Marakana Java Fundamentals Training

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
2.2	3-Apr-2011		AG

Contents

Chapter 1

Overview of Java

Introduction to Java Development

1.1 History of Java

1991	Stealth Project for consumer electronics market (later Green Project) Language called Oak then renamed to Java
1993	First Person Project for set-top boxes
1994	LiveOak Project for new OS HotJava Web Browser
1995	Sun formally announces Java at SunWorld Netscape incorporates support for Java – Internet Explorer add support for Java
1996	**JDK 1.0** Basic support for AWT
1997	**JDK 1.1** JavaBeans, RMI, AWT, JDBC, servlets, JNDI, JFC, EJB
1998	**Java 1.2** Reflection, Swing, Java Collections Framework, plug-in, IDL Becomes known as **Java 2**, Port to Linux, Java Community Process
1999	Jini, PersonalJava, Java 2 Source Code, XML support, JavaServer Pages, Java Editions (J2ME, J2SE, J2EE)
2000	**J2SE 1.3** HotSpot, RMI/CORBA, JavaSound, JNDI , JPDA
2001	Java Connector Architecture, patterns catalog, Java Web Start 1.0, J2EE 1.3
2002	**J2SE 1.4** `assert`, regular expressions, exception chaining, NIO, logging, image I/O, integrated XML/XSLT, integrated Web Start, integrated JCE, JSSE, JAAS, J2EE 1.4 beta (Web Services)
2004	**J2SE 1.5** (Java 5) Generics, auto-boxing, metadata (annotations), enums, “for-each”
2005	Java 10th Anniversary
2006	**Java 1.6** (Java 6)

1.2 What is Java?

Java Technology consists of:

- Java Language
- Java Platform
- Java Tools

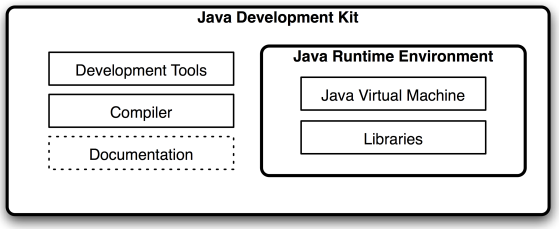


Figure 1.1: Java technology

Java language is a general-purpose programming language. It can be used to develop software for mobile devices, browser-run applets, games, as well as desktop, enterprise (server-side), and scientific applications.

Java platform consists of Java virtual machine (JVM) responsible for hardware abstraction, and a set of libraries that gives Java a rich foundation.

Java tools include the Java compiler as well as various other helper applications that are used for day-to-day development (e.g. debugger).

1.3 Why Java?

- Object oriented
- Interpreted
- Portable
- Simple yet feature-full
- Secure and robust
- Scalable
- High-performance multi-threaded
- Dynamic
- Distributed

Object Oriented
: Everything in Java is coded using OO principles. This facilitates code modularization, reusability, testability, and performance.

Interpreted/Portable
Java source is complied into platform-independent bytecode, which is then interpreted (compiled into native-code) at runtime. Java's slogan is "Write Once, Run Everywhere"

Simple
Java has a familiar syntax, automatic memory management, exception handling, single inheritance, standardized documentation, and a very rich set of libraries (no need to reinvent the wheel).

Secure/Robust
Due to its support for strong type checking, exception handling, and memory management, Java is immune to buffer-overruns, leaked memory, illegal data access. Additionally, Java comes with a Security Manager that provides a sand-box execution model.

Scalable
Thanks to its overall design, Java is scalable both in terms of performance/throughput, and as a development environment. A single user can play a Java game on a mobile phone, or millions of users can shop though a Java-based e-commerce enterprise application.

High-performance/Multi-threaded
With its HotSpot Just-in-Time compiler, Java can achieve (or exceed) performance of native applications. Java supports multi-threaded development out-of-the-box.

Dynamic
Java can load application components at run-time even if it knows nothing about them. Each class has a run-time representation.

Distributed
Java comes with support for networking, as well as for invoking methods on remote (distributed) objects through RMI.

1.4 Java After 15 years

- 6.5+ million software developers
- 1.1+ billion desktops with Java installed
- 3+ billion mobile devices with support for Java
- Embedded devices based on Java: set-top boxes, printers, web cams, games, car navigation systems, lottery terminals, medical devices, parking payment stations, etc.

