Java Quick Reference

- Home
- SCJP2 Study Notes
- Case Studies
- SCJA Notes
- SCJD Notes
- Projects
- Favourite Links
- About
- Feedback

Welcome fellow Java enthusiasts!

This site grew out of a desire to store all the information I discovered in my study of the Java Language in one easily accessible location and format.

If you're brand new to Java you may want to begin by working your way through a number of the on-line tutorials. Click on **Favourite Links** and then choose **Tutorials** for a list of what's available on the Web.

If you're studying for the *Sun Certfied Java Programmer Exam (SCJP)* click on **SCJP2 Study Notes**. You'll find a collection of notes I made while studying for my own SCJP certification which I'm happy to say I passed on December 14th, 2000.

All the best in your studies!

Jane

Java Quick Reference

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

SCJP2 Study Notes

This section contains study notes for the Sun Certified Java 2 Programmer Exam (SCJP2).

The objectives are based on the *Testing Objectives for the New Sun Certified Programmer for Java 2 Platform* posted on Sun's site as of October 1st, 2000.

The exam consists of 59 questions. A passing mark of 61% is required. The time limit, originally 90 minutes, has now been increased to 120 minutes.

NEW 1.4 EXAM as of August, 2002

Sun introduced a new exam version in August, 2002. I've marked up my pages to indicate which objectives have been dropped or added; and, where possible, provided a link to study materials related to the new objectives.

!!! ALWAYS CHECK SUN'S SITE FOR THE LATEST OBJECTIVES !!!

Usage

- use the menu on the left to navigate the various Certification Objective pages
- use the menu on the bottom of the Objective and note pages to navigate notes related to the selected Objective
- save and compile the **Code Examples** to see Java concepts in action
- **Tips** are things to be keep in mind when taking the exam
- **Traps** are things to watch out for when taking the exam

Testing concepts
If you're having a problem with a concept, WRITE SOME CODE to test it! DO NOT use an IDE! Compile all your test code from the command line; this ensures you'll see all the errors the compiler may create.

Why get certfied?

Read an on-line article by David L. Hecksel and Marcus Green in The Certification Magazine

!!! Study Tip !!!

Visit JavaRanch on a regular basis!
It's *the best* site on the Web if you want to learn everything and anything about Java!

Pick up a good certification study guide. There are a number of excellent ones on the market, The Complete Java 2 Certification Study Guide: Programmer's and Developers Exams (With CD-ROM) a.ka. **RHE** is a favourite of many JavaRanchers.

Of course, I like the one I co-authored with my fellow JavaRanch moderators the best<g>

[Mike Meyer's Java 2 Certification Passport]
by Cindy Glass, Jane Griscti, Margarita Isayeva, Ajith Kallambella, and Kathy Sierra

A concise, affordable and portable guide to Sun's Java 2 Exam 310-025

Errata Page

Best of luck in your studies!

Language Fundamentals Certification Objectives

- Identify correctly constructed source files, package declarations, import statements, class declarations (of all forms including inner classes), interface declarations and implementations (for java.lang.Runnable or other interfaces described in the test), method declarations (including the main method that is used to start execution of a class), variable declarations and identifiers.
- State the correspondence between index values in the argument array passed to a main method and command line arguments. Identify all Java programming language keywords and correctly constructed identifiers.
- State the effect of using a variable or array element of any kind when no explicit assignment has been made to it.
- State the range of all primitive data types and declare literal values for String and all primitive types using all permitted formats, bases, and representations.

1.4 Exam Objectives

The objectives are basically the same; the first objective in 1.2 has been restated as:

- Identify correctly constructed package declarations, import statments, class declarations (of all forms including inner classes) interface declarations, method declarations (including the main method that is used to start execution of a class), variable declarations and identifiers.
- Identify classes that correctly implement an interface where that interface is either java.lang.Runnable or a fully specifiec interface in the question.

The second 1.2 objective has been split with an additional note on 'keywords'

- State the correspondence between index values in the argument array passed to a main method and command line arguments.
- Identify all Java programming language keywords. Note: There will not be any questions regarding esoteric distinction between keywords and manifest constants.

	Source	Package	Import	Class	Interface	Constructors
	Methods	main()	Identifiers	Keywords	Defaults	Arrays
	Primitives	# Literals	char Literal	String Literals	Class Literals	

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Operators and Assignments Certification Objectives

(1.4 Objectives are identical)

- Determine the result of applying any operator, including assignment operators and instanceof, to operands of any type, class, scope, or accessibility, or any combination of these.
- Determine the result of applying the boolean equals(Object) method to objects of any combination of the classes java.lang.String, java.lang.Boolean, and java.lang.Object.
- In an expression involving the operators &, |, &&, ||, and variables of known values state which operands are evaluated and the value of the expression.
- Determine the effect upon objects and primitive values of passing variables into methods and performing assignments or other modifying operations in that method.

	Conversions	Promotion	Overflow	Unary	Prefix	Arithmetic
	Bin/Hex/Octal	Bitwise	Shift	Comparison	Logical	Assignment
	Cast	Ternary	String	equals()	Precedence	Bit vs Logic
	Method Invocation					

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Flow Control and Exception Handling Certification Objectives

- Write code using if and switch statements and identify legal argument types for these statements.
- Write code using all forms of loops including labeled and unlabeled use of break and continue, and state the values taken by loop control variables during and after loop execution.
- Write code that makes proper use of exceptions and exception handling clauses (try, catch, finally) and declares methods and overriding methods that throw exceptions.

1.4 Exam: Additional objectives

- Recognize the effect of an exception arising at a sepcified point in a code fragment. Note: The exception may be a runtime exception, a checked exception, or an error (the code may include try, catch, or finally clauses in any legitimate combination).
- Write code that makes proper use of assertions, and distinguish appropriate from inapporopriate uses of assertions.
- Identify correct statements about the assertion mechanism.

For additional study materials try:
Sun: Programming with Assertions
Developerworks: Working with Assertions
JavaWorld: Understand the mechanics of ... new assertion facility

	Statements	if	switch	for	while	do
	Labels	Exceptions	Handling Exceptions	try-catch-finally		

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Declarations and Access Control Certification Objectives

(1.4 objectives are identical)

- Write code that declares, constructs, and initializes arrays of any base type using any of the permitted forms both for declaration and initialization.
(Covered under Language Fundamentals - Array Initialization)

- Declare classes, inner classes, methods, instance variables, static variables, and automatic (method local) variables making appropriate use of all permitted modifiers (such as public, final, static, abstract, and so forth). State the significance of each of these modifiers both singly and in combination, and state the effect of package relationships on declared items qualified by these modifiers.

- For a given class, determine if a default constructor will be created, and if so, state the prototype of that constructor.
(Covered under Language Fundamentals - Constructors)

- State the legal return types for any method given the declarations of all related methods in this or parent class.
(Covered under Language Fundamentals - Method Declarations)

Additional References

- Chapter 6 Objects and Classes from *The Complete Java 2 Certification Stuyd Guide* by Simon Roberts, Philip Heller, Michael Ernest
- Sun Tech Tip: Using Class Methods and Variables
- Sun Tech Tip: Global Variables

	Access Modifiers	Special Modifiers	this and super	Scope	Inheritance	Access Control

Garbage Collection Certification Objectives

- State the behaviour that is guaranteed by the garbage collection system, and write code that explicitly makes objects eligible for collection.

1.4 Exam

The above objective has been expanded as:

- State the behavior that is guaranteed by the garbage collection system.
- Write code that explicitly makes objects eligible for garbage collection.
- Recognize the point in a piece of source code at which an object becomes eligible for garbage collection.

Behaviour | Eligibility | finalize()

Overloading, Overriding, Runtime Types and Object Orientation Certification Objectives

(1.4 Objectives are identical)

- State the benefit of encapsulation in object oriented design and write code that implements tightly encapsulated classes and the relationships "is a" and "has a".
- Write code to invoke overridden or overloaded methods and parental or overloaded constructors; and describe the effect of invoking these methods.
- Write code to construct instances of any concrete class including normal top level classes, inner classes, static inner classes, and anonymous inner classes.

Threads Certification Objectives

- Write code to define, instantiate, and start new threads using both java.lang.Thread and java.lang.Runnable.
- Recognize conditions that might prevent a thread from executing.
- Write code using synchronized, wait, notify, or notifyAll, to protect against concurrent access problems and to communicate between threads. Define the interaction between threads and between threads and object locks when executing synchronized, wait, notify, or notifyAll

1.4 Exam

The third 1.2 objective has been re-worded as:

- Write code using synchronized wait, notify and notifyAll to protect against concurrent access problems and to communicate between threads.
- Define the interaction among threads and object locks when executing synchronized wait, notify or notifyAll

The `java.lang` Package Certification Objectives

- Write code using the following methods of the java.lang.Math class: abs, ceil, floor, max, min, random, round, sin, cos, tan, sqrt.
- Describe the significance of the immutability of String objects.

1.4 Exam : Additional objectives

- Describe the significance of wrapper classes, including making appropriate selections in the wrapper classes to suit specified behavior requirements, stating the result of excecuting a fragment of code that includes an instance of one of the wrapper classes, and writing code using the following methods of the wrappers classees 9e.g, Integer, Double, etc):
  - doubleValue
  - floatValue
  - intValue
  - longValue
  - parseXxx
  - getXxx
  - toString
  - toHexString

Main Classes | Wrapper Classes | Math Class | String Immutability | String Class | StringBuffer Class

The `java.util` Package Certification Objectives

- Make appropriate selection of collection classes/interfaces to suit specified behavior requirements.

1.4 Exam

This objective has been renamed **The Collection Framework** and the following has been added:

- Distinguish between correct and incorrect implementations of hashcode methods.

Also see

- Collections - a tutorial by Joshua Bloch
- The Collection Framework
- The Java Collections Framework

Collections Framework | Collection | Abstract Classes | Iterator | List

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

The `java.awt` Package Certification Objectives

NOT REQUIRED FOR 1.4 EXAM

- Write code using component, container, and LayoutManager classes of the java.awt package to present a GUI with a specified appearance and resize behaviour, and distinguish the responsibilities of layout managers from those of containers.

- Write code to implement listener classes and methods, and in listener methods, extract information from the event to determine the affected component, mouse position, nature and time of the event. State the classname for any specified event listener interface in the java.awt.event package.

Pay Attention to which Layout Managers implement LayoutManager2
• one thing I discovered (after I wrote the exam!) that is of **prime importance** in the way containers handle components when they are resized is knowing which Layout Interface the active LayoutManager implements. Any Layout Manager that extends the LayoutManager2 Interface **keeps track of their own components**. • What this means in practice is that if the layout manager is set **after** components have been added to the container and the layout manager implements the LayoutManager2 interface, **no components will be visible**. • LayoutManager2 type managers **do not query the container for a list of components, they maintain their own list**. • FlowLayout and GridLayout, both implement LayoutManager. When the container is resized they will query the container for a list of the components and then layout them out according to their contract. • CardLayout, BorderLayout, GridBagLayout, BoxLayout, and OverlayLayout implement the LayoutManager2 interface. If the container is resized they rely on their own, internal list of components. Components added to a container **before** the LayoutManager was added will not be known and hence not included in the layout when the container is resized.

Note
I haven't gotten around to re-writing my original notes. They are located at http://members.rogers.com/jgriscti/awt.html

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

The `java.io` Package Certification Objectives

NOT REQUIRED FOR 1.4 EXAM

- Write code that uses objects of the file class to navigate a file system.
- Write code that uses objects of the classes InputStreamReader and OutputStreamWriter to translate between Unicode and either platform default or ISO 8859-1 character encoding and distinguish between conditions under which platform default encoding conversion should be used and conditions under which a specific conversion should be used.
- Select valid constructor arguments for FilterInputStream and FilterOutputStream subclasses from a list of classes in the java.io package.
- Write appropriate code to read, write, and update files using FileInputStream, FileOutputStream and RandomAccessFile objects.
- Describe the permanent effects of the file system of constructing and using FileInputStream, FileOutputStream, and RandomAccessFile objects.

Tip
• focus on the classes mentioned in the objectives and their constructors

Also see

- Introduction to Java I/O

	Pkg Overview	Data Streams	Character Streams	Byte Streams	File Class	Readers & Writers
	Filter Streams	Data Input/Output	Reading & Writing Files	Serialization		

Java Quick Reference

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Sun Sites

- **Sun Certified Programmer for the Java 2 Platform** certification objectives.
- (JSK) Java 2 Platform Standard Edition v 1.3
- (JLS) Java Language Specification

Books

On-line

- Thinking In Java by Bruce Eckel
- Essentials of the Java Programming Language: A Hands on Guide, Part 1
- Essentials of the Java Programming Language: A Hands on Guide, Part 2
- Writing Advanced Applications for the Java Platform

Hardcover

- (JPL) **The Java Programming Language Second Edition** by Ken Arnold and James Gosling, The Java Series, Addison Wesley, 1998
- (CPJ) **Concurrent Programming in Java Second Edition: Design Principles and Patterns** by Doug Lea, The Java Series, Addison Wesley, 2000
- (JCL1) **The Java Class Libraries Second Edition, Volume 1** by Patrick Chan and Rosanna Lee, The Java Series, Addison Wesley, 1998
- (JCL2) **The Java Class Libraries Second Edition, Volume 2** by Patrick Chan and Rosanna Lee, The Java Series, Addison Wesley, 1998
- (JCLS) **The Java Class Libraries Second Edition, Volume 1: Supplemental for the Java 2 Platform, Standard Edition, v1.2** by Patrick Chan, Rosanna Lee, and Douglas Kramer, The Java Series, Addison Wesley, 1999
- (GJ) **Graphic Java: Mastering the AWT** by David M. Geary and Alan L. McClellan, SunSoft Press, 1997
- (JJ) **Java 2 Certification** by Jamie Jaworski, New Riders, 1999
- (BB) **Java Certification Exam Guide for Programmers and Developers** by Barry Boone, McGraw Hill, 1997
- (VA) **Programming with VisualAge for Java** by Marc Carrel-Billiard and John Akerley, Prentice-Hall, 1998

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Extracting Source code for the Java API classes

To extract source code for the Java Class files, check your JDK directory for a src.jar file. In the same directory, enter

```
    jar tf src.jar > srcList.txt
```

This will create a text file listing all the .java files in the src.jar file.
View the text file to locate the path name of the class you're interested in and then type:

```
    jar xf src.jar file pathname

    For example, to extract the Reader.java file
        jar xf src.jar src/java/io/Reader.java
```

Compiling with JDK 1.3 under Win98

If you're having problems compiling check the following:

1. you do NOT have CLASSPATH set in your AUTOEXEC.BAT file (JDK 1.3 does not require the DOS environment variable).
   If the variable is set because of other programs, make sure it begins with a '.\' to ensure the current directory is always included.
2. you are issuing the compile command from within the directory containing the .java source file
3. if you are using the javac switch -classpath DO NOT include an ending '\'

JRE can't locate .jar files under Win98

If you've downloaded some .jar files and installed them, as instructed, to the jdk1.3\jre\lib\ext directory but you're still getting ClassDefNotFound errors when you try to run an application that references the jars; check your system for a Java JRE Plug-in. If one exists, copy the .jar files to *that* ...\jre\lib\ext directory and re-boot.

The Runtime should now be able to find the .jar files properly.

Tips

- an empty source file will compile without error
- if a .java file does not contain a *public* class or interface it can have any name

- a single-type import will take precedence over an import-on-demand
- import-on-demand types do not increase the size of the compiled code ie only the types actually used are added to the code
- while import-on-demand adds no overhead to the compiled code, they can slow down the speed of the compile

- a constructor body can include a return statement providing no value is returned

- any method can throw a Runtime or Error exception without declaring it in the throws clause
- methods having the same name and parameter types do not have the same signature *unless* the parameter types are listed in the same order

- **main()** can be declared **final**
- **main()** is inherited and can be overridden if not declared as **final**
- **args[0]** references first command line argument *after* the application name (arrays in Java are zero-based)
- **main()** can be declared **public static void ...** or **static public void ...**
- the variable name **does not** have to be **args**; can be anything as long as the **type** is **String[]**

- variables can have the same name as a method or a class

- only field variables are automatically initialized to their types default value; local variables must be explicitly initialized
- arrays are initialized to the default value of their type when they are created, not declared, even if they are local variables

- array index operator [] has highest level of precedence
- integer variables can be used as array dimension values

- postfix/prefix operators have the highest level of precedence
- remember that when the postfix operator is used in an expression, the current value of the variable is used

- a class may be assigned to an Interface type if the class implements the interface or one of it's sub-interfaces
- you cannot cast a primitive type to an object reference, or vice versa
- you cannot cast a boolean type to another primitive type

- String operations whose result does not alter the original string (ie calling toUpperCase() on a String that is already in uppercase) return the original string reference; otherwise they return a reference to a **new** String
- Strings are **immutable**; the original String value **can never be changed**
- all the primitive type wrapper classes override the Object.equals() method to compare the

value of the objects; the default Object.equals() method checks if the variables reference the same object

- you do not have to have a default statement in a `switch()` block
- the default statement in a `switch()` blcok can appear anywhere in the construct, does not have to be last

- all sections of the `for()` loop are optional

- `finalize()` can only be executed **once** on any object

Traps

- code with *package* or *import* declarations given in wrong order
- more than one *package* declaration
- file with more than one *public* class or interface declaration
- *filename.java* does not match name of *public* class declared in the file

- single-type imports for two classes in different packages but with the same simple name
- single-type import with the same simple name as a class defined in the source file
- attempting to import a package vs a type ie import java.util vs import java.util.*

- class attempting to extend more than one other class
- class declared both **final** and **abstract**

- an interface method declared as **native** or **synchronized**
- an interface method declared as **static**

- subclass with default constructor when the superclass does not have a no-args constructor or it's no-arg constructor has a throws clause
- constructor declared with a return type

- an **abstract** method also declared **private, native, final, synchronized,** or **strictfp**
- an **abstract** method declared in a non-abstract class
- a **native** or **abstract** method with a method body
- method returning a type which is not convertible to the declared return type
- a **void** method returning a value
- a **static** method referencing **this** or **super**

- **main()** declared other than according to the standard convention

- local (automatic) variables declared with a modifier other than **final**
- identifiers names beginning with a number or # sign

- **main** listed as a possible keyword
- capitalized words listed as possible keywords; particularly wrapper classes Integer, Boolean, etc
- C/C++ keywords listed as possible Java keywords

- an empty string vs **null** as the default value for a String object

- incorrect array declaration statements, particularly:

arrayType [#] varName;

- incorrect array initialization statements, particularly:
  arrayType[] varName = new arrayType[2];
  varName = { value, value, value };
- **negative** values for array index
- **long** value for array index
- array declaration used as an array creation statement

- variables of primitive type handled as Objects

- using the **char** literals \u000A or \u000D in comments or Strings
- String literal "c" assigned to **char** type
- using == operator to compare values of two different string reference variables

- variables requiring narrowing conversion being passed to methods without using a *cast*
- assigning a typed **byte** or **short** variable to a **char** variable

- floating point operation throwing an ArithmeticException
- Bitwise operator precdence is: **&** ^ |

- assigning subclasses with the same parent to each other
- assigning a parent class to a subclass without a cast

- result of an integer operation on byte or short types being assigned to a byte or short without an explicit cast

- a non-boolean value used for operand1 in a ternary expression

- using == to compare the contents of two different String objects

- using a new value based on a short-circuit operation that was never evaluated

- code that results in a primitive value being changed in a method (can't happen)
- code that results in an unchanged object value when it was changed in a method
- failing to cast a value to match a method parameter type ie assuming narrowing conversion on a method call

- a non-boolean value used in a loop or `if( )` statement
- using the assignment operator '=' vs '==' in an loop or `if()` statement

- using an expression vs a value promotable to int in a `switch()` block
- `switch()` blocks with duplicate case values
- `switch()` blocks with incorrectly 'typed' case statements
- `switch()` blocks with missing break statements (unintentionally causing code to fall through to next case)

- attempting to access a variable declared in the initialization outside of the for-loop
- `for()`loop with incorrect initialization expression
- `for()`loop with a non-boolean expression

- a question that targets a specific object for garbage collection (can't be done)
- a question that presumes to **force** the gc to run (can only suggest it run)

Java Quick Reference

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Mock Exams

A complete list of Mock Exams can be found on Maha Anna's JavaRanch site

Another list of Mock Exams by Levteck Getting Certified in Java

A Java SCJP Mock Exam by Ashok Gupta rated, by Levteck, as one of the more difficult mock exams. The site also contains study notes.

Case Studies

Learning *how* to put an OOP application together is not an easy task.

While there is lots of information available on the Java language and numerous books and articles on using various OO methods and notations there are very few resources that marry the two in a format that's helpful to beginners.

One tried and true method of learning how to program is to *study the code created by other programmers*. Posted here are the results of my own look at code written and designed by others.

What's the basis for my choosing a case study? Right now it's pretty simple. The code must be

1. available, preferably on the web
2. it must utilize multiple user defined types

The pages in this section will also be laid out slightly different than the rest of the site.

MailMerge

An example of a classic batch processing application implemented in Java. The design incorporates a **Singleton** pattern.

JCalculator

An example of a calculator component that can be used in any application. The design incorporates a **Command** pattern.

- Home
- SCJP2 Study Notes
- Case Studies
- Tech Articles
- SCJA Notes
- SCJD Notes
- Projects
- Favourite Links
- About
- Feedback

Case Studies - Technical Articles Index

An index to various technical articles on the web.

Basics

- Accessing the environment from Java applications
- Constructor and Initialization Ordering
- Class and Object initialization
- Default Constructors
- Destroying Objects
- How arguments are passed to Java methods
- Interfaces and Constants
- Narrowing and Widening Conversions
- Overload Resolution
- Shadowing
- Understanding Expression Evaluation Order
- Using Assertions
- Using Import Declarations
- Using Variable length argument lists

Class Design

- Abstract Classes
- Abstract Classes vs Interfaces
- Anonymous Classes
- Cloning Objects
- Java Design Patterns 101 (Developerworks tutorial)
- Joshua Bloch: A conversation about design
- Local Classes
- Making defensive copies of objects
- Making deep copies of objects
- Returning multiple values from a method
- Using Adapters
- Using Class methods and variables
- Use stack variables whenever possible
- When not to Overload Methods

Collections

- Using Java Collections
- Collection Utilities

Exceptions

Graphics

I/O

- I/O Redirection
- Improving Java I/O Performance
- Improving I/O Performance with buffering
- Improving Serialization performance with Externalizable
- Piped Streams
- Programming with Buffers
- Programming with File Attributes
- Random Access for Files
- Reading and writing Unicode using I/O Stream Encodings
- Reading from Output Streams
- Serialization in the real world
- Serialization and Transient values
- Temporary Files
- Using Charsets and Encodings
- Using Checksums
- Using ReadResolve
- Using the PushbackReader Class
- Using the Serialiazable Fields API

HTML

- Extracting links from an HTML document

Java Tools/Extras

- A custom utility class for JavaHelp software
- Adding Help to your applications with JavaHelp software
- Capturing Audio with the Sound API
- Creating a HelpSet with JavaHelp software
- Fundamentals of JavaMail API
- Generating custom taglets (JavaDoc)
- Getting started with Java Management Extensions (JMX)
- Reading files from Java Archives (Jars) (An addendum to this article)
- Sending mail with the JavaMail API

Math

- BigDecimal
- Character (using the Character class)
- Formatting BigDecimal Numbers
- Format currencies
- Format Dates
- Formatting Decimal Numbers
- Generating integer random numbers
- Performing exact calculations with floating-point numbers
- Representing currencies

- Some things you should know about Floating Point Arithmetic
- Using Random Numbers for Testing and Simulation
- Working with Number Bases (binary, decimal, octal, hex)

Miscellaneous

- Compiling source directly from a program
- Converting C programs to Java
- Discovering the calling methods name
- Goto statements and Java programming
- Invoking programs from Java applications
- Unpacking Zip files
- Producing MIDI Sound
- Using Method Pointers
- Using runtime.exec to invoke child processes

Optimization

- A Memory Testbed Application / Controlling your Memory Manager

Patterns

- Employ Factory Methods to best advantage
- Singleton: Limit class instances with a modified singleton
- Singleton: Creating thread-safe singletons

Reflection

- Reflection
- Using java.lang.Class
- Using Reflection to Create Class Instances
- Using Reflection to test methods and classes

RMI

- Dynamic Class Loading in RMI
- The LifeCycle of an RMI Server
- Using RMI to access legacy databases
- A Java RMI server framework

Strings

- String vs StringBuffer
- Collators
- Interning Strings
- Optimizing String Performance
- String Concatenation and Performance

Swing

- JTree (manipulating hierarchical data)
- JTree (understanding TreeModel)
- Keymaps
- Loading text files in Swing efficiently
- Look and Feel
- Make a Splash Screen in Swing
- Performance secrets in Swing
- Press Esc to close Dialog windows
- Printing in Swing
- Saving and reconstituting Swing components
- Tracking locations in a Document
- Undoing Text edits
- Using Swing Timers
- Using the GraphicsEnvironment class
- Swing model filtering (Using filter objects to reinterpret data and state models)
- The Java Foundation Classes(The new standard for Java GUI development)
- Using Progress bars and Monitors in Java GUI Applications
- Using Timers in Swing Applications

Text in Swing

- Converting Numeric Entities
- Displaying Multiline text
- Displaying text in multiple styles
- Text Overview
- Text attributes
- Modeling Text in Documents
- Pattern Matching (java.util.regex)
- The Element Interface
- Tabbing
- Sizing text with FontMetrics
- Customizing a Text Editor
- Concurrency in Swing Text

Threads

- Acquire multiple locks in a fixed, global order
- Do not reassign the object reference of a locked object
- Ease your multithreaded application programming (Producer-Consumer)
- Exploiting ThreadLocal to enhance scalability
- Can ThreadLocal solve the double-checked locking problem?
- Minimizing the overhead of synchronized blocks
- Multi-threading in Java programs
- Piped Streams (to communicate between threads)
- Programmatically choose threads for notification

Java Quick Reference

Home
SCJP2 Study Notes
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

This spot will eventually host study notes for the Sun Certified Java Architect Certification Exam.

Useful SCJA sites you may want to check out:

- SCJA 2 Study Notes by Aaron Robinson
- Martin Fowler's where you'll find a wealth of information on UML, Extreme Programming, Patterns and other design topics.
- ArgoUML a free CASE Tool.

Home
SCJP2 Study Notes
Case Studies
SCJA Notes
SCJD Notes
Application Design
GUI Design
Database Processing
Networking
Threads
Errors and Exceptions
Security
Documentation
Projects
Favourite Links
About
Feedback

SCJD Study Notes

Preliminary Notes
• I haven't completed the assignment or passed the SCJD ... these notes are being built as I go.

!!! ALWAYS CHECK SUN'S SITE FOR THE LATEST OBJECTIVES !!!

Overview

The exam consists of two parts:

1. A programming assignment
2. An examination to be taken at a test center. This exam contains multiple-choice and essay questions relating to the programming assignment.

There is no time limit on completing the assignment.

Quote from Sun
Basically, the SCJD is testing your ability to apply the Java core API set to code the solution to a problem. Because it is a programming assignment, you do not have a set time frame in which to complete the assignment. So, you can get the assignment and determine the studying you need to do. Questions and Answers about Java Platform Certification

It is recommended that you track your design decisions as the exam portion will ask you to explain why you opted for one design feature over another. Also, register for the exam immeadiately upon uploading your assignment, while your assignment is still fresh in your mind.

The majority consensus (from what I've seen in the forums) is that the assignment takes roughly 120 hours of work to complete.

Downloading the assignment

Order the assignment from Sun. They will send you information on downloading the assignment within 2 business days. The download will include a **jar** file containing:

- an *Introduction and Index* document
- source code that serves as a starting point
- a binary database file

Assignment features

The assignment requires the following features:

- a GUI for viewing information. Must demonstrate good principles of design. The specific design requirements will be provided in the assignment instructions.
- database extensions to support a flexible search and record locking
- network server functionality for the database systems.
- communications functionality to connect the user interface with the database. The server must be multi-threaded and thread safe.
- the application must be able to run in either stand-alone or network mode
- the GUI interface must be flexible enough to allow the easy implementation of future enhancements

The finished assignment must include:

- source and object code
- Javadoc documentation
- Database server documentation
- User interface (client) documentation
- a README file

Marking

The programming assignment is worth 155 points, you need 124 points to pass

Marks are based on the following criteria:

- General Considerations (58)
  - ease of use (23)
  - coding standards and readability (23)
  - clarity and maintainablity of the design and implementation (12)
- Documentation (20)
  - user documentation (10)
  - javadoc source documentation (5)
  - comments (5)
- User Interface (24)
  - layout uses accepted GUI principles
- Server Design (53)
  - locking (30)
  - error handling (8)
  - search algorithm: clarity and efficiency (15)

Knowledge of the following Technologies is apt to be required

- Application Design: Use cases, CRC, UML, Patterns
- GUI Design using Swing components and event handling
- Database processing
- Networking: Client-Server design, TCP/IP, Sockets, RMI, I/O Streams, Object Serialization
- Threads: implementing multi-threading
- Error and Exception handling
- Security profiles
- Documentation: JavaDoc, User Guide, Install instructions

Other SCJD Resources

- The Dallas SCJD Study Group
- Brian Thorn received full marks for his Documentation (Note: These links have not been working lately. It's possible Mr. Thorn has removed his pages.)
  - Programming Notes example
  - User Documentation example

There doesn't appear to be all that much out there. If you come across a good resource site, please let me know!

SCJD Study Notes - Application Design

The first thing you'll probably do when you download your assignment is read the guidelines and take a look at the included code. Your first impulse may be to jump in and start coding right away! DON'T! The point of the assignment isn't just to produce working code, it's to produce **well designed object-oriented code!**

Stop and ask yourself:

- What constitutes a well-designed Object-Oriented application?
- What features does it have?
- What separates a good design from a poor one?

Do you have a clear idea of the answers?

Knowing what the various OOD terms: encapsulation, inheritance, polymorphism, etc. mean is not the same as knowing how to apply them in a design.

Design is often described as *"more art than science"*. That doesn't help much if you don't have alot of experience in designing OOP applications. Where do you start? How do you begin?

There are a number of *modeling tools*: CRC, Use Cases, UML Diagrams, Patterns, etc. that help you describe an application.

A design is actually a *model* of the *abstracted objects* you will create to build your application. Modeling tools help you to identify the objects you'll need and how they will interact with each other to produce the required results.

You write your class files based on the objects you've modeled.

You might want to poke around the Object Orientation Tips site to find some pointers.

OOD | OOP | Resources

SCJD Study Notes - GUI Design

The JDK comes with a number of Swing demo applications. Check your JDK installed directory, in the demo/jfc directory. They include a JTable example, amongst other things. The most complete demo is SwingSet2.

The following links are some notes I've made on what's available.

- SimpleExample - changing the Look and Feel

SCJD Study Notes - Database Processing

Database

- Use a RandomAccessFile to build a low-level database. Article on JavaWorld

Searching

- Plant your data in a ternary search tree Article on JavaWorld

- Home
- SCJP2 Study Notes
- Case Studies
- SCJA Notes
- SCJD Notes
- Application Design
- GUI Design
- Database Processing
- Networking
- Threads
- Errors and Exceptions
- Security
- Documentation
- Projects
- Favourite Links
- About
- Feedback

SCJD Study Notes - Networking

Networking

- Custom Networking tutorial on Sun's site.
- Chapter 17 Writing the Network Protocol from *Java 2 The Complete Certification Study Guide* by Simon Robers, Philip Heller, and Michael Ernest

Remote Method Invocation (RMI)

- Sun's RMI White Paper
- Sun's Guide to RMI

- Fundamentals of RMI: Short Course By jGuru on Sun's site. Tutorial may be downloaded

- Building a Java Chat Server tutorial by Greg Travis on the IBM Developerworks site. The tutorial covers the problems inherent in building a server and techniques for over-coming them. The tutorial is free and can be downloaded but you need to register first.

- Distributed Computation with Java Remote Method Invocation a basic RMI tutorial by Kevin Henry.

- LifeCycle of an RMI Server (Sun Tech Tip)
- Dynamic Class loading in RMI (Sun Tech Tip)

- JavaWorld RMI Article Index An index of all the RMI articles published at JavaWorld.

RMI Tools

- rmic - The Java RMI Stub Compiler
- rmiregistry - The Java Remote Object Registry
- rmid - The Java RMI Activation System Daemon

Sockets

- All about Sockets Sun tutorial

Home
SCJP2 Study Notes
Case Studies
SCJA Notes
SCJD Notes
Application Design
GUI Design
Database Processing
Networking
Threads
Errors and Exceptions
Security
Documentation
Projects
Favourite Links
About
Feedback

SCJD Study Notes - Threads

- Learn how to implement a read/write lock
- Singletons, critical sections and read/write locks
- Acquire multiple locks in a fixed, global order
- Do not reassign the object reference of a locked object
- Exploiting ThreadLocal to enhance scalability
- Ease your multithreaded application programming (Producer-Consumer)
- Can ThreadLocal solve the double-checked locking problem?
- Minimizing the overhead of synchronized blocks
- Multi-threading in Java programs
- Piped Streams (to communicate between threads)
- Programmatically choose threads for notification
- Protecting shared resources with synchronized blocks
- Understand that for methods, synchronized locks objects, not methods or code
- Using synchronized or volatile when accessing shared variables
- Using Synchronized Statements
- Using Timers to run tasks on a background thread
- Writing efficient thread safe classes
- Double-checked locking: Clever, but broken. Do you know what synchronized really means? (JavaWorld)
- Warning! Threading in a multiprocessor world Find out why many tricks to avoid synchronization overhead just don't work. (JavaWorld)

SCJD Study Notes - Error and Exception Handling

- Sun Tutorial on Exceptions
- Using Exceptions
- Exception Handling: The good, the bad and the ugly (Article by Michael C. Daconta)
- The Proper Way to do Exception Handling (Article by Brian Maso)
- Exceptions in Java: Nothing Exceptional about them (Article by Gaurav Pal and Sonal Bansal)
- Using your own exception classes in Java (Article by Keld H. Hansen)

Home
SCJP2 Study Notes
Case Studies
SCJA Notes
SCJD Notes
Application Design
GUI Design
Database Processing
Networking
Threads
Errors and Exceptions
Security
Documentation
Projects
Favourite Links
About
Feedback

SCJD Study Notes - Security

- Security in Java 2 SDK 1.2 (Sun tutorial)
- Java's Security Architecture (Article by Bill Venners)
- Java security: How to install the security manager and customize your security policy (Article by Bill Venners)
- Java Security API - Example

SCJD Study Notes - Documentation

- JavaDoc Tool Home Page
- How to put comments in your code with JavaDoc
- Java theory and practice: I have to document THAT? Integrated documentation a la Javadoc is both a benefit and a burden

Java Quick Reference

Projects is a rather glorified name for this section. Right now it's just small examples.

- PropertiesViewer - display the system properties returned by `System.getProperties()` in a JTree.
- ClassBrowser - A simple Java class browser.
- FieldValidation - The example uses `InputVerifier`'s to validate user input.
- Calculator - a simple calculator that uses method reflection to invoke commands.
- CalendarComboBox - a custom 'date' input component that mimics a combo-box, displaying a perpetual calendar as it's drop-down.

Java Quick Reference

JavaRanch	JavaRanch hosts numerous discussion groups related to all areas of Java Development: SCJP, SCJA and SCJD Certification, EJB, XML, JSP and much, much more including *CattleDrive* (Java College) where you can write practice assignments and have someone *nitpick* your code for free!
JCHQ	Java Programmer Certification Exam and Training. Popular site created by Marcus Green. Discussions, tutorials, FAQ's and more.
JavaChina	A SCJP Certification site created by Roseanne Zhang. Contains a large Certification FAQ, code examples and much more!

Java Quick Reference

Home
SCJP2 Study Notes
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Site Design

The site is built and maintained by myself using **HomeSite** from Allaire

I swiped the folder and page icons from Jeff Rouyer, author of Dynamic HTML: Web Magic.

Java Quick Reference

Home
SCJP2 Study Notes
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

New 2 Java	Sun site geared to Java newbies. Contains an overview of the language, how to get started learning and using Java, and links to other resources.
Certification	Certification Objectives and exam details.
SDK	Download site for latest Java 2 *Software Development Kit*
JLS	View or download the *Java Language Specification.*
JVM	View or download the *Java Virtual Machine Specification.*
Glossary	Glossary of Java Technology related terms.
Code Conventions	On-line document outlining coding conventions for the Java Programming Language
Technical Articles	Numerous articles on various aspects of the Java platform: Collections, JDBC, Programming, JavaBeans, Graphics, etc.
Tech Tips	Tips, Techniques and sample code.
Bugs	Database containing reported Bugs. You need to register with the site before you can access the database.
Applets	Sample applets contributed by Java enthusiasts or created at Sun.
Code Samples	Code snippets (*examplets*) showing how to handle various common tasks.
Forte Resources	Developer resource for Sun's Java Development IDE, **Forte for Java.** Includes links to the FAQ, Technical Articles, Newsgroups, Documentation and Downloads.

Sun Sites | Tutorials | On-line Books | Resource Sites

Java Quick Reference

Home
SCJP2 Study Notes
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Sun Java Tutorial	*The Java Tutorial* viewable on-line or may be downloaded.
Thinking in Java	Popular tutorial by Bruce Eckel. Download the html version.
Introduction to Computer Science using Java	Interactive tutorial created by Bradley Kjell of Cental Connecticut State University. Assumes no prior programming experience.
Introduction to Programming Using Java	Tutorial written by David Eck of Hobart and William Smith Colleges. May be taken on-line or downloaded.
Java Tutorials	A series of Java tutorials available in the Objective Viewpoint column written by George Crawford III for ACM Crossroads, an student online magazine.
Pure Java Education Center	Collection of Java "how-to" articles, tips, techniques and source code.
Brewing Java	A basic Java tutorial by Elliot Rusty Harold.
Java Tutorials	A collection of introductory Java Tutorials from Free Java Help.
Java 2 Certification	A popular tutorial created by Marcus Green geared to the Java 2 Certification Objectives
Java Jems	Tutorials for beginners and SCJP certification.
JSP Tutorial	Tutorial on Java Servlets
JDBC Tutorial	Tutorial on Java Database Connectivity from the *Web Developers Virtual Library*.

Sun Sites | Tutorials | On-line Books | Resource Sites

Java Quick Reference

Data Structures and Algorithms with Object-Oriented Design Patterns in Java	This book is about the fundamentals of data structures and algorithms--the basic elements from which large and complex software artifacts are built.
Java Free Library	A list of Java online books made available by InformIt.
Java by Example	Complete tutorial on the Java Language.
Java Expert Solutions	Covers a wide variety of topics: RMI, JDBC, Applets, etc.
Focus on Java Programming	This is a free, online textbook on introductory Java™ programming. Lots of excercises and example code.

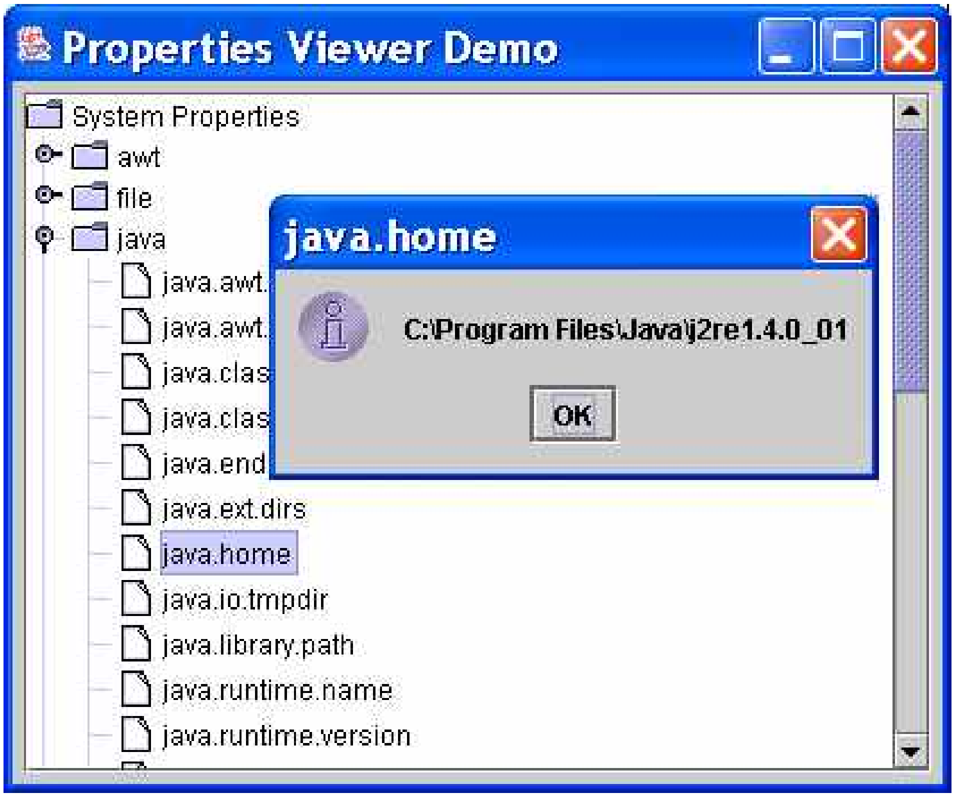
Java Quick Reference

Home
SCJP2 Study Notes
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Java FAQ	Java FAQ from the computer-lang/java/programmers/faq newsgroup. Updated weekly.
JavaFile	Collection of free Java applets.
DigitalCats	Java resource site. Contains articles, links to other resources.
Gamelan	One of the oldest and most popular Java resource sites.
JavaWorld	Java How-To's and Tutorials from *JavaWorld* magazine.
Other resources	A number of people have written study notes and built Java related sites. Browse the Java Certification web-ring (see bottom of page) to find others.

Sun Sites | Tutorials | On-line Books | Resource Sites

Java Project - PropertiesViewer



PropertiesViewer.java

Home | Projects

```java
package ca.janeg.properties;

import java.awt.Dimension;
import java.util.Iterator;
import java.util.Properties;
import java.util.Set;
import java.util.StringTokenizer;
import java.util.TreeMap;
import javax.swing.JPanel;
import javax.swing.JOptionPane;
import javax.swing.JScrollPane;
import javax.swing.SwingUtilities;
import javax.swing.JTree;
import javax.swing.event.TreeSelectionListener;
import javax.swing.event.TreeSelectionEvent;
import javax.swing.tree.TreeSelectionModel;
import javax.swing.tree.DefaultMutableTreeNode;

import javax.swing.JFrame;

/** Displays system properties in a sorted, categorized tree heirarchy.
 *  Select a property node to display its corresponding value.
 *
 * @author Jane Griscti  jane@janeg.ca
 * @version 1.0  Dec-21-2001
 */
public class PropertiesViewer extends JPanel{
    private Properties props = System.getProperties();
    private JTree tree;
    private JPanel owner;

    /** Creates a JPanel containing a JTree. Nodes are categorized
     *  according to the first element of the property name. For example,
     *  all properties beginning with 'java' are categorized under
     *  the node 'java'.
     */
    public PropertiesViewer(){
        super();
        owner = this;
        createSortedTree();
        JScrollPane jsp = new JScrollPane( tree );
        jsp.setPreferredSize( new Dimension( 400, 300 ) );
        jsp.setMinimumSize( getPreferredSize() );
        add( jsp );
    }

    /** Builds the JTree. The properties are given to a TreeMap, which automatically
     *  sorts them. The keys from the TreeMap are used to create the JTree nodes.
     *  A StringTokenizer is used to extract the first portion of the property name
     *  to build category nodes.
     */
    private void createSortedTree(){
        DefaultMutableTreeNode top = new DefaultMutableTreeNode("System Properties");
        Set keySet = new TreeMap(props).keySet();
        Iterator iter = keySet.iterator();
```

```
        DefaultMutableTreeNode key = null;
        DefaultMutableTreeNode category = null;
        String currentCategory = "";
        String newCategory = "";

        while( iter.hasNext() ){
            key = new DefaultMutableTreeNode( iter.next() );
            StringTokenizer stok = new StringTokenizer( (String)key.getUserObject(),
"." );
            newCategory = stok.nextToken();

            if( !currentCategory.equals(newCategory) ){
                currentCategory = newCategory;
                category = new DefaultMutableTreeNode( newCategory );
                top.add( category );
            }
            category.add( key );
        }

        tree = new JTree( top );
        tree.putClientProperty("JTree.lineStyle", "Angled" );

tree.getSelectionModel().setSelectionMode(TreeSelectionModel.SINGLE_TREE_SELECTION);
        tree.addTreeSelectionListener( new TreeListener() );
    }

    /** The JTree listener. When a property node is selected a JOptionPane
     *  is created to display the value associated with the property.
     */
    private class TreeListener implements TreeSelectionListener{

        public void valueChanged(TreeSelectionEvent e) {
            DefaultMutableTreeNode node = (DefaultMutableTreeNode)
                                               tree.getLastSelectedPathComponent();

            if (node == null) return;

            Object nodeInfo = node.getUserObject();

            if (node.isLeaf()) {
                String property = (String)nodeInfo;
                String value = props.getProperty( property );
                if( value.equals("") ){
                    value = "No associated value.";
                }
                JOptionPane.showMessageDialog( owner,
                                               value,
                                               property,
                                               JOptionPane.INFORMATION_MESSAGE);
            }
        }
    }

    /** Demos the PropertiesViewer.
     */
```

```
    public static void main(String[] args){
        JFrame frame = new JFrame("Properties Viewer Demo");
        frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        PropertiesViewer pv = new PropertiesViewer();
        frame.getContentPane().add( pv );
        frame.pack();
        frame.setVisible( true );
    }

}
```

Java Project - ClassBrowser

The GUI

A screen shot of the application.

The UML

The UML diagram.
The ClassBrowser class diagram.
The CBClassGroup class diagram.
The CBClassInfo class diagram.
The CBDocument class diagram.
The CBTreePanel class diagram.
The CBTextPane class diagram.
The FieldGroup class diagram.
The ConstructorGroup class diagram.
The MethodGroup class diagram.
The ParsedClassName class diagram.
The NameComparator class diagram.
The AccessSeparator class diagram.

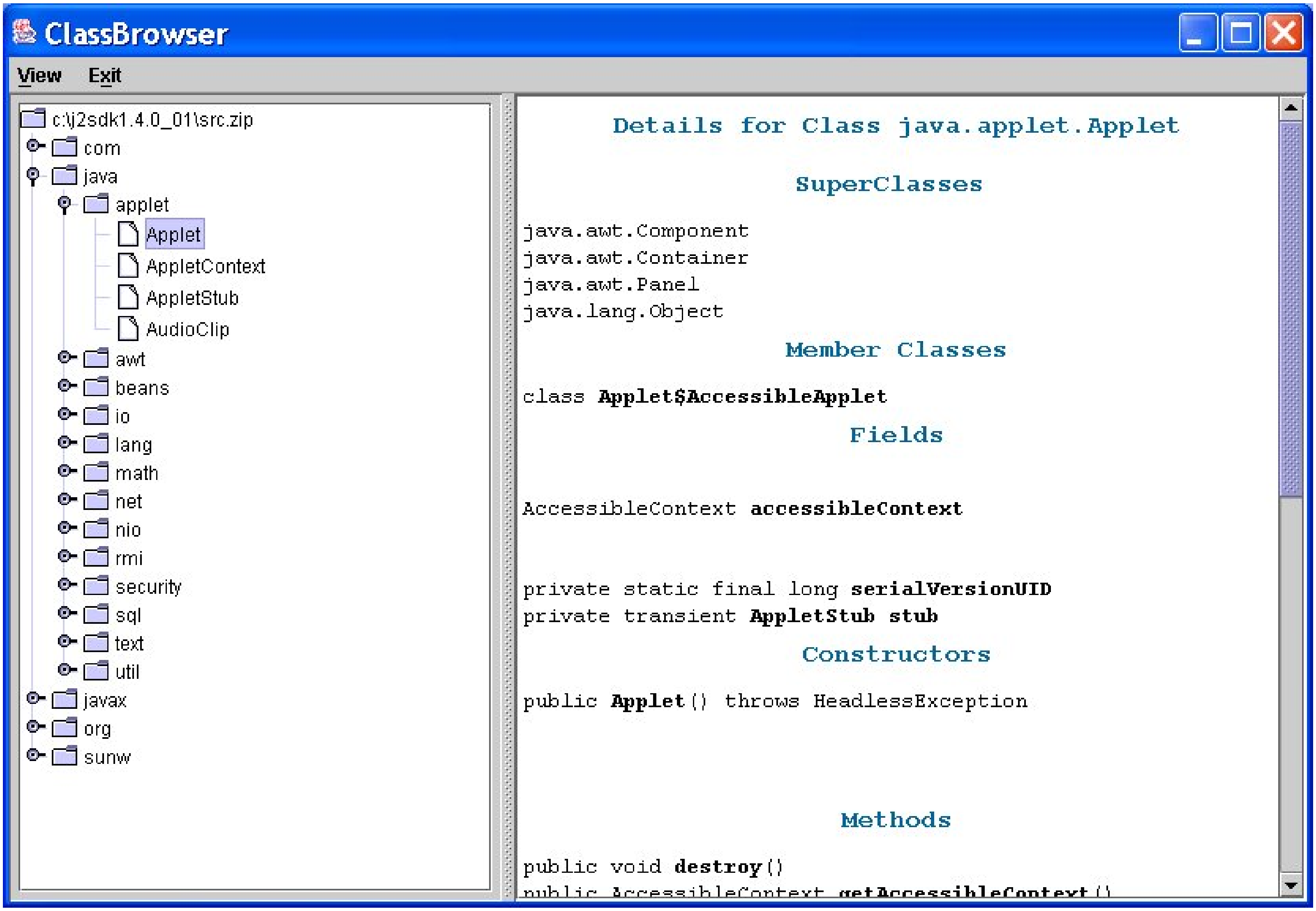
The Source Code

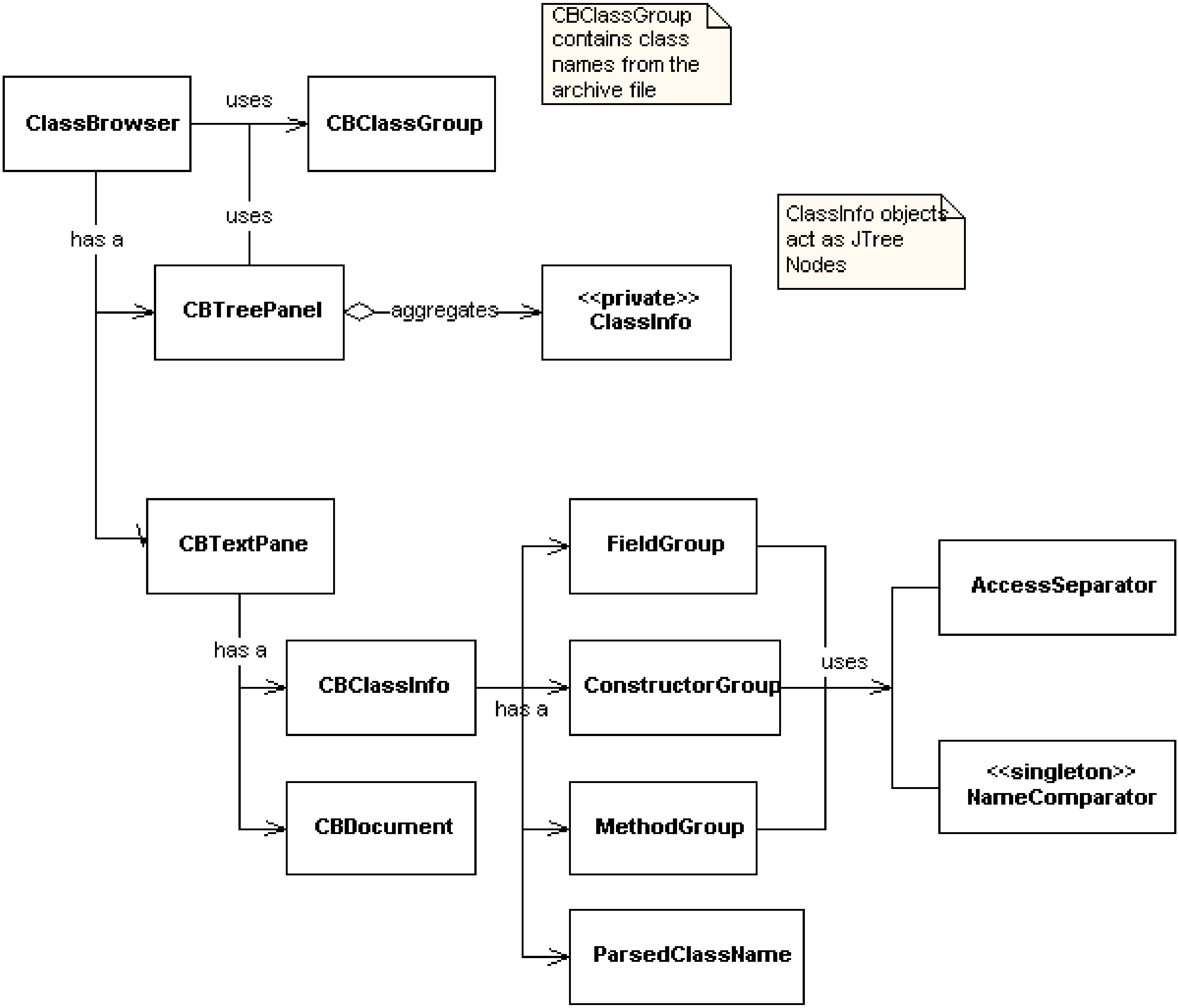
ClassBrowser
AccessSeparator
CBClassGroup
CBClassInfo
CBDocument
CBTextPane
CBTreePanel
ConstructorGroup
FieldGroup
MethodGroup
NameComparator
ParsedClassName

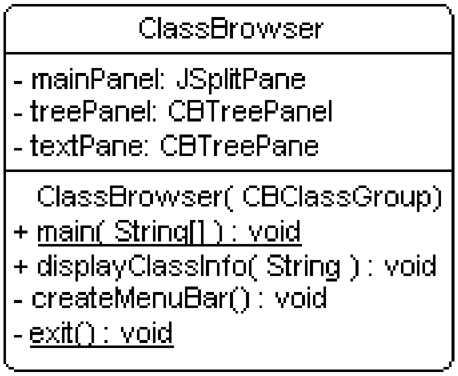
Refactoring Notes

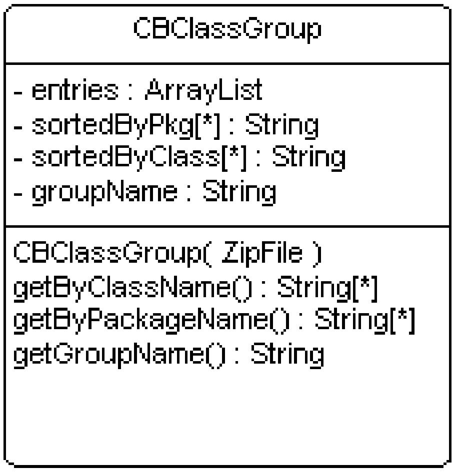
Probably could be refactored to use a **Group** interface or abstract class as the ConstructorGroup, FieldGroup and MethodGroup have identical functionality; the only difference being the type of their attributes.

The text display could also using some cleaning up. It would be nice to display the access groups using different colours: red for 'private', 'green' for public, etc.


