself	`x |= 3`	-5
^=	Assign x bitwisely XORed with another value to itself	`x ^= 3`	-8
<<=	Assign x divided by another integer to itself	`x <<= 1`	-10
>>=	Assign x bitwisely negated with another value to itself	`x >>= 1`	-3
>>>=	Assign x bitwisely negated with another value to itself	`x >>>= 1`	2,305,843,009,213,693,949 (64 bit)

The shift operators are used to shift the bits to the left or right, which is also a quick way to multiply/divide by two:

Operator	Function	Value of x before	Example input	Example output	Value of x after
<<	Logical shift left	-15	`x << 2`	-60	-15
>>	Arithmetic shift right	-15	`x >> 3`	-2	-15
>>>	Logical shift right	-15	`x >>> 3`	2,305,843,009,213,693,937 (64 bit)	-15

Literals

Java **Literals** are syntactic representations of boolean, character, numeric, or string data. Literals provide a means of expressing specific values in your program. For example, in the following statement, an integer variable named `count` is declared and assigned an integer value. The literal **`0`** represents, naturally enough, the value zero.

Code section 3.61: Numeric literal.

```
1  int count = 0;
```

The code section 3.62 contains two number literals followed by two boolean literals at line 1, one string literal followed by one number literal at line 2, and one string literal followed by one real number literal at line 3:

Code section 3.62: Literals.

```
1  (2 > 3) ? true : false;
2  "text".substring(2);
3  System.out.println("Display a hard coded float: " + 37.19f);
```

Boolean Literals

There are two boolean literals

- **`true`** represents a true boolean value
- **`false`** represents a false boolean value

There are no other boolean literals, because there are no other boolean values!

Numeric Literals

There are three types of numeric literals in Java.

Integer Literals

In Java, you may enter integer numbers in several formats:

1. As decimal numbers such as `1995`, `51966`. Negative decimal numbers such as `-42` are actually *expressions* consisting of the integer literal with the unary negation operation `-`.
2. As octal numbers, using a leading `0` (zero) digit and one or more additional octal digits (digits between `0` and `7`), such as `077`. Octal numbers may evaluate to negative numbers; for example `037777777770` is actually the decimal value -8.
3. As hexadecimal numbers, using the form `0x` (or `0X`) followed by one or more hexadecimal digits (digits from `0` to `9`, `a` to `f` or `A` to `F`). For example, 0xCAFEBABEL is the long integer 3405691582. Like octal numbers, hexadecimal literals may represent negative numbers.
4. Starting in J2SE 7.0, as binary numbers, using the form `0b` (or `0B`) followed by one or more binary digits (0 or 1). For example, 0b101010 is the integer 42. Like octal and hex numbers, binary literals may represent negative numbers.

By default, the integer literal primitive type is **`int`**. If you want a **`long`**, add a letter *el* suffix (either the character `l` or the character `L`) to the integer literal. This suffix denotes a *long integer* rather than a standard integer. For example, `3405691582L` is a long integer literal. Long integers are 8 bytes in length as opposed to the standard 4 bytes for **`int`**. It is best practice to use the suffix `L` instead of `l` to avoid confusion with the digit `1` (one) which looks like `l` in many fonts: `200l` ≠ `2001`. If you want a short integer literal, you have to cast it.

Starting in J2SE 7.0, you may insert underscores between digits in a numeric literal. They are ignored but may help readability by allowing the programmer to group digits.

Floating Point Literals

Floating point numbers are expressed as decimal fractions or as exponential notation:

Code section 3.63: Floating point literals.

```
double decimalNumber = 5.0;
decimalNumber = 89d;
decimalNumber = 0.5;
decimalNumber = 10f;
decimalNumber = 3.14159e0;
decimalNumber = 2.718281828459045D;
decimalNumber = 1.0e-6D;
```

Floating point numbers consist of:

1. an optional leading + or - sign, indicating a positive or negative value; if omitted, the value is positive,
2. one of the following number formats
   - *integer digits* (must be followed by either an exponent or a suffix or both, to distinguish it from an integer literal)
   - *integer digits* `.`
   - *integer digits* `.` *integer digits*
   - `.` *integer digits*
3. an optional exponent of the form
   - the exponent indicator `e` or `E`
   - an optional exponent sign + or - (the default being a positive exponent)
   - *integer digits* representing the integer exponent value
4. an optional floating point suffix:
   - either `f` or `F` indicating a single precision (4 bytes) floating point number, or
   - `d` or `D` indicating the number is a double precision floating point number (by default, thus the double precision (8 bytes) is default).

Here, *integer digits* represents one or more of the digits `0` through `9`.

Starting in J2SE 7.0, you may insert underscores between digits in a numeric literal. They are ignored but may help readability by allowing the programmer to group digits.

Character Literals

Character literals are constant valued character expressions embedded in a Java program. Java characters are sixteen bit Unicode characters, ranging from 0 to 65535. Character literals are expressed in Java as a single quote, the character, and a closing single quote (`'a'`, `'7'`, `'$'`, `'π'`). Character literals have the type **`char`**, an unsigned integer primitive type. Character literals may be safely promoted to larger integer types such as **`int`** and **`long`**. Character literals used where a **`short`** or **`byte`** is called for must be cast to **`short`** or **`byte`** since truncation may occur.

String Literals

String literals consist of the double quote character (") (ASCII 34, hex 0x22), zero or more characters (including Unicode characters), followed by a terminating double quote character ("), such as: `"Ceci est une string."`

So a string literal follows the following grammar:

```
<STRING :
        "\""
        (    (~["\"","\\","\n","\r"])
        |("\\"
```

```
            ( ["n","t","b","r","f","\\","'","\""]
            |["0"-"7"](["0"-"7"])?
            |["0"-"3"]["0"-"7"]["0"-"7"]
            )
        )
        )*
        "\"">
```

Within string and character literals, the backslash character can be used to escape special characters, such as unicode escape sequences, or the following special characters:

Name	Character	ASCII	hex
Backspace	\b	8	0x08
TAB	\t	9	0x09
NUL character	\0	0	0x00
newline	\n	10	0x0a
carriage control	\r	13	0xd
double quote	\"	34	0x22
single quote	\'	39	0x27
backslash	\\	92	0x5c

String literals may not contain unescaped newline or linefeed characters. However, the Java compiler will evaluate compile time expressions, so the following String expression results in a string with three lines of text:

Code section 3.64: Multi-line string.

```
1  String text = "This is a String literal\n"
2              + "which spans not one and not two\n"
3              + "but three lines of text.\n";
```

null

`null` is a special Java literal which represents a *null value*: a value which does not refer to any object. It is an error to attempt to dereference the null value — Java will throw a `NullPointerException`. `null` is often used to represent uninitialized state.

Mixed Mode Operations

In concatenation operations, the values in brackets are concatenated first. Then the values are concatenated from the left to the right. Be careful when mixing character literals and integers in String concatenation operations:

Code section 3.65: Concatenation operations.

```
1  int one = '1';
2  int zero = '0';
3
4  System.out.println("120? " + one + '2' + zero);
```

Console for Code section 3.65

```
120? 49248
```

The unexpected results arise because `'1'` and `'0'` are converted twice. The expression is concatenated as such:

```
"120? " + one + '2' + zero
"120? " + 49 + '2' + 48
"120? 49" + '2' + 48
"120? 492" + 48
"120? 49248"
```

1. one and zero are integers. So they store integer values. The integer value of `'1'` is 49 and the integer value of `'0'` is 48.

2. So the first concatenation concatenates `"120? "` and 49. 49 is first converted into String, yielding `"49"` and the concatenation returns the string `"120? 49"`.
3. The second concatenation concatenates `"120? 49"` and `'2'`. `'2'` is converted into the String `"2"` and the concatenation returns the string `"120? 492"`.
4. The concatenation between `"120? 492"` and `'0'` returns the string `"120? 49248"`.

The code section 66 yields the desired result:

Code section 3.66: Correct primitive type.

```
1  char one = '1';
2  char zero = '0';
3
4  System.out.println("120? " + one + '2' + zero);
```

Console for Code section 3.66

```
120? 120
```

Test your knowledge

Question 3.9: Consider the following code:

Question 3.9: New concatenation operations.

```
1  int one = '1';
2  int zero = '0';
3
4  System.out.println("  3? " + (one + '2' + zero));
5  System.out.println("102? " + 100 + '2' + 0);
6  System.out.println("102? " + (100 + '2' + 0));
```

Console for Question 3.9

```
  3? 147
102? 10020
102? 150
```

Explain the results seen.

Answer

For the first line:

```
" 3? " + (one + '2' + zero)
 " 3? " + (49 + '2' + 48)
   " 3? " + (99 + 48)
     " 3? " + 147
      " 3? 147"
```

For the second line:

```
"102? " + 100 + '2' + 0
 "102? 100" + '2' + 0
   "102? 1002" + 0
    "102? 10020"
```

For the last line:

```
"102? " + (100 + '2' + 0)
  "102? " + (150 + 0)
    "102? " + 150
     "102? 150"
```

Methods

Methods are how we communicate with objects. When we invoke or call a method we are asking the object to carry out a task. We can say methods implement the behaviour of objects. For each method we need to give a name, we need to define its input parameters and we need to define its return type. We also need to set its visibility (private, protected or public). If the method throws a checked exception, that needs to be declared as well. It is called a *method definition*. The syntax of method definition is:

```
MyClass {
  ...
  public ReturnType methodName(ParamOneType parameter1, ParamTwoType parameter2) {
      ...
      return returnValue;
  }
  ...
}
```

We can declare that the method does not return anything using the **`void`** Java keyword. For example:

Code section 3.67: Method without returned data.

```
private void methodName(String parameter1, String parameter2) {
  ...
  return;
}
```

When the method returns nothing, the **`return`** keyword at the end of the method is optional. When the execution flow reaches the **`return`** keyword, the method execution is stopped and the execution flow returns to the caller method. The **`return`** keyword can be used anywhere in the method as long as there is a way to execute the instructions below:

Code section 3.68: `return` keyword location.

```
1  private void aMethod(int a, int b) {
2    int c = 0;
3    if (a > 0) {
4      c = a;
5      return;
6    }
7    int c = c + b;
8    return;
9    int c = c * 2;
10 }
```

In the code section 3.68, the **`return`** keyword at line 5 is well placed because the instructions below can be reached when `a` is negative or equal to 0. However, the **`return`** keyword at line 8 is badly placed because the instructions below can't be reached.

Test your knowledge

Question 3.9: Consider the following code:

Question 3.9: Compiler error.

```
1  private int myMethod(int a, int b, boolean c) {
2    b = b + 2;
3    if (a > 0) {
4      a = a + b;
5      return a;
6    } else {
7      a = 0;
8    }
9  }
```

The code above will return a compiler error. Why?

Answer

Answer 3.9: Compiler error.

```
1  private int myMethod(int a, int b, boolean c) {
2    b = b + 2;
3    if (a > 0) {
4      a = a + b;
5      return a;
6    } else {
7      a = 0;
8    }
9  }
```

The method is supposed to return a **`int`** but when a is negative or equal to 0, it returns nothing.

Parameter passing

We can pass any primitive data types or reference data type to a method.

Primitive type parameter

The primitive types are *passed in by value*. It means that as soon as the primitive type is passed in, there is no more link between the value inside the method and the source variable:

Code section 3.69: A method modifying a variable.

```
1  private void modifyValue(int number) {
2    number += 1;
3  }
```

Code section 3.70: Passing primitive value to method.

```
1  int i = 0;
2  modifyValue(i);
3  System.out.println(i);
```

Output for Code section 3.70

```
0
```

As you can see in code section 3.70, the `modifyValue()` method has not modified the value of `i`.

Reference type parameter

The object references are passed by value. It means that:

- There is no more link between the reference inside the method and the source reference,
- The source object itself and the object itself inside the method are still the same.

You must understand the difference between the reference of an object and the object itself. An *object reference* is the link between a variable name and an instance of object:

`Object object` ⇔ `new Object()`

An object reference is a pointer, an address to the object instance.

The object itself is the value of its attributes inside the object instance:

object.firstName ⇒ "James"
object.lastName ⇒ "Gosling"
object.birthDay ⇒ "May 19"

Take a look at the example above:

Code section 3.71: A method modifying an object.

```
1  private void modifyObject(FirstClass anObject) {
2    anObject.setName("Susan");
3  }
```

Code section 3.72: Passing reference value to method.

```
1  FirstClass object = new FirstClass();
2  object.setName("Christin");
3
4  modifyObject(object);
5
6  System.out.println(object.getName());
```

Output for Code section 3.72

```
Susan
```

The name has changed because the method has changed the object itself and not the reference. Now take a look at the other example:

Code section 3.73: A method modifying an object reference.

```
1  private void modifyObject(FirstClass anObject) {
2    anObject = new FirstClass();
3    anObject.setName("Susan");
4  }
```

Code section 3.74: Passing reference value to method.

```
1  FirstClass object = new FirstClass();
2  object.setName("Christin");
3
4  modifyObject(object);
5
6  System.out.println(object.getName());
```

Output for Code section 3.74

```
Christin
```

The name has not changed because the method has changed the reference and not the object itself. The behavior is the same as if the method was in-lined and the parameters were assigned to new variable names:

Code section 3.75: In-lined method.

```
1  FirstClass object = new FirstClass();
2  object.setName("Christin");
3
4  // Start of the method
5  FirstClass anObject = object;
6  anObject = new FirstClass();
7  anObject.setName("Susan");
8  // End of the method
9
10 System.out.println(object.getName());
```

Output for Code section 3.75

```
Christin
```

Variable argument list

Java SE 5.0 added syntactic support for methods with variable argument list (http://docs.oracle.com/javase/1.5.0/docs/guide/language/varargs.html), which simplifies the typesafe usage of methods requiring a variable number of arguments. Less formally, these parameters are called *varargs*[2] (http://www.javabeat.net/qna/645-varargs-in-java-50/). The type of a variable parameter must be followed with . . . , and Java will box all the arguments into an array:

Code section 3.76: A method using vararg parameters.

```
1  public void drawPolygon(Point... points) {
2    //…
3  }
```

When calling the method, a programmer can simply separate the points by commas, without having to explicitly create an array of `Point` objects. Within the method, the points can be referenced as `points[0]`, `points[1]`, etc. If no points are passed, the array has a length of zero.

A method can have both normal parameters and a variable parameter but the variable parameter must always be the last parameter. For instance, if the programmer is required to use a minimum number of parameters, those parameters can be specified before the variable argument:

Code section 3.77: Variable arguments.

```
// A polygon needs at least three points.
public void drawPolygon(Point p1, Point p2, Point p3, Point... otherPoints) {
  //…
}
```

Return parameter

A method may return a value (which can be a primitive type or an object reference). If the method does not return a value we use the **`void`** Java keyword.

However, a method can return only one value so what if you want to return more than one value from a method? You can pass in an object reference to the method, and let the method modify the object properties so the modified values can be considered as an output value from the method. You can also create an Object array inside the method, assign the return values and return the array to the caller. However, this gives a problem if you want to mix primitive data types and object references as the output values from the method.

There is a better approach, define a special return object with the needed return values. Create that object inside the method, assign the values and return the reference to this object. This special object is "bound" to this method and used only for returning values, so do not use a public class. The best way is to use a nested class, see example below:

Code listing 3.12: Multiple returned variables.

```
public class MyObject {
  ...

  /** Nested object is for return values from getPersonInfoById method */
  private static class ReturnObject {
      private int age;
      private String name;

      public void setAge(int age) {
          this.age = age;
      }

      public int getAge() {
          return age;
      }

      public void setName(String name) {
          name = name;
      }

      public String getName() {
          return name;
      }
  } // End of nested class definition

  /** Method using the nested class to return values */
  public ReturnObject getPersonInfoById(int id) {
    int    age;
    String name;
    ...
    // Get the name and age based on the ID from the database
    ...
    ReturnObject result = new ReturnObject();
    result.setAge(age);
    result.setName(name);

    return result;
  }
}
```

In the above example the `getPersonInfoById` method returns an object reference that contains both values of the name and the age. See below how you may use that object:

Code section 3.78: Retrieving the values.

```
1  MyObject object = new MyObject();
2  MyObject.ReturnObject person = object.getPersonInfoById(102);
3
4  System.out.println("Person Name=" + person.getName());
5  System.out.println("Person Age =" + person.getAge());
```

Test your knowledge

Question 3.10: Consider the following code:

Question 3.10: Compiler error.

```
1  private int myMethod(int a, int b, String c) {
2    if (a > 0) {
3      c = "";
4      return c;
5    }
6    int b = b + 2;
7    return b;
8  }
```

The code above will return a compiler error. Why?

Answer

Answer 3.10: Compiler error.

```
1  private int myMethod(int a, int b, String c) {
2    if (a > 0) {
3      c = "";
4      return c;
5    }
6    int b = b + 2;
7    return b;
8  }
```

The method is supposed to return a **`int`** but at line 4, it returns `c`, which is a String.

Special method, the constructor

The constructor is a special method called automatically when an object is created with the **`new`** keyword. Constructor does not have a return value and its name is the same as the class name. Each class must have a constructor. If we do not define one, the compiler will create a default so called **empty constructor** automatically.

`Code listing 3.13: Automatically created constructor.`

```
1  public class MyClass {
2    /**
3    * MyClass Empty Constructor
4    */
5    public MyClass() {
6    }
7  }
```

Static methods

A *static method* is a method that can be called without an object instance. It can be called on the class directly. For example, the `valueOf(String)` method of the `Integer` class is a static method:

Code section 3.79: Static method.

```
Integer i = Integer.valueOf("10");
```

The static keyword makes attributes instance-agnostic. This means that you cannot reference a static attribute of a single object (because such a specific object attribute doesn't exist). Instead, only one instance of a static attribute exists, whether there is one object in the JVM or one hundred. Here is an example of using a static attribute in a static method:

Code section 3.80: Static attribute.

```
private static int count = 0;

public static int getNewInteger() {
  return count++;
}
```

You can notice that when you use `System.out.println()`, `out` is a static attribute of the `System` class. A static attribute is related to a class, not to any object instance. This is how Java achieves one universal output stream that we can use to print output. Here is a more complex use case:

Code listing 3.14: A static attribute.

```
public class MyProgram {

    public static int count = 0;

    public static void main (String[] args) {
        MyProgram.count++;

        MyProgram program1 = new MyProgram();
        program1.count++;

        MyProgram program2 = new MyProgram();
        program2.count++;

        new MyProgram().count++;
        System.out.println(MyProgram.count);
    }
}
```

Output for Code listing 3.14

```
4
```

Test your knowledge

Question 3.11: Visit the Oracle JavaDoc of the class `java.lang.Integer` (http://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html).

How many static fields does this class have?

Answer

4.

- `int MAX_VALUE`,
- `int MIN_VALUE`,
- `int SIZE` and
- `Class<Integer> TYPE`.

To learn how to overload and override a method, see Overloading Methods and Constructors.

API/java.lang.String

`String` is a class built into the Java language defined in the `java.lang` package. It represents character strings. Strings are ubiquitous in Java. Study the `String` class and its methods carefully. It will serve you well to know how to manipulate them skillfully. String literals in Java programs, such as "abc", are implemented as instances of this class like this:

Code section 3.81: String example.

```
1  String str = "This is string literal";
```

On the right hand side a String object is created represented by the string literal. Its object reference is assigned to the `str` variable.

Immutability

Strings are *immutable*; that is, they cannot be modified once created. Whenever it looks as if a String object was modified actually a new String object was created. For instance, the `String.trim()` method returns the string with leading and trailing whitespace removed. Actually, it creates a new trimmed string and then returns it. Pay attention on what happens in Code section 3.82:

Code section 3.82: Immutability.

```
1  String badlyCutText = "     Java is great.     ";
2  System.out.println(badlyCutText);
3
4  badlyCutText.trim();
5  System.out.println(badlyCutText);
```

Output for Code section 3.82

```
     Java is great.
     Java is great.
```

The `trim()` method call does not modify the object so nothing happens. It creates a new trimmed string and then throws it away.

Code section 3.83: Assignment.

```
1  String badlyCutText = "     Java is great.     ";
2  System.out.println(badlyCutText);
3
4  badlyCutText = badlyCutText.trim();
5  System.out.println(badlyCutText);
```

Output for Code section 3.83

```
     Java is great.
Java is great.
```

The returned string is assigned to the variable. It does the job as the `trim()` method has created a new `String` instance.

Concatenation

The Java language provides special support for the string concatenation with operator +:

Code section 3.84: Examples of concatenation.

```
1  System.out.println("First part");
2  System.out.println(" second part");
3  String str = "First part" + " second part";
4  System.out.println(str);
```

Output for Code section 3.84

```
First part
 second part
First part second part
```

The concatenation is not always processed at the same time. Raw string literals concatenation is done at compile time, hence there is a single string literal in the byte code of the class. Concatenation with at least one object is done at runtime.

+ operator can concatenate other objects with strings. For instance, integers will be converted to strings before the concatenation:

Code section 3.85: Concatenation of integers.

```
1  System.out.println("Age=" + 25);
```

Output for Code section 3.85

```
Age=25
```

Each Java object has the `String toString()` inherited from the `Object` class. This method provides a way to convert objects into `Strings`. Most classes override the default behavior to provide more specific (and more useful) data in the returned `String`:

Code section 3.86: Concatenation of objects.

```
1  System.out.println("Age=" + new Integer(31));
```

Output for Code section 3.86

```
Age=31
```

Using StringBuilder/StringBuffer to concatenate strings

Remember that `String` objects are immutable objects. Once a `String` is created, it can not be modified, takes up memory until garbage collected. Be careful of writing a method like this:

Code section 3.87: Raw concatenation.

```
1  public String convertToString(Collection<String> words) {
2    String str = "";
3    // Loops through every element in words collection
4    for (String word : words) {
5      str = str + word + " ";
6    }
7    return str;
8  }
```

On the + operation a new `String` object is created at each iteration. Suppose `words` contains the elements `["Foo", "Bar", "Bam", "Baz"]`. At runtime, the method creates thirteen `Strings`:

1. `""`
2. `"Foo"`
3. `" "`
4. `"Foo "`
5. `"Foo Bar"`
6. `" "`
7. `"Foo Bar "`
8. `"Foo Bar Bam"`
9. `" "`
10. `"Foo Bar Bam "`
11. `"Foo Bar Bam Baz"`
12. `" "`
13. `"Foo Bar Bam Baz "`

Even though only the last one is actually useful.

To avoid unnecessary memory use like this, use the `StringBuilder` class. It provides similar functionality to `Strings`, but stores its data in a mutable way. Only one `StringBuilder` object is created. Also because object creation is time consuming, using `StringBuilder` produces much faster code.

Code section 3.88: Concatenation with `StringBuilder`.

```java
public String convertToString(Collection<String> words) {
  StringBuilder buf = new StringBuilder();
  // Loops through every element in words collection
  for (String word : words) {
    buf.append(word);
    buf.append(" ");
  }
  return buf.toString();
}
```

As `StringBuilder` isn't thread safe (see the chapter on Concurrency) you can't use it in more than one thread. For a multi-thread environment, use `StringBuffer` instead which does the same and is thread safe. However, `StringBuffer` is slower so only use it when it is required. Moreover, before Java 5 only `StringBuffer` existed.

Comparing Strings

Comparing strings is not as easy as it may first seem. Be aware of what you are doing when comparing `String`'s using `==`:

Code section 3.89: Dangerous comparison.

```java
String greeting = "Hello World!";
if (greeting == "Hello World!") {
    System.out.println("Match found.");
}
```

Output for Code section 3.89

```
Match found.
```

The difference between the above and below code is that the above code checks to see if the `String`'s are the same objects in memory which they are. This is as a result of the fact that `String`'s are stored in a place in memory called the String Constant Pool. If the `new` keyword is not explicitly used when creating the `String` it checks to see if it already exists in the Pool and uses the existing one. If it does not exist, a new Object is created. This is what allows Strings to be immutable in Java. To test for equality, use the `equals(Object)` method inherited by every class and defined by `String` to return **`true`** if and only if the object passed in is a `String` contains the exact same data:

Code section 3.90: Right comparison.

```java
String greeting = "Hello World!";
if (greeting.equals("Hello World!")) {
    System.out.println("Match found.");
}
```

Output for Code section 3.90

```
Match found.
```

Remember that the comparison is case sensitive.