Source: http://www.java.com/en/about/

Chapter 2

A Java Hello World Program

Implementing Hello World in Java

2.1 HelloWorld.java

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

- All code is contained within a *class*, in this case `HelloWorld`.
- The file name must match the class name and have a `.java` extension, for example: `HelloWorld.java`
- All executable statements are contained within a *method*, in this case named `main()`.
- Use `System.out.println()` to print text to the terminal.

Classes and methods (including other flow-control structures) are always defined in blocks of code enclosed by curly braces (`{ }`).

All other statements are terminated with a semi-colon (`;`).

Java language is case-sensitive! This means that `HelloWorld` is not the same as `helloworld`, nor is `String` the same as `string`.

There is an generally accepted naming convention in the Java community to capitalize the names of Java classes and use the so-called *CamelCase* (or *CamelHump*) notation, where the first letter of each word is capitalized, but the words are joined together (i.e. no dashes or underscores).

Note
Java source files can be encoded in UTF-8 to support internationalized characters (even for class, method, and variable names). In most situations however, Java sources are stored in US-ASCII character set, and internationalized strings are loaded from external resource files.

2.2 Java Keywords

Table 2.1: Java Keywords

abstract	assert	boolean	break	byte
case	catch	char	class	const
continue	default	do	double	else
enum	extends	final	finally	float
for	goto	if	implements	import
instanceof	int	interface	long	native
new	package	private	protected	public
return	short	static	strictfp	super
switch	synchronized	this	throw	throws
transient	try	void	volatile	while

Keywords `goto` and `const` are reserved, but never used.

Keyword `strictfp` was added in Java 1.2.

Keyword `assert` was added in Java 1.4.

Keyword `enum` was added in Java 1.5.

In addition to these 50 keywords, Java also defined three special literals: `true`, `false`, and `null`.

Keywords in our `HelloWorld` program are in bold:

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

2.3 Java Identifiers

- An *identifier* is the name of a class, variable, field, method, or constructor.
  - Cannot be a Java keyword or a literal
  - Can contain letters, digits, underscores, and dollar signs
  - Cannot start with a digit
- Valid identifier examples: `HelloWorld`, `args`, `String`, `i`, `Car`, `$myVar`, `employeeList2`
- Invalid identifier examples: `1st`, `byte`, `my-Arg`, `*z`
- Java naming conventions for identifiers:
  - Use *CamelCase* for most identifiers (classes, interfaces, variables, and methods).
  - Use an initial capital letter for classes and interfaces, and a lower case letter for variables and methods.
  - For named constants, use all capital letters separated by underscores.
  - Avoid using `$` characters in identifiers.

Identifiers in our `HelloWorld` program are in bold:

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

2.4 Compiling Java Programs

- The JDK comes with a command-line compiler: `javac`.
  - It compiles source code into *Java bytecode*, which is low-level instruction set similar to binary machine code.
  - The bytecode is executed by a *Java virtual machine* (*JVM*), rather than a specific physical processor.
- To compile our `HelloWorld.java`, you could go to the directory containing the source file and execute:

  ```
  javac HelloWorld.java
  ```

- This produces the file `HelloWorld.class`, which contains the Java bytecode.

Use the `javac -help` option to get a full list of available options:

```
Usage: javac <options> <source files>
where possible options include:
  -g                         Generate all debugging info
  -g:none                    Generate no debugging info
  -g:{lines,vars,source}     Generate only some debugging info
  -nowarn                    Generate no warnings
  -verbose                   Output messages about what the compiler is doing
  -deprecation               Output source locations where deprecated APIs are used
  -classpath <path>          Specify where to find user class files and annotation  ↩
      processors
  -cp <path>                 Specify where to find user class files and annotation  ↩
      processors
  -sourcepath <path>         Specify where to find input source files
  -bootclasspath <path>      Override location of bootstrap class files
  -extdirs <dirs>            Override location of installed extensions
  -endorseddirs <dirs>       Override location of endorsed standards path
  -proc:{none,only}          Control whether annotation processing and/or compilation is  ↩
      done.
  -processor <class1>[,<class2>,<class3>...]Names of the annotation processors to run;  ↩
      bypasses default discovery process
  -processorpath <path>      Specify where to find annotation processors
  -d <directory>             Specify where to place generated class files
  -s <directory>             Specify where to place generated source files
  -implicit:{none,class}     Specify whether or not to generate class files for implicitly  ↩
      referenced files
  -encoding <encoding>       Specify character encoding used by source files
  -source <release>          Provide source compatibility with specified release
  -target <release>          Generate class files for specific VM version
  -version                   Version information
  -help                      Print a synopsis of standard options
  -Akey[=value]              Options to pass to annotation processors
  -X                         Print a synopsis of nonstandard options
  -J<flag>                   Pass <flag> directly to the runtime system
```