ClassBrowser
View
Exit
c:\j2sdk1.4.0_01\src.zip
com
java
applet
Applet
AppletContext
AppletStub
AudioClip
awt
beans
io
lang
math
net
nio
rmi
security
sql
text
util
javax
org
sunw
Details for Class java.applet.Applet
SuperClasses
java.awt.Component
java.awt.Container
java.awt.Panel
java.lang.Object
Member Classes
class Applet$AccessibleApplet
Fields
AccessibleContext accessibleContext
private static final long serialVersionUID
private transient AppletStub stub
Constructors
public Applet() throws HeadlessException
Methods
public void destroy()


CBClassGroup contains class names from the archive file
ClassBrowser
uses
CBClassGroup
uses
has a
ClassInfo objects act as JTree Nodes
CBTreePanel
aggregates
<<private>>
ClassInfo
CBTextPane
FieldGroup
AccessSeparator
has a
CBClassInfo
ConstructorGroup
uses
has a
<<singleton>>
NameComparator
CBDocument
MethodGroup
ParsedClassName

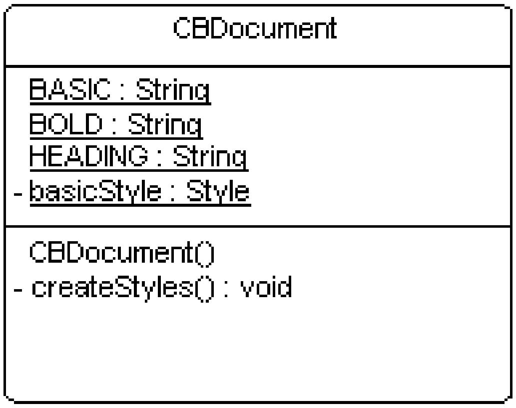

ClassBrowser
- mainPanel: JSplitPane
- treePanel: CBTreePanel
- textPane: CBTreePane
ClassBrowser(CBClassGroup)
+ main(String[]) : void
+ displayClassInfo(String) : void
- createMenuBar() : void
- exit() : void

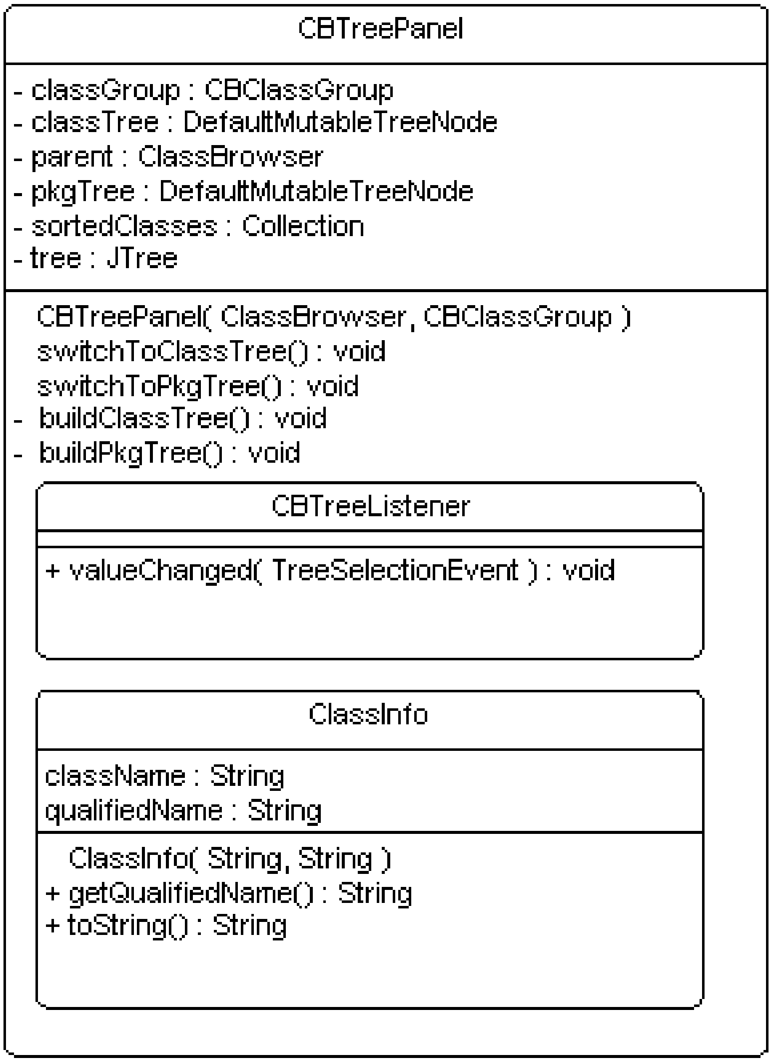

CBClassGroup
- entries : ArrayList
- sortedByPkg[*] : String
- sortedByClass[*] : String
- groupName : String
CBClassGroup(ZipFile)
getByClassName() : String[*]
getByPackageName() : String[*]
getGroupName() : String

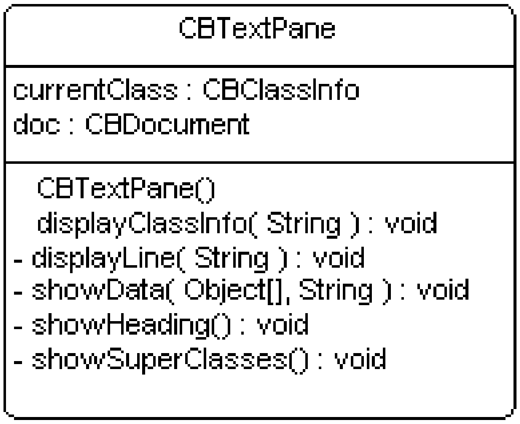
CBClassInfo

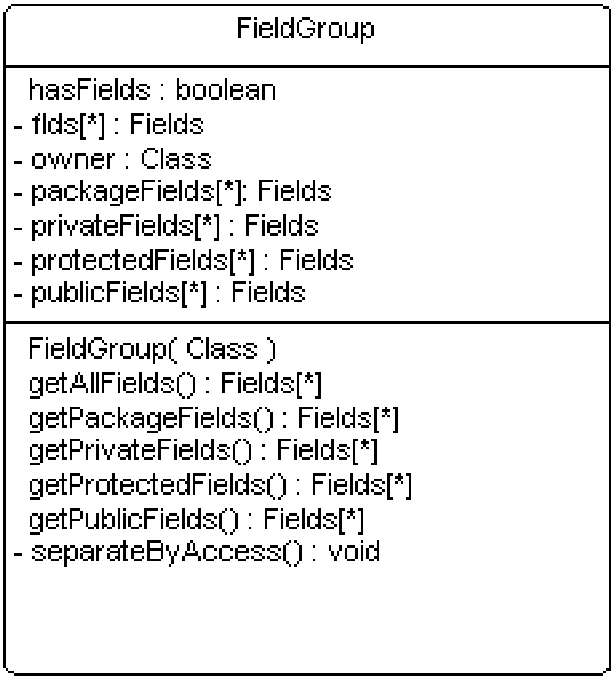
- NAME_DELIMITER : String
- ctors : ConstructorGroup
- flds : FieldGroup
- fullyQualifiedName : String
- memberClasses[*] : Class
- memberInterfaces[*] : Class
- memberPermission : boolean = true
- methods : MethodGroup
- pcn : ParsedClassName
- superClasses[*] : String
- thisClass : String

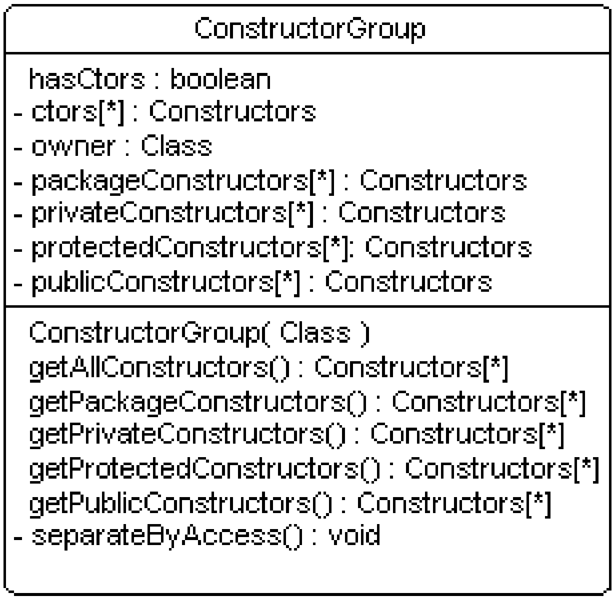
+ CBClassInfo(String)
+ getAllConstructors() : Constructors[*]
+ getAllFields() : Fields[*]
+ getAllMethods() : Methods[*]
+ getFullyQualifiedName() : String
+ getMemberClasses() : Class[*]
+ getMemberInterfaces() : Class[*]
+ getPackageConstructors() : Constructors[*]
+ getPackageFields() : Fields[*]
+ getPackageMethods() : Methods[*]
+ getPackageName() : String
+ getPackages() : String[*]
+ getPrivateConstructors() : Constructors[*]
+ getPrivateFields() : Fields[*]
+ getPrivateMethods() : Methods[*]
+ getProtectedConstructors() : Constructors[*]
+ getProtectedFields() : Fields[*]
+ getProtectedMethods() : Methods[*]
+ getPublicConstructors() : Constructors[*]
+ getPublicFields() : Fields[*]
+ getPublicMethods() : Methods[*]
+ getSimpleName() : String
+ getSuperClasses() : String[*]
+ hasCtors() : boolean
+ hasFields() : boolean
+ hasMethods() : boolean
+ hasSuperClasses() : boolean
+ isInterface() : boolean
+ isMemberAccessAllowed() : boolean
+ toString() : String
- isFullyQualifiedName() : boolean
- loadClassData() : void
- loadMemberClasses() : void
- loadSuperClasses() : void

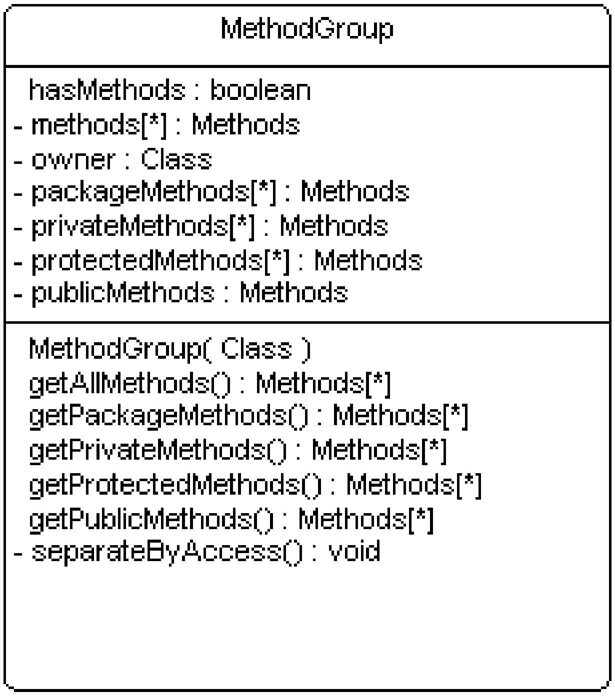

CBDocument
BASIC : String
BOLD : String
HEADING : String
- basicStyle : Style
CBDocument()
- createStyles() : void

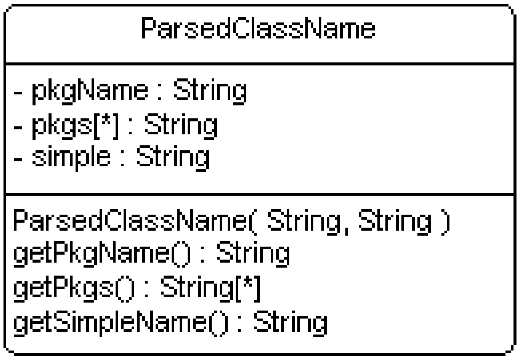

CBTreePanel
- classGroup : CBClassGroup
- classTree : DefaultMutableTreeNode
- parent : ClassBrowser
- pkgTree : DefaultMutableTreeNode
- sortedClasses : Collection
- tree : JTree
CBTreePanel(ClassBrowser, CBClassGroup)
switchToClassTree() : void
switchToPkgTree() : void
- buildClassTree() : void
- buildPkgTree() : void
CBTreeListener
+ valueChanged(TreeSelectionEvent) : void
ClassInfo
className : String
qualifiedName : String
ClassInfo(String, String)
+ getQualifiedName() : String
+ toString() : String

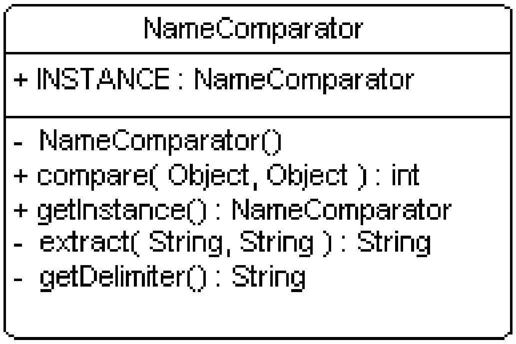

CBTextPane
currentClass : CBClassInfo
doc : CBDocument
CBTextPane()
displayClassInfo(String) : void
- displayLine(String) : void
- showData(Object[], String) : void
- showHeading() : void
- showSuperClasses() : void

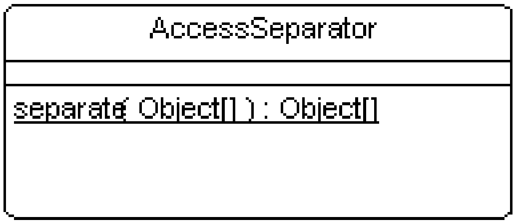

FieldGroup
hasFields : boolean
- flds[*] : Fields
- owner : Class
- packageFields[*]: Fields
- privateFields[*] : Fields
- protectedFields[*] : Fields
- publicFields[*] : Fields
FieldGroup(Class)
getAllFields() : Fields[*]
getPackageFields() : Fields[*]
getPrivateFields() : Fields[*]
getProtectedFields() : Fields[*]
getPublicFields() : Fields[*]
- separateByAccess() : void


ConstructorGroup
hasCtors : boolean
- ctors[*] : Constructors
- owner : Class
- packageConstructors[*] : Constructors
- privateConstructors[*] : Constructors
- protectedConstructors[*]: Constructors
- publicConstructors[*] : Constructors
ConstructorGroup(Class)
getAllConstructors() : Constructors[*]
getPackageConstructors() : Constructors[*]
getPrivateConstructors() : Constructors[*]
getProtectedConstructors() : Constructors[*]
getPublicConstructors() : Constructors[*]
- separateByAccess() : void


MethodGroup
hasMethods : boolean
- methods[*] : Methods
- owner : Class
- packageMethods[*] : Methods
- privateMethods[*] : Methods
- protectedMethods[*] : Methods
- publicMethods : Methods
MethodGroup(Class)
getAllMethods() : Methods[*]
getPackageMethods() : Methods[*]
getPrivateMethods() : Methods[*]
getProtectedMethods() : Methods[*]
getPublicMethods() : Methods[*]
- separateByAccess() : void


ParsedClassName
- pkgName : String
- pkgs[*] : String
- simple : String
ParsedClassName(String, String)
getPkgName() : String
getPkgs() : String[*]
getSimpleName() : String


NameComparator
+ INSTANCE : NameComparator
- NameComparator()
+ compare(Object, Object) : int
+ getInstance() : NameComparator
- extract(String, String) : String
- getDelimiter() : String


AccessSeparator
separate(Object[]) : Object[]

```java
package ca.janeg.cb;

import java.awt.Dimension;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.File;
import java.io.IOException;
import java.util.zip.ZipException;
import java.util.zip.ZipFile;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JScrollPane;
import javax.swing.JSplitPane;

/**
 *  A simple Java class browser.<p>
 *  Takes a .jar or .zip archive, extracts the class names and
 *  displays them in a JTree by package or alphabetically.<p>
 *  Selecting a class displays it's superclasses, fields,
 *  constructors and methods in an adjacent JTextPane.
 *
 *@author     Jane Griscti jane@janeg.ca
 *@created    January 26, 2002
 */
public class ClassBrowser extends JFrame {
    private JSplitPane mainPanel;
    private CBTreePanel treePanel;
    private CBTextPane textPane    = new CBTextPane();


    /**
     *  Constructs a new ClassBrowser object
     *
     *  @param  cbcg  a CBClassGroup object
     */
    public ClassBrowser( final CBClassGroup cbcg ) {
        super( "ClassBrowser" );
        setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

        treePanel = new CBTreePanel( this, cbcg );

        JScrollPane tsp  = new JScrollPane( textPane );
        tsp.setPreferredSize( new Dimension( 500, 300 ) );
        tsp.setMinimumSize( tsp.getPreferredSize() );

        mainPanel = new JSplitPane( JSplitPane.HORIZONTAL_SPLIT,
                                    treePanel, tsp );

        getContentPane().add( mainPanel );
        createMenuBar();

        pack();
```

```
        setVisible( true );
    }


    /**  Builds the menu bar. */
    private void createMenuBar() {

        JMenu menu           = new JMenu( "View" );
        menu.setMnemonic( 'v' );

        JMenuItem pkgItem    = new JMenuItem( "by Packages" );
        JMenuItem classItem  = new JMenuItem( "by Class" );

        pkgItem.addActionListener(
            new ActionListener() {
                public void actionPerformed( ActionEvent evt ) {
                    treePanel.switchToPkgTree();
                }
            }
        );

        classItem.addActionListener(
            new ActionListener() {
                public void actionPerformed( ActionEvent evt ) {
                    treePanel.switchToClassTree();
                }
            }
        );

        pkgItem.setMnemonic( 'p' );
        classItem.setMnemonic( 'c' );

        menu.add( pkgItem );
        menu.add( classItem );

        JMenuItem exitItem = new JMenuItem( "Exit" );
        exitItem.addActionListener (
            new ActionListener() {
                public void actionPerformed( ActionEvent evt ) {
                    dispose();
                    System.exit(0);
                }
            }
        );

        exitItem.setMnemonic( 'x' );

        JMenuBar menuBar     = new JMenuBar();
        menuBar.add( menu );
        menuBar.add( exitItem );
        setJMenuBar( menuBar );
    }
```

```
    void displayClassInfo( final String className ) {
        textPane.displayClassInfo( className );
    }

    private static void exit(){
        System.exit(1);
    }


    /**
     *  The main program for the ClassBrowser class
     *
     *@param  args  The command line arguments
     */
    public static void main( String[] args ) {
        if( args.length == 0 ) {
            System.out.println( "Usage: java ClassBrowser filepath" );
            System.out.println( "  where, filepath is the full path to the archive
file" );
            System.out.println( "         containing the class or source files." );
            System.out.println( "  e.g. c:/j2sdk1.4.0_01/src.zip" );
            exit();
        }

        CBClassGroup cbcg  = null;

        try {
            cbcg = new CBClassGroup( new ZipFile( new File( args[0] ) ) );
        } catch( ZipException e ) {
            System.out.println( args[0] + " is not a valid .jar or .zip file." );
            exit();
        }
        catch( IOException e ) {
            System.out.println( args[0] + " is not a valid file path." );
            exit();
        }

        ClassBrowser cb    = new ClassBrowser( cbcg );
    }
}
```

```
package ca.janeg.cb;

import java.util.ArrayList;

/*
 *  Takes an array of objects and uses their string names to separate
 *  the elements by their access levels.
 *
 *  @author  Jane Griscti    jane@janeg.ca
 *  @created January 13, 2002
 */
class AccessSeparator {

    /*
     *  Checks the name of an object for one of the four access levels:
     *  public, protected, private or default and returns four ArrayLists
     *  with the objects separated accordingly.
     */
    static Object[] separate( final Object[] obj ) {
        ArrayList pub  = new ArrayList();
        ArrayList pro  = new ArrayList();
        ArrayList pri  = new ArrayList();
        ArrayList pkg  = new ArrayList();

        String name    = null;
        int index      = 0;
        for( int i = 0; i < obj.length; i++ ) {
            name = obj[i].toString();

            if( name.indexOf( "public" ) >= 0 ) {
                pub.add( obj[i] );
            } else if( name.indexOf( "protected" ) >= 0 ) {
                pro.add( obj[i] );
            } else if( name.indexOf( "private" ) >= 0 ) {
                pri.add( obj[i] );
            } else {
                pkg.add( obj[i] );
            }
        }

        return new Object[]{pub, pro, pri, pkg};
    }
}
```

```java
package ca.janeg.cb;

import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.Comparator;
import java.util.Enumeration;
import java.util.StringTokenizer;
import java.util.zip.ZipEntry;
import java.util.zip.ZipFile;


/**
 *  Constructs a new CBClassGroup object by extracting
 *  class names from a .jar or .zip archive file.
 *  Extracted class names are stored for retreival by package or
 *  alphabetically by name.
 *
 *@author     Jane Griscti jane@janeg.ca
 *@created    January 5, 2002
 */
class CBClassGroup {
    private ArrayList entries       = new ArrayList();
    private String[] sortedByPkg;
    private String[] sortedByClass;
    private String groupName;


    CBClassGroup( final ZipFile zip ) throws IOException {
        groupName = zip.getName();

        Enumeration allEntries  = zip.entries();

        ZipEntry zipEntry       = null;
        String name;

        while( allEntries.hasMoreElements() ) {
            zipEntry = (ZipEntry)allEntries.nextElement();
            name = zipEntry.getName();

            // only want full paths, not partials
            if( name.endsWith( ".java" ) || name.endsWith( ".class" ) ) {
                    // drop the .java or .class ending
                    StringTokenizer stok  = new StringTokenizer( name, "." );
                    String token          = stok.nextToken();
                    entries.add( token );
            }
        }

        Collections.sort( (ArrayList)entries );
```

```
        sortedByPkg = (String[])entries.toArray( new String[0] );

        Collections.sort( (ArrayList)entries, CBNameComparator.getInstance() );
        sortedByClass = (String[])entries.toArray( new String[0] );
        entries = null;

    }


    /**
     *  Gets the class name entries sorted by package.
     *
     *@return    An array of class names sorted by package.
     */
    String[] getByPackageName() {
        return sortedByPkg;
    }


    /**
     *  Gets the class name entries sorted by class.
     *
     *@return    An array of class names sorted by the class simple name.
     */
    String[] getByClassName() {
        return sortedByClass;
    }


    /**
     *  Gets the name of the group of entries.
     *
     *@return    The fullpath name of the file containing this group of entries.
     */
    String getGroupName() {
        return groupName;
    }

}
```

```
package ca.janeg.cb;

import java.lang.reflect.Array;
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;

/**
 *  A CBClassInfo object used to load a class and store pertinent class
 *  information: superclasses, fields, methods, constructor names.
 *
 *  @author     Jane Griscti jane@janeg.ca
 *  @created    January 8, 2002
 */
public class CBClassInfo {
    private final static String NAME_DELIMITER  = ".";
    private final String fullyQualifiedName;
    private final ParsedClassName pcn;

    private Class thisClass;
    private String[] superClasses;
    private FieldGroup flds;
    private MethodGroup methods;
    private ConstructorGroup ctors;
    private Class[] memberClasses;
    private Class[] memberInterfaces;

    private boolean memberPermission            = true;


    /**
     *  Constructs a new CBClassInfo object. Checks for a fully qualified class
     *  name; however, this does not guarantee that the class is available to be
     *  loaded. <p>
     *
     *  A 'fully qualified name' consists of the classes package name and simple
     *  name given in dot-notation format. For example, java.lang.Object<p>
     *
     *  A class may only be loaded, and its information retreived, if it is
     *  available to the JVM via the bootstrap loader or the system classpath.
     *
     * @param       name    a fully qualified class name
     * @exception   ClassNotFoundException  if name is not a fully qualified class
     *      name
     */
    public CBClassInfo( final String name ) throws ClassNotFoundException {
        if( !isFullyQualifiedName( name ) ) {
            throw new ClassNotFoundException( " '" + name + "' is not a fully
qualified class name." );
        }
```

```
        fullyQualifiedName = name;
        pcn = new ParsedClassName( name, NAME_DELIMITER );
        loadClassData();
    }


    private boolean isFullyQualifiedName( final String name ) {
        return name.indexOf( NAME_DELIMITER ) > 0;
    }


    private void loadSuperClasses() {

        Class subclass    = thisClass;
        Class superclass  = subclass.getSuperclass();

        ArrayList tmp     = new ArrayList();

        while( superclass != null ) {
            String className  = superclass.getName();
            tmp.add( className );

            subclass = superclass;
            superclass = subclass.getSuperclass();
        }
        Collections.sort( tmp );
        superClasses = (String[])tmp.toArray( new String[0] );

        tmp = null;
    }


    private void loadMemberClasses() throws SecurityException {
        Class[] members  = thisClass.getDeclaredClasses();

        if( members.length > 0 ) {
            ArrayList mInter  = new ArrayList();
            ArrayList mClass  = new ArrayList();

            for( int i = 0; i < members.length; i++ ) {
                if( members[i].isInterface() ) {
                    mInter.add( members[i] );
                } else {
                    mClass.add( members[i] );
                }
            }

            if( !mClass.isEmpty() ) {
                memberClasses = (Class[])mClass.toArray( new Class[0] );
            }

            if( !mInter.isEmpty() ) {
```

```
                memberInterfaces = (Class[])mInter.toArray( new Class[0] );
            }
        }
    }

    private void loadClassData() throws ClassNotFoundException {

        thisClass = Class.forName( fullyQualifiedName );
        loadSuperClasses();
        flds = new FieldGroup( thisClass );
        methods = new MethodGroup( thisClass );
        ctors = new ConstructorGroup( thisClass );

        try {
            loadMemberClasses();
        } catch( SecurityException e ) {
            memberPermission = false;
        }
    }


    /**
     *  Returns the simpleName attribute of the CBClassInfo object
     *
     *@return    The simpleName value
     */
    public String getSimpleName() {
        return pcn.getSimpleName();
    }


    /**
     *  Returns the fullyQualifiedName attribute of the CBClassInfo object
     *
     *@return    The fullyQualifiedName value
     */
    public String getFullyQualifiedName() {
        return fullyQualifiedName;
    }


    /**
     *  Returns the packageName attribute of the CBClassInfo object
     *
     *@return    The packageName value
     */
    public String getPackageName() {
        return pcn.getPackageName();
    }


    /**
     *  Returns the package names associated with the class represented by
```

```
     *  this object.
     *
     *@return    The packages value
     */
    public String[] getPackages() {
        return pcn.getPackages();
    }


    /**
     *  Returns all the fields declared in the class represented by this object.
     *
     *@return    an object array containing Field objects
     */
    public Field[] getAllFields() {
        return flds.getAllFields();
    }


    /**
     *  Returns all the public fields declared in the class represented by this
     *  object.
     *
     *@return    an object array containing Field objects
     */
    public Field[] getPublicFields() {
        return flds.getPublicFields();
    }


    /**
     *  Returns all the private fields declared in the class represented by this
     *  object.
     *
     *@return    an object array containing Field objects
     */
    public Field[] getPrivateFields() {
        return flds.getPrivateFields();
    }


    /**
     *  Returns all the package fields declared in the class represented by this
     *  object. *
     *
     *@return    an object array containing Field objects
     */
    public Field[] getPackageFields() {
        return flds.getPackageFields();
    }
```

```
    /**
     *  Returns all the protected fields declared in the class represented by
     *  this object.
     *
     *@return    an object array containing Field objects
     */
    public Field[] getProtectedFields() {
        return flds.getProtectedFields();
    }



    /**
     *  Returns all the super classes the class represented by this object
     *  inherits from.
     *
     *@return    an object array containing Class objects
     */
    public String[] getSuperClasses() {
        return superClasses;
    }


    /**
     *  Returns all the methods declared in the class represented by this
     *  object.
     *
     *@return    an object array containing Method objects
     */
    public Method[] getAllMethods() {
        return methods.getAllMethods();
    }


    /**
     *  Returns all the public methods declared in the class represented by this
     *  object.
     *
     *@return    an object array containing Method objects
     */
    public Method[] getPublicMethods() {
        return methods.getPublicMethods();
    }


    /**
     *  Returns all the private methods declared in the class represented by
     *  this object.
     *
     *@return    an object array containing Method objects
     */
    public Method[] getPrivateMethods() {
        return methods.getPrivateMethods();
```

```
    }


    /**
     *  Returns all the package methods declared in the class represented by
     *  this object. *
     *
     *@return    an object array containing Method objects
     */
    public Method[] getPackageMethods() {
        return methods.getPackageMethods();
    }



    /**
     *  Returns all the protected methods declared in the class represented by
     *  this object.
     *
     *@return    an object array containing Method objects
     */
    public Method[] getProtectedMethods() {
        return methods.getProtectedMethods();
    }


    /**
     *  Returns all the constructors declared in the class represented by this
     *  object.
     *
     *@return    an object array containing Constructor objects
     */
    public Constructor[] getAllConstructors() {
        return ctors.getAllConstructors();
    }


    /**
     *  Returns all the public constructors declared in the class represented by
     *  this object.
     *
     *@return    an object array containing Constructor objects
     */
    public Constructor[] getPublicConstructors() {
        return ctors.getPublicConstructors();
    }


    /**
     *  Returns all the private constructors declared in the class represented
     *  by this object.
     *
     *@return    an object array containing Constructor objects
```

```
     */
    public Constructor[] getPrivateConstructors() {
        return ctors.getPrivateConstructors();
    }


    /**
     *  Returns all the package constructors declared in the class represented
     *  by this object. *
     *
     *@return    an object array containing Constructor objects
     */
    public Constructor[] getPackageConstructors() {
        return ctors.getPackageConstructors();
    }



    /**
     *  Returns all the protected constructors declared in the class represented
     *  by this object.
     *
     *@return    an object array containing Constructor objects
     */
    public Constructor[] getProtectedConstructors() {
        return ctors.getProtectedConstructors();
    }


    /**
     *  Returns all the classes declared as members of the class represented by
     *  this object if the package security allows access to the information.
     *
     *@return    an object array of Class objects
     *@see       isMemberAccessAllowed()
     */
    public Class[] getMemberClasses() {
        return memberClasses;
    }


    /**
     *  Returns all the interfaces declared as members of the class represented
     *  by this object if the package security allows access to the information.
     *
     *  @return    an object array of Class objects
     *  @see       isMemberAccessAllowed()
     */
    public Class[] getMemberInterfaces() {
        return memberInterfaces;
    }

    /**
```

```
     *  Returns true if the class has declared fields.
     */
    public boolean hasFields(){
        return flds.hasFields ? true : false;
    }

    /**
     *  Returns true if the class has declared methods.
     */
     public boolean hasMethods() {
        return methods.hasMethods ? true : false;
     }

    /**
     *  Returns true if the class has declared constructors.
     */
     public boolean hasCtors() {
        return ctors.hasCtors ? true : false;
     }

    /**
     *  Returns true if the class has super classes.
     */
     public boolean hasSuperClasses() {
        return Array.getLength( superClasses ) > 0;
     }

    /**
     *  Gets the interface attribute of the CBClassInfo object
     *
     *@return    The interface value
     */
    public boolean isInterface() {
        return thisClass.isInterface();
    }


    /**
     *  Gets the memberAccessAllowed attribute of the CBClassInfo object
     *
     *@return    The memberAccessAllowed value
     */
    public boolean isMemberAccessAllowed() {
        return memberPermission;
    }


    /**
     *  Returns a textual description of the object.
     *
     *@return    the name of the class represented by this object
     */
    public String toString() {
```

```
        return "A ClassInfo object for the '" + fullyQualifiedName +
            "' class.";
    }

}
```

```java
package ca.janeg.cb;

import java.awt.Color;
import javax.swing.text.BadLocationException;
import javax.swing.text.DefaultStyledDocument;
import javax.swing.text.Style;
import javax.swing.text.StyleConstants;

/**
 *  A customized DefaultStyledDocument used by the CBTextPane
 *  component to display class details as formatted text.
 *
 *@author     Jane Griscti jane@janeg.ca
 *@created    January 5, 2002
 */
class CBDocument extends DefaultStyledDocument {

    private static Style basicStyle;

    final static String BASIC        = "Basic";
    final static String HEADING      = "Heading";
    final static String BOLD         = "Bold";


    /**  Constructs a new CBDocument object */
    CBDocument() {
        createStyles();
    }


    /**  Adds three styles to the document: Heading, Basic and Bold */
    private void createStyles() {

        // Create the top-level style, with the required font
        basicStyle = addStyle( BASIC, null );
        StyleConstants.setFontFamily( basicStyle, "Courier New" );
        StyleConstants.setFontSize( basicStyle, 14 );
        StyleConstants.setForeground( basicStyle, Color.black );
        StyleConstants.setFirstLineIndent( basicStyle, 50.0f );
        StyleConstants.setSpaceAbove( basicStyle, 6 );
        StyleConstants.setSpaceBelow( basicStyle, 0 );

        // Heading: centered, bold, larger font
        Style s  = addStyle( HEADING, basicStyle );
        StyleConstants.setBold( s, true );
        StyleConstants.setFontSize( s, 16 );
```

```
        StyleConstants.setForeground( s, new Color( 0x006699 ) );
        StyleConstants.setAlignment( s, StyleConstants.ALIGN_CENTER );
        StyleConstants.setSpaceBelow( s, 12 );

        // BoldText
        s = addStyle( BOLD, basicStyle );
        StyleConstants.setBold( s, true );
    }

}
```

```java
package ca.janeg.cb;

import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.util.StringTokenizer;
import javax.swing.JOptionPane;
import javax.swing.JTextPane;
import javax.swing.text.AttributeSet;
import javax.swing.text.BadLocationException;
import javax.swing.text.Style;

/**
 *  A component to display formatted text detailing the superclasses,
 *  interfaces, fields, constructor, and methods of a selected class.
 *
 *  @author     Jane Griscti jane@janeg.ca
 *  @created    January 5, 2002
 */
class CBTextPane extends JTextPane {
    CBClassInfo currentClass;
    CBDocument doc;


    /**  Construct a new CBTextPane object */
    CBTextPane() {
        super();
    }


    /**
     *  Formats the class name and assigns it to the first line of the display
     *  document.
     */
    private void showHeading() {
        String head  = null;

        if( currentClass.isInterface() ) {
            head = "Details for Interface " + currentClass.getFullyQualifiedName();
        } else {
            head = "Details for Class " + currentClass.getFullyQualifiedName();
        }

        try {
            AttributeSet s  = doc.getStyle( doc.HEADING );
            doc.insertString( doc.getLength(),
                head + "\n",
                s );
            doc.setLogicalStyle( doc.getLength() - 1, (Style)s );
        } catch( BadLocationException e ) {
            JOptionPane.showMessageDialog( this,
                "Error displaying details. /n" + e,
                "Display Error",
                JOptionPane.ERROR_MESSAGE );
```

```
            return;
        }
    }


    /**
     *  Retreives the class superclasses, formats their names and adds them to
     *  the display document
     */
    private void showSuperClasses() {
        String[] supers  = currentClass.getSuperClasses();

        if( supers == null ) {
            return;
        }

        AttributeSet s   = doc.getStyle( doc.HEADING );
        try {
            doc.insertString( doc.getLength(),
                "SuperClasses \n",
                s );
        } catch( BadLocationException e ) {
            JOptionPane.showMessageDialog( this,
                "Error displaying details. /n" + e,
                "Display Error",
                JOptionPane.ERROR_MESSAGE );
            return;
        }

        doc.setLogicalStyle( doc.getLength() - 1, (Style)s );

        for( int i = 0; i < supers.length; i++ ) {

            try {
                doc.insertString( doc.getLength(),
                    supers[i] + "\n",
                    doc.getStyle( doc.BASIC ) );
            } catch( BadLocationException e ) {
                JOptionPane.showMessageDialog( this,
                    "Error displaying details. /n" + e,
                    "Display Error",
                    JOptionPane.ERROR_MESSAGE );
                return;
            }
        }
    }


    /**
     *  Formats the class details and adds them to the display document.
     *
     *@param  data  An array of Interface, Field, Constructor, or Method objects
     *@param  type  Description of Parameter
     */
```

```
    private void showData( final Object[] data, final String type ) {

        if( data == null ) {
            return;
        }

        try {
            if( type != "" ) {
                AttributeSet s  = doc.getStyle( doc.HEADING );
                doc.insertString( doc.getLength(),
                    type + "\n",
                    s );
                doc.setLogicalStyle( doc.getLength() - 1, (Style)s );
            }else{
                doc.insertString( doc.getLength(),
                                  "\n",
                                  doc.getStyle( doc.BASIC ) );
            }

            for( int i = 0; i < data.length; i++ ) {
                displayLine( data[i].toString() );
                doc.insertString( doc.getLength(),
                    "\n",
                    doc.getStyle( doc.BASIC ) );
            }
        } catch( BadLocationException e ) {
            JOptionPane.showMessageDialog( this,
                "Error displaying details. /n" + e,
                "Display Error",
                JOptionPane.ERROR_MESSAGE );
            return;
        }
    }


    /**
     *  Write a new line in the document
     *
     *  @param  line  the text to be displayed
     */
    private void displayLine( final String line ) {
        String className      = currentClass.getSimpleName();
        StringTokenizer stok  = new StringTokenizer( line, " (", true );
        String token          = new String( "" );

        while( stok.hasMoreTokens() ) {
            token = stok.nextToken();

            try {
                if( token.indexOf( className ) == -1 ) {
                    if( token.lastIndexOf( '.' ) > 0 &&
                        !token.endsWith( ")" ) ) {
                            int pos  = token.lastIndexOf( '.' );
                            token = token.substring( pos + 1 );
```

```
                }

                doc.insertString( doc.getLength(),
                    token,
                    doc.getStyle( doc.BASIC ) );
            } else {
                // show field, method, ctor name in bold
                int pos  = token.lastIndexOf( '.' );
                doc.insertString( doc.getLength(),
                    token.substring( pos + 1 ),
                    doc.getStyle( doc.BOLD ) );
            }
        } catch( BadLocationException e ) {
            JOptionPane.showMessageDialog( this,
                "Error displaying details. /n" + e,
                "Display Error",
                JOptionPane.ERROR_MESSAGE );
            return;
        }
    }

}


/**
 *  Replaces the current content with the details of the supplied class. All
 *  content is displayed using a StyledDocument.
 *
 *@param  str  the name of the class for which details will be displayed
 */
void displayClassInfo( final String str ) {

    try {
        currentClass = new CBClassInfo( str );
    } catch( ClassNotFoundException e ) {
        JOptionPane.showMessageDialog( this,
            "Unable to load class " + str +
            "\nPlease check your classpath.",
            "Error Loading Class",
            JOptionPane.ERROR_MESSAGE );
        return;
    }

    doc = new CBDocument();
    setStyledDocument( doc );

    showHeading();

    if( currentClass.hasSuperClasses() ) {
        showSuperClasses();
        Class[] inter       = currentClass.getMemberInterfaces();
        showData( inter, "Interfaces" );
    }
```

```
        Class[] members      = currentClass.getMemberClasses();
        showData( members, "Member Classes");

        if( currentClass.hasFields() ) {
            Field[] flds         = currentClass.getPublicFields();
            showData( flds, "Fields" );
            flds                 = currentClass.getPackageFields();
            showData( flds, "" );
            flds                 = currentClass.getProtectedFields();
            showData( flds, "" );
            flds                 = currentClass.getPrivateFields();
            showData( flds, "" );
        }

        if( currentClass.hasCtors() ) {
            Constructor[] ctors  = currentClass.getPublicConstructors();
            showData( ctors, "Constructors" );
            ctors  = currentClass.getProtectedConstructors();
            showData( ctors, "" );
            ctors  = currentClass.getPackageConstructors();
            showData( ctors, "" );
            ctors  = currentClass.getPrivateConstructors();
            showData( ctors, "" );
        }

        if( currentClass.hasMethods() ) {
            Method[] methods     = currentClass.getPublicMethods();
            showData( methods, "Methods" );
            methods     = currentClass.getProtectedMethods();
            showData( methods, "" );
            methods     = currentClass.getPackageMethods();
            showData( methods, "" );
            methods     = currentClass.getPrivateMethods();
            showData( methods, "" );
        }
        setCaretPosition( 0 );
    }
}
```

```
package ca.janeg.cb;

import java.awt.Dimension;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.ListIterator;
import java.util.StringTokenizer;
import java.util.TreeMap;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTree;
import javax.swing.event.TreeSelectionEvent;
import javax.swing.event.TreeSelectionListener;
import javax.swing.tree.DefaultMutableTreeNode;
import javax.swing.tree.DefaultTreeModel;
import javax.swing.tree.TreeNode;
import javax.swing.tree.TreePath;
import javax.swing.tree.TreeSelectionModel;

/**
 *  Builds and contains the JTree used to display the class heirarchy.
 *
 *  @author     Jane Griscti jane@janeg.ca
 *  @created    January 26, 2002
 */
class CBTreePanel extends JPanel {

    private ClassBrowser parent;
    private JTree tree                              = new JTree();
    private DefaultMutableTreeNode classTree;
    private DefaultMutableTreeNode pkgTree;
    private CBClassGroup classGroup;
    private Collection sortedClasses                = new ArrayList();


    /**
     *  Constructs a CBTreePanel object.
     *
     *  @param  frame   the ClassBrowser object to contain the panel
     *  @param  cbcg    the CBClassGroup to be displayed
     */
    CBTreePanel( final ClassBrowser frame, final CBClassGroup cbcg ) {
        super();
        parent = frame;
        classGroup = cbcg;

        buildPkgTree();
        buildClassTree();
```

```
        switchToPkgTree();

        tree.putClientProperty( "JTree.lineStyle", "Angled" );
        tree.getSelectionModel().setSelectionMode(
TreeSelectionModel.SINGLE_TREE_SELECTION );
        tree.addTreeSelectionListener( new CBTreeListener() );

        JScrollPane jsp  = new JScrollPane( tree );
        jsp.setPreferredSize( new Dimension( 300, 500 ) );
        jsp.setMinimumSize( jsp.getPreferredSize() );
        add( jsp );
    }


    /**  Builds a tree model based on the class package names. */
    private void buildPkgTree() {
        DefaultMutableTreeNode top       = new DefaultMutableTreeNode(
classGroup.getGroupName() );
        DefaultMutableTreeNode prevNode;
        DefaultMutableTreeNode node;

        String element;
        String key                       = "";
        StringBuffer keyBuf              = new StringBuffer( "" );
        String keyBufStr;
        TreeMap map                      = new TreeMap();
        prevNode = top;
        String[] pkgs                    = classGroup.getByPackageName();

        // build tree nodes
        for( int i = 0; i < pkgs.length; i++ ) {
            element = pkgs[i];
            keyBuf = new StringBuffer( element.length() );
            keyBufStr = "";

            StringTokenizer stok  = new StringTokenizer( element, "/" );
            ClassInfo data        = null;
            int tokenCount        = 0;

            while( stok.hasMoreTokens() ) {
                key = stok.nextToken();
                tokenCount++;

                keyBuf.append( key + '.' );
                keyBufStr = keyBuf.toString();

                if( map.containsKey( keyBufStr ) ) {
                    prevNode = (DefaultMutableTreeNode)map.get( keyBufStr );
                } else {
```

```
                    data = new ClassInfo( keyBufStr, key );
                    node = new DefaultMutableTreeNode( data );

                    // check for top level package names
                    if( tokenCount == 1 ) {
                        top.add( node );
                    } else {
                        prevNode.add( node );
                    }

                    prevNode = node;
                    map.put( keyBufStr, node );
                    sortedClasses.add( data );
                }
            }
        }

        pkgTree = top;
    }


    /*
     *  Builds a tree model based on the class names.<p>
     *
     *  Note: This is not built by using the CBClassGroup sorted classes. It
     *  uses the same ClassInfo objects created for the package tree.
     */
    private void buildClassTree() {

        Collections.sort( (ArrayList)sortedClasses,
            CBNameComparator.getInstance() );
        ListIterator liter              = ( (ArrayList)sortedClasses
).listIterator();
        DefaultMutableTreeNode classTop  = new DefaultMutableTreeNode(
classGroup.getGroupName() );
        DefaultMutableTreeNode node;
        ClassInfo element;

        while( liter.hasNext() ) {
            element = (ClassInfo)liter.next();
            node = new DefaultMutableTreeNode( element );
            classTop.add( node );
        }

        sortedClasses = null;        // finished with sorted classes
        classTree = classTop;
    }
```

```
    /**
     *  Switches the JTree model to the sorted class tree model.
     *  The display is automatically updated.
     */
    void switchToClassTree() {
        DefaultTreeModel model  = (DefaultTreeModel)tree.getModel();
        model.setRoot( classTree );
        model.reload();
    }


    /**
     *  Switches the JTree model to the package name tree model.
     *  The display is  automatically updated.
     */
    void switchToPkgTree() {
        DefaultTreeModel model  = (DefaultTreeModel)tree.getModel();
        model.setRoot( pkgTree );
        model.reload();
    }


    /**
     *  The listener for the JTree contained in CBTreePanel.
     *
     *  @author     Jane Griscti jane@janeg.ca
     *  @created    January 26, 2002
     */
    private class CBTreeListener implements TreeSelectionListener {

        public void valueChanged( TreeSelectionEvent e ) {
            DefaultMutableTreeNode node  = (DefaultMutableTreeNode)
                tree.getLastSelectedPathComponent();

            if( node == null ) {
                return;
            }

            if( node.isLeaf() ) {
                ClassInfo classInfo  = (ClassInfo)node.getUserObject();
                parent.displayClassInfo( classInfo.qualifiedName );
            }
        }
    }


    /**
     *  Separates the class name from the package name and stores them
     *  separately. A ClassInfo object acts as a leaf node in the JTree.
```

```
     *
     *  @author     Jane Griscti jane@janeg.ca
     *  @created    January 5, 2002
     */
    class ClassInfo {

        String qualifiedName;
        String className;


        /**
         *  Constructs a new ClassInfo object
         *
         *  @param  fullpath  the fully qualifed class name
         *  @param  name      the simple class name
         */
        ClassInfo( String fullpath, String name ) {
            fullpath = fullpath.substring( 0, fullpath.length() - 1 );
            qualifiedName = fullpath;
            className = name;
        }

        public String getQualifiedName() {
            return qualifiedName;
        }


        /**
         *  Overrides Object.toString() to provide each node with a display
         *  name; that of the class it represents.
         *
         *@return    Description of the Returned Value
         */
        public String toString() {
            return className;
        }
    }

}
```

```java
package ca.janeg.cb;

import java.lang.reflect.Array;
import java.lang.reflect.Constructor;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Comparator;

/**
 *  A constructor group object contains class constructor information separated
 *  into groups based on their access privileges. Each grouping is sorted on the
 *  constructors simple name.
 *
 *@author     Jane Griscti jane@janeg.ca
 *@created    January 13, 2002
 */
class ConstructorGroup {

    private final Class owner;
    private Constructor[] ctors;
    private Constructor[] publicConstructors;
    private Constructor[] protectedConstructors;
    private Constructor[] packageConstructors;
    private Constructor[] privateConstructors;

    boolean hasCtors;


    /**
     *  Creates a ConstructorGroup object.
     *
     *@param  owner  the class object the methods are derived from
     */
    ConstructorGroup( final Class owner ) {
        this.owner = owner;
        ctors = owner.getDeclaredConstructors();
        Arrays.sort( ctors, NameComparator.getInstance() );

        hasCtors = Array.getLength( ctors ) > 0;
        if( hasCtors ) separateByAccess();
    }


    private void separateByAccess() {
        Object[] obj  = AccessSeparator.separate( ctors );

        ArrayList al  = (ArrayList)obj[0];
        publicConstructors = (Constructor[])al.toArray( new Constructor[0] );

        al = (ArrayList)obj[1];
        protectedConstructors = (Constructor[])al.toArray( new Constructor[0] );
```

```
        al = (ArrayList)obj[2];
        privateConstructors = (Constructor[])al.toArray( new Constructor[0] );

        al = (ArrayList)obj[3];
        packageConstructors = (Constructor[])al.toArray( new Constructor[0] );
    }



    Constructor[] getPublicConstructors() {
        return publicConstructors;
    }


    Constructor[] getProtectedConstructors() {
        return protectedConstructors;
    }


    Constructor[] getPrivateConstructors() {
        return privateConstructors;
    }


    Constructor[] getPackageConstructors() {
        return packageConstructors;
    }


    Constructor[] getAllConstructors() {
        return ctors;
    }

}
```

```
package ca.janeg.cb;

import java.lang.reflect.Array;
import java.lang.reflect.Field;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Comparator;

/**
 *  A field group object contains class field information separated into groups
 *  based on their access privileges. Each grouping is sorted on the fields
 *  simple name.
 *
 *@author     Jane Griscti jane@janeg.ca
 *@created    January 13, 2002
 */

class FieldGroup {

    private final Class owner;
    private Field[] flds;
    private Field[] publicFields;
    private Field[] protectedFields;
    private Field[] packageFields;
    private Field[] privateFields;

    boolean hasFields;


    /**
     *  Creates a new FieldGroup object.
     *
     *@param  owner  the class object the fields are derived from
     */
    FieldGroup( final Class owner ) {
        this.owner = owner;
        flds = owner.getDeclaredFields();
        Arrays.sort( flds, NameComparator.getInstance() );

        hasFields = Array.getLength( flds ) > 0;

        if( hasFields )  separateByAccess();
    }

    // separate fields based on their access level
    private void separateByAccess() {
        Object[] obj  = AccessSeparator.separate( flds );

        ArrayList al  = (ArrayList)obj[0];
        publicFields = (Field[])al.toArray( new Field[0] );
```

```
        al = (ArrayList)obj[1];
        protectedFields = (Field[])al.toArray( new Field[0] );

        al = (ArrayList)obj[2];
        privateFields = (Field[])al.toArray( new Field[0] );

        al = (ArrayList)obj[3];
        packageFields = (Field[])al.toArray( new Field[0] );
    }



    Field[] getPublicFields() {
        return publicFields;
    }


    Field[] getProtectedFields() {
        return protectedFields;
    }


    Field[] getPrivateFields() {
        return privateFields;
    }


    Field[] getPackageFields() {
        return packageFields;
    }


    Field[] getAllFields() {
        return flds;
    }

}
```

```
package ca.janeg.cb;

import java.lang.reflect.Array;
import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Comparator;
import java.util.StringTokenizer;

/**
 *  A method group object contains class method information separated into
 *  groups based on their access privileges. Each grouping is sorted on the
 *  methods simple name.
 *
 *  @author     Jane Griscti jane@janeg.ca
 *  @created    January 13, 2002
 */
class MethodGroup {

    private final Class owner;
    private Method[] methods;
    private Method[] publicMethods;
    private Method[] protectedMethods;
    private Method[] packageMethods;
    private Method[] privateMethods;

    boolean hasMethods;


    /**
     *  Creates a MethodGroup object.
     *
     *@param  owner  the class object the methods are derived from
     */
    MethodGroup( final Class owner ) {
        this.owner = owner;
        methods = owner.getDeclaredMethods();
        Arrays.sort( methods, NameComparator.getInstance() );

        hasMethods = Array.getLength( methods ) > 0;

        if( hasMethods ) separateByAccess();
    }


    // separate methods based on their access level
    private void separateByAccess() {
        Object[] obj  = AccessSeparator.separate( methods );
```

```
        ArrayList al  = (ArrayList)obj[0];
        publicMethods = (Method[])al.toArray( new Method[0] );

        al = (ArrayList)obj[1];
        protectedMethods = (Method[])al.toArray( new Method[0] );

        al = (ArrayList)obj[2];
        privateMethods = (Method[])al.toArray( new Method[0] );

        al = (ArrayList)obj[3];
        packageMethods = (Method[])al.toArray( new Method[0] );
    }



    Method[] getPublicMethods() {
        return publicMethods;
    }


    Method[] getProtectedMethods() {
        return protectedMethods;
    }


    Method[] getPrivateMethods() {
        return privateMethods;
    }


    Method[] getPackageMethods() {
        return packageMethods;
    }


    Method[] getAllMethods() {
        return methods;
    }

}
```

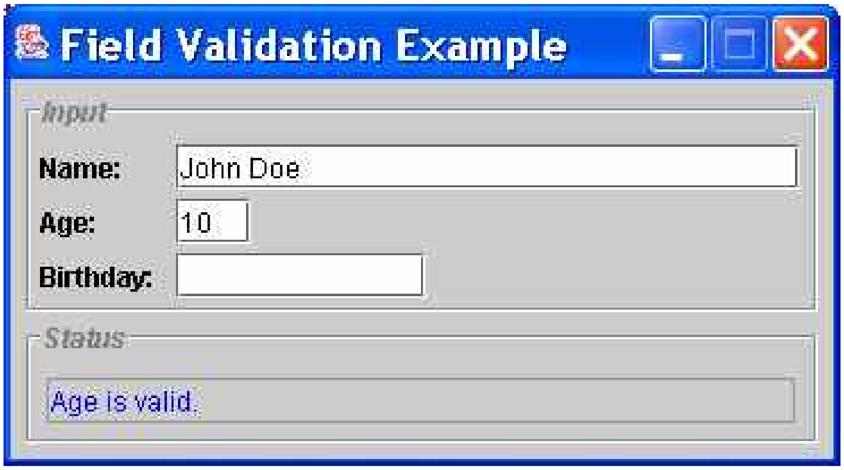
```
package ca.janeg.cb;

import java.util.Comparator;

/**
 *  Compares fully qualified class, constructor, field and method
 *  names based on their simple name; ignores character case.
 *
 *  @author     Jane Griscti jane@janeg.ca
 *  @created    January 13, 2002
 */

class NameComparator implements Comparator {

    private final static NameComparator INSTANCE  = new NameComparator();


    /*
     *  Ensure only one NameComparator is created (Singleton)
     */
    private NameComparator() { }


    private String getDelimiter( final String str ) {
        String delimiter  = "";
        if( str.indexOf( "/" ) > 0 ) {
            delimiter = "/";
        } else if( str.indexOf( "." ) > 0 ) {
            delimiter = ".";
        }

        return delimiter;
    }


    private String extract( final String str, final String delimiter ) {
        String result  = str;

        // drop any parameters if it's a method or constructor name
        if( str.indexOf( "(" ) > 0 ) {
            result = str.substring( 0, str.indexOf( "(" ) );
        }

        if( delimiter != "" ) {
            int index  = result.lastIndexOf( delimiter );
            result = result.substring( index + 1 );
        }

        return result;
    }
```

```
    /**
     *  Returns a singleton instance of NameComparator
     *
     *@return    a NameComparator object
     */
    public static NameComparator getInstance() {
        return INSTANCE;
    }


    /**
     *  Compares two objects
     *
     *@param  o1  the first object being compared
     *@param  o2  the second object being compared
     *@return     a negative integer, zero, or a positive integer as the first
     *      argument is less than, equal to, or greater than the second.
     */
    public int compare( final Object o1, final Object o2 ) {
        String s1           = o1.toString();
        String s2           = o2.toString();

        String s1Delimiter  = getDelimiter( s1 );
        String s2Delimiter  = getDelimiter( s2 );

        s1 = extract( o1.toString(), s1Delimiter );
        s2 = extract( o2.toString(), s2Delimiter );

        return s1.compareToIgnoreCase( s2 );
    }
}
```

```
package ca.janeg.cb;

import java.util.StringTokenizer;

/**
 *  A ParsedClassName takes a fully qualified class name and breaks into it's
 *  component parts using the given delimiter.
 *
 *@author     Jane Grisct jane@janeg.ca
 *@created    January 26, 2002
 */
class ParsedClassName {

    private String simple;
    private String[] pkgs;
    private String pkgName;


    ParsedClassName( final String name, final String delimiter ) {

        StringTokenizer stok  = new StringTokenizer( name, delimiter );
        int tokens            = stok.countTokens();

        if( tokens > 1 ) {
            StringBuffer buf  = new StringBuffer( name.length() );
            pkgs = new String[tokens - 1];
            String tok        = "";

            for( int i = 0; i < tokens - 1; i++ ) {
                tok = stok.nextToken();
                pkgs[i] = tok;
                buf.append( tok + '.' );
            }
            pkgName = buf.substring( 0, buf.length() - 1 );
        }
        simple = stok.nextToken();
    }


    String getSimpleName() {
        return simple;
    }


    String[] getPackages() {
        return pkgs;
    }


    String getPackageName() {
```

```
        return pkgName;
    }


    /**
     *  The main program for the ParsedClassName class; used for testing.
     *
     *@param  args  The command line arguments
     */
    public static void main( String[] args ) {
        // good example
        ParsedClassName pcn  = new ParsedClassName(
            "java.awt.text.resources.DateFormatZoneData_en", "." );
        System.out.println( pcn.getSimpleName() );
        System.out.println( pcn.getPackageName() );
        for( int i = 0; i < pcn.pkgs.length; i++ ) {
            System.out.println( pcn.pkgs[i] );
        }
        System.out.println();

        // works ok with empty tokens
        pcn = new ParsedClassName( "java..awt.Button", "." );
        System.out.println( pcn.getSimpleName() );
        System.out.println( pcn.getPackageName() );
        for( int i = 0; i < pcn.pkgs.length; i++ ) {
            System.out.println( pcn.pkgs[i] );
        }

        // works ok with ending delimiter
        System.out.println();
        pcn = new ParsedClassName( "java..awt.Frame.", "." );
        System.out.println( pcn.getSimpleName() );
        System.out.println( pcn.getPackageName() );
        for( int i = 0; i < pcn.pkgs.length; i++ ) {
            System.out.println( pcn.pkgs[i] );
        }

    }
}
```

Java Project - FieldValidation



FieldValidation.java
Utils.java

Each field is assigned an `InputVerifier` which checks the contents of a field when it is exited. If the input does not fall within the verifiers parameters, focus is automatically returned to the field. A corresponding message is displayed in the 'status' area.

The `static` method `center()` from the `Utils` class is used to center the window on the desktop.

Home | Projects

FieldValidation

```java
package ca.janeg.project;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;

import javax.swing.BorderFactory;
import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.InputVerifier;
import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.border.Border;
import javax.swing.border.TitledBorder;

import ca.janeg.swing.Utils;

/**
 *  An example of validating user input fields using
 *  <code>javax.swing.InputVerifier</code>.
 *
 *  The verifiers are defined as inner classes.
 *
 *  References:
 *  <ul>
 *      <li><a href="http://www.javaworld.com/javaworld/jw-06-2001/jw-0622-traps.html?">
 *      JavaWorld article by Michael Daconta</a></li>
 *      <li><a href="http://developer.java.sun.com/developer/JDCTechTips/2001/tt1120.html">
 *      JDC Tech Tip - VALIDATING NUMERICAL INPUT IN A JTEXTFIELD</a></li>
 *
 *  </ul>
 *  @author Jane Griscti, jane@janeg.ca
 */
public class FieldValidation {

    private final static DateFormat dateFormat =
                        new SimpleDateFormat( "MM/dd/yyyy" );
    private final  Font font = new Font( null,
                              Font.BOLD | Font.ITALIC,
                              12 );

```

```
    private final JFrame frame          = new JFrame();
    private final JTextField name       = new JTextField( 25 );
    private final JTextField age        = new JTextField( 3 );
    private final JTextField birthday   = new JTextField( 10 );
    private final JTextField status     = new JTextField( 30 );

    public FieldValidation(){
        frame.setTitle( "Field Validation Example" );

        // assign a verifier to each input field
        age.setInputVerifier( new AgeVerifier() );
        birthday.setInputVerifier( new BirthdayVerifier() );
        name.setInputVerifier( new BlankFieldVerifier() );

        buildGUI();
    }

    /*
     *  Build the example GUI.
     */
    private void buildGUI(){

        JPanel mainPanel = new JPanel();
        mainPanel.setLayout( new BoxLayout( mainPanel, BoxLayout.Y_AXIS ) );
        mainPanel.setBorder( BorderFactory.createCompoundBorder(
                               BorderFactory.createEmptyBorder( 5,5,5,5 ),
                               mainPanel.getBorder() ) );

        mainPanel.add( buildInputPanel() );
        mainPanel.add( buildStatusPanel() );

        frame.getContentPane().add( mainPanel, BorderLayout.CENTER );

        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent wevt) {
                System.exit(0);
            }
        });

        frame.setResizable( false );
        frame.pack();
        Utils.center( frame );
        frame.setVisible( true );
    }

    /*
     *  Build the GUI input panel.
     */
    private JPanel buildInputPanel(){
        JPanel panel = new JPanel();

        Border border = BorderFactory.createTitledBorder(
                                BorderFactory.createEtchedBorder(),
                                "Input",
                                TitledBorder.LEADING,
```

```
                                TitledBorder.TOP,
                                font,
                                Color.GRAY );

        panel.setLayout( new BoxLayout( panel,
                                        BoxLayout.Y_AXIS ) );
        panel.setBorder( border );

        panel.add( buildField( name, "Name:" ) );
        panel.add( buildField( age, "Age:" ) );
        panel.add( buildField( birthday, "Birthday:" ) );

        return panel;
    }

    /*
     *  Build an input field to be displayed in the input panel.
     */
    private JPanel buildField( JComponent comp, String label ){

        comp.setMinimumSize( comp.getPreferredSize() );
        comp.setMaximumSize( comp.getPreferredSize() );

        JPanel panel = new JPanel();
        panel.setBorder( BorderFactory.createEmptyBorder( 2,2,2,2 ) );

        panel.setLayout( new BoxLayout( panel,
                                        BoxLayout.X_AXIS ) );

        Box leftBox = new Box( BoxLayout.X_AXIS );
        leftBox.setPreferredSize( new Dimension( 60, 20 ) );
        leftBox.add( new JLabel( label ) );

        Box rightBox = new Box( BoxLayout.X_AXIS );
        rightBox.add( comp );

        panel.add( leftBox );
        panel.add( rightBox );
        panel.add( Box.createHorizontalGlue() );

        return panel;
    }

    /*
     *  Build the GUI status panel.
     */
    private JPanel buildStatusPanel(){
        JPanel panel = new JPanel();

        Border border = BorderFactory.createTitledBorder(
                                BorderFactory.createEtchedBorder(),
                                "Status",
                                TitledBorder.LEADING,
                                TitledBorder.TOP,
                                font,
```

```
                                Color.GRAY );
        panel.setBorder( border );

        status.setEditable( false );
        status.setForeground( Color.BLUE );
        status.setText( "Ready" );
        panel.add( status );
        return panel;
    }

    /*
     *  Checks to ensure a field is not blank.
     *
     *  The 'shouldYieldFocus()' method produces
     *  a 'beep' if the validation fails. It is inherited
     *  by the other field verifiers.
     */
    private class BlankFieldVerifier extends InputVerifier {

        public boolean verify(JComponent comp) {
            JTextField fld = (JTextField) comp;
            String content = fld.getText();

            boolean isValid = true;
            if (content.length() == 0) {
                status.setText("Field cannot be blank.");
                isValid = false;
            }

            return isValid;
        }

        public boolean shouldYieldFocus(JComponent input) {
            boolean valid = super.shouldYieldFocus(input);

            if (!valid) {
                frame.getToolkit().beep();
            }
            return valid;
        }

    }

    /*
     *  Checks the age field to ensure it is not
     *  empty and that it contains an integer value.
     */
    private class AgeVerifier extends BlankFieldVerifier {

        public boolean verify(JComponent comp) {

            JTextField fld = (JTextField) comp;
            String content = fld.getText();

            boolean isValid = true;
```

```

            try {
                Integer.parseInt(content);
            } catch (NumberFormatException nfe) {
                fld.setText("");
                status.setText("Age must be a number.");
                isValid = false;
            }

            if (isValid) {
                status.setText("Age is valid.");
            }

            return isValid;
        }

    }

    /*
     *  Checks the birthday field to ensure it is not blank
     *  and it contains a valid date string. There is no
     *  range checking on the date.
     */
    private class BirthdayVerifier extends BlankFieldVerifier {
        public boolean verify(JComponent comp) {

            JTextField fld = (JTextField) comp;
            String content = fld.getText();

            boolean isValid = true;
            try {
                dateFormat.parse(content);
            } catch (ParseException e) {
                fld.setText("");
                status.setText("Birthday must be mm/dd/yyyy.");
                isValid = false;
            }

            if (isValid) {
                status.setText("Birthday is valid.");
            }
            return isValid;
        }
    }

    /**
     *  Main entry point for the class.
     */
    public static void main(String[] args){
        new FieldValidation();
    }

}

```

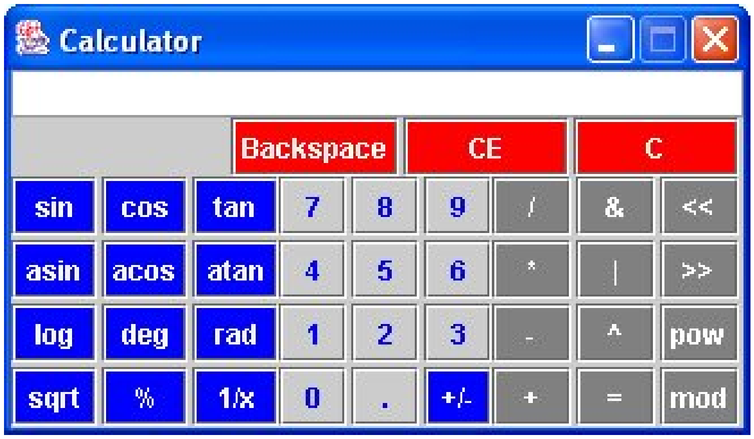
FieldValidation

Utils

```java
package ca.janeg.swing;
import java.awt.Dimension;
import java.awt.Toolkit;
import java.awt.Window;

/**
 *  Utility methods for Swing components.
 *
 *
 *  @author Jane Griscti, jane@janeg.ca
 */
public class Utils {

    /**
     *  Center a component on the screen.
     *
     *  Source:
     *  <a href="http://javaalmanac.com/egs/java.awt/screen_CenterScreen.html">
     *  The Java Almanac</a>
     *  @param window the component to be centered.
     */
    public static void center( Window window ) {

        // Get the size of the screen
        Dimension dim = Toolkit.getDefaultToolkit().getScreenSize();

        // Determine the new location of the window
        int w = window.getSize().width;
        int h = window.getSize().height;
        int x = (dim.width - w) / 2;
        int y = (dim.height - h) / 2;

        // Move the window
        window.setLocation(x, y);

    }

}
```

Utils

Java Project - Calculator



Calculator.java
CalculatorEngine.java

This is a simple implementation of a Calculator. I started with some code I found in *Object-Oriented Programming and Java* by Danny C.C. Poo and Derek B.K. Kiong which implemented the four binary operations: + - / * and = in the class `CalculatorEngine`. I added the unary functions and built a Swing GUI.

Design Decisions

- CalculatorEngine

  The original code returned `Double.toString( value )`. This worked fine from the command line but gave me problems when I was designing the GUI; exponential numbers were being returned.

  I then tried using a `JFormattedTextField` in the GUI with a `DecimalFormat`. This also presented difficulties. The default pattern for `DecimalFormat` is `"#,##0.0#"`. The display always showed `0.0`. I only wanted to show decimal digits if the user had selected the decimal key. I changed the pattern to `#,###.#"` and invoked `setDecimalSeperatorAlwaysShown( false )` but then the decimal did not show up until the user selected another digit key and if that happened to be a zero, in any decimal position, it was not shown until a number between 1 and 9 was selected.

  In the end I gave up and decided to modify `CalculatorEngine`, adding the `display` field, a `NumberFormatter` and modifying the code to keep the `value` and `display` attributes in sync.

- Calculator

  The key to the GUI is displaying the various buttons in a pleasing manner and finding an easy way to invoke their actions. By default, each `JButton`'s action command is set to the value of the button label. This got me thinking about how nice it would be if, when a user selected the `cos` button, the button action listener could invoke `engine.cos()`. The reflection mechanism in Java allows for just such a scenario.

  I also wanted the buttons appearance to vary according to their functions: digit, unary, binary, control. To accomplish this I created an inner class `CalcButton` which implements `ActionListener` and then created a number of subclasses to handle the different colour settings for each function.