Code section 3.91: Comparison with lowercase.

```java
String greeting = "Hello World!";
if (greeting.equals("hello world!")) {
    System.out.println("Match found.");
}
```

Output for Code section 3.91

```

```

To order `String` objects, use the **`compareTo()`** method, which can be accessed wherever we use a String datatype. The `compareTo()` method returns a negative, zero, or positive number if the parameter is less than, equal to, or greater than the object on which it is called. Let's take a look at an example:

Code section 3.92: Order.

```java
String person1 = "Peter";
String person2 = "John";
if (person1.compareTo(person2) > 0) {
    // Badly ordered
    String temp = person1;
    person1 = person2;
    person2 = temp;
}
```

The code section 3.92 is comparing the String variable `person1` to `person2`. If `person1` is different even in the slightest manner, we will get a value above or below 0 depending on the exact difference. The result is negative if this String object lexicographically precedes the argument string. The result is positive if this String object lexicographically follows the argument string. Take a look at the Java API (http://docs.oracle.com/javase/7/docs/api/java/lang/String.html#compareTo%28java.lang.String%29) for more details.

Splitting a String

Sometimes it is useful to split a string into separate strings, based on a *regular expressions*. The `String` class has a **`split()`** method, since Java 1.4, that will return a String array:

Code section 3.93: Order.

```
String person = "Brown, John:100 Yonge Street, Toronto:(416)777-9999";
...
String[] personData = person.split(":");
...
String name    = personData[0];
String address = personData[1];
String phone   = personData[2];
```

Another useful application could be to *split* the String text based on the new line character, so you could process the text line by line.

Substrings

It may also be sometimes useful to create **substrings**, or strings using the order of letters from an existing string. This can be done in two methods.

The first method involves creating a substring out of the characters of a string from a given index to the end:

Code section 3.94: Truncating string.

```
String str = "coffee";
System.out.println(str.substring(3));
```

Output for Code section 3.94

```
fee
```

The index of the first character in a string is 0.

```
c o f f e e
0 1 2 3 4 5
```

By counting from there, it is apparent that the character in index 3 is the second "f" in "coffee". This is known as the `beginIndex`. All characters from the `beginIndex` until the end of the string will be copied into the new substring.

The second method involves a user-defined `beginIndex` and `endIndex`:

Code section 3.95: Extraction of string.

```
String str = "supporting";
System.out.println(str.substring(3, 7));
```

Output for Code section 3.95

```
port
```

The string returned by `substring()` would be "port".

```
s u p p o r t i n g
0 1 2 3 4 5 6 7 8 9
```

Please note that the endIndex is **not** inclusive. This means that the last character will be of the index `endIndex-1`. Therefore, in this example, every character from index 3 to index 6, inclusive, was copied into the substring.

It is easy to mistake the method `substring()` for `subString()` (which does not exist and would return with a syntax error on compilation). *Substring* is considered to be one word. This is why the method name does not seem to follow the common syntax of Java. Just remember that this style only applies to methods or other elements that are made up of more than one word.

String cases

The `String` class also allows for the modification of cases. The two methods that make this possible are `toLowerCase()` and `toUpperCase()`.

Code section 3.96: Case modification.

```
1  String str = "wIkIbOoKs";
2  System.out.println(str.toLowerCase());
3  System.out.println(str.toUpperCase());
```

Output for Code section 3.96

```
wikibooks
WIKIBOOKS
```

These methods are useful to do a search which is not case sensitive:

Code section 3.97: Text search.

```
1  String word = "Integer";
2  String text = "A number without a decimal part is an integer."
3    + " Integers are a list of digits.";
4
5  ...
6
7  // Remove the case
8  String lowerCaseWord = word.toLowerCase();
9  String lowerCaseText = text.toLowerCase();
10
11 // Search
12 int index = lowerCaseText.indexOf(lowerCaseWord);
13 while (index != -1) {
14   System.out.println(word
15     + " appears at column "
16     + (index + 1)
17     + ".");
18   index = lowerCaseText.indexOf(lowerCaseWord, index + 1);
19 }
```

Output for Code section 3.97

```
Integer appears at column 38.
Integer appears at column 47.
```

Test your knowledge

Question 3.12: You have mail addresses in the following form: *<firstName>.<LastName>@<companyName>*.org

Write the `String getDisplayName(String)` method that receives the mail string as parameter and returns the readable person name like this: `LASTNAME Firstname`

Answer

Answer 3.12: `getDisplayName()`

```
1  public static String getDisplayName(String mail) {
2    String displayName = null;
3
4    if (mail != null) {
5      String[] mailParts = mail.split("@");
6      String namePart = mailParts[0];
7      String[] namesParts = namePart.split("\\.");
8
9      // The last name
10     String lastName = namesParts[1];
11     lastName = lastName.toUpperCase();
12
13     // The first name
14     String firstName = namesParts[0];
15
16     String firstNameInitial = firstName.substring(0, 1);
17     firstNameInitial = firstNameInitial.toUpperCase();
18
19     String firstNameEnd = firstName.substring(1);
20     firstNameEnd = firstNameEnd.toLowerCase();
21
```

```
    // Concatenation
    StringBuilder displayNameBuilder = new StringBuilder(lastName).append("
").append(firstNameInitial).append(firstNameEnd);
    displayName = displayNameBuilder.toString();
  }

  return displayName;
}
```

1. We only process non null strings,
2. We first split the mail into two parts to separate the personal information from the company information and we keep the name data,
3. Then we split the name information to separate the first name from the last name. As the `split()` method use regular expression and `.` is a wildcard character, we have to escape it (`\.`). However, in a string, the `\` is also a special character, so we need to escape it too (`\\.`),
4. The last name is just capitalized,
5. As the case of all the first name characters will not be the same, we have to cut the first name. Only the first name initial will be capitalized,
6. Now we can concatenate all the fragments. We prefer to use a `StringBuilder` to do that.

See also

- Java API: java.lang.String (http://docs.oracle.com/javase/7/docs/api/java/lang/String.html)
- Java API: java.lang.StringBuffer (http://docs.oracle.com/javase/7/docs/api/java/lang/StringBuffer.html)
- Java API: java.lang.StringBuilder (http://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html)

Classes, Objects and Types

An **object** is composed of **fields** and **methods**. The fields, also called *data members*, *characteristics*, *attributes*, or *properties*, describe the state of the object. The methods generally describe the actions associated with a particular object. Think of an object as a noun, its fields as adjectives describing that noun, and its methods as the verbs that can be performed by or on that noun.

For example, a sports car is an object. Some of its fields might be its height, weight, acceleration, and speed. An object's fields just hold data about that object. Some of the methods of the sports car could be "drive", "park", "race", etc. The methods really don't mean much unless associated with the sports car, and the same goes for the fields.

The blueprint that lets us build our sports car object is called a *class*. A class doesn't tell us how fast our sports car goes, or what color it is, but it does tell us that our sports car will have a field representing speed and color, and that they will be say, a number and a word (or hex color code), respectively. The class also lays out the methods for us, telling the car how to park and drive, but these methods can't take any action with just the blueprint — they need an object to have an effect.

In Java, a class is located in a file similar to its own name. If you want to have a class called `SportsCar`, its source file needs to be `SportsCar.java`. The class is created by placing the following in the source file:

Code listing 3.13: SportsCar.java

```
public class SportsCar {
   /* Insert your code here */
}
```

The class doesn't do anything yet, as you will need to add methods and field variables first.

The objects are different from the primitive types because:

1. The primitive types are not instantiated.

2. In the memory, for a primitive type only its value is stored. For an object, also a reference to an instance can be stored.
3. In the memory, the allocated space of a primitive type is fixed, whatever their value. The allocated space of an object can vary, for instance either the object is instantiated or not.
4. The primitive types don't have methods callable on them.
5. A primitive type can't be inherited.

Instantiation and constructors

In order to get from class to object, we "build" our object by *instantiation*. Instantiation simply means to create an *instance* of a class. Instance and object are very similar terms and are sometimes interchangeable, but remember that an instance refers to a *specific object*, which was created from a class.

This instantiation is brought about by one of the class's methods, called a *constructor*. As its name implies, a constructor builds the object based on the blueprint. Behind the scenes, this means that computer memory is being allocated for the instance, and values are being assigned to the data members.

In general there are four constructor types: default, non-default, copy, and cloning.

A **default constructor** will build the most basic instance. Generally, this means assigning all the fields values like null, zero, or an empty string. Nothing would stop you, however, from setting the color of your default sports car color to red, but this is generally bad programming style. Another programmer would be confused if your basic car came out red instead of say, colorless.

Code section 3.79: A default constructor.

```
1  SportsCar car = new SportsCar();
```

A **non-default constructor** is designed to create an object instance with prescribed values for most, if not all, of the object's fields. The car is red, goes from 0-60 in 12 seconds, tops out at 190mph, etc.

Code section 3.80: A non-default constructor.

```
1  SportsCar car = new SportsCar("red", 12, 190);
```

A **copy constructor** is not included in the Java language, however one can easily create a constructor that does the same as a copy constructor. It's important to understand what it is. As the name implies, a copy constructor creates a new instance to be a duplicate of an already existing one. In Java, this can be also accomplished by creating the instance with the default constructor, and then using the assignment operator to equivocate them. This is not possible in all languages though, so just keep the terminology under your belt.

Java has the concept of **cloning an object**, and the end results are similar to the copy constructor. Cloning an object is faster than creation with the `new` keyword, because all the object memory is copied at once to the destination cloned object. This is possible by implementing the `Cloneable` interface, which allows the method `Object.clone()` to perform a field-by-field copy.

Code section 3.81: Cloning object.

```
1  SportsCar car = oldCar.clone();
```

Type

When an object is created, a reference to the object is also created. The object can not be accessed directly in Java, only through this object reference. This object reference has a *type* assigned to it. We need this type when passing the object reference to a method as a parameter. Java does strong type checking.

Type is basically a list of features/operations, that can be performed through that object reference. The object reference type is basically a contract that guarantees that those operations will be there at run time.

When a car is created, it comes with a list of features/operations listed in the user manual that guarantees that those will be there when the car is used.

When you create an object from a class by default its type is the same as its class. It means that all the features/operations the class defined are there and available, and can be used. See below:

Code section 3.82: Default type.

```
(new ClassName()).operations();
```

You can assign this to a variable having the same type as the class:

Code section 3.83: A variable having the same type as the class.

```
ClassName objRefVariable = new ClassName();
objRefVariable.operations();
```

You can assign the created object reference to the class, super class, or to an interface the class implements:

Code section 3.84: Using the super class.

```
SuperClass objectRef = new ClassName(); // features/operations list are defined by the SuperClass class
...
Interface inter = new ClassName(); // features/operations list are defined by the interface
```

In the car analogy, the created car may have different **Type**s of drivers. We create separate user manuals for them, an Average user manual, a Power user manual, a Child user manual, or a Handicapped user manual. Each type of user manual describes only those features/operations appropriate for the type of driver. For instance, the Power driver may have additional gears to switch to higher speeds, that are not available to other type of users...

When the car key is passed from an adult to a child we are replacing the user manuals, that is called *Type Casting*.

In Java, casts can occur in three ways:

- up casting going up in the inheritance tree, until we reach the `Object`
- up casting to an interface the class implements
- down casting until we reach the class the object was created from

Autoboxing/unboxing

Autoboxing and unboxing, language features since Java 1.5, make the programmer's life much easier when it comes to working with the primitive wrapper types. Consider this code fragment:

Code section 3.85: Traditional object creation.

```
int age = 23;
Integer ageObject = new Integer(age);
```

Primitive wrapper objects were Java's way of allowing one to treat primitive data types as though they were objects. Consequently, one was expected to *wrap* one's primitive data type with the corresponding primitive wrapper object, as shown above.

Since Java 1.5, one may write as below and the compiler will automatically create the wrap object. The extra step of wrapping the primitive is no longer required. It has been *automatically boxed up* on your behalf:

Code section 3.86: Autoboxing.

```
int age = 23;
Integer ageObject = age;
```

Keep in mind that the compiler still creates the missing wrapper code, so one doesn't really gain anything performance-wise. Consider this feature a programmer convenience, not a performance booster.

Each primitive type has a class wrapper:

Primitive type	Class wrapper
`byte`	`java.lang.Byte`
`char`	`java.lang.Character`
`short`	`java.lang.Short`
`int`	`java.lang.Integer`
`long`	`java.lang.Long`
`float`	`java.lang.Float`
`double`	`java.lang.Double`
`boolean`	`java.lang.Boolean`
`void`	`java.lang.Void`

Unboxing uses the same process in reverse. Study the following code for a moment. The **`if`** statement requires a **`boolean`** primitive value, yet it was given a Boolean wrapper object. No problem! Java 1.5 will automatically *unbox* this.

Code section 3.87: Unboxing.

```
1  Boolean canMove = new Boolean(true);
2
3  if (canMove) {
4    System.out.println("This code is legal in Java 1.5");
5  }
```

Test your knowledge

Question 3.11: Consider the following code:

Question 3.11: Autoboxing/unboxing.

```
5  Integer a = 10;
6  Integer b = a + 2;
7  System.out.println(b);
```

How many autoboxings and unboxings are there in this code?

Answer

Answer 3.11: Autoboxing/unboxing.

```
1  Integer a = 10;
2  Integer b = a + 2;
3  System.out.println(b);
```

3

- 1 autoboxing at line 1 to assign.
- 1 unboxing at line 2 to do the addition.
- 1 autoboxing at line 2 to assign.
- No autoboxing nor unboxing at line 3 as `println()` supports the `Integer` class as parameter.

Methods in the `Object` class

Methods in the `java.lang.Object` class are inherited, and thus shared in common by all classes.

The `clone` method

The `java.lang.Object.clone()` method returns a new object that is a copy of the current object. Classes must implement the marker interface `java.lang.Cloneable` to indicate that they can be cloned.

The `equals` method

The `java.lang.Object.equals(java.lang.Object)` method compares the object to another object and returns a `boolean` result indicating if the two objects are equal. Semantically, this method compares the contents of the objects whereas the equality comparison operator "==" compares the object references. The `equals` method is used by many of the data structure classes in the `java.util` package. Some of these data structure classes also rely on the `Object.hashCode` method—see the `hashCode` method for details on the contract between `equals` and `hashCode`. Implementing equals() isn't always as easy as it seems, see 'Secrets of equals() (http://www.angelikalanger.com/Articles/JavaSolutions/SecretsOfEquals/Equals.html)' for more information.

The `finalize` method

The `java.lang.Object.finalize()` method is called exactly once before the garbage collector frees the memory for object. A class overrides `finalize` to perform any clean up that must be performed before an object is reclaimed. Most objects do not need to override `finalize`.

There is no guarantee when the `finalize` method will be called, or the order in which the `finalize` method will be called for multiple objects. If the JVM exits without performing garbage collection, the OS may free the objects, in which case the `finalize` method doesn't get called.

The `finalize` method should always be declared `protected` to prevent other classes from calling the `finalize` method.

```
protected void finalize() throws Throwable { ... }
```

The `getClass` method

The `java.lang.Object.getClass()` method returns the `java.lang.Class` object for the class that was used to instantiate the object. The class object is the base class of reflection in Java. Additional reflection support is provided in the `java.lang.reflect` package.

The `hashCode` method

The `java.lang.Object.hashCode()` method returns an integer (`int`). This integer can be used to distinguish objects although not completely. It quickly separates most of the objects and those with the same *hash code* are separated later in another way. It is used by the classes that provide associative arrays, for instance, those that implement the `java.util.Map` interface . They use the *hash code* to store the object in the associative array. A good `hashCode` implementation will return a hash code:

- Stable: does not change
- Evenly distributed: the hash codes of unequal objects tend to be unequal and the hash codes are evenly distributed across integer values.

The second point means that two different objects can have the same *hash code* so two objects with the same *hash code* are not necessarily the same!

Since associative arrays depend on both the `equals` and `hashCode` methods, there is an important contract between these two methods that must be maintained if the objects are to be inserted into a `Map`:

For two objects *a* and *b*

- `a.equals(b) == b.equals(a)`
- if `a.equals(b)` then `a.hashCode() == b.hashCode()`
- but ~~`if a.hashCode() == b.hashCode() then a.equals(b)`~~

In order to maintain this contract, a class that overrides the `equals` method must also override the `hashCode` method, and vice versa, so that `hashCode` is based on the same properties (or a subset of the properties) as `equals`.

A further contract that the map has with the object is that the results of the `hashCode` and `equals` methods will not change once the object has been inserted into the map. For this reason, it is generally a good practice to base the hash function on immutable properties of the object.

The `toString` method

The `java.lang.Object.toString()` method returns a `java.lang.String` that contains a text representation of the object. The ***`toString`*** method is implicitly called by the compiler when an object operand is used with the string concatenation operators (+ and +=).

The wait and notify thread signaling methods

Every object has two wait lists for threads associated with it. One wait list is used by the `synchronized` keyword to acquire the mutex lock associated with the object. If the mutex lock is currently held by another thread, the current thread is added to the list of blocked threads waiting on the mutex lock. The other wait list is used for signaling between threads accomplished through the `wait` and `notify` and `notifyAll` methods.

Use of wait/notify allows efficient coordination of tasks between threads. When one thread needs to wait for another thread to complete an operation, or needs to wait until an event occurs, the thread can suspend its execution and wait to be notified when the event occurs. This is in contrast to polling, where the thread repeatedly sleeps for a short period of time and then checks a flag or other condition indicator. Polling is both more computationally expensive, as the thread has to continue checking, and less responsive since the thread won't notice the condition has changed until the next time to check.

The `wait` methods

There are three overloaded versions of the `wait` method to support different ways to specify the timeout value: `java.lang.Object.wait()`, `java.lang.Object.wait(long)` and `java.lang.Object.wait(long, int)`. The first method uses a timeout value of zero (0), which means that the wait does not timeout; the second method takes the number of milliseconds as a timeout; the third method takes the number of nanoseconds as a timeout, calculated as `1000000 * timeout + nanos`.

The thread calling `wait` is blocked (removed from the set of executable threads) and added to the object's wait list. The thread remains in the object's wait list until one of three events occurs:

1. another thread calls the object's `notify` or `notifyAll` method;
2. another thread calls the thread's `java.lang.Thread.interrupt` method; or
3. a non-zero timeout that was specified in the call to `wait` expires.

The `wait` method must be called inside of a block or method synchronized on the object. This insures that there are no race conditions between `wait` and `notify`. When the thread is placed in the wait list, the thread releases the object's mutex lock. After the thread is removed from the wait list and added to the set of executable threads, it must acquire the object's mutex lock before continuing execution.

The `notify` and `notifyAll` methods

The `java.lang.Object.notify()` and `java.lang.Object.notifyAll()` methods remove one or more threads from an object's wait list and add them to the set of executable threads. `notify` removes a single thread from the wait list, while `notifyAll` removes all threads from the wait list. Which thread is removed by `notify` is unspecified and dependent on the JVM implementation.

The notify methods must be called inside of a block or method synchronized on the object. This insures that there are no race conditions between `wait` and `notify`.

Keywords

Keywords are special tokens in the language which have reserved use in the language. Keywords may not be used as identifiers in Java — you cannot declare a field whose name is a keyword, for instance.

Examples of keywords are the primitive types, `int` and `boolean`; the control flow statements `for` and `if`; access modifiers such as `public`, and special words which mark the declaration and definition of Java classes, packages, and interfaces: `class`, `package`, `interface`.

Below are all the Java language keywords:

- `abstract`
- `assert` (since Java 1.4)
- `boolean`
- `break`
- `byte`
- `case`
- `catch`
- `char`
- `class`
- `const` (not used)
- `continue`
- `default`
- `do`
- `double`
- `else`
- `enum` (since Java 5.0)
- `extends`
- `final`
- `finally`
- `float`
- `for`
- `goto` (not used)
- `if`
- `implements`
- `import`
- `instanceof`
- `int`
- `interface`
- `long`
- `native`
- `new`
- `package`
- `private`
- `protected`
- `public`
- `return`
- `short`
- `static`
- `strictfp` (since Java 1.2)
- `super`
- `switch`
- `synchronized`
- `this`
- `throw`
- `throws`
- `transient`
- `try`
- `void`
- `volatile`
- `while`

In addition, the identifiers `null`, `true`, and `false` denote literal values and may not be used to create identifiers.

abstract

`abstract` is a Java keyword. It can be applied to a class and methods. An *abstract* class cannot be directly instantiated. It must be placed before the variable type or the method return type. It is recommended to place it after the access modifier and after the `static` keyword. A non-abstract class is a *concrete* class. An abstract class cannot be `final`.

Only an abstract class can have abstract methods. An abstract method is only declared, not implemented:

Code listing 1: AbstractClass.java

```
public abstract class AbstractClass {
    // This method does not have a body; it is abstract.
    public abstract void abstractMethod();

    // This method does have a body; it is implemented in the abstract class and gives a default behavior.
    public void concreteMethod() {
        System.out.println("Already coded.");
    }
}
```

An abstract method cannot be `final`, `static` nor `native`. Either you instantiate a concrete sub-class, either you instantiate the abstract class by implementing its abstract methods alongside a new statement:

Code section 1: Abstract class use.

```
AbstractClass myInstance = new AbstractClass() {
    public void abstractMethod() {
        System.out.println("Implementation.");

```

```
    }
};
```

A private method cannot be **`abstract`**.

assert

`assert` is a Java keyword used to define an *assert statement*. An assert statement is used to declare an expected boolean condition in a program. If the program is running with assertions enabled, then the condition is checked at runtime. If the condition is false, the Java runtime system throws an `AssertionError`.

Assertions may be declared using the following syntax:

```
assert expression1 [: expression2];
```

`expression1` is a boolean that will throw the assertion if it is false. When it is thrown, the assertion error exception is created with the parameter `expression2` (if applicable).

An example:

```
assert list != null && list.size() > 0 : "list variable is null or empty";
Object value = list.get(0);
```

Assertions are usually used as a debugging aid. They should not be used instead of validating arguments to public methods, or in place of a more precise runtime error exception.

Assertions are enabled with the Java `-ea` or `-enableassertions` runtime option. See your Java environment documentation for additional options for controlling assertions.

boolean

`boolean` is a keyword which designates the **`boolean`** primitive type. There are only two possible **`boolean`** values: **`true`** and **`false`**. The default value for **`boolean`** fields is **`false`**.

The following is a declaration of a **`private`** **`boolean`** field named `initialized`, and its use in a method named `synchronizeConnection`.

Code section 1: Connection synchronization.

```
1  private boolean initialized = false;
2
3  public void synchronizeConnection() {
4     if (!initialized) {
5        connection = connect();
6        initialized = true;
7     }
8  }
```

The previous code only creates a connection once (at the first method call). Note that there is no automatic conversion between integer types (such as **`int`**) to **`boolean`** as is possible in some languages like C. Instead, one must use an equivalent expression such as `(i != 0)` which evaluates to **`true`** if `i` is not zero.

break

`break` is a Java keyword.

Jumps (breaks) out from a loop. Also used at **switch** statement.

For example:

```
for ( int i=0; i < maxLoopIter; i++ ) {
   System.out.println("Iter=" +i);
   if ( i == 5 ) {
      break;  // -- 5 iteration is enough --
   }
}
```

See also:

- Java Programming/Keywords/switch

byte

byte is a keyword which designates the 8 bit signed integer primitive type.

The `java.lang.Byte` class is the nominal wrapper class when you need to store a **byte** value but an object reference is required.

Syntax:

```
byte <variable-name> = <integer-value>;
```

For example:

```
byte b = 65;
```

or

```
byte b = 'A'
```

The number 65 is the code for 'A' in ASCII.

See also:

- Java Programming/Primitive Types

case

case is a Java keyword.

This is part of the **switch** statement, to find if the value passed to the switch statement matches a value followed by case.

For example:

```
int i = 3;
switch(i) {
case 1:
   System.out.println("The number is 1.");
   break;
case 2:
   System.out.println("The number is 2.");
   break;
```

```
case 3:
   System.out.println("The number is 3."); // this line will print
   break;
case 4:
   System.out.println("The number is 4.");
   break;
case 5:
   System.out.println("The number is 5.");
   break;
default:
  System.out.println("The number is not 1, 2, 3, 4, or 5.");
}
```

catch

`catch` is a keyword.