You can view the generated byte-code, using the `-c` option to `javap`, the Java class disassembler. For example:

```
javap -c HelloWorld
```

2.5 Running Java Programs

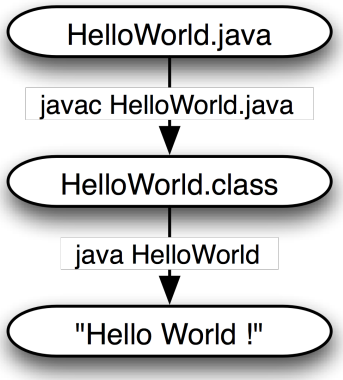


Figure 2.1: Compiling and running a Java program

- To run the bytecode, execute:

```
java HelloWorld
```

- Do not include the `.class` extension.
- The `java` command starts the JVM and executes the class bytecode.
- The JVM abstracts O/S and H/W from the Java application.

It is the Java virtual machine that provides the layer of insulation to Java programs so that they do not depend on the underlying operating system or hardware semantics.

This allows compiled Java applications to run on any platform that has a Java virtual machine written for it:

- AIX
- BSD
- HP-UX
- Linux
- Mac OS X
- Solaris
- Windows

2.6 Comments in Java Code

- `// single-line comment`
  - Comments-out everything until a line break
- `/* Multi-line comment */`

  - Useful for commenting out a section of code
  - Cannot be nested within other multi-line comments
- `/** JavaDoc comments */`
  - Similar to multi-line comments but used to document Java code (classes, methods, fields)
  - Extracted using `javadoc` command-line utility

```
/**
 * My first class
 * @author student
 * @version 1.0
 */
public class HelloWorld {

    /**
     * Main method that runs when the program is started.
     * @param args command-line arguments
     */
    public static void main (String[] args) {
        /*
        This is how you can print to STDOUT
         */
        System.out.println("Hello World"); // don't forget the semi-colon

        /*
        System.out.println("I was also going to say...");
        System.out.println("but I changed my mind");
        */
    }
}
```

2.7 The `main()` Method

- A Java application is a public Java class with a `main()` method.
  - The `main()` method is the entry point into the application.
  - The signature of the method is always:

    ```
    public static void main(String[] args)
    ```

  - Command-line arguments are passed through the `args` parameter, which is an array of `Strings`

Applets and servlets do not have a `main()` method because they are run through *template methods* in the context of a framework. For example, a web container is responsible for instantiating a servlet object. It then invokes a standard set of methods defined by the `Servlet` class at appropriate points in the servlet lifecycle; for example, `init()` to initialize the servlet, `doGet()` to handle an HTTP GET message, `destroy()` when the web container is about to destroy the object, and so on. The parent class provides default implementations of these methods, but the derived servlet class can override any of the template methods to provide servlet-specific functionality.

Chapter 3

Data Types

Primitive Data Types in Java

3.1 Declaring and Assigning Variables

- The syntax for declaring a variable is:
  DataType variableName[= expression];
- Examples:

```
float j;
int i = 5 + 3;
```

- Java is a strongly typed language
  - Every variable must have an explicit type
  - Expressions assigned to the variable must be a compatible type
- Local variables (declared within methods) must be declared and initialized before they are used
- Class and instance variables can be declared anywhere and are initialized to defaults:
  - `0` for numerical typed variables
  - `false` for boolean variables
  - `null` for object reference variables

In contrast to a language like C, in Java the local variables do not have to be declared at the beginning of program blocks (defined by curly braces). They can be declared when they need to be used.

```
public static void main(String[] args) {
    System.out.print("Number of arguments: ");
    int len = args.length;
    System.out.println(len);
}
```

Multiple variables can be declared on a singe line: `int i, j, k;`

Multiple variables can be initialized at the same time: `i = j = k = 5;`

There is an generally accepted naming convention in the Java community to start variable names with a lower-case letter and use a so-called *CamelCase* (or *CamelHump*) notation, where the first letter of each subsequent word is capitalized, but the words are joined together (i.e. no dashes or underscores).