All in all the whole thing came out fairly clean<g>. There is one small flaw that I'm aware of, if the result of a unary operation such as `mod` is zero, the display shows nothing when really it should show a '0'. Haven't figured out how to get around this yet. If you have a solution, please let me know <g>

Home | Projects

Calculator

```
/* ***************************************************************************
 *
 *       File:  Calculator.java
 *    Package:  ca.janeg.calc
 *
 *   Contains:  Inner classes
 *                  CalcButton
 *                      DigitButton
 *                      FunctionButton
 *                      UnaryButton
 *                      ControlButton
 *
 * References:   Visual Components: Sum It Up with JCalculator
 *               by Claude Duguay,
 *               Article at http://archive.devx.com
 *               (Layout)
 *
 *               The Java Programming Language: 2nd Edition
 *               by Ken Arnold and James Gosling
 *               Addison-Wesley, 1998, 7th Printing 2000 (p311)
 *
 *               The Java Developers Almanac 1.4 (online)
 *                  http://javaalmanac.com/egs/java.awt/screen_CenterScreen.html
 *                  http://www.javaalmanac.com/egs/javax.swing/LookFeelNative.html
 *
 * Date         Author          Changes
 * ------------ -------------   ----------------------------------------------
 * Oct 17, 2002 Jane Griscti    Created
 * Oct 22, 2002 Jane Griscti    Cleaned up comments, layouts and action listener
 * Oct 23, 2002 Jane Griscti    changed CalcButton to use a white foreground as
 *                              the default button color and removed redundant
 *                              calls from the subclasses
 *                              re-arranged the code in the class body to place
 *                              inner classes after all methods except main()
 * *************************************************************************** */


package ca.janeg.calc;

import java.awt.Color;
import java.awt.Component;
import java.awt.Container;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.GridLayout;
import java.awt.Insets;
import java.awt.Toolkit;
import java.awt.Window;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
```

```
import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.UIManager;
import javax.swing.UnsupportedLookAndFeelException;

/**
 *  A GUI interface for <code>CalculatorEngine</code>.
 *
 *  @author     Jane Griscti    jane@janeg.ca
 *  @version    1.0             Oct 17, 2002
 */
public class Calculator extends JFrame {
    private final Class ENGINE;
    private final CalculatorEngine engine = new CalculatorEngine();
    private final JTextField display = new JTextField();

    /**
     *  Create a new calculator instance.
     */
    public Calculator(){
        setDefaultCloseOperation( JFrame.DISPOSE_ON_CLOSE );
        setTitle( "Calculator" );

        display.setEditable( false );
        display.setBackground( Color.WHITE );

        // set up a Class object used in actionPerformed()
        // to invoke methods on the CalculatorEngine
        ENGINE = engine.getClass();

        buildGUI();
        pack();
        setResizable( false );
        setLAF();
        center( this );
        setVisible( true );
    }

    private void buildGUI(){

        Container cp = getContentPane();
        cp.setLayout( new BoxLayout( cp, BoxLayout.Y_AXIS ) );

        cp.add( display );
        cp.add( buildControlPanel() );
        cp.add( buildButtonPanels() );
    }

    private JPanel buildControlPanel(){
        JPanel panel = new JPanel();
```

```
        panel.setLayout(new BoxLayout(panel, BoxLayout.X_AXIS));

        panel.add( Box.createHorizontalGlue() );
        panel.add( new ControlButton( "Backspace", "backspace" ) );
        panel.add( Box.createRigidArea( new Dimension( 2, 0 ) ) );

        JPanel panel2 = new JPanel( new GridLayout( 1, 1, 2, 2 ) );
        panel2.add( new ControlButton( "CE", "clearEntry" ) );
        panel2.add( new ControlButton( "C", "clear" ) );
        panel.add( panel2 );

        return panel;
    }

    private JPanel buildButtonPanels() {
        JPanel buttons = new JPanel();
        buttons.setLayout(new BoxLayout(buttons, BoxLayout.X_AXIS));
        buttons.setFont(new Font("Courier", 10, Font.BOLD));

        buttons.add( buildUnaryPanel() );
        buttons.add( buildDigitPanel() );
        buttons.add( buildFunctionPanel() );

        return buttons;
    }

    private JPanel buildDigitPanel(){
        JPanel panel = new JPanel();
        panel.setLayout( new GridLayout( 4, 3, 2, 2 ) );

        panel.add( new DigitButton( "7" ) );
        panel.add( new DigitButton( "8" ) );
        panel.add( new DigitButton( "9" ) );

        panel.add( new DigitButton( "4" ) );
        panel.add( new DigitButton( "5" ) );
        panel.add( new DigitButton( "6" ) );

        panel.add( new DigitButton( "1" ) );
        panel.add( new DigitButton( "2" ) );
        panel.add( new DigitButton( "3" ) );

        panel.add( new DigitButton( "0" ) );
        panel.add( new DigitButton( "." ) );

        // not a digit but added here to balance out the panel
        panel.add( new UnaryButton( " +/- ", "sign" ) );

        return panel;
    }


    private JPanel buildFunctionPanel(){
        JPanel buttons = new JPanel( new GridLayout( 4, 3, 2, 2 ) );
```

```
        buttons.add( new FunctionButton( "/", "divide" ) );
        buttons.add( new FunctionButton( "&", "and" ) );
        buttons.add( new FunctionButton( "<<", "leftShift" ) );

        buttons.add( new FunctionButton( "*", "multiply" ) );
        buttons.add( new FunctionButton( "|", "divide" ) );
        buttons.add( new FunctionButton( ">>", "rightShift" ) );

        buttons.add( new FunctionButton( "-", "subtract" ) );
        buttons.add( new FunctionButton( "^" , "xor" ) );
        buttons.add( new FunctionButton( "pow" ) );

        buttons.add( new FunctionButton( "+", "add" ) );
        buttons.add( new FunctionButton( "=", "equals" ) );
        buttons.add( new FunctionButton( "mod" ) );

        return buttons;

    }

    private JPanel buildUnaryPanel(){
        JPanel buttons = new JPanel( new GridLayout( 4, 3, 2, 2 ) );

        buttons.add( new UnaryButton( "sin" ) );
        buttons.add( new UnaryButton( "cos" ) );
        buttons.add( new UnaryButton( "tan" ) );
        buttons.add( new UnaryButton( "asin" ) );

        buttons.add( new UnaryButton( "acos" ) );
        buttons.add( new UnaryButton( "atan" ) );
        buttons.add( new UnaryButton( "log" ) );
        buttons.add( new UnaryButton( "deg", "degrees" ) );

        buttons.add( new UnaryButton( "rad", "radians" ) );
        buttons.add( new UnaryButton( "sqrt" ) );

        buttons.add( new UnaryButton( "%", "percent" ) );
        buttons.add( new UnaryButton( "1/x", "reciprocal" ) );

        return buttons;

    }

    /*
     *  Center a component on the screen.
     *
     *  @param window the component to be centered.
     */
    private void center( Window window ) {

        // Get the size of the screen
        Dimension dim = Toolkit.getDefaultToolkit().getScreenSize();

        // Determine the new location of the window
        int w = window.getSize().width;
```

```
        int h = window.getSize().height;
        int x = (dim.width - w) / 2;
        int y = (dim.height - h) / 2;

        // Move the window
        window.setLocation(x, y);
    }

    /*
     *  Set the Look and Feel to the system look and feel.
     */
    private void setLAF() {
        // Get the native look and feel class name
        String nativeLF = UIManager.getSystemLookAndFeelClassName();

        // Install the look and feel
        try {
            UIManager.setLookAndFeel(nativeLF);
        } catch (InstantiationException e) {
            System.out.println( e.getMessage() );
        } catch (ClassNotFoundException e) {
            System.out.println( e.getMessage() );
        } catch (UnsupportedLookAndFeelException e) {
            System.out.println( e.getMessage() );
        } catch (IllegalAccessException e) {
            System.out.println( e.getMessage() );
        }
    }


    /*
     *  Helper class to handle button formatting.
     *  Each button acts as its own listener.
     */
    private class CalcButton extends JButton implements ActionListener{

        CalcButton( String s, String action ){
            super( s );
            setActionCommand( action );
            setMargin( new Insets( 2, 2, 2, 2 ) );
            setForeground( Color.WHITE );
            addActionListener( this );
        }

        /*
         *  Captures the button events and then uses 'reflection'
         *  to invoke the right method in the calculator engine
         *
         *  Digit buttons are handled slightly different as they
         *  all use the digit( int ) method and their values must
         *  be passed as arguments.
         *
         *  The digit button for the decimal has special handling;
         *  new Integer( "." ) throws a NumberFormatException,
         *  have to use new Integer( '.' ) which converts the ASCII
```

```
      *  value of '.' to an integer.
      *
      */

     public void actionPerformed(ActionEvent e) {

         String methodName = e.getActionCommand();

         Method method = null;

         try {
             if ( e.getSource() instanceof DigitButton ) {
                 method =
                     ENGINE.getMethod("digit", new Class[] { int.class });

                 if (methodName.equals(".")) {
                     method.invoke(engine, new Object[] { new Integer( '.' )});
                 } else {
                     method.invoke(engine, new Object[] {
                                             new Integer( methodName )});
                 }
             } else {
                 method = ENGINE.getMethod(methodName, null);
                 method.invoke(engine, null);
             }
         } catch (NoSuchMethodException ex) {
             System.out.println("No such method: " + methodName);
         } catch (IllegalAccessException ea) {
             System.out.println("Illegal access" + methodName);
         } catch (InvocationTargetException et) {
             System.out.println("Target exception: " + methodName);
         }

         display.setText(engine.display());
     }
 }

 private class DigitButton extends CalcButton {
     DigitButton( String s ){
         super( s, s );
         setForeground( Color.BLUE );
     }
 }

 private class FunctionButton extends CalcButton {
     FunctionButton( String s ){
         this( s, s );
     }

     FunctionButton( String s, String action ){
         super( s, action );
         setBackground( Color.GRAY );
     }
 }
```

```
    private class ControlButton extends CalcButton{
        ControlButton( String s ){
            this( s, s );
        }

        ControlButton( String s, String action ){
            super( s, action );
            setBackground( Color.RED );
        }
    }

    private class UnaryButton extends CalcButton {
        UnaryButton( String s ){
            this( s, s );
        }

        UnaryButton( String s, String action ){
            super( s, action );
            setBackground( Color.BLUE );
        }
    }

    /**
     *  Main entry point for the program
     */
    public static void main(String[] args) {
        new Calculator();
    }
}
```

Calculator

CalculatorEngine

```
/* ***************************************************************************
 *
 *        File:  CalculatorEngine.java
 *     Package:  ca.janeg.calc
 *
 * References:  Object Oriented Programming and Java,
 *              by Danny C.C. Poo and Derek B.K. Kiong, Springer, 1999 (p48-49)
 *
 *
 * Date         Author           Changes
 * ------------ -------------    ----------------------------------------------
 * Oct 17, 2002 Jane Griscti    Created
 * Oct 18, 2002 Jane Griscti    Added unary functions %, sqrt, reciprocal, etc
 * Oct 20, 2002 Jane Griscti    Added var display, number formatter and related
 *                              methods
 *                              Added integer binary operations: xor, or, and
 *                              leftShift, rightShift
 * Oct 21, 2002 Jane Griscti    Cleaned up comments
 * Oct 22, 2002 Jane Griscti    Added trig and log unary functions
 * *************************************************************************** */

package ca.janeg.calc;

import java.text.DecimalFormat;
import java.text.NumberFormat;

/**
 *  A class to perform standard calculator operations.
 *  For example,
 *
 *  <pre>
 *      CalculatorEngine c = new CalculatorEngine();
 *      c.digit( 1 );
 *      c.digit( 2 );
 *      c.add();
 *      c.digit( 1 );
 *      c.digit( 3 );
 *      c.equals();
 *      System.out.println( c.display() );
 *  </pre>
 *
 *  Accuracy is limited to fifteen decimal places.
 *
 *  @author     Jane Griscti    jane@janeg.ca
 *  @version    1.2             Oct 20, 2002
 */
public class CalculatorEngine {

    private StringBuffer display    = new StringBuffer( 64 );
    private DecimalFormat df        = (DecimalFormat)NumberFormat.getInstance();
    private boolean newOp           = false;
    private boolean inDecimals      = false;
```

```
    private double  value;          // current digits
    private double  keep;           // previous value or operation result
    private int     toDo;           // binary operation waiting for 2nd value
    private int     decimalCount;   // number of decimal positions in current
                                    //   value

    /**
     *  Creates a new <code>CalculatorEngine</code> object.
     */
    public CalculatorEngine(){
        super();
        df.setMaximumFractionDigits( 15 );
    }

    /* -- Digits and the decimal point handler -- */

    /**
     *  Accept a digit or decimal as input.
     */
    public void digit(final int n ){

        /*
         *  Strategy:
         *      1. Start a new value if at the beginning of a new operation.
         *
         *      2. Append the input character, setting the decimal flag if it's
         *         a decimal point or increasing the decimal count if we're
         *         already into decimals.
         *
         *      3. Convert the revised input string to a double for use in
         *         calculations; forcing input errors to return a 0.0 value.
         */

        if( newOp ){
            display.delete( 0, display.length() );
            newOp = false;
        }

        char c = (char)n;

        if( c == '.' ){
            display.append( '.' );
            inDecimals = true;
        }else if( !inDecimals ){
            display.append( n );
        }else{
            if( decimalCount < 16 ){
                display.append( n );
                decimalCount++;
            }
        }

        try{
            value = Double.parseDouble( display.toString() );
```

```
        }catch( NumberFormatException e ){
            value = Double.parseDouble( "0.0" );
        }
    }

    /* -- Binary operations --
     *
     *    A binary operation signals the engine to:
     *      1. store the current value
     *      2. set the 'toDo' flag with the requested operation
     *      3. accept input for a second value
     *      4. perform the 'toDo' op when '=' or another binary operation
     *         is requested
     */

    /**
     *  Add the next input value to the previous value
     */
    public void add(){
        binaryOperation( "+" );
    }

    /**
     *  Subtract the next input value from the previous value
     */
    public void subtract(){
        binaryOperation( "-" );
    }

    /**
     *  Multiply the next input value by the previous value
     */
    public void multiply(){
        binaryOperation( "*" );
    }

    /**
     *  Divide the previous value by the next input value
     */
    public void divide(){
        binaryOperation( "/" );
    }

    /**
     *  Bitwise And ( & )
     */
    public void and(){
        binaryOperation( "&" );
    }

    /**
     *  Bitwise Or ( | )
     */
    public void or(){
        binaryOperation( "|" );
```

```
    }

    /**
     *  Bitwise ( ^ )
     */
    public void xor(){
        binaryOperation( "^" );
    }

    /**
     *  Bitwise left shift ( < )
     */
    public void leftShift(){
        binaryOperation( "<" );
    }

    /**
     *  Bitwise right shift ( > )
     */
    public void rightShift(){
        binaryOperation( ">" );
    }

    /**
     *  Modulous ( % )
     */
    public void mod(){
        binaryOperation( "m" );
    }

    /**
     *  Raise the previous value to the 'power; of the next input value
     */
    public void pow(){
        binaryOperation( "p" );
    }

    /**
     *  Perform any waiting binary operation and clear previous value
     */
    public void equals(){
        compute();
        toDo = 0;
        newOp = true;
    }

    /*
     *  Setup registers for next input value
     */
    private void binaryOperation( final String op ){

        if( toDo == 0 ){
            keep = value;
        }else{
            compute();
```

```
        }

        value = 0;
        toDo = op.hashCode();
        resetDecimals();
        setDisplay();
    }

    /*
     *  Perform a binary operation
     */
    private void compute(){

        switch( toDo ){
            case '+':   value = keep + value;   break;
            case '-':   value = keep - value;   break;
            case '*':   value = keep * value;   break;
            case '/':
                if( value != 0 ){           // ignore divide by zero
                    value = keep / value;
                }
            case '&':   value = (int)keep & (int)value;     break;
            case '|':   value = (int)keep | (int)value;     break;
            case '^':   value = (int)keep ^ (int)value;     break;
            case '<':   value = (int)keep << (int)value;    break;
            case '>':   value = (int)keep >> (int)value;    break;
            case 'm':   value = keep % value;               break;
            case 'p':   value = Math.pow( keep, value );    break;
        }

        keep = value;
        setDisplay();
    }

    /* -- Unary Operations -- */

    /**
     *  Compute the square of the current value
     */
    public void sqrt(){
        value = Math.sqrt( value );
        unaryOperation();
    }

    /**
     *  Reverse the sign on the current value
     */
    public void sign(){
        value = value * -1;
        unaryOperation();
    }

    /**
     *  Convert the current value to a percent
     */
```

```
    public void percent(){
        value = value / 100;
        unaryOperation();
    }

    /**
     *  Convert the current value to it's reciprocal value
     */
    public void reciprocal(){
        if( value > 0 ){
            value = 1 / value;
        }else{
            value = 0;
        }
        unaryOperation();
    }

    /**
     *  Compute the sine of the current value.
     */
    public void sin(){
        value = Math.sin( value );
        unaryOperation();
    }

    /**
     *  Compute the cosine of the current value
     */
    public void cos(){
        value = Math.cos( value );
        unaryOperation();
    }

    /**
     *  Compute the tan of the current value
     */
    public void tan(){
        value = Math.tan( value );
        unaryOperation();
    }

    /**
     *  Compute the asine of the current value
     */
    public void asin(){
        value = Math.asin( value );
        unaryOperation();
    }

    /**
     *  Compute the acosine of the current value
     */
    public void acos(){
        value = Math.acos( value );
        unaryOperation();
```

```
    }

    /**
     *  Compute the atan of the current value
     */
    public void atan(){
        value = Math.atan( value );
        unaryOperation();
    }

    /**
     *  Compute the log of the current value
     */
    public void log(){
        value = Math.log( value );
        unaryOperation();
    }

    /**
     *  Convert the current value to degrees
     */
    public void degrees(){
        value = Math.toDegrees( value );
        unaryOperation();
    }

    /**
     *  Convert the current value to radians
     */
    public void radians(){
        value = Math.toRadians( value );
        unaryOperation();
    }


    /*
     *  Setup flag to signal start of a new operation and
     *  set the display to match the value generated by a
     *  unary operation
     */
    private void unaryOperation(){
        newOp = true;
        setDisplay();
    }

    /* -- Control operations -- */

    /**
     *  Delete the last entered digit
     */
    public void backspace(){
        display.deleteCharAt( display.length() - 1 );
        value = Double.parseDouble( display.toString() );
        setDisplay();
    }
```

```
    /**
     *  Clear all values
     */
    public void clear(){
        display.delete( 0, display.length() );
        value = 0;
        keep = 0;
        toDo = 0;
        resetDecimals();
    }

    /**
     *  Clear the current value
     */
    public void clearEntry(){
        display.delete( 0, display.length() );
        value = 0;
        resetDecimals();
    }

    /*
     *  Reset the decimal flag and counter
     */
    private void resetDecimals(){
        inDecimals = false;
        decimalCount = 0;
    }

    /**
     *  Convert the current value to a formatted string for
     *  display
     */
    private void setDisplay(){
        if( value == 0 ){
            display.delete( 0, display.length() );
        }else{
            display.replace( 0, display.length(), df.format( value ) );
        }
    }

    /**
     *  Returns the current value as a decimal formatted string
     */
    public String display(){
        return display.toString();
    }

}
```

CalculatorEngine

Java Project - CalendarComboBox

CalendarComboBox.java

As a Notes developer, I've gotten used to having a date input box with a perpetual calendar. I thought it would be nice to have one for my Java projects. It turned out to be less difficult to create than I'd originally imagined.

The one truly nice thing about Java is the richness of it's API. I was able to create the `CalendarComboBox` by simply arranging a number of existing components: `JFormattedTextField, BasicArrowButton, JTable,` and `Popup`. Of course, code always looks simple once it's finished. Originally I didn't know the `BasicArrowButton` and `Popup` classes even existed. It took some poking around in the API and Java source code related to `JComboBox` before I tracked them down.

I also needed to figure out how to build an array to hold the days in a month and leverage the various date related classes: `Calendar, GregorianCalendar, DateFormat,` and `DateFormatSymbols.` Mr. Dunn's book, *Java Rules* was particularly useful in helping me understand how these classes worked.

And last, but not least, were the layout experiments. I got stuck for a few hours on the calendar display; the buttons in the navigation panel kept changing size, it was very distracting. Finally realized that part of the problem was the `JLabel` component I was using to display the month and year name and the fact that I was using a `BoxLayout`. Once I changed the label to a `JTextField` and the calendar panel layout to `BorderLayout`, with the navigation portion placed in `BorderLayout.NORTH` and the table in `BorderLayout.CENTER` the display started to behave itself.

I ran across a few other snags, they are hightlighted in the code comments. Below are my reasons for designing the class as I did.

Design Decisions

- **Class fields**

  The values represented by these fields are common to the system the class is running on. The data is based on the system `Locale` which is not likely to change; at least, not during the active life of a running application.

- **Field access modifiers**

  All fields (except `popup`) are declared `private` and `final`. This is good coding practice.

  The `private` keyword helps to enforce encapsulation and forces you to think about your classes public interface. In this case, only one field, `current` needed to be publicly exposed; a gettor method, `public Calendar getDate()` was provided to return `current` as it's reasonable to assume an external class would need access to the currently selected date.

The keyword `final` emphasizes that the fields are required and that references cannot be accidently modifed during the life of an object. It also notifies the compiler that the code relating to these values can be safely optimized. Another advantage is that it helps ensure that everything the object requires to work correctly will be available once it is created; if you fail to initialize a final variable during object creation the compiler complains.

- **Why `popup` isn't `final`**

  The API recommends using `PopupFactory` to create `Popup` objects. `PopupFactory` caches popup objects, managing their reuse and disposal. As the programmer's at Sun have been kind enough to supply me with a class that can manage popup's it seemed sensible to use it rather than create a `final` popup reference and attempt to manage it myself.

- **Listeners as inner classes**

  There are three basic ways to implement listeners: as external classes, as inner classes or as anonymous classes. The only reason to implement one as an external class is if it could possibly be used by another class; yet listeners are generally very specific in nature and certainly are specific in this case so there was nothing to be gained by implementing them as external classes.

  Anonymous listener classes are generally used if they are required by only one element in the class and if they can be written in nine or ten lines of code. When I started writing the class I had no idea how long a particular listeners code would be and I did know that one listener, `ButtonListener`, would be required by three elements, not one. So again, there was little to be gained by implementing the listeners as anonymous classes. Add to that the difficulty of maintaining code that is peppered with anonymous classes and the choice of using inner classes became even more attractive.

- **The `registerListeners()` method**

  For the most part, this is simply a personal preference. I find it easier to keep track of listeners when they are all located in one spot. Having a separate method to handle them just makes life easier for me.

Summary

If you've avoided creating custom components, thinking they're to much trouble or that you need to be an expert programmer to create them, here's the proof that it just ain't so! They can be alot easier to create than you realize.

If you end up using the class in one of your applications please let me know how it fares<g>

Home | Projects

CalendarComboBox

```
/* ***************************************************************************
 *
 *       File:  CalendarWidget.java
 *    Package:  ca.janeg.calendar
 *
 *   Contains:      ButtonActionListener
 *                  CalendarModel
 *                  CalendarSelectionListener
 *                  InputListener
 *
 * References:  'Java Rules' by Douglas Dunn
 *              Addison-Wesley, 2002 (Chapter 5, section 13 - 19)
 *
 *              'Professional Java Custom UI Components'
 *              by Kenneth F. Krutsch, David S. Cargo, Virginia Howlett
 *              WROX Press, 2001 (Chapter 1-3)
 *
 * Date         Author          Changes
 * ------------ -------------   ----------------------------------------------
 * Oct 24, 2002 Jane Griscti    Created
 * Oct 27, 2002 jg              Cleaned up calendar display
 * Oct 30, 2002 jg              added ctor CalendarComboBox( Calendar )
 * Oct 31, 2002 jg              Added listeners and Popup
 * Nov  1, 2002 jg              Cleaned up InputListener code to only accept
 *                              valid dates
 * Nov  2, 2002 jg              modified getPopup() to handle display when
 *                              component is positioned at the bottom of the screen
 * Nov  3, 2002 jg              changed some instance variables to class variables
 * Mar 29, 2003 jg              added setDate() contributed by James Waldrop
 * *************************************************************************** */
package ca.janeg.calendar;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.Point;
import java.awt.Toolkit;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import java.text.DateFormat;
import java.text.DateFormatSymbols;
import java.text.ParseException;
import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;

import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.JFormattedTextField;
```

```
import javax.swing.JPanel;
import javax.swing.JTable;
import javax.swing.JTextField;
import javax.swing.ListSelectionModel;
import javax.swing.Popup;
import javax.swing.PopupFactory;
import javax.swing.SwingConstants;
import javax.swing.border.LineBorder;
import javax.swing.event.ListSelectionEvent;
import javax.swing.event.ListSelectionListener;
import javax.swing.plaf.basic.BasicArrowButton;
import javax.swing.table.DefaultTableModel;
import javax.swing.table.JTableHeader;
import javax.swing.table.TableColumn;

/**
 *  A custom component that mimics a combo box, displaying
 *  a perpetual calendar rather than a 'list'.
 *
 *  @author     Jane Griscti    jane@janeg.ca
 *  @version    1.0             Oct 24, 2002
 */
public class CalendarComboBox extends JPanel {

    // -- class fields
    private static final DateFormatSymbols  dfs         = new DateFormatSymbols();
    private static final String[]           months      = dfs.getMonths();
    private static final String[]           dayNames    = new String[ 7 ];
    private static final Toolkit            toolkit     =
Toolkit.getDefaultToolkit();
    private static final Dimension          screenSize  = toolkit.getScreenSize();
    private static final PopupFactory       factory     =
                                                PopupFactory.getSharedInstance();

    // -- instance fields used with 'combo-box' panel
    private final JPanel                inputPanel  = new JPanel();

    private final JFormattedTextField   input
                            = new JFormattedTextField( new Date() );
    private final BasicArrowButton      comboBtn
                            = new BasicArrowButton( SwingConstants.SOUTH );

    // -- instance fields used with calendar panel
    private final JPanel                calPanel    = new JPanel();
    private final JTextField            calLabel    = new JTextField( 11 );
    private final Calendar              current     = new GregorianCalendar();
    private final CalendarModel         display     = new CalendarModel( 6, 6 );
    private final JTable                table       = new JTable( display );

    private final BasicArrowButton      nextBtn =
                        new BasicArrowButton( SwingConstants.EAST );
    private final BasicArrowButton      prevBtn =
                        new BasicArrowButton( SwingConstants.WEST );
    private final BasicArrowButton      closeCalendarBtn =
                        new BasicArrowButton( SwingConstants.NORTH );
```

```
    private Popup popup;

    /**
     *  Create a new calendar combo-box object set with today's date.
     */
    public CalendarComboBox(){
        this( new GregorianCalendar() );
    }

    /**
     *  Create a new calendar combo-box object set with the given date.
     *
     *  @param cal  a calendar object
     *  @see java.util.GregorianCalendar
     */
    public CalendarComboBox( final Calendar cal ){
        super();

        // set the calendar and input box date
        Date date = cal.getTime();
        current.setTime( date );
        input.setValue( date );

        // create the GUI elements and assign listeners
        buildInputPanel();
        buildCalendarDisplay();
        registerListeners();

        // intially, only display the input panel
        add( inputPanel );
    }

    /*
     *  Creates a field and 'combo box' button above the calendar
     *  to allow user input.
     */
    private void buildInputPanel(){
        inputPanel.setLayout( new BoxLayout( inputPanel, BoxLayout.X_AXIS ) );

        input.setColumns( 12 );
        inputPanel.add( input );

        comboBtn.setActionCommand( "combo" );
        inputPanel.add( comboBtn );
    }

    /*
     *  Builds the calendar panel to be displayed in the popup
     */
    private void buildCalendarDisplay(){

        //  Allow for individual cell selection and turn off
        //  grid lines.
        table.setCellSelectionEnabled(true);
        table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

```
        table.setShowGrid( false );

        //  Calendar (table) column headers
        //    Set column headers to weekday names as given by
        //    the default Locale.
        //
        //    Need to re-map the retreived names. If used as is,
        //    the table model ends up with an extra empty column as
        //    the returned names begin at index 1, not zero.
        String[] names = dfs.getShortWeekdays();

        for( int i=1; i<names.length; i++ ){
            dayNames[ i - 1 ] = "" + names[ i ].charAt( 0 );
        }

        display.setColumnIdentifiers( dayNames );
        table.setModel( display );

        //  Set the column widths. Need to turn
        //  auto resizing off to make this work.
        table.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
        int count = table.getColumnCount();

        for( int i = 0; i < count; i ++ ){
            TableColumn col = table.getColumnModel().getColumn( i );
            col.setPreferredWidth( 20 );
        }

        //  Column headers are only displayed automatically
        //  if the table is put in a JScrollPane. Don't want
        //  to use one here, so need to add the headers
        //  manually.
        JTableHeader header = table.getTableHeader();
        header.setFont( header.getFont().deriveFont( Font.BOLD ) );

        JPanel panel = new JPanel();
        panel.setLayout( new BoxLayout( panel, BoxLayout.Y_AXIS ) );
        panel.add( header );
        panel.add( table );

        calPanel.setBorder( new LineBorder( Color.BLACK ) );
        calPanel.setLayout( new BorderLayout() );
        calPanel.add( buildCalendarNavigationPanel(), BorderLayout.NORTH );
        calPanel.add( panel );
    }

    /*
     * Creates a small panel above the month table to display the month and
     * year along with the 'prevBtn', 'nextBtn' month selection buttons
     * and a 'closeCalendarBtn'.
     */
    private JPanel buildCalendarNavigationPanel(){
        JPanel panel = new JPanel();
        panel.setLayout( new BoxLayout( panel, BoxLayout.X_AXIS ) );
```

```
        //  Add a text display of the selected month and year.
        //  A JTextField is used for the label instead of a JLabel
        //  as it is easier to ensure a consistent size; JLabel
        //  expands and contracts with the text size
        calLabel.setEditable( false );
        int fontSize = calLabel.getFont().getSize();
        calLabel.setFont( calLabel.getFont().deriveFont( Font.PLAIN, fontSize - 2 )
);
        panel.add( calLabel );

        // set button commands and add to panel
        prevBtn.setActionCommand( "prevBtn" );
        nextBtn.setActionCommand( "nextBtn" );
        closeCalendarBtn.setActionCommand( "close" );

        panel.add( prevBtn );
        panel.add( nextBtn );
        panel.add( closeCalendarBtn );

        return panel;
    }

    /*
     *  Register all required listeners with appropriate
     *  components
     */
    private void registerListeners(){

        ButtonActionListener btnListener = new ButtonActionListener();

        // 'Combo-box' listeners
        input.addKeyListener( new InputListener() );
        comboBtn.addActionListener( btnListener );

        //  Calendar (table) selection listener
        //    Must be added to both the table selection model
        //    and the column selection model; otherwise, new
        //    column selections on the same row are not recognized
        CalendarSelectionListener listener = new CalendarSelectionListener();
        table.getSelectionModel().addListSelectionListener( listener );
        table.getColumnModel().getSelectionModel()
                            .addListSelectionListener( listener );

        // Calendar navigation listeners
        prevBtn.addActionListener( btnListener );
        nextBtn.addActionListener( btnListener );
        closeCalendarBtn.addActionListener( btnListener );

    }

    /*
     *  Fill the table model with the days in the selected month.
     *  Rows in the table correspond to 'weeks', columns to 'days'.
     *
     *  Strategy:
```

```
     *      1. get the first calendar day in the new month
     *      2. find it's position in the first week of the month to
     *         determine the starting column for the day numbers
     *      3. find the actual number of days in the month
     *      4. fill the calendar with the day values, erasing any days
     *         left over from the old month
     */
    private void updateTable( Calendar cal ){

        Calendar dayOne = new GregorianCalendar(
                cal.get( Calendar.YEAR ),
                cal.get( Calendar.MONTH ),
                1 );

        // compute the number of  days in the month and
        // the start column for the first day in the first week
        int actualDays = cal.getActualMaximum( Calendar.DATE );
        int startIndex = dayOne.get( Calendar.DAY_OF_WEEK ) - 1;

        // fill the calendar for the new month
        int day = 1;
        for( int row = 0; row < 6 ; row++ ){
            for( int col = 0; col < 7; col++ ){
                if( ( col < startIndex && row == 0 ) || day > actualDays ){
                    // overwrite any left over values from old month
                    display.setValueAt( "", row, col );
                }else{
                    display.setValueAt( new Integer( day ), row, col );
                    day++;
                }
            }
        }

        // set the month, year label
        calLabel.setText( months[ cal.get( Calendar.MONTH ) ] +
                          ", " + cal.get( Calendar.YEAR ) );

        // set the calendar selection
        table.changeSelection( cal.get( Calendar.WEEK_OF_MONTH ) - 1,
                               cal.get( Calendar.DAY_OF_WEEK ) - 1,
                               false, false );
    }

    /*
     *  Gets a Popup to hold the calendar display and determines
     *  it's position on the screen.
     */
    private Popup getPopup(){
        Point p = input.getLocationOnScreen();
        Dimension inputSize = input.getPreferredSize();
        Dimension calendarSize = calPanel.getPreferredSize();

        if( ( p.y + calendarSize.height ) < screenSize.height) {
            // will fit below input panel
            popup = factory.getPopup( input, calPanel,
```

```
                                        p.x, p.y + (int)inputSize.height );
        } else {
            // need to fit it above input panel
            popup = factory.getPopup( input, calPanel,
                                       p.x, p.y - (int)calendarSize.height );
        }
        return popup;
    }

    /*
     *  Returns the currently selected date as a <code>Calendar</code> object.
     *
     *  @return Calendar    the currently selected calendar date
     */
    public Calendar getDate(){
        return current;
    }

    /**
     * Sets the current date and updates the UI to reflect the new date.
     * @param newDate the new date as a <code>Date</code> object.
     * @see Date
     * @author James Waldrop
     */
    public void setDate(Date newDate) {
        current.setTime(newDate);
        input.setValue(current.getTime());
    }

    /*
     *  Creates a custom model to back the table.
     */
    private class CalendarModel extends DefaultTableModel {

        public CalendarModel( int row, int col ){
            super( row, col );
        }

        /**
         *  Overrides the method to return an Integer class
         *  type for all columns. The numbers are automatically
         *  right-aligned by a default renderer that's supplied
         *  as part of JTable.
         */
        public Class getColumnClass( int column ){
            return Integer.class;
        }

        /**
         *  Overrides the method to disable cell editing.
         *  The default is editable.
         */
        public boolean isCellEditable( int row, int col ){
            return false;
        }
```

```
    }

    /*
     *  Captures the 'prevBtn', 'nextBtn', 'comboBtn' and
     *  'closeCalendarBtn' actions.
     *
     *  The combo button is disabled when the popup is shown
     *  and enabled when the popup is hidden. Failure to do
     *  so results in the popup screen area not being cleared
     *  correctly if the user clicks the button while the popup
     *  is being displayed.
     */
    private class ButtonActionListener implements ActionListener {
        public void actionPerformed( ActionEvent e ){
            String cmd = e.getActionCommand();

            if( cmd.equals( "prevBtn" ) ){
                current.add( Calendar.MONTH, -1 );
                input.setValue( current.getTime() );
            }else if( cmd.equals( "nextBtn" ) ){
                current.add( Calendar.MONTH, 1 );
                input.setValue( current.getTime() );
            }else if( cmd.equals( "close" ) ){
                popup.hide();
                comboBtn.setEnabled( true );
            }else{
                comboBtn.setEnabled( false );
                popup = getPopup();
                popup.show();
            }

            updateTable( current );
        }
    }

    /*
     *  Captures a user selection in the calendar display and
     *  changes the value in the 'combo box' to match the selected date.
     *
     */
    private class CalendarSelectionListener implements ListSelectionListener {

        public void valueChanged(ListSelectionEvent e){
            if ( !e.getValueIsAdjusting() ) {
                int row = table.getSelectedRow();
                int col = table.getSelectedColumn();

                Object value = null;
                try{
                    value = display.getValueAt(row, col);
                }catch( ArrayIndexOutOfBoundsException ex ){
                    // ignore, happens when the calendar is
                    // displayed for the first time
                }
```

```
                if( value instanceof Integer ){
                    int day = ( (Integer)value ).intValue();
                    current.set( Calendar.DATE, day );
                    input.setValue( current.getTime() );
                }
            }
        }
    }

    /*
     *  Captures user input in the 'combo box'
     *  If the input is a valid date and the user pressed
     *  ENTER or TAB, the calendar selection is updated
     */
    private class InputListener extends KeyAdapter {
        public void keyTyped(KeyEvent e) {

            DateFormat df = DateFormat.getDateInstance();
            Date date = null;

            try{
                date = df.parse( input.getText() );
            }catch( ParseException ex ){
                // ignore invalid dates
            }

            // change the calendar selection if the date is valid
            // and the user hit ENTER or TAB
            char c = e.getKeyChar();
            if(  date != null &&
                ( c == KeyEvent.VK_ENTER || c == KeyEvent.VK_TAB ) ) {
                current.setTime( date );
                updateTable( current );
            }
        }
    }
}
```

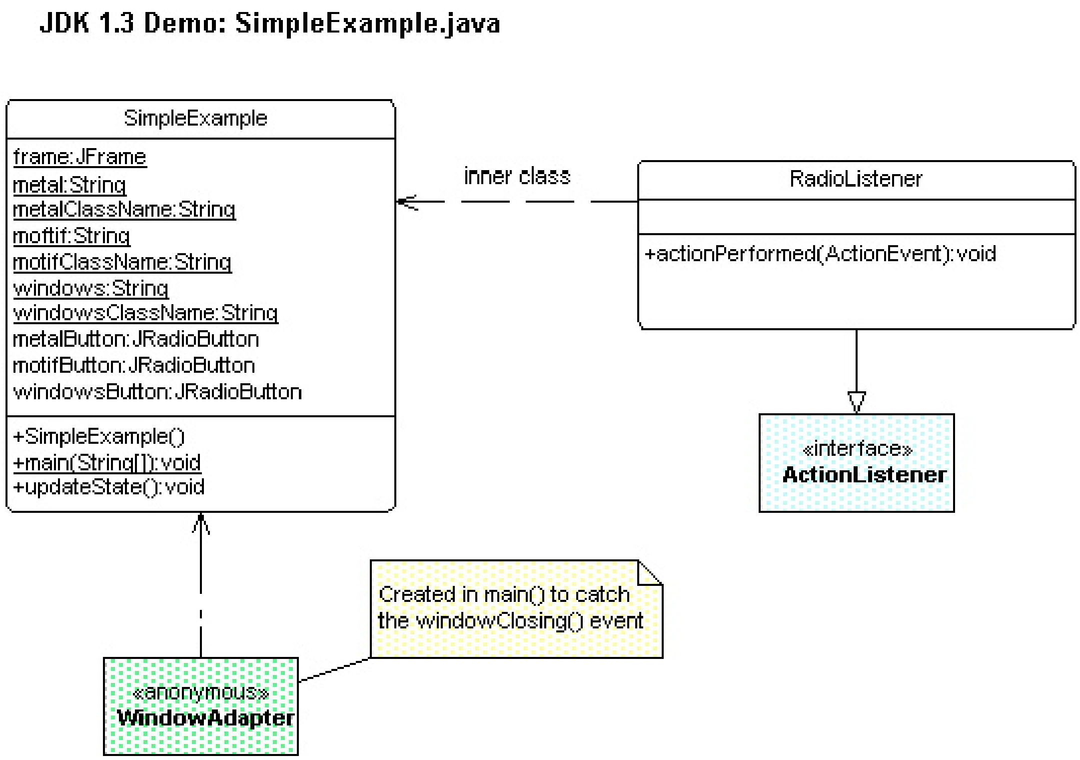
CalendarComboBox

SCJD Study Notes - GUI Design

SimpleExample Demonstrates

- changing the Look and Feel
- JRadioButtons, ButtonGroup, and mnemonics
- setting up an ActionListener as an inner class
- creating an anonymous WindowAdapter, implementing WindowClosing

UML



Mnemonics

A mnemonic allows the user to activate a button by holding ALT + the assigned mnemonic character. Setting the mnemonic for a button is relatively simple, just call the the `setMnemonic(char c)` method.

```
    Button b = new Button("Hello");
    b.setMnemonic('h');
```

That's it, no other coding required. One thing that's nice, if you're in Metal Look and Feel and you a tool tip, any assigned mnemonic is appended to the tip as 'ALT+x' where 'x' = whatever characters been assigned.

ToolTip

The demo doesn't include tool tips (the text you see when the mouse is over the component) but assigning one is easy; just invoke the `setToolTextTip(String)` method.

```
b.setToolTextTip("The Hello button");
```

This will work for every component as the method is defined in JComponent (the superclass of all Swing components).

						Resources

Home
SCJP2 Study Notes
Case Studies
SCJA Notes
SCJD Notes
Application Design
GUI Design
Database Processing
Networking
Threads
Errors and Exceptions
Security
Documentation
Projects
Favourite Links
About
Feedback

SCJD Study Notes - GUI Design

Design Theory

- Principles of good GUI Design by James Hobart
- The Design of Graphic User Interfaces on-line course.
- The Three Models Used in Designing for Ease of Use IBM Design site.
- Building user interfaces for object-oriented systems, Part 1 thru 6 JavaWorld articles by Allen Holub
- Java Look and Feel Design Guidelines from Sun

Swing

- Swing by Matthew Robinson and Pavel Vorobiev. Book which can be viewed on-line or downloaded as a Word'97 document.
- Write high-performance RMI servers and Swing clients by Andy Krumel
- Rendering cells in Swing's JTable component by Brett Spell
- Add an undo/redo function to your Java apps with Swing by Tomer Meshorer
- Using the Swing Action Architecture by Mark Davidson (Sun article)
- Using Timers in Swing Applications by Hans Muller and Kathy Walrath (Sun article)
- Threads and Swing by Hans Muller and Kathy Walrath (Sun article)
- Using Dynamic Proxies to Generate Event Listeners Dynamically by Mark Davidson (Sun article)
- Card Panel - an Alternative to Card Layout by Hans Muller (Sun article)
- Testing Java Swing-Based Applications by J. D. Newmarch, University of Canberra

Swing Resources and Articles at Sun

- Creating a GUI with JFC/Swing tutorial
- Index of Swing Articles
- Java TM Look and Feel Graphics Repository, a collection of Toolbar Icons from Sun

Resource

SCJD Study Notes - Application Design - OOD

Note
• These notes are derived from the book Object-Oriented Design in Java by Stephen Gilbert and Bill McCarty

A **class** is a programming construct; a template used to create **objects**. Try to think in terms of the object vs the class when you start a design. The design process involves building a *model* of an object using *abstraction*.

An **interface** describes the *services* the client wants accomplished ie the object's capabilities or functionality. A **public** interface describes the objects **contract with users.**
"Always start by designing a minimal public interface."

The **implementation** is how the object goes about providing the services

In Procedural programming design is based on the implementation; it is *task* oriented. Object-Oriented programming design is based on the *interface*; it is *service* oriented. You need to be concerned, initially, with *what* an object can do, **not** how it does it.

Encapsulation hides the non-essentials ie it hides the implementation details. This is not about setting every field to *private* and writing public *gettors* and *settors*. You need to make sure your *public interface* **does not** rely on how the objects behaviour is implemented. Think what would happen if every time you upgraded your PC you had to learn a new keyboard layout! Sales would plummet and programmers would become extinct.

When you begin to design an object, you need to act like an investigative reporter and discover the:

- **WHO**
- **WHERE**, and
- **WHAT**

of an object's existance.

- **Who** is going to use the object? What *clients*(actors) are going to use the object you're designing
- **Where** is your object going to exist? What hardware and software is involved? Will it exist in a framework ie inside other objects? What operating system will it run on?
- **What** functions should it have from the user's point of view? What services can it be reasonably expected to provide?

As a first step, describe, in a single paragraph, exactly what the object you're building should do (requirements). This paragraph is informal and written from a user's perspective ie "I want an object that can display the current date and the time in an analog or digital format." **not** "This object uses the Java Date class and JPanel to display the date and time. The analog display blah, blah, blah ...."

State and Behaviour

State

An objects *attributes* define its state (condition). The attributes can be defined as:

1. Instance fields. An instance is one object created from a class. The instance attributes are unique to each object. For example, a Name class might have two attributes: firstName and lastName. Every object created from the Name class would have a different value for each attribute.

2. Class fields. A state that holds true for every object in the class. For example, an Employee class may include an id attribute that holds the last id number and is incremented every time a new Employee is created. The value in the id field would be common to all Employee objects.
3. Class constants. Pre-defined conditions that can be applied to all objects in the class. For example, a class that defines buffer objects may have a MAX_BUFFER value.

Behaviour

Design Traps

It might be easier to describe well-designed code in terms of what **it is not** vs what **it is**. The following is a summary of such information gleaned from various sources:

Source: Object-Oriented Design in Java by Stephen Gilbert and Bill McCarty

Data Warehouse Trap

An object is **not** a repository for data that the rest of your program will use! An object should manipulate it's own data; not pass it to other parts of the program which then manipulate it.

Spectral Object Trap

An object is **not** a collection of methods you pass data to. Objects with no data are ghosts.

Multiple Personality Trap

An object should model only one object. Every data element and every method should contribute to that object.

SCJD Study Notes - Application Design - OOP

Must Read
• If you read only one book before you start your SCJD assignment make it Effective Java by Joshua Bloch. This is an excellent book that will give you new insights into how the Java language is best utilized. It contains 57 items grouped into categories: Creating and Destroying Objects, Classes and Interfaces, General Programming, Threads, etc. that describe the programming idioms that work best along with the how and why of implementing them.

Other sources worth investigating:

- JavaIdioms
- The Essence of Object-Oriented Programming with Java and UML
- The Pragmatic Programmer
- Design Techniques Articles about Java program design by Bill Venners

- Home
- SCJP2 Study Notes
- Case Studies
- SCJA Notes
- SCJD Notes
- Application Design
- GUI Design
- Database Processing
- Networking
- Threads
- Errors and Exceptions
- Security
- Documentation
- Projects
- Favourite Links
- About
- Feedback

SCJD Study Notes - Application Design

Modeling Tools	
Use Case	A semi-formal description of what a user wants from a system and how they expect to interact with the system to bring about a specific result. Generally people; however, a user can also be another system or another piece of the same system. Sometimes referred to as *scenarios*. • Structuring Use Cases with Goals by Alistair Cockburn • Use and Abuse Cases(PDF) by Martin Fowler • Modeling Essential Use Cases by Scott W. Ambler • Roles before Objects by Doug Lea • Dealing with Roles(PDF) by Martin Fowler
CRC	**Class-Responsibility-Collaboration** cards. Martin Fowler calls it *"One of the most valuable techniques for learning OO" (UML Distilled p9)* • A Laboratory For Teaching Object-Oriented Thinking by Ward Cunningham and Kent Beck, the developers of CRC.
Interaction Diagrams	Two flavours: sequence and collaboration. Useful when trying to capture the behaviour of several objects within a single use case. Martin Fowler recommends using State diagrams to model the behaviour of one object across multiple use cases. *(UML Distilled p78)* • Introduction to UML sequence diagrams by Scott W. Ambler
Class Diagrams	Classes describe objects in the domain and the static relationships that exist between them. Detail the class data (attributes) and operations (behaviour). • A general discussion of Class Diagrams by Martin Fowler. Includes tips on when and how they are best utilized. • UML Tutorial - Class Diagrams(PDF) by Robert C. Martin • Class Diagrams in Analysis an exercise in developing Class Diagrams from a Use Case accompanied by lecture notes (PDF) which explain the analysis process.
Design Patterns	Patterns are example models of processes that crop up repeatedly in software development. For example, developers are often faced with problems that require moving through a list or collection. The *Iterator* pattern describes a standard technique for handling iterations. • The Design Patterns Java Companion by James Cooper • Implementing Basic Design Patterns in Java by Doug Lea • Speaking on the Observer pattern How can you use the Observer pattern in your Java design? (JavaWorld)

On-line Analysis and Design Tutorials/Lectures

- Interactive Web Tutorial for OOP by Deniz Zubair choudhury

- The Essence of Object-Oriented Programming with Java and UML by Bruce E. Wampler (draft of book)
- Techniques for Object Oriented Analysis and Design by Martin Fowler
- Systems Analysis and Design A series of lectures and practical exercises based on the book *Object-Oriented Systems Analysis and Design using UML* by Simon Bennet, Steve McRobb, Ray Farmer
- Object-Oriented Analysis and Design lecture series by J.W. Schmidt, Claudia Niederée, and Michael Skusa
- A Commercially Robust Process for the Development of OO Software Systems(PDF)

Free Modeling Software

- DOME free modeling software from Honeywell.
- mUml from *MountField Computers* free for non-commercial use. Written entirely in Java using Swing GUI. Capabilities allow you to draw all 9 UML diagrams in colour. Diagrams can be saved as JPEGs or saved as HTML pages.
- If you have Visio v4, v5 or Visio 2000 you can download a free Visio Stencil and Template for UML courtesy of Navision and Paul Hruby.
- ArgoUML free case tool; part of the Tigris.org open-source platform.
- If you're using **Linux** or **Sun Solaris**, you can download a free copy of JVision for non-commercial use. (Sorry, if you're using Windows it will cost you.)

Miscellaneous

- UML Reference Card Allen Holub has put together a great page with annotated UML diagrams.
- UML Dictionary put together by Kendall Scott, author of *The Unified Modeling Language User Guide* and four UML/OOP related books.

Java Case Study - Mail Merge

Source

The code for this study is from Developing Java Software, 2nd Edition by Russel Winder and Graham Roberts and may be downloaded from the authors support site.

Home | Case Studies

Java Case Study - Mail Merge - Overview

Problem Statement

Implement an application, in Java, that will merge an address file with a letter file. The letter file is a LaTex document. The Java application will invoke LaTex via the operating system. The Latex application will process and print each newly created document.

Address File Structure

The address file will contain element groups tagged as follows:

```
<NAME>
<TELEPHONE>
<FAX>
<EMAIL>
<ADDRESS>
```

The address element must be the last in the group. Street, city and country information must be separated by commas. If the same element appears more than once within a group, the value of the last element is used.

Sample LaTex File

```
\documentclass{rlw_letter}

\begin{document}
\begin{letter}{<NAME>\\
<ADDRESS>}
\opening{Dear <NAME>,}

This is just some text to show where the text of the letter would be.