It's part of a `try` block. If an exception is thrown inside a try block, the exception will be compared to any of the catch part of the block. If the exception match with one of the exception in the catch part, the exception will be handled there.

For example:

```
try {
   //...
     throw new MyException_1();
   //...
} catch ( MyException_1 e ) {
   // --- Handle the Exception_1 here --
} catch ( MyException_2 e ) {
   // --- Handle the Exception_2 here --
}
```

See also:

- Java Programming/Keywords/try

char

`char` is a keyword. It defines a character primitive type. `char` can be created from character literals and numeric representation. Character literals consist of a single quote character (') (ASCII 39, hex 0x27), a single character, and a close quote ('), such as `'w'`. Instead of a character, you can also use unicode escape sequences, but there must be exactly one.

Syntax:

char ***variable name*** $_1$ = '***character*** $_1$';

Code section 1: Three examples.

```
1  char oneChar1 = 'A';
2  char oneChar2 = 65;
3  char oneChar3 = '\u0041';
4  System.out.println(oneChar1);
5  System.out.println(oneChar2);
6  System.out.println(oneChar3);
```

Output for Code section 1

```
A
A
A
```

65 is the numeric representation of character 'A' , or its ASCII code.

The nominal wrapper class is the `java.lang.Character` class when you need to store a `char` value but an object reference is required.

Code section 2: `char` wrapping.

```
char aCharPrimitiveType = 'A';
Character aCharacterObject = aCharPrimitiveType;
```

See also:

- Java Programming/Primitive Types

class

`class` is a Java keyword which begins the declaration and definition of a class.

The general syntax of a class declaration, using Extended Backus-Naur Form, is

```
class-declaration ::= [access-modifiers] class identifier
                      [extends-clause] [implements-clause]
                      class-body

extends-clause ::= extends class-name
implements-clause ::= implements interface-names
interface-names ::= interface-name [, interface-names]
class-body ::= { [member-declarations] }
member-declarations = member-declaration [member-declarations]
member-declaration = field-declaration
                   | initializer
                   | constructor
                   | method-declaration
                   | class-declaration
```

The `extends` word is optional. If omitted, the class extends the `Object` class, as all Java classes inherit from it.

See also:

- Java Programming/Keywords/new

const

`const` is a **reserved keyword**, presently not being used.

In other programming languages, such as C, const is often used to declare a constant. However, in Java, `final` is used instead.

continue

`continue` is a Java keyword. It skips the remainder of the loop and continues with the next iteration.

For example:

```
int maxLoopIter = 7;

for (int i = 0; i < maxLoopIter; i++ ) {
   if (i == 5) {
      continue;  // -- 5 iteration is skipped --
   }
   System.println("Iteration = " + i);
}
```

results in

```
0
1
2
3
4
6
7
```

See also

- Java Programming/Statements

default

`default` is a Java keyword.

This is an optional part of the **`switch`** statement, which only executes if none of the above cases are matched.

See also:

- Java Programming/Keywords/switch

do

`do` is a Java keyword.

It starts a do-while looping block. The do-while loop is functionally similar to the while loop, except the condition is evaluated *after* the statements execute

Syntax:

```
do {
    //statements;
} while (condition);
```

For example:

```
do {
    i++;
} while ( i < maxLoopIter );
```

See also:

- Java Programming/Statements
- Java Programming/Keywords/for
- Java Programming/Keywords/while

double

`double` is a keyword which designates the 64 bit float primitive type.

The `java.lang.Double` class is the nominal wrapper class when you need to store a **`double`** value but an object reference is required.

Syntax:

```
double <variable-name> = <float-value>;
```

For example:

```
double d = 65.55;
```

See also:

- Java Programming/Primitive Types

else

else is a Java keyword. It is an optional part of a branching statement. It starts the 'false' statement block.

The general syntax of a **if**, using Extended Backus-Naur Form, is

```
branching-statement ::= if condition-clause
                                single-statement | block-statement
                        [ else
                                single-statement | block-statement ]

condition-clause    ::= ( Boolean Expression )
single-statement    ::= Statement
block-statement     ::= { Statement [ Statement ] }
```

For example:

```
if ( expression ) {
   System.out.println("'True' statement block");
} else {
   System.out.println("'False' statement block");
}
```

See also:

- Java Programming/Keywords/if

enum

```
/** Grades of courses */
  enum Grade { A, B, C, D, F };
  // ...
  private Grade gradeA = Grade.A;
```

This enumeration constant then can be passed in to methods:

```
student.assignGrade(gradeA);
  /**
   * Assigns the grade for this course to the student
   * @param GRADE  Grade to be assigned
   */
  public void assignGrade(final Grade GRADE) {
    grade = GRADE;
  }
```

An enumeration may also have parameters:

```
public enum DayOfWeek {
  /** Enumeration constants */
  MONDAY(1), TUESDAY(2), WEDNESDAY(3), THURSDAY(4), FRIDAY(5), SATURDAY(6), SUNDAY(0);

  /** Code for the days of the week */
  private byte dayCode = 0;

  /**
```

```
    * Private constructor
    * @param VALUE  Value that stands for a day of the week.
    */
  private DayOfWeek(final byte VALUE) {
    dayCode = java.lang.Math.abs(VALUE%7);
  }

  /**
   * Gets the day code
   * @return  The day code
   */
  public byte getDayCode() {
    return dayCode;
  }
}
```

It is also possible to let an enumeration implement interfaces other than `java.lang.Comparable` and `java.io.Serializable`, which are already implicitly implemented by each enumeration:

```
public enum DayOfWeek implements Runnable {
  MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;

  /**
   * Run method prints all elements
   */
  public void run() {
    System.out.println("name() = " + name() +
      ", toString() = \"" + toString() + "\"");
  }
}
```

extends

`extends` is a Java keyword.

Used in **`class`** and **`interface`** definition to declare the class or interface that is to be extended.

Syntax:

```
public class MyClass extends SuperClass
{
  //...
}

public interface MyInterface extends SuperInterface
{
  //...
}
```

In Java 1.5 and later, the "extends" keyword is also used to specify an upper bound on a type parameter in Generics.

```
class Foo<T extends Number> { /*...*/ }
```

See also:

- Java Programming/Creating Objects
- Java Programming/Keywords/class

final

`final` is a keyword. Beware! It has distinct meanings depending whether it is used for a class, a method, or for a variable. It must be placed before the variable type or the method return type. It is recommended to place it after the access modifier and after the **`static`** keyword.

Code section 1: Keyword order.

```
1  private static final long serialVersionUID = -5437975414336623381L;
```

For a variable

The `final` keyword only allows a single assignment for the variable. That is to say, once the variable has been assigned, its value is in read-only. If the variable is a primitive type, its value will no longer change. If it is an object, only its reference will no longer change. Keep in mind that its value can still be changed.

Code section 2: Forbidden double assignment.

```
1  final int a = 1;
2  a = 2;
```

Code section 3: Only modify the value of the object.

```
1  final ArrayList list = new ArrayList();
2  System.out.println(list.size());
3  list.add("One item");
4  System.out.println(list.size());
```

Console for Code section 3

```
0
1
```

A final variable is often used for universal constants, such as pi:

Code section 4: Pi constant.

```
1  static final double PI = 3.1415926;
```

The `final` keyword can also be used for method parameters:

Code section 5: Final method parameter.

```
1  public int method(final int inputInteger) {
2     int outputInteger = inputInteger + 1;
3     return outputInteger;
4  }
```

It is useful for methods that use side effects to update some objects. Such methods modify the content of an object passed in parameter. The method caller will receive the object update. This will fail if the object parameter has been reassigned in the method. Another object will be updated instead. Final method parameter can also be used to keep the code clean.

The `final` keyword is similar to `const` in other languages and the `readonly` keyword in C#. A final variable cannot be **`volatile`**.

For a class

The `final` keyword forbids the creation of a subclass. It is the case of the `Integer` or `String` class.

Code listing 1: SealedClass.java

```
1  public final class SealedClass {
2    public static void main(String[] args) {
3    }
4  }
```

A final class cannot be **`abstract`**. The `final` keyword is similar to `sealed` keyword in C#.

For a method

The `final` keyword forbids to overwrite the method in a subclass. It is useless if the class is already final and a private method is implicitly **`final`**. A final method cannot be **`abstract`**.

Code listing 2: NoOverwriting.java

```
1  public class NoOverwriting {
2    public final void sealedMethod() {
3    }
4  }
```

Interest

The `final` keyword is mostly used to guarantee a good usage of the code. For instance (non-`static`) methods, this allows the compiler to expand the method (similar to an inline function) if the method is small enough. Sometimes it is required to use it. For instance, a nested class can only access the members of the top-level class if they are final.

See also Access Modifiers.

finally

`finally` is a keyword which is an optional ending part of the **`try`** block.

Code section 1: `try` block.

```
1  try {
2    // ...
3  } catch (MyException1 e) {
4    // Handle the Exception1 here
5  } catch (MyException2 e) {
6    // Handle the Exception2 here
7  } finally {
8    // This will always be executed no matter what happens
9  }
```

The code inside the finally block will always be executed. This is also true for cases when there is an exception or even executed **`return`** statement in the try block.

Three things can happen in a try block. First, no exception is thrown:

Code section 2: No exception is thrown.

```
1  System.out.println("Before the try block");
2  try {
3    System.out.println("Inside the try block");
4  } catch (MyException1 e) {
5    System.out.println("Handle the Exception1");
6  } catch (MyException2 e) {
7    System.out.println("Handle the Exception2");
8  } finally {
9    System.out.println("Execute the finally block");
10 }
11 System.out.println("Continue");
```

Console for Code section 2

```
Before the try block
Inside the try block
Execute the finally block
Continue
```

You can see that we have passed in the **`try`** block, then we have executed the **`finally`** block and we have continued the execution. Now, a caught exception is thrown:

Code section 3: A caught exception is thrown.

Console for Code section 3

```
System.out.println("Before the try block");
try {
  System.out.println("Enter inside the try block");
  throw new MyException1();
  System.out.println("Terminate the try block");
} catch (MyException1 e) {
  System.out.println("Handle the Exception1");
} catch (MyException2 e) {
  System.out.println("Handle the Exception2");
} finally {
  System.out.println("Execute the finally block");
}
System.out.println("Continue");
```

```
Before the try block
Enter inside the try block
Handle the Exception1
Execute the finally block
Continue
```

We have passed in the **`try`** block until where the exception occurred, then we have executed the matching **`catch`** block, the **`finally`** block and we have continued the execution. Now, an uncaught exception is thrown:

Code section 4: An uncaught exception is thrown.

```
System.out.println("Before the try block");
try {
  System.out.println("Enter inside the try block");
  throw new Exception();
  System.out.println("Terminate the try block");
} catch (MyException1 e) {
  System.out.println("Handle the Exception1");
} catch (MyException2 e) {
  System.out.println("Handle the Exception2");
} finally {
  System.out.println("Execute the finally block");
}
System.out.println("Continue");
```

Console for Code section 4

```
Before the try block
Enter inside the try block
Execute the finally block
```

We have passed in the **`try`** block until where the exception occurred and we have executed the **`finally`** block. **NO CODE** after the try-catch block has been executed. If there is an exception that happens before the try-catch block, the **`finally`** block is not executed.

If **`return`** statement is used inside finally, it overrides the return statement in the try-catch block. For instance, the construct

Code section 5: Return statement.

```
try {
  return 11;
}  finally {
  return 12;
}
```

will return 12, not 11. Professional code almost never contains statements that alter execution order (like **`return`**, **`break`**, **`continue`**) inside the finally block, as such code is more difficult to read and maintain.

float

`float` is a keyword which designates the 32 bit float primitive type.

The `java.lang.Float` class is the nominal wrapper class when you need to store a **`float`** value but an object reference is required.

Syntax:

```
float <variable-name> = <float-value>;
```

For example:

```
float price = 49.95;
```

See also:

- Java Programming/Primitive Types

for

for is a Java keyword.

It starts a looping block.

The general syntax of a **for**, using Extended Backus-Naur Form, is

```
for-looping-statement ::= for condition-clause
                                  single-statement | block-statement

condition-clause    ::= ( before-statement;  Boolean Expression ; after-statement )
single-statement    ::= Statement
block-statement     ::= { Statement [ Statement ] }
```

For example:

```
for ( int i=0; i < maxLoopIter; i++ ) {
    System.println("Iter: " +i);
}
```

See also:

- Java Programming/Keywords/while
- Java Programming/Keywords/do

goto

goto is a **reserved keyword**, presently not being used.

if

if is a Java keyword. It starts a branching statement.

The general syntax of a **if**, using Extended Backus-Naur Form, is

```
branching-statement ::= if condition-clause
                                  single-statement | block-statement
                       [ else
                                  single-statement | block-statement ]

condition-clause    ::= ( Boolean Expression )
single-statement    ::= Statement
block-statement     ::= { Statement [ Statements ] }
```

For example:

```
if ( boolean Expression )
{
   System.out.println("'True' statement block");
}
else
{
   System.out.println("'False' statement block");
}
```

See also:

- Java Programming/Keywords/else

implements

implements is a Java keyword.

Used in **class** definition to declare the Interfaces that are to be implemented by the class.

Syntax:

```
public class MyClass implements MyInterface1, MyInterface2
 {
   ...
 }
```

See also:

- Java Programming/Creating Objects
- Java Programming/Keywords/class
- Java Programming/Keywords/interface

import

import is a Java keyword.

It declares a Java class to use in the code below the import statement. Once a Java class is declared, then the class name can be used in the code without specifying the package the class belongs to.

Use the '*' character to declare all the classes belonging to the package.

Syntax:

```
import package.JavaClass;
import package.*;
```

The static import construct allows unqualified access to static members without inheriting from the type containing the static members:

```
    import static java.lang.Math.PI;
```

Once the static members have been imported, they may be used without qualification:

```
    double r = cos(PI * theta);
```

Caveat: use static import very sparingly to avoid polluting the program's namespace!

See also:

- Java Programming/Packages

instanceof

instanceof is a keyword.

It checks if an object reference is an instance of a type, and returns a boolean value;

The <object-reference> **`instanceof`** `Object` will return true for all non-null object references, since all Java objects are inherited from `Object`. **`instanceof`** will always return **`false`** if <object-reference> is **`null`**.

Syntax:

```
<object-reference> instanceof TypeName
```

For example:

```
class Fruit
 {
  //...
 }
 class Apple extends Fruit
 {
  //...
 }
 class Orange extends Fruit
 {
  //...
 }
 public class Test
 {
    public static void main(String[] args)
    {
       Collection<Object> coll = new ArrayList<Object>();

       Apple app1 = new Apple();
       Apple app2 = new Apple();
       coll.add(app1);
       coll.add(app2);

       Orange or1 = new Orange();
       Orange or2 = new Orange();
       coll.add(or1);
       coll.add(or2);

       printColl(coll);
    }

    private static String printColl( Collection<?> coll )
    {
       for (Object obj : coll)
       {
          if ( obj instanceof Object )
          {
             System.out.print("It is a Java Object and");
          }
          if ( obj instanceof Fruit )
          {
             System.out.print("It is a Fruit and");
          }
          if ( obj instanceof Apple )
          {
             System.out.println("it is an Apple");
          }
          if ( obj instanceof Orange )
          {
             System.out.println("it is an Orange");
          }
       }
    }
 }
```

Run the program:

```
java Test
```

The output:

```
"It is a Java Object and It is a Fruit and it is an Apple"
"It is a Java Object and It is a Fruit and it is an Apple"
"It is a Java Object and It is a Fruit and it is an Orange"
"It is a Java Object and It is a Fruit and it is an Orange"
```

Note that the **instanceof** operator can also be applied to interfaces. For example, if the example above was enhanced with the interface

```
interface Edible
{
 //...
}
```

and the classes modified such that they implemented this interface

```
class Orange extends Fruit implements Edible
 {
  ...
 }
```

we could ask if our object were edible.

```
if ( obj instanceof Edible )
 {
   System.out.println("it is edible");
 }
```

int

int is a keyword which designates the 32 bit signed integer primitive type.

The `java.lang.Integer` class is the nominal wrapper class when you need to store an **int** value but an object reference is required.

Syntax:

```
int <variable-name> = <integer-value>;
```

For example:

```
int i = 65;
```

See also:

- Java Programming/Primitive Types

interface

interface is a Java keyword. It starts the declaration of a Java Interface.

For example:

```
public interface SampleInterface
{
   public void method1();
   //...
}
```

See also:

- Java Programming/Keywords/new

long

long is a keyword which designates the 64 bit signed integer primitive type.

The `java.lang.Long` class is the nominal wrapper class when you need to store a **long** value but an object reference is required.

Syntax:

```
long <variable-name> = <integer-value>;
```

For example:

```
long timestamp = 1269898201;
```

See also:

- Java Programming/Primitive Types

native

native is a java keyword. It marks a method, that it will be implemented in other languages, not in Java. The method is declared without a body and cannot be **abstract**. It works together with JNI (Java Native Interface).

Syntax:

```
[public|protected|private] native method();
```

Native methods were used in the past to write performance critical sections but with java getting faster this is now less common. Native methods are currently needed when

- You need to call from java a library, written in another language.
- You need to access system or hardware resources that are only reachable from the other language (typically C). Actually, many system functions that interact with real computer (disk and network IO, for instance) can only do this because they call native code.

To complete writing native method, you need to process your class with `javah` tool that will generate a header code in C. You then need to provide implementation of the header code, produce dynamically loadable library (`.so` under Linux, `.dll` under Windows) and load it (in the simplest case with `System.load(library_file_name)` . The code completion is trivial if only primitive types like integers are passed but gets more complex if it is needed to exchange strings or objects from the C code. In general, everything can be on C level, including creation of the new objects and calling back methods, written in java.

To call the code in some other language (including C++), you need to write a bridge from C to that language. This is usually trivial as most of languages are callable from C.

See also

- [3] (http://java.sun.com/developer/onlineTraining/Programming/JDCBook/jni.html) - JNI programming tutorial.
- [4] (http://java.sun.com/j2se/1.3/docs/guide/jni/spec/jniTOC.doc.html) - JNI specification.

new

new is a Java keyword. It creates a Java object and allocates memory for it on the heap. new is also used for array creation, as arrays are also objects.

Syntax:

```
<JavaType> <variable> = new <JavaObject>();
```

For example:

```
LinkedList list = new LinkedList();
int[] intArray = new int[10];
String[][] stringMatrix = new String[5][10];
```

See also:

- Java Programming/Creating Objects

package

package is a Java keyword. It declares a 'name space' for the Java class. It must be put at the top of the Java file, it should be the first Java statement line.

To ensure that the package name will be unique across vendors, usually the company url is used starting in backword.

Syntax:

```
package package;
```

For example:

```
package com.mycompany.myapplication.mymodule;
```

See also:

- Java Programming/Packages
- Java Programming/Keywords/import

private

private is a Java keyword which declares a member's access as private. That is, the member is only visible within the class, not from any other class (including subclasses). The visibility of **private** members extends to nested classes.

Please note: Because access modifiers are not handled at instance level but at class level, private members of an object are visible from other instances of the same class!

Syntax:

```
private void method();
```

See also:

- Java Programming/Access Modifiers

protected

protected is a Java keyword.

This keyword is an access modifier, used before a method or other class member to signify that the method or variable can only be accessed by elements residing in its own class or classes in the same package (as it would be for the default visibility level) but moreover from subclasses of its own class, including subclasses in foreign packages (if the access is made on an expression, whose type is the type of this subclass).

Syntax:

```
protected <returnType> <methodName>(<parameters>);
```

For example:

```
protected int getAge();

protected void setYearOfBirth(int year);
```

See also:

- Java Programming/Scope#Access modifiers

public

public is a Java keyword which declares a member's access as public. Public members are visible to all other classes. This means that any other class can access a **public** field or method. Further, other classes can modify **public** fields unless the field is declared as **final**.

A best practice is to give fields **private** access and reserve **public** access to only the set of methods and **final** fields that define the class' public constants. This helps with encapsulation and information hiding, since it allows you to change the implementation of a class without affecting the consumers who use only the public API of the class.

Below is an example of an immutable **public** class named `Length` which maintains **private** instance fields named `units` and `magnitude` but provides a **public** constructor and two **public** accessor methods.

Code listing: Length.java

```
1  package org.wikibooks.java;
2
3  public class Length {
4     private double magnitude;
5     private String units;
6
7     public Length(double magnitude, String units) {
8        if ((units == null) || (units.trim().length() == 0)) {
9            throw new IllegalArgumentException("non-null, non-empty units required.");
10       }
11
12       this.magnitude = magnitude;
13       this.units = units;
14    }
15
16    public double getMagnitude() {
17       return magnitude;
18    }
19
20    public String getUnits() {
21       return units;
22    }
23 }
```

return

return is a Java keyword.

Returns a primitive value, or an object reference, or nothing(void). It does not return object values, only object references.

Syntax:

```
return variable;  // --- Returns variable
or
return;           // --- Returns nothing
```

short

short is a keyword. It defines a 16 bit signed integer primitive type.

Syntax:

```
short <variable-name> = <integer-value>;
```

For example:

```
short age = 65;
```

See also:

- Java Programming/Primitive Types

static

static is a Java keyword. It can be applied to a field, a method or an inner class. A static field, method or class has a single instance for the whole class that defines it, even if there is no instance of this class in the program. For instance, a Java entry point (`main()`) has to be static. A static method cannot be **abstract**. It must be placed before the variable type or the method return type. It is recommended to place it after the access modifier and before the **final** keyword:

Code section 1: Static field and method.

```
1  public static final double PI = 3.1415926535;
2
3  public static void main(final String[] arguments) {
4     //…
5  }
```

The static items can be called on an instantiated object or directly on the class:

Code section 2: Static item calls.

```
1  double aNumber = MyClass.PI;
2  MyClass.main(new String[0]);
```

Static methods cannot call nonstatic methods. The **this** current object reference is also not available in static methods.

Interest

- Static variables can be used as data sharing amongst objects of the same class. For example to implement a counter that stores the number of objects created at a given time can be defined as so:

Code listing 1: CountedObject.java

```
1  public CountedObject {
2     private static int counter;
3     …
4     public AClass() {
5        …
6        counter++;
7     }
8     …
9     public int getNumberOfObjectsCreated() {
10       return counter;
11    }
12 }
```

The `counter` variable is incremented each time an object is created.

Public static variable should not be used, as these become **global** variables that can be accessed from everywhere in the program. Global constants can be used, however. See below:

Code section 3: Constant definition.

```
1  public static final String CONSTANT_VAR = "Const";
```

- Static methods can be used for utility functions or for functions that do not belong to any particular object. For example:

Code listing 2: ArithmeticToolbox.java

```
1  public ArithmeticToolbox {
2     …
3     public static int addTwoNumbers(final int firstNumber, final int secondNumber) {
4          return firstNumber + secondNumber;
5     }
6  }
```

See also Static methods

strictfp

strictfp is a java keyword, since Java 1.2 .

It makes sure that floating point calculations result precisely the same regardless of the underlying operating system and hardware platform, even if more precision could be obtained. This is compatible with the earlier version of Java 1.1 . If you need that use it.

Syntax for classes:

```
public strictfp class MyClass
{
  //...
}
```

Syntax for methods:

```
public strictfp void method()
{
  ...
}
```

See also:

- http://en.wikipedia.org/wiki/Strictfp

super

super is a keyword.

- It is used inside a sub-class method definition to call a method defined in the super class. Private methods of the super-class cannot be called. Only public and protected methods can be called by the **super** keyword.
- It is also used by class constructors to invoke constructors of its parent class.
- Super keyword are not used in static Method.