3.2 Java Primitive Types

Table 3.1: Java Primitive Types

Type	Size	Range	Default*
`boolean`	1 bit	`true` or `false`	`false`
`byte`	8 bits	[-128, 127]	`0`
`short`	16 bits	[-32,768, 32,767]	`0`
`char`	16 bits	[`'\u0000'`, `'\uffff'`] or [0, 65535]	`'\u0000'`
`int`	32 bits	[-2,147,483,648 to 2,147,483,647]	`0`
`long`	64 bits	[-2^{63}, 2^{63}-1]	`0`
`float`	32 bits	32-bit IEEE 754 floating-point	`0.0`
`double`	64 bits	64-bit IEEE 754 floating-point	`0.0`

* The default value for uninitialized class or instance variables of this type.

Even though Java is a fully object oriented language, for performance reasons it defines eight primitive data types. All of these types also have their respective wrapper classes (for true OO) in the default `java.lang` package (which we will see later).

In addition to the eight primitive types, there is also a concept of a `void` type, which is only used for a method return type (i.e. to indicate that nothing is returned).

Examples:

```
boolean bln = true; // booleans can only be 'true' or 'false'
byte b = 0x20;      // using hexadecimal notation
short s = 500;      // small integer
char c = 'A';       // must use single quotes to denote characters
char tab = '\t';    // other specials: \n, \r, \f, \b, \\, \', \"
int i = 1000000;    // decimal notation
int j = 0x3FA0B3;   // hexadecimal notation
int k = 0777;       // octal notation
float f = 1.5f;     // trailing 'f' distinguishes from double
long l = 2000000L;  // trailing 'L' distinguishes from int
double pi = 3.141592653589793;  // doubles are higher precision
double large = 1.3e100;     // using the exponent notation
```

3.3 Conversion Between Types

- Implicit up-casting: a lower-precision type is automatically converted to a higher precision type if needed:

```
int i = 'A';
```

- Explicit down-casting: a manual conversion is required if there is a potential for a loss of precision, using the notation: `(`*`lowerPrecType`*`)` *`higherPrecValue`*

```
int i = (int) 123.456;
```

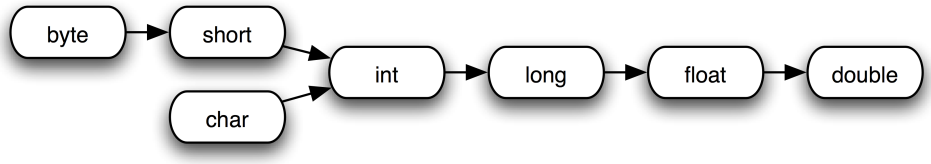


Figure 3.1: Implicit up-casting of primitive types

Note
Java has a clear separation between numerical types and `boolean`s. They are not interchangeable. This means that an `int` cannot be used in place of a `boolean` expression, for example in an `if` statement:

```
int someInteger = 0;

/*
 * This will not compile, because someInteger is not
 * a boolean type (or a boolean expression)
 */

if (someInteger) {
    // Do something
}

// This will compile
if (someInteger != 0) {
    // Do something
}
```

Note
Strings are not converted automatically into integers or other primitive data types they may represent. Strings must be parsed:

```
int myInt = Integer.parseInt(args[0]);
```

3.4 Declaring Arrays

- An *array* is a simple data structure to hold a series of data elements of the same type.
- Declare an array variable in one of two ways:
  - With `[]` after the variable type: `int[] values;`
  - With `[]` after the variable name: `int values[];`
- Arrays can be single- or multi-dimensional.
  - A two dimensional array could be declared with: `double values[][];`
- Array elements are integer indexed.