\closing{Yours sincerely,}
\end{letter}
\end{document}
```

Home | Case Studies | TOC | Next

LaTex

LaTex is a typesetting system used in the production of technical and scientific documentation.

For more information see The LaTex Home Page

Java Case Study - Mail Merge - User Defined Types

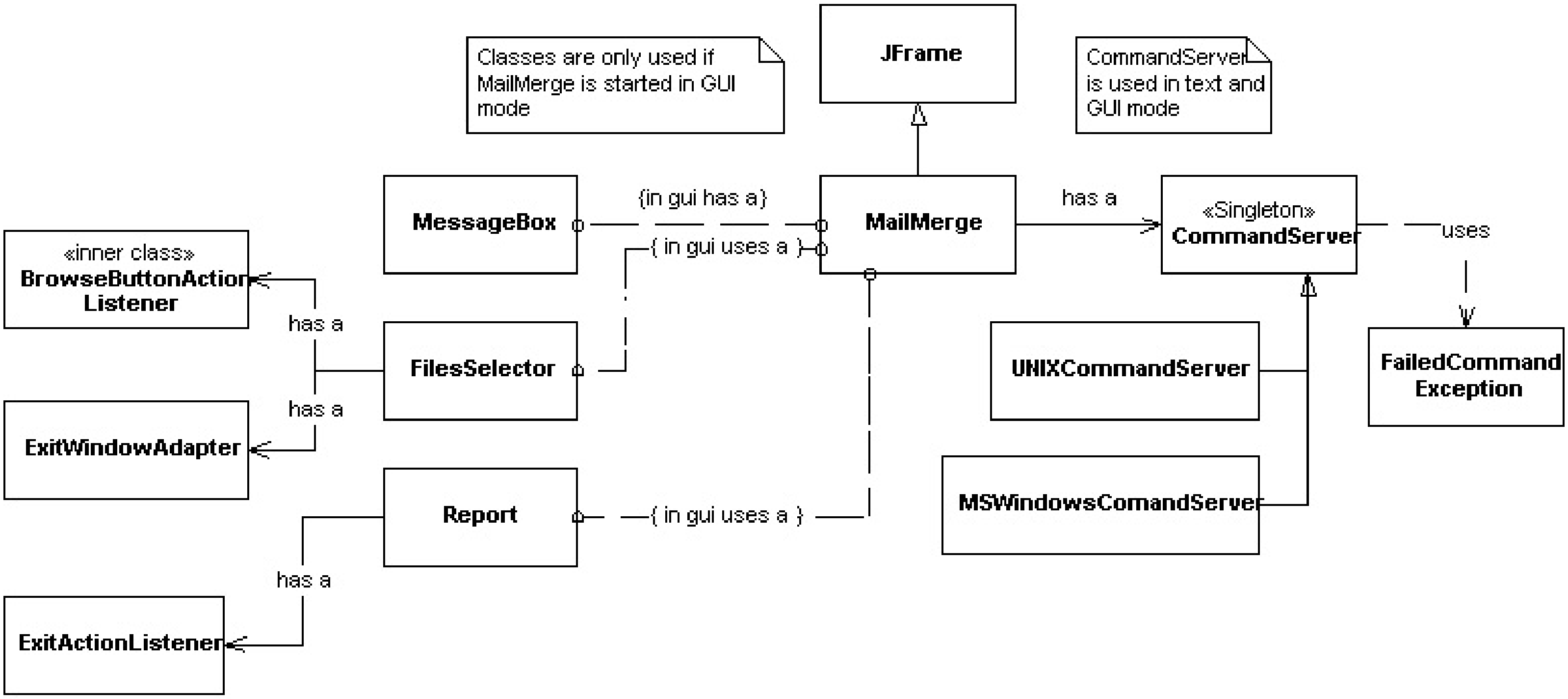
User Defined Types

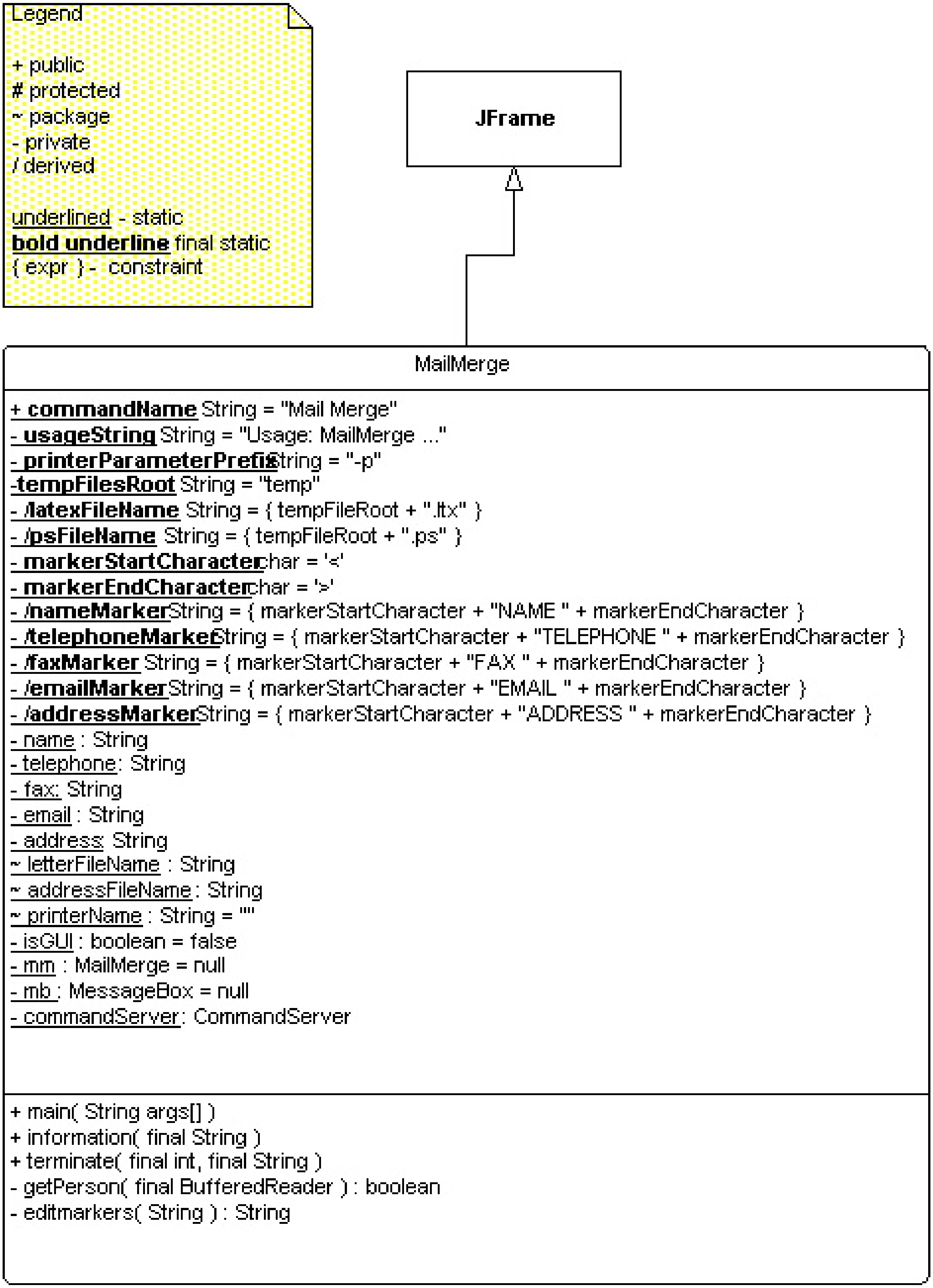
Full UML Class Diagram

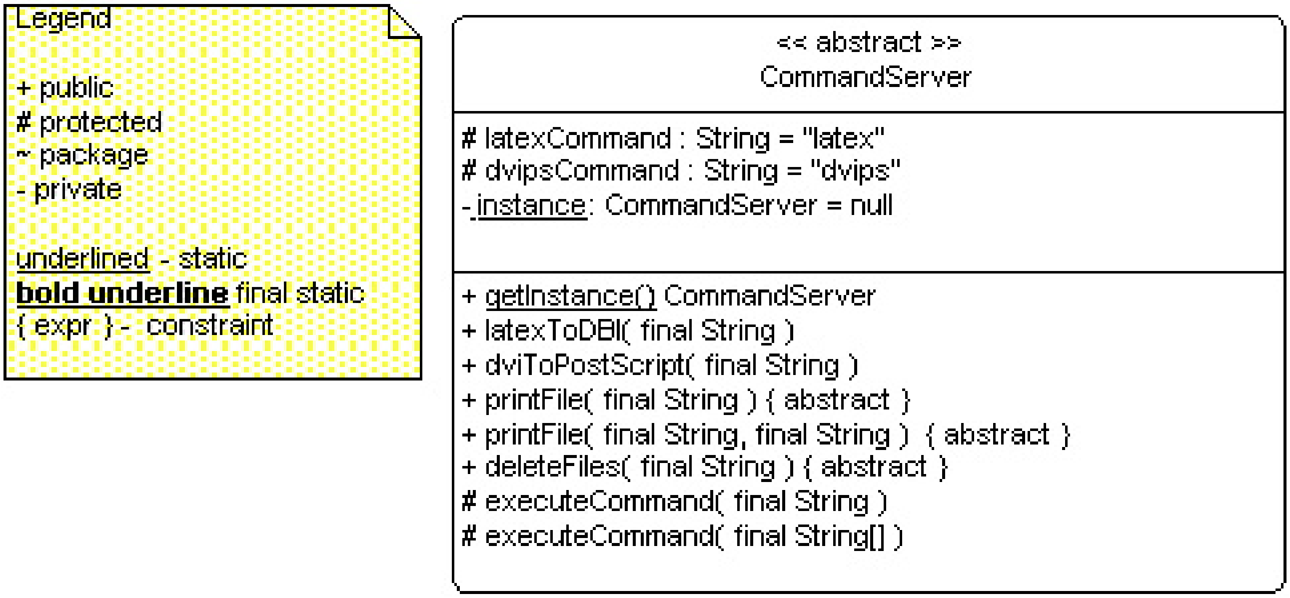
The application is implemented with the following user defined types:

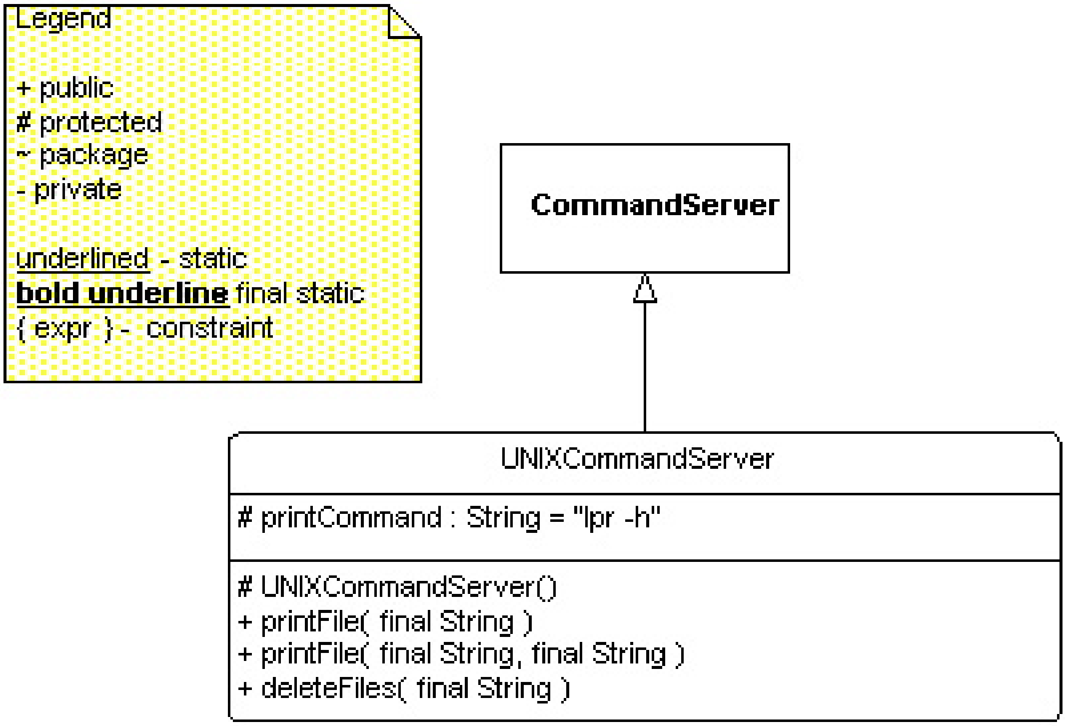
- MailMerge
- CommandServer
  - UNIXCommandServer
  - MSWindowsCommandServer
- FailedCommandException
- MessageBox
- FilesSelector
  - FilesSelector$BrowseButtonActionListener
- Report
- ExitActionListener
- ExitWindowAdapter

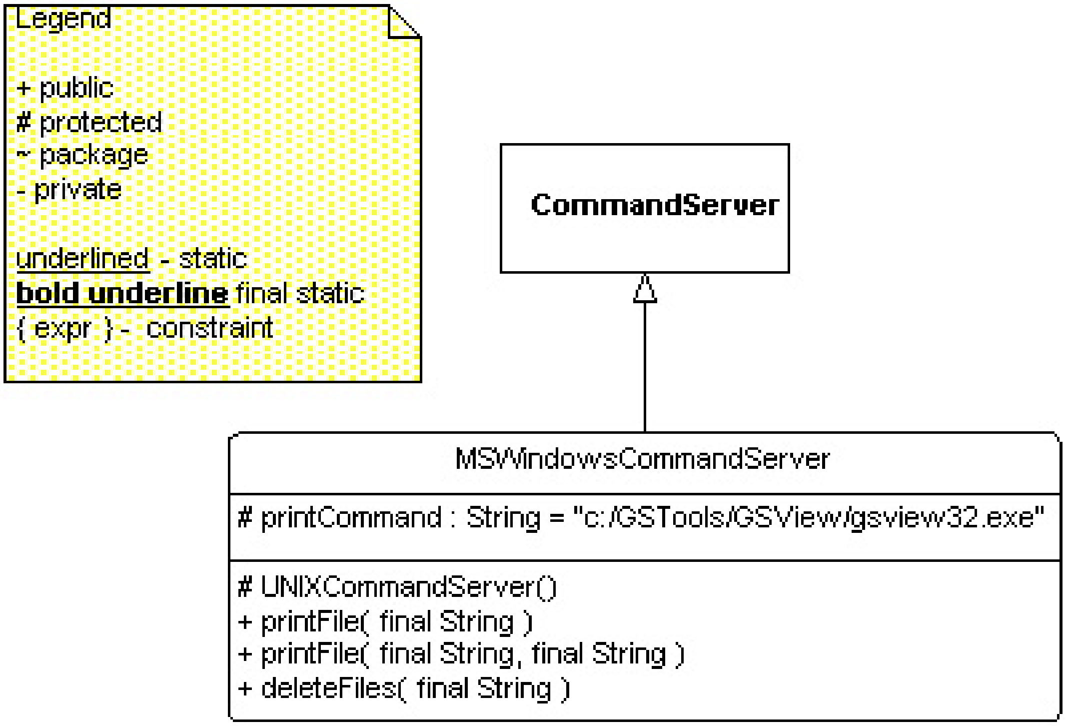
Home | Case Studies | Previous | TOC | Next

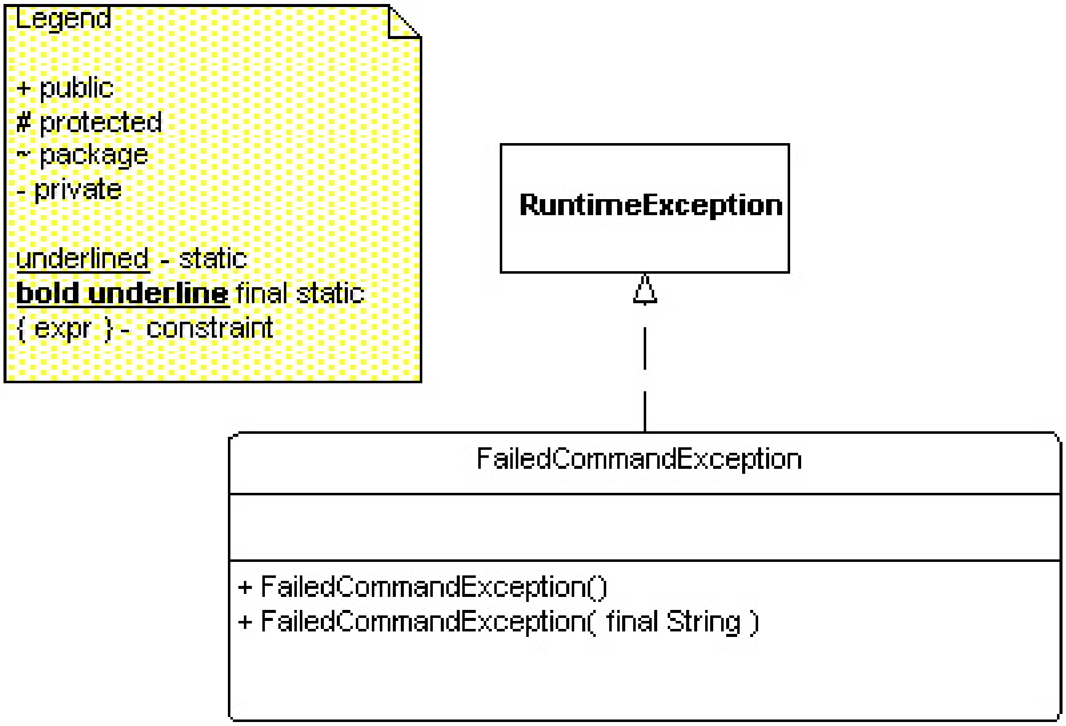

Classes are only used if MailMerge is started in GUI mode
JFrame
CommandServer is used in text and GUI mode
MessageBox
{in gui has a}
MailMerge
has a
«Singleton»
CommandServer
uses
«inner class»
BrowseButtonAction Listener
{ in gui uses a }
has a
FilesSelector
UNIXCommandServer
FailedCommand Exception
has a
ExitWindowAdapter
MSWindowsComandServer
Report
{ in gui uses a }
has a
ExitActionListener

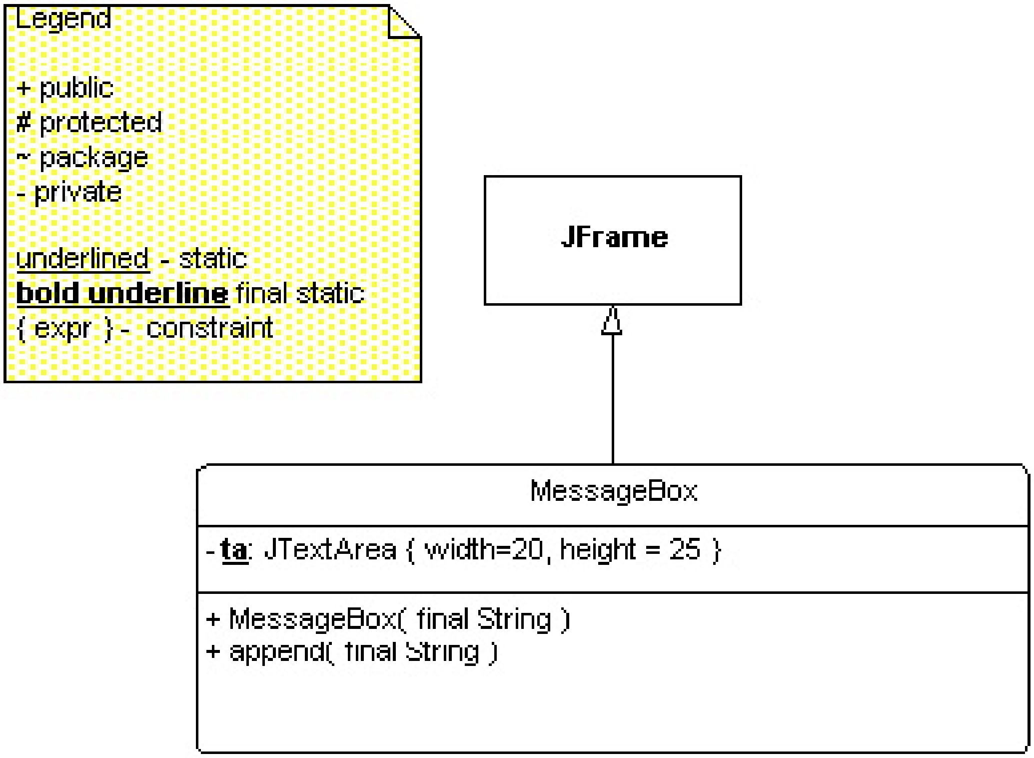

Legend
+ public
# protected
~ package
- private
/ derived
underlined - static
bold underline final static
{ expr } - constraint
JFrame
MailMerge
+ commandName String = "Mail Merge"
- usageString String = "Usage: MailMerge ..."
- printerParameterPrefix String = "-p"
-tempFilesRoot String = "temp"
- /latexFileName String = { tempFileRoot + ".ltx" }
- /psFileName String = { tempFileRoot + ".ps" }
- markerStartCharacter char = '<'
- markerEndCharacter char = '>'
- /nameMarker String = { markerStartCharacter + "NAME " + markerEndCharacter }
- /telephoneMarker String = { markerStartCharacter + "TELEPHONE " + markerEndCharacter }
- /faxMarker String = { markerStartCharacter + "FAX " + markerEndCharacter }
- /emailMarker String = { markerStartCharacter + "EMAIL " + markerEndCharacter }
- /addressMarker String = { markerStartCharacter + "ADDRESS " + markerEndCharacter }
- name : String
- telephone: String
- fax: String
- email : String
- address: String
~ letterFileName : String
~ addressFileName : String
~ printerName : String = ""
- isGUI : boolean = false
- mm : MailMerge = null
- mb : MessageBox = null
- commandServer: CommandServer
+ main(String args[])
+ information(final String)
+ terminate(final int, final String)
- getPerson(final BufferedReader) : boolean
- editmarkers(String) : String

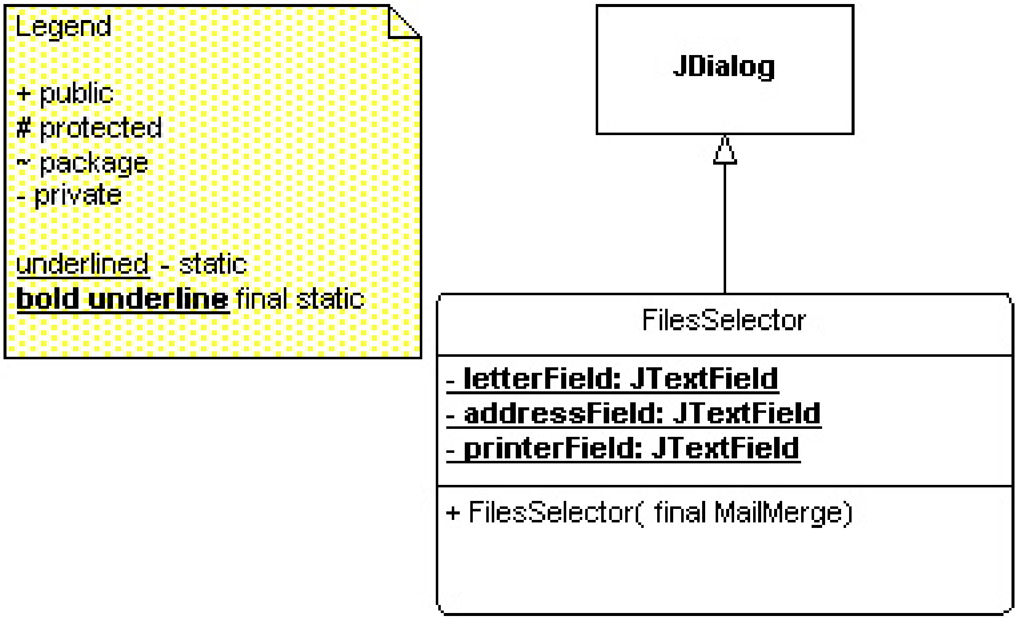

Legend
+ public
# protected
~ package
- private
underlined - static
bold underline final static
{ expr } - constraint
<< abstract >>
CommandServer
latexCommand : String = "latex"
dvipsCommand : String = "dvips"
- instance: CommandServer = null
+ getInstance() CommandServer
+ latexToDBI(final String)
+ dviToPostScript(final String)
+ printFile(final String) { abstract }
+ printFile(final String, final String) { abstract }
+ deleteFiles(final String) { abstract }
# executeCommand(final String)
# executeCommand(final String[])

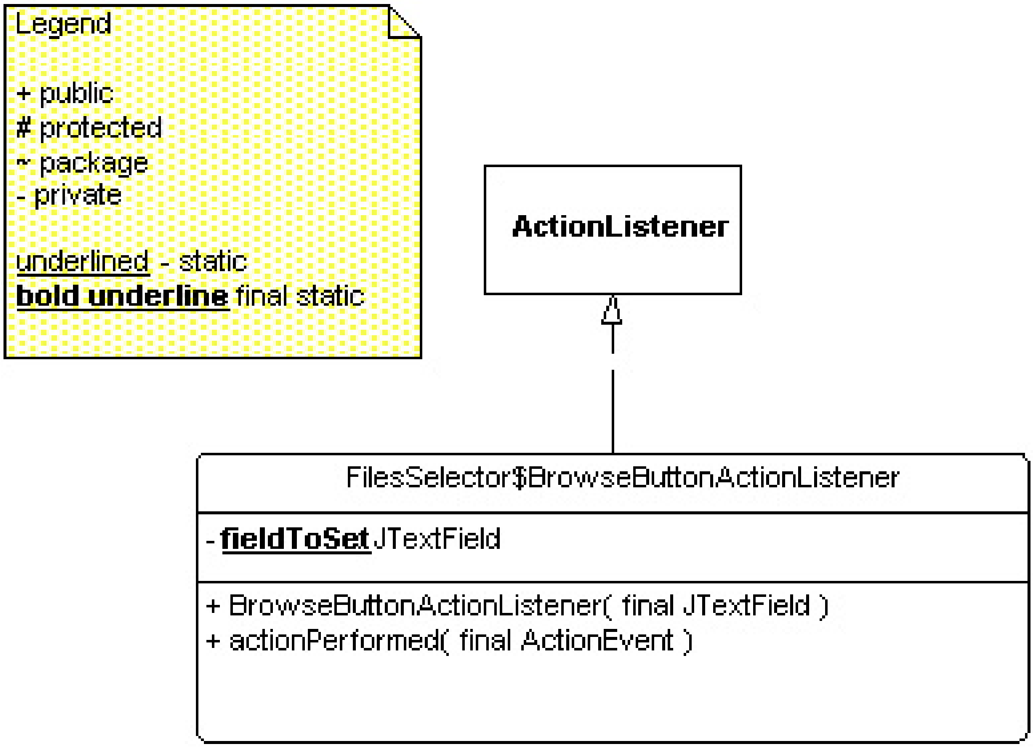

Legend
+ public
# protected
~ package
- private
underlined - static
bold underline final static
{ expr } - constraint
CommandServer
UNIXCommandServer
printCommand : String = "lpr -h"
UNIXCommandServer()
+ printFile(final String)
+ printFile(final String, final String)
+ deleteFiles(final String)

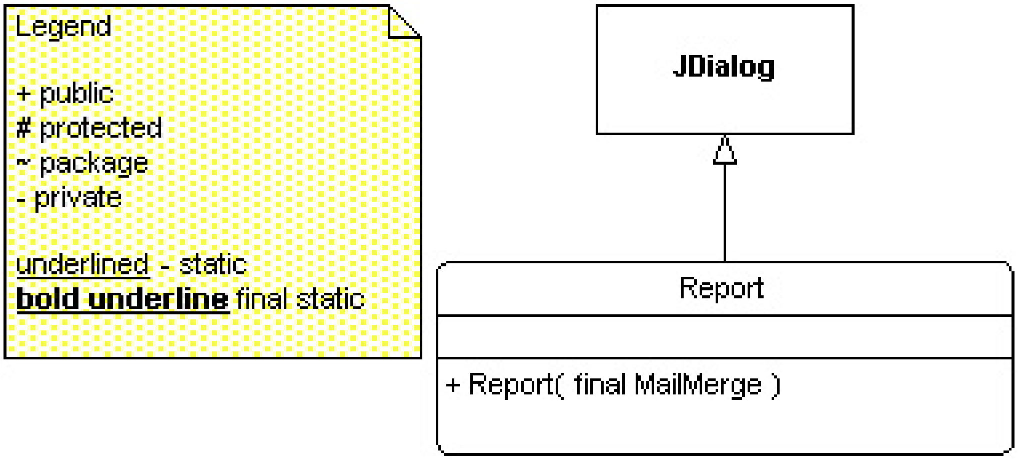

Legend
+ public
# protected
~ package
- private
underlined - static
bold underline final static
{ expr } - constraint
CommandServer
MSWindowsCommandServer
printCommand : String = "c:/GSTools/GSView/gsview32.exe"
UNIXCommandServer()
+ printFile(final String)
+ printFile(final String, final String)
+ deleteFiles(final String)

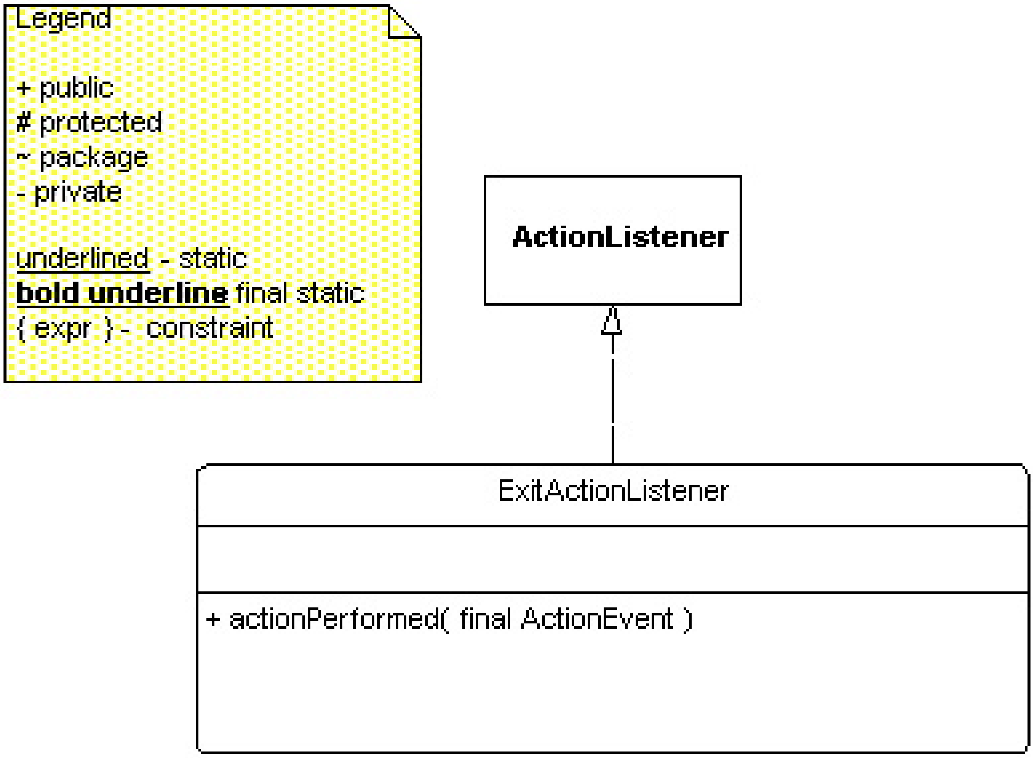

Legend
+ public
# protected
~ package
- private
underlined - static
bold underline final static
{ expr } - constraint
RuntimeException
FailedCommandException
+ FailedCommandException()
+ FailedCommandException(final String)

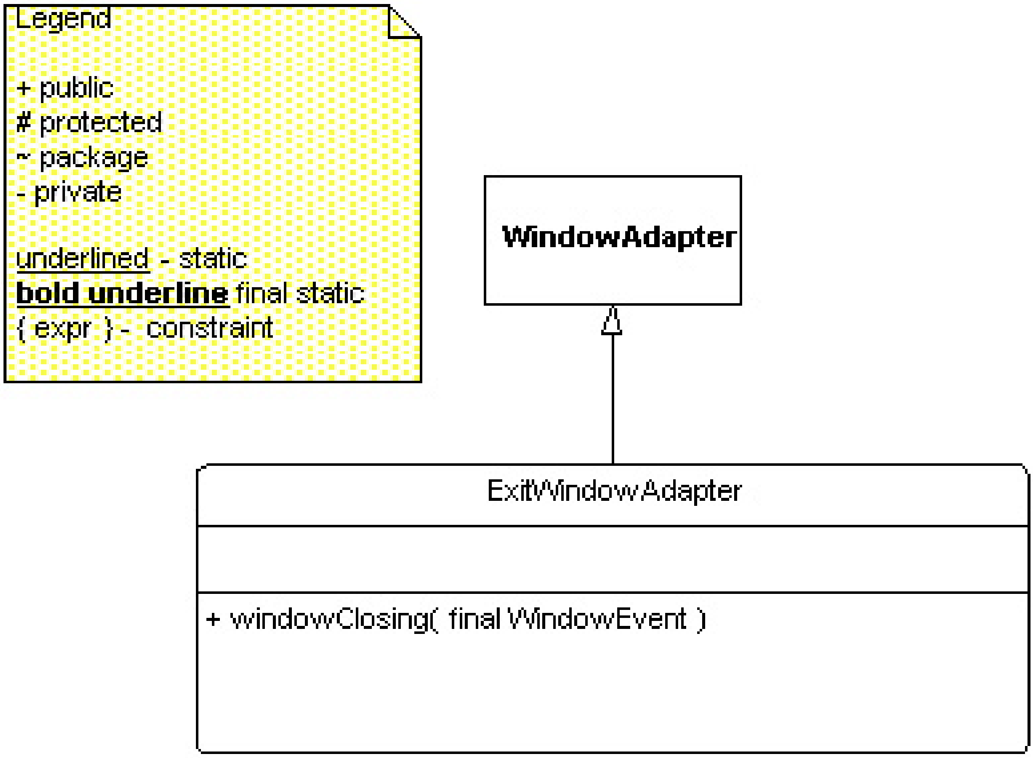

Legend
+ public
# protected
~ package
- private
underlined - static
bold underline final static
{ expr } - constraint
JFrame
MessageBox
- ta: JTextArea { width=20, height = 25 }
+ MessageBox(final String)
+ append(final String)


Legend
+ public
# protected
~ package
- private
underlined - static
bold underline final static
JDialog
FilesSelector
- letterField: JTextField
- addressField: JTextField
- printerField: JTextField
+ FilesSelector(final MailMerge)


Legend
+ public
# protected
~ package
- private
underlined - static
bold underline final static
ActionListener
FilesSelector$BrowseButtonActionListener
- fieldToSet JTextField
+ BrowseButtonActionListener(final JTextField)
+ actionPerformed(final ActionEvent)


Legend
+ public
# protected
~ package
- private
underlined - static
bold underline final static
JDialog
Report
+ Report(final MailMerge)


Legend
+ public
# protected
~ package
- private
underlined - static
bold underline final static
{ expr } - constraint
ActionListener
ExitActionListener
+ actionPerformed(final ActionEvent)


Legend
+ public
# protected
~ package
- private
underlined - static
bold underline final static
{ expr } - constraint
WindowAdapter
ExitWindowAdapter
+ windowClosing(final WindowEvent)

Java Case Study - Mail Merge - Quasi Pseudo-Code

When MailMerge is started it ...

```
    gets an instance of CommandServer based on the operating system
    if( args == 0 ) {
        creates another instance of MailMerge
        creates a FileSelector, passing it the new MailMerge instance
            the FileSelector captures user input: letterFileName, addressFileName, 
printerName
            and updates the fields in the MailMerge instance
            destroys itself when the user dismisses it
        creates a Report to display values input by user
            destroys itself when the user dismisses it
    } else {
        retrieves the file and printer names from the command line arguments
    }
    opens the files
    reads the letter file into memory
    for( each record in the address file ) {
        reads a record
        displays the values
        creates a temporary file
        merges the record with the letterfile
        writes the merged result to the temporary file
        sends commands to the operating system via the CommandServer to
            create a DVI file
            convert the DVI file to a PostScript file
            spool the postscript file to the printer
            delete the temporary files
    }
    closes the address file
    exits
```

Java Case Study - Mail Merge - Notes on Design

Text and GUI Modes

The application was designed to be run in text mode. A GUI interface was added later. This involved having `MailMerge` extend `JFrame`. The main window, however, is never displayed. The original class spawns another instance of `MailMerge` which acts as the parent of the GUI elements.

If `MailMerge` is started with no command line parameters a FileSelector dialog is displayed. The user enters the file names or clicks a **browse** button which displays a JFileChooser dialog. When finished, he clicks **ok**. At that point the input is saved to the `MailMerge` instance originally passed to `FileSelector`. Because `MailMerge` has only **static** fields, updating an instance of `MailMerge` effectively updates the original `MailMerge` (remember, only one copy of a static field exists for all instances of the class).

Java Libraries

Standard Java library classes were used for file handling:

```
    java.io.BufferedReader          java.io.BufferedWriter
    java.io.File                    java.io.FileReader          java.io.FileWriter
    java.io.FileNotFoundException   java.io.IOException
```

All of these are listed using **import-by-type** versus **import-on-demand** statements. (See the import statements in the MailMerge source code).

The standard classes `String` and `StringBuffer` were used for string manipulation.

Standard Swing classes were extended to create all the GUI elements.

```
javax.swing.JFrame          javax.swing.JDialog         javax.swing.JOptionPane
javax.swing.JButton         javax.swing.JFileChooser    javax.swing.JPanel
javax.swing.JTextField      javax.swing.JLabel
```

Event Listeners

When a listener is required for an event specific to the class it is implemented as an anonymous class.

For example, the listener attached to the **okButton** in **FileSelector** is declared as an anonymous class implementing the **ActionListener** interface (see the source code for FileSelector)

When a listener is required for an event specific to the class but can be used by more than one component belonging to the class, it is implemented as an inner class.

For example, a **FileSelector** dialog has two **browse** buttons both of which, when clicked, result in a **JFileChooser** dialog being displayed. The **BrowseButtonActionListener** class is declared within the **FileSelector** class. It implements the **ActionListener** interface and provides a constructor that takes a **JTextField**. The value of the parameter is saved so that each new instance of the listener knows which field it must set.

When a listener is required for a class but its functionality is not specific to the class (it has a behaviour that could apply in other situations) it is implemented as a separate class.

For example, the ExitWindowAdapter, which simply calls **System.exit(0)**, is implemented as a separate class; allowing it to be re-used by other classes.

Event listeners are named according to the interface they implement or the adapter they extend and the component they will be registered with.

For example, rather than name the listener responsible for closing a window as **ExitWindow** it is named **ExitWindowAdapter**. From the name it is evident that the class will cause a **window** componet to be exited and that the class extends the **WindowAdapter** class versus implementing **WindowListener** interface.

Passing parameters

All method parameters (except those in MailMerge.editMarkers()) are passed as **final**. It is considered good practice to pass parameters as *final* if the method will not modify the value in any way. The use of *final* signals this intent. Also, it allows the compiler to optimize the code for better performance.

Passing commands to the operating system

CommandServer is implemented using the **Singleton** pattern. It has a **private** constructor. The only way to instantiate the class is by calling the **public static getInstance()** method. The first call to the method creates an instance of the class and assigns it to a private static field, **instance**. Subsequent calls to **getInstance()** will return the *same instance*.

An instance specific to the operating system is required as:

1. there is no wildcard expansion unless the operating systems command shell is explicitly started
2. operating systems have different command syntaxes
3. the Java method used to pass commands to the operating system does **not** start a command shell

Home | Case Studies | Previous | TOC | Next

Java Case Study - Mail Merge - Using an Abstract class

CommandServer was implemented as an **abstract** class. Why?

In this instance, the bulk of the code is identical across operating systems. The subclasses **UNIXCommandServer** and **MSWindowsCommandServer** are *specializations*. Had the type been defined as an interface a good portion of the code would need to be repeated in each implementation class.

An alternative would have been to define **CommandServer** as an **interface** and provide a separate **CommandServerImpl** class that defined the common code. This skeletal implemention could then be extended by subclasses.

Not sure the alternative would buy anything in this example. Especially if you decided to add additional functionality. Right now the class has two methods `printFile()` and `deleteFiles()`. Operating systems offer a vast array of commands and it's highly likely that one day you'll want to add more methods to handle them. If CommandServer was defined as an *interface* adding methods would break existing code; all types based on the interface would need to add implementation for the new methods.

Java Case Study - Mail Merge - Extending RuntimeException

Generally you hear that you should extend **Exception** versus **RuntimeException** when you define your own exceptions. RuntimeExceptions are used for exceptions that an application cannot reasonably be expected to handle.

The **FailedCommandException** is thrown when a CommandServer object cannot execute LaTex, print the file or delete temporary files. The circumstances surrounding the events are a result of the operating system setup. LaTex may not be installed, GhostScript may not be installed, the user may not have delete authority for the drive he's accessing.

The application cannot be reasonably expected to handle these situations therefore the choice of extending **RuntimeException** versus **Exception** is justified.

Home | Case Studies | Previous | TOC | Next

Java Case Study - Mail Merge - The GUI implementation

The handling of the GUI is rather awkward. The initial MailMerge window is never displayed and a second MailMerge instance is created and passed to GUI components. This method also hides the manner in which the required fields in the original MailMerge are updated; it's not intuitive.

One possible alternative would be to create a seperate object to hold the data input fields, add a field of that data type, add a JFrame field and add a second constructor that creates the JFrame and takes the new object as a parameter. The JFrame would be used as the parent of any GUI components and, if the application is started in text mode, no GUI elements would be created.

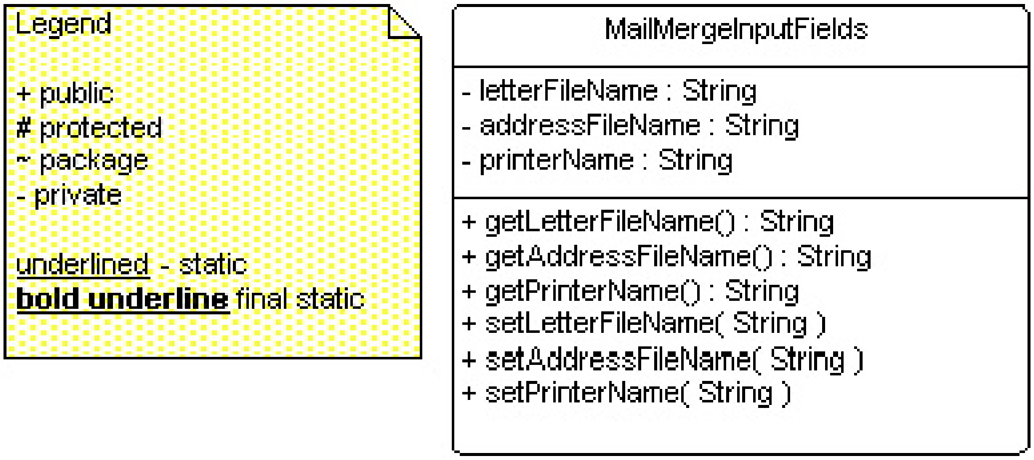
Trying to refactor a program helps you test your understanding of how it works. Tried the above as a refactoring exercise, revising the code as follows:

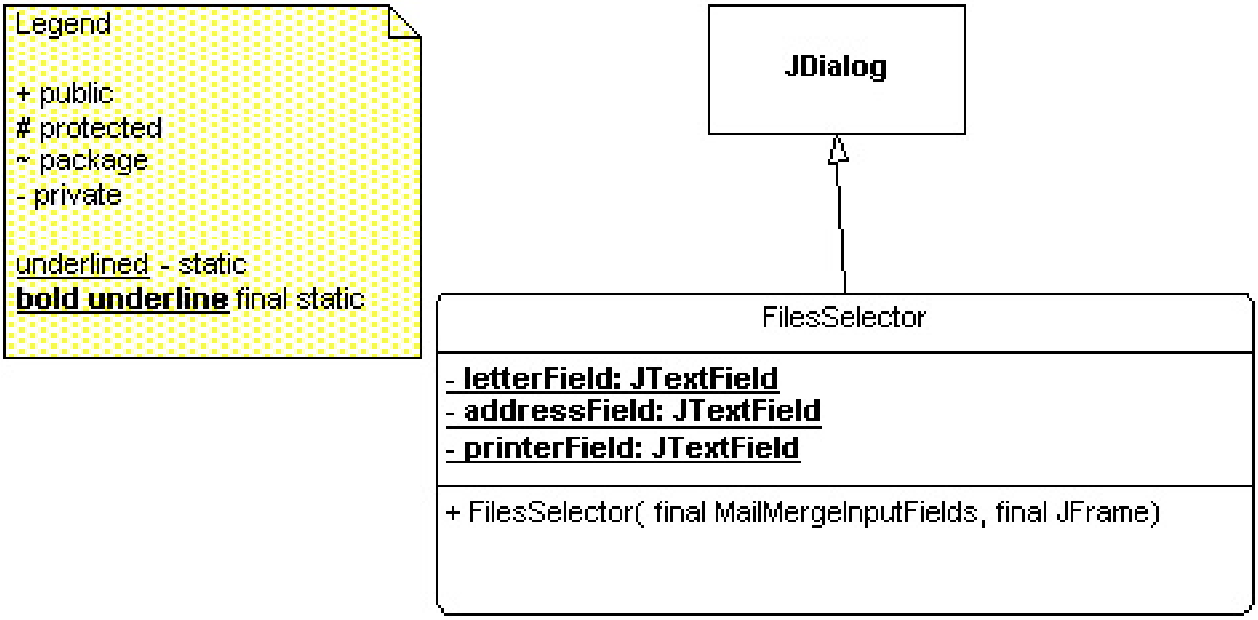
1. Created a new class, MailMergeInputFields
2. Modified the FileSelector and Report classes to take a MailMergeInputFields object and a **JFrame** object instead of a MailMerge object. Changed all references in the classes to use the new object.
3. Modified the MailMerge class:
    - removed the inheritance to JFrame
    - replaced the indivual fields **letterFileName, addressFileName** and **printerName** with a **MailMergeInputFields** object
    - removed the MailMerge instance and added a **JFrame** reference.
    - changed all references in the MailMerge class to use the new MailMergeInputFields object and JFrame reference where necesssary.

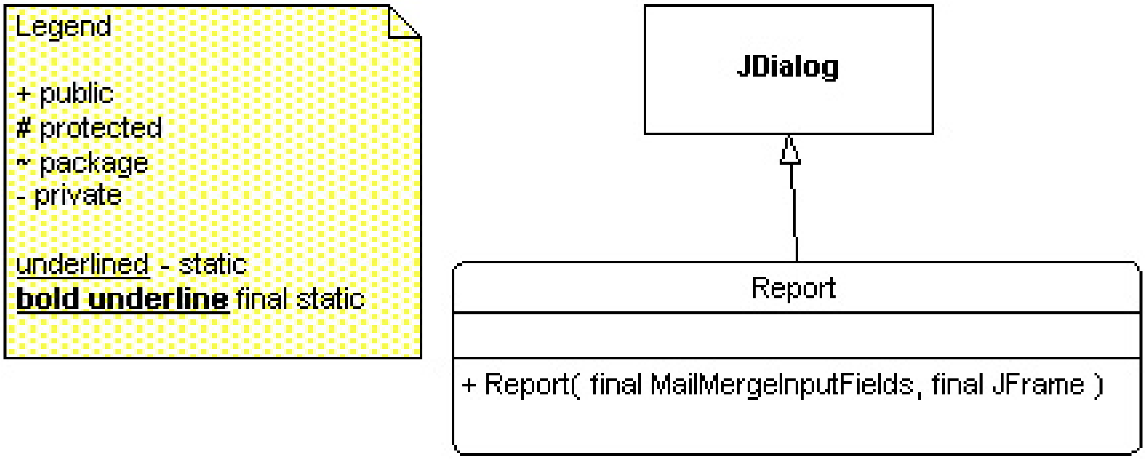
Made a few other minor changes: split the code in `main()` into two separate methods, `setup()` and `processFiles()` and modified the `terminate()` method so it could be used as a single exit point from the application. The revised class files are:

- MailMergeInputFields
- MailMerge Note: the commands directed to LaTex have been commented out as it is not installed on my system.
- FilesSelector
- Report
- Revised UML Diagram

Home | Case Studies | Previous | TOC | Next

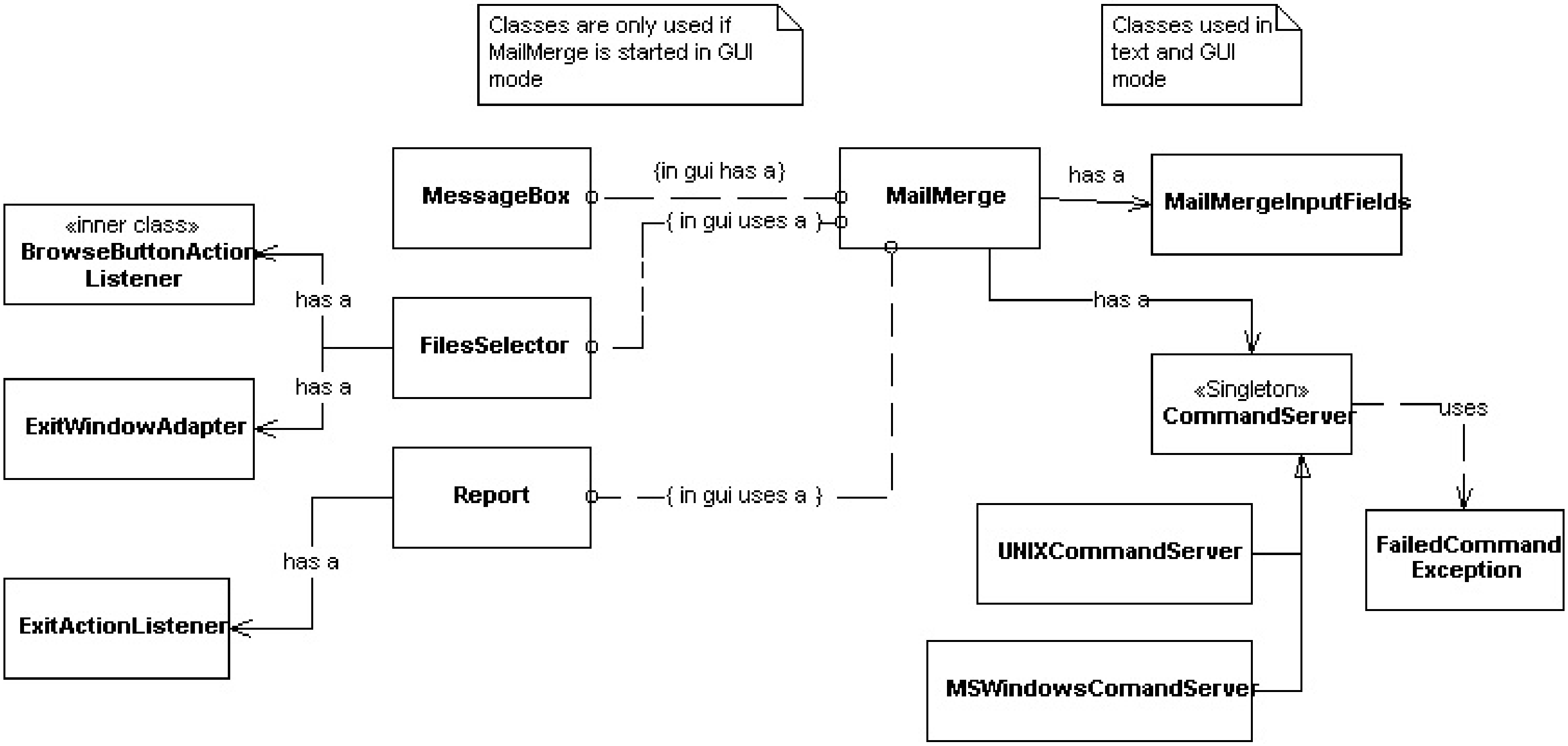

Legend
+ public
# protected
~ package
- private
underlined - static
bold underline final static
MailMergeInputFields
- letterFileName : String
- addressFileName : String
- printerName : String
+ getLetterFileName() : String
+ getAddressFileName() : String
+ getPrinterName() : String
+ setLetterFileName(String)
+ setAddressFileName(String)
+ setPrinterName(String)


Legend
+ public
# protected
~ package
- private
underlined - static
bold underline final static
JDialog
FilesSelector
- letterField: JTextField
- addressField: JTextField
- printerField: JTextField
+ FilesSelector(final MailMergeInputFields, final JFrame)



Legend
+ public
# protected
~ package
- private
underlined - static
bold underline final static
JDialog
Report
+ Report(final MailMergeInputFields, final JFrame)


Legend
+ public
# protected
~ package
- private
/ derived
underlined - static
bold underline final static
{ expr } - constraint
MailMerge
+ commandName String = "Mail Merge"
- usageString String = "Usage: MailMerge ..."
- printerParameterPrefix String = "-p"
-tempFilesRoot String = "temp"
- /latexFileName String = { tempFileRoot + ".ltx" }
- /psFileName String = { tempFileRoot + ".ps" }
- markerStartCharacter char = '<'
- markerEndCharacter char = '>'
- /nameMarker String = { markerStartCharacter + "NAME " + markerEndCharacter }
- /telephoneMarker String = { markerStartCharacter + "TELEPHONE " + markerEndCharacter }
- /faxMarker String = { markerStartCharacter + "FAX " + markerEndCharacter }
- /emailMarker String = { markerStartCharacter + "EMAIL " + markerEndCharacter }
- /addressMarker String = { markerStartCharacter + "ADDRESS " + markerEndCharacter }
- name : String
- telephone: String
- fax: String
- email : String
- address: String
- inputFields: MailMergeInputFields
- isGUI : boolean = false
- frame : JFrame
- mb : MessageBox = null
- commandServer: CommandServer
+ main(String args[])
+ setup(String[])
+ processFiles()
+ information(final String)
+ terminate(final int, final String)
- getPerson(final BufferedReader) : boolean
- editmarkers(String) : String


Classes are only used if MailMerge is started in GUI mode
Classes used in text and GUI mode
MessageBox
{in gui has a}
MailMerge
has a
MailMergeInputFields
«inner class»
BrowseButtonAction Listener
{ in gui uses a }
has a
FilesSelector
has a
«Singleton»
CommandServer
uses
ExitWindowAdapter
Report
{ in gui uses a }
UNIXCommandServer
FailedCommand Exception
has a
ExitActionListener
MSWindowsComandServer

Java Case Study - Mail Merge - Solves a problem or a problem domain?

Does the example solve a specific problem or solve a problem domain?

As the original problem domain was defined as *"create a system to merge addresses with a LaTex document"* the application does, for the most part, provide a solution for the domain.

There are hard-coded elements that narrow the domain to a Windows system which uses **Ghostview** to print PostScript files and has the sofware stored at `c:/GSTools/GSView/`. If the code was used on a system that stored Ghostview in another location it would fail. The code would have to be altered either by hard-coding the new location (not a recommended solution) or by providing the user a means to input the required location.

Had the problem domain been defined in wider terms i.e. as *"develop a mail merge system"*, then no, the application would not provide a solution for the domain.

The scope of a problem domain can be as narrow or as wide as the user wants. One of the key problems in establishing requirements is determining *exactly* what the problem domain is. If the person requesting the system and the person desigining the system have different ideas concerning what is inside the domain then the resulting system will either fail to meet user expectations or go beyond user expectations. In the first instance you'll have an unhappy user. You may also have an unhappy user in the second instance, especially if you could have produced code that met the users expectations in half the time and at half the cost.

Home | Case Studies | Previous | TOC

Java Case Study - JCalculator

Source

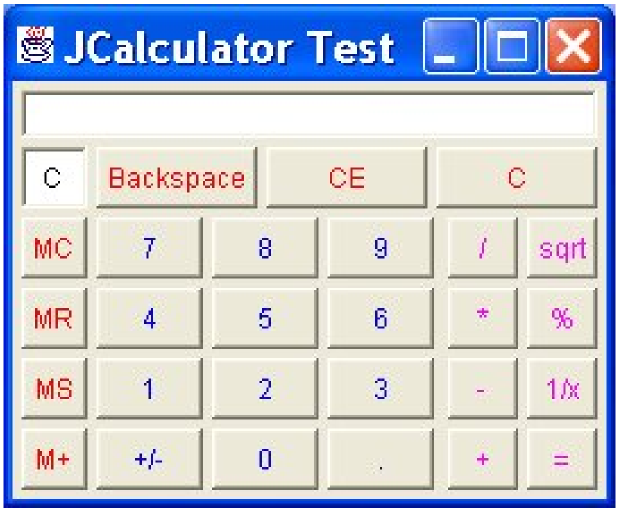
The code for this study is from *Sum it up with JCalculator* an article by Claude Duguay in *JavaPro*, *August 2001*, *Vol.5 No. 8*,and may be downloaded from Devx

Java Case Study - JCalculator - Overview

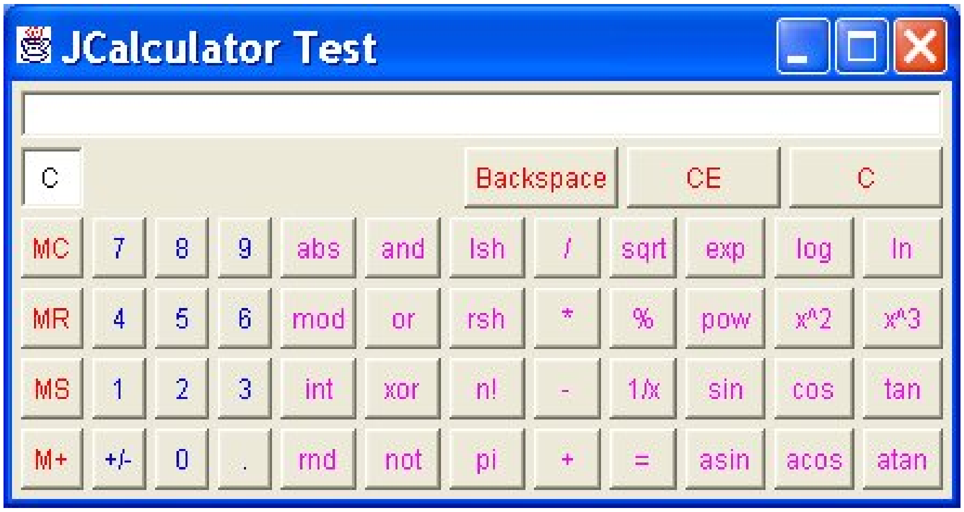
Problem Statement

Implement a numerical calculator that can easily be added to any Swing application. Provide basic arithmetic and trigonometric functions along with features found on most standard calculators: clear an entry, clear all entries, memory clear, memory recall, etc. The application can be started in simple or expanded mode.

GUI Simple View



GUI - Exanded View



Java Case Study - JCalculator - User Defined Types

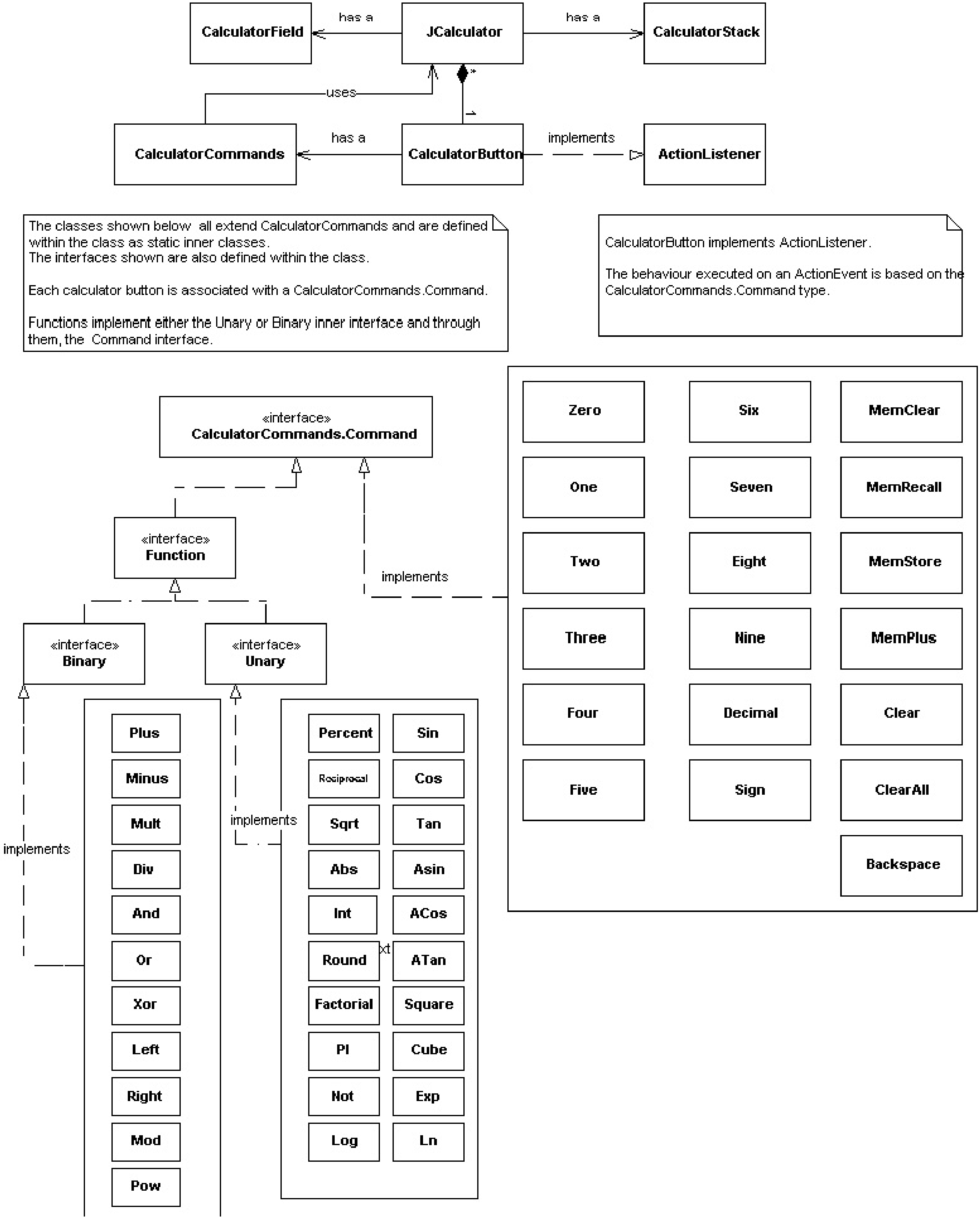
User Defined Types

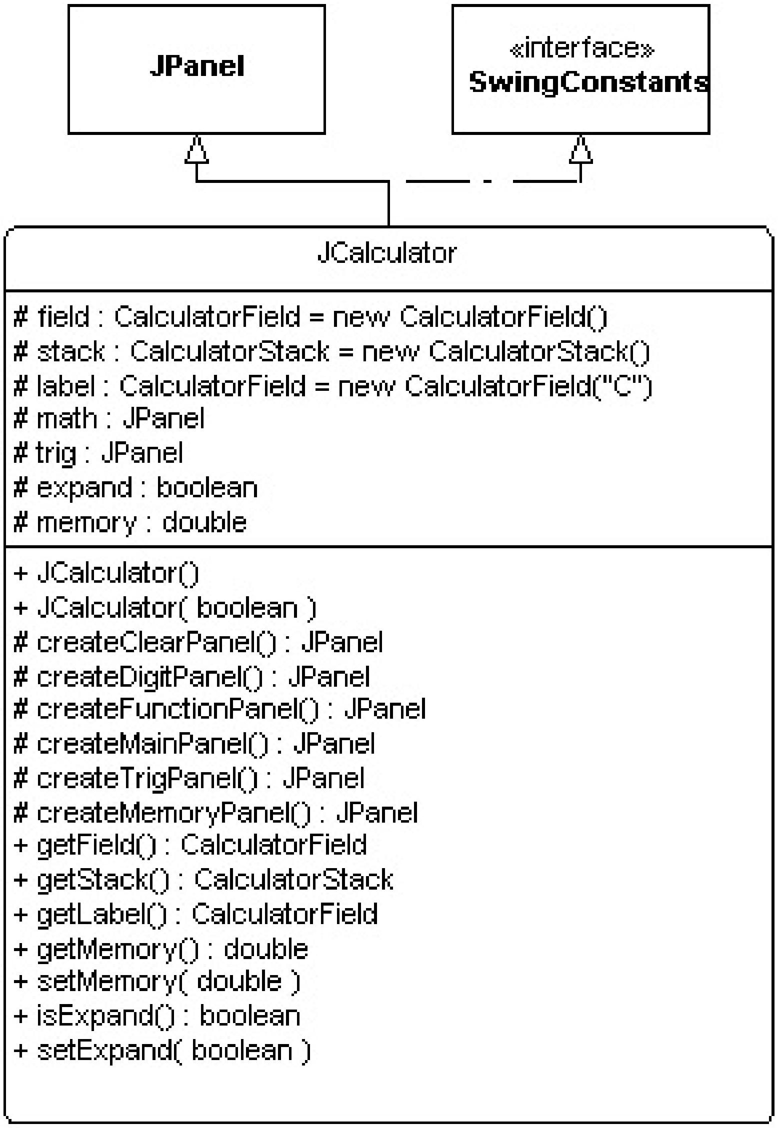
Full UML Class Diagram

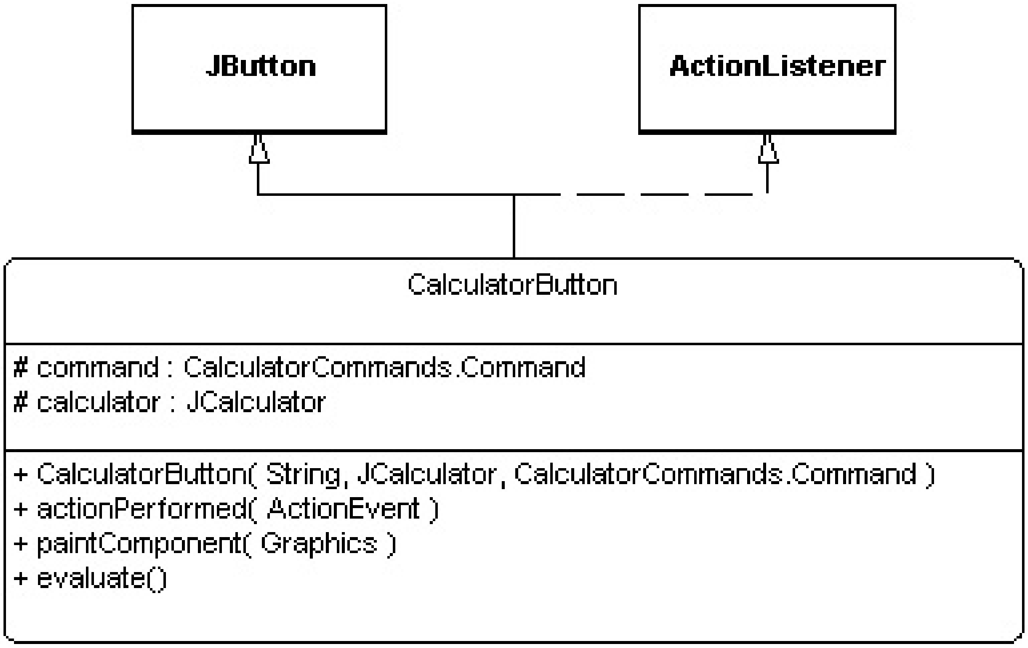
The application is implemented with the following user defined types:

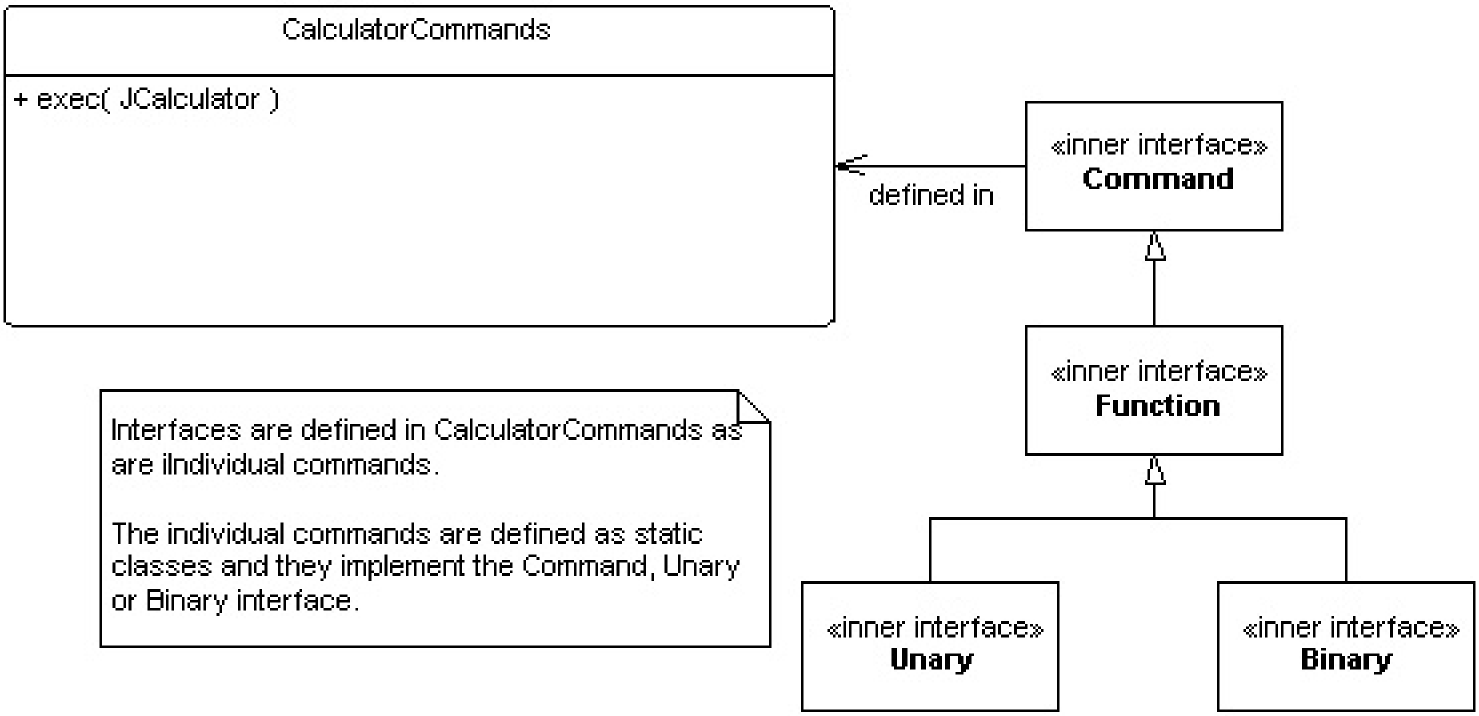
- JCalculator - defines a calculator object
- CalculatorButton - defines a button used by the calculator object
- CalculatorCommands - defines the commands associated with calculator buttons
- CalculatorField - defines objects used to display information in the calculator
- CalculatorStack - defines an object to hold the intermediary results of calculator button operations

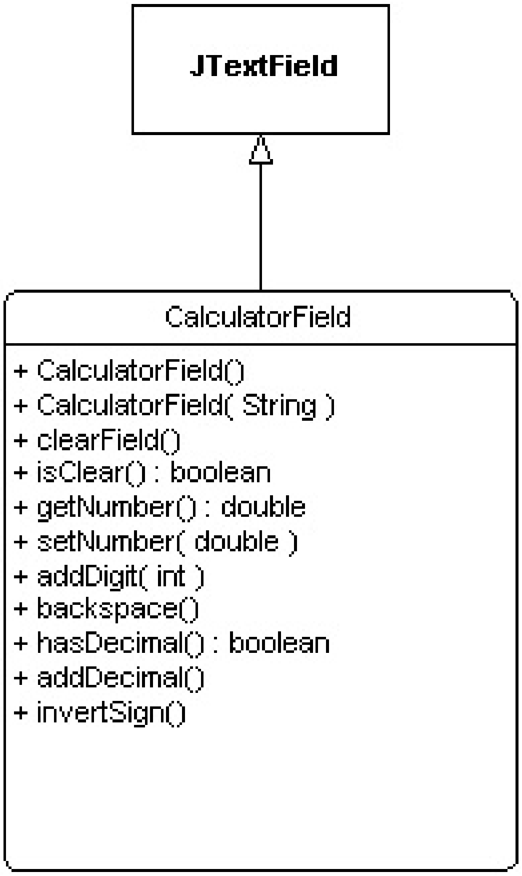
Home | Case Studies | Previous | TOC | Next

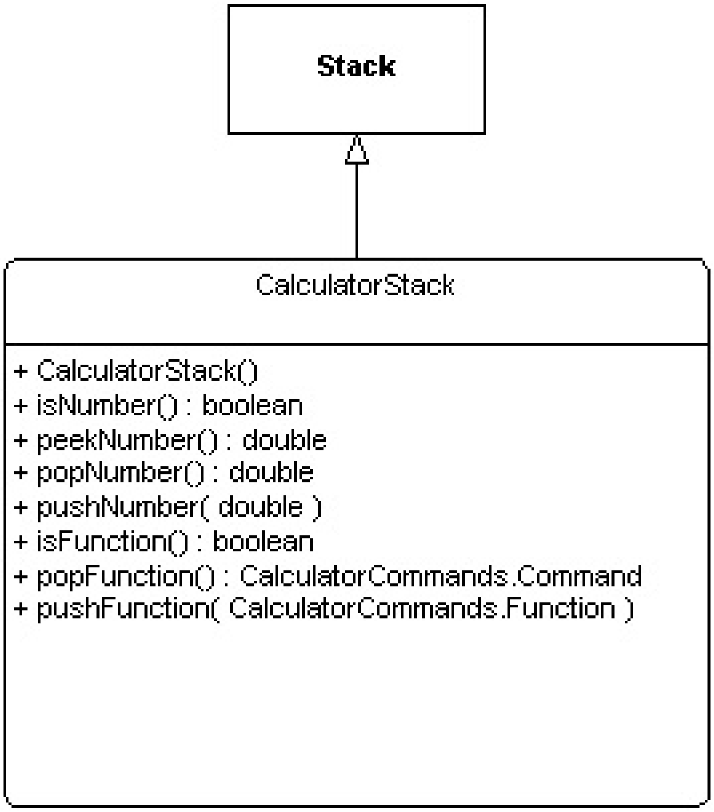

CalculatorField
has a
JCalculator
has a
CalculatorStack
uses
CalculatorCommands
has a
CalculatorButton
implements
ActionListener
The classes shown below all extend CalculatorCommands and are defined within the class as static inner classes.
The interfaces shown are also defined within the class.
Each calculator button is associated with a CalculatorCommands.Command.
Functions implement either the Unary or Binary inner interface and through them, the Command interface.
CalculatorButton implements ActionListener.
The behaviour executed on an ActionEvent is based on the CalculatorCommands.Command type.
«interface»
CalculatorCommands.Command
«interface»
Function
implements
«interface»
Binary
«interface»
Unary
implements
implements
Plus
Minus
Mult
Div
And
Or
Xor
Left
Right
Mod
Pow
Percent
Sin
Reciprocal
Cos
Sqrt
Tan
Abs
Asin
Int
ACos
Round
ATan
Factorial
Square
PI
Cube
Not
Exp
Log
Ln
Zero
Six
MemClear
One
Seven
MemRecall
Two
Eight
MemStore
Three
Nine
MemPlus
Four
Decimal
Clear
Five
Sign
ClearAll
Backspace


JPanel
«interface»
SwingConstants
JCalculator
field : CalculatorField = new CalculatorField()
stack : CalculatorStack = new CalculatorStack()
label : CalculatorField = new CalculatorField("C")
math : JPanel
trig : JPanel
expand : boolean
memory : double
+ JCalculator()
+ JCalculator(boolean)
# createClearPanel() : JPanel
# createDigitPanel() : JPanel
# createFunctionPanel() : JPanel
# createMainPanel() : JPanel
# createTrigPanel() : JPanel
# createMemoryPanel() : JPanel
+ getField() : CalculatorField
+ getStack() : CalculatorStack
+ getLabel() : CalculatorField
+ getMemory() : double
+ setMemory(double)
+ isExpand() : boolean
+ setExpand(boolean)


JButton
ActionListener
CalculatorButton
command : CalculatorCommands.Command
calculator : JCalculator
+ CalculatorButton(String, JCalculator, CalculatorCommands.Command)
+ actionPerformed(ActionEvent)
+ paintComponent(Graphics)
+ evaluate()


CalculatorCommands
+ exec(JCalculator)
defined in
«inner interface»
Command
«inner interface»
Function
«inner interface»
Unary
«inner interface»
Binary
Interfaces are defined in CalculatorCommands as are iIndividual commands.
The individual commands are defined as static classes and they implement the Command, Unary or Binary interface.


JTextField
CalculatorField
+ CalculatorField()
+ CalculatorField(String)
+ clearField()
+ isClear() : boolean
+ getNumber() : double
+ setNumber(double)
+ addDigit(int)
+ backspace()
+ hasDecimal() : boolean
+ addDecimal()
+ invertSign()


Stack
CalculatorStack
+ CalculatorStack()
+ isNumber() : boolean
+ peekNumber() : double
+ popNumber() : double
+ pushNumber(double)
+ isFunction() : boolean
+ popFunction() : CalculatorCommands.Command
+ pushFunction(CalculatorCommands.Function)

Java Case Study - JCalculator - Where the action is

Where the action is

The design utilizes the [Command Pattern] with all the action being handled by `CalculatorButton`. The operation performed is determined by the buttons *command object*. The `CalculatorButton` constructor takes the following form:

```
        public CalculatorButton( String text,
                                 JCalculator calculator,
                                 CalculatorCommands.Command command)
```

where, `String` is the buttons label text, `JCalculator` is a reference to the current calculator object and `CalculatorCommands.Command` is a command object.

`CalculatorCommands.Command` is an interface defined within the `CalculatorCommands` class. The `CalculatorCommands` class contains the definitions for three other interfaces: `Function, Unary` and `Binary`. The `Function` interface extends the `Command` interface and `Unary` and `Binary` extend `Function`. Individual command objects directly implement `Command, Unary` or `Binary`. All of them are of type `CalculatorCommands.Command`. (see [CalculatorCommands])

Each command object implements the `exec()` method declared in the `Command` interface. In the example, all the individual commands i.e. One, Plus, Clear, etc. are declared as `static member classes` of `CalculatorCommands`. Each calculator button is given a specific command object. For example, code that adds a `CalculatorButton` to the `JCalculator` object is:

```
        add( new CalculatorButton( "8", this, new CalculatorCommands.Eight() ) );
```

where "8" is the label that will appear on the button, `this` is the current `JCalculator` object and `new CalculatorCommands.Eight()` creates the command object that will be associated with the `CalculatorButton`. When a calculator button is clicked, an `ActionEvent` is generated and listeners are notified. A `CalculatorButton` object acts as it's own listener by implementing the `actionPerfomed()` method of the `ActionListener` interface.

The implementation of the Command pattern in this example is a little unusual. Generally, the `actionPerfomed()` method in a Command pattern example is very simple:

```
    public void actionPerformed(ActionEvent e){
        command.exec();
    }
```

where all the behaviour associated with the button would be implemented in the `exec()` method of the command object. In this example, the button itself is controlling some of the behaviour.