Syntax:

```
super.<method-name>([zero or more arguments]);
```

or:

```
super([zero or more arguments]);
```

For example:

Code listing 1: SuperClass.java

```
1  public class SuperClass {
2     public void printHello() {
3        System.out.println("Hello from SuperClass");
4        return;
5     }
6  }
```

Code listing 2: SubClass.java

```
1  public class SubClass extends SuperClass {
2     public void printHello() {
3        super.printHello();
4        System.out.println("Hello from SubClass");
5        return;
6     }
7     public static main(String[] args) {
8        SubClass obj = new SubClass();
9        obj.printHello();
10    }
11 }
```

Running the above program:

Command for Code listing 2

```
$Java SubClass
```

Output of Code listing 2

```
Hello from SuperClass
Hello from SubClass
```

In Java 1.5 and later, the "super" keyword is also used to specify a lower bound on a wildcard type parameter in Generics.

Code section 1: A lower bound on a wildcard type parameter.

```
public void sort(Comparator<? super T> comp) {
  ...
}
```

See also:

- **`extends`**

switch

`switch` is a Java keyword.

It is a branching operation, based on a number. The 'number' must be either **`char`**, **`byte`**, **`short`**, or **`int`** primitive type.

Syntax:

```
switch ( <integer-var> )
{
   case <label1>: <statements>;
   case <label2>: <statements>;
   ...
   case <labeln>: <statements>;
   default: <statements>;
}
```

When the <integer-var> value match one of the <label>, then: The statements after the matched label will be executed including the following label's statements, until the end of the **`switch`** block, or until a **`break`** keyword is reached.

For example:

```
int var = 3;
switch ( var )
{
   case 1:
      System.out.println( "Case: 1" );
      System.out.println( "Execute until break" );
      break;
   case 2:
      System.out.println( "Case: 2" );
      System.out.println( "Execute until break" );
      break;
   case 3:
      System.out.println( "Case: 3" );
      System.out.println( "Execute until break" );
      break;
   case 4:
      System.out.println( "Case: 4" );
      System.out.println( "Execute until break" );
      break;
   default:
      System.out.println( "Case: default" );
      System.out.println( "Execute until break" );
      break;
}
```

The output from the above code is:

```
Case: 3
Execute until break
```

The same code can be written with if-else blocks":

```
int var = 3;
if ( var == 1 ) {
```

```
    System.out.println( "Case: 1" );
    System.out.println( "Execute until break" );
} else if ( var == 2 ) {
    System.out.println( "Case: 2" );
    System.out.println( "Execute until break" );
} else if ( var == 3 ) {
    System.out.println( "Case: 3" );
    System.out.println( "Execute until break" );
} else if ( var == 4 ) {
    System.out.println( "Case: 4" );
    System.out.println( "Execute until break" );
} else {
    // -- This is the default part --
    System.out.println( "Case: default" );
    System.out.println( "Execute until break" );
}
```

See also:

- Java Programming/Keywords/if

synchronized

synchronized is a keyword.

It marks a *critical section*. A *critical section* is where one and only one thread is executing. So to enter into the marked code the threads are *synchronized*, only one can enter, the others have to wait. For more information see Synchronizing Threads Methods or [5] (http://java.sun.com/docs/books/tutorial/essential/concurrency/syncmeth.html).

The **synchronized** keyword can be used in two ways:

- Create a **synchronized** block
- Mark a method **synchronized**

A **synchronized** block is marked as:

Code section 1: Synchronized block.

```
synchronized(<object_reference>) {
    // Thread.currentThread() has a lock on object_reference. All other threads trying to access it will
    // be blocked until the current thread releases the lock.
}
```

The syntax to mark a method **synchronized** is:

Code section 2: Synchronized method.

```
public synchronized void method() {
    // Thread.currentThread() has a lock on this object, i.e. a synchronized method is the same as
    // calling { synchronized(this) {…} }.
}
```

The synchronization is always associated to an object. If the method is static, the associated object is the class. If the method is non-static, the associated object is the instance. While it is allowed to declare an **abstract** method as **synchronized**, it is meaningless to do so since synchronization is an aspect of the implementation, not the declaration, and abstract methods do not have an implementation.

Singleton example

As an example, we can show a thread-safe version of a singleton:

Code listing 1: Singleton.java

```java
/**
 * The singleton class that can be instantiated only once with lazy instantiation
 */
public class Singleton {
    /** Static class instance */
    private volatile static Singleton instance = null;

    /**
     * Standard private constructor
     */
    private Singleton() {
        // Some initialisation
    }

    /**
     * Getter of the singleton instance
     * @return The only instance
     */
    public static Singleton getInstance() {
        if (instance == null) {
            // If the instance does not exist, go in time-consuming
            // section:
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }

        return instance;
    }
}
```

this

`this` is a Java keyword. It contains the current object reference.

1. Solves ambiguity between instance variables and parameters .
2. Used to pass current object as a parameter to another method .

Syntax:

```
this.method();
or
this.variable;
```

Example #1 for case 1:

```java
public class MyClass
 {
    //...
    private String value;
    //...
    public void setMemberVar( String value )
    {
        this.value= value;
    }
 }
```

Example #2 for case 1:

```java
public class MyClass
 {
    MyClass(int a, int b) {
        System.out.println("int a: " + a);
        System.out.println("int b: " + b);
    }
    MyClass(int a) {
        this(a, 0);
    }
    //...
    public static void main(String[] args) {
        new MyClass(1, 2);
```

```
        new MyClass(5);
    }
}
```

throw

throw is a keyword; it 'throws' an exception. In a throw statement, the three types of objects that can be thrown are: `Exception`, `java:Throwable`, and `java:Error`

Syntax:

```
throw <Exception Ref>;
```

For example:

```
public Customer findCustomer( String name ) throws '''CustomerNotFoundException'''
 {
    Customer custRet = null;

    Iterator iter = _customerList.iterator();
    while ( iter.hasNext() )
    {
        Customer cust = (Customer) iter.next();
        if ( cust.getName().equals( name ) )
        {
           // --- Customer find --
           custRet = cust;
           break;
        }
     }
     if ( custRet == null )
     {
        // --- Customer not found ---
        throw new '''CustomerNotFoundException'''( "Customer "+ name + " was not found" );
     }

    return custRet
  }
```

See also

- Java Programming/Keywords/throws

throws

throws is a Java keyword. It is used in a method definition to declare the Exceptions to be thrown by the method.

Syntax:

```
public myMethod() throws MyException1, MyException2
{MyException1
  ...
}
```

Example:

```
class MyDefinedException extends Exception
 {
  public MyDefinedException(String str)
  {
     super(str);
  }
 }

 public class MyClass
```

```
{
   public static void showMyName(String str) throws MyDefinedException
   {
         if(str.equals("What is your Name?"))
               throw new MyDefinedException("My name is Blah Blah");
   }
   public static void main(String a[])
   {
      try
      {
         showMyName("What is your Name?");
      }
      catch(MyDefinedException mde)
      {
         mde.printStackTrace();
      }
    }
}
```

transient

`transient` is a Java keyword which marks a member variable not to be serialized when it is persisted to streams of bytes. When an object is transferred through the network, the object needs to be 'serialized'. Serialization converts the object state to serial bytes. Those bytes are sent over the network and the object is recreated from those bytes. Member variables marked by the java **`transient`** keyword are not transferred; they are lost intentionally.

Syntax:

```
private transient <member-variable>;
or
transient private <member-variable>;
```

For example:

```
public class Foo implements Serializable
 {
   private String saveMe;
   private transient String dontSaveMe;
   private transient String password;
   //...
 }
```

See also:

- Java language specification reference: jls (https://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.3.1.3)
- Serializable Interface. Serializable (https://en.wikipedia.org/wiki/Serialization#Java)

try

`try` is a keyword.

It starts a try block. If an Exception is thrown inside a try block, the Exception will be compared to any of the catch part of the block. If the Exception matches with one of the Exceptions in the catch part, the exception will be handled there.

Three things can happen in a try block:

- No exception is thrown:
  - the code in the try block
  - plus the code in the finally block will be executed
  - plus the code after the try-catch block is executed
- An exception is thrown and a match is found among the catch blocks:
  - the code in the try block until the exception occurred is executed

- plus the matched catch block is executed
- plus the finally block is executed
- plus the code after the try-catch block is executed

- An exception is thrown and no match found among the catch blocks:
  - the code in the try block until the exception occurred is executed
  - plus the finally block is executed
  - **NO CODE** after the try-catch block is executed

For example:

```
public void method() throws NoMatchedException
 {
   try {
     //...
     throw new '''MyException_1'''();
     //...
   } catch ( MyException_1 e ) {
     // --- '''Handle the Exception_1 here''' --
   } catch ( MyException_2 e ) {
     // --- Handle the Exception_2 here --
   } finally {
     // --- This will always be executed no matter what --
   }
   // --- Code after the try-catch block
 }
```

How the catch-blocks are evaluated see Catching Rule

See also:

- Java Programming/Keywords/catch
- Java Programming/Keywords/finally
- Java Programming/Throwing and Catching Exceptions#Catching Rule

void

`void` is a Java keyword.

Used at method declaration and definition to specify that the method does not return any type, the method returns **`void`**. It is not a type and there is no void references/pointers as in C/C++.

For example:

```
public void method()
 {
   //...
   return;   // -- In this case the return is optional
//and not necessary to use public but some changes will be there
 }
```

See also:

- Java Programming/Keywords/return

volatile

`volatile` is a keyword.

When member variables are marked with this keyword, it changes the runtime behavior in a way that is noticeable when multiple threads access these variables. Without the volatile keyword, one thread could observe another thread update member variables in an order that is not consistent with what is specified in sourcecode. Unlike the synchronized keyword, concurrent access to a volatile field is allowed.

Syntax:

```
private volatile <member-variable>;
or
volatile private <member-variable>;
```

For example:

```
private volatile changingVar;
```

See also:

- Java Programming/Keywords/synchronized

while

while is a Java keyword.

It starts a looping block.

The general syntax of a **while**, using Extended Backus-Naur Form, is

```
while-looping-statement ::= while condition-clause
                                  single-statement | block-statement

condition-clause    ::= ( Boolean Expression )
single-statement    ::= Statement
block-statement     ::= { Statement [ Statements ] }
```

For example:

```
while ( i < maxLoopIter )
 {
    System.out.println("Iter=" +i++);
 }
```

See also:

- Java Programming/Statements
- Java Programming/Keywords/for
- Java Programming/Keywords/do

Packages

If your application becomes quite big, you may have many classes. Although you can browse them in their alphabetic order, it becomes confusing. So your application classes can be sorted into *packages*.

A package is a name space that mainly contains classes and interfaces. For instance, the standard class `ArrayList` is in the package `java.util`. For this class, `java.util.ArrayList` is called its *fully qualified name* because this syntax has no ambiguity. Classes in different packages can have the same name. For example, you have the two classes `java.util.Date` and `java.sql.Date` which are not the same. If no package is declared in a class, its package is the default package.

Package declaration

In a class, a package is declared at the top of the source code using the keyword **`package`**:

Code listing 3.14: BusinessClass.java

```
1  package business;
2
3  public class BusinessClass {
4  }
```

If your class is declared in a package, say `business`, your class must be placed in a subfolder called `business` from the root of your application folder. This is how the compiler and the class loader find the Java files on the file system. You can declare your class in a subpackage, say `engine`. So the full package is `business.engine` and the class must be placed in a subsubfolder called `engine` in the subfolder `business` (not in a folder called `business.engine`).

Import and class usage

The simpliest way to use a class declared in a package is to prefix the class name with its package:

Code section 3.88: Package declaration.

```
1  business.BusinessClass myBusinessClass = new business.BusinessClass();
```

If you are using the class from a class in the same package, you don't have to specify the package. If another class with the same name exists in another package, it will use the local class.

The syntax above is a bit verbose. You can import the class by using the **`import`** Java keyword at the top of the file and then only specify its name:

Code listing 3.15: MyClass.java

```
1  import business.BusinessClass;
2
3  public class MyClass {
4    public static void main(String[] args) {
5      BusinessClass myBusinessClass = new BusinessClass();
6    }
7  }
```

Note that you can't import two classes with the same name in two different packages.

The classes `Integer` and `String` belongs to the package `java.lang` but they don't need to be imported as the `java.lang` package is implicitly imported in all classes.

Wildcard imports

It is possible to import an entire package, using an asterisk:

Code section 3.89: Wildcard imports.

```
1  import javax.swing.*;
```

While it may seem convenient, it may cause problems if you make a typographical error. For example, if you use the above import to use JFrame, but then type `JFraim frame = new JFraim();`, the Java compiler will report an error similar to "Cannot find symbol: JFraim". Even though it seems as if it was imported, the compiler is giving the error report at the first mention of JFraim, which is half-way through your code, instead of the point where you imported JFrame along with everything else in javax.swing.

If you change this to `import javax.swing.JFraim;` the error will be at the import instead of within your code.

Furthermore, if you `import javax.swing.*;` and `import java.util.*;`, and javax.swing.Queue is later added in a future version of Java, your code that uses Queue (java.util) will fail to compile. This particular example is fairly unlikely, but if you are working with non-Oracle libraries, it may be more likely to happen.

Package convention

A package name should start with a lower character. This eases to distinguish a package from a class name. In some operating systems, the directory names are not case sensitive. So package names should be lowercase.

The Java package needs to be unique across Vendors to avoid name collisions. For that reason Vendors usually use their domain name in reverse order. That is guaranteed to be unique. For example a company called *Your Company Inc.*, would use a package name something like this: `com.yourcompany.yourapplicationname.yourmodule.YourClass`.

Importing packages from .jar files

If you are importing library packages and classes that reside in a `.jar` file, you must ensure that the file is in the current classpath (both at compile- and execution-time). Apart from this requirement, importing these packages and classes is the same as if they were in their full, expanded, directory structure.

Javac

For example, to compile and run a class from a project's top directory (that contains the two directories `/source` and `/libraries`) you could use the following command:

Compilation

```
$ javac -classpath libraries/lib.jar source/MainClass.java
```

And then to run it, similarly:

Execution

```
$ java -classpath libraries/lib.jar source/MainClass
```

The above is simplified, and demands that `MainClass` be in the default package, or a package called `source`, which isn't very desirable.

BlueJ

With BlueJ just click on *Tools, Preferences, Libraries*, and add the .jar one by one.

Class loading/package

The runtime identity of a class in Java is defined by the fully qualified class name and its defining class loader. This means that the same class, loaded by two different class loaders, is seen by the Virtual Machine as two completely different types.

Arrays

An **array** is similar to a table of objects or primitive types, keyed by index. You may have noticed the strange parameter of the default `main()` method (`String[] args`) since the beginning of the book. It is an array. Let's handle this parameter:

Code listing 3.15: The default array parameter.

```
1  public class ArrayExample {
2    public static void main(String[] args) {
3      for (int i = 0; i < args.length; ++i) {
4        System.out.println("Argument #" + (i + 1) + ": " + args[i]);
5      }
6    }
7  }
```

Console for Code listing 3.15

```
$ java ArrayExample This is a test
Argument #1 This
Argument #2 is
Argument #3 a
Argument #4 test
```

In the code listing 3.15, the array is `args`. It is an array of `String` objects (here those objects are the words that have been typed by the user at the program launching). At line 4, One contained object is accessed using its index in the array. You can see that its value is printed on the standard output. Note that the strings have been put in the array with the right order.

Fundamentals

In Java, an array is an object. This object has a given type for the contained primitive types or objects (`int`, `char`, `String`, ...). An array can be declared in several ways:

Code section 3.52: Array declarations.

```
1  int[] array1 = null;
2  int array2[] = null;
```

Those syntaxes are identical but the first one is recommended. It can also be instantiated in several ways:

Code section 3.53: Array instantiations.

```
1  array1 = new int[10];
2  int[] array0 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; //this only works in the declaration
3  array1 = new int[]{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

At line 1, we instantiate an array of 10 items that get the default value (which is 0 for `int`). At lines 2 and 3, we instantiate arrays of 10 given items. It will each be given an index according to its order. We can know the size of the array using the `length` attribute:

Code section 3.54: The array size.

```
1  int nbItems = 10;
2  Object[] array3 = new Object[nbItems];
3  System.out.println(array3.length);
```

Output for Code section 3.54

```
10
```

Arrays are allocated at runtime, so the specified size in an array creation expression may be a variable (rather than a constant expression as in C). However, the size of an instantiated array never changes. If you need to change the size, you have to create a new instance. Items can be accessed by their index. Beware! The first index is 0:

Code section 3.55: The array indexes.

```
1  char[] array4 = {'a', 'b', 'c', 'd', 'e'};
2  System.out.println(array4[2]);
3  array4[4] = 'z';
4  System.out.println(array4[4]);
```

Output for Code section 3.55

```
c
z
```

If you attempt to access to a too high index or negative index, you will get an `ArrayIndexOutOfBoundsException`.

Test your knowledge

Question 3.20: Consider the following code:

Question 3.20: Question20.java

```
1  public class Question20 {
2    public static void main(String[] args) {
3      String[] listOfWord = {"beggars", "can't", "be", "choosers"};
4      System.out.println(listOfWord[1]);
5      System.out.println(listOfWord[listOfWord.length-1]);
6    }
7  }
```

What will be printed in the standard output?

Answer

Output for Question 3.20

```
can't
choosers
```

Indexes start at 0. So the index 1 point at the second string (`can't`). There are 4 items so the size of the array is 4. Hence the item pointed by the index 3 is the last one (`choosers`).

Two-Dimensional Arrays

Actually, there are no two-dimensional arrays in Java. However, an array can contain any class of object, including an array:

Code section 3.56: Two-dimensional arrays.

```
1  String[][] twoDimArray = {{"a", "b", "c", "d", "e"},
2                            {"f", "g", "h", "i", "j"},
3                            {"k", "l", "m", "n", "o"}};
4
5  int[][] twoDimIntArray = {{ 0,  1,  2,  3,  4},
6                            {10, 11, 12, 13, 14},
7                            {20, 21, 22, 23, 24}};
```

It's not exactly equivalent to two-dimensional arrays because the size of the sub-arrays may vary. The sub-array reference can even be null. Consider:

Code section 3.57: Weird two-dimensional array.

```
1  String[][] weirdTwoDimArray = {{"10", "11", "12"},
2                                 null,
3                                 {"20", "21", "22", "23", "24"}};
```

Note that the length of a two-dimensional array is the number of one-dimensional arrays it contains. In the above example, `weirdTwoDimArray.length` is 3, whereas `weirdTwoDimArray[2].length` is 5.

In the code section 3.58, we defined an array that has three elements, each element contains an array having 5 elements. We could create the array having the 5 elements first and use that one in the initialize block.

Code section 3.58: Included array.

```
1  String[] oneDimArray = {"00", "01", "02", "03", "04"};
2  String[][] twoDimArray = {oneDimArray,
3
```

```
4                                {"10", "11", "12", "13", "14"},
                                 {"20", "21", "22", "23", "24"}};
```

Test your knowledge

Question 3.21: Consider the following code:

Question 3.21: The alphabet.

```
1  String[][] alphabet = {{"a", "b", "c", "d", "e"},
2                         {"f", "g", "h", "i", "j"},
3                         {"k", "l", "m", "n", "o"},
4                         {"p", "q", "r", "s", "t"},
5                         {"u", "v", "w", "x", "y"},
6                         {"z"}};
```

Print the whole alphabet in the standard output.

Answer

Question 3.21: Answer21.java

```
1  public class Answer21 {
2    public static void main(String[] args) {
3      String[][] alphabet = {{"a", "b", "c", "d", "e"},
4                             {"f", "g", "h", "i", "j"},
5                             {"k", "l", "m", "n", "o"},
6                             {"p", "q", "r", "s", "t"},
7                             {"u", "v", "w", "x", "y"},
8                             {"z"}};
9
10     for (int i = 0; i < alphabet.length; i++) {
11       for (int j = 0; j < alphabet[i].length; j++) {
12         System.out.println(alphabet[i][j]);
13       }
14     }
15   }
16 }
```

`i` will be the indexes of the main array and `j` will be the indexes of all the sub-arrays. We have to first iterate on the main array. We have to read the size of the array. Then we iterate on each sub-array. We have to read the size of each array as it may vary. Doing so, we iterate on all the sub-array items using the indexes. All the items will be read in the right order.

Multidimensional Array

Going further any number of dimensional array can be defined.

elementType[][]...[] ***arrayName***

or

elementType arrayName[][]...[]

Mathematical functions

The `java.lang.Math` class allows the use of many common mathematical functions that can be used while creating programs.

Since it is in the `java.lang` package, the `Math` class does not need to be imported. However, in programs extensively utilizing these functions, a static import can be used.

Math constants

There are two constants in the `Math` class that are fairly accurate approximations of irrational mathematical numbers.

Math.E

The `Math.E` constant represents the value of Euler's number (*e*), the base of the natural logarithm.

Code section 3.20: `Math.E`

```
1  public static final double E = 2.718281828459045;
```

Math.PI

The `Math.PI` constant represents the value of pi, the ratio of a circle's circumference to its diameter.

Code section 3.21: `Math.PI`

```
1  public static final double PI = 3.141592653589793;
```

Math methods

Exponential methods

There are several methods in the `Math` class that deal with exponential functions.


Java Programming/Print version

Contents

Exponentiation

The power method, `double Math.pow(double, double)`, returns the first parameter to the power of the second parameter. For example, a call to `Math.pow(2, 10)` will return a value of 1024 (2^{10}).

The `Math.exp(double)` method, a special case of `pow`, returns *e* to the power of the parameter. In addition, `double Math.expm1(double)` returns (e^x - 1). Both of these methods are more accurate and convenient in these special cases.

Java also provides special cases of the pow function for square roots and cube roots of doubles, `double Math.sqrt(double)` and `double Math.cbrt(double)`.

Logarithms

Java has no general logarithm function; when needed this can be simulated using the change-of-base theorem.

`double Math.log(double)` returns the natural logarithm of the parameter (**not the common logarithm**, as its name suggests!).

`double Math.log10(double)` returns the common (base-10) logarithm of the parameter.

`double Math.log1p(double)` returns ln(parameter+1). It is recommended for small values.

Trigonometric and hyperbolic methods

The trigonometric methods of the `Math` class allow users to easily deal with trigonometric functions in programs. All accept only **`double`**s. Please note that all values using these methods are initially passed and returned in **radians**, *not degrees*. However, conversions are possible.

Trigonometric functions

The three main trigonometric methods are `Math.sin(x)`, `Math.cos(x)`, and `Math.tan(x)`, which are used to find the sine, cosine, and tangent, respectively, of any given number. So, for example, a call to `Math.sin(Math.PI/2)` would return a value of about 1. Although methods for finding the cosecant, secant, and cotangent are not available, these values can be found by taking the reciprocal of the sine, cosine, and tangent, respectively. For example, the cosecant of pi/2 could be found using `1/Math.sin(Math.PI/2)`.

Inverse trigonometric functions

Java provides inverse counterparts to the trigonometric functions: `Math.asin(x)`, and `Math.acos(x)`, `Math.atan(x)`.

Hyperbolic functions

In addition, hyperbolic functions are available: `Math.sinh(x)`, `Math.cosh(x)`, and `Math.tanh(x)`.

Radian/degree conversion

To convert between degree and radian measures of angles, two methods are available, `Math.toRadians(x)` and `Math.toDegrees(x)`. While using `Math.toRadians(x)`, a degrees value must be passed in, and that value in radians (the degree value multiplied by pi/180) will be returned. The `Math.toDegrees(x)` method takes in a value in radians and the value in degrees (the radian value multiplied by 180/pi) is returned.

Absolute value: `Math.abs`

The absolute value method of the `Math` class is compatible with the **`int`**, **`long`**, **`float`**, and **`double`** types. The data returned is the absolute value of parameter (how far away it is from zero) in the same data type. For example:

Code section 3.22: `Math.abs`