- Use *arrayName*.length to get the array length.
- Elements are indexed from 0 to *arrayName*.length - 1
- Access individual elements with *arrayName*[*index*]

Examples of array declarations:

```
String[] args;      // single-dimensional array of String objects
int[] numbers;      // single-dimensional array of ints
byte[] buffer;      // single-dimensional array of bytes
short[][] shorts;   // double-dimensional array of shorts
```

To get the array length:

```
int arrayLength = myNums.length;    // 5
int nameLength = name.length;       // 3
int bufLength = buf[2].length;      // 1024
```

3.5 Creating and Initializing Array Objects

- Arrays are implemented as objects.
- You can use the new operator to create an array of an indicated type and length, as in:

  ```
  int[] values = new int[10];
  ```

  - Elements are initialized automatically to type-appropriate default values.
- Alternatively, you can use an *array initializer* to create an array with a set of initial values.
  - The array initializer consists of a sequence of comma-separated expressions of an appropriate type, enclosed within braces. For example:

    ```
    int[] values = {5, 4, 3, 2, 1};
    ```

  - The array object is sized automatically based on the number of initial values.

Examples of array initializations:

```
int[] myNums = { 1, 3, 5, 7, 9 };   // 5 elements
char[] name = new char[3];
name[0] = 'T';
name[1] = 'o';
name[2] = 'm';
short[][] matrix = { {1,4,7}, {5,6,3}, {2,8,9} };
byte[][] bufs = new byte[10][1024];
```

To access an element of an array:

```
int myNum = myNums[3];      // 7
char firstLetter = name[0]; // 'T'
matrix[1][2] = 0;       // used to be 3
```

3.6 Modifying Array Size

- Once created, the size of an array cannot change.
  - If you need to grow an array, you must create a larger array object of the same type, and then copy the elements from the old array to the new array.
  - The `System.arraycopy()` method is an efficient way to copy the existing elements to the new array. For example:

```
int[] values = {5, 4, 3, 2, 1}; // A 5-element int array
int[] newValues = new int[10];  // A 10-element int array

// Copy all elements from values to newValues
System.arraycopy(values, 0, newValues, 0, values.length);

// Assign the array back to values
values = newValues;
```

In the example above, the `values` and `newValues` variables do not actually contain the arrays, they *refer* to the array objects. We'll examine this concept more in the Object Oriented module.

3.7 Strings

- Strings are objects.
  - In Java, strings are instances of the `String` class.
  - Unlike C/C++, you can't treat a `String` directly as an array of `chars`.
  - There are methods that allow you to create a `String` from an array of `chars`, and vice versa.
- Enclosing a sequence of literal characters within double quotes automatically creates a `String` object: `"this is my string"`
- Strings are *immutable*
  - Once created, a `String` cannot be modified.
  - But a `String` has methods to perform various transformations, each returning a new `String` object.
- The `String` class has many methods, including: `length`, `replace`, `substring`, `indexOf`, `equals`, `trim`, `split`, `toUpperCase`, `endsWith`, etc.

To get the length of a `String` do:

```
String s = "my string";
int len = s.length();
```

To parse a primitive value from a `String`, do:

```
boolean b = Boolean.valueOf(s).booleanValue();
byte b = Byte.parseByte(s);
short s = Short.parseShort(s);
char c = s.charAt(0);
int i = Integer.parseInt(s);
float f = Float.parseFloat(s);
long l = Long.parseLong(s);
double d = Double.parseDouble(s);
```

We will examine the `String` API later in class.

Chapter 4

Operators

Operators in Java

4.1 Arithmetic Operators

Operator	Use	Description
`+`	`x + y`	Adds `x` and `y`
`-`	`x - y`	Subtracts `y` from `x`
	`-x`	Arithmetically negates `x`
`*`	`x * y`	Multiplies `x` by `y`
`/`	`x / y`	Divides `x` by `y`
`%`	`x % y`	Computes the remainder of dividing `x` by `y`

In Java, you need to be aware of the *type* of the result of a binary (two-argument) arithmetic operator.

- If either operand is of type `double`, the other is converted to `double`.
- Otherwise, if either operand is of type `float`, the other is converted to `float`.
- Otherwise, if either operand is of type `long`, the other is converted to `long`.
- Otherwise, *both* operands are converted to type `int`.

For unary (single-argument) arithmetic operators:

- If the operand is of type `byte`, `short`, or `char`, the result is a value of type `int`.
- Otherwise, a unary numeric operand remains as is and is not converted.