```
    public void actionPerformed(ActionEvent event)
      {
        if (command != null)
        {
          if (command instanceof CalculatorCommands.Unary)
          {
            evaluate();
            CalculatorStack stack = calculator.getStack();
            stack.pushFunction((CalculatorCommands.Function)command);
            evaluate();
          }
```

```
        if (command instanceof CalculatorCommands.Binary)
        {
          evaluate();
          CalculatorField field = calculator.getField();
          CalculatorStack stack = calculator.getStack();
          stack.pushNumber(field.getNumber());
          stack.pushFunction((CalculatorCommands.Function)command);
          field.clearField();
        }
        if (!(command instanceof CalculatorCommands.Function))
        {
          command.exec(calculator);
        }
      }
      // Handle '='
      else evaluate();
    }
```

Command logic is being handled within the `actionPerformed()` method and a portion of the command behaviour is implemented in the `CalculatorButton.evaluate()` method rather than by the command object itself.

```
  protected void evaluate()
    {
      CalculatorStack stack = calculator.getStack();
      if (!stack.isEmpty() && stack.isFunction())
      {
        CalculatorField field = calculator.getField();
        CalculatorCommands.Function function = stack.popFunction();
        stack.pushNumber(field.getNumber());
        function.exec(calculator);
        field.setNumber(stack.popNumber());
      }
    }
```

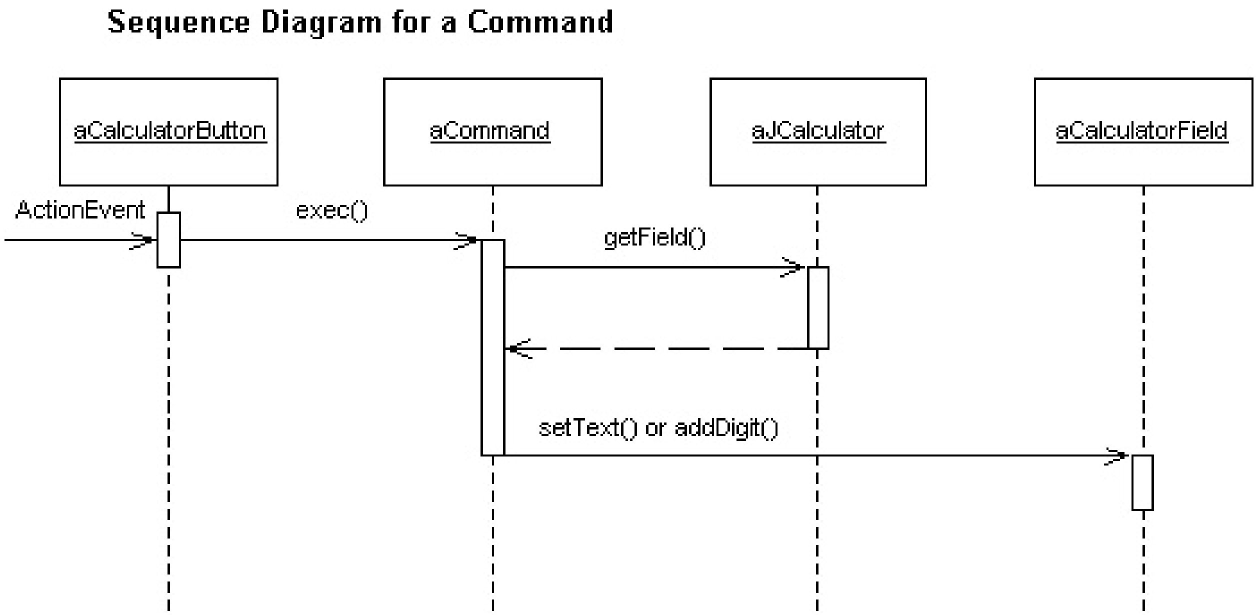
This makes it a little more difficult to work out what is actually happening when a calculator button is clicked on. *UML Sequence diagrams* can help when you're trying to sort out interactions between objects.

Home | Case Studies | Previous | TOC | Next

Java Case Study - JCalculator - Command Behaviour

Command Behaviour

Command objects which are not *Functions* (One, Two, Clear, etc) directly implement the `CalculatorCommands.Command` interface. When a button associated with a `Command` type is clicked, the `actionPerformed()` method invokes the objects `exec()` method; the `CalculatorButton.evaluate()` method is not invoked.



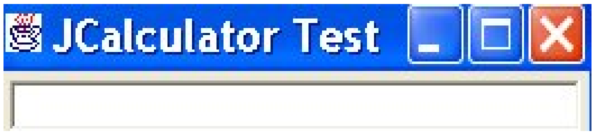
From the above we can see that

1. when an `ActionEvent` is triggered it is sent to a `CalculatorButton`
2. the button invokes the `exec()` method of its assoicated command object, passing it a reference to the calculator object
3. the command object uses the reference to the calculator object to get a reference to the `CalculatorField` being used to display the numbers entered by the user and the results of any calculations
4. the command object then uses the field reference to set the text in the field or add a digit to the text already being displayed

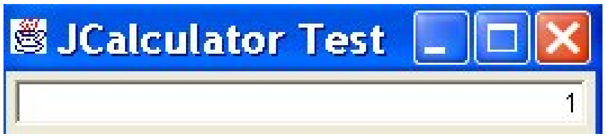
Let's say a user clicks on the "1" calculator button. This generates an `ActionEvent` and the button is notified; invoking its `actionPerfomed()` method. The method checks to make sure the command associated with itself is not a `Function` and calls the `exec()` method of it's Command object. In this case, the object is type `One`. The `exec()` method in the `One` class is implemented as follows:

```
public void exec(JCalculator calculator)
{
  CalculatorField field = calculator.getField();
  field.addDigit(1);
}
```

The `addDigit()` of `CalculatorField` actually concatenates the digit '1' to any text currently being displayed. For example, what we see before we click on the '1' button is:



What we see after clicking the '1' button is:

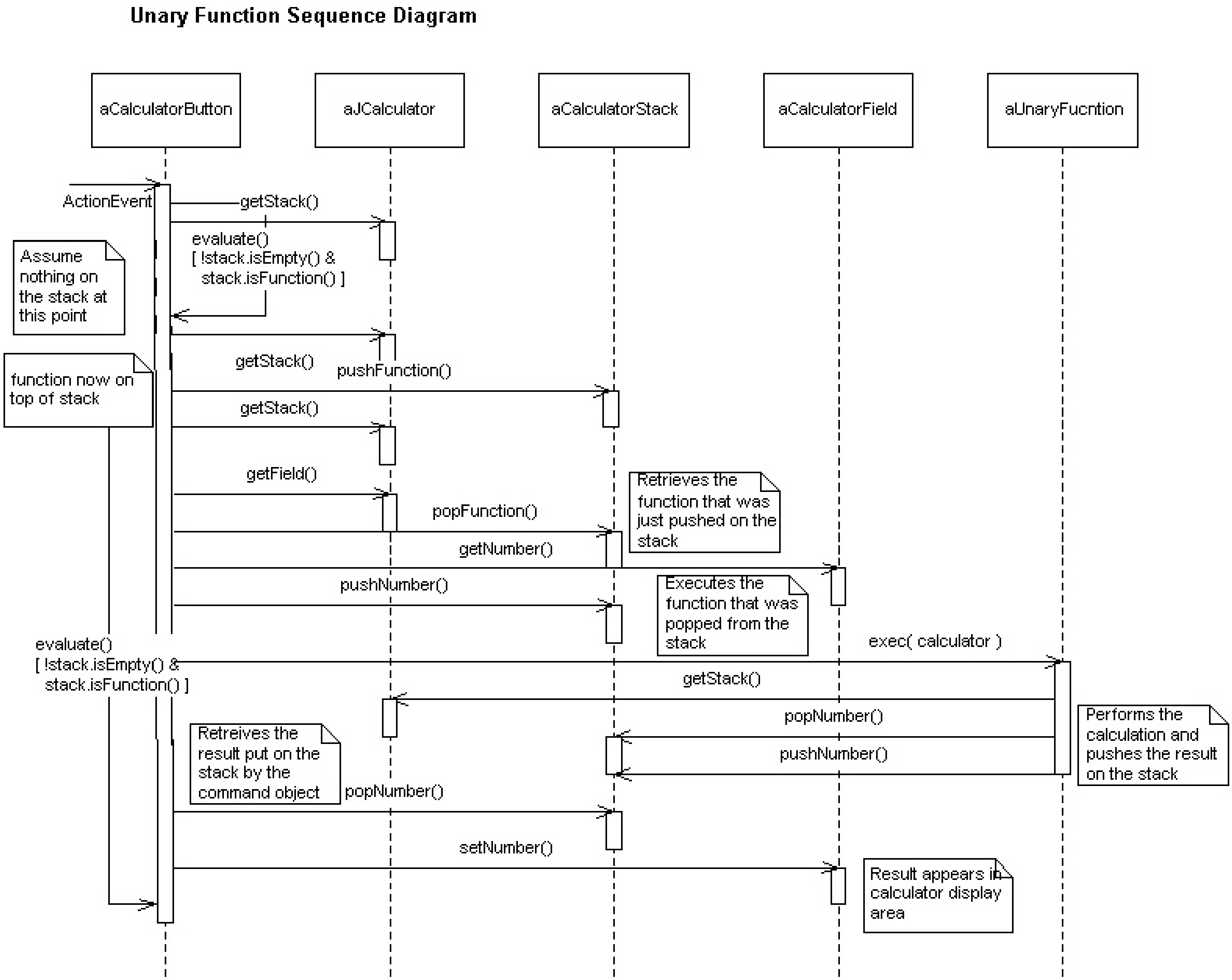


That's fairly straight forward. Unary function commands are a bit more complicated.

Home | Case Studies | Previous | TOC | Next

Java Case Study - JCalculator - Unary Function Behaviour

Unary Function Behaviour



Every button has an associated CalculatorCommands.Command object reference and a JCalculator reference.

Lets assume the calculator button that was clicked has a `Sqrt` command of type CalculatorCommands.Sqrt which implements Unary. The code for Sqrt is:

```
public static class Sqrt implements Unary
  {
    public void exec(JCalculator calculator)
    {
      CalculatorStack stack = calculator.getStack();
      stack.pushNumber(Math.sqrt(stack.popNumber()));
    }
  }
```

The Sqrt button is clicked, an ActionEvent is raised and the buttons actionPerformed() method is invoked.

1. the actionPerformed() method checks its objects (the buttons) command object type and determines its of type CalculatorCommands.Unary
2. it invokes its own evaluate() method
3. the buttons reference to the calculator object is used to get a reference to the calculators stack
4. the stack isEmpty() method is invoked and returns 'true' so evaluate() returns control to the actionPerformed() method
5. the actionPerformed() method retrieves a reference to the calculators stack through the buttons calculator reference
6. the buttons command object (in this case aSqrt) is pushed onto the stack
7. the buttons evaluate() method is again invoked
8. a reference to the calculators stack is retrieved
9. this time the stack is not empty and its top object is a function (the Sqrt object)
10. the buttons calculator reference is used to retrieve a reference to the calculators display field
11. the Sqrt object is popped off the calculators stack
12. the field reference is used to retrieve the number currently displayed
13. the retrieved number is pushed onto the stack
14. the Sqrt objects exec() method is invoked and a copy of the buttons calculator reference is passed as an argument
15. the Sqrt object uses its calculator reference (the one passed to exec()) to get a reference to the calculator stack
16. the number pushed onto the stack in evaluate() is popped off the stack and used as an argument to Math.sqrt()
17. the result returned by Math.sqrt() is pushed onto the stack
18. control returns to the buttons evaluate() method
19. the result pushed onto the stack by the Sqrt object is retrieved and passed to the calculator field by invoking the fields setNumber() method
20. the evaluate() method returns control to the actionPerformed() method
21. there is nothing else to do

Whew! There are an awful lot of busy objects! The actual job of providing the square of a number is handled by the command object, Sqrt, but the responsibility for getting everything ready for the Sqrt object is being handled by the button object.

Hmmm ... still not all that clear; lets try to think of it as a conversation between actors. The cast:

- Button - a CalculatorButton
- Calculator - a JCalculator that belongs to Button
- Sqrt - a CalculatorCommands.Unary that belongs to Button
- Math.sqrt - a friend of Sqrt's
- ActionPerformed - Button's helper
- Evaluate - Button's helper
- Stack - a CalculatorStack that belongs to Buttons calculator
- Display - a CalculatorField that belongs to Buttons calculator

```
    Button:             "Hey, I've just been clicked! ActionPerformed, you need to 
get to work!"

    ActionPerformed:    "Ok. Do we have a command? Oh yeah, a Sqrt and its a Unary. 
Evaluate
                         can you check things for me?"

    Evaluate:           "Sure thing. Calculator, pass me your Stack for a minute."

    Calculator:         "Here he is." [Hands Stack to Evaluate]

    Evaluate:            "Oops .. its empty nothing for me to do. ActionPerformed, 
it's up to you."

    ActionPerformed:    "Calculator, let me have your Stack. Stack, here, take a copy 
of Sqrt."
                        [Hands Sqrt to Stack] "Ok Evaluate, your turn again."
```

```
    Evaluate:           "Calculator, can you let me see Stack again?"

    Calculator:         "Sure." [Hands Stack to Evaluate]

    Evaluate:           "Alright Stack are you empty?"

    Stack:              "Nope."

    Evaluate:           "Do you have a function?"

    Stack:              "Let me see, yup, I got a function on top of me."

    Evaluate:           "Great. Calculator, can you give me your Display?"

                        [Calculator hands Display to Evaluate]

    Evaluate:           "Stack, let me have that function and Display, you give Stack
                         the number you're holding."

                        [Stack hands Sqrt to Evaluate and Display gives Stack a
number]

    Evaluate:           "Ok Sqrt, you do your thing. Oops, here you need to talk to
Calculator"
                        [Hands Sqrt a connection to Calculator]

    Sqrt:               "Calculator, give me Stack please."

                        [Calculator hands over Stack]
  Stack (sotto voice): "Hey, I'm tired of being man handled! Geesh, don't you guys
have
                           anything better to do!"

    Sqrt:               "Stack, give me the number your holding. I need to pass it to
Math.sqrt."

    Math.sqrt:          "Here Sqrt, I did my thing with the number, you can have it
back now."

    Sqrt:               "Here you go Stack ... take number back now. Hey Evaluate,
I'm finished."

    Evaluate:           "Stack, let me have the number Sqrt just gave you." [Stack
hands
                         the number to Evaluate]. "Display, can you show this to
everyone?"

    Display:            "Sure thing." [Display takes the number and holds it up for
all
                         to see]

    Evaluate:           "Ok ActionPerformed, I'm finished!"

    ActionPerformed:    "Button, we're all done now."

    Button:             "Thanks guys. What a team!"

                                [The End]
```

That's a little clearer. The calculator is basically acting as a holder for all the objects. The `actionPerformed()` and `evaluate()` methods in `CalculatorButton` are directing events and the actual work/function is being handled by the

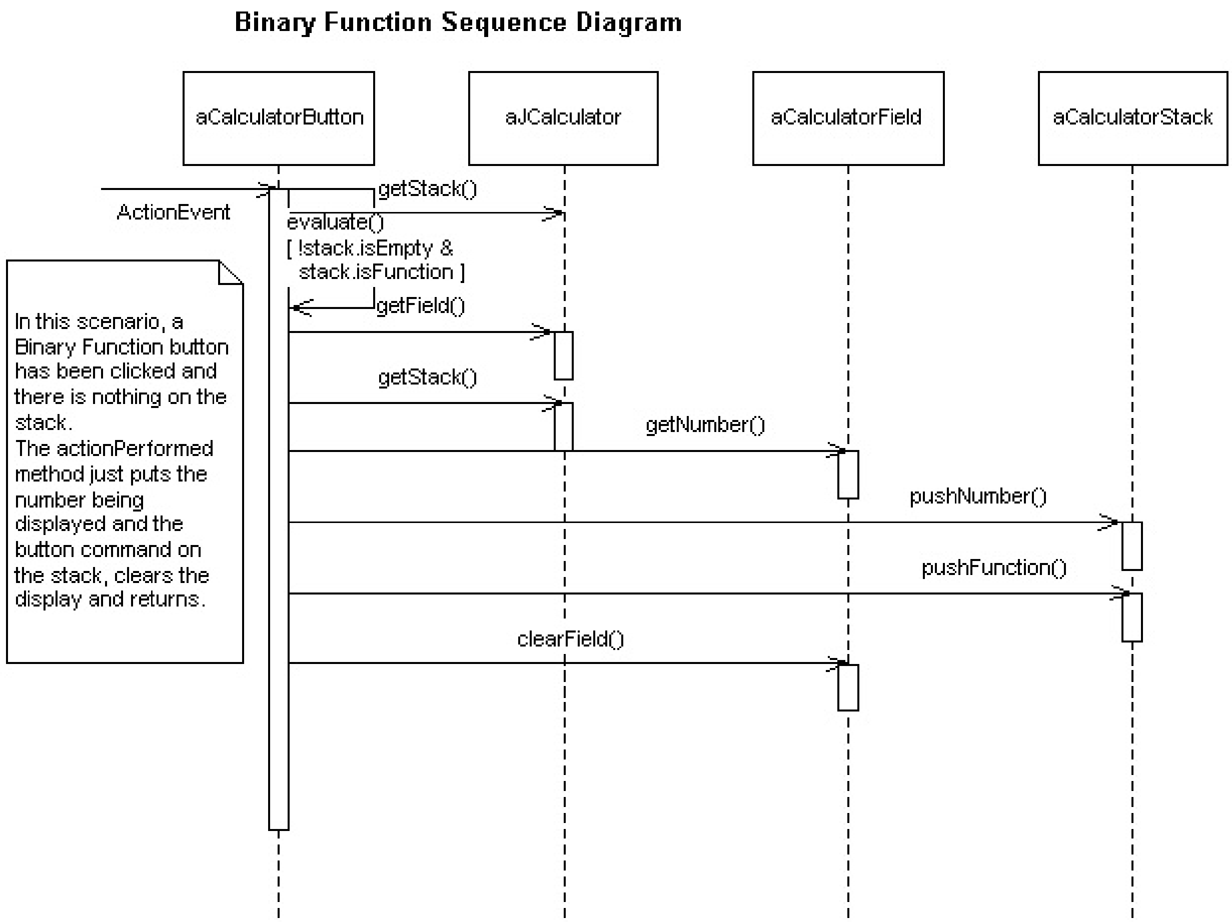
`CalculatorCommands.Command` object.

So what happens if the command object is a Binary function?

Home | Case Studies | Previous | TOC | Next

Java Case Study - JCalculator - Binary Function Behaviour

Binary Function Behaviour

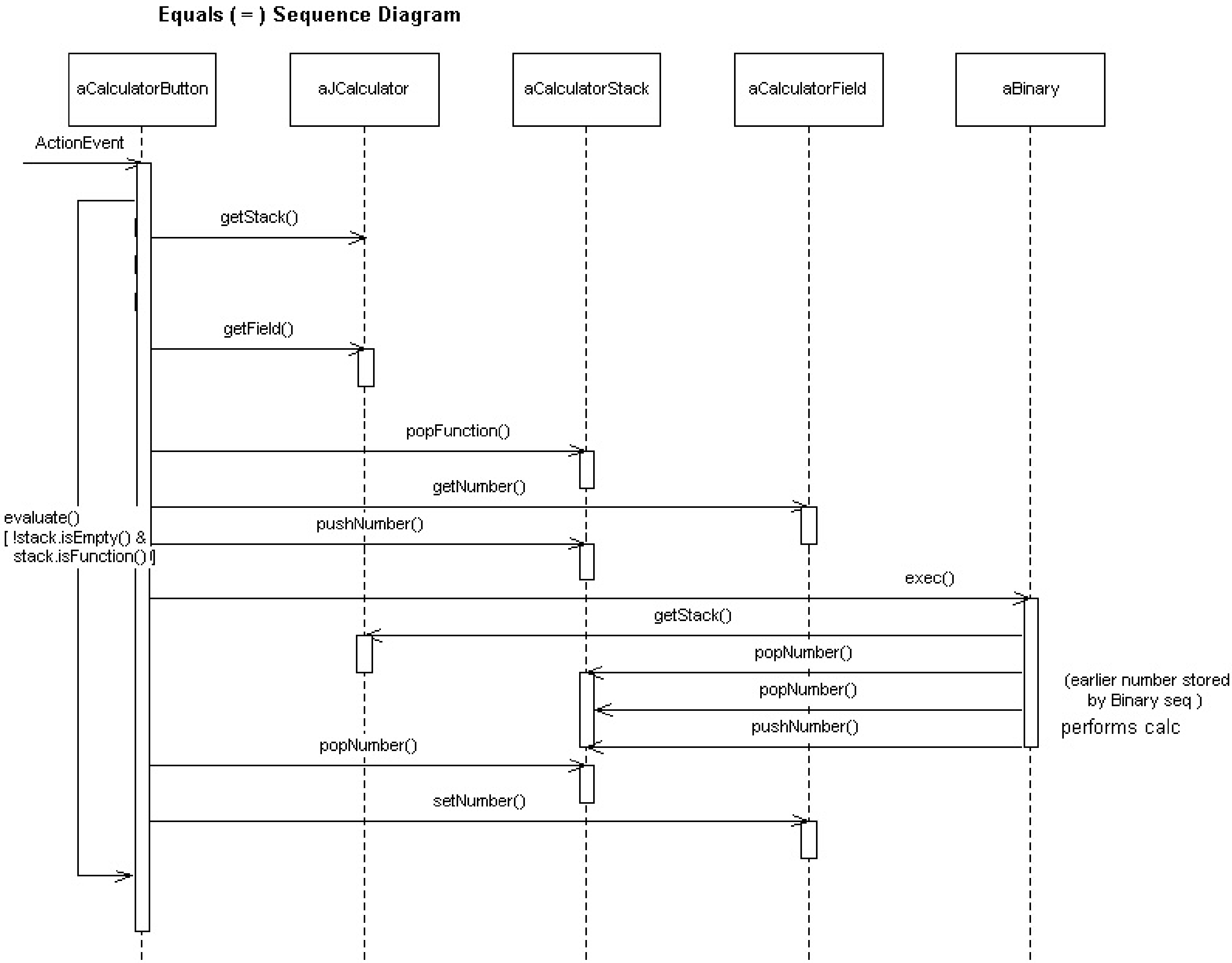


Hmmm .. the stack is being setup but no calculations are happening. How does a Binary function get executed? You'd expect the user to enter another digit followed by "=". Lets take another look at `actionPerformed()`.

```
public void actionPerformed(ActionEvent event)
        {
          if (command != null)
          {
            if (command instanceof CalculatorCommands.Unary)
            {
              evaluate();
              CalculatorStack stack = calculator.getStack();
              stack.pushFunction((CalculatorCommands.Function)command);
              evaluate();
            }
            if (command instanceof CalculatorCommands.Binary)
            {
              evaluate();
              CalculatorField field = calculator.getField();
              CalculatorStack stack = calculator.getStack();
              stack.pushNumber(field.getNumber());
              stack.pushFunction((CalculatorCommands.Function)command);
              field.clearField();
            }
            if (!(command instanceof CalculatorCommands.Function))
            {
```

```
            command.exec(calculator);
          }
        }
        // Handle '='
        else evaluate();
      }
```

It looks like "=" is not a Command object. Only the evaluate() method comes into play. Ok, so let's say the user enters '1 + 1 =', what happens?



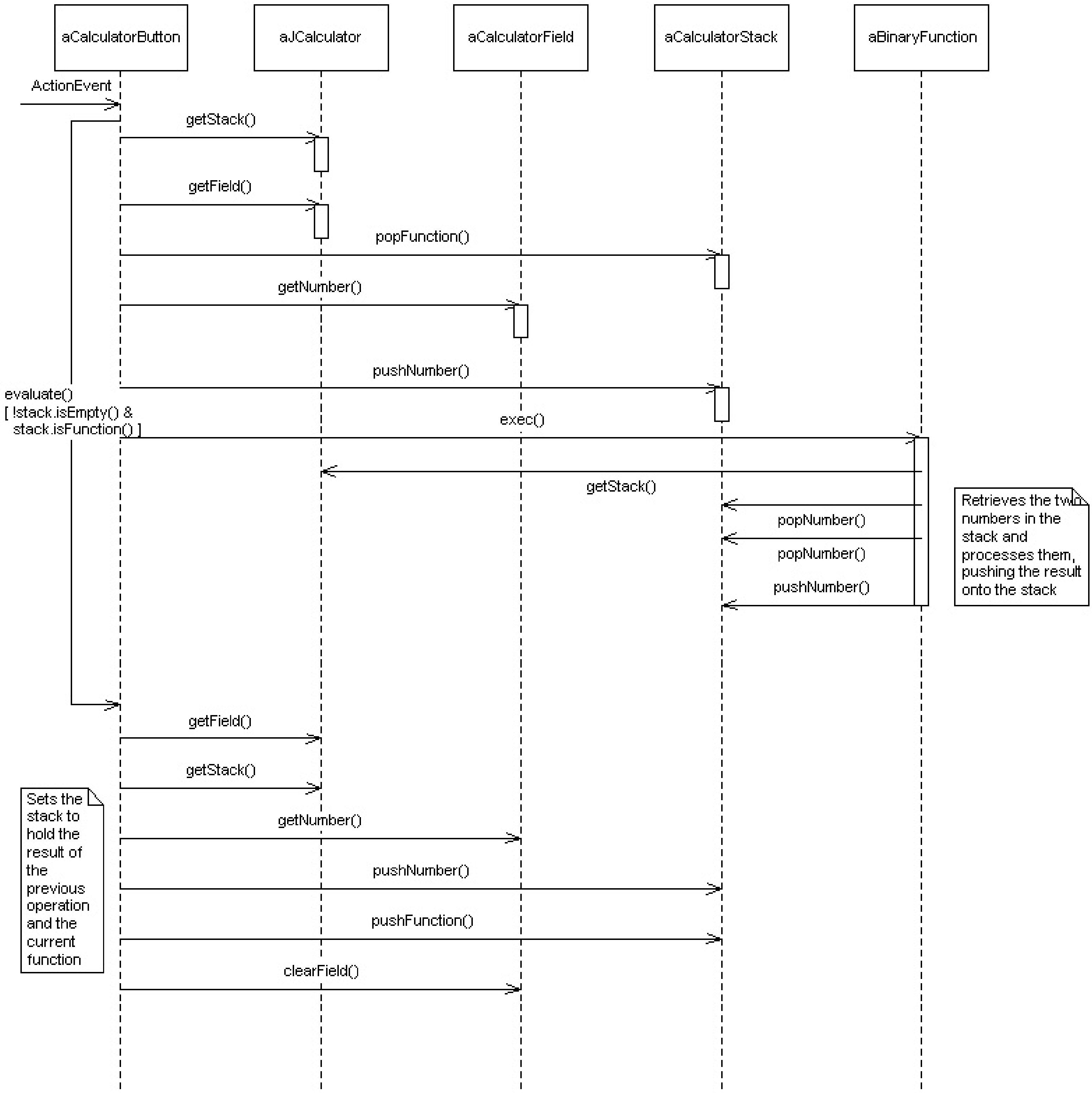
This time there is something on the stack; the '1' and '+' placed there earlier, so evaluate() pops off the function, pushes the current number (the one in the display area) onto the stack and then invokes the functions exec() method.

Then Binary.exec() method then retrieves both the numbers from the stack and performs its operation, pushing the result back onto the stack.

The evaluate() method then pops the result off the stack and calls the display fields setNumber() method; which shows the result of the operation to the user.

But what happens if '=' isn't pressed after the second digit is entered? What if the user enters '1 + 1 + 2' before hitting '='?

Multiple Binary Function Sequence Diagram



Almost the same thing as happens when '=' is pressed except that the previous function is evaluated and the result of the operation is placed on the stack followed by the current function. The '1 + 1 + 2' would result in the following:

```
User Enters            Field Display          Stack
    1                      1                  empty
    +                      blank              + 1
    1                      1                  + 1
    +                      blank              + 2    ( result of previous stack
```

```
value '1'
                                                            '+' current display
value of '1' )
        2                       2                  + 2
        =                       4                  empty  ( previous result value is
added to
                                                            current display value )
```

Home | Case Studies | Previous | TOC | Next

Java Case Study - JCalculator - Summary

Summary

Well, originally I thought that this was an elegant, straight forward design. After working through the sequence diagrams though, I now have my doubts. I don't think working out code logic should be quite that difficult!

Admittedly, the code was not written as a demonstration of design technique; the author provided it as a plug-in component that can be used in any program; and it does work quite nicely.

Home | Case Studies | Previous | TOC |

The `java.io` Package - Overview

- the package contains three main groups of classes and interfaces
  1. classes to build data streams
  2. classes and interfaces for serialization
  3. classes and interfaces for working with the file system

Data Streams (JCL1)

- data streams that read values from a data source are *input* streams
- data streams that write values to a data repository are *output* streams
- the data can be either **byte** or **character** values

DataStream SuperClasses	
Byte Streams	Character Streams
abstract class InputStream	abstract class Reader
abstract class OutputStream	abstract class Writer

- there are two classes which convert bytes to characters

```
class InputStreamReader extends Reader
class OutputStreamWriter extends Writer
```

- data containers, for example files, usually provide methods which return a stream for either reading or writing
- data streams can be chained together

Filter Streams

- *filter streams* perform some processing or filtering as the data is passed through
- a **filter ouput** stream performs the processing **before** the data is written out
- a **filter input** stream performs the processing **after** the data is read from its original source

FilterStream SuperClasses	
Byte Streams	Character Streams
class FilterInputStream	class FilterReader
class FilterOutputStream	class FilterWriter

- there are number of filter streams for both byte streams

```
BufferedInputStream           BufferedOutputStream
DataInputStream               DataOutputStream
LineNumberInputStream         PrintStream
PushbackInputStream
```

- and character streams

```
BufferedReader                PrintWriter
LineNumberReader
PushbackReader
```

In-Memory Streams

- there are also classes for reading and writing data held in memory

```
ByteArrayInputStream                 CharArrayReader
ByteArrayOutputStream                CharArrayWriter
                                     StringReader
                                     StringWriter
```

- the StringReader/Writer classes read data from a StringBuffer object

Pipes

- there are classes that allow you to build streams that operate between **threads**

```
PipedInputStream                     PipedReader
PipedOutputStream                    PipedWriter
```

Files

- there are a number of classes for working with the file system

```
File
FileDescriptor

FileInputStream                      FileReader
FileOutputStream                     FileWriter
FilenameFilter
FilePermission                       RandomAccessFile
```

- note that File, FileDescriptor and RandomAccessFile are direct subclasses of Object

Note
The File class can be used to create **directories**

Serialization (JCL1)

- *serialization* is the process of converting an **object** to a stream of bytes in such a manner that the original **object** can be rebuilt (lets you write an object to a file or other data container)

	Pkg Overview	Data Streams	Character Streams	Byte Streams	File Class	Readers & Writers
	Filter Streams	Data Input/Output	Reading & Writing Files	Serialization		

The `java.io` Package - Data Streams

- Suns Java Tutorial on I/O also breaks up the classes into Data Sink Streams and Data Processing Streams
- a **sink** is a specialized data container ie strings, files, pipes

Sink Type	Character Streams	Byte Streams
Memory	`CharArrayReader, CharArrayWriter`	`ByteArrayInputStream, ByteArrayOutputStream`
	`StringReader, StringWriter`	`StringBufferInputStream`
Pipe	`PipedReader, PipedWriter`	`PipedInputStream, PipedOutputStream`
File	`FileReader, FileWriter`	`FileInputStream, FileOutputStream`

- data processing streams perform some type of operation ie buffering or character encoding

Process	CharacterStreams	Byte Streams
Buffering	`BufferedReader, BufferedWriter`	`BufferedInputStream, BufferedOutputStream`
Filtering	`FilterReader, FilterWriter`	`FilterInputStream, FilterOutputStream`
Converting between Bytes and Characters	`InputStreamReader, OutputStreamWriter`	
Concatenation		`SequenceInputStream`
Object Serialization		`ObjectInputStream, ObjectOutputStream`
Data Conversion		`DataInputStream, DataOutputStream`
Counting	`LineNumberReader`	`LineNumberInputStream`
Peeking Ahead	`PushbackReader`	`PushbackInputStream`
Printing	`PrintWriter`	`PrintStream`

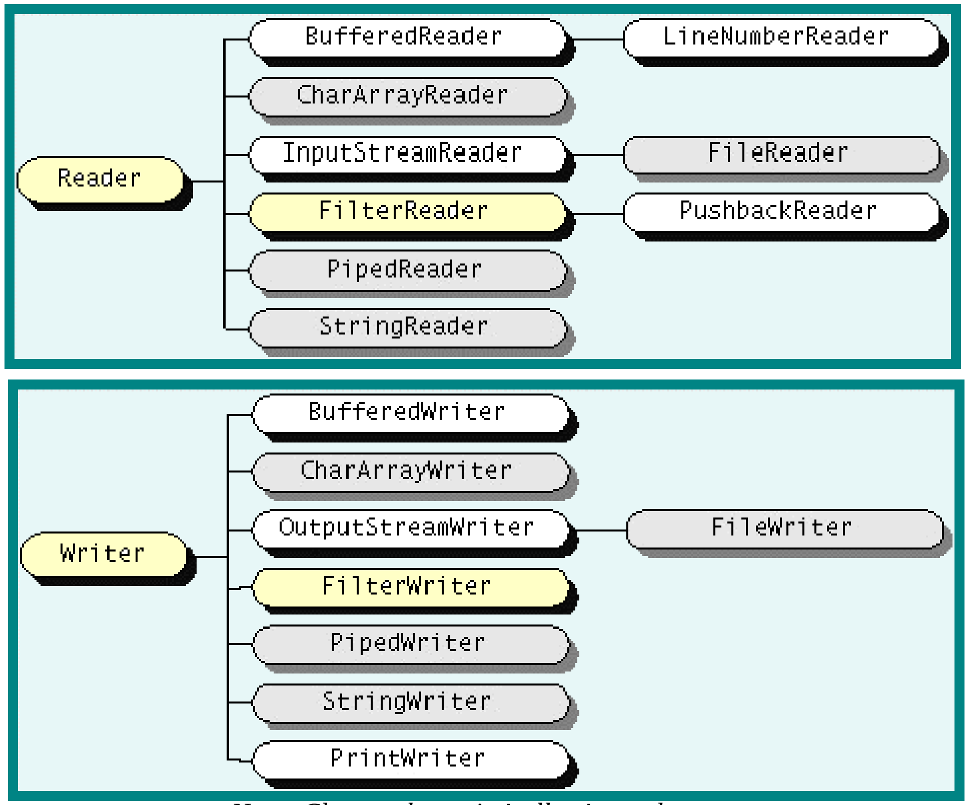
Pkg Overview | Data Streams | Character Streams | Byte Streams | File Class | Readers & Writers
Filter Streams | Data Input/Output | Reading & Writing Files | Serialization

Java Quick Reference

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

The `java.io` Package - Character Streams

- **Reader** and **Writer** are the **abstract** superclasses for all character streams.



Note: Classes shown in 'yellow' are abstract.
Items shown in 'gray' read and write from data sinks.
Images are from the Sun Tutorial on I/O

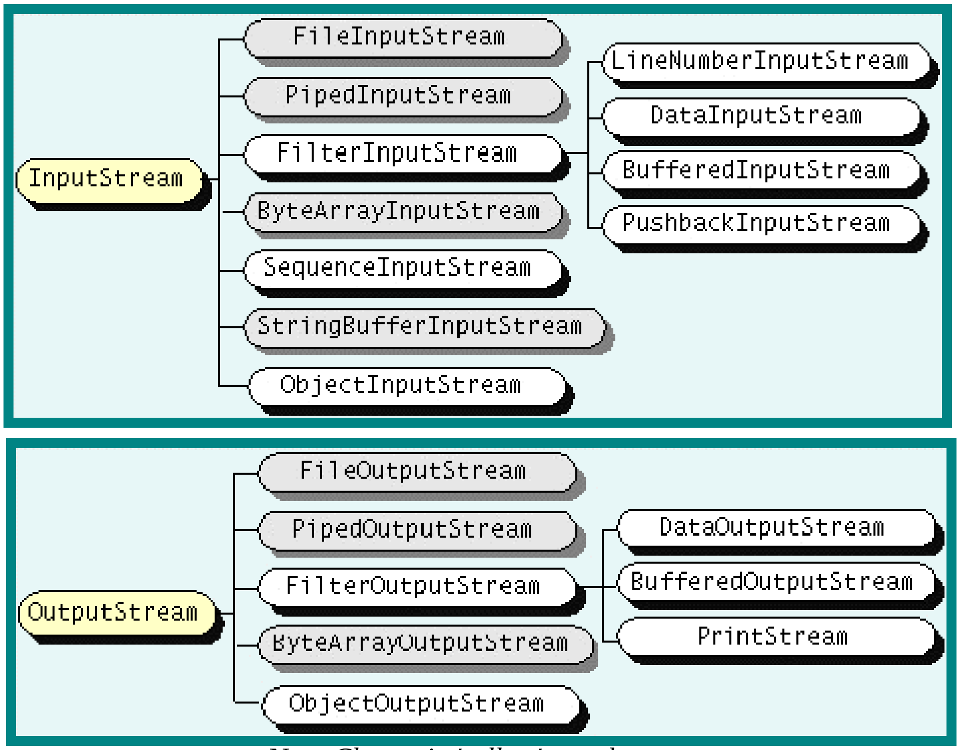
- character streams can read or write any Unicode character set.
- Byte streams are limited to ISO-Latin-1 8-bit encoding.

	Pkg Overview	Data Streams	Character Streams	Byte Streams	File Class	Readers & Writers
	Filter Streams	Data Input/Output	Reading & Writing Files	Serialization		

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

The `java.io` Package - Byte Streams

- use **InputStream** and **OutputStream** classes to read 8-bit bytes



Note:*Classes in 'yellow' are abstract.*
Classes in 'gray' read and write to data sinks.
Images from Sun Java I/O tutorial

- **ObjectInputStream** and **ObjectOutputStream** are used for **serialization**

!!! Warning !!!
These classes cannot be used to read or write Unicode characters.

	Pkg Overview	Data Streams	Character Streams	Byte Streams	File Class	Readers & Writers
	Filter Streams	Data Input/Output	Reading & Writing Files	Serialization		

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

The `java.io` Package - File Class

- used to access file and directory objects using the file-naming, path conventions of the implementing operating system
- the class has three constructors

```
File(String pathname)
File(String parent, String child)
File(File parent, String child)

where,
    parent      is the pathname
    child       is the filename
```

- used to create an *instance* of a File BUT does not actually create a file

```
// does not create a file on the system
new File("test.txt");
```

- however, you can use the createNewFile() method

```
File f = new File("test.txt");
// returns 'false' if file exists
f.createNewFile();
```

- or, the method **createTempFile()** which creates the file in the default temporary directory using specified file extensions
- the class has four CONSTANTS which define properties of the file conventions on the operating system

char separatorChar
: the field is initialized to hold the system separator
/ for UNIX
\ for Win32
: for Mac

String separator
: a string representation of the separatorChar

char pathSeparator
: initialized to hold the character used by the system to separate file names in a list
: for UNIX
; for Win32

String pathSeparator
: string representation of the pathSeparator character

FileName Methods

- there are a number of methods for retreiving filenames, paths, etc

```
getAbsolutePath()        getAbsoluteFile()
getCanonicalPath()       getCanonicalFile()
getName()
getParent()              getParentFile()
getPath()
compareTo()
```

```
toURL()
```

- the **absolute path** is system dependent and may include **relative** indicators

```
For example, the following code creates a file
'test2.txt' in the directory directly above the
current directory:

File f1 = new File("..", "test2.txt")
f1.createNewFile();

System.out.println( f1.getAbsolutePath() );

Output (on Win98):
D:\Java\jeg\io\..\test2.txt
```

- the **canonical path** is the same as the **absolute path** BUT all relative indicators are resolved

```
For example,

System.out.println( f1.getCanonicalPath() );

Output (on Win98):
// '..' in absolute path is resolved
D:\Java\jeg\test2.txt
```

- **toURL()** will construct a valid URL identifier for the File

```
System.out.println( f.toURL() );

Output:
file:/D:/Java/jeg/io/test1.txt
```

Note
• the File class overrides the Object.equals() method. • Two files are equal() if they have the same **path**, NOT if they refer to the same underlying file system object.

File Status Methods

- there are methods to check the status of a file

```
canRead()        lastModified()      isDirectory()
canWrite()       setLastModified()   isFile()
exists()         setReadOnly()       isHidden()
length()                             isAbsolute()
```

Modifiying Files and Directories

- there are a number of methods for modifiying files and creating directories

```
delete()            mkdir()         listFiles()
deleteOnExit()      mkdirs()        listRoots()
renameTo()
```

- **list()** and **listFiles()** can be used with FilenameFilters ie '*'
- **listRoots()** returns the system **drives**
- while **renameTo()** will change the name of the file on the system, the reference will return the **original path and name**

```
// File object reference
```

```
File f  = new File("test.txt");
f.createNewFile();      // creates the file

// new File reference
File f2 = new File("testRename.txt");
f.renameTo(f2);     // renames the file

System.out.println( f.getAbsolutePath() );

Output (on Win98):
D:\Java\jeg\io\test1.txt // original path for 'f'

And if you check to see which file actually
exists on the system:

System.out.println( f.exists() );
System.out.println( f2.exists() );

Output:
  false
  true
```

Note
• There is no method which allows you to change directories!

Security

- many of the above methods will work correctly only if they are allowed by the **security permissions**
- for example, an Applet would probably not be allowed to create a new file

Source Code for Examples

- TestFileClass.java

The `java.io` Package - Readers and Writers

InputStreamReader

- **InputStreamReader** extends **Reader** and has one subclass, **FileReader**
- **InputStreamReader** reads bytes and translates them to Unicode characters using the specified character encoding or the default system encoding
- the class has two constructors

```
InputStreamReader(InputStream in)
InputStreamReader(InputStream in, String enc)
```

- to use an InputStreamReader you must first create an instance of it for a byte input stream. You can then read the stream using any of the **Reader** methods.

OutputStreamWriter

- **OutputStreamWriter** extends **Writer** and has one subclass, **FileWriter**
- **OutputStreamWriter** translates between Unicode characters and bytes using the specified character encoding or the default system encoding
- the class also has two constructors

```
OuputStreamWriter(OutputStream out)
OuputStreamWriter(OutputStream out, String enc)
```

- you use OutputStreamWriter by first creating an instance of it for a byte output stream; you can then write to the stream using an **Writer** methods.

Character Encoding

- Character encodings specify how 8-bit bytes are translated to 16-bit Unicode
- they are represented by Strings which follow the naming standards set by IANA Character Registry
- every implementation of Java is required to support the following sets:

```
US-ASCII      Seven-bit ASCII, a.k.a. ISO646-US, a.k.a. the
              Basic Latin block of the Unicode character
              set
ISO-8859-1
              ISO Latin Alphabet No. 1, a.k.a. ISO-LATIN-1
UTF-8
              Eight-bit Unicode Transformation Format
UTF-16BE
              Sixteen-bit Unicode Transformation Format,
              big-endian byte order
UTF-16LE
              Sixteen-bit Unicode Transformation Format,
              little-endian byte order
UTF-16
              Sixteen-bit Unicode Transformation Format,
              byte order specified by a mandatory initial
              byte-order mark (either order accepted on
              input, big-endian used on output)
```

- specific platforms ie those used in Japan, China, Mid-East, etc, may include other encodings

- the streams are used to read and write data encoded in a character set which is different than the default system encoding
- For example (JPL pg238), to read bytes encoded under ISO 8859-6 for Arabic characters

```
public Reader readArabic(String file) throws IOException {
    InputStream fileIn = new FileInputSgream(file);
    return new InputStreamReader(fileIn, "iso-8859-6");
}
```


	Pkg Overview	Data Streams	Character Streams	Byte Streams	File Class	Readers & Writers
	Filter Streams	Data Input/Output	Reading & Writing Files	Serialization		

The `java.io` Package - Filters

- Filters sit between input and output streams, processing the bytes being transferred
- **FilterInputStream** extends **InputStream** and **FilterOutputStream** extends **OutputStream**
- this means a FilterInputStream, and any of it's subclasses, can take any InputStream as an argument and a FilterOutputStream can take any OutputStream
- this allows the **chaining** of filter streams ie a FilterInputStream can take another FilterInputStream; the original source can be an object that is not a filter stream as long as it isn't reading from another input stream
- Output filters can also be chained, you can have as many filters as you like but the last in the chain must be an OutpuStream
- both methods simply override all their inherited methods, passing all their processing along to the underlying Input or Output stream

FilterInputStream Subclasses

Subclass	Constructors
BufferedInputStream	BufferedInputStream(InputStream in) BufferedInputStream(InputStream in, int size)
DataInputStream	DataInputStream(InputStream in)
LineNumberInputStream	LineNumberInputStream(InputStream in)
PushbackInputStream	PushbackInputStream(InputStream in) PushbackInputStream(InputStream in, int size)

FilterOutputStream Subclasses

Subclass	Constructors
BufferedOutputStream	BufferedOutputStream(OutputStream out) BufferedOutputStream(OutputStream out, int size)
DataOutputStream	DataOutputStream(OutputStream out)
PrintStream	PrintStream(OutputStream out) PrintStream(OutputStream out, boolean autoflush)

- technically, you should use **PrintWriter** when doing character related I/O , **PrintStream** is included for historical reasons. It should only be used with **System.in** as it assumes Latin-1 character encoding.
- most Reader and Writer classes can also act as filters as most of them already have constructors which take another character stream
- to create your own filter streams
  1. Create subclasses of FilterInputStream and FilterOutputStream
  2. Override the read() and write() methods
  3. Override any other methods you might need
  4. Make sure the input and output streams work together

Summary

- If its an input filter, it can take any InputStream object.
- If an output filter, it can take any OutputStream object.
- FilterWriter classes take a Writer object.
- FilterReader classes take a Reader object.

Source code for Examples

- TestFilterWriter.java

The `java.io` Package - Data Input and Output

- DataInputStream and DataOutputStream, like all filters, must be attached to some other stream
- DataInputStream implements DataInput and has one ctor
  `DataInputStream(InputStream in)`
- DataOutputStream implements DataOutput has one ctor
  `DataOutputStream(OutputStream out)`
  and one field written which contains the number of bytes written.
  Note: if this overflows it is set to Integer.MAX_VALUE.
- DataInputStream has specialized **read()** methods, and DataOutputStream, specialized **write()** methods to handle the various primitive types and UTF-8 characters

DataInputStream Methods	DataOutputStream Methods
	write(int oneByte)
read(byte[] buf) read(byte[] buf, int offset, int count)	write(byte[] buf) write(byte[] buf, int offset, int count)
readBoolean()	writeBoolean(boolean b)
readByte()	writeByte(int val)
	writeBytes(String str)
readChar()	writeChar(int val)
	writeChars(String str)
readDouble()	writeDouble(double val)
readFloat()	writeFloat(float val)
readFully(byte[] buf) readFully(byte[] buf, int offset, int count)	
readInt()	writeInt(int val)
readLine()	
readLong()	writeLong(long val)
readShort()	writeShort(int val)
readUnsignedByte()	
readUnsignedShort()	
readUTF()	writeUTF(String str)
skipBytes()	
Items in red are deprecated. All the methods throw IOException	

Source Code for Examples

- DataIOTest.java

| Pkg Overview | Data Streams | Character Streams | Byte Streams | File Class | Readers & Writers |

The `java.io` Package - Reading and Writing Files

- FileStreams have three types of constructors
  1. a constructor that takes a filename as a String
  2. a constructor that takes a File object
  3. a constructor that takes a FileDescriptor object
- when constructors (1) or (2) are used, a new FileDescriptor object is created. This can be accessed by calling getFD()
- a FileDescriptor object represents a system-dependent value that describes an open file
- FileOutputStream has one additonal constructor

```
FileOutputStream(String name, boolean append)
```

- if the file exists, you can set **append** to true to force the write to occur at the end of the file; otherwise, the existing file is overwritten
- FileOutputStream (and FileWriter) have a **flush()** method that forces the underlying buffer to be flushed.

Note
• **flush()** does NOT guarantee that the contents will be written to disk. To guarantee the data is written to disk use the FileDescriptor method **sync()**

- FileReader and FileWriter read and write 16-bit Unicode characters
- FileInputStream and FileOutputStream read and write bytes

Random Access Files (JPL pg 258)

- the RandomAccessFile class is NOT a subclass of InputStream, OutputStream, Reader or Writer; instead it incorporates all their functionaly plus additional methods by implementing the DataInput and DataOutput interfaces.

Note
• You cannot use a RandomAccessFile object where any of the other input and output streams are required.

- the class has two constructors

```
public RandomAccessFile(String name, String mode)
public RandomAccessFile(File file, String mode)
```

- the **mode** argument must be either **"r"** or **"rw"** to indicate if the file is to be opened for reading only or reading and writing
- if the file is opened for writing and it does not exist; it will be created
- as with the other File streams, a FileDescriptor object is created when the file is opened
- the class allows you to set a read/write pointer to any position in the file
- key methods are:

```
public long getFilePointer() throws IOException
public void seek(long pos) throws IOException
public void skipBytes(int count) throws IOException
public long length() throws IOException
```

Source code examples

- Read a file using FileInputStream
- Write to a file using FileWriter
- Copy a file (jung.txt) to another file using FileReader and FileWriter
- An example, CopyBytes, to do the same thing using FileInputStream and FileOutputStream
- Test using a RandomAccessFile

	Pkg Overview	Data Streams	Character Streams	Byte Streams	File Class	Readers & Writers
	Filter Streams	Data Input/Output	Reading & Writing Files	Serialization		

The bigger the crowd the more negligible the individual becomes. But if the individual, overwhelmed by the sense of his own puniness and impotence, should feel that his life has lost its meaning--which, after all, is not identical with the public welfare and higher standards of living--then he is already on the road to State slavery and, without knowing or wanting it, has become its proselyte.

The man who looks only outside and quails before the big battalions has no resource with which to combat the evidence of his senses and his reason. But that is just what is happening today: we are all fascinated and overawed by statistical truths and large numbers and are daily apprised of the nullity and futility of the individual personality, since it is not represented and personified by any mass organization.

C.G. Jung, The Undiscovered Self, New American Library, 1958

The `java.io` Package - Serialization

Serialization (JCL1)

- *serialization* is the process of converting an **object** to a stream of bytes in such a manner that the original **object** can be rebuilt (lets you write an object to a file or other data container)
- an object can be serialized only if it's class implements the Serializable or Externalizable interface; it's superclass must have a no-arg default constructor or be Serializable itself
- a classes **serializable fields** are all of its nontransient and nonstatic fields; this applies to all public, protected, package and private fields (JCL1)

Note
Only the accessible fields of the superclasses are serialized

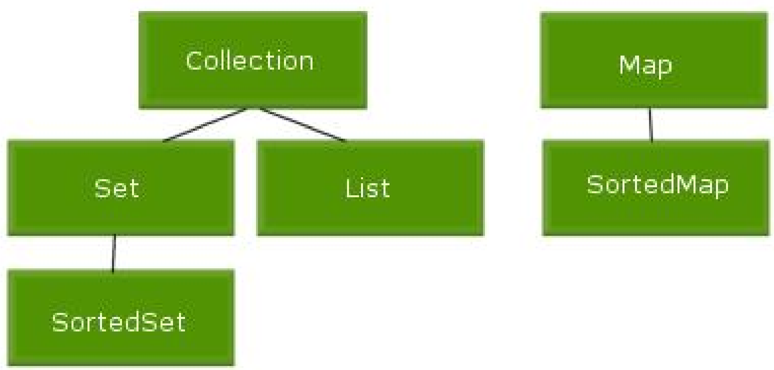
- the serialized fields are written out using **ObjectOutputStream.defaultWriteObject()** and read back using **ObjectOutputStream.defaultReadObject()**
- all the objects referred to directly or indirectly are also serialized
- if a field contains an object that is not serializable, a **NotSerializableException** is thrown
- *deserialization* is the process of restoring a serialized object to a copy of the original object
- all Java primitive types, arrays, Strings and objects can be serialized/deserialized
- primitive types can be serialized using DataInputStream Interface and deserialized using DataOutputStream Interface

	Pkg Overview	Data Streams	Character Streams	Byte Streams	File Class	Readers & Writers
	Filter Streams	Data Input/Output	Reading & Writing Files	Serialization		

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

The `java.util` Package - The Collections Framework

- a *collection* is a *container* or *object* that groups multiple objects into a single unit
- a *Collections Framework* provides a unified system for organizing and handling collections and is based on four elements:
  1. Interfaces that characterize common collection types
  2. Abstract Classes which can be used as a starting point for custom collections and which are extended by the JDK implementation classes
  3. Classes which provide implementations of the Interfaces
  4. Algorithms that provide behaviours commonly required when using collections ie search, sort, iterate, etc.
- the Collection Framework in Java has six core collection Interfaces:



- the *Collection Framework* also provides an interface for traversing collections: Iterator and it's subinterface ListIterator
- the Iterator interface should be used in preference to the earlier Enumeration interface

	Collections Framework	Collection	Abstract Classes	Iterator	List	

- Home
- SCJP2 Study Notes
- Language Fundamentals
- Operators and Assignments
- Flow Control and Exceptions
- Declarations and Access Control
- Garbage Collection
- Overloading and Overriding
- Threads
- The java.lang Package
- The java.util Package
- The java.awt Package
- The java.io Package
- References
- Miscellaneous Notes
- Tips & Traps
- Mock Exams
- Case Studies
- SCJA Notes
- SCJD Notes
- Projects
- Favourite Links
- About
- Feedback

The `java.util` Package - The Collection Interface

- this is the root interface for the collection heirarchy
- it is not directly implemented by an SDK class; instead they implement the subinterfaces List or Set
- it is typically used to manipulate and pass collections around in a generic manner
- classes which implement Collection or one of it's subinterfaces must provide two constructors
    1. a default, no-argument constructor, which creates an empty collection, and
    2. a constructor which takes a Collection as an argument and creates a new collection with the same elements as the specified collection

Query Methods	
contains(Object o)	returns true if the collection contains the specified element
isEmpty()	returns true if the collection has no elements
iterator()	returns an Iterator object. There is no guarantee as to the order of the returned elements unless the collection is an instance of a class that guarantees the order.
size()	returns the number of elements in the collection or Integer.MAX_VALUE if the collection equals or exceeds Integer.MAX_VALUE
toArray()	returns the collection elements as an array. If the collection class guarantees an order, the array elements are in the guaranteed order.
toArray(Object a[])	returns all the elements in the collection whose type is that of the array type. If the collection does not fit in the array, a new array of the same type is returned. If the array is larger than the collection, the array element after the last collection element is set to null
Bulk Methods	
containsAll(Collection c)	returns true if the collection contains the all the elements in the specified collection
addAll(Collection c)	adds all the elements in the specified collection to this collection
clear()	removes all the elements in the collection
removeAll(Collection c)	removes all the this collections elements that are in the specified collection.
retainAll(Collection c)	retains all the elements in this collection that are contained in the specified collection
Modification Methods	

add(Object o)	adds an element to the collection. Returns false if the element is not added as the collection class guarantees no duplicates.
remove(Object o)	removes the specified object from the collection, if it exists.
equals() and hasCode()	
equals(Object o)	programmers may override the Object.equals() method to implement collection specific comparisons eg "value" comparison vs "reference" comparison
hashCode()	programmers overriding equals() must also override Object.hashCode()

Tips

- any SDK class which implements Collection or any of it's subinterfaces will contain the two required constructors *CollectionName()* and *CollectionName(Collection c)*

Collections Framework | Collection | Abstract Classes | Iterator | List

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

The `java.util` Package - The List Interface

- provide skeletal implementations that can be used as the basis for building custom collection classes
- available classes are:

    1. AbstractCollection
    2. AbstractList
    3. AbstractMap
    4. AbstractSequential
    5. AbstractSet

- JSK implementations extend the applicable Abstract class and implement the appropriate Interface

	Collections Framework	Collection	Abstract Classes	Iterator	List	

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

The `java.util` Package - The Iterator Interface

- used to sequentially access collection elements
- element order depends on the collection ie List elements are presented as they appear in the List, Set elements can be in any order

Iterator Methods	
hasNext()	returns true if the iteration has more elements
next()	returns the next element in the iteration
remove()	removes the most recently retrieved element from the underlying collection

- has one subinterface, ListIterator, which allows a programmer to traverse a List in either direction and make modifications to the underlying List

java.util.ListIterator Methods	
Query Methods	
hasNext()	returns true if there are more elements in a forward direction
hasPrevious()	returns true if there are more elements in a backward direction
next()	returns the next element in the List
nextIndex()	returns the index of the next element in the list, or, the size of the list if there are no more elements
previous()	returns the previous element in the List
previousIndex()	returns the index of the previous element in the list. If positioned at the first element, returns -1
Modification Methods	
add(Object obj)	inserts the new object immeadiately before the element which would be returned by next().
remove()	removes the last element in the List retrieved by a next() or previous() operation. Can only be made once after a next() or previous() operation and cannot be made if there has been an intervening add().
set(Object obj)	replaces the last element in the List retrieved by a next() or previous() operation; there can be no intervening call to add() or remove().

Collections Framework | Collection | Abstract Classes | Iterator | List

The `java.util` Package - The List Interface

- a List is a collection whose elements can be accessed by an index
- the indices are zero-based
- a list has methods for inserting and removing elements
- a list can contain **duplicate** elements
- a List provides a special ListIterator which allows you to move backwards and forwards through the elements
- there are three basic ways in which a List can be modified:
  1. add an element
  2. remove an element
  3. replace an element
- a list can support any or none of the above; attempts to modify a list that does not support the above will result in an UnsupportedOperationException
- there is no way to append Lists unless you provide your own method

java.util Implementations of List	
ArrayList	extends AbstractList implements List, Cloneable, Serializable Elements are ordered. Internally uses an array to store elements. Index access is quick, while adding and removing elements, except at the end of the array, is expensive.
LinkedList	extends AbstractSequentialList implements List, Cloneable, Serializable Elements are ordered. Internally uses a doubly linked list to store elements. Adding and removing elements involves updating two links; index access is slow as the entire list must be traversed. LinkedList retains a reference to both the first and last elements; retrieving the first or last element is efficient.
Vector	extends AbstractList implements List, Cloneable, Serializable Older class that was modified in JDK 1.2 to implement List. An expansible array. The vector will grow automatically to take new objects. You can also shrink a Vector. Otherwise, manipulated the same as an array. May contain null elements. All methods are synchronized

List Methods

Positional Methods	
get(int index)	returns the element at the specified position
set(int index, Object element)	replaces the element at the specified position with the given object
add(int index, Object element)	inserts the specified element at the specified position, shifting all the elements and adds one to their index values
remove(int index)	removes the element at the specified position, shifiting all the elements and subtracting one from their indices
Search Methods	
indexOf(Object o)	returns the index of the first occurence of the specified element or -1 if it is not found
lastIndexOf(Object o)	returns the index of the last occurence of the specified element or -1 if it is not found
List Iterator	
listIterator()	returns a list iterator of the elements in their proper sequence
listIterator(int index)	returns a list iterator of elements starting at the specified index
subList(int fromIndex, int toIndex)	returns the portion of the list between the specified indices exclusive of the toIndex element

	Collections Framework	Collection	Abstract Classes	Iterator	List	

The `java.lang` Package Certification - Main Classes

- the java.lang Package contains classes that are fundamental to the Java programming language
- it is always implicitly imported
- the most important classes are Object and Class

Object

- the Object class is at the root of the class heirarchy, all other classes inherit it's methods

protected Object clone() throws CloneNotSupportedException
: returns an identical copy of an object. The object must implement the Cloneable interface

public boolean equals(Object obj)
: returns true if obj is the same object as the referenced object

protected void finalize() throws Throwable
: called by the garbage collector prior to collecting the object

public final Class getClass()
: returns the runtime class of an object

public int hashCode()
: returns a distinct integer representing a unique object; supports hash tables

public final void notify()
: wakes up a single thread waiting on the object's monitor

public final void notifyAll()
: wakes up all threads waiting on the object's monitor

public String toString()
: returns a string representation of the object

public final void wait() throws InterruptedException,
public final void wait(long timeout) throws InterruptedException,
public fianl void wait(long timeout, int nanos) throws InterruptedException
: causes the current thread to wait until another thread invokes notify() or notifyAll() for this object, or, the specified time elaspses

Class

- the Class class was introduced in JDK 1.2
- instances of the Class class represent classes and interfaces in a running Java application
- also represents arrays, primitive types and void, all of which are Class instances at runtime
- objects of the Class class are automatically created as classes are loaded by the JVM; they are known as class descriptors
- provides over 30 methods which can be used to obtain information on a running class
- some of the more useful methods are: getName(), toString(), getSuperclass(), isInterface(), newInstance()

Other classes

- **Wrapper classes** used to represent primitive types as Objects: Boolean, Byte, Short, Character, Integer, Float, Long and Double
- **Math** class provides commonly used mathematical functions ie cos, sine, tan
- **String** and **StringBuffer** classes provide commonly used operations on character strings
- **System operation** classes: ClassLoader, SecurityManager, Runtime, Process and System which manage the dynamic loading of classes, creation of external processes, security, and host inquiries ie time of day
- **Package** class is new to JDK 1.2. Provides methods for obtaining package version information stored in the manifest of jar files. Useful methods include: getPackage(), getAllPackages(), which provide package objects that are known to the class loader, and isCompatibleWith() which is used to determine wether a package is comparable to a particular version.
- all the **Exception** and **Error** classes, including **Throwable**

Interfaces

- Cloneable. Contains no methods. Used to differentiate between objects that are cloneable and non-cloneable.
- Comparable, new in JDK 1.2. Defines the compareTo() method. Objects implementing this interface can be compared and sorted.
- Runnable. Defines the run() method which is invoked when a thread is activated.

	Main Classes	Wrapper Classes	Math Class	String Immutability	String Class	StringBuffer Class

The `java.lang` Package Certification - Wrapper Classes

- one for each primitive type: Boolean, Byte, Character, Double, Float, Integer, Long, and Short
- Byte, Double, Float, Integer and Short extend the abstract Number class
- all are **public final** ie cannot be extended
- get around limitations of primitive types
- allow objects to be created from primitive types
- all the classes have two constructor forms
  - a constructor that takes the primitive type and creates an object eg Character(char), Integer(int)
  - a constructor that converts a String into an object eg Integer("1"). Throws a NumberFormatException if the String cannot be converted to a number

Note
• The Character class does not have a constructor that takes a String argument

- all, except Character, have a valueOf(String s) method which is equivalent to new Type(String s)
- all have a typeValue() method which returns the value of the object as it's primitive type. These are all abstract methods defined in Number and overridden in each class
  - public byte byteValue()
  - public short shortValue()
  - public int intValue()
  - public long longValue()
  - public float floatValue()
  - public double doubleValue()
- all the classes override equals(), hashCode() and toString() in Object
  - equals() returns true if the values of the compared objects are the same
  - hashCode() returns the same hashcode for objects of the same type having the same value
  - toString() returns the string representation of the objects value
- all have a public static final TYPE field which is the Class object for that primitive type
- all have two static fields MIN_VALUE and MAX_VALUE for the minimum and maximum values that can be held by the type

Void

- there is also a wrapper class for Void which cannot be instantiated.

Note
• The constructors and methods described above do NOT exist for the Void class although it does have the TYPE field.

Character

- contains two methods for returning the numeric value of a character in the various number systems

    - public static int digit(char ch, int radix)
    - public static int getNumber(char ch)
- and one method to return the character value of a number
    - public static char forDigit(int digit, int radix)
- has two case conversion methods
    - public static char toLowerCase(char ch)
    - public static char toUpperCase(char ch)
- also contains a variety of other methods to test wether a character is of a specific type eg isLetter(), isDefined(), isSpaceChar(), etc
- getType() returns an int that defines a character's Unicode type

Integer, Short, Byte and Long

- all have *parseType* methods eg parseInt(), parseShort, etc that take a String and parse it into the appropriate type
- the Integer and Long classes also have the static methods toBinaryString(), toOctalString() and toHexString() which take an integer value and convert it to the appropriate String representation

Float and Double

- both classes have static fields which define POSITIVE_INFINITY, NEGATIVE_INFINITY, and NaN
- and the following methods to test a value
    - public boolean isNan()
    - public static boolean isNaN(type value)
    - public boolean isInfinite()
    - public static boolean isInfinite(type value)
- Float also has a constructor that takes a double value
- both classes have methods to convert a value into a bit pattern or vice versa
    - public static int floatToIntBits(float value)
    - public static float intBitsToFloat(int bits)
    - public static long doubleToLongBits(double value)
    - public static double longBitsToDouble(long bits)

	Main Classes	Wrapper Classes	Math Class	String Immutability	String Class	StringBuffer Class

- Home
- SCJP2 Study Notes
- Language Fundamentals
- Operators and Assignments
- Flow Control and Exceptions
- Declarations and Access Control
- Garbage Collection
- Overloading and Overriding
- Threads
- The java.lang Package
- The java.util Package
- The java.awt Package
- The java.io Package
- References
- Miscellaneous Notes
- Tips & Traps
- Mock Exams
- Case Studies
- SCJA Notes
- SCJD Notes
- Projects
- Favourite Links
- About
- Feedback

The `java.lang` Package Certification - Math Class

- contains static constants **E** and **PI**