```
1  int result = Math.abs(-3);
```

In this example, `result` will contain a value of 3.

Maximum and minimum values

These methods are very simple comparing functions. Instead of using **`if`**...**`else`** statements, one can use the `Math.max(x1, x2)` and `Math.min(x1, x2)` methods. The `Math.max(x1, x2)` simply returns the greater of the two values, while the `Math.min(x1, x2)` returns the lesser of the two. Acceptable types for these methods include **`int`**, **`long`**, **`float`**, and **`double`**.

Functions dealing with floating-point representation

Java 1.5 and 1.6 introduced several non-mathematical functions specific to the computer floating-point representation of numbers.

`Math.ulp(double)` and `Math.ulp(float)` return an ulp, the smallest value which, when added to the argument, would be recognized as larger than the argument.

`Math.copySign` returns the value of the first argument with the sign of the second argument. It can be used to determine the sign of a zero value.

`Math.getExponent` returns (as an **`int`**) the exponent used to scale the floating-point argument in computer representation.

Rounding number example

Sometimes, we are not only interested in mathematically correct rounded numbers, but we want that a fixed number of significant digits are always displayed, regardless of the number used. Here is an example program that returns always the correct string. You are invited to modify it such that it does the same and is simpler!

The constant class contains repeating constants that should exist only once in the code so that to avoid inadvertent changes. (If the one constant is changed inadvertently, it is most likely to be seen, as it is used at several locations.)

Code listing 3.20: StringUtils.java

```java
/**
 * Class that comprises of constant values & string utilities.
 *
 * @since 2013-09-05
 * @version 2014-10-14
 */
public class StringUtils {
    /** Dash or minus constant */
    public static final char DASH = '-';
    /** The exponent sign in a scientific number, or the capital letter E */
    public static final char EXPONENT = 'E';
    /** The full stop or period */
    public static final char PERIOD = '.';
    /** The zero string constant used at several places */
    public static final String ZERO = "0";

    /**
     * Removes all occurrences of the filter character in the text.
     *
     * @param text Text to be filtered
     * @param filter The character to be removed.
     * @return the string
     */
    public static String filter(final String text, final String filter) {
        final String[] words = text.split("[" + filter + "]");

        switch (words.length) {
            case 0: return text;
            case 1: return words[0];
            default:
                final StringBuilder filteredText = new StringBuilder();

                for (final String word : words) {
                    filteredText.append(word);
                }

                return filteredText.toString();
        }
    }
}
```

The MathsUtils class is like an addition to the `java.lang.Math` class and contains the rounding calculations.

Code listing 3.21: MathsUtils.java

```java
package string;

/**
 * Class for special mathematical calculations.<br/>
 * ATTENTION:<br/>Should depend only on standard Java libraries!
 *
 * @since 2013-09-05
 * @version 2014-10-14
*/
public class MathsUtils {

    // CONSTANTS
    // ----------------------------------------

    /** The exponent sign in a scientific number, or the capital letter E. */
    public static final char EXPONENT = 'E';

    /** Value after which the language switches from scientific to double */
    private static final double E_TO_DOUBLE = 1E-3;

```

```
21      /** The zero string constant used at several places. */
22      public static final String ZERO = "0";
23
24      /** The string of zeros */
25      private static final String ZEROS = "00000000000000000000000000000000";
26
27      // METHODS
28      // ----------------------------------------
29
30      /**
31       * Determines, if the number uses a scientific representation.
32       *
33       * @param number the number
34       * @return true, if it is a scientific number, false otherwise
35       */
36      private static boolean isScientific(final double number) {
37          return ((new Double(number)).toString().indexOf(EXPONENT) > 0);
38      }
39
40      /**
41       * Determines how many zeros are to be appended after the decimal digits.
42       *
43       * @param significantsAfter Requested significant digits after decimal
44       * @param separator Language-specific decimal separator
45       * @param number Rounded number
46       * @return Requested value
47       */
48      private static byte calculateMissingSignificantZeros(
49              final byte significantsAfter,
50              final char separator,
51              final double number) {
52
53          final byte after = findSignificantsAfterDecimal(separator, number);
54
55          final byte zeros =
56                  (byte) (significantsAfter - ((after == 0) ? 1 : after));
57
58          return ((zeros >= 0) ? zeros : 0);
59      }
60
61      /**
62       * Finds the insignificant zeros after the decimal separator.
63       *
64       * @param separator Language-specific decimal separator
65       * @param number the number
66       * @return the byte
67       */
68      private static byte findInsignificantZerosAfterDecimal(
69              final char separator,
70              final double number) {
71
72          if ((Math.abs(number) >= 1) || isScientific(number)) {
73              return 0;
74          } else {
75              final StringBuilder string = new StringBuilder();
76
77              string.append(number);
78              string.delete(0,
79                      string.indexOf(new Character(separator).toString()) + 1);
80
81              // Determine what to match:
82              final String regularExpression = "[1-9]";
83
84              final String[] split = string.toString().split(regularExpression);
85
86              return (split.length > 0) ? (byte) split[0].length() : 0;
87          }
88      }
89
90      /**
91       * Calculates the number of all significant digits (without the sign and
92       * the decimal separator).
93       *
94       * @param significantsAfter Requested significant digits after decimal
95       * @param separator Language-specific decimal separator
96       * @param number Value where the digits are to be counted
97       * @return Number of significant digits
98       */
99      private static byte findSignificantDigits(final byte significantsAfter,
100             final char separator,
101             final double number) {
102
103         if (number == 0) { return 0; }
104         else {
105             String mantissa =
106                     findMantissa(separator, new Double(number).toString());
107
108             if (number == (long)number) {
109                 mantissa = mantissa.substring(0, mantissa.length() - 1);
110             }
111
112             mantissa = retrieveDigits(separator, mantissa);
113             // Find the position of the first non-zero digit:
```

```
            short nonZeroAt = 0;

            for (; (nonZeroAt < mantissa.length())
                    && (mantissa.charAt(nonZeroAt) == '0'); nonZeroAt++) ;

            return (byte)mantissa.substring(nonZeroAt).length();
        }
    }

    /**
     * Determines the number of significant digits after the decimal separator
     * knowing the total number of significant digits and the number before the
     * decimal separator.
     *
     * @param significantsBefore Number of significant digits before separator
     * @param significantDigits Number of all significant digits
     * @return Number of significant decimals after the separator
     */
    private static byte findSignificantsAfterDecimal(
            final byte significantsBefore,
            final byte significantDigits) {

        final byte afterDecimal =
                (byte) (significantDigits - significantsBefore);

        return (byte) ((afterDecimal > 0) ? afterDecimal : 0);
    }

    /**
     * Determines the number of digits before the decimal point.
     *
     * @param separator Language-specific decimal separator
     * @param number Value to be scrutinised
     * @return Number of digits before the decimal separator
     */
    private static byte findSignificantsBeforeDecimal(final char separator,
                                                      final double number) {

        final String value = new Double(number).toString();

        // Return immediately, if result is clear: Special handling at
        // crossroads of floating point and exponential numbers:
        if ((number == 0) || (Math.abs(number) >= E_TO_DOUBLE)
                && (Math.abs(number) < 1)) {

            return 0;
        } else if ((Math.abs(number) > 0) && (Math.abs(number) < E_TO_DOUBLE)) {
            return 1;
        } else {
            byte significants = 0;
            // Significant digits to the right of decimal separator:
            for (byte b = 0; b < value.length(); b++) {
                if (value.charAt(b) == separator) {
                    break;
                } else if (value.charAt(b) != StringUtils.DASH) {
                    significants++;
                }
            }

            return significants;
        }
    }

    /**
     * Returns the exponent part of the double number.
     *
     * @param number Value of which the exponent is of interest
     * @return Exponent of the number or zero.
     */
    private static short findExponent(final double number) {
        return new Short(findExponent((new Double(number)).toString()));
    }

    /**
     * Finds the exponent of a number.
     *
     * @param value Value where an exponent is to be searched
     * @return Exponent, if it exists, or "0".
     */
    private static String findExponent(final String value) {
        final short exponentAt = (short) value.indexOf(EXPONENT);

        if (exponentAt < 0) { return ZERO; }
        else {
            return value.substring(exponentAt + 1);
        }
    }

    /**
     * Finds the mantissa of a number.
     *
     * @param separator Language-specific decimal separator
     * @param value Value where the mantissa is to be found
```

```
     * @return Mantissa of the number
     */
    private static String findMantissa(final char separator,
                                       final String value) {

        String strValue = value;

        final short exponentAt = (short) strValue.indexOf(EXPONENT);

        if (exponentAt > -1) {
            strValue = strValue.substring(0, exponentAt);
        }
        return strValue;
    }

    /**
     * Retrieves the digits of the value without decimal separator or sign.
     *
     * @param separator
     * @param number Mantissa to be scrutinised
     * @return The digits only
     */
    private static String retrieveDigits(final char separator, String number) {
        // Strip off exponent part, if it exists:
        short eAt = (short)number.indexOf(EXPONENT);

        if (eAt > -1) {
            number = number.substring(0, eAt);
        }

        return number.replace((new Character(StringUtils.DASH)).toString(), "").
                replace((new Character(separator)).toString(), "");
    }


    // ---- Public methods ----------------------

    /**
     * Returns the number of digits in the long value.
     *
     * @param value the value
     * @return the byte
     */
    public static byte digits(final long value) {
        return (byte) StringUtils.filter(Long.toString(value), ".,").length();
    }

    /**
     * Finds the significant digits after the decimal separator of a mantissa.
     *
     * @param separator Language-specific decimal separator
     * @param number Value to be scrutinised
     * @return Number of significant zeros after decimal separator.
     */
    public static byte findSignificantsAfterDecimal(final char separator,
                                                    final double number) {

        if (number == 0) { return 1; }
        else {
            String value = (new Double(number)).toString();

            final short separatorAt = (short) value.indexOf(separator);

            if (separatorAt > -1) {
                value = value.substring(separatorAt + 1);
            }

            final short exponentAt = (short) value.indexOf(EXPONENT);

            if (exponentAt > 0) {
                value = value.substring(0, exponentAt);
            }

            final Long longValue = new Long(value).longValue();

            if (Math.abs(number) < 1) {
                return (byte) longValue.toString().length();
            } else if (longValue == 0) {
                return 0;
            } else {
                return (byte) (("0." + value).length() - 2);
            }
        }
    }

    /**
     * Calculates the power of the base to the exponent without changing the
     * least-significant digits of a number.
     *
     * @param basis
     * @param exponent
     * @return basis to power of exponent
     */
```

```
300     public static double power(final int basis, final short exponent) {
301         return power((short) basis, exponent);
302     }
303 
304     /**
305      * Calculates the power of the base to the exponent without changing the
306      * least-significant digits of a number.
307      *
308      * @param basis the basis
309      * @param exponent the exponent
310      * @return basis to power of exponent
311      */
312     public static double power(final short basis, final short exponent) {
313         if (basis == 0) {
314             return (exponent != 0) ? 1 : 0;
315         } else {
316             if (exponent == 0) {
317                 return 1;
318             } else {
319                 // The Math method power does change the least significant
320                 // digits after the decimal separator and is therefore useless.
321                 double result = 1;
322                 short s = 0;
323 
324                 if (exponent > 0) {
325                     for (; s < exponent; s++) {
326                         result *= basis;
327                     }
328                 } else if (exponent < 0) {
329                     for (s = exponent; s < 0; s++) {
330                         result /= basis;
331                     }
332                 }
333 
334                 return result;
335             }
336         }
337     }
338 
339     /**
340      * Rounds a number to the decimal places.
341      *
342      * @param significantsAfter Requested significant digits after decimal
343      * @param separator Language-specific decimal separator
344      * @param number Number to be rounded
345      * @return Rounded number to the requested decimal places
346      */
347     public static double round(final byte significantsAfter,
348                                final char separator,
349                                final double number) {
350 
351         if (number == 0) { return 0; }
352         else {
353             final double constant = power(10, (short)
354                     (findInsignificantZerosAfterDecimal(separator, number)
355                             + significantsAfter));
356             final short dExponent = findExponent(number);
357 
358             short exponent = dExponent;
359 
360             double value = number*constant*Math.pow(10, -exponent);
361             final String exponentSign =
362                     (exponent < 0) ? String.valueOf(StringUtils.DASH) : "";
363 
364             if (exponent != 0) {
365                 exponent = (short) Math.abs(exponent);
366 
367                 value = round(value);
368             } else {
369                 value = round(value)/constant;
370             }
371 
372             // Power method cannot be used, as the exponentiated number may
373             // exceed the maximal Long value.
374             exponent -= Math.signum(dExponent)*(findSignificantDigits
375                     (significantsAfter, separator, value) - 1);
376 
377             if (dExponent != 0) {
378                 String strValue = Double.toString(value);
379 
380                 strValue = strValue.substring(0, strValue.indexOf(separator))
381                         + EXPONENT + exponentSign + Short.toString(exponent);
382 
383                 value = new Double(strValue);
384             }
385 
386             return value;
387         }
388     }
389 
390     /**
391      * Rounds a number according to mathematical rules.
392      *
```

```
393       * @param value the value
394       * @return the double
395       */
396      public static double round(final double value) {
397          return (long) (value + .5);
398      }
399
400      /**
401       * Rounds to a fixed number of significant digits.
402       *
403       * @param significantDigits Requested number of significant digits
404       * @param separator Language-specific decimal separator
405       * @param dNumber Number to be rounded
406       * @return Rounded number
407       */
408      public static String roundToString(final byte significantDigits,
409                                         final char separator,
410                                         double dNumber) {
411
412          // Number of significants that *are* before the decimal separator:
413          final byte significantsBefore =
414              findSignificantsBeforeDecimal(separator, dNumber);
415          // Number of decimals that *should* be after the decimal separator:
416          final byte significantsAfter = findSignificantsAfterDecimal(
417                  significantsBefore, significantDigits);
418          // Round to the specified number of digits after decimal separator:
419          final double rounded = MathsUtils.round(significantsAfter, separator, dNumber);
420
421          final String exponent = findExponent((new Double(rounded)).toString());
422          final String mantissa = findMantissa(separator,
423                          (new Double(rounded)).toString());
424
425          final double dMantissa = new Double(mantissa).doubleValue();
426          final StringBuilder result = new StringBuilder(mantissa);
427          // Determine the significant digits in this number:
428          final byte significants = findSignificantDigits(significantsAfter,
429                  separator, dMantissa);
430          // Add lagging zeros, if necessary:
431          if (significants <= significantDigits) {
432              if (significantsAfter != 0) {
433                  result.append(ZEROS.substring(0,
434                          calculateMissingSignificantZeros(significantsAfter,
435                                  separator, dMantissa)));
436              } else {
437                  // Cut off the decimal separator & after decimal digits:
438                  final short decimal = (short) result.indexOf(
439                          new Character(separator).toString());
440
441                  if (decimal > -1) {
442                      result.setLength(decimal);
443                  }
444              }
445          } else if (significantsBefore > significantDigits) {
446              dNumber /= power(10, (short) (significantsBefore - significantDigits));
447
448              dNumber = round(dNumber);
449
450              final short digits =
451                      (short) (significantDigits + ((dNumber < 0) ? 1 : 0));
452
453              final String strDouble = (new Double(dNumber)).toString().substring(0, digits);
454
455              result.setLength(0);
456              result.append(strDouble + ZEROS.substring(0,
457                      significantsBefore - significantDigits));
458          }
459
460          if (new Short(exponent) != 0) {
461              result.append(EXPONENT + exponent);
462          }
463
464          return result.toString();
465      } // public static String roundToString(…)
466
467      /**
468       * Rounds to a fixed number of significant digits.
469       *
470       * @param separator Language-specific decimal separator
471       * @param significantDigits Requested number of significant digits
472       * @param value Number to be rounded
473       * @return Rounded number
474       */
475      public static String roundToString(final char separator,
476                                         final int significantDigits,
477                                         float value) {
478
479          return roundToString((byte)significantDigits, separator,
480                  (double)value);
481      }
482  } // class MathsUtils
```

The code is tested with the following JUnit test:

Code listing 3.22: MathsUtilsTest.java

```
1  package string;
2
3  import static org.junit.Assert.assertEquals;
4  import static org.junit.Assert.assertFalse;
5  import static org.junit.Assert.assertTrue;
6
7  import java.util.Vector;
8
9  import org.junit.Test;
10
11 /**
12  * The JUnit test for the <code>MathsUtils</code> class.
13  *
14  * @since 2013-03-26
15  * @version 2014-10-14
16  */
17 public class MathsUtilsTest {
18
19     /**
20      * Method that adds a negative and a positive value to values.
21      *
22      * @param d the double value
23      * @param values the values
24      */
25     private static void addValue(final double d, Vector<Double> values) {
26         values.add(-d);
27         values.add(d);
28     }
29
30     // Public methods ------
31
32     /**
33      * Tests the round method with a double parameter.
34      */
35     @Test
36     public void testRoundToStringDoubleByteCharDouble() {
37         // Test rounding
38         final Vector<Double> values = new Vector<Double>();
39         final Vector<String> strValues = new Vector<String>();
40
41         values.add(0.0);
42         strValues.add("0.00000");
43         addValue(1.4012984643248202e-45, values);
44         strValues.add("-1.4012E-45");
45         strValues.add("1.4013E-45");
46         addValue(1.999999757e-5, values);
47         strValues.add("-1.9999E-5");
48         strValues.add("2.0000E-5");
49         addValue(1.999999757e-4, values);
50         strValues.add("-1.9999E-4");
51         strValues.add("2.0000E-4");
52         addValue(1.999999757e-3, values);
53         strValues.add("-0.0019999");
54         strValues.add("0.0020000");
55         addValue(0.000640589, values);
56         strValues.add("-6.4058E-4");
57         strValues.add("6.4059E-4");
58         addValue(0.3396899998188019, values);
59         strValues.add("-0.33968");
60         strValues.add("0.33969");
61         addValue(0.34, values);
62         strValues.add("-0.33999");
63         strValues.add("0.34000");
64         addValue(7.07, values);
65         strValues.add("-7.0699");
66         strValues.add("7.0700");
67         addValue(118.188, values);
68         strValues.add("-118.18");
69         strValues.add("118.19");
70         addValue(118.2, values);
71         strValues.add("-118.19");
72         strValues.add("118.20");
73         addValue(123.405009, values);
74         strValues.add("-123.40");
75         strValues.add("123.41");
76         addValue(30.76994323730469, values);
77         strValues.add("-30.769");
78         strValues.add("30.770");
79         addValue(130.76994323730469, values);
80         strValues.add("-130.76");
81         strValues.add("130.77");
82         addValue(540, values);
83         strValues.add("-539.99");
84         strValues.add("540.00");
85         addValue(12345, values);
86         strValues.add("-12344");
```

```
        strValues.add("12345");
        addValue(123456, values);
        strValues.add("-123450");
        strValues.add("123460");
        addValue(540911, values);
        strValues.add("-540900");
        strValues.add("540910");
        addValue(9.223372036854776e56, values);
        strValues.add("-9.2233E56");
        strValues.add("9.2234E56");

        byte i = 0;
        final byte significants = 5;

        for (final double element : values) {
            final String strValue;

            try {
                strValue = MathsUtils.roundToString(significants, StringUtils.PERIOD, element);

                System.out.println(" MathsUtils.round(" + significants  + ", '"
                        + StringUtils.PERIOD + "', " + element + ") ==> "
                        + strValue + " = " + strValues.get(i));
                assertEquals("Testing roundToString", strValue, strValues.get(i++));
            } catch (final Exception e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }

}   // class MathsUtilsTest
```

The output of the JUnit test follows:

Output for code listing 3.22

```
MathsUtils.round(5, '.', 0.0) ==> 0.00000 = 0.00000
MathsUtils.round(5, '.', -1.4012984643248202E-45) ==> -1.4012E-45 = -1.4012E-45
MathsUtils.round(5, '.', 1.4012984643248202E-45) ==> 1.4013E-45 = 1.4013E-45
MathsUtils.round(5, '.', -1.999999757E-5) ==> -1.9999E-5 = -1.9999E-5
MathsUtils.round(5, '.', 1.999999757E-5) ==> 2.0000E-5 = 2.0000E-5
MathsUtils.round(5, '.', -1.999999757E-4) ==> -1.9999E-4 = -1.9999E-4
MathsUtils.round(5, '.', 1.999999757E-4) ==> 2.0000E-4 = 2.0000E-4
MathsUtils.round(5, '.', -0.001999999757) ==> -0.0019999 = -0.0019999
MathsUtils.round(5, '.', 0.001999999757) ==> 0.0020000 = 0.0020000
MathsUtils.round(5, '.', -6.40589E-4) ==> -6.4058E-4 = -6.4058E-4
MathsUtils.round(5, '.', 6.40589E-4) ==> 6.4059E-4 = 6.4059E-4
MathsUtils.round(5, '.', -0.3396899998188019) ==> -0.33968 = -0.33968
MathsUtils.round(5, '.', 0.3396899998188019) ==> 0.33969 = 0.33969
MathsUtils.round(5, '.', -0.34) ==> -0.33999 = -0.33999
MathsUtils.round(5, '.', 0.34) ==> 0.34000 = 0.34000
MathsUtils.round(5, '.', -7.07) ==> -7.0699 = -7.0699
MathsUtils.round(5, '.', 7.07) ==> 7.0700 = 7.0700
MathsUtils.round(5, '.', -118.188) ==> -118.18 = -118.18
MathsUtils.round(5, '.', 118.188) ==> 118.19 = 118.19
MathsUtils.round(5, '.', -118.2) ==> -118.19 = -118.19
MathsUtils.round(5, '.', 118.2) ==> 118.20 = 118.20
MathsUtils.round(5, '.', -123.405009) ==> -123.40 = -123.40
MathsUtils.round(5, '.', 123.405009) ==> 123.41 = 123.41
MathsUtils.round(5, '.', -30.76994323730469) ==> -30.769 = -30.769
MathsUtils.round(5, '.', 30.76994323730469) ==> 30.770 = 30.770
MathsUtils.round(5, '.', -130.7699432373047) ==> -130.76 = -130.76
MathsUtils.round(5, '.', 130.7699432373047) ==> 130.77 = 130.77
MathsUtils.round(5, '.', -540.0) ==> -539.99 = -539.99
MathsUtils.round(5, '.', 540.0) ==> 540.00 = 540.00
MathsUtils.round(5, '.', -12345.0) ==> -12344 = -12344
MathsUtils.round(5, '.', 12345.0) ==> 12345 = 12345
MathsUtils.round(5, '.', -123456.0) ==> -123450 = -123450
MathsUtils.round(5, '.', 123456.0) ==> 123460 = 123460
MathsUtils.round(5, '.', -540911.0) ==> -540900 = -540900
MathsUtils.round(5, '.', 540911.0) ==> 540910 = 540910
MathsUtils.round(5, '.', -9.223372036854776E56) ==> -9.2233E56 = -9.2233E56
MathsUtils.round(5, '.', 9.223372036854776E56) ==> 9.2234E56 = 9.2234E56
```

If you are interested in a comparison with C#, take a look at the rounding number example there. If you are interested in a comparison with C++, you can compare this code here with the same example over there.

Notice that in the expression starting with `if ((D == 0)`, I have to use OR instead of the `||` because of a bug in the source template.

Large numbers

The integer primitive type with the largest range of value is the **long**, from -2^{63} to $2^{63}-1$. If you need greater or lesser values, you have to use the `BigInteger` class in the package `java.math`. A `BigInteger` object can represent any integer (as large as the RAM on the computer can hold) as it is not mapped on a primitive type. Respectively, you need to use the `BigDecimal` class for great decimal numbers.

However, as these perform much slower than primitive types, it is recommended to use primitive types when it is possible.

BigInteger

The `BigInteger` class represents integers of almost any size. As with other objects, they need to be constructed. Unlike regular numbers, the `BigInteger` represents an immutable object - methods in use by the `BigInteger` class will return a new copy of a `BigInteger`.

To instantiate a `BigInteger`, you can create it from either byte array, or from a string. For example:

Code section 3.23: 1 quintillion, or 10^18. Too large to fit in a long.

```
1  BigInteger i = new BigInteger("1000000000000000000");
```

`BigInteger` cannot use the normal Java operators. They use the methods provided by the class.

Code section 3.24: Multiplications and an addition.

```
1  BigInteger a = new BigInteger("3");
2  BigInteger b = new BigInteger("4");
3
4  // c = a^2 + b^2
5  BigInteger c = a.multiply(a).add(b.multiply(b));
```

It is possible to convert to a **long**, but the **long** may not be large enough.

Code section 3.25: Conversion.

```
1  BigInteger aBigInteger = new BigInteger("3");
2  long aLong = aBigInteger.longValue();
```

BigDecimal

The `BigInteger` class cannot handle decimal numbers. The `BigDecimal` class represents a floating point value of arbitrary precision. It is composed of both a `BigInteger`, and a scale value (represented by a 32-bit integer).

Random numbers

To generate random numbers the `Math.random()` method can be used, which returns a **double**, greater than or equal to 0.0 and less than 1.0.

The following code returns a random integer between n and m (where $n <= randomNumber < m$):

Code section 3.30: A random integer.