This can result in unexpected behavior for a newcomer to Java programming. For example, the code below results in a compilation error:

```
short a = 1;
short b = 2;
short c = a + b;     // Compilation error
```

The reason is that the result of the addition is an `int` value, which cannot be stored in a variable typed `short` unless you explicitly cast it, as in:

```
short c = (short) (a + b);
```

Also, there are a couple of quirks to keep in mind regarding division by 0:

- A non-zero floating-point value divided by 0 results in a signed `Infinity`. `0.0/0.0` results in `NaN`.
- A non-zero integral value divided by integral 0 results in an `ArithmeticException` thrown at runtime.

Note
In Java, unlike C/C++, it is legal to compute the remainder of floating-point numbers. The result is the remainder after dividing the dividend by the divisor an integral number of times. For example, `7.9 % 1.2` results in `0.7` (approximately, given the precision of a `float` or `double` type).

4.2 Shortcut Arithmetic Operators

Operator	Use	Description
`++`	`x++`	`y = x++;` is the same as `y = x; x = x + 1;`
	`++x`	`y = ++x;` is the same as `x = x + 1; y = x;`
`--`	`x--`	`y = x--;` is the same as `y = x; x = x - 1;`
	`--x`	`y = --x;` is the same as `x = x - 1; y = x;`

The location of the shortcut operator is only important if the operator is used in an expression where the operand value is assigned to another variable or returned from a method.

That is to say that:

```
x++;
```

is equivalent to:

```
++x;
```

But in an assignment:

```
y = x++;
```

is not the same as:

```
y = ++x;
```

Shortcut operators are often used in `for` loops to increment the loop index variable (as we will see soon).

4.3 String Concatenation

- The plus (+) operator performs string concatenation.
  - The result is a new `String` object, for example:

```
String first = "George";
String last = "Washington";
String name = first + " " + last;   // "George Washington"
```

- You can concatenate a `String` with Java primitive types.
  - Java automatically generates a `String` representation of the primitive value for concatenation, for example:

```
int count = 8;
String msg = "There are " + count + " cows.";   // "There are 8 cows."
```

- The arithmetic + operator has the same order of precedence as the `String` concatenation +.
  - In a complex expression, they are evaluated from left to right, unless you use parentheses to override, for example:

```
String test = 1 + 1 + " equals " + 1 + 1;   // "2 equals 11"
```

- For other object types used as string concatenation operands, Java automatically invokes the object's `toString()` method, for example:

```
Date now = new Date();
String msg = "The time is: " + now;
// "The time is: Mon Feb 07 18:04:48 PST 2011"
```

4.4 Relational Operators

Operator	Use	Description
>	`x > y`	`x` is greater than `y`
>=	`x >= y`	`x` is greater than or equal to `y`
<	`x < y`	`x` is less than `y`
<=	`x <= y`	`x` is less than or equal to `y`
==	`x == y`	`x` is equal to `y`
!=	`x != y`	`x` is not equal to `y`

Only numerical primitive types (`byte`, `short`, `char`, `int`, `float`, `long`, and `double`) can be compared using the ordering operators (>, >=, <, and <=). Objects and `booleans` can only be tested for [in]equality (==, !=).

Due to Java's type safety (`booleans` are not integers and vice versa), it is not possible to make the common mistake:

```
int x = 5;
int y = 3 + 2;

// Does not compile
if (x = y) {
    // Do something
}
```

```
// You must use equality operator
if (x == y) {
    // Do something
}
```

4.5 Logical Boolean Operators

Operator	Use	Evaluates to `true` if
`&&`	`x && y`	Both `x` and `y` are `true`
`||`	`x || y`	Either `x` or `y` are `true`
`!`	`!x`	`x` is not `true`

- All operands to these operators must be `boolean` values.
- The logical `&&` and `\\` operators are subject to *short circuit evaluation*.

These logical operators accept `boolean` values only. Attempting to use numerical or other operand types results in a compilation error.

As with many modern programming languages, Java uses *short circuit evaluation* when evaluating logical boolean operators. This means that conditional operators evaluate the second operand only when needed:

- In `x && y`, `y` is evaluated only if `x` is `true`. If `x` is `false`, the expression immediately evaluates to `false`.
- In `x || y`, `y` is evaluated only if `x` is `false`. If `x` is `true`, the expression immediately evaluates to `true`.