```
E:    2.718281828459045
PI:   3.141592653589793
```

- contains methods for common mathematical operations ie abs, sin, exp, round, etc.
- all methods are **static**
- the Math class cannot be instantiated
- methods involving angles use radians vs degrees and minutes
- all methods, except **round()**, return a **double**
- all methods take at least one **double** as an argument, except **random** which takes no arguments
- the following methods are overloaded to return and handle int, long and float
  - static *type* abs(*type* a)
  - static *type* max(*type* a, *type* b)
  - static *type* min(*type* a, *type* b)

IEEEremainder

- calculates the remainder as defined by IEEE-754
- the remainder operator, %, makes values symmetric around zero ie negative and positive values return corresponding remainders

```
 7 % 2.5:   2.0
-7 % 2.5:  -2.0
```

- Math.IEEEremainder keeps resulting values y units apart

```
Math.IEEEremainder( 7, 2.5):         -0.5
Math.IEEEremainder(-7, 2.5):          0.5
```

abs()

- returns the absolute or **positive** value of the argument

```
Math.abs(1234.59):                   1234.59
Math.abs(-0.0):                      0.0
Math.abs(Float.NEGATIVE_INFINITY):   Infinity
Math.abs(Float.NaN):                 NaN
```

- EXCEPT if the value is equal to Integer.MIN_VALUE, in which case, it returns the value as a negative

```
Math.abs(Integer.MIN_VALUE):         -2147483648
```

ceil()

- returns the smallest double value not less than the argument and equal to an integer (counts up)

- if the argument is already an integer, returns the argument
- if the argument is NaN or infinity, returns the argument
- if the argument is between -1.0 and 0, returns 0

```
Math.ceil( 9.01):       10.0    // counts up (away from zero)
Math.ceil(-9.01):       -9.0    // counts up (towards zero)
Math.ceil(10):          10.0
Math.ceil(-0.03):       -0.0
Math.ceil(Double.NaN):  NaN
```

floor()

- returns the largest double value not greater than the argument and equal to an integer (counts down)
- if the argument is an integer, returns the argument
- if the argument is NaN, infinity, negative or positive zero, returns the argument
- if the argument is between -0 and 0, returns -0

```
Math.floor( 9.01):       9.0     // counts down (towards zero)
Math.floor(-9.01):       -10.0   // counts down (away from zero)
Math.floor(10):          10.0
Math.floor(-0.03):       -1.0
Math.floor(Double.NaN):  NaN
```

min() and max()

- min() returns the smallest of two values
- max() returns the largest of two values

```
Math.min(-1.5, 1.5):     -1.5
Math.max(-1.5, 1.5):     1.5
Math.min(0.0, -0.0):     -0.0   // zeros are not equivalent
Math.min(Float.NaN,
         Float.POSITIVE_INFINITY));   NaN
```

random()

- returns a pseudo-random positive double number between 0.0 and 1.0
- if you want to seed the number or generate random numbers in different ranges use the java.util.Random class

```
Math.random():           0.2379468138972043
```

round()

- has two versions
  - public static long round(double a)
  - public static int round(float a)
- only method that does not return a **double**
- adds 0.5 to the argument and returns the closest int
- if the argument is not a number, returns zero
- if the argument is a negative infinity or less than the MIN_VALUE for the type, returns the MIN_VALUE
- if the argument is a positive infinity or greater than the MAX_VALUE for the type, returns the MAX_VALUE

```
Math.round( 1.5):                          2
Math.round(-1.5):                         -1
Math.round(Float.NaN):                     0
Math.round(Float.NEGATIVE_INFINITY):    -2147483648
Math.round(Double.POSITIVE_INFINITY):    9223372036854775807
Math.round(Float.MAX_VALUE):             2147483647
                            (Float.MAX_VALUE is 3.4028235E38)
```

Note
• If the value is Float.MAX_VALUE the round method returns Integer.MAX_VALUE

rint()

- rounds to the closest integer
- if integers are equidistant, favours the even integer

```
Math.rint( 5.5):         6.0
Math.rint(-5.5):        -6.0
Math.rint( 5.49):        5.0
Math.rint(-5.49):       -5.0
```

sqrt()

- returns the positive square root of a number
- returns NaN if argument is negative

```
Math.sqrt(45):          6.708203932499369
Math.sqrt(-45):         NaN
```

pow(double a, double b)

- returns the first argument raised to the power of the second argument

```
Math.pow(2,2):          4.0
```

Trigometric functions

- all results are returned in **radians**
- there are 2 * PI degrees in a circle, ie 2/PI = 90 degrees

sin(double a)

 if the result is NaN or infinity, returns NaN
 if the result is negative zero, returns -0.0

cos(double a)

 if the result is NaN or infinity, returns NaN

tan(double a)

 if the result is NaN or infinity, returns NaN
 if the result is negative zero, returns -0.0

asin(double a)

 returns a value between **-PI/2** and **PI/2**
 if the result is NaN or absolute value is greater than 1, returns NaN
 if the result is negative zero, returns -0.0

acos(double a)

 returns a value between **0.0** and **PI**
 if the result is NaN or absolute value is greater than 1, returns NaN

atan(double a)

returns a value between **-PI/2** and **PI/2**
if the result is NaN, returns NaN
if the result is negative zero, returns -0.0

- atan2(double a, double b) converts rectangular co-ordinates to polar co-ordinates
- has two additional methods, new in JDK 1.2, to convert between radians and degrees
  - double toRadians(double angdeg)
  - double toDegrees(double angdeg)

```
Math.sin(90):               0.8939966636005579
Math.cos(90):               -0.4480736161291701
Math.tan(90):               -1.995200412208242
Math.asin(-0):              0.0
Math.acos(-0):              1.5707963267948966
Math.atan(90):              1.5596856728972892

Math.toRadians(90)          1.5707963267948966
Math.toDegrees(Math.PI/2):  90.0
```

Logarithms

- two functions to handle logs
  - double log(double a)
  - double exp(double a)
- log() returns the natural logarithm of the argument
- if the argument is less than zero, returns NaN
- if the argument is positive infinity, returns positive infinity
- if the argument is -0.0 or 0.0, returns negative infinity

```
Math.log(10):               2.302585092994046
Math.log(-10):              NaN
Math.log(0.0):              -Infinity
```

- exp() returns e to the power of the argument
- if the argument is NaN, returns NaN
- if the argument is positive infinity, returns positive infinity
- if the argument is negative infinity, returns positive zero

```
Math.exp(5):                         148.4131591025766
Math.exp(Float.NaN):                 NaN
Math.exp(Float.POSITIVE_INFINITY):   Infinity
Math.exp(Float.NEGATIVE_INFINITY):   0.0
```

Example Code

- TestMath.java

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

The `java.lang` Package Certification - String Immutability

- String objects are **read-only** or **immutable** ie the **contents** of a String object **never change**

```
String str = "Hello";
str = "Goodbye";
```

- in the above example, the second assignment of "Goodbye" to String, what actually happens is that a new string "Goodbye" is created and the **object reference** of the new string is stored in the variable str
- operations that seem to modify a String object actually create **new** read-only String objects; leaving the original object unchanged
- the StringBuffer class provides **mutable** or **flexible** string handling

Also see

String literals

Main Classes | Wrapper Classes | Math Class | String Immutability | String Class | StringBuffer Class

Language Fundamentals - String Literals

- **String literals** are enclosed in double quotes

```
"This is a string literal."
```

- A **string constant expression** occurs when two or more string literals are concatenated

```
"This is " + "a string " + "constant expression."
```

- Character escape codes can be used in String literals

```
"A line with a carriage return \r"
```

!!! Warning !!!
You cannot use the character literals **\u000a** (newline) or **\u000d** (carriage return) in String literals as they will be interpreted as *LineTerminators*, not as input characters (JLS §3.10.5) `"A line with unicode carriage return character \u000d"`

- If you use octal values in Strings to represent characters be sure to use a zero prefix (JPL pg33)
  Note: the zero prefix is not required for octal values in **char** literals

```
"\0116"      octal value equivalent to escape
               char \t followed by 6 "\t6"
"\116"       interpreted as letter N
```

Each String literal is a reference to an object of class String. String literals or strings that are the values of constant expressions, are **interned** so as to share unique instances.

public String.intern() (JSK 1.3)

"Returns a canonical representation for the string object.

A pool of strings, initially empty, is maintained privately by the class String.
When the intern method is invoked, if the pool already contains a string equal to this String object as determined by the equals(Object) method, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.

It follows that for any two strings s and t, s.intern() == t.intern() is true if and only if s.equals(t) is true.

All literal strings and string-valued constant expressions are interned."

Output from (JLS § 3.10.5) example code:

- the JLS gives example code using literals in the following classes:
  - class test
  - class Other (in the same java file as class test)
  - class other.Other (in a different package)

- the code gives the following output:

```
String variables initialized as:
         String hello = "Hello"
         String    lo = "lo"

     (1) hello == "Hello"                true
     (2) Other.hello == hello            true
     (3) other.Other.hello == hello      true
     (4) hello == ("Hel"+"lo")           true
     (5) hello == ("Hel"+lo).intern()    true
     (6) hello == ("Hel" + lo)           false
```

- literal strings will represent the **same reference** if they are created
  1. in the same class and in the same package
  2. in different classes within the same package
  3. in different classes in different packages
  4. using constant expressions computed at compile time
  5. by explicitly using the **intern()** method and the resulting string is already in the string pool
- literal strings will represent **different references** if they are newly created at runtime (Line 6)

Summary
• if String objects having the same data are created using a **constant expression**, a **string literal**, a reference to an existing string, or by explicitly using the intern() method, their references will be the **same** • if String objects having the same data are created explicitly with the **new** operator or their values are computed at runtime, their references will be **different**

```
 String str1 = "Lions and Tigers and Bears!";
 String str2 = "Lions and Tigers and Bears!";
 String str3 = str2;
 String str4 = new String("Lions and Tigers and Bears!");
 String str5 = " Oh my!";
 String str6 = "Lions and Tigers and Bears! Oh my!";
 String str7 = str1 + str5;
 String str8 = (str1 +" Oh my!").intern();

Comparison output:
str1 == str2  -> true     // the str2 literal existed ("interned")
str1 == str3  -> true     // hold the same reference
str1 == str4  -> false    // str4 explicitly created
str2 == str3  -> true     // hold the same reference
str2 == str4  -> false    // str4 explicitly created
str3 == str4  -> false    // str4 explicitly created
str6 == str7  -> false    // str7 computed at runtime
str6 == str8  -> true     // explicit use of intern() at runtime
```

JSK 1.3 for the java.lang.String class states:
"Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared."

In other words, because the compiler knows the strings original value cannot be changed once it's created it can safely use existing data and avoid cluttering up memory with duplicates.

Example code

- TestStringLiteral.java

Traps

- using == operator to compare contents of two string reference variables pointing to different String objects

	Source	Package	Import	Class	Interface	Constructors
	Methods	main()	Identifiers	Keywords	Defaults	Arrays
	Primitives	# Literals	char Literal	String Literals	Class Literals	

Language Fundamentals - Source Files

A Java source code file or **compilation unit** has three basic parts, each of which is optional (JLS §7.3):

- A **package** declaration
- **import** declarations
- top-level **class** and **interface** declarations

Package declaration

- if used, it must be the first non-comment statement in the source code file
- you can not declare more than one
- syntax: package packageName;

Import declarations

- if used, must be the first non-comment statement directly following the package declaration.
- you can use as many import statements as you want
- if no package statement appears in the source code file, the import statement must be the first non-comment statement in the file

top-level class and interface declarations

- A **top-level** class or interface is defined as any class or interface whose declaration is not contained within the body of any other class or interface declaration. (JLS §8 and §9).
- you can declare multiple classes and interfaces within a file with the following caveats:
  - The Sun SDK allows **one** and only one **public** class or interface within a source code file.
  - The filename must **exactly match** the name of the public class or interface declared in the file and have the **.java** extension

Non-public classes may have main() methods. If they have no access modifier (package access) they may still be run from the command-line using the classname.

Example Code

- TestPkgImport.java

Tips

- an empty source file will compile without error
- if a `.java` file does not contain a *public* class or interface it can have any name

Traps

- code with *package* or *import* declarations given in wrong order
- more than one *package* declaration
- file with more than one *public* class or interface declaration

- *filename.java* does not match *public* class name as declared within the file

	Source	Package	Import	Class	Interface	Constructors
	Methods	main()	Identifiers	Keywords	Defaults	Arrays
	Primitives	# Literals	char Literal	String Literals	Class Literals	

Language Fundamentals - Package Declarations

Syntax

```
package packageName;
```

- packages provide a **naming context** and an **organizational structure** for Java compilation units
- package names are hierarchical with component names separated by dots (JPL pg 25)
- the standard convention for package naming is to use the reversed internet domain name of whoever's creating the package. For example:

```
    com.sun.java.awt        // Sun packages
    com.ibm.utils           // IBM packages
    com.acme.tools          // Acme company packages
```

- the package naming structure directly maps to a directory structure. For example, if you were developing the Acme company packages your compilation units for the com.acme.tools package would be in:

```
directory.............. com
  subdirectory..........acme
    subdirectory.........tools
```

- the Java compiler uses a combination of the CLASSPATH and package name to locate the source file
- host systems may store packages in databases (JLS §7.2.2)
- if used, it must be the first statement in the source code file
- you can not declare more than one
- the package naming structure is for ease of organization only, it does not confer a special relationship (JLS §7.1) ie

```
There is no special relationship between the packages:

        com.acme.tools,         and
        com.acme.utils

The fact that they share a common subpackage, acme,
has no meaning in terms of a types scope.
```

Unnamed packages (JLS §7.4.2)

- if no package declaration is found, the class or interface is made part of an **unnamed package**
- every implementation of Java must provide for at least **one** unnamed package
- most systems allow for one unnamed package per directory

Language Fundamentals - Import Declarations

Syntax

```
import packageName.*;                // type-import-on-demand
import packageName.ClassName;        // single-type-import
import packageName.InterfaceName;    // single-type-import
```

- the import statement is used to reference classes and interfaces declared in other packages
- the **type-import-on-demand** import statement will cause the package to be searched when a type is declared for a class which has not been declared within the source file
- duplicate type-import-on-demand statements are ignored (JLS §7.5.2)
- the **java.lang** package is automatically imported in every compilation unit, it does not have to be specifically imported
- you can access classes and interfaces from other packages without first importing them but you must use their **fully qualified names** For example:

```
If you import the java.awt.Button class by using:
    import java.awt.*;      ,or,
    import java.awt.Button;

You can create a Button by coding:

    Button myButton = new Button();

Without the package import you'd need to code:

    java.awt.Button myButton = new  java.awt.Button();
```

- imported types are available to all classes and interfaces within the same compilation unit (JLS §7.5)
- it is legal to import a single-type and a package having the same names (JLS §7.5.4)

```
i.e.
    import java.util.Vector;
    import Vector.pest;         // no compile error
```

Also see

- Sun Tech Tip: Using Import Declarations

Tips

- a single-type import will take precedence over an import-on-demand
- import-on-demand types do not increase the size of the compiled code ie only the types actually used are added to the code
- I've read that while import-on-demand adds no overhead to the compiled code, they can slow down the speed of the compile; however, Peter van der Linden, in *Just Java 2, 4th Edition* says it ain't so and my guess is he knows ... he's a kernel programmer for Sun

Traps

- single-type imports for two classes in different packages but with the same simple name
- single-type import with the same simple name as a class defined in the source file
- attempting to import a package vs a type ie import java.util vs import java.util.*

	Source	Package	Import	Class	Interface	Constructors
	Methods	main()	Identifiers	Keywords	Defaults	Arrays
	Primitives	# Literals	char Literal	String Literals	Class Literals	

Language Fundamentals - Class Declarations

Syntax (JJ pg 137)

```
modifiers class ClassName extendsClause implementsClause {
    // Class body
}
```

The *modifiers*, *extendsClause* and *implementsClause* are all optional.

Modifiers

```
public protected private
abstract static final strictfp
```

- if two or more modifiers are used in a declaration it is customary, but not required, to show them in the order given (JLS 8.1.1)
- **no modifiers** are allowed in **Anonymous** class declarations (JJ pg 147)
- A class may **not** be both **final** and **abstract** as an abstract class implies extension
- package access (no access modifier declared) is also referred to as **friendly access**
- a compile error occurs if the same modifier appears more than once in a declaration (JLS §8.1.1)

extendsClause (JJ pg 137)

- consists of the **extends** keyword followed by the name of the class being extended
- the extended class is referred to as the **parent** or **superclass**
- multiple extends are illegal ie a class may have only **one** superclass
- if no extends clause is used, the class automatically inherits from the **java.lang.Object** class
- a compile error occurs if a **final** class appears in the extends clause (JLS §8.1.1.2)
- an **Anonymous** class **cannot** have an extends clause (JPL pg74)

implementsClause (JJ pg 137)

- identifies interfaces implemented by the class
- consists of the **implements** keyword followed by a comma seperated list of the names of the interfaces to be implemented

```
class X implements interfaceA, interfaceB, ... { }
```

- a class must provide a method implementation (execution code) for every method declared in or inherited by the interface
- if an interface is **not** provided in the implements clause, the class does not implement the interface **even if** it provides an implementation for every method declared in the interface

Class body (JJ pg 138)

- the class body declares **members** (field variables and methods), constructors and initializers
- class members may also be inner classes or interfaces

Traps

- class attempting to extend more than one other class
- class declared both **final** and **abstract**

	Source	Package	Import	Class	Interface	Constructors
	Methods	main()	Identifiers	Keywords	Defaults	Arrays
	Primitives	# Literals	char Literal	String Literals	Class Literals	

Language Fundamentals - Interface Declarations

Syntax (JJ pg142)

```
modifiers interface InterfaceName extendsClause {
    // Interface body
 }
```

The *modifiers* and *extendsClause* are optional.

A compile time error occurs if an interface has a simple name the same as any of it's enclosing classes or interfaces (JLS §9.1)

Modifiers (JLS §9.1.1)

```
public protected private
abstract static strictfp
```

Note

• **top-level** interfaces may only be declared **public**
`private interface A {}          // compile error`
`protected interface B{}         // compile error`

- inner interfaces may be declared private and protected BUT only if they are defined in a class

```
public interface A {
    private interface B {}          // compile error
    protected interface C {}        // compile error
}

public class A {
    private interface B {}          // compiles OK
    protected interface C {}        // compile OK

}
```

- a compile error occurs if the same modifier appears more than once in an interface declaration (JLS §9.1.1)
- every interface is implicitly **abstract**; the modifier is obsolete and should not be used in new programs (JLS §9.1.1.1)

extendsClause

- consists of the **extends** keyword followed by a comma separated list of the interfaces being extended.

Note

- Classes are based on **single-inheritance**, they can only extend one class.
- Interfaces are allowed **multiple-inheritance**, they can extend more than one interface.

```
interface InterfaceA extends
  interfaceX, interfaceY, ... {}
```

Interface body

- an interface body may contain constant declarations, abstract method declarations, inner classes and inner interfaces
- **fields** in an interface are implicitly **static** and **final** ie they MUST be **constants** (JLS§9.3)
- **methods** in an interface are implicitly **abstract** and **public**; they CANNOT be **static** (JLS§9.4)
- **methods** cannot be declared **strictfp, native** or **synchronized** (JLS§9.4)
- **member classes** declared in an interface are implicitly **public** and **static** (JLS§9.5)

Also see

- Sun Tutorial: Interfaces and packages
- Tech Tip: Abstract classes vs interfaces

Code Examples

- TestInterfaceModifiers.java
- TestInterfaceInClass.java

Traps

- an interface method declared as **native** or **synchronized**
- an interface method declared as **static**

	Source	Package	Import	Class	Interface	Constructors
	Methods	main()	Identifiers	Keywords	Defaults	Arrays
	Primitives	# Literals	char Literal	String Literals	Class Literals	

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Language Fundamentals - Constructor Declarations

Syntax (JLS §8.8)

```
modifiers ClassName(arguments) throwsClause {
    // Constructor body
}
```

The *modifers*, *ClassName*, *arguments*, and *throwsClause* are optional.

[**Note:** all are optional in the sense that your class does NOT have to declare a constructor (see following on default constructors);however, if you do include a constructor *modifiers*, *arguments* and the *throws clause* are optional.]

- a constructor **can use** the access modifiers **public, protected** or **private** or have no access modifier (package access)
- a constructor **can not** use the modifiers **abstract, static, final, native, synchronized** or **strictfp** (JLS §8.8.3)
- constructors are not considered *class members*, they are **not inherited**
- if a class constructor is not declared, a default constructor is supplied by the compiler

```
modifiers ClassName() {
    super();
}
```

- the default constructor has the same access modifier as the class itself, either: public, protected, private or package (no modifier)
- to prevent a class from being instantiated outside the class declaration you can create a **private** constructor.

Note

A method having the same name as the class is not treated as a constructor
 	
`public void MyClassName() {}    // not a constructor`
`public MyClassName() {}         // constructor`
 	
A constructor cannot have a return type.

Also see

Sun Tech Tip: Default Constructors

Tips

- a constructor body can include a return statement providing no value is returned

Traps

- subclass with default constructor when the superclass does not have a no-args constructor or it's no-arg constructor has a throws clause

- constructor declared with a return type

	Source	Package	Import	Class	Interface	Constructors
	Methods	main()	Identifiers	Keywords	Defaults	Arrays
	Primitives	# Literals	char Literal	String Literals	Class Literals	

Language Fundamentals - Method Declarations

Syntax (JJ pg88)

```
modifiers returnValue methodName(parameterList)
   throwsClause
{
    // Method body
}
```

The *modifiers* and *throwsClause* are optional.

Modifiers

- legal access modifiers: public, protected, private or package (none declared)
- legal special modifiers: abstract, final, native, static, or synchronized
- a **static** method is referred to as a **class method**
- a **non-static** method is referred to as an **instance method**
- the access modifier of an overriding method must provide at least as much access as the method being overridden. (JLS §8.4.6.3)

```
Original Method Access  Overriding method must be
 public                   public
 protected                public or protected
 package                  package, public or protected
```

returnValue (JLS §8.4.5)

- legal return types: void, any primitive data type, an Object reference or Array type
- if **void** is used, the method may not use a return statement with an expression

```
return;             // legal
return(x);          // illegal
```

- if a primitive data type is used, the method must return a value that is promotable to the declared type
- if an array type is used, the method must return a value of the same array type. For example, if the returnType is String[][] then the method must return a String[][] array
- a method can declare a return type without having a return statement in its body

```
class DizzyDean {
     int pitch() { throw new RuntimeException("90 mph?!"); }
}
```

parameterList

- consists of a comma-separated list of parameter declarations

```
myMethod(int a, long c, boolean flag){}
```

- a parameter may also be declared final

```
        myMethod(final int i){}
```

throwsClause

- consists of the keyword **throws** and a comma-separated list of the exceptions that may be thrown
- identifies all the **checked** exceptions that may be thrown but not caught by the method
- the throws clause must include exceptions that may be thrown by another method being invoked by the declared method
- it is not necessary to throw the predefined exceptions which are subclasses of the **Error** or **RuntimeException** classes (JLS §8.4.4)
- a method that overrides another method cannot be declared to throw more checked exceptions than the method being overidden.(JLS § 8.4.4)

```
class classA {
     void methodA() throws exX, exY{
         // method body
     }
}


class classB extends classA {
     void methodA() throws exX { // can throw less exceptions
        // method body
    }
}

class classC extends classA {
    void methodA() throws exX, exY, exZ {  // illegal
      // method body
    }
}
```

Method Signature

- A method signature is made up of the method name and parameter list (it does not include the return type)
- it is illegal for a class to declare two methods with the same signature

Method body

- a **static** method cannot use **this** or **super** operators in it's body (static implies a **class** method unrelated to any specific instance) (JLS §8.4.3.2)
- a method declared **native** or **abstract** has a **semi-colon (;)** for a body. Do not use curly braces {}. (JLS §8.4.5)

```
    Example of native and abstract method declarations:

        public native void close() throws IOException;
        public abstract void open() throws IOException;

    versus non-native or abstract method declaration:

        public void close() throws IOException {
            // Method body
        }
```

- if a method is declared **void** then the body should not include a return statement that has an expression (JLS §8.4.5)

```
public void methodA() {
    return( 1 + 1 );          // illegal
}

public void methodA() {
    return;                   // legal
}
```

Also see

Understanding that parameters are passed by value and not by reference

Code Examples

- TestMethods.java

Tips

- any method can throw a Runtime or Error exception without declaring it in the throws clause
- methods having the same name and parameter types do not have the same signature *unless* the parameter types are listed in the same order

Traps

- an **abstract** method also declared **private, native, final, synchronized,** or **strictfp**
- an **abstract** method declared in a non-abstract class
- a **native** or **abstract** method with a method body
- method returning a type which is not convertible to the declared return type
- a **void** method returning a value
- a **static** method referencing **this** or **super**

Language Fundamentals - main()

Syntax

```
public static void main(String[] args) {
    // method body
}
```

- entry point for a Java application
- required by all Java applications (not required in Applets)
- **must be** declared **public static void**
- **void** must appear before **main()**

```
Example:
    static public void main(String[] args){}    // legal
    public static void main(String[] args){}    // legal
    public void static main(String[] args){}    // illegal
```

- can also be declared **final**
- **main()** has only one argument: a String array
- the argument can be declared in many ways and the variable name does not have to be **args**

```
Example:

        main( String args[] )
        main( String [] args )
        main( String[] params )
        main( String[] args )     // standard convention
```

- the **args** array is used to access **command line arguments**

```
Example:
 java MyApp test this out
```

- the args array uses a **zero based** index therefore args[0] would return "test" in the above example
- an application can have more than one **main()** method as **every class** can have a **main()** method
- which main() is used by an application depends on the class started at runtime
- advantage is that each class can use it's own **main()** as a testing structure for the class
- **main()** is **inherited** and can be overridden if not declared **final**

Code compiled with JDK 1.3 will work ok even it is declared **private, protected** or has no access modifier; however, for the purpose of the certification exam the *correct* method declaration is **public static void main(String[] varname)**
(see discussion at JavaRanch)

Code Examples

- TestMain.java

Tips

- **main()** can be declared **final**
- **main()** is inherited and can be overridden if not declared as **final**
- **args[0]** references first command line argument *after* the application name (arrays in Java are zero-based)
- **main()** can be declared **public static void ...** or **static public void ...**
- the variable name **does not** have to be **args**; can be anything as long as the **type** is **String[]**

Traps

- **main()** declared other than according to the standard convention

Language Fundamentals - Variable declarations and Identifiers

Syntax

```
        modifiers Type declarator;

Example:
        public int i;
        private long myNumber;
        protected myVar = 10;
```

- variables provide named access to data stored in memory
- variables may be declared as a primitive type or a reference type
- Java supports two different kinds of variables: field or class variables and local or automatic variables
- **field** variables are declared as members of a class; they store information (data) relating to an object
- valid field modifiers: public, protected, private, final, static, transient, volatile
- **local** or **automatic** variables are declared within methods; they are temporary placeholders which store values and references to data for objects being operated on by the method
- valid local modifiers: final

Identifiers

- an identifier is an unlimited-length sequence of Java letters and Java digits
- an identifier cannot have the same spelling as a Java keyword, boolean literal, or null literal
- valid identifiers begin with one of the following:
  - a Unicode letter
  - the underscore character (_)
  - a dollar sign ($)
- JLS §3.8 recommends that the dollar sign only be used for identifiers that are mechanically generated (ie within IDE's)
- JPL pg 5.4 recommends sticking to one language when writing identifiers as a number of characters look alike in various languages but have seperate Unicode values
- methods and variables can have the same names; method identifiers always take the form

  ```
  methodName()
  ```

  the parantheses allow Java to recognize the identifier as a method vs a variable and therefore distinguish between the two.

Naming Conventions

- Package names - lowercase.for.all.components
- Class and Interface names - CaptializedWithInternalWordsCaptialized
- Method names - firstWordLowercaseButInternalWordsCapitalized()
- Variable names - firstWordLowercaseButInternalWordsCaptialized
- Constants - UPPER_CASE_WITH_UNDERSCORES

Tips

- variables can have the same name as a method or a class

Traps

- local (automatic) variables declared with a modifier other than **final**
- identifier names beginning with a number or # sign

	Source	Package	Import	Class	Interface	Constructors
	Methods	main()	Identifiers	Keywords	Defaults	Arrays
	Primitives	# Literals	char Literal	String Literals	Class Literals	

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Language Fundamentals - Keywords

Keyword type	Keywords
Primitive types	boolean, byte, char, double, float, int, long, short
Access modifiers	public, private, protected
Special modifiers	abstract, final, native, static, strictfp, synchronized, transient, volatile
Control flow	if, else, do, while, switch, case, default, for, break, continue
OOP specific	class, extends, implements, import, instanceof, interface, new, package, super, this
Exception handling	catch, finally, try, throw, throws
Method specific	return, void
Unused *	const, goto

* Note
• **const** and **goto** are not used in Java however they are reserved as keywords. • **true** and **false** are **Boolean Literals**; **null** is a **null Literal**. They cannot be used as identifiers.

Tips

- Java keywords are **always lowercase**; you can immeadiately eliminate any capitalized words appearing in a question listing possible keywords

Traps

- **main** listed as a possible keyword
- capitalized words listed as possible keywords; particularly wrapper classes Integer, Boolean, etc
- C/C++ keywords listed as possible Java keywords

	Source	Package	Import	Class	Interface	Constructors
	Methods	main()	Identifiers	Keywords	Defaults	Arrays
	Primitives	# Literals	char Literal	String Literals	Class Literals	

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Language Fundamentals - Default values

Type	Default value
boolean	false
byte	0
char	'\u0000'
short	0
int	0
long	0l
float	0.0f
double	0.0d
Object	null
Array	based on Array type

Automatic Initialization

- Field variables (class members) are automatically initialized to default values
- Local variables (method or constructor variables) are **not** automatically initialized
- Arrays, whether field or local variables, are automatically initialized to the default values of their declared type

```
class CheckInit {

    // field variable
    static int i;
    // field array reference variable
    static String[] s = new String[10];

    static void myMethod(){

        int j;                // local variable
        int[] a = new int[5]; // local variable array

        // causes compile error if not explicitly initialized
        j = 10;

        System.out.println("  Local variable: " + j);
        System.out.println(" Local array ref: " + a[3]);
    }

    public static void main(String[] args) {
        System.out.println("Field variable i: " + i);
        System.out.println(" Field array ref: " + s[2]);
        myMethod();
    }
}

Output of CheckInit:

    Field variable i: 0         // default value of int
     Field array ref: null      // default value for String[]
      Local variable: 10        // explicit value
     Local array ref: 0         // default value of int[]
```

Timing and duration of variable initializations (JLS §4.5.3)

Variable Type	Definition	Initialization

Class (Field)	Declared with the **static** keyword within a class or interface	Created when the class or interface is prepared. Automatically initialized to the default value of its type Duration: as long as the class is loaded
Instance (Field)	Declared within a class **without** the keyword **static**	Created when a new instance is created Automatically initialized to the default value of its type Duration: for the life of the instance object
Array components	unnamed variables created when an array object is created not when declared	intialized to the default value of the array type Duration: until the array is no longer referenced
Method parameters	named argument values passed to a method	a new parameter variable is created each time the method is invoked initialized with the corresponding argument value from the method call Duration: method execution
Constructor parameters	named argument values passed to the constructor	a new parameter variable is created each time a new instance is created or the constructor is called initialized to the corresponding argument value Duration: construction execution
Exception-handling parameter	variables in a catch clause	a new exception-handling parameter is created each time an exception is caught by a catch clause initialized with the actual object associated with the exception Duration: catch clause execution
Local variables	declared by local variable declarations	a new local variable is created whenever flow of control enters a new block or **for** statement initialized to whatever value is explicitly set within the block or **for** statement Duration: execution of the block or **for** statement

Tips

- only field variables are automatically initialized to their types default value; local variables must be explicitly initialized
- arrays are initialized to the default value of their type when they are created, not declared, even if they are local variables

Traps

- an empty string vs **null** as the default value for a String object

Language Fundamentals - Arrays

Array declarations

- arrays are Java objects
- all Java arrays are technically one-dimensional. Two-dimensional arrays are arrays of arrays.
- declaring an array **does not** create an array object or allocate space in memory; it creates a variable with a reference to an array
- array variable declarations must indicate a dimension by using []

```
Examples of valid array declarations: (JJ pg84)

    String[]s;
    String []s;
    String [] s;
    String [ ] s;       // extra white space ignored
    String[] s;
    String[   ] s;      // extra white space ignored
    String s[];
    String s [];
    String s [   ];     // extra white space ignored

    String[] s[];
    String[][]s;
    String s [] [  ];   // extra white space ignored
```

- declaring the size of the array with the following notation is **illegal**

```
    String[5] s;        // illegal declaration
```

- the **standard convention** for declaring arrays is:

```
    String[] s;         // one-dimensional array
    String[][] s;       // two-dimensional array
```

Initializing arrays

- all arrays are zero-based
- arrays must be indexed by **int** values or **byte, short** or **char** values (as these can be promoted to int) (JLS §10.4)
- using a **long** index value to access an array causes a **compile error**
- attempting to access an array with an index less than 0 or greater than the length of the array causes an **ArrayIndexOutOfBoundsException** to be thrown at runtime (JLS §10.4)
- since arrays are Objects they can be initialized using the **new** operator
- when created, arrays are automatically initialized with the default value of their type

```
String[] s = new String[100];   // default values: null
boolean[] b = new boolean[4];   // default values: false
int[] i = new int[10][10];      // default values: 0
```

- array references declared as **members** are initialized to **null** BUT array references declared in methods **are not** initialized

```
class TestArray  {
```

```
    int[] arr;          // member declaration, initialized to 'null'

    public static void main(String[] args) {
        int[] arr1;     // reference variable 'arr1' not initialized

        // compiles ok
        System.out.println("arr:" + new TestArray().arr);
        // compile error
        System.out.println("arr1: " + arr1);
    }
}
```

- as arrays are allocated at runtime, you can use a variable to set their dimension

```
int arrSize = 100;
String[] myArray = new String[arrSize];
```

- you can use curly braces {} as part of an array declaration to initialize the array

```
String[] oneDimArray = { "abc","def","xyz" };
```

Note

• Curly braces {} can only be used in array declaration statements.
 	
`String[] s;`
`// illegal initialization`
`s = { "abc", "def", "hij");`
 	
`int[] arr = new int[] {1,2,3};  // legal`

- you can assign an array a **null** value but you can't create an empty array by using a blank index

```
int[] array = null;            // legal
// illegal initialization
int[] array = new int[];
```

Initializing two-dimensional arrays

- the first dimension represents the rows, the second dimension, the columns
- curly braces {} may also be used to initialize two dimensional arrays. Again they are only valid in array declaration statements.

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

- you can initialize the row dimension without initializing the columns but not vice versa

```
int[][] myArray = new int[5][];
// illegal
int[][] myArray = new int[][5];
```

- the length of the columns can vary

```
class TestTwoDimArrays {
   // initialize # of rows
   static int [][] myArray = new int[3][];

   public static void main(String[] args) {

   myArray[0] = new int[3];   // initialize # of cols
   myArray[1] = new int[4];   // in each row
   myArray[2] = new int[5];

   for(int i=0; i<3; i++)    // fill and print the array
       fillArray(i, i+3);

   System.out.println();
   } // end main()

   private static void fillArray(int row, int col) {
```

```
            for( int i=0; i<col; i++)
                myArray[row][i] = i;

            for( int i=0; i<col; i++)
                System.out.print(myArray[row][i]);

            System.out.println();
        }

    }

    Output of TestTwoDimArrays:

        012
        0123
        01234
```

Also see

Sun Tech Tip: Manipulating Java Arrays

Code Examples

- TestTwoDimArrays.java

Tips

- array index operator [] has highest level of precedence
- integer variables can be used as array dimension values

Traps

- incorrect array declaration statements, particularly:
  arrayType [#] varName;
- incorrect array initialization statements, particularly:
  arrayType[] varName = new arrayType[2];
  varName = { value, value, value };
- **negative** values for array index
- **long** value for array index
- array declaration used as an array creation statement

	Source	Package	Import	Class	Interface	Constructors
	Methods	main()	Identifiers	Keywords	Defaults	Arrays
	Primitives	# Literals	char Literal	String Literals	Class Literals	

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Language Fundamentals - Primitive Types

Data Type	Bit Size	Range	Min/Max values	Default
boolean	n/a	true or false	n/a	false
byte	signed 8-bit integer	$-(2^7)$ to 2^7-1	-128 to 127	0
char	16-bit Unicode 2.0 character	0 to $2^{16}-1$	0 to 65,535	\0000
short	signed 16-bit integer	$-(2^{15})$ to $2^{15}-1$	-32,768 to 32,767	0
int	signed 32-bit integer	$-(2^{31})$ to $2^{31}-1$	-2,147,483,648 to 2,147,483,467	0
long	signed 64-bit integer	$-(2^{63})$ to $2^{63}-1$	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0l
float	signed 32-bit floating-point	NEGATIVE_INFINITY to POSITIVE_INFINITY	Can also have the value NaN (Not a number)	0.0f
double	signed 64-bit floating-point	NEGATIVE_INFINITY to POSITIVE_INFINITY	Can also have the value NaN (Not a number)	0.0d

- arithmetic with floating-point numbers will never throw an exception; instead one of the constant values: NEGATIVE_INFINITY, POSITIVE_INFINITY, or NaN are returned (BB pg 123)
- Variables declared as primitive types are **not** object references. They are placeholders for storing primitive values (JJ pg29)
- by default integer values are of type **int** and floating-point values are of type **double**
- **float** values are **single-precision**
- **double** values are **double-precision**

Wrapper classes

- all the primitive types have corresponding **wrapper** classes which allow you to create objects of type Integer, Boolean, Float, etc.
- the wrapper classes have the same names as the primitive types **except** they begin with a Captial.

!!! Warning - do NOT mix up the Types !!!

```
    boolean b;

IS NOT THE SAME AS

    Boolean b;

You can say:    boolean b = true;
You CANNOT say:
    Boolean b = true;  -> Boolean is a class, must use
    Boolean b = new Boolean(true);
```

Also see

Differentiate between reference and primitive types

Traps

- variables of primitive types handled as Objects

	Source	Package	Import	Class	Interface	Constructors
	Methods	main()	Identifiers	Keywords	Defaults	Arrays
	Primitives	# Literals	char Literal	String Literals	Class Literals	

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Language Fundamentals - Numeric Literals

- numeric constants are written using **literals**

Integer literals

- Integer constants are strings of octal, decimal, or hexidecimal digits

```
decimal     base 10          10
octal       base 8           010    (8)    // preceded by a zero
hex         base 16          0xA   (16)    // preceded by 0x
```

- Integer constants are **long** if they end in **l** or **L**

```
    32l or 32L          // capital L recommended use
```

- if an **int** literal is assigned to a **short** or a **byte** and it's value is within legal range, the literal is assumed to be a short or a byte.

```
    byte  b  = 5;          // assumed to be a byte
    short s  = 32500;      // assumed to be a short
    short sh = 50000;      // illegal
```

- In all other cases you must explicitly cast when assigning an **int** to a short or byte. (JPL pg 108)

```
    int  i = 5;          // declared and initialized int
    byte b;              // declared byte

    b = i;               // causes compile error
    b = (byte)i;         // compiles
```

Floating-point literals JPL pg 108

- floating-point numbers are expressed as **decimal numbers** with an optional decimal point

```
Examples of valid floating-point numbers:

    0.10
    1.
    .0001
    1.8e1       // 'e' = exponential
```

- at least one digit must be present
- floating-point constants are **double** values unless they are suffixed with an **f** or **F**
- if a **d** or **D** suffix is used they are **double** values

```
    10.5                // assumed double value
    10.5F               // float value
```

- a double constant cannot be assigned to a float variable even if the double is within the float value range; however, a double can be cast as a float

```
    double d = 3.213;       // double constant
    float  f;
```

```
        f = d;                  // compile error
        f = (float)d;           // compiles
```

Traps

- assigning a non-integer literal to a byte, short or character
- assigning a double literal to a float

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Language Fundamentals - Character Literals

- the **char** type represents 16-bit Unicode characters
- Unicode is a superset of the ASCII character set which allows non-English language characters
- any Unicode character can be written as a literal using the Escape character (backslash \) and it's hexadecimal representation

```
'\udddd'    // where 'dddd' = hex digit (0 - F)
```

- single characters are represented within single quotes

```
'a'          // char literal
'9'          // char literal
```

- there are three exceptions that require the use of the Escape character

```
single quote    ' \' '      displays as '
double quote    ' \" '      displays as "
backslash       ' \\ '      displays as \
```

- there are certain special characters which can be represented by escape sequences

Esc Char	Unicode Char	Definition
\n	\u000A	newline
\t	\u0009	tab
\b	\u0008	backspace
\r	\u000D	return
\f	\u000C	form feed
\ddd		octal value

- Octal character constants can have three digits or less (\000 through \377)

!!! Warning !!!
The compiler translates Unicode characters at the beginning of the compile cycle. Using the Unicode escape characters **\u000A** for *newline* and **\u000D** for *return* in a String or comment produces a compile-error as they are interpreted, literally, as 'end-of-line'. Always use the special characters '\n' or '\r'

Traps

- String literal "c" assigned to **char** type

	Source	Package	Import	Class	Interface	Constructors
	Methods	main()	Identifiers	Keywords	Defaults	Arrays
	Primitives	# Literals	char Literal	String Literals	Class Literals	

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Language Fundamentals - Class Literals

- new in JDK 1.1
- class literals are created by appending **.class** to the name of a primitive or reference type

```
System.out.println(int.class);
// output: int
System.out.println(System.class);
// output: java.lang.System
```

- you cannot use a variable with .class

```
int i = 5;
String s = "Hello";

System.out.println(i.class);  // compile error
System.out.println(s.class);  // compile error
```

Also see

Sun Tech Tip: Using Class Literals

	Source	Package	Import	Class	Interface	Constructors
	Methods	main()	Identifiers	Keywords	Defaults	Arrays
	Primitives	# Literals	char Literal	String Literals	Class Literals	

The `java.lang` Package Certification - String Class

- Strings can be created **implicitly** by:
  1. using a quoted string ie "Hello", or,
  2. by using + or += on two String objects to create a new one
- strings can be created **explicitly** by using the **new** operator
- new String() creates an empty string
- new String(String value) creates a new string that is a copy of the string object **value**
- two basic String methods are
  - public int length()
  - public char charAt(int index). Index values range from 0 to length()-1
- any String method requiring an **index** will throw an IndexOutOfBoundsException if 0 > index > length()-1
- there are also a number of indexOf() methods which allow you to find the first and last position of a character or substring within a string

```
indexOf(char ch)            // first position of 'ch'
indexOf(String str)         // first position of 'str'
lastIndexOf(char ch)        // last position of 'ch'
lastIndexOf(String str)     // last position of 'str'
```

- each of the above methods also have overloads that allow a second int start argument which specifies the character position other than 0 from which to begin the search
- all the methods return -1 if the character or string is not found

Comparison

- characters in strings are compared numerically by their Unicode values
- equals() method returns true if both string objects are of the **same** length and have the **same** sequence of Unicode characters
- equalsIgnoreCase() can be used to compare strings, ignoring wether a character is lowercase or uppercase
- compareTo returns an int that is <, =, or > than 0 if one string, based on it's Unicode characters, is less-than, equal to or greater-than another string
- regions of strings can also be compared

```
public boolean regionMatches(int start, String other,
                             int ostart, int len)
public boolean regionMatches(boolean ignoreCase, int start,
                             String other,
                             int ostart, int len)
```

- an area of each string is compared for the number of characters specified by **len**
- simple tests for the beginning and ending of strings can be done using

```
public boolean startsWith(String prefix, int toOffset)
public boolean startsWith(String prefix)
public boolean endsWith(String suffix)
```

Note

- These methods return **true** if a comparison is done with an empty string

```
"String".endsWith("");    // true
"String".startsWith("");  // true
```

Comparisons using intern()

- two utility methods hashCode() and intern() are available
- hashCode() returns the same hash value for any two strings having the same contents
- intern() returns a String that has the same contents as the one it is invoked on AND any two strings having the same content return **the same String object** allowing comparisons to be done using String references vs string contents
- using intern() for comparison purposes is equivalent to comparing contents but is much **faster**

Related strings

- several methods return **new** strings that are like the original but with the specified modifications

```
public String concat(String str)
public String replace(char oldChar, char newChar)
public String substring(int beginIndex)
public String substring(int beginIndex, int endIndex)
public String toLowerCase()
public String toUpperCase()
public String trim()
```

Because all of the above methods return **new** strings; comparisons such as

```
String s = "String";    // in the pool

if(" String ".trim() == s)
   System.out.println("Equal");
else
   System.out.println("Not Equal");
```

OR

```
if(" String ".trim() == "String")
   System.out.println("Equal");
else
   System.out.println("Not Equal");
```

- produce NOT EQUAL. The string pool is NOT checked for a matching string and as a result the string object references are always different or, not equal (refer to String Literals - intern() for more info on the string pool)
- HOWEVER, if the invoked method **does not produce a different string** ie the resulting string, after the method invocation, is the same as the original, THEN the original object reference is returned by the method and the results are EQUAL

```
if("String".substring(0,6) == "String")
    System.out.println("Equal");
else
    System.out.println("Not Equal");

if("String".replace('t','t') == "String")
    System.out.println("Equal");
else
```

```
            System.out.println("Not Equal");
```

Strings and Arrays

- there are a number of constructors and methods that will convert a character array to a String and vice versa

```
public String(char[] value)
public String(char[] value, int offset, int count)

public static String copyValueOf(char[] data)
public static String copyValueOf(char[] data,
                                 int offset, int count)
public void getChars(int srcBegin, intSrcEnd,
                     char[] dst, int dstBegin)
public char[] toCharArray()
```

- there are also a number of constructors and methods that convert 8-bit character arrays to and from 16-bit String objects

```
public String(byte bytes[], int offset, int length)
public String(byte bytes[])

public byte[] getBytes()
public String(byte bytes[], int offset, int length, String enc)
public String(byte, bytes[], String enc)
public byte[] getBytes(String enc)
```

- where **enc** is the standard name for the character language encoding ie **UTF8** or **ISO-Latin-1**

Also see

Sun Tech Tip: Interning Strings

Example Code

- TestStringOperations.java
- TestStringCompares.java
- TestStringModifications.java
- TestStringMethods.java
- TestParseLine.java

The `java.lang` Package Certification - StringBuffer Class

- used to modify or manipulate the contents of a string
- StringBuffer objects are NOT implicitly created; the following will not compile

```
StringBuffer sb = "Hello";
StringBuffer sb = { "Hello" };
```

- you must use the **new** operator to invoke one of three constructors

```
public StringBuffer()
public StringBuffer(int length)
public StringBuffer(String str)
```

- every StringBuffer has an initial capacity (length) of 16 characters
- if the internal buffer overflows it is automatically made larger however it is more efficient to specify the capacity only once,
- there are three methods available to manage capacity

```
public StringBuffer(int capacity)
public synchronized void ensureCapacity(int minimum)
public int capacity()
public int length()
public void setLength(int newLength)
```

- the String methods which return a new object ie concat(), replace(), etc actually use StringBuffer behind the scenes to make the modifications and then returns the final String using toString(). For example, the following code (JJ pg 208)

```
String s = "";
s = s + "a" + "b";
```

- is treated, by the compiler, as something similar to

```
String s = "";
s = new StringBuffer("").append("a").append("b").toString();
```

- the StringBuffer class **does not** inherit from String
- to use a StringBuffer object as a parameter to a method requiring a String, use the StringBuffer **toString()** method. For example, to print the result of a StringBuffer object manipulation

```
StringBuffer sb = new StringBuffer("Hello");
sb.append(" World");
System.out.println(sb.toString() );
```

- StringBuffer has overloaded append() and insert() methods to convert any type, including Object and character arrays, to a String; both methods return the original StringBuffer object
- the reverse() method returns the original StringBuffer object with the characters in reverse order
- you can access and modify specific characters or a range of characters

```
public char charAt(int index)
public void setCharAt(int index, char ch)
```

```
public StringBuffer replace(int start, int end, String str)

public StringBuffer deleteCharAt(int index)
public StringBuffer delete(int start, int end)

Note: the subString() method returns a String
public String subString(int start)
public String subString(int start, int end)
```

- there are no methods to remove part of a buffer; you need to create a character array and build a new buffer with the portion of the array you're interested in; this can be done using

```
public void getChars(int srcBegin, int srcEnd,
                     char dst[], int dstBegin)
```

Example Code

- TestStringBuffer.java

	Main Classes	Wrapper Classes	Math Class	String Immutability	String Class	StringBuffer Class

Threads - Overview

- on an operating system a **running program** is known as a **process**
- a single process can have seperate runnable tasks called **threads**
- a thread is a single sequential flow of control within a process
- a thread is also referred to as a **lightweight process**
- with a single-processor CPU, only one thread is executing at any given time
- the CPU quickly switches between active threads giving the illusion that they are all executing at the same time (logical concurrency)
- on multi-processor systems several threads are actually executing at the same time (physical concurrency)
- **multi-programming** occurs when **multiple programs or processes** are executed
- **multi-threading** occurs when concurrency exists amoung threads running in a single process (also referred to as multi-tasking)
- Java provides support for multi-threading as a part of the language
- support centers on the:
  - java.lang.Thread class
  - java.lang.Runnable interface
  - java.lang.Object methods wait(), notify(), and notifyAll
  - synchronized keyword
- every Java program has at least one thread which is executed when main() is invoked
- all user-level threads are explicitly constructed and started from the main thread or by a thread originally started from main()
- when the last **user** thread completes any **daemon** threads are stopped and the application stops
- a thread's default daemon status is the same as that of thread creating it
- you can check the daemon status using **isDaemon()**
- you can set the daemon status using **setDaemon()**.

  You cannot change a thread's status after it has been started
- **main()** daemon status is **false**
- if you want all your threads to quit when **main()** completes you can set their status to daemon using **setDaemon(true)**
- there are two basic ways to create and run threads
  1. by subclassing the Thread class
  2. by implementing the Runnable interface

Also see

- Sun Tutorial on Threads
- IBM Redbook: Java Thin-Client Programming - Introduction to Threads
- SunTech Tip: Why Use Threads?
- Beware the daemons
- Exploring Java, Chapter 6, Threads

Example Code

- Source code to check daemon status of main()

	Overview	Thread Class	Runnable Interface	Thread States	Scheduling	Ending a Thread
	Execution	Synchronization	Locking Protocols	synchronized keyword	wait()	notify(), notifyAll()
	Thread Mechanics					

Threads - The Thread Class

- the easiest way to create a thread is by subclassing java.lang.Thread

```
class BasicThread extends Thread {
    char c;

    BasicThread(char c) {
        this.c = c;
    }
}
```

- to actually start the thread running you must invoke its start() method

```
BasicThread bt  = new BasicThread('!');
BasicThread bt1 = new BasicThread('*');
bt.start();
bt1.start();
```

- the start() method allocates system resources required for a thread, schedules the thread to run and invokes the run() method
- the above code will execute but nothing will happen
- if you want your thread to do something you need to override the run() method
- the run() method is actually defined in the Runnable interface which the class Thread implements

```
public void run() {
    for(int i=0; i<100; i++) {
        System.out.print(c);
    }
}
```

- if the above code is added and the threads started you see something like:

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*
!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!****
***********************************************
```

- the output is intermingled because the threads are running concurrently and are interleaved
- you can alter thread processing with program control mechanisms
- one way is to use the sleep() method which is defined in the Thread class
- the sleep() method stops the execution of a thread for a given number of milliseconds
- it also throws an InterruptedException so you need to wrap it in a try-catch block
- adding sleep() to the run method can alter the threads execution

Note
• the sleep() method uses a timed wait() but does not tie up the current object's lock (for information on *locks* see Synchronization)

```
New run() method:

    public void run() {
        for(int i=0; i<100; i++) {
```

```
            System.out.print(c);

            try{
                sleep((int)(Math.random() * 10));
            } catch( InterruptedException e ) {
                System.out.println("Interrupted");
            }
        }
    }
```

```
Example output:

*!*!!*!*!**!!!*!!*!***!*!!***!*!*!*!*!!!!***!!*!*
!!*!**!*!!*!**!*!**!!*!**!!*!!*!*!**!*!***!!!!*!*
!!*!**!*!*!*!*!*!*!!*!*!*!!!*!*!!!*!*!!*!*!*!*!
!*!**!*!*!**!**!!*!***!!!****!*!!****!*!**!!**!!!
****
```

- you can give a thread a name by creating it with a String argument

```
    Thread t = new Thread("Thread1");
```

- if a thread is created without a name, one is automatically generated in the form **Thread-n**, where **n** is an integer
- the following is output from TwoThreadsTest which creates two SimpleThread's and displays their automatically generated names using the getName() method of the Thread class.

```
0 Thread-0
0 Thread-1
1 Thread-0
1 Thread-1
2 Thread-0
2 Thread-1
3 Thread-0
3 Thread-1
4 Thread-1
4 Thread-0
DONE! Thread-0
DONE! Thread-1
```

ThreadGroup

- you can group threads using the **ThreadGroup** class
- this allows multiple threads to be handled as one unit ie for setting priority, destroying, etc
- threads in the same group can access information about other threads in the group but not about the parent thread or threads in other groups
- a ThreadGroup can have both daemon and nondaemon threads

Example Code

- BasicThread.java
- Bouncing Ball Applet: UpDown.java
- PrimeNumbers Applet: Ex1.java
- Java Is Hot: Thread1.java
- Traffic Simulation: Traffic.java and SetOfLights.java

	Overview	Thread Class	Runnable Interface	Thread States	Scheduling	Ending a Thread
	Execution	Synchronization	Locking Protocols	synchronized keyword	wait()	notify(), notifyAll()
	Thread Mechanics					

Threads - Thread Synchronization

- every instance of class Object and its subclass's has a **lock**
- primitive data type fields (Scalar fields) can only be locked via their enclosing class
- **fields** cannot be marked as synchronized however they can be declared **volatile** which orders the way they can be used or you can write synchronized **accessor methods**
- **array** objects can be synchronized BUT their elements cannot, nor can their elements be declared **volatile**
- Class instances are Objects and can be synchronized via **static synchronized** methods

Synchronized blocks

- allow you to execute synchronized code that locks an object without requiring you to invoke a synchronized method

```
synchronized( expr ) {
    // 'expr' must evaluate to an Object
}
```

Synchronized methods

- declaring a method as synchronized ie synchronized void f() is equivalent to

```
void f() { synchronized(this) {
    // body of method
  }
}
```

- the synchronized keyword is NOT considered part of a method's signature. IT IS NOT AUTOMATICALLY INHERITED when subclasses override superclass methods
- methods in **Interfaces** CANNOT be declared synchronized
- constructors CANNOT be declared synchronized however they can contain synchronized blocks
- synchronized methods in subclasses use the same locks as their superclasses
- synchronization of an Inner Class is independent on it's outer class
- a non-static inner class method can lock it's containing class by using a synchronized block

```
synchronized(OuterClass.this) {
    // body
}
```

Locking

- locking follows a built-in acquire-release protocol controlled by the synchronized keyword
- a lock is acquired on entry to a synchronized method or block and released on exit, even if the exit is the result of an exception
- you cannot forget to release a lock
- locks operate on a **per thread** basis, not on a per-invocation basis
- Java uses **re-entrant** locks ie a thread cannot lock on itself

```
class Reentrant {
```

```
    public synchronized void a() {
        b();
        System.out.println("here I am, in a()");
    }
    public synchronized void b() {
        System.out.println("here I am, in b()");
    }
}
```

- in the above code, the synchronized method **a()**, when executed, obtains a lock on it's own object. It then calls synchronized method **b()** which also needs to acquire a lock on it's own object
- if Java did not allow a thread to reacquire it's own lock method b() would be unable to proceed until method a() completed and released the lock; and method a() would be unable to complete until method b() completed. Result: deadlock
- as Java does allow reentrant locks, the code compiles and runs without a problem
- the locking protocol is only followed for synchronized methods, it DOES NOT prevent unsynchronized methods from accessing the object
- once a thread releases a lock, another thread may acquire it BUT there is no guarantee as to WHICH thread will acquire the lock next

Class fields and methods

- locking an object does not automatically protect access to **static** fields
- protecting static fields requires a **synchronized static** block or method
- static synchronized statements obtain a lock on the Class vs an instance of the class
- a synchronized instance method can obtain a lock on the class

```
synchronized(ClassName.class) {
    // body
}
```

- the static lock on a class is not related to any other class including it's superclasses
- a lock on a static **method** has no effect on any instances of that class (JPL pg 185)
- you cannot effectively protect static fields in a superclass by adding a new static synchronized method in a subclass; an explicit block synchronization is the preferred way
- nor should you use synchronized(getClass()); this locks the actual Class which might be different from the class in which the static fields are declared

Example Code

- Source code for reentrant example

	Overview	Thread Class	Runnable Interface	Thread States	Scheduling	Ending a Thread
	Execution	Synchronization	Locking Protocols	synchronized keyword	wait()	notify(), notifyAll()
	Thread Mechanics					

Threads - The Runnable Interface

- the Runnable interface declares a single method: run()
- you can execute a Runnable object in its own thread by passing it to a Thread constructor
- here's the BasicThread class modified to use the Runnable interface

```
class RunBasicThread implements Runnable{
char c;

RunBasicThread(char c) {
    this.c = c;
}

// override run() method in interface
public void run() {
    for(int i=0; i<100; i++) {
        System.out.print(c);

        try{
           Thread.sleep((int)(Math.random() * 10));
        } catch( InterruptedException e ) {
            System.out.println("Interrupted Exception caught");
        }
    }
}

public static void main(String[] args) {

    RunBasicThread bt = new RunBasicThread('!');
    RunBasicThread bt1 = new RunBasicThread('*');

    // start RunBasicThread objects as threads
    new Thread(bt).start();
    new Thread(bt1).start();
}
}
```

- the most significant code revisions are shown in red
- note that you can still make use of the methods declared in the Thread class but you now have to use a qualified name ie Thread.sleep() and you have to pass your runnable object to the thread when it is created ie new Thread(bt).start()
- the Clock applet is an example of an Applet (based on the Sun Thread tutorial) using the Runnable interface:

When to implement Runnable vs subclassing Thread

- Whenever your class has to extend another class, use Runnable. This is particularly true when using Applets

Example Code

- Click Applet
- Bouncing Ball applet: UpDown_1

- PrimeNumbers applet: Ex1_a
- Java Is Hot: Thread2

Threads - Thread States

- each thread has a life-cycle all it's own
- during it's life-cycle it can exist in a number of **states**
  - New
  - Runnable
  - Not Runnable
  - Dead

Note
• These states are those used in the Sun Java Thread tutorial. Other references may use 'ready', 'waiting' or other terminology to describe the Runnable and Non-runnable states.