```
int randomNumber = n + (int)(Math.random() * ( m - n ));
```

Alternatively, the `java.util.Random` class provides methods for generating random **`boolean`**s, **`byte`**s, **`float`**s, **`int`**s, **`long`**s and 'Gaussians' (**`double`**s from a normal distribution with mean 0.0 and standard deviation 1.0). For example, the following code is equivalent to that above:

Code section 3.31: A random integer with Gaussian.

```
Random random = new Random();
int randomNumber = n + random.nextInt(m - n);
```

As an example using random numbers, we can make a program that uses a Random object to simulate flipping a coin 20 times:

Code listing 3.25: CoinFlipper.java

```
import java.util.Random;

public class CoinFlipper {

  public static void main(String[] args) {
    // The number of times to flip the coin
    final int TIMES_TO_FLIP = 20;
    int heads = 0;
    int tails = 0;
    // Create a Random object
    Random random = new Random();
    for (int i = 0; i < TIMES_TO_FLIP; i++) {
      // 0 or 1
      int result = random.nextInt(2);
      if (result == 1) {
        System.out.println("Heads");
        heads++;
      } else {
        System.out.println("Tails");
        tails++;
      }
    }
    System.out.println("There were "
            + heads
            + " heads and "
            + tails
            + " tails");
  }
}
```

Possible output for code listing 3.25

```
Heads
Tails
Tails
Tails
Heads
Tails
Heads
Heads
Heads
Heads
Heads
Heads
Tails
Tails
Tails
Tails
Heads
Tails
Tails
Tails
There were 9 heads and 11 tails
```

Of course, if you run the program you will probably get different results.

Truly random numbers

Both `Math.random()` and the `Random` class produce pseudorandom numbers. This is good enough for a lot of applications, but remember that it is not *truly* random. If you want a more secure random number generator, Java provides the `java.security.SecureRandom` package. What happens with `Math.random()` and the `Random` class is that a 'seed' is chosen from which the pseudorandom numbers are generated. `SecureRandom` increases the security to ensure that the seed which is used by the pseudorandom number generator is non-deterministic — that is, you cannot simply put the machine in the same state to get the same set of results. Once you have created a `SecureRandom` instance, you can use it in the same way as you can the `Random` class.

If you want *truly* random numbers, you can get a hardware random number generator or use a randomness generation service.

Unicode

Most Java program text consists of ASCII characters, but any Unicode character can be used as part of identifier names, in comments, and in character and string literals. For example, π (which is the Greek Lowercase Letter **pi**) is a valid Java identifier:

Code section 3.100: Pi.

```
1  double π = Math.PI;
```

and in a string literal:

Code section 3.101: Pi literal.

```
1  String pi = "π";
```

Unicode escape sequences

Unicode characters can also be expressed through Unicode Escape Sequences. Unicode escape sequences may appear anywhere in a Java source file (including inside identifiers, comments, and string literals).

Unicode escape sequences consist of

1. a backslash '\' (ASCII character 92, hex 0x5c),
2. a 'u' (ASCII 117, hex 0x75)
3. optionally one or more additional 'u' characters, and
4. four hexadecimal digits (the characters '0' through '9' or 'a' through 'f' or 'A' through 'F').

Such sequences represent the UTF-16 encoding of a Unicode character. For example, 'a' is equivalent to '\u0061'. This escape method does not support characters beyond U+FFFF or you have to make use of surrogate pairs.[1]

Any and all characters in a program may be expressed in Unicode escape characters, but such programs are not very readable, except by the Java compiler - in addition, they are not very compact.

One can find a full list of the characters here.

π may also be represented in Java as the *Unicode escape sequence* `\u03C0`. Thus, the following is a valid, but not very readable, declaration and assignment:

Code section 3.102: Unicode escape sequences for Pi.

```
1  double \u03C0 = Math.PI;
```

The following demonstrates the use of Unicode escape sequences in other Java syntax:

Code section 3.103: Unicode escape sequences in a string literal.

```
1  // Declare Strings pi and quote which contain \u03C0 and \u0027 respectively:
2  String pi = "\u03C0";
3  String quote = "\u0027";
```

Note that a Unicode escape sequence functions just like any other character in the source code. E.g., `\u0022` (double quote, ") needs to be quoted in a string just like ".

Code section 3.104: Double quote.

```
1  // Declare Strings doubleQuote1 and doubleQuote2 which both contain " (double quote):
2  String doubleQuote1 = "\"";
3  String doubleQuote2 = "\\u0022"; // "\u0022" doesn't work since """ doesn't work.
```

International language support

The language distinguishes between bytes and characters. Characters are stored internally using UCS-2, although as of J2SE 5.0, the language also supports using UTF-16 and its surrogates. Java program source may therefore contain any Unicode character.

The following is thus perfectly valid Java code; it contains Chinese characters in the class and variable names as well as in a string literal:

Code listing 3.50: 哈嘍世界.java

```
1  public class 哈嘍世界 {
2      private String 文本 = "哈嘍世界";
3  }
```

References

1. "3.1 Unicode", The Java™ Language Specification [1] (http://download.oracle.com/otn-pub/jcp/jls-7-mr3-fullv-oth-JSpec/JLS-JavaSE7-Full.pdf), Java SE 7 Edition, pp. 15-16.

Comments

A comment allows to insert text that will not be compiled nor interpreted. It can appear anywhere in the source code where whitespaces are allowed.

It is useful for explaining what the source code does by:

- explaining the adopted technical choice: why this given algorithm and not another, why calling this given method...
- explaining what should be done in the next steps (the TODO list): improvement, issue to fix...
- giving the required explanation to understand the code and be able to update it yourself later or by other developers.

It can also be used to make the compiler ignore a portion of code: temporary code for debugging, code under development...

Syntax

The comments in Java use the same syntax as in C++.

An end-of-line comment starts with two slashes and ends with the end of the line. This syntax can be used on a single line too.

Code section 3.105: Slash-slash comment.

```
1  // A comment to give an example
2
3  int n = 10; // 10 articles
```

A comment on several lines is framed with '/' + '*' and '*' + '/'.

Code section 3.106: Slash-star comment in multiple lines.

```
1  /*
2   * This is a comment
3   * on several lines.
```

```
 */

/* This also works; slash-star comments may be on a single line. */

/*
Disable debugging code:

int a = 10;
while (a-- > 0) System.out.println("DEBUG: tab["+a+"]=" + tab[a]);
*/
```

By convention, subsequent lines of slash-star comments begin with a star aligned under the star in the open comment sequence, but this is not required. Never nest a slash-star comment in another slash-star comment. If you accidentally nest such comments, you will probably get a syntax error from the compiler soon after the first star-slash sequence.

Code section 3.107: Nested slash-star comment.

```
/* This comment appears to contain /* a nested comment. */
 * The comment ends after the first star-slash and
 * everything after the star-slash sequence is parsed
 * as non-comment source.
 */
```

If you need to have the sequence */ inside a comment you can use html numeric entities: `*&#47;`.

Slash-star comments may also be placed between any Java tokens, though not recommended:

Code section 3.108: Inline slash-star comment.

```
int i = /* maximum integer */ Integer.MAX_VALUE;
```

However, comments are not parsed as comments when they occur in string literals.

Code section 3.109: String literal.

```
String text = "/* This is not a comment. */";
```

It results in a 33 character string.

Test your knowledge

Question 3.26: Consider the following code:

Question 3.26: Commented code.

```
int a = 0;
// a = a + 1;
a = a + 1;
/*
a = a + 1;
*/
a = a + 1;
// /*
a = a + 1;
// */
a = a /*+ 1*/;
a = a + 1; // a = a + 1;
System.out.println("a=" + a);
```

What is printed in the standard output?

Answer

Output for Answer 3.26

```
a=4
```

Answer 3.26: Commented code.

```
1  int a = 0;
2  // a = a + 1;
3  a = a + 1;
4  /*
5  a = a + 1;
6  */
7  a = a + 1;
8  // /*
9  a = a + 1;
10 // */
11 a = a /*+ 1*/;
12 a = a + 1; // a = a + 1;
13 System.out.println("a=" + a);
```

The highlighted lines are code lines but line 11 does nothing and only the first part of line 12 is code.

Comments and unicode

Be aware that Java still interprets Unicode sequences within comments. For example, the Unicode sequence `\u002a\u002f` (whose codepoints correspond to */) is processed early in the Java compiler's lexical scanning of the source file, even before comments are processed, so this is a valid star-slash comment in Java:

Code section 3.110: Unicode sequence interruption.

```
1  /* This is a comment. \u002a\u002f
2  String statement = "This is not a comment.";
```

and is lexically equivalent to

Code section 3.111: Unicode sequence interruption effect.

```
1  /* This is a comment. */
2  String statement = "This is not a comment.";
```

(The `'*'` character is Unicode `002A` and the `'/'` character is Unicode `002F`.)

Similar caveats apply to newline characters in slash-slash comments.

For example:

Code section 3.112: New line.

```
1  // This is a single line comment \u000a This is code
```

That is because `\u000a` is Unicode for a new line, making the compiler think that you have added a new line when you haven't.

Javadoc comments

Javadoc comments are a special case of slash-star comments.

Code section 3.113: Javadoc comment.

```
1  /**
2   * Comments which start with slash-star-star are Javadoc comments.
3   * These are used to extract documentation from the Java source.
```

```
 * More on javadoc will be covered later.
 */
```

Coding conventions

The Java code conventions are defined by Oracle in the coding conventions (http://www.oracle.com/technetwork/java/codeconventions-150003.pdf) document. In short, these conventions ask the user to use camel case when defining classes, methods, or variables. Classes start with a capital letter and should be nouns, like `CalendarDialogView`. For methods, the names should be verbs in imperative form, like `getBrakeSystemType`, and should start with a lowercase letter.

It is important to get used to and follow coding conventions, so that code written by multiple programmers will appear the same. Projects may re-define the standard code conventions to better fit their needs. Examples include a list of allowed abbreviations, as these can often make the code difficult to understand for other designers. Documentation should always accompany code.

One example from the coding conventions is how to define a constant. Constants should be written with capital letters in Java, where the words are separated by an underscore ('_') character. In the Java coding conventions, a constant is a `static final` field in a class.

The reason for this diversion is that Java is not 100% object-oriented and discerns between "simple" and "complex" types. These will be handled in detail in the following sections. An example for a simple type is the `byte` type. An example for a complex type is a class. A subset of the complex types are classes that cannot be modified after creation, like a `String`, which is a concatenation of characters.

For instance, consider the following "constants":

```
public class MotorVehicle {
  /** Number of motors */
  private static final int MOTORS = 1;

  /** Name of a motor */
  private static final String MOTOR_NAME = "Mercedes V8";

  /** The motor object */
  private static final Motor THE_MOTOR = new MercedesMotor();

  /**
   * Constructor
   */
  public MotorVehicle() {
    MOTORS = 2;                    // Gives a syntax error as MOTORS has already been assigned a value.
    THE_MOTOR = new ToshibaMotor(); // Gives a syntax error as THE_MOTOR has already been assigned a value.
    MOTOR_NAME.toLowercase();      // Does not give a syntax error, because it returns a new String rather than editing the MOTOR_NAME variable.
    THE_MOTOR.fillFuel(20.5);      // Does not give a syntax error, as it changes a variable in the motor object, not the variable itself.
  }
}
```

Classes and Objects

Classes and Objects

An object-oriented program is built from objects. A class is a "template" that is used to create objects. The class defines the values the object can contain and the operations that can be performed on the object.

After compilation, a class is stored on the file system in a '(class-name).class' file.

The class is loaded into memory when we are about to create the first object from that class, or when we call one of its static functions.

During class loading all the class static variables are initialized. Also operations defined in a `static { ... }` block are executed. Once a class is loaded it stays in memory, and the class static variables won't be initialized again.

After the class is loaded into memory, objects can be created from that class. When an object is created, its member variables are initialized, but the class static variables are not.

When there are no more references to an object, the garbage collector will destroy the object and free its memory, so that the memory can be reused to hold new objects.

Defining Classes

Fundamentals

Every class in Java can be composed of the following elements:

- **fields**, **member variables** or **instance variables —** Fields are variables that hold data specific to each object. For example, an employee might have an ID number. There is one field for each object of a class.
- **member methods** or **instance methods —** Member methods perform operations on an object. For example, an employee might have a method to issue his paycheck or to access his name.
- **static fields** or **class fields —** Static fields are common to any object of the same class. For example, a static field within the Employee class could keep track of the last ID number issued. Each static field exists only once in the class, regardless of how many objects are created for that class.
- **static methods** or **class methods —** Static methods are methods that do not affect a specific object.
- **inner classes —** Sometimes it is useful to contain a class within another one if it is useless outside of the class or should not be accessed outside the class.
- **Constructors —** A special method that generates a new object.
- **Parameterized types —** Since 1.5, *parameterized types* can be assigned to a class during definition. The *parameterized types* will be substituted with the types specified at the class's instantiation. It is done by the compiler. It is similar to the C language macro '#define' statement, where a preprocessor evaluates the macros.

Code listing 4.1: Employee.java

```
public class Employee {              // This defines the Employee class.
                                     // The public modifier indicates that
                                     // it can be accessed by any other class

    private static int nextID;       // Define a static field. Only one copy of this will exist,
                                     // no matter how many Employees are created.

    private int myID;                // Define fields that will be stored
    private String myName;           // for each Employee. The private modifier indicates that
                                     // only code inside the Employee class can access it.

    public Employee(String name) {   // This is a constructor. You can pass a name to the constructor
                                     // and it will give you a newly created Employee object.
        myName = name;
        myID = nextID;               // Automatically assign an ID to the object
        nextID++;                    // Increment the ID counter
    }

    public String getName() {        // This is a member method that returns the
                                     // Employee object's name.
        return myName;               // Note how it can access the private field myName.
    }

    public int getID() {             // This is another member method.

        return myID;
    }

    public static int getNextID() {  // This is a static method that returns the next ID
                                     // that will be assigned if another Employee is created.
        return nextID;
    }
}
```

The following Java code would produce this output:

Code listing 4.2: EmployeeList.java

```
1  public class EmployeeList {
2      public static void main(String[] args) {
3
4          System.out.println(Employee.getNextID());
5
6          Employee a = new Employee("John Doe");
7          Employee b = new Employee("Jane Smith");
8          Employee c = new Employee("Sally Brown");
9
10         System.out.println(Employee.getNextID());
11
12         System.out.println(a.getID() + ": " + a.getName());
13         System.out.println(b.getID() + ": " + b.getName());
14         System.out.println(c.getID() + ": " + c.getName());
15     }
16 }
```

Console for Code listing 4.2

```
0
3
0: John Doe
1: Jane Smith
2: Sally Brown
```

Constructors

A constructor is called to initialize an object immediately after the object has been allocated:

Code listing 4.3: Cheese.java

```
1  public class Cheese {
2    // This is a constructor
3    public Cheese() {
4      System.out.println("Construct an instance");
5    }
6  }
```

Typically, a constructor is invoked using the **new** keyword:

Code section 4.1: A constructor call.

```
1  Cheese cheese = new Cheese();
```

The constructor syntax is close to the method syntax. However, the constructor has the same name as the name of the class (with the same case) and the constructor has no return type. The second point is the most important difference as a method can also have the same name as the class, which is not recommended:

Code listing 4.4: Cheese.java

```
1  public class Cheese {
2    // This is a method with the same name as the class
3    public void Cheese() {
4      System.out.println("A method execution.");
5    }
6  }
```

The returned object is always a valid, meaningful object, as opposed to relying on a separate initialization method. A constructor cannot be **abstract**, **final**, **native**, **static**, **strictfp** nor **synchronized**. However, a constructor, like methods, can be overloaded and take parameters.

Code listing 4.5: Cheese.java

```
1  public class Cheese {
2      // This is a constructor
3      public Cheese() {
4          doStuff();
```

```
    }

    // This is another constructor
    public Cheese(int weight) {
        doStuff();
    }

    // This is yet another constructor
    public Cheese(String type, int weight) {
        doStuff();
    }
}
```

By convention, a constructor that accepts an object of its own type as a parameter and copies the data members is called a *copy constructor*. One interesting feature of constructors is that if and only if you do not specify a constructor in your class, the compiler will create one for you. This default constructor, if written out would look like:

Code listing 4.6: Cheese.java

```
public class Cheese {
    public Cheese() {
        super();
    }
}
```

The `super()` command calls the constructor of the superclass. If there is no explicit call to `super(...)` or `this(...)`, then the default superclass constructor `super();` is called before the body of the constructor is executed. That said, there are instances where you need to add in the call manually. For example, if you write even one constructor, no matter what parameters it takes, the compiler will not add a default constructor. The code listing 4.8 results in a runtime error:

Code listing 4.7: Cheese.java

```
public class Cheese {
    public Cheese(int weight, String type) {
        doStuff();
    }
}
```

Code listing 4.8: Mouse.java

```
public class Mouse {
    public void eatCheese() {
        Cheese c = new Cheese(); // Oops, compile time error!
    }
}
```

This is something to keep in mind when extending existing classes. Either make a default constructor, or make sure every class that inherits your class uses the correct constructor.

Initializers

Initializers are blocks of code that are executed at the same time as initializers for fields.

Static initializers

Static initializers are blocks of code that are executed at the same time as initializers for static fields. Static field initializers and static initializers are executed in the order declared. The static initialization is executed after the class is loaded.

Code section 4.2: Static initializer.

```
static int count = 20;
static int[] squares;
static {  // a static initializer
```

```
    squares = new int[count];
    for (int i = 0; i < count; i++)
        squares[i] = i * i;
}
static int x = squares[5];  // x is assigned the value 25
```

Instance initializers

Instance initializers are blocks of code that are executed at the same time as initializers for instance (non-`static`) fields. Instance field initializers and instance initializers are executed in the order declared. Both instance initializers and instance field initializers are executed during the invocation of a constructor. The initializers are executed immediately after the superclass constructor and before the body of the constructor.

Inheritance

Inheritance is one of the most powerful mechanisms of the Object Oriented Programming. It allows the reuse of the members of a class (called the *superclass* or the *mother class*) in another class (called *subclass*, *child class* or the *derived class*) that inherits from it. This way, classes can be built by successive inheritance.

In Java, this mechanism is enabled by the **`extends`** keyword. Example:

Code listing 4.9: Vehicle.java

```
public class Vehicle {
    public int speed;
    public int numberOfSeats;
}
```

Code listing 4.10: Car.java

```
public class Car extends Vehicle {
    public Car() {
        this.speed = 90;
        this.numberOfSeats = 5;
    }
}
```

In the Code listing 4.10, the class `Car` inherits from `Vehicle`, which means that the attributes `speed` and `numberOfSeats` are present in the class `Car`, whereas they are defined in the class `Vehicle`. Also, the constructor defined in the class `Car` allows to initialize those attributes. In Java, the inheritance mechanism allows to define a class hierarchy with all the classes. Without explicit inheritance, a class implicitly inherits from the `Object` class. This `Object` class is the root of the class hierarchy.

Some classes can't be inherited. Those classes are defined with the **`final`** keyword. For instance, the `Integer` class can't have subclasses. It is called a *final* class.

The `Object` class

At the instantiating, the child class receives the features inherited from its superclass, which also has received the features inherited from its own superclass and so on to the `Object` class. This mechanism allows to define reusable global classes, whose user details the behavior in the derived more specific classes.

In Java, a class can only inherit from one class. Java does not allow you to create a subclass from two classes, as that would require creating complicated rules to disambiguate fields and methods inherited from multiple superclasses. If there is a need for Java to inherit from multiple sources, the best option is through interfaces, described in the next chapter.

The *super* keyword

The **super** keyword allows access to the members of the superclass of a class, as you can use **this** to access the members of the current class. Example:

Code listing 4.11: Plane.java

```
1  public class Plane extends Vehicle {
2      public Plane() {
3          super();
4      }
5  }
```

In this example, the constructor of the `Plane` class calls the constructor of its superclass `Vehicle`. You can only use **super** to access the members of the superclass inside the child class. If you use it from another class, it accesses the superclass of the other class. This keyword also allows you to explicitly access the members of the superclass, for instance, in the case where there is a method with the same name in your class (overriding, ...). Example :

Code listing 4.12: Vehicle.java

```
1  public class Vehicle {
2      // ...
3      public void run() throws Exception {
4          position += speed;
5      }
6  }
```

Code listing 4.13: Plane.java

```
1  public class Plane extends Vehicle {
2      // ...
3      public void run() throws Exception {
4          if (0 < height) {
5             throw new Exception("A plane can't run in flight.");
6          } else {
7             super.run();
8          }
9      }
10 }
```

Test your knowledge

Question 4.1: Consider the following classes.

Question 4.1: Class1.java

```
1  public class Class1 {
2      public static final int CONSTANT_OF_CLASS_1 = 9;
3      public int myAttributeOfClass1 = 40;
4      public void myMethodOfClass1(int i) {
5      }
6  }
```

Question 4.1: Class2.java

```
1  public class Class2 extends Class1 {
2      public int myAttributeOfClass2 = 10;
3      public void myMethodOfClass2(int i) {
4      }
5  }
```

Question 4.1: Class3.java

```
1  public class Class3 {
2      public static final int CONSTANT_OF_CLASS_3 = 9;
3      public void myMethodOfClass3(int i) {
```

```
    }
}
```

Question 4.1: Question1.java

```
public class Question1 extends Class2 {
    public static final int CONSTANT = 2;
    public int myAttribute = 20;
    public void myMethod(int i) {
    }
}
```

List all the attributes and methods that can be accessed in the class `Question1`.

Answer

- `CONSTANT_OF_CLASS_1`
- `myAttributeOfClass1`
- `myMethodOfClass1(int)`
- `myAttributeOfClass2`
- `myMethodOfClass2(int)`
- `CONSTANT`
- `myAttribute`
- `myMethod(int)`

`Question1` inherits from `Class1` and `Class2` but not from `Class3`.

See also the Object Oriented Programming book about the inheritance concept.

Interfaces

An interface is an abstraction of class with no implementation details. For example, `java.lang.Comparable` is a standard interface in Java. You cannot instantiate an interface. An interface is not a class but it is written the same way. The first difference is that you do not use the **`class`** keyword but the **`interface`** keyword to define it. Then, there are fields and methods you cannot define here:

- A field is always a constant: it is always public, static and final, even if you do not mention it.
- A method must be public and abstract, but it is not required to write the **`public`** and **`abstract`** keywords.
- Constructors are forbidden.

An interface represents a *contract*:

Code listing 4.14: SimpleInterface.java

```
public interface SimpleInterface {
    public static final int CONSTANT1 = 1;
    int method1(String parameter);
}
```

You can see that the `method1()` method is abstract (unimplemented). To use an interface, you have to define a class that implements it, using the **`implements`** keyword:

Code listing 4.15: ClassWithInterface.java

```
public class ClassWithInterface implements SimpleInterface {
    public int method1(String parameter) {
        return 0;
```

```
    }
}
```

A class can implement several interface, separated by a comma. Java interfaces behave much like the concept of the Objective-C protocol. It is recommended to name an interface `<verb>able`, to mean the type of action this interface would enable on a class. However, it is not recommended to start the name of an interface by `I` as in C++. It is useless. Your IDE will help you instead.

Interest

If you have objects from different classes that do not have a common superclass, you can't call the same method in those classes, even if the two classes implement a method with the same signature.

Code listing 4.16: OneClass.java

```
public class OneClass {
    public int method1(String parameter) {
        return 1;
    }
}
```

Code listing 4.17: AnotherClass.java

```
public class AnotherClass {
    public int method1(String parameter) {
        return 2;
    }
}
```

Code section 4.16: Impossible call.

```
public static void main(String[] args) {
    doAction(new OneClass());
    doAction(new AnotherClass());
}

public void doAction(Object anObject) {
    anObject.method1("Hello!");
}
```

The solution is to write an interface that defines the method that should be implemented in the two classes as the `SimpleInterface` in the Code listing 4.14 and then the both classes can implement the interface as in the Code listing 4.15.

Code section 4.17: Interface use.