For example:

```
int i = 0;
int j = 5;
if ( (i != 0) && ((j / i) > 1) ) {
    // Do something
}
```

Because we use a logical AND (`&&`), the first expression evaluates to `false` and so the second expression is not tested and the overall condition is `false`.

4.6 Bitwise Operators

Operator	Use	Evaluates to `true` if
`~`	`~x`	Bitwise complement of `x`
`&`	`x & y`	AND all bits of `x` and `y`
`|`	`x | y`	OR all bits of `x` and `y`
`^`	`x ^ y`	XOR all bits of `x` and `y`
`>>`	`x >> y`	Shift `x` right by `y` bits, with sign extension
`>>>`	`x >>> y`	Shift `x` right by `y` bits, with `0` fill
`<<`	`x << y`	Shift `x` left by `y` bits

In general, these bitwise operators may be applied to integral values only. For the `~`, `&`, `\|`, and `^` operators:

- If either operand is of type `long`, the other is converted to `long`.
- Otherwise, *both* operands are converted to type `int`.

For the shift operators:

- If the value being shifted is of type `long`, the result is of type `long`.
- Otherwise, the result is of type `int`.

Operation	Decimal Result	Binary Result
`x`	71	00000000000000000000000001000111
`y`	5	00000000000000000000000000000101
`~x`	-72	11111111111111111111111110111000
`x & y`	5	00000000000000000000000000000101
`x | y`	71	00000000000000000000000001000111
`x ^ y`	66	00000000000000000000000001000010
`x >> y`	2	00000000000000000000000000000010
`x >>> y`	2	00000000000000000000000000000010
`x << y`	2272	00000000000000000000100011100000

The bitwise operators may also be used with `boolean` values to produce a `boolean` result. However, you cannot combine `boolean` and integral operands in a bitwise expression.

4.7 Assignment Operators

Operator	Use	Shortcut for
`=`	`x = y`	`x = y`
`+=`	`x += y`	`x = x + y`
`-=`	`x -= y`	`x = x - y`
`*=`	`x *= y`	`x = x * y`
`/=`	`x /= y`	`x = x / y`
`%=`	`x %= y`	`x = x % y`
`&=`	`x &= y`	`x = x & y` (also works for `boolean` values)
`|=`	`x |= y`	`x = x | y` (also works for `boolean` values)
`^=`	`x ^= y`	`x = x ^ y` (also works for `boolean` values)
`>>=`	`x >>= y`	`x = x >> y`
`>>>=`	`x >>>= y`	`x = x >>> y`
`<<=`	`x <<= y`	`x = x << y`

Actually, a compound assignment expression of the form *`E1 op= E2`* is equivalent to *`E1 = (T) ((E1) op (E1))`*, where *`T`* is the type of *`E1`*.

For example, the following code is compiles without error:

```
short x = 3;
x += 4.6;
```

and results in `x` having the value `7` because it is equivalent to:

```
short x = 3;
x = (short) (x + 4.6);
```

4.8 Other Operators

Operator	Use	Description
`()`	`(x + y) * z`	Require operator precedence
`?:`	`z = b ? x : y`	Equivalent to: `if (b) { z = x; } else { z = y; }`
`[]`	`array[0]`	Access array element
`.`	`str.length()`	Access object method or field
`(type)`	`int x = (int) 1.2;`	Cast from one type to another
`new`	`d = new Date();`	Create a new object
`instanceof`	`o instanceof String`	Check for object type, returning `boolean`

The short-cut *conditional operator* (`?:`) (also known as the "if-then-else operator" or the "ternary operator") is used very often to make Java code more compact, so it is worthwhile to get accustomed to it.

For example:

```
int x = 0;
int y = 5;

// Long way

int z1;
if (x == 0) {
    z1 = 0;
} else {
    z1 = x / y;
}

// Short-cut way
int z2 = (x == 0) ? 0 : x / y;
```

We'll explore the other operators listed on this slide in more depth throughout this class.