New

a new thread is an empty Thread object; no system resources have been allocated as yet.
Calling any thread method other than start() causes an IllegalThreadStateException

Runnable

a thread enters the Runnable state after the start() method is invoked. The start() method allocates system resources, schedules the thread, and calls the threads's run() method. When the thread actually *runs* is determined by the **scheduler**

Not Runnable

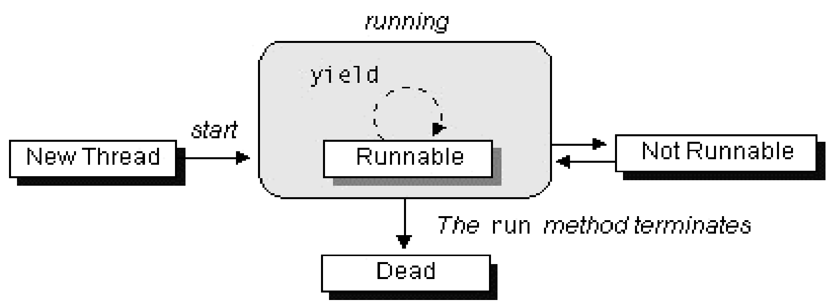
a thread is **not runnable** when
- it's sleep() method is invoked
- it's wait() method is invoked
- it is blocked on I/O ie waiting on system resources to perform an input or output operation

the thread becomes **runnable** again when a specific condition has been met based on the action which put it in the **not runnable** state
- when the number of milliseconds called in sleep() have elapsed
- when the condition it is waiting on has changed and it receives a notify() or notifyAll() message
- when the required system resources are made available and the I/O completes

Dead

a thread enters the **dead** state when it's run() method completes.
an **interrupt** does not kill a thread
the **destroy()** method kills a thread dead but does not release any of it's object locks

Life Cycle of a Thread from Sun Thread Tutorial



- a thread can bounce between **runnable** and **not runnable** states as a result of either

	1. scheduling, or 2. programmer control

Java Quick Reference

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Threads - Thread Scheduling

- execution of multiple threads in some order on a single CPU system is called **scheduling**
- Java uses **fixed-priority scheduling algorithms** to decide which thread to execute
- the thread with the highest priority runs first
- if another thread with a higher priority is started, Java makes the lower priority thread wait
- if more than one thread exists with the same priority, Java quickly switches between them in round-robin fashion BUT only if the operating system uses time-slicing (see below)

Priorities

- it's possible to assign a thread priority
- the Thread class contains three integer priority constants
  1. [1] MIN_PRIORITY
  2. [5] NORM_PRIORITY
  3. [10] MAX_PRIORITY
- the default thread priority is NORM_PRIORITY
- when a thread is created, it takes the priority of the thread which created it
- you can check a threads priority using getPriority()
- you can change a threads priority using setPriority()
- if you change the priority on an executing thread to a lesser priority, it may stop executing as there may be another thread with a higher-priority (BB pg 259)

Actual Scheduling depends on the OS

- the above act as a guide to scheduling however the actual implementation depends on the Operating System
- most operating systems use one of two scheduling methods
  1. Preemptive scheduling
  2. Time slicing
- In **preemptive scheduling** the highest priority thread continues to run until it dies, waits, or is preempted by a thread of higher priority
- In **time slicing** a thread runs for a specific time and then enters the **runnable** state; at which point the scheduler decides wether to return to the thread or schedule a different thread (method used by Win95/NT)
- DO NOT rely on **thread priority** as a guarantee that the highest priority thread will always be running; the operating system has the final say
- priorities are used as **guides** to efficiency
- priority manipulations CANNOT be used as a substitute for locking (see synchronization)

General Conventions for setting priorities (CPJ pg 16)

- following represent the general conventions for setting thread priorities based on the type of activity the thread is involved in

```
Range              Use
-----   ----------------------------
  10    Crisis management
 7-9    Interactive, event-driven
```

```
4-6    IO
2-3    Background computation
 1     Run only if nothing else can
```


	Overview	Thread Class	Runnable Interface	Thread States	Scheduling	Ending a Thread
	Execution	Synchronization	Locking Protocols	synchronized keyword	wait()	notify(), notifyAll()
	Thread Mechanics					

Threads - Ending a Thread

- a thread normally ends when it's execution completes
- there are other methods of stopping it, some of which should not be used

interrupt()

- interrupting a thread tells it that you want it to pay attention
- it does not force the thread to halt although it will wake up a sleeping thread
- **isInterrupted** checks to see if a thread is in an interrupted state
- the static method **interrupted** can be used to clear a thread's interrupted state
- if a thread is waiting or sleeping and the thread is interrupted, the methods wait() and sleep() will throw an **InterruptedException**

join()

- one thread can wait for another to complete using the **join()** method
- invoking **join()** guarantees that the method will not return until the threads **run()** method has completed
- **join()** will also take a milliseconds argument which will cause it to wait until the thread completes for the designated time period

destroy()

- **destroy()** kills a thread dead without releasing any of it's locks which could leave other threads blocked forever
- it's use should be avoided

stop()

- you can force a thread to end by calling **stop()** which in turn throws the Error **ThreadDeath**
- you can also throw ThreadDeath yourself
- ThreadDeath SHOULD NOT BE CAUGHT!
- NOTE: stop() is a deprecated method and should not be used!!!

suspend() and resume()

- both methods are deprecated and should not be used!!

setDaemon(true)

- if you want your thread to stop executing when **main()** completes, set it's daemon status to **true** using **setDaemon(true)**

yeild()

- Java does not time-slice ie it will not preempt a currently executing thread to allow another thread of the same priority to run
- the operating system **may** use time-slicing but you should not rely on time-slicing when creating threads
- a well behaved thread will use the **yield()** method to voluntarily yield to the CPU, giving it a

chance to run another thread of the same priority.

- if no threads of the same priority are available, execution of the yielding thread continues.
- Note: lower priority threads are ignored.
- the **yield()** method only *hints* to the JVM that if there are other runnable threads the scheduler should run one in place of the current thread. The JVM may interpret this hint any way it likes ie how the yield is handled is dependent on the JVM implementation for the operating system

Also see

Sun Tech Tip: Handling Interrupted Exceptions

	Overview	Thread Class	Runnable Interface	Thread States	Scheduling	Ending a Thread
	Execution	Synchronization	Locking Protocols	synchronized keyword	wait()	notify(), notifyAll()
	Thread Mechanics					

Threads - Thread Execution

Alive or Dead?

- the Thread class includes an **isAlive()** method which returns **true** if a thread has been *started* and not *stopped*
- a thread **stops** when its **run()** method finishes executing
- the **isAlive()** method returns **false** if the thread is **new** or **dead**
- there is no way to detect if a thread is not alive because it was never started or because it is dead
- there is also no way to detect if a live thread is Runnable or Not Runnable
- neither can a thread identify which thread started it

Why a thread might not be executing (BB pg 270)

- the thread does not have the highest priority and can't get CPU time
  Example: LowPriority
- the thread has been put to sleep via **sleep()** method
  Example: Sleeping
- there is more than one thread with the same priority and the JVM is switching between the threads
  Example: SamePriority
- the thread's **wait()** method has been invoked
  Example: Waiting
- the thread's **yield()** method has been invoked
  Example: Yielding
- the **suspend()** method has been invoked (this is a deprecated method and should no longer be used)

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Threads - Thread Locking Protocols

Note
• This information is not required for the Certification exam. I've included it because I found it useful in helping me to understand how thread *locks* or *monitors* actually worked.

- each program has an area of **main memory** where it stores it's classes, arrays and variables
- the main memory has a **master copy** of every variable and contains one **lock** for each object
- this main memory area is accessible by all the programs threads
- threads can only communicate thru the shared main memory
- each thread has a **working memory** where it keeps copies of the values of variables it uses or assigns
- to access a shared variable, a thread obtains a lock and flushes its working memory, guaranteeing the shared value will be loaded from **main memory**
- as a thread executes, it operates on its working copies
- when a synchronized block or method is entered, actions by the thread and main memory must occur in a specific order
  1. the thread obtains a lock on the object and flushes its working copy of the object
  2. main memory **reads** the objects value from it's master copy
     1. the thread **loads** the value passed by the main memorys read operation
     2. the thread **uses** it's working copy of the object, passing it to it's excuatable engine
     3. the thread **assigns** the resulting value back to it's working copy
     4. the thread **stores** the new value, passing it back to main memory
  3. main memory **writes** the value passed by the threads store action back to the master copy
  4. the thread releases it's lock on the object
- every **read** action by main memory must be followed by a **load** action in the thread
- every **store** action in the thread must be followed by a **write** action in main memory
- the **read** and **write** actions in main must be executed in the order they were performed in the thread
- every **use** action in a thread must be followed by an **assign** action however an **assign** does not necessarily have to be proceeded by a **use**
- all **use** and **assign** actions must occur in the order dictated by the threads executable code
- **assign** must follow a **load** before a **store** can occur or another **load** can occur
- every **lock** action by a thread MUST be paired with an **unlock**
- as Java allows re-entrant locks, a thread may obtain multiple locks which must be paired with matching unlocks
- only one thread at a time can hold a lock on an object
- a thread is not permitted to unlock a lock it doesn't own
- a thread can only release it's lock after it has performed a **store**

Special case: double and long variables

- double and long variables are handled as two 32-bit variables
- if the variables are not declared volatile and if they are being used by two or more threads the

final result may be a combination of both thread actions

volatile

- declaring a thread volatile prevents the compiler from optimizing and in-lining the code; forcing the thread to reread the value every time the variable is accessed

```
int value = 5;
for(;;) {
    display.showValue(value);
    Thread.sleep(1000);         // wait one second
}
```

- in the above example, value is assigned a literal, under normal conditions, if display.showValue() does not make any changes to value the compiler would in-line the code, assuming value will not be changed while the method is running
- however, if you have other threads that can change value then you should declare it as volatile
- this will stop the compiler from in-lining the code and force the value to be reread each time the loop iterates

Threads - synchronized kewyord

- threads often need to share a common resource ie a file, with one thread reading from the file while another thread writes to the file
- this is an example of a **producer/consumer** relationship

Race conditions

- race conditions occur when multiple, asynchronously executing threads access the same object returning unexpected (wrong) results
- they can be avoided by **synchronizing** the methods which access the shared resource
- the Sun Thread tutorial has an example which uses a Producer class, and a Consumer class which respectively write and read integers from a CubbyHole class. If the CubbyHole class is unsynchronized, as in the following code:

```
public class CubbyHole {
    private int contents;

    public int get() {
        return contents;
    }

    public synchronized void put(int value) {
        contents = value;
    }
}
```

Example output from an unsynchronized Producer/Consumer

```
Consumer #1 got: 0
Producer #1 put: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Producer #1 put: 1
Producer #1 put: 2
Producer #1 put: 3
Producer #1 put: 4
Producer #1 put: 5
Producer #1 put: 6
Producer #1 put: 7
Producer #1 put: 8
Producer #1 put: 9
```

- results are unpredictable; a number may be read before a number has been produced or multiple numbers may be produced with only one or two being read
- adding synchronization ensures that a number is first produced, then read in the correct order

```
public class CubbyHole {
    private int contents;
    private boolean available = false;

    public synchronized int get() {
        while (available == false) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        available = false;
        notifyAll();
        return contents;
    }

    public synchronized void put(int value) {
        while (available == true) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        contents = value;
        available = true;
        notifyAll();
    }
}
```

- the keyword **synchronized** is added to the method declarations
- the Object methods **wait()** and **notifyAll()** are used to communicate between executing threads

Output after code is synchronized

```
Producer #1 put: 0
Consumer #1 got: 0
Producer #1 put: 1
Consumer #1 got: 1
Producer #1 put: 2
Consumer #1 got: 2
Producer #1 put: 3
Consumer #1 got: 3
Producer #1 put: 4
Consumer #1 got: 4
Producer #1 put: 5
Consumer #1 got: 5
Producer #1 put: 6
Consumer #1 got: 6
Producer #1 put: 7
Consumer #1 got: 7
Producer #1 put: 8
Consumer #1 got: 8
Producer #1 put: 9
Consumer #1 got: 9
```

Source for synchronized version

Other examples using synchronized

- An example of using synchronized methods and object locks Thread3

- An example using a synchronized statement on a common object Thread4
- An example of synchronizing access to variables Account
- An example of a museum which uses Walkmen radios for tours: WalkmanHire uses Museum, Counter, and Visitors classes.

Also see

Sun Tech Tip: Using Synchronized Statements
Acquire multiple locks in a fixed, global order to avoid deadlock

	Overview	Thread Class	Runnable Interface	Thread States	Scheduling	Ending a Thread
	Execution	Synchronization	Locking Protocols	synchronized keyword	wait()	notify(), notifyAll()
	Thread Mechanics					

Threads - wait() method

- the **wait()** method causes a thread to release the lock it is holding on an object; allowing another thread to run
- the **wait()** method is defined in the **Object** class
- **wait()** can only be invoked from within **synchronized** code
- it should **always** be wrapped in a **try** block as it throws IOExceptions
- there are actually three **wait()** methods
  1. wait()
  2. wait(long timeout)
  3. wait(long timeout, int nanos)
- the **timeout** is measured in **milliseconds**
- **nanos** is measured in **nanoseconds**
- wait() can only invoked by the thread that own's the lock on the object
- when wait() is called, the thread becomes disabled for scheduling and lies dormant until one of four things occur:
  1. another thread invokes the **notify()** method for this object and the scheduler arbitrarily chooses to run the thread
  2. another thread invokes the **notifyAll()** method for this object
  3. another thread **interrupts** this thread
  4. the specified wait() time elapses
- when one of the above occurs, the thread becomes re-available to the Thread scheduler and competes for a lock on the object
- once it regains the lock on the object, everything resumes **as if** no suspension had occurred
- if the thread was **interrupted** by another thread, an InterruptedException is thrown BUT not until after the thread regains it's lock on the object

Throws

- the wait() method throws three exceptions
  1. IllegalArgumentException - if the timeout value passed is invalid
  2. IllegalMonitorStateException - if the current thread does not own the object's lock
  3. InterruptedException - if another thread interrupts the current thread. The interrupted status of the current thread is cleared

Threads - notify() and notifyAll() methods

- the **notify()** and **notifyAll()** methods are defined in the Object class
- they can only be used within **synchronized** code
- notify() wakes up a single thread which is waiting on the object's lock
- if there is more than one thread waiting, the choice is arbitrary ie there is no way to specify which waiting thread should be re-awakened
- **notifyAll()** wakes up ALL waiting threads; the scheduler decides which one will run
- if there are no waiting threads, the notifys are forgotten
- only notifications that occur **after** a thread has moved to wait state will effect it; earlier notifies are irrelevant

Threads - Thread mechanics

Wait Sets (CPJ pg 184)

- just as every object has a **lock** it also has a **wait set** that is manipulated using **wait(), notify(), notifyAll()** and **Thread.interrupt**
- objects having locks and wait sets are referred to as **monitors**
- any object can act as a monitor
- each object's wait set is maintained internally by the JVM and holds threads blocked by **wait** until a corresponding **notify** is received or the waits are otherwise released
- the methods **wait(), notify()** and **notifyAll()** can only be invoked when the synchronized lock is held on their target

wait()

the following happens when wait() is invoked

- if the current thread has been interrupted, the method exits immeadiately and throws an **InterruptedException**; otherwise, the thread is blocked
- the JVM places the thread in the wait set associated with the target object
- the lock for the target is released but all other locks held by the thread are retained. A full release occurs even if the lock is re-entrantly held due to the thread having nested synchronized calls
- when the thread resumes (ie wait state ends) the lock status is fully restored

timed waits()

- if a timed wait() has not been notified before it's time is up, it releases automatically
- there is no way to tell if a wait has returned due to notification or timeout
- the thread may resume at any arbitrary time after it has timed out based on thread contention, scheduling and timer granularities

notify()

the following happens when notify() is invoked

- the JVM arbitrarily chooses a thread, if one exists, from the target's wait set
- the thread must re-obtain it's synchronized lock on the target object. It will **always** be blocked at least until the thread calling notify() releases it's lock or if some other thread obtains the lock first
- once the lock is obtained, the thread resumes from the point of it's wait

notifyAll()

- works the same as notify() except all waiting threads are removed from the target wait set and allowed to compete for the lock
- only one thread can obtain the lock so they continue one at a time

Thread.interrupt

- if a thread suspended in wait is invoked, the same notify mechanics apply except that after re-acquiring the lock, an InterruptedException is thrown
- if an interrupt and notify occur together there is NO guarantee as to which will take

precedence

Example Code

- Using wait() and notify() to control access to a shared resource Thread5

	Overview	Thread Class	Runnable Interface	Thread States	Scheduling	Ending a Thread
	Execution	Synchronization	Locking Protocols	synchronized keyword	wait()	notify(), notifyAll()
	Thread Mechanics					

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Overloading, Overriding, Runtime Types and Object Orientation - Encapsulation

- objects have both **state** (details about itself) and **behaviour** (what it can do)
- a software object maintains information about its **state** in **variables**
- what an object can do, its behaviour, is implemented with **methods**
- in Object-oriented programming (OOPs), an object is **encapsulated** when its variables and methods are combined into a single component
- encapsulation also involves **access** to an object; its **interface**
- a tightly encapsulated object hides all it's variables and provides public **accessor** methods ie the only way you can use the object is by invoking it's public methods

"Hiding data behind methods so that it is inaccessible to other objects is the fundamental basis of data encapsulation." (JPL pg.12)

- encapsulation has two main benefits: (VA pg44)
  1. modularity
  2. maintainablity

Modularity

- because the object encapsulates all it's variables and the methods needed to make it work, it is a self-contained entity that can be maintained independently of other objects

Maintainability

- because the object hides it's implementation details behind a well-defined interface, the details can be changed without affecting other parts of the program

Example

```
class TestBook{
  public static void main(String[] args) {
    Book b1 = new Book();
    System.out.println(b1);

//  b1.title = "Java Programming Language";
//  b1.author = "Ken Arnold and James Gosling";

   // must use accessor methods
   b1.setTitle("The Java Programming Language: Second Edition");
   b1.setAuthor("Ken Arnold and James Gosling");

// System.out.println(b1.title, b1.author);

   System.out.println(" Title: " + b1.getTitle() );
   System.out.println("Author: " + b1.getAuthor() );
    }
}
```

- In the example code, the instance variables **title** and **author** are **private**; they can only be accessed by their gettor and settor methods
- any attempt to directly set or get the variables produces a compile error

Example Code

- TestBook.java

	Encapsulation	Polymorphism	isA/hasA	Overloading	Overriding	Field Variables
	Initialization	Top-level Classes	Inner Classes	Static Nested Classes	Local Classes	Anonymous Classes

Overloading, Overriding, Runtime Types and Object Orientation - Polymorphism

- *polymorphism* translates from Greek as **many forms** (*poly - many morph - forms)*
- in OOP's it refers to
  the propensity of objects to react differently to the same method (VA pg 110)
- method **overloading** is the primary way polymorphism is implemented in Java

Overloading methods

- overloaded methods:
  1. appear in the same class or a subclass
  2. have the **same name** but,
  3. have different **parameter lists**, and,
  4. can have different **return types**
- an example of an overloaded method is **print()** in the java.io.PrintStream class

```
public void print(boolean b)
public void print(char c)
public void print(char[] s)
public void print(float f)
public void print(double d)
public void print(int i)
public void print(long l)
public void print(Object obj)
public void print(String s)
```

- the actual method called depends on the object being passed to the method
- Java uses **late-binding** to support polymorphism; which means the decision as to which of the many methods should be used is deferred until runtime

Overriding methods

- late-binding also supports **overriding**
- overriding allows a subclass to **re-define** a method it inherits from it's superclass
- overriding methods:
  1. appear in subclasses
  2. have the **same name** as a superclass method
  3. have the **same parameter list** as a superclass method
  4. have the **same return type** as as a superclass method
  5. the **access modifier** for the overriding method may not be more restrictive than the access modifier of the superclass method
     - if the superclass method is **public**, the overriding method must be **public**
     - if the superclass method is **protected**, the overriding method may be **protected** or **public**
     - if the superclass method is **package**, the overriding method may be **packagage**, **protected**, or **public**
     - if the superclass methods is **private**, it is **not inherited** and overriding is not an issue

6. the **throws** clause of the overriding method may only include exceptions that can be thrown by the superclass method, including it's subclasses

```
class LBException extends Exception {}
class LBException1 extends LBException {}

In superclass:
    public void testEx() throws LBException {
        throw new LBException();
    }

In subclass:

    public void testEx() throws LBException1 {
        throw new LBException1();
    }
```

- overriding is allowed as LBException1 thrown in the subclass is itself a subclass of the exception LBException thrown in the superclass method

Side effect of late-binding

- it is Java's use of late-binding which allows you to declare an object as one type at compile-time but executes based on the actual type at runtime

```
class LB_1 {

    public String retValue(String s) {
        return "In LB_1 with " + s;
    }

}

class LB_2 extends LB_1 {

    public String retValue(String s) {
        return "In LB_2 with " + s;
    }

}
```

- if you create an LB_2 object and assign it to an LB_1 object reference, it will compile ok
- at runtime, if you invoke the retValue(String s) method on the LB_1 reference, the LB_2 retValue(String s) method is used, not the LB_1 method

```
    LB_2 lb2 = new LB_2();
    LB_1 lb3 = lb2;         // compiles ok

    System.out.println(lb3.retValue("Today"));

Output:
    In LB_2 with Today
```

Example Code

- TestLateBinding.java

	Encapsulation	Polymorphism	isA/hasA	Overloading	Overriding	Field Variables

Overloading, Overriding, Runtime Types and Object Orientation - is A vs Has A

- **is a** defines a direct relationship between a superclass and a subclass
- **has a** identifies a relationship in which one object contains another object (defined by **field variables**)

Examples

- A circle is a shape that has a center point and a radius. (JJ pg 138)

```
public class Circle extends Shape { // a circle is a shape
    Point  center;                  // a circle has a point
    double radius;                  // a circle has a radius
}
```

- Define a class hierarchy for the following classes (BB pg14):
  1. An Employee class that maintains an employee number.
  2. A Full-time employee class that maintains an employee number, hours worked per week and calculates it's own pay using a salary() method.
  3. A Retired employee class that maintains an employee number, the number of years worked, and calculates it's own pay using a salary() method.

```
public class Employee {      // an employee
    long   id;               // has an id, and
    String status;           // a status
}

abstract class EmployeeStatus extends Employee {
    abstract double salary();
}

// fulltime is a status
class FullTime extends EmployeeStatus {
    double hrs;
    double salary(){
         return hrs * 60.0;
    }
}

// retired is a status
class Retired extends EmployeeStatus {
    int years;
    double salary() {
        return 0;
    }
}
```

- Create classes for 2DShape, Circle, Square and Point. Points have an (x,y) location. Circles have an (x,y) location and a radius. Squares have an (x,y) location and a length. (BB pg17)

```
class Point {        // a point
    int x;           // has an x-location, and
    int y;           // a y-location
```

```
}

class 2DShape {      // all 2DShapes
    Point p;         // have a point
}

class Circle extends 2DShape {  // a circle is a 2DShape
    double radius;              // and has a radius
}

class Square extends 2DShape {  // a circle is a 2DShape
    double length;              // and has length
}
```


	Encapsulation	Polymorphism	isA/hasA	Overloading	Overriding	Field Variables
	Initialization	Top-level Classes	Inner Classes	Static Nested Classes	Local Classes	Anonymous Classes

Overloading, Overriding, Runtime Types and Object Orientation - Overloading Methods

- overloaded methods can have the same name but must have different parameter lists
- parameter lists are considered different if the order of the arguments are different
- a subclass method can overload a superclass method

Examples (based on BB pg 194-5)

- the following code shows the method test(int i, long j) in a Super class, and method test(long j, int i) in a Sub class

```
Super class:
    test(int i, long j);

Sub class
    test(long j, int i);
```

- this code will compile fine if any variables passed to the methods are easily recognizable as either an `int` or a `long`

```
    Sub sb = new Sub();
    // second arg is defined as L(ong); no ambiguity
    sb.test(100, 3000L);

Output:
    uses test(int i, long j) in Super class
```

- however, if the compiler cannot differentiate between a long and an int a compiler error will occur

```
    Sub sb = new Sub();
    // causes compile-error, 3000 can also be an int
    sb.test(100, 3000);

Ouput:
    compile-error: reference to test() is ambiguous
```

!!! Warning !!!
• When analyzing code, watch for ambiguous references that can cause compile errors.

Overloading constructors

- you can overload constructors within the same class

```
    class SuperCtor {
        SuperCtor(){}
        SuperCtor(int i) {} // compiles ok
    }
```

- you can't overload them in subclasses as they must have the same name as the class (ie they would have to have the superclass name and would therefore not be constructors in the subclass)

```
class SuperCtor {
    SuperCtor(){}
}

class SubCtor() {
    SuperCtor(){}  // compile-error
}
```

Also see

- Polymorphism
- Sun Tech Tip: Overload Resolution

Example Code

- TestOverload.java
- TestOverloadCtor.java

Overloading, Overriding, Runtime Types and Object Orientation - Overriding Methods

- **fields** cannot be overridden but they can be **hidden** ie if you declare a field in a subclass with the same name as one in the superclass, the superclass field can only be accessed using **super** or the superclasses type
- a subclass can override methods in it's superclass and change it's implementation
- it must have the same **return type**, **name**, and **parameter list** and can only throw **exceptions** of the same class/subclass as those declared in the original method

```
class Super {
    void test() {
        System.out.println("In Super.test()");
    }
}

class Sub extends Super {
    void test() {           // overrides test() in Super
        System.out.println("In Sub.test()");
    }
}
```

- cannot have **weaker** access rights than the original method

```
In Sub class:
    // compile-error, original has package access
    private void test() {}
    protected void test() {}    // compiles ok
    public void test() {}       // compiles ok
```

- you can have multiple **overloaded** methods in a class but **only one** overriding method

```
In Sub class:
  void test() {}            // overrides test() in Super
  public void test() {}  // compile-error: test() already declared
                         // different access modifiers not part of
                         // method signature for naming purposes
  void test(String str) {}// compiles ok, overloads test()
```

- Only accessible **non-static** methods can be overridden
- **static** methods can be **hidden** ie you can declare a static method in the subclass with the same signature as a static method in the superclass. The superclass method will **not be** accessible from a subclass reference
- any class can override methods from its superclass to declare them **abstract**, turning a concrete method into an abstract one at that point in the type tree. Useful when a class's default implementation is invalid for part of the class hierarchy (JPL pg 77)

Overriding with constructors

- you cannot override a constructor in a superclass as they are not inherited
- you cannot override a constructor in the same class as they would both have the same signatures; get an 'already declared' compile-error
- if you're instantiating a Subclass object and if the Superclass constructor calls a method that is overridden in the Subclass, the Subclass method will called from the superclass

constructor -- NOT the one in the superclass

```
class Super {
    Super(){
        System.out.println("In Super constructor");
        test();
    }

    void test() {
        System.out.println("In Super.test()");
    }
}

class Sub extends Super {
    Sub() {
        System.out.println("In Sub constructor");
    }

    void test() {       // overrides test() in Super
        System.out.println("In Sub.test()");
    }
}

Output if Sub sb = new Sub() is invoked:
    In Super Constructor
    In Sub.test()
    In Sub Constructor
```

Also see

- Polymorphism

Example Code

- TestOverride.java

	Encapsulation	Polymorphism	isA/hasA	Overloading	Overriding	Field Variables
	Initialization	Top-level Classes	Inner Classes	Static Nested Classes	Local Classes	Anonymous Classes

Overloading, Overriding, Runtime Types and Object Orientation - Field Variables

- consider the following scenario:
  1. Super has a field 'i' and a method test() which displays 'i'
  2. Sub is a subclass of Super with it's own field 'i' and method test()
  3. Super calls the test() method in it's constructor
- Which value for 'i' will be displayed when a Sub object is instantiated?
- Answer: the default value of the field 'i' in Sub
- the subclass object is instantiated as follows:

```
the Superclass constructor is called
the Subclass method test() is used
as the Subclass has not been fully initialized,
   the default value of it's field variable is displayed
the Subclass variables are initialized
the Subclass constructor is called
```

- When an overridden method is called from a superclass constructor both the Subclass method and field variables are used
- Which methods and variables are used when an object reference for a Superclass is created and the assigned object is a Subclass type?
- Answer: both the Subclass methods and variables are used. The declared type is only valid at compile-time. The actual object type is used at runtime.
- **BUT**
- if you access the field variable directly, ie not through a method, the variable for the **declared type** is returned.

```
Creating a Super objref and pointing it to Sub obj
  // subclass obj stored in super reference
  Super sp1 = sb;

  // field variable in Sub object
  test() in Sub uses i: 20.0

  // field variable in Super object
  sp1.i 10
```

Example Code

- TestFields.java

	Encapsulation	Polymorphism	isA/hasA	Overloading	Overriding	Field Variables
	Initialization	Top-level Classes	Inner Classes	Static Nested Classes	Local Classes	Anonymous Classes

Overloading, Overriding, Runtime Types and Object Orientation - Initialization

Steps that occur when a new instance is created (JLS§12.5)

1. memory is allocated for all the instance variables in the class and instance variables in **all** of it's superclasses
2. the instance variables are set to their default values
3. the constructor used in the creation expression is called according to the following:
    1. arguments for the constructor are assigned to newly created parameter variables
    2. if the constructor begins with this(); invoke the constructor recursively following the same five steps
    3. if the constructor does not begin with this(), then invoke, explicitly or implicitly, the corresponding superclass constructor using super(). These are processed recursively following the same 5 steps.
    4. execute the instance initializers and instance variables for this class
    5. execute the remainder of the constructor body

```
   Example:

   class Point {
            int x, y;
            Point() { x = 1; y = 1; }
    }

   class ColoredPoint extends Point {
            int color = 0xFF00FF;
   }
   class Test {
            public static void main(String[] args) {
                    ColoredPoint cp = new ColoredPoint();
                    System.out.println(cp.color);
            }
    }

When the new instance of ColoredPoint is created:
1. first memory is allocated for the fields 'color' in
   ColoredPoint and then for the fields 'x, y' in Point
2. the fields are initialized to their default values
3. the no-arg ColoredPoint constructor is invoked.
   As none exists, the superclasses no-arg constructor
   is invoked. This is done implicitly ie the compiler
   added the default no-arg ctor at compile time
4. the Point ctor does not begin with this() so an
   invocation is made to the no-arg ctor for Object
   (Point's superclass)
5. any instance variable initializers of Object are
   invoked and the body of the no-arg ctor is executed
6. next, all the instance initializers for Point's instance
   variables are invoked and the body of the Point
   constructor is executed.
7. initializer for instance variables of ColoredPoint
```

```
    are invoked and the body of the ctor is executed.
```

JLS §12.4.1

- before a class is initialized it's direct superclass must be initialized but interfaces implemented by the class need not be initialized
- a reference to a **class field** only causes the initialization of it's class even if it is referred to by a subclass ie if 'taxi' is a static field in 'Super' class and is referenced by 'Sub.taxi'; only 'Super' is initialized; not 'Sub'
- the initialization of an Interface does not implicitly cause initialization of it's SuperInterfaces

JLS §8.8.5.1

- a constructor beginning with **this()** or **super()** can not use any class or superclass instance variables as an argument to a parameter

No argument constructor

- ONLY the no-arg constructor is called **implicitly** when new instances are created

```
New ClassB instance    // extends ClassA, has no ctor
    ClassA() ctor

New ClassD instance    // extends ClassA, has a no-arg ctor
    ClassA() ctor
    ClassD() ctor

New ClassF instance with no-args  // ClassF extends ClassE
                                  // which extends ClassA
    ClassA() ctor
    ClassE() ctor
    ClassF() ctor

// invoked with different ctor
New ClassF instance with parameter
    ClassA() ctor    // no-arg ctor's of superclasses implicitly
    ClassE() ctor    // called
    ClassF(String name) ctor
```

- if the constructor being invoked explicitly calls a superclass constructor then the superclass no-arg constructor is **not** implicitly invoked

```
ClassC extends ClassB which extends ClassA

// (no call to super(str) in ClassC(String str))
New ClassC instance created
    ClassA() ctor           // implicitly called
    ClassB() ctor           // implicitly called
    Hello

// (ClassC(String str) explicitly calls super(str))
New ClassC instance created
    ClassA() ctor    // implicitly called
    In ClassB        // explicit call;
                     // NO implicit call to ClassB()
    Hello
```

!!! Remember !!!

1. If NO constructor exists, the compiler will add a default no-arg constructor
2. The no-arg constructor of all superclasses in the hierarchy will be invoked and executed BEFORE the type constructor is executed UNLESS the type constructor explicitly calls another superclass constructor
3. There are NO IMPLICIT invocations to any other constructors

Also see

Sun Tech Tip: Constructor and Initialization Ordering

Example Code

- TestCtor.java
- TestCtor_1.java

	Encapsulation	Polymorphism	isA/hasA	Overloading	Overriding	Field Variables
	Initialization	Top-level Classes	Inner Classes	Static Nested Classes	Local Classes	Anonymous Classes

Overloading, Overriding, Runtime Types and Object Orientation - Top-level Classes

- a top-level class can be declared **public, final** or **abstract**
- or it can have no access modifier which defaults to **package** or **friendly** access

```
public class TestTopLevel {}
final  class FinalClass {}
abstract class AbstractClass {}
class PackageClass {}
```

- you can have more than one top-level class in a source code file; however, you can have **only one public** class in a source code file

Example Code

- TestTopLevel.java

Overloading, Overriding, Runtime Types and Object Orientation - Inner Classes

- Inner classes are non-static classes defined within other classes (JLS§8.1.2)

```
class Outer {
    class Inner {}      // class definition within the
                        // the body of class Outer
}
```

- the compiled class files for the above are: Outer.class and Outer$Inner.class
- the Inner class type is: Outer.Inner
- instances of inner classes can be created in a number of ways

```
Create an Outer class object:
    Outer o1 = new Outer();

Then create an Inner class object:
    Outer.Inner i1 = o1.new Inner();

Or, create the inner class directly:
    Outer.Inner i2 = new Outer().new Inner();

Or, create one from within the outer class constructor
    class Outer {
        Outer() {
            new Inner();
        }
    }
```

- inner classes may have no declared access modifier, defaulting the class access to **package**
- or, inner classes may be declared **public, protected, private, abstract, static** or **final**

```
class Outer {
    public class PublicInner{}
    protected class ProtectedInner {}
    private class PrivateInner{}
    abstract class AbstractInner {}
    final class FinalInner {}
    static class StaticInner {}
}
```

- each instance of a non-static inner class is associated with an instance of their outer class
- **static inner classes** are a special case. See Static Inner Classes
- inner classes may not declare static initializers or static members unless they are compile time constants ie static final var = value; (JLS§8.1.2)
- you cannot declare an interface as a member of an inner class; interfaces are never inner (JLS§8.1.2)
- inner classes may inherit static members (JLS§8.1.2)
- the inner class can access the variables and methods declared in the outer class
- to refer to a field or method in the outer class instance from within the inner class, use Outer.this.fldname

Example Code

- TestInner.java

	Encapsulation	Polymorphism	isA/hasA	Overloading	Overriding	Field Variables
	Initialization	Top-level Classes	Inner Classes	Static Nested Classes	Local Classes	Anonymous Classes

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Overloading, Overriding, Runtime Types and Object Orientation - Static Nested Classes

- a **static inner class** behaves like any **top-level** class except that its name and accessibility are defined by its enclosing class (JPL pg 50) ie use new Outer.Inner() when calling from another class
- formally called **top-level nested classes** (JPL pg 50)

Note
There is alot of confusion over the terminology involving 'static nested classes'. **They are not inner classes!** While the formal name, as stated in the Java Programming Language, Second Edition by Ken Arnold and James Gosling, is 'top-level nested', it is a bit of an oxymoron. Joshua Bloch, author of Effective Java, prefers the term 'static member class' which provides a clearer sense of how such classes are utilized.

```
class Outer {
    public static void main(String[] args) {
        int x = Inner.value;
    }

    static class Inner {
        static int value = 100;
    }
}
```

- they are not associated with an instance of their outer class ie you can create an Inner class object from within the Outer class using new Inner(); you do not need to create an Outer class object first as is required with non-static inner classes
- static inner classes can directly access static fields of the outer class but must use an instance of the outer class to access the outer classes instance fields

Example Code

- TestStaticInnerClass.java

	Encapsulation	Polymorphism	isA/hasA	Overloading	Overriding	Field Variables
	Initialization	Top-level Classes	Inner Classes	Static Nested Classes	Local Classes	Anonymous Classes

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Overloading, Overriding, Runtime Types and Object Orientation - Local Classes

- may be declared within a block of code

```
class Outer {
    void display() {
        class Local {
            // body of Local class
        }
    }
}
```

- the compiled name of the above Local class is: **Outer1Local.class**
- local inner classes are **not** class members and are not tied to an instance of the enclosing class
- as they are not class members, they **cannot** be instantiated outside of the code block in which they are declared by using the class as a reference ie new Outer.new Local(); won't work
- they **may not** be declared **private, public, protected,** or **static**. **May** be declared **final**
- they may access static and non-static members of the enclosing class
- they may only access **final** variables or parameters of the enclosing code block

Example Code

- TestLocalInner.java

	Encapsulation	Polymorphism	isA/hasA	Overloading	Overriding	Field Variables
	Initialization	Top-level Classes	Inner Classes	Static Nested Classes	Local Classes	Anonymous Classes

Overloading, Overriding, Runtime Types and Object Orientation - Anonymous Classes

- **anonymous** classes are classes which have no **name**
- they are declared and defined using the name of the class or interface they extend ie new Enumeration()
- no **modifiers**, **extends** or **implements** are allowed
- if any parameters are passed the superclass must have a corresponding constructor
- Anonymous classes do not have constructors of their own as constructors always take the name of the class and Anonymous classes have no name
- even though you cannot use an extends clause, you can extend the superclass by overriding methods

  you cannot 'overload' or 'add' new methods. See example code
- Once you create an anonymous class and override a method, that method is used until the class is unloaded
- they are most often used to implement an event listener interface or extend an adapter class

Example Code

- TestAnonymous.java
- TestAnonymousClass.java

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Garbage Collection Certification - Behaviour

- the Java garbage collector consists of three basic activities:
  1. monitors program objects to determine when they are no longer required
  2. informs selected objects that they should release any non-memory resources
  3. destroys objects and reclaims their memory resources
- the gc operates as a seperate asynchronous background thread that tracks all program objects
- an object ceases to be needed by a program when it is no longer **reachable**
- an object is reachable if a reference to the object exists in any variables of any executing code
- an object is subject to garbage collection when it can no longer be reached **but** it is not necessarily garbage collected immeadiately
- there are no guarantees as to when the gc will reclaim an object or the order in which objects are reclaimed
- **there is no way to tell if and when an object will be collected**, you can only tell when an object becomes **eligible** for garbage collection
- you can **request** garbage collection by calling one of

```
Runtime.getRuntime().gc() // no guarantee gc will run
System.gc() // no guarantee gc will run
```

- you can also request that the finalize() method be run for objects deemed eligible for collection but which have not yet had their finalization code run

```
Runtime.runFinalization()
System.runFinalization()
```

Also see

- Garbage Collection in Java
- Sun Tech Tip: Reference Objects
- Sun Tech Tip: Performance tip: Garbage Collection and setting to null

Traps

- a question that targets a specific object for garbage collection (can't be done)
- a question that presumes to **force** the gc to run (can only suggest it run)

Behaviour | Eligibility | finalize()

Garbage Collection Certification - Eligibility

- variables and objects are eligible for garbage collection when they become unreachable
- following summarizes the normal duration of a declared object or variable

Declaration	Duration
static field	as long as the class is loaded
instance field	for the life of the instance
Array components	as long as the array is referenced
Method parameters	until method execution ends
Constructor parameters	until the constructor execution ends
Exception handling parameters	until the catch clause completes execution
Local variables	in a **for-loop**, until the loop completes in a code-block, until the code block completes

- any variable set to **null** automatically becomes eligible for garbage collection

gc and the String pool
You may run across mock exam questions or discussions that state Strings created as part of the String pool are never garbage collected. This may well be true as garbage collection is **implementation dependent**. The certification exam will not contain questions that rely on specific implementations of any JVM; for exam purposes it is unlikely you will see any gc questions involving pooled Strings. Just remember **once an Object reference has been set to null it is eligible for garbage collection.**

Behaviour | Eligibility | finalize()

Garbage Collection Certification - finalize()

Syntax (JDK 1.3)

```
protected void finalize() throws Throwable {}
```

- every class inherits the **finalize()** method from **java.lang.Object**
- the method is called by the garbage collector when it determines no more references to the object exist
- the **Object** finalize method performs no actions but it may be overridden by any class
- normally it should be overridden to clean-up non-Java resources ie closing a file
- if overridding finalize() it is good programming practice to use a **try-catch-finally** statement and to always call **super.finalize()** (JPL pg 47-48). This is a saftey measure to ensure you do not inadvertently miss closing a resource used by the objects calling class

```
protected void finalize() throws Throwable {
    try {
        close();        // close open files
    } finally {
        super.finalize();
    }
}
```

- any **exception** thrown by **finalize()** during garbage collection halts the finalization but is otherwise **ignored**
- **finalize()** is **never run more than once** on any object

Also see

- Object finalization and cleanup - JavaWorld, June 1998
- Sun Tech Tip: Using finally versus finalize to guarantee quick resource cleanup

Example Code

- TestGC.java

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Declarations and Access Control - Access Modifiers

Modifier	Used with	Description
public	Classes Interfaces Constructors Inner Classes Methods Field variables	A Class or Interface may be accessed from outside its package. Constructors, Inner Classes, Methods and Field variables may be accessed from wherever their class is accessed.
protected	Constructors Inner Classes Methods Field variables	May be accessed by other classes in the same package or from any subclasses of the class in which they are declared.
private	Constructors Inner Classes Methods Field variables	May be accessed only from within the class in which they are declared.
no modifier	Classes Interfaces Constructors Inner Classes Methods Field variables	May only be accessed from within the package in which they are declared.

	Access Modifiers	Special Modifiers	this and super	Scope	Inheritance	Access Control

Declarations and Access Control - Special Modifiers

Modifier	Used with	Description
abstract	Classes Interfaces Methods	Declares a Class or Method that is incomplete. All Interfaces are implicitly abstract so the modifier is redundant. A Class which has an abstract Method must be declared abstract.
final	Classes Field variables Methods Method parameters Local variables	Indicates a definition is complete and cannot be changed. Classes may not be extended. Field variables may not be modified once a value is assigned. Methods cannot be overridden. Required for Method parameters and Local variables if they are to be used by an Inner Class. Note: A Class may not be both final and abstract.
native	Methods	Indicates a platform-specific method written in another language. Note: a method cannot be both native and abstract
static	Initializers Methods Variables	Indicates an initializer, method or variable belongs to a class vs an instance (object) of the class. Static initializers are processed once, when the class is loaded. Static methods are used to access static variables. They may not be used to access non-static variables unless they specify an instance of the class.
synchronized	Methods	Indicates a method acquires a lock on an object before it executes. Used to control access to objects shared by multiple threads.
transient	Variables	Indicates the variable is not part of the permanent state of an object and may not be serialized (written to a stream).
volatile	Variables	Indicates a variable may be changed by more than one thread. Each thread has it's own copy of a working variable. Volatile ensures the variable is compared to the master copy each time it is accessed.

Access Modifiers | Special Modifiers | this and super | Scope | Inheritance | Access Control

Declarations and Access Control - this and super

this (JLS §15.8.3)

- this is an Object-Oriented Programming (OOP) operator
- it is used to refer to the current instance of an object
- it can be used in the body of a class constructor to refer to the object being created
- it can be used in the body of an instance method or initializer to refer to the object whose method is being executed
- it **cannot** be used in a static method or initializer
- most commonly appears in constructors
- can be used to explicitly call another constructor
- when used in a constructor it **must** be the first statment in the constructors body

```
class Super {

    int x;
    int y;

    Super(){
        System.out.println("Super object being created.");
    }

    Super( int x, int y ) {
        this();             // call no-arg constructor
        this.x = x;
        this.y = y;
    }
}
```

super

- super is an Object-Oriented Programming (OOP) operator
- used to call a constructor declared in a classes superclass
- commonly used in constructors and to access hidden fields or invoke overridden methods in the superclass
- if used in a constructor, **must** be the first statment in constructor body

Remember
• Constructors are not inherited!

```
class Subclass extends Super {

    int w;

    Subclass(){
        this(0,0,0);        // call 3-param constructor
    }

    Subclass( int x, int y ) {
        this(x,y,0);        // call 3-param constructor
```

```
    }

    Subclass( int x, int y, int w ) {
        super(x,y);          // call superclass constructor
        this.w = w;
    }
}
```

Remember

• You **cannot** use this() and super() in the **same** constructor.
 	
`Subclass( int x, int y, int w) {`
`    this();`
`    super(x,y);          // compile-error`
`}`

Example Code

- *TestThisAndSuper.java*

TestThisAndSuper.java

```
class TestThisAndSuper {

    public static void main(String[] args) {

        Super sup = new Super(10,15);
        System.out.println("Super x: " + sup.x + " y: " + sup.y);

        Subclass sub = new Subclass(20,25,30);
        System.out.println("Sub x: " + sub.x + " y: " + sub.y + " w: " + sub.w);
    }
}

class Super {

    int x;
    int y;

    Super(){
        System.out.println("Super object being created.");
    }

    Super( int x, int y ) {
        this();              // call no-arg constructor
        this.x = x;
        this.y = y;
    }
}
```

```
class Subclass extends Super {

    int w;

    Subclass(){
        this(0,0,0);        // call 3-param constructor
    }

    Subclass( int x, int y ) {
        this(x,y,0);        // call 3-param constructor
    }

    Subclass( int x, int y, int w ) {
        super(x,y);         // call superclass constructor
        this.w = w;
    }
}
```

TestThisAndSuper.java

Declarations and Access Control - Scope

- names are used to identify entities declared in a program ie classes, methods, variables, parameters, etc
- each name or identifier occupies a particular *namespace*
- every declaration has a *scope*; the areas of a program from which it can be accessed by its simple name

Declaration	Scope (accessible from)
package	all compilation units within the package
import	all the classes and interfaces within the compilation unit (source code file)
class or interface	all other declarations within the same file
label	the statements immeadiately enclosed by the labeled statement ie if a loop is labelled, everything declared within the loop-construct has access to the label
member	the body of the class and anything declared within the class
parameter	the body of the method or constructor
local variable	the code block in which the declaration occurs
local class	the enclosing block including the local class body
local variable in a **for-loop** initializer	the body of the for-loop
parameter in a **catch** clause	the body of the catch clause

Order of searching for an identifier (JPL pg 113 and JLS §6.5)

- when a name (identifier) is used; the meaning, or scope, of it's name is searched for based on where it appears in the code starting with:

1. if used in a code block, for-loop, or in a catch clause, search is for a local variable within the enclosing construct
2. if in a method or constructor, searches for a matching parameter
3. search continues for a class or interface member, including inherited members
4. if its a nested type, searches enclosing block or class. If its a static type, only static members of enclosing blocks or classes are searched.
5. explicitly named imported types
6. other types declared in the same package
7. implicitly named imported types
8. packages on the host system

Shadowing (JLS §6.3.1)

- Because of the way identifiers are looked up; *shadowing declarations* can occur
- For example, a field declaration can be shadowed by a local variable declaration

```
class TestShadowing {
  static int x = 1;           // field variable

  public static void main(String[] args) {
      int x = 0;              // local variable
```

```
      System.out.println("x = " + x);
      System.out.println("TestShadowing.x = " + TestShadowing.x)
  }
}

Output:
    x = 0
    TestShadowing.x = 1
```

- because the identifier x is used within a code block main() a search is made for a declaration of x within the body of main(). As one is found, int x = 0, the simple identifier name x is assumed to be within scope as a local variable
- to access the field variable x, you must use its fully-qualified name TestShadowing.x

Note
• it was not necessary to instantiate an instance of the TestShadowing object to access the *static* field variable. If x had been an instance variable it would have been necessary to create a new instance of TestShadowing and use it's reference to access x

Hiding

- *Shadowing* is **not** the same as **hiding**
- hiding applies to members that would normally be inherited but are not because of a declaration of the same identifier in a subclass (JLS § 6.1.3)

```
class SuperA {
    int x = 10;
}

class SubA extends SuperA {
    int x = 20;           // hides x in superclass
}
```

- a **method** can hide a method in the superclass by **overriding** it

static Methods cannot be overridden

- a method **cannot** override a **static** method in the superclass; however, it can hide it by using the same declaration

```
class SuperA {
    static void method2() {
    }
}

class SubA extends SuperA() {
    void method2() {
        // declaration causes a compile-error
    }

    static void method2() {
        // compiles ok
    }
}
```

- static methods are *hidden* vs overridden as the JLS states they "cannot be overridden" so the compiler never compares subclass method declarations to static superclass method declarations.
- a **static** method in a subclass **cannot** hide an instance method in the superclass (JLS §8.4.6.2)

```
class SuperA {
    void method1() {
    }
}

class SubA extends SuperA() {

    static void method1() {
        // compile-error
    }
}
```

- a hidden method can be accessed by using super(), casting to the superclass or using the methods fully qualified name (JLS §8.4.6.2)

  ```
  ((SuperA)y).method2();  // cast to access hidden method
  ```
- instance variables can hide static and non-static variables in the superclass (JLS §8.4.6.1)

Obscuring (JLS §6.3.2)

- there may be times when a simple name could be interpreted as a variable, a type or a package
- based on the rules, a variable will be chosen before a type, and a type before a package
- in such situations a declaration is said to be **obscured**
- following naming conventions helps to avoid obscuring (see Naming conventions).

Also See

Tech Tip on Sun Site re: Shadowing, hiding, etc

Example Code

- TestShadowing.java
- TestHiding.java

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Declarations and Access Control - Inheritance

Package members (JLS §6.4.1)

- package members include all subpackages, and all top-level class and interface types declared in the package source files
- subpackages are determined by the host system.
- the java package always includes the subpackages lang and io and may include others
- no two distinct members of a package may have the same simple name

Class and Interface members (JLS §6.4.2 & §6.4.3)

- class members are fields, methods, classes or interfaces declared within the body of the class or inherited by the class
- **constructors** are **not** members
- a field or method can have the same simple name
- a member class or interface can have the same name as a field or method
- a class can have two different field variables with the same simple name if they are declared in different interfaces and are inherited **but** they can only be accessed using their fully-qualified names (compile-error: ambiguous results if simple names are used)
- a class can have two or more methods with the same simple-name if their signatures are different (overloading)
- a class may have a method with the same simple-name and signature as an inherited method. The original method is not inherited and the new member is said to *implement* it, if the original was *abstract* or *override* it

Array members (JLS §6.4.4)

- the public final field length which contains the number of components in the array (may be zero or any positive number)
- the public method clone which overrides the method clone in Object and throws no checked exceptions
- all members inherited from class Object

Access Modifiers | Special Modifiers | this and super | Scope | Inheritance | Access Control

Declarations and Access Control - Access Control

- accessibility is a static that can be determined at compile time
- it depends only on types and declaration modifiers
- accessibility effects inheritance of class members including **hiding** and **overriding**

Determining accessibility (JLS §6.6.1)

1. a **package** is always accessible
2. a **public** class or interface is accessible from any code as long as it's compilation unit is reachable by the code
3. an array is accessible if and only if it's element type is accessible
4. a member of a reference type (ie a class, interface, field or method of an object reference) or a class constructor is accessible only if the member was declared to allow access
   - declared public, all code can access the member
   - declared protected, accessible from other code within the same package or from subclasses in other packages **if** the outside code is involved in the implementation of the class. For example, the following produces a compile-error

```
package point;

class Point {
    protected int x, y;
}

package threepoint;
import point.Point;

class ThreePoints extends Point {
    protected int z;

    public void delta(Point p) {
       p.x += this.x; // compile-error: cannot access p.x
       p.y += this.y; // compile-error: cannot access p.y
    }
}

Even though ThreePoints is a subclass of Point, it cannot
access the protected fields in Point. The subclass must be
involved in the implementation of Point. The fact that the
code is within the body of a subclass is irrelevant. To the
compiler, Point is a type reference and p.x and p.y are
declared protected in the type Point.

If the parameter is changed to ThreePoints p the code will
compile as the type ThreePoints inherits the protected fields
x and y from Point.
```

- declared private, accessible only from within the body of the enclosing class; private members are **not** inherited

```
class Private1 {

    private boolean state;

    Private1() {
        System.out.println("Private1 state: " + state);
    }

}

class Private2 extends Private1 {

    Private2() {
        // compile-error
        System.out.println("Private1 state: " + state);
    }
}
```

- if no access was declared, default access applies ie accesible only from code within the same package

Example Code

- Point.java
- ThreePoint.java
- TestPrivateAccess.java

	Access Modifiers	Special Modifiers	this and super	Scope	Inheritance	Access Control

Flow Control and Exception Handling - Statements

Java Statements (JJ pg108)	
Statement	Description
empty	consists of **;** and performs no operation

block	group of statements enclosed in **{}**. Treated as a single statement when used with other statements
	`{ x +=y;`
	`  if( x < 10 )`
	`    return y;`
	`}`

declaration	declares a variable with a particular type and optionally assigns a value: int x = 10;

labeled	any statment may be labled using **identifier:**
	`startLoop:`
	`  for( ;; ){}`

assignment	evaluates an expression and assigns the result to a variable: x = y + z;
invocation	calls an object method: s.toString();
return	returns a value from a method call: return x;
Object creation	creates a new instance of a given class: String s = new String("abc");

if..else	selects between two alternatives
	`if( a==b )`
	`    // do this`
	`else`
	`   // do this`

switch	selects from various alternatives
	`switch( a ) {`
	`  case 1:`
	`  case 2:`
	`  case 3:`
	`  default:`

for	executes a set of statements for a defined number of iterations
	`for( int i=0; i<10; i++ ) {`
	` // do this`
	`}`

while	executes a block of statements while a condition is **true**
	<pre>while(!done) {
	    // do this
	}</pre>

do	executes a block of statments while a condition is **false**
	<pre>do {
	  // this
	}while(!done);</pre>

break	transfers the flow of control to a labeled block or out of an enclosing statement
continue	forces a loop to start the next iteration

try-catch-finally	catches and processes exception errors that occur during the execution of a given block of code
	<pre>try {
	      // some operation
	} catch (Exception e) {
	      // handle the exception
	} finally {
	      // do this
	}</pre>

throw	throw an exception

synchronized	gets a lock on an object and executes a statement block
	<pre>synchronized(obj){
	    obj.setProperty(x);
	}</pre>

	Statements	if	switch	for	while	do
	Labels	Exceptions	Handling Exceptions	try-catch-finally		

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Flow Control and Exception Handling - if...else Statement

Syntax

```
if( boolean expression ) {
    statement1;
    ...
    statementn;
}

if( boolean expression ) {
    statement1;
    ....
    statementn;
} else {
    statementa;
    ...
    statementz;
}
```

- the expression **must** be a boolean type
- curly braces are only required if there is more than one execution statement
- in the first form, the statements are only executed if the boolean expression evaluates to **true**
- in the second form, the first set of statements are executed if the boolean expression evaluates to **true**; otherwise, the statements following **else** are executed
- may be nested

```
if( x == y ) {
  // do this
} else if( x > y ) {        // nested 'if'
  // do this
} else {
  // do this
}
```

Example Code

- TestIf.java

Traps

- a non-boolean value used in the `if( )`
- using the assignment operator '=' vs '=='

	Statements	if	switch	for	while	do
	Labels	Exceptions	Handling Exceptions	try-catch-finally		

Flow Control and Exception Handling - switch Statement

Syntax (JLS §14.10)

```
switch( expression ) {
    case value1:
        statement1;
        break;
    case value2:
        statement2;
        break;
    case value3:
        statement3;
        break;
    ...
    case valuen:
        statement n;
        break;
    default:
        statements;
}
```

- transfers control depending on the value of an expression
- the type of the expression must be byte, char, short or int
- **case labels** must be **constant expressions** capable of being represented by the switch expression type

Watch for mismatching case constants!
`char c;` `switch( c ) { case 'a': case 'b': case "c": // String, not character! case 'd': }`

```
char c;

switch( c ) {
    case 'a':
    case 'b':
    case "c":       // String, not character!
    case 'd':
}
```

- no two case constant expressions may be the same
- the **default** case does **not** have to be at the end of the code block
- if no case matches the expression, the **default** case will be executed
- if **break** is omitted between case blocks the code will **fallthrough**, continuing to execute statements until a **break** statement or the end of the switch block is encountered

Example Code

- TestSwitch.java

Tips

- you do not have to have a default statement
- the default statement can appear anywhere in the construct, does not have to be last

Traps

- using an expression vs a value promotable to int
- duplicate case values
- case statements with wrong type
- missing break statements

	Statements	if	switch	for	while	do
	Labels	Exceptions	Handling Exceptions	try-catch-finally		

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Flow Control and Exception Handling - for Statement

Syntax (JLS §14.13)

```
for( initialization; boolean expression; iteration ) {
    statement(s);
}
```

- executes some **initialization** code, then repeatedly executes a **boolean expression** and some **iteration** code until the **expression** is **false**
- all three parts are optional ie the following examples are legal

```
for( intialization; ; )
for( ; expression; iteration )
for( ; ; iteration)
for( ; ; )                    // endless loop
```

Initialization

- initializes variables used within the loop
- if variables are declared within the loop, they are discarded after the loop completes
- For example, in the following code the initialization variable **i** is declared outside the for loop; so it's value is still available once the loop completes

```
int i;
for ( i=0; i<10 ; i++ ) {
    // do something
}
System.out.println("value of i: " + i );
```

- In the following code, x is declared and initialized inside the for-loop and is therefore only accessible within the loop

```
for ( int x=0; x<10 ; x++ ) {
    // do something
}
// compile-error, cannot resolve symbol: x
System.out.println("value of i: " + x );
```

- can be more than one initialization statement but the variables must either be declared outside the for-loop or the type for the variables must be declared at the beginning

```
Following compiles and runs ok:

    for( int x=10, y=0; x>y; x--, y++){
        System.out.println( x + "\t" + y);
    }

Following produces compile error

    int x;
```

```
for( x=10, int y=0; x>y; x--, y++){
    System.out.println( x + "\t" + y);
}
```

Boolean expression

- if the expression evaluates to **true** the loop continues; otherwise, the loop is exited and execution continues after the loop statment block

Iteration

- if the expression evaluates to true, the block statements are executed and then the **iteration** occurs
- if the expression evaluates to false, iteration does not occur

Break statement

- you can use a **break** statement to exit a for-loop at any time
- the break forces processing to the line following the for-loop statement block

```
for( i=0; i<10; i++ ){
    if( i==5 ) break;
}
// process continues here after the break
```

Continue statement

- you can use **continue** to force processing to the next loop iteration

```
for( i=0; i<10; i++){
    if( i==5)
        continue;   // skip printing 5
    else
        System.out.println(i);
}
```

Example Code

- TestFor.java
- Jaworski Exam Question 7, Chapter 5
- Jaworski Exam Question 8, Chapter 5

Tips

- all sections of the `for()` loop are optional

Traps

- attempting to access a variable declared in the initialization outside of the for-loop
- incorrect initialization expression
- non-boolean expression

	Labels	Exceptions	Handling Exceptions	try-catch-finally		

Flow Control and Exception Handling - while Statement

Syntax (JLS §14.11)

```
while( boolean expression ) {
    statement(s);
}
```

- executes **statement(s)** repeatedly until the value of **expression** is false
- if the expression is **false** the first time, the statements will never execute

```
int i = 1;

while( i>3 ) {
    System.out.println("This shouldn't print");
}
```

- both the **break** and **continue** statements can be used to alter the processing of a while loop

```
while( i < 10 ){
    if( i == 5 ) break;     // break out of loop
    System.out.println(i);
    i++;
}

while( i < 10 ){
    if( i==5 ) {
       i++;
       continue;   // force next loop
    }

    System.out.println(i);
    i++;
}
```

Example Code

- TestWhile.java

Traps

- non-boolean expression
- using '=' instead of '==' for boolean expression

	Statements	if	switch	for	while	do
	Labels	Exceptions	Handling Exceptions	try-catch-finally		

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Flow Control and Exception Handling - while Statement

Syntax (JLS §14.12)

```
do {
        statement(s);
} while(boolean expression);
```

- executes statement(s) until expression is **false**
- statement(s) are always executed at least once

```
do {
    System.out.println("Always executed at least once");
} while( false );
```

- **break** can be used to alter do-loop processing

```
do {
     if( i==6 ) break;          // exit loop
     System.out.println(i);
     i++;
}while( i < 10 );
```

- **continue** can be used to alter do-loop processing

```
do {

     if( i==6 ) {
         i--;
         continue;       // skip 6
     } else {
         System.out.println(i);
         i--;
     }

} while ( i >= 0);
```

Example Code

- TestDo.java

	Statements	if	switch	for	while	do
	Labels	Exceptions	Handling Exceptions	try-catch-finally		

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Flow Control and Exception Handling - Label Statements

Syntax

```
identifier:
```

- can be used with loops or statement blocks
- must precede the statement
- useful with **break** or **continue** which normally terminate or continue the innermost block

```
outer:
    for( i=0; i<10; i++ ){
        for( j=10; j>0; j--){
            if( j == 5 ) {
             break outer;       // exit entire loop
            }
        }
    }

Output: 0 5

outer:
    for( i=0; i<10; i++ ){
        for( j=10; j>0; j--) {
            if( j== 5 ) {
              continue outer;   // next iteration of i
            }
        }
    }

Output: 10 5
```

Note
• two or more statements can have the same name as long as one is not enclosed within the other

Example Code

- TestLabels.java
- Jaworski Exam Question 9, Chapter 5
- Jaworski Exam, Question 10, Chapter 5

	Statements	if	switch	for	while	do
	Labels	Exceptions	Handling Exceptions	try-catch-finally		

Flow Control and Exception Handling - Exceptions

Definition:

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions. (Sun tutorial: Handling Errors with Exceptions)

- exceptions provide a clean way to check for errors
- they are an explicit part of a methods contract
- exceptions are thrown at runtime if errors occur when a class is loaded or during method execution
- **runtime exceptions** are objects of the classes **java.lang.RuntimeException**, **java.lang.Error** or their subclasses
- runtime exceptions are also called **unchecked exceptions**
- code may also throw an exception using the **throw** statement
- these are **non-runtime** or **checked** exceptions
- any exceptions you create in your code should extend **java.lang.Exception** which implements the interface **java.lang.Throwable**
- you create your own exceptions to add useful data to an error message or, if you are interested in a particular error
- **both** forms of exceptions (checked and unchecked) may be caught and handled in exception-handling code
- an uncaught exception is caught by a **default handler** which halts execution and displays an error message
- exception handling is done using the **try-catch-finally** statment

	Statements	if	switch	for	while	do
	Labels	Exceptions	Handling Exceptions	try-catch-finally		

Flow Control and Exception Handling - Exception Handling

- methods must declare which checked exceptions they may throw in their **throws clause**

```
public void methodName throws Exception1, Exception2,()
```

- you do not have to include any checked exception which will be **caught** and handled within the method
- a method can throw multiple exceptions
- a method **can only throw** exceptions that have been declared in the throws clause
- an overriding method **cannot** throw any checked exceptions which are not part of the original methods throws clause
- the throws clause must also include any possible exceptions that can be thrown by the method
- if you invoke a method that has a checked exception in its throws clause you can
  1. catch and handle the exception
  2. catch it and throw one of the exceptions listed in the method throws clause
  3. declare the exception in your throws clause
- a method which does not have a throws clause may still throw **unchecked** exceptions or errors
- these exceptions and errors can occur at any time, in any code

```
Standard Unchecked Exceptions:

ArithmeticException                    IllegalTrheadStateException
ArrayStoreException                    IndexOutOfBoundsException
ClassCastException                     MissingResourceException
EmptyStackException                    NegativeArraySizeException
IllegalArgumentException               NoSuchElementException
IllegalMonitorStateException           NullPointerException
IllegalStateException                  NumberFormatException
                   SecurityException

Standard Unchecked Errors:

AbstractMethodError                    NoSuchFieldError
ClassFormatError                       NoSuchMethodError
ExceptionInInitializerError            OutOfMemoryError
IllegalAccessError                     StackOverflowError
IncompatibleClassChangeError           ThreadDeath
InstantiationError                     UnknownError
InternalError                          UnsatisfiedLinkError
LinkageError                           VerifyError
NoClassDefFoundError                   VirtualMachineError
```

- Static initializers, instance initializers, and class or variable initializers **must not** produce any checked exceptions
- exceptions are thrown using the **throw** statement

```
throw Expression;
throw new ExampleException();
```

	- or by invoking a method that throws an exception - the **expression** must be an instance of a **Throwable** object ie the exception class must implement Throwable

	Statements	if	switch	for	while	do
	Labels	Exceptions	Handling Exceptions	try-catch-finally		

Flow Control and Exception Handling - try-catch-finally

Syntax (JPL pg155)

```
try {
    statements;
} catch (exceptionType1 identifier1) {      // one or multiple
    statements;
} catch (exceptionType2 identifier2) {
    statements;
}
...
} finally {                                 // one or none
    statements;
}
```

- **must include** either one catch clause or a finally clause
- can be multiple catch clauses but **only one** finally clause
- the **try** statements are executed until an exception is thrown or it completes successfully
- a compile-error occurs if the code included in the **try** statement will never throw one of the caught *checked* exceptions (runtime exceptions *never* need to be caught)
- if an exception is thrown, each **catch** clause is inspected in turn for a type to which the exception can be assigned; be sure to order them from most specific to least specific
- when a match is found, the exception object is assigned to the identifier and the catch statements are executed
- if no matching catch clause is found, the exception percolates up to any outer try block that may handle it
- a catch clause may throw another exception
- if a **finally** clause is included, it's statements are executed after all other try-catch processing is complete
- the finally clause executes wether or not an exception is thrown or a break or continue are encountered

Note
• If a **catch** clause invokes **System.exit()** the finally clause WILL NOT execute.