```
public static void main(String[] args) {
    doAction(new ClassWithInterface());
    doAction(new AnotherClassWithInterface());
}

public void doAction(SimpleInterface anObject) {
    anObject.method1("Hello!");
}
```

You can also implement this using a common super class but a class can only inherit from one super class whereas it can implement several interfaces.

Java does not support full orthogonal multiple inheritance (i.e. Java does not allow you to create a subclass from two classes). Multiple inheritance in C++ has complicated rules to disambiguate fields and methods inherited from multiple superclasses and types that are inherited multiple times. By separating interface from implementation, interfaces offer much of the benefit of multiple inheritance with less complexity and ambiguity. The price of no multiple inheritance is some code redundancy; since interfaces only define the signature of a class but cannot contain

any implementation, every class inheriting an interface must provide the implementation of the defined methods, unlike in pure multiple inheritance, where the implementation is also inherited. The major benefit of that is that all Java objects can have a common ancestor (a class called `Object`).

When overriding methods defined in interfaces there are several rules to be followed:

- Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.
- The signature of the interface method and the same return type or subtype should be maintained when implementing the methods.
- All the methods of the interface need to be defined in the class, unless the class that implements the interface is abstract.

Extending interfaces

An interface can extend several interfaces, similar to the way that a class can extend another class, using the **`extends`** keyword:

Code listing 4.18: InterfaceA.java

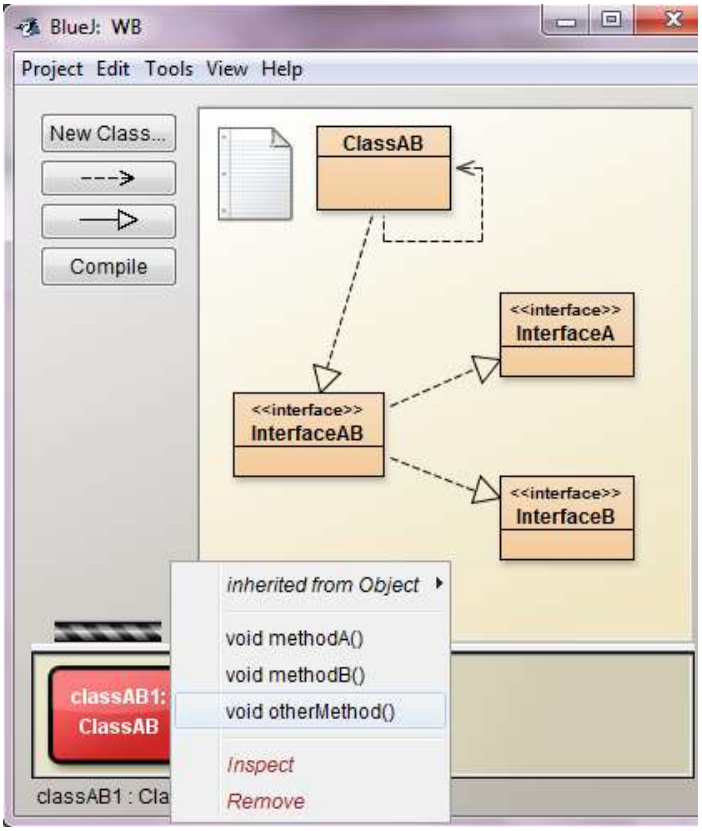
```
1  public interface InterfaceA {
2    public void methodA();
3  }
```

Code listing 4.19: InterfaceB.java

```
1  public interface InterfaceB {
2    public void methodB();
3  }
```

Code listing 4.20: InterfaceAB.java

```
1  public interface InterfaceAB extends InterfaceA,
   InterfaceB {
2    public void otherMethod();
3  }
```



Execution of this example on BlueJ.

This way, a class implementing the `InterfaceAB` interface has to implement the `methodA()`, the `methodB()` and the `otherMethod()` methods:

Code listing 4.21: ClassAB.java

```
1  public class ClassAB implements InterfaceAB {
2    public void methodA() {
3      System.out.println("A");
4    }
5
6    public void methodB() {
7      System.out.println("B");
8    }
9
10   public void otherMethod() {
11     System.out.println("foo");
12   }
13
14   public static void main(String[] args) {
15     ClassAB classAb = new ClassAB();
16     classAb.methodA();
17     classAb.methodB();
18     classAb.otherMethod();
19   }
20 }
```

Doing so, a `ClassAB` object can be casted into `InterfaceA`, `InterfaceB` and `InterfaceAB`.

Test your knowledge

Question 4.2: Consider the following interfaces.

Question 4.2: Walkable.java

```
1  public interface Walkable {
2      void walk();
3  }
```

Question 4.2: Jumpable.java

```
1  public interface Jumpable {
2      void jump();
3  }
```

Question 4.2: Swimable.java

```
1  public interface Swimable {
2      void swim();
3  }
```

Question 4.2: Movable.java

```
1  public interface Movable extends Walkable, Jumpable {
2  }
```

List all the methods that an implementing class of `Movable` should implement.

Answer

- `walk()`
- `jump()`

Answer 4.2: Person.java

```
1  public class Person implements Movable {
2      public void walk() {
3          System.out.println("Do something.");
4      }
5
6      public void jump() {
7          System.out.println("Do something.");
8      }
9  }
```

Question 4.3: Consider the following classes and the following code.

Question 4.3: ConsoleLogger.java

```
1  import java.util.Date;
2
3  public class ConsoleLogger {
4      public void printLog(String log) {
5          System.out.println(new Date() + ": " + log);
6      }
7  }
```

Question 4.3: FileLogger.java

```
1  import java.io.File;
2  import java.io.FileOutputStream;
3
```

```
4  public class FileLogger {
5    public void printLog(String log) {
6      try {
7        File file = new File("log.txt");
8        FileOutputStream stream = new FileOutputStream(file);
9        byte[] logInBytes = (new Date() + ": " + log).getBytes();
10
11       stream.write(logInBytes);
12
13       stream.flush();
14       stream.close();
15     } catch (Exception e) {
16       e.printStackTrace();
17     }
18   }
19 }
```

Question 4.3: Common code.

```
1  Object[] loggerArray = new Object[2];
2  loggerArray[0] = new ConsoleLogger();
3  loggerArray[1] = new FileLogger();
4
5  for (Object logger : loggerArray) {
6      // logger.printLog("Check point.");
7  }
```

Change the implementation of the code in order to be able to uncomment the commented line without compile error.

Answer

You have to create an interface that defines the method `printLog(String)` and makes `ConsoleLogger` and `FileLogger` implement it:

Answer 4.3: Logger.java

```
1  public interface Logger {
2      void printLog(String log);
3  }
```

Answer 4.3: ConsoleLogger.java

```
1  import java.util.Date;
2
3  public class ConsoleLogger implements Logger {
4      public void printLog(String log) {
5          System.out.println(new Date() + ": " + log);
6      }
7  }
```

Answer 4.3: FileLogger.java

```
1  import java.io.File;
2  import java.io.FileOutputStream;
3
4  public class FileLogger implements Logger {
5    public void printLog(String log) {
6      try {
7        File file = new File("log.txt");
8        FileOutputStream stream = new FileOutputStream(file);
9        byte[] logInBytes = (new Date() + ": " + log).getBytes();
10
11       stream.write(logInBytes);
12
13       stream.flush();
14       stream.close();
15     } catch (Exception e) {
16       e.printStackTrace();
17     }
18   }
19 }
```

Now your code has to cast the objects to the `Logger` type and then you can uncomment the code.

Answer 4.3: Common code.

```
Logger[] loggerArray = new Logger[2];
loggerArray[0] = new ConsoleLogger();
loggerArray[1] = new FileLogger();

for (Logger logger : loggerArray) {
    logger.printLog("Check point.");
}
```

Overloading Methods and Constructors

Method overloading

In a class, there can be several methods with the same name. However they must have a different *signature*. The signature of a method is comprised of its name, its parameter types and the order of its parameters. The signature of a method is **not** comprised of its return type nor its visibility nor the exceptions it may throw. The practice of defining two or more methods within the same class that share the same name but have different parameters is called *overloading methods*.

Methods with the same name in a class are called *overloaded methods*. Overloading methods offers no specific benefit to the JVM but it is useful to the programmer to have several methods do the same things but with different parameters. For example, we may have the operation `runAroundThe` represented as two methods with the same name, but different input parameter types:

Code section 4.22: Method overloading.

```
public void runAroundThe(Building block) {
  ...
}

public void runAroundThe(Park park) {
  ...
}
```

One type can be the subclass of the other:

Code listing 4.11: ClassName.java

```
public class ClassName {

  public static void sayClassName(Object aObject) {
    System.out.println("Object");
  }

  public static void sayClassName(String aString) {
    System.out.println("String");
  }

  public static void main(String[] args) {
    String aString = new String();
    sayClassName(aString);

    Object aObject = new String();
    sayClassName(aObject);
  }
}
```

Console for Code listing 4.11

```
String
Object
```

Although both methods would be fit to call the method with the `String` parameter, it is the method with the nearest type that will be called instead. To be more accurate, it will call the method whose parameter type is a subclass of the parameter type of the other method. So, `aObject` will output `Object`. Beware! The parameter type is defined by the

declared type of an object, **not** its *instantiated* type!

The following two method definitions are valid

Code section 4.23: Method overloading with the type order.

```
1  public void logIt(String param, Error err) {
2    ...
3  }
4
5  public void logIt(Error err, String param) {
6    ...
7  }
```

because the type order is different. If both input parameters were type String, that would be a problem since the compiler would not be able to distinguish between the two:

Code section 4.24: Bad method overloading.

```
1  public void logIt(String param, String err) {
2    ...
3  }
4
5  public void logIt(String err, String param) {
6    ...
7  }
```

The compiler would give an error for the following method definitions as well:

Code section 4.25: Another bad method overloading.

```
1  public void logIt(String param) {
2    ...
3  }
4
5  public String logIt(String param) {
6    String retValue;
7    ...
8    return retValue;
9  }
```

Note, the return type is not part of the unique signature. Why not? The reason is that a method can be called without assigning its return value to a variable. This feature came from C and C++. So for the call:

Code section 4.26: Ambiguous method call.

```
1  logIt(msg);
```

the compiler would not know which method to call. It is also the case for the thrown exceptions.

Test your knowledge

Question 4.6: Which methods of the `Question6` class will cause compile errors?

`Question6.java`

```
1  public class Question6 {
2
3    public void example1() {
4    }
5
6    public int example1() {
7    }
8
9    public void example2(int x) {
10   }
11
12   public void example2(int y) {
```

```
  }

  private void example3() {
  }

  public void example3() {
  }

  public String example4(int x) {
    return null;
  }

  public String example4() {
    return null;
  }
}
```

Answer

Question6.java

```
public class Question6 {

  public void example1() {
  }

  public int example1() {
  }

  public void example2(int x) {
  }

  public void example2(int y) {
  }

  private void example3() {
  }

  public void example3() {
  }

  public String example4(int x) {
    return null;
  }

  public String example4() {
    return null;
  }
}
```

The `example1`, `example2` and `example3` methods will cause compile errors. The `example1` methods cannot co-exist because they have the same signature (remember, return type is **not** part of the signature). The `example2` methods cannot co-exist because the names of the parameters are not part of the signature. The `example3` methods cannot co-exist because the visibility of the methods are not part of the signature. The `example4` methods can co-exist, because they have different method signatures.

Variable Argument

Instead of overloading, you can use a dynamic number of arguments. After the last parameter, you can pass optional unlimited parameters of the same type. These parameters are defined by adding a last parameter and adding `...` after its type. The dynamic arguments will be received as an array:

Code section 4.27: Variable argument.

```
public void registerPersonInAgenda(String firstName, String lastName, Date... meeting) {
  String[] person = {firstName, lastName};
  lastPosition = lastPosition + 1;
  contactArray[lastPosition] = person;

  if (meeting.length > 0) {
    Date[] temporaryMeetings = new Date[registeredMeetings.length + meeting.length];
    for (i = 0; i < registeredMeetings.length; i++) {
      temporaryMeetings[i] = registeredMeetings[i];
    }
    for (i = 0; i < meeting.length; i++) {
      temporaryMeetings[registeredMeetings.length + i] = meeting[i];
```

```
        }
        registeredMeetings = temporaryMeetings;
      }
    }
```

The above method can be called with a dynamic number of arguments, for example:

Code section 4.27: Constructor calls.

```
registerPersonInAgenda("John", "Doe");
registerPersonInAgenda("Mark", "Lee", new Date(), new Date());
```

This feature was not available before Java 1.5 .

Constructor overloading

The constructor can be overloaded. You can define more than one constructor with different parameters. For example:

Code listing 4.12: Constructors.

```
public class MyClass {

  private String memberField;

  /**
   * MyClass Constructor, there is no input parameter
   */
  public MyClass() {
     ...
  }

  /**
   * MyClass Constructor, there is one input parameter
   */
   public MyClass(String param1) {
     memberField = param1;
     ...
  }
}
```

In the code listing 4.12, we defined two constructors, one with no input parameter, and one with one input parameter. You may ask which constructor will be called. Its depends how the object is created with the **`new`** keyword. See below:

Code section 4.29: Constructor calls.

```
// The constructor with no input parameter will be called
MyClass obj1 = new MyClass();

// The constructor with one input param. will be called
MyClass obj2 = new MyClass("Init Value");
```

In the code section 4.29, we created two objects from the same class, or we can also say that `obj1` and `obj2` both have the same type. The difference between the two is that in the first one the `memberField` field is not initialized, in the second one that is initialized to `"Init Value"`. A constructor may also be called from another constructor, see below:

Code listing 4.13: Constructor pooling.

```
public class MyClass {

  private String memberField;

  /**
   * MyClass Constructor, there is no input parameter
   */
  public MyClass() {
    MyClass("Default Value");
```

```
  }

  /**
   * MyClass Constructor, there is one input parameter
   */
  public MyClass(String param1) {
    memberField = param1;
    ...
  }
}
```

In the code listing 4.13, the constructor with no input parameter calls the other constructor with the default initial value. This call must be the first instruction of a constructor or else a compiler error will occur. The code gives an option to the user, to create the object with the default value or create the object with a specified value. The first constructor could have been written using the **`this`** keyword as well:

Code section 4.30: Another constructor pooling.

```
  public MyClass() {
    this("Default Value");
  }
```

Such a call reduces the code repetition.

Method overriding

To easily remember what can be done in method overriding, keep in mind that all you can do in an object of a class, you can also do in an object of a subclass, only the behavior can change. A subclass should be *covariant*.

Although a method signature has to be unique inside a class, the same method signature can be defined in different classes. If we define a method that exists in the super class then we override the super class method. It is called *method overriding*. This is different from method overloading. Method overloading happens with methods with the same name but different signature. Method overriding happens with methods with the same name and same signature between inherited classes.

The return type can cause the same problem we saw above. When we override a super class method the return type also must be the same. If that is not the same, the compiler will give you an error.

Beware! If a class declares two public methods with the same name, and a subclass overrides one of them, the subclass still inherits the other method. In this respect, the Java programming language differs from C++.

Method overriding is related to *dynamic linking*, or *runtime binding*. In order for the Method Overriding to work, the method call that is going to be called can not be determined at compilation time. It will be decided at runtime, and will be looked up in a table.

Code section 4.31: Runtime binding.

```
MyClass obj;

if (new java.util.Calendar().get(java.util.Calendar.AM_PM) == java.util.Calendar.AM) {
  // Executed during a morning
  obj = new SubOfMyClass();
} else {
  // Executed during an afternoon
  obj = new MyClass();
}

obj.myMethod();
```

In the code section 4.31, the expression at line 3 is true if it is executed during a morning and false if it is executed during an afternoon. Thus, the instance of `obj` will be a `MyClass` or a `SubOfMyClass` depending on the execution time. So it is impossible to determine the method address at compile time. Because the `obj` reference can point to an object and all its sub objects, and that will be known only at runtime, a table is kept with all the possible method addresses to be called. Do not confuse:

Code section 4.32: Declared type and instantiated type.

```
obj.myMethod(myParameter);
```

The implementation of this method is searched using the instantiated type of the called object (`obj`) and the declared type of the parameter object (`myParameter`).

Also another rule is that when you do an override, the visibility of the new method that overrides the super class method can not be reduced. The visibility can be increased, however. So if the super class method visibility is **`public`**, the override method can not be **`package`**, or **`private`**. An override method must throw the same exceptions as the super class, or their subexceptions.

`super` references to the parent class (i.e. `super.someMethod()`). It can be used in a subclass to access inherited methods that the subclass has overridden or inherited fields that the subclass has hidden.

A common mistake is to think that if we can override methods, we could also override member variables. This is not the case, as it is useless. You can not redefine a variable that is private in the super class as such a variable is not visible.

Object Lifecycle

Before a Java object can be created the class byte code must be loaded from the file system (with `.class` extension) to memory. This process of locating the byte code for a given class name and converting that code into a Java class instance is known as *class loading*. There is one class created for each type of Java class.

All objects in Java programs are created on heap memory. An object is created based on its class. You can consider a class as a blueprint, template, or a description how to create an object. When an object is created, memory is allocated to hold the object properties. An object reference pointing to that memory location is also created. To use the object in the future, that object reference has to be stored as a local variable or as an object member variable.

The Java Virtual Machine (JVM) keeps track of the usage of object references. If there are no more reference to the object, the object can not be used any more and becomes garbage. After a while the heap memory will be full of unused objects. The JVM collects those garbage objects and frees the memory they allocated, so the memory can be reused again when a new object is created. See below a simple example:

Code section 4.30: Object creation.

```
{
  // Create an object
  MyObject obj = new MyObject();

  // Use the object
  obj.printMyValues();
}
```

The `obj` variable contains the object reference pointing to an object created from the `MyObject` class. The `obj` object reference is in scope inside the `{ }`. After the `}` the object becomes garbage. Object references can be passed in to methods and can be returned from methods.

Creating object with the `new` keyword

99% of new objects are created using the **`new`** keyword.

Code listing 4.13: MyProgram.java

```
public class MyProgram {

  public static void main(String[] args) {
    // Create 'MyObject' for the first time the application is started
    MyObject obj = new MyObject();
```

```
   }
}
```

When an object from the `MyObject` class is created for the first time, the JVM searches the file system for the definition of the class, that is the Java byte code. The file has the extension of `*.class`. The **CLASSPATH** environment variable contains locations where Java classes are stored. The JVM is looking for the `MyObject.class` file. Depending on which package the class belongs to, the package name will be translated to a directory path.

When the `MyObject.class` file is found, the JVM's class loader loads the class in memory, and creates a java.lang.Class object. The JVM stores the code in memory, allocates memory for the **`static`** variables, and executes any static initialize block. Memory is not allocated for the object member variables at this point, memory will be allocated for them when an instance of the class, an object, is created.

There is no limit on how many objects from the same class can be created. Code and **`static`** variables are stored only once, no matter how many objects are created. Memory is allocated for the object member variables when the object is created. Thus, the size of an object is determined not by its code's size but by the memory it needs for its member variables to be stored.

Creating object by cloning an object

Cloning is not automatically available to classes. There is some help though, as all Java objects inherit the `protected Object clone()` method. This base method would allocate the memory and do the bit by bit copying of the object's states.

You may ask why we need this clone method. Can't we create a constructor, pass in the same object and do the copying variable by variable? An example would be (note that accessing the private `memberVar` variable of `obj` is legal as this is in the same class):

Code listing 4.14: MyObject.java

```
public class MyObject {
   private int memberVar;
...
   MyObject(MyObject obj) {
      this.memberVar = obj.memberVar;
    ...
   }
...
}
```

This method works but object creation with the **`new`** keyword is time-consuming. The `clone()` method copies the whole object's memory in one operation and this is much faster than using the new keyword and copying each variable so if you need to create lots of objects with the same type, performance will be better if you create one object and clone new ones from it. See below a factory method that will return a new object using cloning.

Code section 4.31: Object cloning.

```
HashTable cacheTemplate = new HashTable();
...
/** Clone Customer object for performance reason */
public Customer createCustomerObject() {
   // See if a template object exists in our cache
   Customer template = cacheTemplate.get("Customer");
   if (template == null) {
      // Create template
      template = new Customer();
      cacheTemplate.put("Customer", template);
   }
   return template.clone();
}
```

Now, let's see how to make the Customer object cloneable.

1. First the `Customer` class needs to implement the `Cloneable` Interface.
2. Override and make the `clone()` method **`public`**, as that is **`protected`** in the Object class.

3. Call the `super.clone()`method at the beginning of your `clone` method.
4. Override the `clone()` method in all the subclasses of `Customer`.

Code listing 4.15: Customer.java

```
1  public class Customer implements Cloneable {
2   ...
3      public Object clone() throws CloneNotSupportedException {
4         Object obj = super.clone();
5
6         return obj;
7      }
8  }
```

In the code listing 4.15 we used cloning for speed up object creation. Another use of cloning could be to take a snapshot of an object that can change in time. Let's say we want to store Customer objects in a collection, but we want to disassociate them from the 'live' objects. So before adding the object, we clone them, so if the original object changes from that point forward, the added object won't. Also let's say that the Customer object has a reference to an Activity object that contains the customer activities. Now we are facing a problem, it is not enough to clone the Customer object, we also need to clone the referenced objects. The solution:

1. Make the Activity class also cloneable
2. Make sure that if the Activity class has other 'changeable' object references, those have to be cloned as well, as seen below
3. Change the Customer class `clone()` method as follows:

Code listing 4.16: Customer.java

```
1  public class Customer implements Cloneable {
2    Activity activity;
3    ...
4      public Customer clone() throws CloneNotSupportedException {
5        Customer clonedCustomer = (Customer) super.clone();
6
7        // Clone the object referenced objects
8        if (activity != null) {
9           clonedCustomer.setActivity((Activity) activity.clone());
10       }
11       return clonedCustomer;
12     }
13 }
```

Note that only mutable objects need to be cloned. References to unchangeable objects such as a String can be used in the cloned object without worry.

Re-creating an object received from a remote source

When an object is sent through a network, the object needs to be **re-created** at the receiving host.

Object Serialization

The term *Object Serialization* refers to the act of converting the object to a byte stream. The byte stream can be stored on the file system or can be sent through a network.
At a later time the object can be re-created from that stream of bytes. The only requirement is that the same class has to be available both times, when the object is serialized and also when the object is re-created. If that happens on different servers, then the same class must be available on both servers. Same class means that exactly the same version of the class must be available, otherwise the object won't be able to be re-created.
This is a maintenance problem for those applications where java serialization is used to make objects persistent or to sent the object through the network.
When a class is modified, there could be a problem re-creating those objects that were serialized using an earlier version of the class.

Java has built-in support for serialization, using the `Serializable` interface; however, a class must first implement the `Serializable` interface.

By default, a class will have all of its fields serialized when converted into a data stream (with **`transient`** fields being skipped). If additional handling is required beyond the default of writing all fields, you need to provide an implementation for the following three methods:

```
private void writeObject(java.io.ObjectOutputStream out) throws IOException;

private void readObject(java.io.ObjectInputStream in) throws IOException,
ClassNotFoundException;

private void readObjectNoData() throws ObjectStreamException;
```

If the object needs to write or provide a replacement object during serialization, it needs to implement the following two methods, with any access specifier:

```
Object writeReplace() throws ObjectStreamException;

Object readResolve() throws ObjectStreamException;
```

Normally, a minor change to the class can cause the serialization to fail. You can still allow the class to be loaded by defining the serialization version id:

Code section 4.32: Serialization version id.

```
1  private static final long serialVersionUID = 42L;
```

Destroying objects

Unlike in many other object-oriented programming languages, Java performs automatic garbage collection — any unreferenced objects are automatically erased from memory — and prohibits the user from manually destroying objects.

finalize()

When an object is garbage-collected, the programmer may want to manually perform cleanup, such as closing any open input/output streams. To accomplish this, the `finalize()` method is used. Note that `finalize()` should never be manually called, except to call a super class' finalize method from a derived class' finalize method. Also, we can not rely on when the `finalize()` method will be called. If the java application exits before the object is garbage-collected, the `finalize()` method may never be called.

Code section 4.33: Finalization.