Also see

- Sun Tutorial on Handling Errors with Exceptions
- Sun Tech Tip: Finally Clause

Example Code

- Example code from Java 2 Certification
- Some other exception handling code
- Jaworski Exam Question 14, Chapter 5
- Jaworski Exam Question 15, Chapter 5

	Statements	if	switch	for	while	do
	Labels	Exceptions	Handling Exceptions	try-catch-finally		

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Conversions

Implicit conversions (JPL pg 121)

- conversions which happen automatically
- any primitive type value can be converted to a type which supports a larger value (widening primitive conversion)
- implicit conversion occurs from integer to floating point values but not vice versa
- you can use an object of one type wherever a reference to one of it's supertypes is required ie you can reference up the class hierarchy but not down
- you can assign a null object reference to any object reference

Explicit conversion (JPL pg 122)

- when one type cannot be assigned to another type through implicit conversion you can use the cast operator

Identity Conversion (JLS §5.1.1)

- any type can be converted to it's own type
- **only** conversion allowed for **boolean** primitive type

Widening Primitive Conversion (JLS §5.1.2)

```
byte -> short -> int -> long -> float -> double
char -> int -> long -> float -> double
```

- widening conversions of integer types preserve the exact original value of the number
- runtime errors never occur as a result of widening conversion
- which is why widening conversion does not allow **byte** and **short** values to be converted to **char** as the *char* type is unsigned while *byte* and *short* are signed; the byte and short would lose information

```
byte  b =  126;
short s = 1000;
char  c;

c = b;  // compile error: possible loss of precision
c = s;  // compile error: possible loss of precision
```

- widening conversion of an int or long to a float may result in loss of precision however the new float value will be the correctly rounded equivalent of the original number
- the same applies when a long is widened to a double

Narrowing Primitive Converson (JLS §5.1.3)

```
double -> float -> long -> int > char -> short > byte
```

- narrowing primitive conversion may lose information about the overall magnitude of the number and may also lose precision
- runtime errors never occur as a result of narrowing conversion **because** compile time errors occur if you try it; need to use cast operator

- narrowing conversion loses all but the lowest bits (see Working with Binary, Octal and Hex numbers)
- narrowing from floating-point numbers to integer numbers occurs within the following minimum and maximum values (values are rounded-toward-zero)

```
long:  -9223372036854775808..9223372036854775807
int:   -2147483648..2147483647
short: 0..-1
char:  0..65535
byte:  0..-1
```

- if the floating-point value is **NaN** the result is an int or long value of zero

Widening Reference Conversion (JLS §5.1.4)

- convert from any class, interface or array reference to an Object reference
- convert from any class to any interface that it implements
- convert from any class, interface or array type to a null reference
- convert from any subinterface to any interface it extends
- from any array to type Cloneable or type java.io.Serializable
- from any array of references to an array of compatible reference types
- the above conversions never produce a runtime error or require special action

You can't instantiate an interface reference as interfaces are **always** abstract

```
    SuperInterface si = new SuperInterface();   // compile-error
```

Narrowing Reference Conversion (JLS §5.1.5)

- from Object to any other class, interface or array type
- from any superclass to a subclass
- from any non-final class to any interface as long as the class does **not** implement the interface
- from any interface to any non-final class
- from any interface to any final class providing the final class implements the interface
- from any interface to any other non-superinterface and providing neither interface contains methods with the same signature
- from any array of reference types to any other array of reference types as long as the types of each array are compatible under the Narrowing Reference rules

The above will be allowed at compile time but may throw a runtime ClassCastException if the types are not compatible

Summary
• **widening** conversions do not require casts and will not produce compile or runtime errors • **narrowing** conversions require explicit casts. Will compile ok but may result in runtime **ClassCastException** errors

String Conversions

- every other type, including null, can be converted to String

Method Conversion

- each argument is converted to the type of the method parameters
- widening conversion is implicit

- narrowing conversion is **not** implicit (values must be cast)

Forbidden Conversions (JLS §5.1.7)

- reference to primitive
- primitive to reference (excepting String)
- **null** to primitive
- reference or primitive to boolean
- boolean to reference (excepting String) or primitive
- one class to another unless they have a superclass/subclass relationship (excepting String)
- final class to interface unless the final class implements the interface
- class to array unless the class is Object
- array to any class other than Object or String
- array to any interface other than java.io.Serializable or Cloneable
- interface to interface if they contain methods with the same signature

Also see

Sun Tech Tip: Narrowing and Widening Conversions

Example Code

- TestConversions.java

Traps

- variables requiring narrowing conversion being passed to methods without using a *cast*
- assigning a typed **byte** or **short** variable to a **char** variable

	Conversions	Promotion	Overflow	Unary	Prefix	Arithmetic
	Bin/Hex/Octal	Bitwise	Shift	Comparison	Logical	Assignment
	Cast	Ternary	String	equals()	Precedence	Bit vs Logic
	Method Invocation					

Operators and Assignments - Cast Operator

- the cast operator (type) is used to convert numeric values from one numeric type to another or to change an object reference to a compatible type
- used to enable conversions that would normally be disallowed by the compiler

```
byte a = 1;
byte b = 2;

byte c = a + b;         // a and b are promoted to int
byte c = (byte)(a + b); // compiles ok
```

Casting with Object references (JLS §5.5, JJ pg 67)

- a reference of any object can be cast to a reference of type Object
- a reference to an object can be cast into a reference of type ClassName if the actual class of the object, when it was created, is a subclass of ClassName
- a reference to an object can be cast into a reference of type InterfaceName if the class of the object implements Interface, if the object is a subinterface of InterfaceName or if the object is an array type and InterfaceName is the Cloneable interface

If you cast up the class hierarchy you do not have to use the cast operator; if you are cast down the class hierarchy you must use the cast operator (BB pg 41)

However, the compiler uses the declared type to verify the correctness of each method call; which means you cannot invoke a subclass method from a superclass reference. (See post by Michael Ernest at JavaRanch)

- a cast may work at compile-time but fail at runtime if the actual class of the object cannot be converted legally
- while you can cast up and down the class hierarchy, you cannot cast sideways
- you can cast an object reference using String

```
Example from Java 2 Certification by Jamie Jaworski, pg 69

 String s1 = "abc";
 String s2 = "def";
 Vector v  = new Vector();
 v.add(s1);
 s2 = (String) v.elementAt(0); // cast allowed
 System.out.println();
 System.out.println("Value of s2: \t\t" + s2);

 output: abc
```

  Note: if the String cast is omitted, the type of **v.elementAt(0)** is an Object and a compile error (incompatible types) results.

- you cannot use String as a cast type for a primitive type
  String s = (String)x is invalid
  you can use String s = new Byte(x).toString();

```
X x = new X();
Y y = new Y();
Z z = new Z();
```

```
X xy = new Y();  // compiles ok (up the hierarchy)
X xz = new Z();  // compiles ok (up the hierarchy)
Y yz = new Z();  // incompatible type

Y y1 = new X();  // X is not a Y
Z z1 = new X();  // X is not a Z

X x1 =  y;       // compiles ok (y is subclass)
X x2 =  z;       // compiles ok (z is subclass)

Y y1 = (Y) x;    // compiles ok but produces runtime error
Z z1 = (Z) x;    // compiles ok but produces runtime error
Y y2 = (Y) x1;   // compiles and runs ok (x1 is type Y)
Z z2 = (Z) x2;   // compiles and runs ok (x2 is type Z)
Y y3 = (Y) z;    // inconvertible types (casts sideways)
Z z3 = (Z) y;    // inconvertible types (casts sideways)

Object o  = z;
Object o1 = (Y)o; // compiles ok but produces runtime error
```

The casts work at compile time since the cast variable could conceivably be of a compatible type; however, at runtime the type of the variable is known and if it cannot guarantee to implement the contract of the cast type a java.lang.CastClassException will be thrown.

Casting with arrays

- to cast an object reference to an array type reference, the object must be an array of a component type that is compatible with the component type of the array type reference

```
double arr[] = {1.5, 2.256, 3.59};
int    arr1[] = (int) arr;    // compile-error

X[] arrX = { new X(), new X(), new X() };
Y[] arrY = { new Y(), new Y(), new Y() };

arrX = arrY;     // compiles ok
```

Also see:

- Conversions

Example Code

- TestCast.java

Tips

- you cannot cast a primitive type to an object reference, or vice versa
- you cannot cast a boolean type to another primitive type

Traps

- result of an integer operation on byte or short types being assigned to a byte or short without an explicit cast

	Conversions	Promotion	Overflow	Unary	Prefix	Arithmetic
	Bin/Hex/Octal	Bitwise	Shift	Comparison	Logical	Assignment
	Cast	Ternary	String	equals()	Precedence	Bit vs Logic
	Method Invocation					

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Operators and Assignments - Numeric Promotion

Unary Numeric Promotion

- the Unary operators + and - when applied to byte, char or short numeric types result in the operand being automatically promoted to an int.(JLS §5.6.1)

```
Example producing compile error:

    byte b  = 5;          // assign byte value
    byte b1 = +b;         // found int, required byte
```

- unary promotion also applies for all shift operators. A **long** operator does not force the conversion of a left-hand **int** operator to **long**(JLS§5.6.1)

Binary Numeric Promotion

- when operands are of different types, automatic binary numeric promotion occurs with the smaller operand type being converted to the larger.
- the following rules are applied in the order given. (JLS §5.6.2)
  - if either operand is a double, the other operand is converted to double
  - otherwise, if one of the operands is a float, the other operand is converted to a float
  - otherwise, if one of the operands is a long, the other operand is converted to a long
  - otherwise, both operands are converted to int

```
Examples producing compile-errors:

    byte = byte + byte;          // found int, required byte
    int  = float + int;          // found float, required int
    long = float + long;         // found float, required long
    float = double + float;      // found double, required float
```

Remember to check the type of the variable to which results are assigned

Rules apply to following operators:

- Additive: + and -
- Multiplicative: *, /, and %
- Comparison: <, <=, >, and >=
- Equality: = and !=
- Bitwise: &, ^, and |

Special case for Ternary conditional operator (JLS §15.25)

- if one of the operands is byte and the other is short then the type of the expression is short

```
    byte = true ? byte : short // found short, required byte
```

- if one of the operands is a constant of type int and the other operand has a type of byte, short, or char and the value of the int operand is within the other type range, the type of the expression will be the type of the non-int operand.

```
short = true  ? short : 1000; // compiles and runs OK
short = false ? short : 1000; // compiles and runs OK
```

Example Code

- TestNumericPromotion.java

Traps

- expression assigning **byte** or **short** operations to a **byte** or **short** variable

	Conversions	Promotion	Overflow	Unary	Prefix	Arithmetic
	Bin/Hex/Octal	Bitwise	Shift	Comparison	Logical	Assignment
	Cast	Ternary	String	equals()	Precedence	Bit vs Logic
	Method Invocation					

Operators and Assignments - Overflow and Underflow

- an **overflow** results when a calculated value is larger than the number of bytes allowed for its type
- a **underflow** results when a calculated value is smaller than the number of bytes assigned to its type
- Java handles overflows by discarding the high-order-bytes that won't fit into the number of bytes allowed by its type (JJ pg 52)

```
 int n = 2000000000;
 System.out.println(n * n); // output: -1651507200

An int is 32-bits, the result of n*n is
4,000,000,000,000,000,000 which
needs 64-bits which in binary is:

    ---------- high-order bytes -------
    00110111 10000010 11011010 11001110

    ------- low order bytes -----------
    10011101 10010000 00000000 00000000

because an 32-bit cannot retain the number,
the 4 high-order bytes are
dropped leaving the four low-order bytes:

    10011101 10010000 00000000 00000000

which represent 1651507200 and since the right most bit
is a 1 the sign value is negative
```

- overflow or underflow conditions **never throw a runtime exception**; instead the sign of the result may not be the same as that expected in the mathematical result

You probably won't need to calculate overflows or underflows on the exam but should understand how they work.

(also see Working with Hex, Octal and Binary numbers)

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Operators and Assignments - Binary/Octal/Hex and Decimal Number Systems

Probably not directly required on exam but helpful when using bitwise and logical operators.

Decimal system

- the decimal number system we use every day is built on base ten 10_{10}
- it is based on 10 positions numbered 0 thru 9
- each position corresponds to a power of 10

```
   1024 = 1 x 10^3  -> 1 x 1000  = 1000
          0 x 10^2  -> 0 x  100  =  000
          2 x 10^1  -> 2 x   10  =   20
          4 x 10^0  -> 4 x    1  =    4
                                   ----
                                   1024
```

Binary system

- computer memory is based on the electrical representation of data
- each memory position is represented by a **bit** which can be either 'on' or 'off'. This makes it easier to represent computer memory using a base 2 number system rather than the base 10 decimal system.
- the binary system represents numbers by a series of 1's and 0's which correspond to 'on' and 'off' values
- a 1 represents an 'on' position, a 0, an 'off' position
- a **byte** is represented by 8 bits numbered 0 to 7 from left to right
- the leftmost bit is called the **high-order bit**, the right most bit, the **low-order bit**
- in the decimal system, each position corresponds to a power of 10, in the binary system, each position corresponds to a power of 2

```
01001001 = 0 x 2^7    -> 0 x 128  =  0
           1 x 2^6    -> 1 x  64  = 64
           0 x 2^5    -> 0 x  32  =  0
           0 x 2^4    -> 0 x  16  =  0
           1 x 2^3    -> 1 x   8  =  8
           0 x 2^2    -> 0 x   4  =  0
           0 x 2^1    -> 0 x   2  =  0
           1 x 2^0    -> 1 x   1  =  1
                                    --
                                    73
```

- the largest number which can be represented by a byte is 255 or
  128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255 or the bit pattern: 1111 1111
- the smallest number is 0 represented by the bit pattern: 0000 0000
- 0 to 255 gives 256 possible values

Two's-complement

- the two's complement method allows us to represent negative and positive values within the

0 to 256 bit positions

- in this system the numbers 0 thru 127 represent themselves and the numbers 128 to 256 represent **negative numbers** where 255 = -1, 254 = -2, 253 = -3, ...
- -1 is represented by 256 - 1 = 255, -127 is represented by 256-127 = 129, and -50 would be represented by 256 - 50 = 206
- the high-order bit (the 7th position) is reserved for the sign value of a number
- a 0 in the high order bit means 'the sign value is set to positive'
- a 1 in the high-order bit means 'the sign value is set to negative'

```
01111111 =   0 +  64 + 32 + 16 + 8 + 4 + 2 + 1  =   127
10000000 = 128 or set sign negative
11111111 =   0 -  64 - 32 - 16 - 8 - 4 - 2 - 1  =  -127
```

- larger numbers are represented by increasing the number of bits in a memory block
- this is done in multiples of 8 hence 16-bit, 32-bit and 64-bit memory
- 16-bit memory can store numbers up to 2^{16}-1, 32-bit, 2^{32}-1, 64-bit, 2^{64}-1
- if signed numbers are being used, the left-most bit still represents the sign
- so, 16-bits allows us to store 0 to 65,535 positions (2^{16} - 1), 32-bits, 0 to 4,294,967,295 (2^{32} - 1)
- using two's-complement arithmetic with 32-bit memory, subtract the negative number from 65,536 to find it's positive complement ie -336 would be represented by 65536 - 336 = 65200

Octal system

- uses base 8
- octal digits are represented by 0 thru 7
- each position is a power of 8
- each octal number can be represented by 3 binary digits
- $2^2+2^1+2^0$ = 4+2+1 = 7

Decimal	Octal	Binary
0	0	000
1	1	001
2	2	010
3	3	011
4	4	100
5	5	101
6	6	110
7	7	111

- to convert from Octal to Binary just replace the octal digit with the corresponding binary pattern

```
    Octal: 17       Binary: 001 111
```

- to convert from Binary to Octal just replace the binary pattern with the corresponding octal digit

```
    Binary: 111 010 Octal:  72
```

Hexidecimal system

- the hexidecimal system uses a base of 16
- hexidecimal digits are represented by 0 thru 9 and the letters A,B,C,D,E,F
- one hexidecimal digit corresponds to a four-digit binary number

- $2^3+2^2+2^1+2^0 = 8 + 4 + 2 + 1 = 15$

Decimal	Hex	Binary	Decimal	Hex	Binary
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	10	A	1010
3	3	0011	11	B	1011
4	4	0100	12	C	1100
5	5	0101	13	D	1101
6	6	0110	14	E	1110
7	7	0111	15	F	1111

- this makes it easy to convert a number from binary to hex (just replace the binary pattern with the hex digits) or from hex to binary (replace the hex digit with the binary pattern)

```
Binary: 0000 1111                  -> Hex:     0x0F
Binary: 1011 0011 0000 0010        -> Hex:     0xB302

Hex:    0xA0FF                     -> Binary:  1010 0000 1111 1111
Hex:    0xF075                     -> Binary:  1111 0000 0111 0101
```

Converting between number systems

- to convert a decimal number to a Hex, Octal or Binary number divide by the required base, the resulting remainders, in reverse order represent the required value

```
Convert Decimal 49 to Binary 49:

    49 / 2 = 24      remainder: 1      ( 49 - 2*24 = 49 - 48 = 1)
    24 / 2 = 12      remainder: 0      ( 24 - 2*12 = 24 - 24 = 0)
    12 / 2 =  6      remainder: 0      ( 12 - 2* 6 = 12 - 12 = 0)
     6 / 2 =  3      remainder: 0      (  6 - 2* 3 =  6 -  6 = 0)
     3 / 2 =  1      remainder: 1      (  3 - 2* 1 =  3 -  2 = 1)
     1 / 2 =  0      remainder: 1      (  1 - 2* 0 =  1 -  0 = 1)
```

Proof: $110001 = 32+16+0+0+0+0+1 = 49_{10}$

```
Convert Decimal 49 to Octal 49:

    49 / 8 = 6      remainder:  1    ( 49 - 8*6 = 49 - 48 = 1)
     6 / 8 = 0      remainder:  6    (  6 - 8*0 =  6 -  0 = 6)
```

Proof: $61_8 = (6 * 8^1) + (1 * 8^0) = 48 + 1 = 49_{10}$

```
Convert Decimal 49 to Hexidecimal 49:

    49 / 16 = 3     remainder:  1     ( 49 - 16*3 = 49 - 48 = 1)
     3 / 16 = 0     remainder:  3     (  3 - 16*0 =  3 -  0 = 3)
```

Proof: $31_{16} = (3*16^1) + (1*16^0) = 48 + 1 = 49_{10}$

- in Java octal numbers are represented by 3 digits beginning with a zero, so 61_8 would be written as 061
- Hex numbers are always represented by 4 digits preceeded by 0x, so 31_{16} would be written as 0x0031
- when converting large decimal numbers to binary, the simplest method is to convert to Hex and then to binary

```
Convert 4823 to Hex:

4823 / 16 = 301 remainder:  7  (4823 - 16*301 = 4823 - 4816 =  7)
 301 / 16 =  18 remainder: 13  ( 301 - 16* 18 =  301 -  288 = 13)
  18 / 16 =   1 remainder:  2  (  18 - 16*  1 =   18 -   16 =  2)
   1 / 16 =   0 remainder:  1  (   1 - 16*  0 =    1 -    0 =  1)

 Hex value: 0x12D7
 Hex value converted to binary: 0001 0010 1101 0111
```

- when converting large binary numbers, the simplest method is to convert to Hex and then to decimal

```
Convert Binary 0001 0010 1101 0111 to decimal

    0001    1    1 * 16^3 = 4096
    0010    2    2 * 16^2 =  512
    1101    D   13 * 16^1 =  208
    0111    7    7 * 16^0 =    7
                           ----
                           4823
```

Study aids

- If you have Windows 95 you can use the Calculator in the Scientific mode (Start->Programs->Accessories->Calculator) to check results of decimal to hex, binary, and octal conversions.
- You can also use the Java Integer wrapper class to output binary, hex and octal strings. Example: System.out.println(Integer.toBinaryString(-29));
- Marcus Greene has a great applet that lets you play around with bit-shifting at http://www.software.u-net.com/applets/BitShift/BitShiftAr.html

References:

- *C: Step-by-Step* by Mitchell Waite and Stephen Prata, SAMS, 1991
- Hexidecimal and Octal Notation

	Conversions	Promotion	Overflow	Unary	Prefix	Arithmetic
	Bin/Hex/Octal	Bitwise	Shift	Comparison	Logical	Assignment
	Cast	Ternary	String	equals()	Precedence	Bit vs Logic
	Method Invocation					

Operators and Assignments - Unary Operators

- operate on a single operand
- the unary ~ , + and **-** operators can only be applied to numeric primitive types
- the unary ! (logical complement) can only be applied to a boolean type
- rules of unary numeric promotion apply

Unary ~ Bitwise complement (inversion) (JLS §15.15.5)

- only used with integer values
- inverts the bits ie a 0-bit becomes 1-bit and vice versa
- in all cases **~x equals (-x)-1**

```
byte b0 = 7;              // binary: 0000 0111
byte b1 = ~b0;            // binary: 1111 1000 ( -8 )

~7     = -7 -1       = -8
~3578  = -3578-1     = -3579
~-1234 = -(-1234)-1  =  1233
```

Unary ! Logical complement (JLS §15.15.6)

- returns the logical complement of a boolean type

```
!(false) = true;    // complement of 'false' is 'true'
!(true)  = false;   // complement of 'true' is 'false'
```

Unary + operator (JLS §15.15.3)

- the result of the unary + operator is a **value** not a variable

```
byte b = +5;        // result: 5
```

The unary plus (+) operator has no effect on the sign of a value; it is included for symmetry only and to allow the declaration of constants
ie **MIN_VALUE = +2.0;** (JPL pg 128)

Unary - operator (JLS §15.15.4)

- for **integers** negation effect is the same as subtraction from zero
- two's complement is used for integers so for all values of x, **-**x equals (~x)+1

```
byte b;
b = -5;            // result: -5
b = (~5) + 1;      // result: -5
```

- negation of the maximum negative int or long value results in the same number. An overflow occurs but no exception is thrown.

```
int i;
long l = 0L;
i = -(-2147483648);          // result: -2147483648
```

```
l = -(-9223372036854775808L) // result: -9223372036854775808;
```

- for **floating-point** negation is **not** the same as subtraction from zero
- the unary **(-)** operator merely negates the sign of the value

```
double d = 0D;
d = -(15.63);                    // result: -15.63
d = -(-15.63);                   // result:  15.63
```

Example Code

- TestUnaryQuestions.java

	Conversions	Promotion	Overflow	Unary	Prefix	Arithmetic
	Bin/Hex/Octal	Bitwise	Shift	Comparison	Logical	Assignment
	Cast	Ternary	String	equals()	Precedence	Bit vs Logic
	Method Invocation					

Operators and Assignments - Prefix and Postfix Operators

- the operators ++ and -- are used to increment or decrement a variable value by 1
- **binary numeric promotion** is applied on both the 1 and the variable value before the addition or subtraction occurs (ie at a minimum both values are promoted to an int) BUT the type of the expression is the **type of the variable** so narrowing conversion is applied if necessary ie if the original variable is a **byte, short,** or **char** the result is narrowed to the corresponding type

```
byte b = 2;
byte b1;

b1 = ++b;  // no error although promotion occurs
b = 127;
b1 = ++b;  // result: -128 (no error as fits within byte type)
```

- the expression has the same type as the variable
- they can appear before a variable (prefix) or after a variable (postfix)
- cannot be used with final variables

Prefix (++x) and Postfix (x++) increment operators

- 1 is added to the value of the variable and the result is stored back in the variable
- both operators have the same effect as x = x + 1;

```
int x;

x = 0;
++x;                        // result: 1

x = 0;
x++;                        // result: 1
```

Prefix (--x) and Postfix (x--) decrement operators

- 1 is subtracted from the value of the variable and the result is stored back in the variable
- both operators have the same effect as x = x - 1

```
int x;

x = 0;
--x;                        // result: -1

x = 0;
x--;                        // result: -1
```

Using prefix and postfix operators in expressions

- when a **prefix** expression (++x or **--**x) is used as part of an expression, the value returned is the value calculated **after** the prefix operator is applied

```
int x = 0;
int y = 0;
y = ++x;            // result: y=1, x=1

x is incremented by 1 and the result is assigned to y
```

- when a **postfix** expression (x++ or x**--**) is used as part of an expression, the value returned is the value calculated **before** the postfix operator is applied

```
int x = 0;
int y = 0;
y = x++;            // result: y=0, x=1

original value of x is stored, x is incremented,
original value of x is assigned to y
```

- when using the postfix form of the operators do not try constructs like

```
int x = 0;
x = x++;   // result: 0, x is not incremented

original value of x is saved (x0rig)
x is incremented
x0rig is assigned to x
therefore, x will always equal original value
```

Effect on 'char' type

- both the prefix and postfix forms may be used on **char** types

```
char c = 'a';
c++;                // result: b
--c;                // result: a
```

- the type of the variable **does not change**

Example Code

- TestPrefixAndPostfix.java

Tips

- postfix/prefix operators have the highest level of precedence
- remember that when the postfix operator is used in an expression, the current value of the variable is used

	Conversions	Promotion	Overflow	Unary	Prefix	Arithmetic
	Bin/Hex/Octal	Bitwise	Shift	Comparison	Logical	Assignment
	Cast	Ternary	String	equals()	Precedence	Bit vs Logic
	Method Invocation					

Operators and Assignments - Arithmetic Operators

Additive operators (JLS §15.18)

- + and -
- have the same precedence and are left-associative
- operands must be primitive numeric types (see exception for String and +) or compile error occurs

Multiplicative operators (JLS §15.17)

- *, /, %
- have the same precedence and are left-associative
- operands must be primitive numeric types or compile error occurs;

Integer Division and Division by Zero (JJ pg 50, JLS §15.17.2)

- integer division rounds towards 0; ie result is truncated

```
    10 / 3 = 3;     // truncated result
```

- if the value of the divisor in **integer division** is 0 an ArithmeticException is thrown

```
    10 / 0          // runtime error: ArithmeticException
```

- if the value of the divisor in **floating-point division** is 0 **no exception** is thrown; the value of the results are as follows:
  - division of a positive floating-point value: POSITIVE_INFINITY
  - division of a negative floating-point value: NEGATIVE_INFINITY
  - division of a floating-point value by -0: POSITIVE_INFINITY

```
 10.34 /  0          // result:  Infinity
-10.34 /  0          // result: -Infinity
 10.34 / -0          // result:  Infinity
  0    /  0          // result:  NaN (Not a number)
```

Modulo operations (JLS §15.17.3)

- the modulo operator % is also called the **remainder operator** as it returns the remainder, or fractional part, of a division operation
- x % y is equivalent to x - ((int) (x/y) * y)
- can be used with both integer and floating-point numbers
- following rules apply as to the sign of the result:
  - result is negative if the divdend is negative
  - result is positive if the divdend is positive
  - if the divisor is zero, a runtime ArithmeticException is thrown
  - if the dividend is a floating-point value and the divisor is zero, no exception is thrown and the result is NaN

```
    5 % 3  =  2
```

```
  -5 % 3  = -2
 5.0 % 3  =  2.0
-5.0 % 3  = -2.0
 5.0 % 0  = NaN              // not a number
```

Also see:

- Binary numeric promotion
- Overflow and underflow
- Sun Tech Tip: Division by Zero

Example Code

TestArithmetic.java

Traps

- floating point operation throwing an ArithmeticException

	Conversions	Promotion	Overflow	Unary	Prefix	Arithmetic
	Bin/Hex/Octal	Bitwise	Shift	Comparison	Logical	Assignment
	Cast	Ternary	String	equals()	Precedence	Bit vs Logic
	Method Invocation					

Operators and Assignments - String Operators

- the + and += operators both work on Strings
- operators actually signfy **concatenation**
- the result of the operation is a **new** string
- Strings are **objects**, not primitive types, and are **read-only** and **immutable**; the contents never change
- String variables store **references to a string object** NOT the string itself

```
String str  = "Hello";
String str1 = "Universe!";
String str2 = str + str1; // join the two strings together

String str3 = "";
       str3 += str; // += only works with an initialized var
String str4 = str2;
```

- in the above code a **reference** to the string "Hello" is stored in the variable **str**
- a **reference** to the string "Universe!" is stored in the variable **str1**
- a **reference** to a **new** string "Hello Universe!" is stored in the variable **str2**
- the **reference** for a **new string** "Hello" is stored in variable **str3**

```
str3 == str  // false (ref to different String objects)
```

- the **reference** for **str2** is stored in variable **str4**

```
  str4 == str2  // true (references are the same)
```

Where it can get confusing

- the String class creates a **pool** of Strings
- when you create a String by using the **new** operator or by using the + and += operators (the string is computed at runtime) you are implicitly telling the compiler to **create a new String object**
- when you create a String by assigning a string literal the compiler searches the existing string pool for an exact match. If it finds one, a new string is NOT created. Instead the variable is assigned a reference to the existing pooled string.

```
String str5 = "Hello Universe!"; // created in the string pool
String str6 = "Hello Universe!";

str5 == str2  // false (str2 is not part of the pool, created
                // using '+' operator)
str5 == str6  // true (matched an existing string found
              // in the pool)
```

- to actually compare the **contents** of String objects use the String method **equals()**

```
str5.equals(str2);  // true (both objects hold the same string
                    // characters)
```

Strings and primitive types

- by the rules of String Conversion (see Conversion) any type can be converted to a string
- this includes the primitive types
- for primitive types, conversion occurs by the compiler calling **Type.toString(x)** behind the scenes.

```
int x = 10;
System.out.println("Result: " + x);

is the same as

System.out.println("Result: " + (Integer.toString(x)) );
```

Also see:

- String literals
- Sun Tech Tip: Interning Strings

Example Code

- TestStringOperators.java

Tips

- String operations whose result does not alter the original string (ie calling toUpperCase() on a String that is already in uppercase) return the original string reference; otherwise they return a reference to a **new** String
- Strings are **immutable**; the original String value **can never be changed**

Traps

- using == to compare the contents of two different String objects

	Conversions	Promotion	Overflow	Unary	Prefix	Arithmetic
	Bin/Hex/Octal	Bitwise	Shift	Comparison	Logical	Assignment
	Cast	Ternary	String	equals()	Precedence	Bit vs Logic
	Method Invocation					

Operators and Assignments - Bitwise Operators

- & AND, | OR, ^ exclusive OR
- used for operations on integer and boolean values (see logical bitwise operators)
- results are calculated bit-by-bit
- binary numeric promotion rules apply
- left associative
- order of precedence: &, ^, |

& AND operator

- returns a 1 if corresponding bits in both operands have a 1, otherwise returns a 0

```
 63     = 00000000 00000000 00000000 00111111
252     = 00000000 00000000 00000000 11111100
          -----------------------------------
          00000000 00000000 00000000 00111100 -> 60
```

| OR operator

- returns a 0 if corresponding bits in both operands are 0, otherwise returns a 1

```
 63     = 00000000 00000000 00000000 00111111
252     = 00000000 00000000 00000000 11111100
          -----------------------------------
          00000000 00000000 00000000 11111111 -> 255
```

^ exclusive OR

- returns a 0 if the corresponding bits of both operands are both 0 or both 1, otherwise returns a 1

```
 63     = 00000000 00000000 00000000 00111111
252     = 00000000 00000000 00000000 11111100
          -----------------------------------
          00000000 00000000 00000000 11000011 -> 195
```

Also see

- Unary bitwise complement operator ~
- Logical (boolean) bitwise operators

Example Code

- TestBitwise.jsva

Tips

- precdence order is: **&** ^ |

	Conversions	Promotion	Overflow	Unary	Prefix	Arithmetic
	Bin/Hex/Octal	Bitwise	Shift	Comparison	Logical	Assignment
	Cast	Ternary	String	equals()	Precedence	Bit vs Logic
	Method Invocation					

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Operators and Assignments - Logical Operators

Boolean logical operators & | and ^

- when both operands are **boolean** the result of the bitwise operators & | and ^ is a **boolean**
- **& - true** if **both** operands are **true**, otherwise false
- ^ - **true** if both operands are different, otherwise false
- | - **false** if **both** operands are **false**, otherwise, true

```
true & true   = true;       // both operands true
true & false  = false;      // one operand is false

true ^ false  = true;       // both operands are different
true ^ true   = false;      // both operands are the same

true  | false = true;       // one operand is true
false | false = false;      // both operands are false
```

Conditional AND Operator &&

- both operands **must be boolean**
- result is a boolean
- returns **true** if both operands are true, otherwise false
- evaluates the right-hand operand only if the left-hand operand is **true**

```
true && true  = true;      // both operands evaluated
false && true = false;     // only left-operand evaluated
```

Conditional OR Operator ||

- both operands must be boolean
- result is a boolean
- returns **true** if one of the operands is true
- evaluates the right-hand operand only if the left-hand operands is **false**

```
false || true  = true;    // both operands evaluated
false || false = false;
true  || false = true;    // only lef-operand evaluated
true  || true  = true;
```

The conditional operators are also referred to as **short-circuit** operators.

Also see

- Integer Bitwise operators

Example Code

- TestLogical.java

- Home
- SCJP2 Study Notes
- Language Fundamentals
- Operators and Assignments
- Flow Control and Exceptions
- Declarations and Access Control
- Garbage Collection
- Overloading and Overriding
- Threads
- The java.lang Package
- The java.util Package
- The java.awt Package
- The java.io Package
- References
- Miscellaneous Notes
- Tips & Traps
- Mock Exams
- Case Studies
- SCJA Notes
- SCJD Notes
- Projects
- Favourite Links
- About
- Feedback

Operators and Assignments - Shift Operators

- << left-shift, >> right-shift, >>> unsigned right-shift
- only used on integer values
- **binary numeric promotion** is **not** performed on the operands; instead **unary promotion** is performed on each operand separately (JLS §15.19)
- both operands are individually promoted to int if their type is byte, short or char
- a long shift operator does not force a left-hand int value promotion to long (JLS§5.6.1)
- left-associative
- left-hand operator represents the number to be shifted
- right-hand operator specifies the shift distance

```
    value << 2      // 2 is the distance to be shifted
```

- when the value to be shifted (left-operand) is an int, only the last 5 digits of the right-hand operand are used to perform the shift. The actual size of the shift is the value of the right-hand operand masked by 31 (0x1f). ie the shift distance is always between 0 and 31 (if shift value is > 32 shift is 32%value)

```
35              00000000 00000000 00000000 00100011
31 -> 0x1f      00000000 00000000 00000000 00011111
&               -----------------------------------
Shift value     00000000 00000000 00000000 00000011   -> 3

-29             11111111 11111111 11111111 11100011
31 -> 0x1f      00000000 00000000 00000000 00011111
&               -----------------------------------
Shift value     00000000 00000000 00000000 00000011   -> 3
```

- when the value to be shifted (left-operand) is a long, only the last 6 digits of the right-hand operand are used to perform the shift. The actual size of the shift is the value of the right-hand operand masked by 63 (0x3D) ie the shift distance is always between 0 and 63 (if shift value is greater than 64 shift is 64%value)
- the shift occurs at runtime on a bit-by-bit basis

Left-shift << (JLS §15.19)

- bits are shifted to the left based on the value of the right-operand
- new right hand bits are zero filled
- equivalent to left-operand times two to the power of the right-operand
  For example, 16 << 5 = 16 * 2^5 = 512

```
Decimal 16           00000000000000000000000000010000

Left-shift 5     00000000000000000000000000010000
  fill right     00000000000000000000000000010000000000
  discard left       00000000000000000000001000000000
```

- the sign-bit is shifted to the left as well, so it can be dropped off or a different sign can replace it

Right-shift >> (JLS §15.19)

- bits are shifted to the right based on value of right-operand

- new left hand bits are filled with the value of the left-operand high-order bit therefore the **sign** of the left-hand operator is **always** retained
- for non-negative integers, a right-shift is equivalent to dividing the left-hand operator by two to the power of the right-hand operator
  For example: 16 >> 2 = 16 / 2^2 = 4

```
Decimal 16        00000000000000000000000000010000

Right-shift 2       00000000000000000000000000010000
  fill left       00000000000000000000000000000100
  discard right   00000000000000000000000000000100  -> Decimal 4

Decimal -16       11111111111111111111111111110000

Right-shift 2       11111111111111111111111111110000
  fill left       1111111111111111111111111111110000
  discard right   11111111111111111111111111111100  -> Decimal -4
```

Unsigned right-shift >>> (JLS §15.19)

- identical to the right-shift operator only the left-bits are zero filled
- because the left-operand high-order bit is not retained, the sign value can change
- if the left-hand operand is positive, the result is the same as a right-shift
- if the left-hand operand is negative, the result is equivalent to the left-hand operand right-shifted by the number indicated by the right-hand operand **plus** two left-shifted by the inverted value of the right-hand operand
  For example: -16 >>> 2 = (-16 >> 2) + (2 << ~2) = 1,073,741,820

```
Decimal 16        00000000000000000000000000010000

Right-shift 2       00000000000000000000000000010000
  fill left       00000000000000000000000000000100
  discard right   00000000000000000000000000000100  -> Decimal 4

Decimal -16       11111111111111111111111111110000

>>> 2               11111111111111111111111111110000
  fill left       0011111111111111111111111111110000
  discard right   00111111111111111111111111111100
```

Don't panic that it will take you forever to convert a shift question on the exam. You probably won't get more than one or two questions and they'll likely involve numbers under 20.

Also see

- [Working with Binary, Hex, and Octal numbers]

Example Code

- [TestShift.java]

	Conversions	Promotion	Overflow	Unary	Prefix	Arithmetic
	Bin/Hex/Octal	Bitwise	Shift	Comparison	Logical	Assignment
	Cast	Ternary	String	equals()	Precedence	Bit vs Logic

Operators and Assignments - Comparison Operators

- used to compare primitive types and object references
- organized into three subgroups: relational, equality and the instanceof operator

Relational operators (< <= > >=) (JLS §15.20.1)

- produce a boolean result
- work with integers and floating-point numbers
- binary numeric promotion rules apply for numeric types
- any relational expression with NaN is false
- positive and negative zero are considered equal therefore -0.0 < 0.0 is false and -0.0 <= 0.0 is true

This is not true for Math.min() and Math.max(), which treats -0.0 as being strictly smaller than 0.0

- results, otherwise, are the same as their mathematical equivalents

```
               Less than: 5 <  6              true
   Less than or equal to: 5 <= 5              true
            Greater than: 5 >  6              false
Greater than or equal to: 5 >= 5              true

               Less than: -0.0 < 0.0          false
   Less than or equal to: -0.0 <= 0.0         true
            Greater than: 5 >  NaN            false
```

Equality operators (== !=) (JLS § 15.21)

- produce a boolean result
- lower precedence than the relational operators
- are used to compare primitive types, including boolean, and object references
- binary numeric promotion rules apply for numeric types
- if either operand is a Nan the result is false for == but true for !=
- -0.0 and 0.0 are considered **equal**
- if the operands are object references, the result is true if both refer to the same object or array or if both are null
- if the operands are String objects, the result is false unless they refer to the same String object, even if the two objects contain the same characters (to compare the characters in a String object use the String.equals() method) (see String Literals)

```
   Equals: 5 == 5.0           true
Not Equal: 5 != 5.0           false
   Equals: arr1 == arr2       false [different array objects]
   Equals: arr1 == arr3       true [ref to same array object]
Not Equal: arr1 != arr2       true
Not Equal: arr1 != arr3       false
   Equals: s1 == s2           true [same literal]
   Equals: s1 == s3           true [same object reference]
```

```
    Equals: s1 == s4            false [s4 is new object]
```

instanceof Type Comparison Operator (JLS §15.20.2, JJ pg 60)

- left-operand must be a reference object or null; cannot use primitive types
- right-operand must be a Class, Interface name or Array type
- determines if the left-operand is an instance of the class, interface or array type specified by the right-operand
- returns the **boolean** value true if:
  - left-operand is a class or subclass of the right-operand
  - left-operand is an interface or subinterface of the right-operand
  - left-operand is an array of the same class, subclass or interface, subinterface of the right-operand array type

```
arr instanceof String[]    -> true    // arr = array of Strings
myNull instanceof Object   -> false   // null is not an object
arr1 instanceof int[]      -> true    // arr1 is an arry of int
```

Example Code

- TestComparison.java

	Conversions	Promotion	Overflow	Unary	Prefix	Arithmetic
	Bin/Hex/Octal	Bitwise	Shift	Comparison	Logical	Assignment
	Cast	Ternary	String	equals()	Precedence	Bit vs Logic
	Method Invocation					

Operators and Assignments - Assignment Operators

- 12 assignment operators:
  = *= /= %= += -= <<= >>= >>>= &= ^= |=
- all are right-associative ie a=b=c groups as a=(b=c) vs (a=b)=c **except** the simple assignment operator = which is left-associative
  eg a+b+c = (a+b)+c
- all are used with primitive data types except = and += which can be used with Strings
- all operators of the form op = cast their result to the type of the left-operand
- there is no implicit cast with the simple assignment operator =
- in all cases, x oper= y is equivalent to x = x oper y

```
For Example:

    x += y      is equivalent to    x = x + y
    x %= y      is equivalent to    x = x % y
    x |= y      is equivalent to    x = x | y
```

Rules for Simple Assignment and Object references

- if the left-hand operand is a class, the right-operand must be either a null, or of the same class or subclass type as the class of the left-operand
  - if class B extends class A,
    A a = new B() is **ok**
    as class B is guaranteed to fully implement class A
  - B b = new A() is **not ok**
    as there is no guarantee that the new A object will implement everything in class B
- if the left-hand operand is an interface, the right-operand must be either a null, or of the same interface or superinterface of the left-operand, or, a class that implements the interface or it's superinterface
  - if interface InB extends InA, class C implements InA, and class D implements InB
    InA inA = new C(); is **ok**
    as class C guarantees to implement everything in interface InA
    InB inB = new C(); is **not ok**
    as the new C is not guaranteed to implement everything in interface InB
  - InA inA1 = new D(); is **ok**
    as class D is guaranteed to implement everything in interface A through it's implementation of InB which extends InA

```
    class A{}
    class B extends A{}
    interface InA {}
    interface InB extends InA {}
    class C implements InA {}
    class D implements InB {}


    A a1       = new B(); // B is a subclass of A
//  B b1       = new A(); // incompatible types
```

```
    InA inA    = new C(); // C implements InA
    InB inB    = new D(); // D implements InB
    InA inA2   = new D(); // D implements InA as a superinterface
//  InB inB2   = new C(); // incompatible types

    inA        = inB;   // InA is a superinterface of InB
//  inB        = inA;   // compile-error: incompatible types

    Object o1  = inA;   // an Object type can take any reference
    Object o2  = inB;
    Object o3  = new C();

    C      c   = new Object(); // incompatible types

    B    b2    = null;  // any object reference can take a null
    InA  inA3  = null;
```

Summary
• If everything in the left-operands type contract can be met through the contract of the right-operand type, then the assignment will work. It doesn't matter if the right-operand type implements *more* than the left-operand type; as long as it implements what the left-operand type contract guarantees.

!!! Warning !!!
• The compiler treats the object on the right-side of the assignment **as if** it was the same type as the object on the left-side of the assignment. At runtime, the real class of the object is always used.

```
Class of object a1   -> Class B     // declared type was Class A
Class of object o1   -> Class D     // declared type was Object
Class of object o2   -> Class D     // declared type was Object
Class of object o3   -> Class C     // declared type was Object
```

Array assignments (JLS §5.2)

- an array can only be assigned to a variable of the same array type, of type Object, of interface Cloneable, or of interface java.io.Serializable

```
   int intArr[]  = { 1,2,3 };
   int intArr1[] = intArr;     // compiles ok

// String arr[] = new A();     // incompatible types
// String arr[] = inA;         // incompatible types

   Object         obj = intArr; // compiles ok
//                inA = intArr; // incompatible types
   Serializable   inS = intArr; // compiles ok
   Cloneable      inC = intArr; // compiles ok
```

Also see

- Conversions
- Sun Tech Tip: Definite Assignment

Example Code

- TestAssignment.java

Tips

- a class may be assigned to an Interface type if the class implements the interface or one of it's sub-interfaces

Traps

- assigning subclasses with the same parent to each other
- assigning a parent class to a subclass without a cast

	Conversions	Promotion	Overflow	Unary	Prefix	Arithmetic
	Bin/Hex/Octal	Bitwise	Shift	Comparison	Logical	Assignment
	Cast	Ternary	String	equals()	Precedence	Bit vs Logic
	Method Invocation					

Operators and Assignments - Ternary Operator

Syntax

```
operand1 ? operand2 : operand3
```

- also referred to as the **conditional operator**
- if operand1 is true, operand2 is returned, else operand3 is returned

```
 true ? op2 : op3            // op2 returned
false ? op2 : op3            // op3 returned
```

- operand1 **must be** a **boolean** type
- operand1 can be an expression that evaluates to a boolean type

```
 (5 == 5) ? "yes" : "no"     // output: yes
```

- operand1 and operand2 must be promotable numeric types or castable object references, or null
- if one of operand2 or operand3 is a byte and the other a short, the type of the returned value will be a short

```
byte = true ? byte : short  // found short, required byte
```

- if one of operand2 or operand3 is a byte, short or char and the other is a constant int value which will fit within the other operands range, the type of the returned value will be the type of the other operand

```
short =  true ? short : 1000 // compiles and runs ok
short = false ? short : 1000 // compiles and runs ok
```

- otherwise, normal binary numeric promotion applies
- if one of operand2 or operand3 is a null, the type of the return will be the type of the other operand
- if both operand2 and operand3 are different types, one of them must be compatible (castable) to the other type

```
Class_A a = new Class_A();
Class_B b = new Class_B();  // subclass of Class_A
Class_C c = new Class_C();
Class_A a1 = b;
Class_C c1;

c1 = false ? a : c;         // compile-error: incompatible types
a1 =  true ? b : a;         // returns class type of Class_B
```

Example Code

- TestTernary.java

Traps

- a non-boolean value or expression used for operand1

	Conversions	Promotion	Overflow	Unary	Prefix	Arithmetic
	Bin/Hex/Octal	Bitwise	Shift	Comparison	Logical	Assignment
	Cast	Ternary	String	equals()	Precedence	Bit vs Logic
	Method Invocation					

- Home
- SCJP2 Study Notes
- Language Fundamentals
- Operators and Assignments
- Flow Control and Exceptions
- Declarations and Access Control
- Garbage Collection
- Overloading and Overriding
- Threads
- The java.lang Package
- The java.util Package
- The java.awt Package
- The java.io Package
- References
- Miscellaneous Notes
- Tips & Traps
- Mock Exams
- Case Studies
- SCJA Notes
- SCJD Notes
- Projects
- Favourite Links
- About
- Feedback

Operators and Assignments - Boolean equals()

- defined in **java.lang.Object** therefore inherited by all classes
- returns true if and only if the two variables being compared hold a reference to the **same** object

To check if objects are of the same class use the Comparison operator: instanceof

```
Class_A a = new Class_A();
Class_B b = new Class_B();
Class_C c = new Class_A();
Class_B d = b;
Class_A e = null;

a.equals(b)            // false (different obj refs)
a.equals(c)            // false (different obj refs)
b.equals(d)            // true  (same object refs)
a.equals(e)            // false (always returned when
                       // compared to a null)
```

- java.lang.String overrides the java.lang.Object equals() method to return true if and only if the objects being compared contain the same sequence of characters.

```
String s0 = "Hello";
String s1 = new String("Hello");    // force new string object
String s2 = s0;

s0.equals(s1)          // true (diff objects, same chars)
s0.equals(s2)          // true (same chars, coincidence
                       //       they are same objects)
```

- java.lang.Boolean overrides the java.lang.Object equals() method, returning true if and only if the Boolean objects represent the same boolean value

```
Boolean b0 = new Boolean(true);
Boolean b1 = new Boolean(false);
Boolean b2 = new Boolean(true);
Boolean b3 = b1;

b0.equals(b1)          // false (different boolean values)
b0.equals(b2)          // true  (same boolean values)
b1.equals(b3)          // true  (same boolean values)
```

FYI
You cannot assign values to Boolean types with either of the following constructs:

```
Boolean b3 = new Boolean();
boolean b4 = true;

b3 = b4;      // compile-error: incompatible types
b3 = true;    // compile-error: incompatible types
```

Example Code

- TestBooleanEquals.java

Tips

- all the primitive type wrapper classes override the Object.equals() method to compare the *value* of the objects; the default Object.equals() checks if the variables reference the same object

	Conversions	Promotion	Overflow	Unary	Prefix	Arithmetic
	Bin/Hex/Octal	Bitwise	Shift	Comparison	Logical	Assignment
	Cast	Ternary	String	equals()	Precedence	Bit vs Logic
	Method Invocation					

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Operators and Assignments - Precedence

Operator precdence (JPL pg 378)	
Operator type	Operators
Postfix	[] . (params) expr++ expr--
Unary	++expr --expr +expr -expr ~ !
Creation or Cast	new (type)expr
Multiplicative	* / %
Additive	+ -
Shift	<< >> >>>
Relational	< > >= <= instanceof
Equality	== !=
Bitwise AND	&
Bitwise exclusive OR	^
Bitwise inclusive OR	|
Logical AND	&&
Logical OR	||
Ternary	?:
Assignment	= += -= *= /= %= >>= <<= >>>= &= ^= |=

- Precedence can be overridden using **parantheses**

```
 5 + 3  * 2                  // Result: 11
(5 + 3) * 2                  // Result: 16
```

- when two operators of the same precedence are next to each other, associativity rules apply
- all binary operators (except assignment operators) are **left-associative**
- **assignment** is **right-associative**

```
a - b + c is evaluated as (a - b) + c
5 - 2 + 1    // Result: 4, not 2


a = b = c is evaluated as  a = (b = c)

int a;
int b = 5;
int c = 1;

a = b = c;              // Result: 1
```

Possible problem areas

- where boolean expressions are used to control loops

```
while( v = stream.next() != null )
    processValue(v);

according to precedence rules, evaluates as
```

```
        v = (stream.next() != null )

    not the intended
        (v = stream.next()) != null
```

Example Code

- TestPrecedence.java

	Conversions	Promotion	Overflow	Unary	Prefix	Arithmetic
	Bin/Hex/Octal	Bitwise	Shift	Comparison	Logical	Assignment
	Cast	Ternary	String	equals()	Precedence	Bit vs Logic
	Method Invocation					

- Home
- SCJP2 Study Notes
- Language Fundamentals
- Operators and Assignments
- Flow Control and Exceptions
- Declarations and Access Control
- Garbage Collection
- Overloading and Overriding
- Threads
- The java.lang Package
- The java.util Package
- The java.awt Package
- The java.io Package
- References
- Miscellaneous Notes
- Tips & Traps
- Mock Exams
- Case Studies
- SCJA Notes
- SCJD Notes
- Projects
- Favourite Links
- About
- Feedback

Operators and Assignments - Bitwise vs Logical Operators

- the operand of every expression is evaluated before the operation is performed **except** for the **short-circuit** operators (&&, ¦¦) and ternary operator
- behaviour can produce unexpected results if you're not careful. For example, the following code illustrates what can occur if you try to set the value of a variable in an **if** condition and you always expect the new value to be available:

```
int i = 10;
int j = 12;

if( (i<j) ¦  (i=3) > 5 )   // value of i after oper: 3
if( (i<j) ¦¦ (i=3) > 5 )   // value of i after oper: 10

if( (i>j) &  (i=3) > 5 )   // value of i after oper: 3
if( (i>j) && (i=3) > 5 )   // value of i after oper: 10
```

- with & and ¦ both operands are always evaluated
- with && and ¦¦ the second operand is only evaluated when it is necessary
- with ¦¦ (i<j) evaluates to **true**; there is no need to check the other operand as ¦¦ returns **true** if either of the operands are **true**
- with && (i>j) evaluates to **false**; there is no need to check the other operand as && returns **true** only if both operands are **true**. In this case one is **false** so there is no need to check the other operand

Also see

Bitwise operators
Logical operators

Example Code

- TestBitwiseAndLogical.java

Traps

- using a new value based on a short-circuit operation that was never evaluated

	Conversions	Promotion	Overflow	Unary	Prefix	Arithmetic
	Bin/Hex/Octal	Bitwise	Shift	Comparison	Logical	Assignment
	Cast	Ternary	String	equals()	Precedence	Bit vs Logic
	Method Invocation					

Home
SCJP2 Study Notes
Language Fundamentals
Operators and Assignments
Flow Control and Exceptions
Declarations and Access Control
Garbage Collection
Overloading and Overriding
Threads
The java.lang Package
The java.util Package
The java.awt Package
The java.io Package
References
Miscellaneous Notes
Tips & Traps
Mock Exams
Case Studies
SCJA Notes
SCJD Notes
Projects
Favourite Links
About
Feedback

Operators and Assignments - Method Invocation

- when you pass a **primitive** value to a method, a copy of the value is made available to the method, not the value itself
- any changes made to the value in the method do not affect the original value

```
    int i = 50;
    changeValue(i);     // where method multiplies i by 3

Output:
    Original value of i:                     -> 50
    Value of i in the method:                -> 150
    Value of i after method invocation:      -> 50
```

Passing object references

- when you pass an **object reference** to a method, a copy of the reference is passed. Operations in the method which change the object reference do not affect the original; however, changes to the object itself within the method **affect the original object**

```
int[] array = { 10,10,10 }   // original array
changeObjectReference(array) // set the reference to a new array
changeActualObject(array)    // set the 2nd element of the array

Output:
  Original array values:                      10, 10, 10
  Array in the method:                        20, 20, 20
  After Object reference changed in method:   10, 10, 10
  After object changed in method:             10, 20, 10
```

Method invocation conversion (JLS §5.3)

- each argument is converted to the type of the method parameters
- widening conversion is implicit
- narrowing conversion is **not** implicit (values must be cast)

Also see:

- Conversion
- Understanding that parameters are passed by value and not by reference

Example Code

- TestMethodInvocation.java

Traps

- code that results in a primitive value being changed in a method (can't happen)
- code that results in an unchanged object value when it was changed in a method
- failing to cast a value to match a method parameter type ie assuming narrowing conversion

	on a method call

	Conversions	Promotion	Overflow	Unary	Prefix	Arithmetic
	Bin/Hex/Octal	Bitwise	Shift	Comparison	Logical	Assignment
	Cast	Ternary	String	equals()	Precedence	Bit vs Logic
	Method Invocation					