```
1  protected void finalize() throws Throwable {
2    try {
3      doCleanup();        // Perform some cleanup.  If it fails for some reason, it is ignored.
4    } finally {
5      super.finalize(); // Call finalize on the parent object
6    }
7  }
```

The garbage-collector thread runs in a lower priority than the other threads. If the application creates objects faster than the garbage-collector can claim back memory, the program can run out of memory.

The finalize method is required only if there are resources beyond the direct control of the Java Virtual Machine that needs to be cleaned up. In particular, there is no need to explicitly close an OutputStream, since the OutputStream will close itself when it gets finalized. Instead, the finalize method is used to release either native or remote resources controlled by the class.

Class loading

One of the main concerns of a developer writing hot re-deployable applications is to understand how class loading works. Within the internals of the class loading mechanism lies the answer to questions like:

- What happens if I pack a newer version of an utility library with my application, while an older version of the same library lingers somewhere in the server's lib directory?

- How can I use two different versions of the same utility library, simultaneously, within the same instance of the application server?
- What version of an utility class I am currently using?
- Why do I need to mess with all this class loading stuff anyway?

Scope

Scope

The scope of a class, a variable or a method is its visibility and its accessibility. The visibility or accessibility means that you can use the item from a given place.

Scope of method parameters

A method parameter is visible inside of the entire method but not visible outside the method.

Code listing 3.14: Scope.java

```
1  public class Scope {
2
3      public void method1(int i) {
4          i = i++;
5          method2();
6          int j = i * 2;
7      }
8
9      public void method2() {
10         int k = 20;
11     }
12
13     public static void main(String[] args) {
14         method1(10);
15     }
16 }
```

In code listing 3.14, `i` is visible within the entire `method1` method but not in the `method2` and the `main` methods.

Scope of local variables

A local variable is visible after its declaration until the end of the block in which the local variable has been created.

Code section 3.50: Local variables.

```
1  {
2  ...
3       // myNumber is NOT visible
4    {
5       // myNumber is NOT visible
6       int myNumber;
7       // myNumber is visible
8      {
9        ...
10       // myNumber is visible
11     }
12       // myNumber is visible
13   }
14      // myNumber is NOT visible
15 ...
16 }
```

Access modifiers

You surely would have noticed by now, the words `public`, `protected` and `private` at the beginning of class's method declarations used in this book. These keywords are called the **access modifiers** in the Java language syntax, and they define the scope of a given item.

For a class

- If a class has **public** visibility, the class can be referenced by anywhere in the program.
- If a class has **protected** visibility, the class can be referenced only in the package where the class is defined.
- If a class has **private** visibility, (it can happen only if the class is defined nested in another class) the class can be accessed only in the outer class.

For a variable

- If a variable is defined in a **public** class and it has **public** visibility, the variable can be referenced anywhere in the application through the class it is defined in.
- If a variable has **protected** visibility, the variable can be referenced only in the sub-classes and in the same package through the class it is defined in.
- If a variable has **package** visibility, the variable can be referenced only in the same package through the class it is defined in.
- If a variable has **private** visibility, the variable can be accessed only in the class it is defined in.

For a method

- If a method is defined in a **public** class and it has **public** visibility, the method can be called anywhere in the application through the class it is defined in.
- If a method has **protected** visibility, the method can be called only in the sub-classes and in the same package through the class it is defined in.
- If a method has **package** visibility, the method can be called only in the same package through the class it is defined in.
- If a method has **private** visibility, the method can be called only in the class it is defined in.

For an interface

The interface methods and interfaces are always `public`. You do not need to specify the access modifier. It will default to `public`. For clarity it is considered a good practice to put the `public` keyword.

The same way all member variables defined in the Interface by default will become `static` `final` once inherited in a class.

Summary

	Class	Nested class	Method, or Member variable	Interface	Interface method signature
`public`	visible from anywhere	same as its class	same as its class	**visible from anywhere**	**visible from anywhere**
`protected`	N/A	its class and its subclass	its class and its subclass, and from its package	N/A	N/A
package	**only from its package**	**only from its package**	**only from its package**	N/A	N/A
`private`	N/A	only from its class	only from its class	N/A	N/A

The cases in bold are the default.

Utility

A general guideline for visibilities is to only make a member as visible as it needs to be. Don't make a member public if it only needs to be private.

Doing so, you can rewrite a class and change all the private members without making compilation errors, even you don't know all the classes that will use your class as long as you do not change the signature of the public members.

Field encapsulation

Generally, it is best to make data private or protected. Access to the data is controlled by *setter* and *getter* methods. This lets the programmer control access to data, allowing him/her to check for and handle invalid data.

Code section 3.51: Encapsulation.

```
1  private String name;
2
3  /**
4   * This is a getter method because it accesses data from the object.
5   */
6  public String getName() {
7    return name;
8  }
9
10 /**
11  * This is a setter method because it changes data in the object.
12  */
13 public boolean setName(String newName) {
14   if (newName == null) {
15     return false;
16   } else {
17     name = newName;
18     return true;
19   }
20 }
```

In the code section 3.51, the `setName()` method will only change the value of `name` if the new name is not null. Because `setName()` is conditionally changing name, it is wise to return a boolean to let the program know if the change was successful.

Test your knowledge

Question 3.15: Consider the following class.

Question 3.15: Question15.java

```
1  public class Question15 {
2
3      public static final int QKQKQKQK_MULTIPLIER = 2;
4
5      public int ijijijijijijijijAwfulName = 20;
6
7      private int ununununununununununCrummyName = 10;
8
9      private void mememememememeUglyName(int i) {
10         i = i++;
11         tltltltltltltltltltlBadName();
12         int j = i * QKQKQKQK_MULTIPLIER;
13     }
14
15     public void tltltltltltltltltltlBadName() {
16         int k = ijijijijijijijijAwfulName;
17     }
18
19     public static void main(String[] args) {
20         mememememememeUglyName(ununununununununununCrummyName);
21     }
22 }
```

List the fields and methods of this class that can be renamed without changing or even knowing the client classes.

Answer

1. `ununununununununununCrummyName`
2. `mememememememeUglyName()`

Every field or method that is public can be directly called by a client class so this class would return a compile error if the field or the method has a new name.

Nested Classes

In Java you can define a class inside another class. A class can be nested inside another class or inside a method. A class that is not nested is called a *top-level* class and a class defining a nested class is an *outer* class.

Inner classes

Nesting a class inside a class

When a class is declared inside another class, the nested class' access modifier can be **`public`**, **`private`**, **`protected`** or `package(default)`.

Code listing 4.10: OuterClass.java

```
1  public class OuterClass {
2     private String outerInstanceVar;
3
4     public class InnerClass {
5        public void printVars() {
6           System.out.println("Print Outer Class Instance Var.:" + outerInstanceVar);
7        }
8     }
9  }
```

The inner class has access to the enclosing class instance's variables and methods, even private ones, as seen above. This makes it very different from the nested class in C++, which are equivalent to the "static" inner classes, see below.

An inner object has a reference to the outer object. In other words, all inner objects are tied to the outer object. The inner object can only be created through a reference to the 'outer' object. See below.

Code section 4.20: Outer class call.

```
1  public void testInner() {
2      ...
3      OuterClass outer = new OuterClass();
4      OuterClass.InnerClass inner = outer.new InnerClass();
5      ...
6  }
```

Note that inner objects, because they are tied to the outer object, cannot contain static variables or methods.

When in a non-static method of the outer class, you can directly use `new InnerClass()`, since the class instance is implied to be `this`.

You can directly access the reference to the outer object from within an inner class with the syntax `OuterClass.this`; although this is usually unnecessary because you already have access to its fields and methods.

Inner classes compile to separate ".class" bytecode files, with the name of the enclosing class, followed by a "$", followed by the name of the inner class. So for example, the above inner class would be compiled to a file named "OuterClass$InnerClass.class".

Static nested classes

A nested class can be declared *static*. These classes are not bound to an instance of the outer defining class. A static nested class has no enclosing instance, and therefore cannot access instance variables and methods of the outer class. You do not specify an instance when creating a static inner class. This is equivalent to the inner classes in C++.

Nesting a class inside a method

These inner classes, also called *local classes*, cannot have access modifiers, like local variables, since the class is 'private' to the method. The inner class can be only **`abstract`** or **`final`**.

Code listing 4.11: OuterClass.java

```
1  public class OuterClass {
2     public void method() {
3        class InnerClass {
4
5        }
6     }
7  }
```

In addition to instance variables of the enclosing class, local classes can also access local variables of the enclosing method, but only ones that are declared `final`. This is because the local class instance might outlive the invocation of the method, and so needs its own copy of the variable. To avoid problems with having two different copies of a mutable variable with the same name in the same scope, it is required to be `final`, so it cannot be changed.

Anonymous Classes

In Java, a class definition and its instantiation can be combined into a single step. By doing that the class does not require a name. Those classes are called anonymous classes. An anonymous class can be defined and instantiated in contexts where a reference can be used, and it is a nested class to an existing class. Anonymous class is a special case of a class local to a method; hence they also can access final local variables of the enclosing method.

Anonymous classes are most useful to create an instance of an interface or adapter class without needing a brand new class.

Code listing 4.12: ActionListener.java

```
1  public interface ActionListener {
2      public void actionPerformed();
3  }
```

Code section 4.21: Anonymous class.

```
1  ActionListener listener = new ActionListener() {
2          public void actionPerformed() {
3              // Implementation of the action event
4              ...
5              return;
6          }
7      };
```

In the above example the class that implements the `ActionListener` is **anonymous**. The class is defined where it is instantiated.

The above code is harder to read than if the class is explicitly defined, so why use it? If many implementations are needed for an interface, those classes are used only in one particular place, and it would be hard to come up with names for them, using an **anonymous** inner class makes sense.

The following example uses an anonymous inner class to implement an action listener.

Code listing 4.13: MyApp.java

```
1  import java.awt.Button;
2  import java.awt.event.ActionEvent;
```

```
import java.awt.event.ActionListener;

class MyApp {
   Button aButton = new Button();

   MyApp() {
       aButton.addActionListener(new ActionListener() {
               public void actionPerformed(ActionEvent e) {
                   System.out.println("Hello There");
               }
           }
       );
   }
}
```

The following example does the same thing, but it names the class that implements the action listener.

Code listing 4.14: MyApp.java

```
import java.awt.Button;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

class MyApp {
   Button aButton = new Button();

   // Nested class to implement the action listener
   class MyActionListener implements ActionListener {
       public void actionPerformed(ActionEvent e) {
           System.out.println("Hello There");
       }
   }
   MyApp() {
       aButton.addActionListener(new MyActionListener());
   }
}
```

Using **anonymous** classes is especially preferable when you intend to use many different classes that each implement the same interface.

Generics

Java is a strongly typed language, so a field in a class may be typed like this:

Code listing 4.34: Repository.java

```
public class Repository {

   public Integer item;

   public Integer getItem() {
      return item;
   }

   public void setItem(Integer newItem) {
      item = newItem;
   }
}
```

This ensures that, only `Integer` objects can be put in the field and a `ClassCastException` can't occur at runtime, only compile-time error can occur. Unfortunately, it can be used only with `Integer` objects. If you want to use the same class in another context with `Strings`, you have to generalize the type like this:

Code listing 4.35: Repository.java

```
public class Repository {

```

```
    public Object item;

    public Object getItem() {
        return item;
    }

    public void setItem(Integer newItem) {
        item = newItem;
    }

    public void setItem(String newItem) {
        item = newItem;
    }
}
```

But you will have `ClassCastException` at runtime again and you can't easily use your field. The solution is to use Generics.

Generic class

A generic class does not hard code the type of a field, a return value or a parameter. The class only indicates that a generic type should be the same, for a given object instance. The generic type is not specified in the class definition. It is specified during object instantiation. This allows the generic type to be different from an instance to another. So we should write our class this way:

Code listing 4.36: Repository.java

```
public class Repository<T> {

    public T item;

    public T getItem() {
        return item;
    }

    public void setItem(T newItem) {
        item = newItem;
    }
}
```

Here, the generic type is defined after the name of the class. Any new identifier can be chosen. Here, we have chosen *T*, which is the most common choice. The actual type is defined at the object instantiation:

Code section 4.35: Instantiation.

```
Repository<Integer> arithmeticRepository = new Repository<Integer>();
arithmeticRepository.setItem(new Integer(1));
Integer number = arithmeticRepository.getItem();

Repository<String> textualRepository = new Repository<String>();
textualRepository.setItem("Hello!");
String message = textualRepository.getItem();
```

Although each object instance has its own type, each object instance is still strongly typed:

Code section 4.36: Compile error.

```
Repository<Integer> arithmeticRepository = new Repository<Integer>();
arithmeticRepository.setItem("Hello!");
```

A class can define as many generic types as you like. Choose a different identifier for each generic type and separate them by a comma:

Code listing 4.37: Repository.java

```
public class Repository<T, U> {

    public T item;
```

```
4
5     public U anotherItem;
6
7     public T getItem() {
8        return item;
9     }
10
11    public void setItem(T newItem) {
12       item = newItem;
13    }
14
15    public U getAnotherItem() {
16       return anotherItem;
17    }
18
19    public void setAnotherItem(U newItem) {
20       anotherItem = newItem;
21    }
22 }
```

When a type that is defined with generic (for example, `Collection<T>`) is not used with generics (for example, `Collection`) is called a *raw type*.

Generic method

A generic type can be defined for just a method:

Code section 4.37: Generic method.

```
1  public <D> D assign(Collection<D> generic, D obj) {
2    generic.add(obj);
3    return obj;
4  }
```

Here a new identifier (*D*) has been chosen at the beginning of the method declaration. The type is *specific to a method call* and different types can be used for the same object instance:

Code section 4.38: Generic method call.

```
1  Collection<Integer> numbers = new ArrayList<Integer>();
2  Integer number = assign(numbers, new Integer(1));
3  Collection<String> texts = new ArrayList<String>();
4  String text = assign(texts, "Store it.");
```

The actual type will be defined by the type of the method parameter. Hence, the generic type can't be defined only for the return value as it wouldn't be resolved. See the Class<T> section for a solution.

Test your knowledge

Question 4.8: Consider the following class.

Question 4.8: Question8.java

```
1  public class Question8<T> {
2    public T item;
3
4    public T getItem() {
5      return item;
6    }
7
8    public void setItem(T newItem) {
9      item = newItem;
10   }
11
12   public static void main(String[] args) {
13     Question8<String> aQuestion = new Question8<String>();
14     aQuestion.setItem("Open your mind.");
15     aQuestion.display(aQuestion.getItem());
16   }
17
18   public void display(String parameter) {
19     System.out.println("Here is the text: " + parameter);
```

```
20    }
21
22    public void display(Integer parameter) {
23      System.out.println("Here is the number: " + parameter);
24    }
25
26    public void display(Object parameter) {
27      System.out.println("Here is the object: " + parameter);
28    }
29  }
```

What will be displayed on the console?

Answer

Console for Answer 4.8

```
Here is the text: Open your mind.
```

`aQuestion.getItem()` is typed as a string.

Wildcard Types

As we have seen above, generics give the impression that a new container type is created with each different type parameter. We have also seen that in addition to the normal type checking, the type parameter has to match as well when we assign generics variables. In some cases this is too restrictive. What if we would like to relax this additional checking? What if we would like to define a collection variable that can hold any generic collection, regardless of the parameter type it holds? The wildcard type is represented by the character **<?>**, and pronounced **Unknown**, or **Any-Type**. Any-Type can be expressed also by `<? extends Object>`. Any-Type includes Interfaces, not only Classes. So now we can define a collection whose element type matches anything. See below:

Code section 4.39: Wildcard type.

```
1  Collection<?> collUnknown;
```

Upper bounded wildcards

You can specify a restriction on the types of classes that may be used. For example, `<? extends ClassName>` only allows objects of class `ClassName` or a subclass. For example, to create a collection that may only contain "Serializable" objects, specify:

Code section 4.40: Collection of serializable subobjects.

```
1  Collection<String> textColl = new ArrayList<String>();
2
3  Collection<? extends Serializable> serColl = textColl;
```

The above code is valid because the `String` class is serializable. Use of a class that is not serializable would cause a compilation error. The added items can be retrieved as `Serializable` object. You can call methods of the `Serializable` interface or cast it to `String`. The following collection can only contain objects that extend the class `Animal`.

Code listing 4.38: Dog.java

```
1  class Dog extends Animal {
2  }
```

Code section 4.41: Example of subclass.

```
// Create "Animal Collection" variable
Collection<? extends Animal> animalColl = new ArrayList<Dog>();
```

Lower bounded wildcards

`<? super ClassName>` specifies a restriction on the types of classes that may be used. For example, to declare a Comparator that can compare Dogs, you use:

Code section 4.42: Superclass.

```
Comparator<? super Dog> myComparator;
```

Now suppose you define a comparator that can compare Animals:

Code section 4.43: Comparator.

```
class AnimalComparator implements Comparator<Animal> {
  int compare(Animal a, Animal b) {
   //...
  }
}
```

Since `Dogs` are `Animals`, you can use this comparator to compare Dogs also. Comparators for any superclass of Dog can also compare Dog; but comparators for any strict subclass cannot.

Code section 4.44: Generic comparator.

```
Comparator<Animal> myAnimalComparator = new AnimalComparator();

static int compareTwoDogs(Comparator<? super Dog> comp, Dog dog1, Dog dog2) {
  return comp.compare(dog1, dog2);
}
```

The above code is valid because the `Animal` class is a supertype of the `Dog` class. Use of a class that is not a supertype would cause a compilation error.

Unbounded wildcard

The advantage of the unbounded wildcard (i.e. `<?>`) compared to a raw type (i.e. without generic) is to explicitly say that the parameterized type is unknown, not *any type*. That way, all the operations that implies to know the type are forbidden to avoid unsafe operation. Consider the following code:

Code section 4.45: Unsafe operation.

```
public void addAtBottom(Collection anyCollection) {
  anyCollection.add(new Integer(1));
}
```

This code will compile but this code may corrupt the collection if the collection only contains strings:

Code section 4.46: Corruption of list.

```
List<String> col = new
ArrayList<String>();
addAtBottom(col);
col.get(0).endsWith(".");
```

Console for Code section 4.46

```
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer
incompatible with java.lang.String
at Example.main(Example.java:17)
```

This situation could have been avoided if the `addAtBottom(Collection)` method was defined with an unbounded wildcard: `addAtBottom(Collection<?>)`. With this signature, it is impossible to compile a code that is dependent of the parameterized type. Only independent methods of a collection (`clear()`, `isEmpty()`, `iterator()`, `remove(Object o)`, `size()`, ...) can be called. For instance, `addAtBottom(Collection<?>)` could contain the following code:

Code section 4.47: Safe operation.

```
1  public void addAtBottom(Collection<?> anyCollection) {
2     Iterator<?> iterator = anyCollection.iterator();
3     while (iterator.hasNext()) {
4        System.out.print(iterator.next());
5     }
6  }
```

Class<T>

Since Java 1.5, the class `java.lang.Class` is generic. It is an interesting example of using generics for something other than a container class. For example, the type of String.class is `Class<String>`, and the type of Serializable.class is `Class<Serializable>`. This can be used to improve the type safety of your reflection code. In particular, since the `newInstance()` method in Class now returns T, you can get more precise types when creating objects reflectively. Now we can use the `newInstance()` method to return a new object with exact type, without casting. An example with generics:

Code section 4.48: Automatic cast.

```
1  Customer cust = Utility.createAnyObject(Customer.class);  // No casting
2  ...
3  public static <T> T createAnyObject(Class<T> cls) {
4      T ret = null;
5      try {
6          ret = cls.newInstance();
7      } catch (Exception e) {
8          // Exception Handling
9      }
10     return ret;
11 }
```

The same code without generics:

Code section 4.49: Former version.

```
1  Customer cust = (Customer) Utility.createAnyObject(Customer.class);  // Casting is needed
2  ...
3  public static Object createAnyObject(Class cls) {
4      Object ret = null;
5      try {
6          ret = cls.newInstance();
7      } catch (Exception e) {
8          // Exception Handling
9      }
10     return ret;
11 }
```

Motivation

Java was long criticized for the need to explicitly type-cast an element when it was taken out of a "container/collection" class. There was no way to enforce that a "collection" class contains only one type of object (e.g., to forbid *at compile time* that an `Integer` object is added to a `Collection` that should only contain `Strings`). This is possible since Java 1.5. In the first couple of years of Java evolution, Java did not have a real competitor. This has changed by the appearance of Microsoft C#. With Generics Java is better suited to compete against C#. Similar constructs to Java Generics exist in other languages, see Generic programming for more information. Generics were added to the Java language syntax in version 1.5. This means that code using Generics will not compile with Java 1.4 and less. Use of generics is optional. For backwards compatibility with pre-Generics code, it is okay to use generic classes without the generics type specification (`<T>`). In such a case, when you retrieve an object reference from a generic object, you will have to manually cast it from type Object to the correct type.

Note for C++ programmers

Java Generics are similar to C++ Templates in that both were added for the same reason. The syntax of Java Generic and C++ Template are also similar. There are some differences however. The C++ template can be seen as a kind of macro, in that a new copy of the code is generated for each generic type referenced. All extra code for templates is generated at compiler time. In contrast, Java Generics are built into the language. The same code is used for each generic type. For example:

Code section 4.50: Java generics.

```
1  Collection<String>  collString  = new ArrayList<String>();
2  Collection<Integer> collInteger = new ArrayList<Integer>();
```

Both these objects appear as the same type at runtime (both `ArrayList`'s). The generic type information is erased during compilation (type erasure). For example:

Code section 4.51: Type erasure.

```
1  public <T> void method(T argument) {
2    T variable;
3    …
4  }
```

is transformed by erasure into:

Code section 4.52: Transformation.

```
1  public void method(Object argument) {
2    Object variable;
3    …
4  }
```

Test your knowledge

Question 4.9: Consider the following class.

Question 4.9: Question9.java

```
1  import java.util.ArrayList;
2  import java.util.Collection;
3
4  public class Question9 {
5    public static void main(String[] args) {
6      Collection<String> collection1 = new ArrayList<String>();
7      Collection<? extends Object> collection2 = new ArrayList<String>();
8      Collection<? extends String> collection3 = new ArrayList<String>();
9      Collection<? extends String> collection4 = new ArrayList<Object>();
10     Collection<? super Object> collection5 = new ArrayList<String>();
11     Collection<? super Object> collection6 = new ArrayList<Object>();
12     Collection<?> collection7 = new ArrayList<String>();
13     Collection<? extends Object> collection8 = new ArrayList<?>();
14     Collection<? extends Object> collection9 = new ArrayList<Object>();
15     Collection<? extends Integer> collection10 = new ArrayList<String>();
16     Collection<String> collection11 = new ArrayList<? extends String>();
17     Collection collection12 = new ArrayList<String>();
18   }
19 }
```

Which lines will generate a compile error?

Answer

Answer 4.9: Answer9.java

```
1  import java.util.ArrayList;
2  import java.util.Collection;
```

```
3
4  public class Answer9 {
5    public static void main(String[] args) {
6      Collection<String> collection1 = new ArrayList<String>();
7      Collection<? extends Object> collection2 = new ArrayList<String>();
8      Collection<? extends String> collection3 = new ArrayList<String>();
9      Collection<? extends String> collection4 = new ArrayList<Object>();
10     Collection<? super Object> collection5 = new ArrayList<String>();
11     Collection<? super Object> collection6 = new ArrayList<Object>();
12     Collection<?> collection7 = new ArrayList<String>();
13     Collection<? extends Object> collection8 = new ArrayList<?>();
14     Collection<? extends Object> collection9 = new ArrayList<Object>();
15     Collection<? extends Integer> collection10 = new ArrayList<String>();
16     Collection<String> collection11 = new ArrayList<? extends String>();
17     Collection collection12 = new ArrayList<String>();
18   }
19 }
```

- Line 9: `Object` does not extend `String`.
- Line 10: `String` is not a superclass of `Object`.
- Line 13: `ArrayList<?>` can't be instantiated.
- Line 15: `Integer` does not extend `String`.
- Line 16: `ArrayList<? extends String>` can't be instantiated.