4.9 Operator Precedence

1. `[]`, `()`, `.`
2. `++`, `--`, `~`, `!`, `(type)`, `new`, `-`
3. `*`, `/`, `%`
4. `+`, `-`

5. `<<`, `>>`, `>>>`
6. `<`, `<=`, `>`, `>=`, `instanceof`
7. `==`, `!=`
8. `&`
9. `^`
10. `|`
11. `&&`
12. `||`
13. `?  :`
14. `=`, `*=`, `/=`, `+=`, `-=`, `%=`, `<<=`, `>>=`, `>>>=`, `&=`, `^=`, `|=`

Use the `()` operator to change precedence of operators:

```
x << (((((x == 0 ? 0 : (x / y)) + 5) * z) > 4) ? 3 : 7)
```

Other than in the most trivial situations, It is always a good practice to use `()` to document your intentions, regardless of precedence rules.

For example, change:

```
int x = 5 - y * 2 > 0 ? y : 2 * y;
```

to:

```
int x = ((5 - (y * 2)) > 0) ? y : (2 * y);
```

even though both lines will assign the same value to `x`

Chapter 5

Statements and Flow Control

Controlling the Flow of Java Programs

5.1 Expressions

- An *expression* is a code fragment consisting of variables, constants, method invocations, and operators that evaluates to a single value.
- Examples:
  - `2 + 2`
  - `a = b + c`
  - `myMethod()`

5.2 Statements and Blocks

- A *statement* is a complete unit of execution.
- Many types of expressions can be made into statements by terminating them with a semicolon (`;`).
  - Assignment expressions: `answer = 42;`
  - Increment or decrement expressions: `a++;` or `b--;`
  - Method invocations: `System.out.println("Hello, world!");`
  - Object creation: `Account client = new Account();`
- A *declaration statement* declares a variable, optionally assigning it an initial value. For example: `int count;`
- A *block* is a group of zero or more statements between balanced braces (`{ }`).
  - A block can be used anywhere a single statement is allowed.
- A *control flow statement* is a structure controlling the execution of other statements. We'll explore these through the rest of this module.

Java *whitespace* characters are the space, tab, carriage return, newline (aka linefeed), and formfeed characters.

Java requires one or more whitespace characters between keywords and/or identifiers. Other whitespace characters that do not occur within quoted string literals are not considered significant and serve only to increase the readability of code.

5.3 Local Variable Storage: The Stack

- All variables declared within a block are known as *local variables*, which are stored in a memory location known as the *stack*. Examples of where blocks are used include:
  - Method definitions
  - Flow control statements
  - Other blocks contained within a sequence of statements
- The stack grows and shrinks as the program runs.
  - Entering a block expands the stack, storing local variables in memory.
  - Leaving a block shrinks the stack, flushing local variables from memory.
- The *scope* of a local variable — that is, the context in which is visible and accessible to your program — is limited to the block in which it is declared, and all nested blocks.

> **Note**
> Java does not allow you to declare a local variable with the same name as a local variable already in the same scope.

```
public static void main(String[] args) {
    int x = 1;
    // A local scope where args and x are visible

    while (x <= 5) {
        String msg = "x is now " + x;
        // A local scope where args, x, and msg are visible
        System.out.println(msg);
    }
    // msg is no longer in scope, and has been flushed from memory
}
```

As previously mentioned, local variables must be initialized before they are used.

A local variable is visible only in the block of code (or sub blocks) where it is declared. Along with the variable's value, its name is also flushed as the stack shrinks, allowing the same variable to be re-declared in another block of code.

> **Note**
> Method parameters are also considered local variables, because they too are declared (and initialized automatically by the calling code) on the stack.

> **Tip**
> It is a good practice to limit the scope of local variables as much as possible. This promotes variable name reuse. In the case of objects, limiting local variable scope allows the JVM to quickly reclaim the memory occupied by objects that are no longer in use.

5.4 The `return` Statement

- The `return` statement breaks out of the entire method.
- It must return a value of the type specified in the method's signature.
- It must not return a value if the method's return value is of type `void`.
- The value can be an expression:

```
public static int max(int x, int y) {
    return (x > y) ? x : y;
}
```

```
public class FindNumber {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("Usage: FindNumber <num>");
            return;
        }
        int[] numbers = {1, 3, 4, 5, 7, 5, 2, 8, 9, 6};
        int findNumber = Integer.parseInt(args[0]);
        System.out.println(find(numbers, findNumber));
    }

    public static int find(int[] nums, int num) {
        for (int i = 0; i < nums.length; i++) {
            if (nums[i] == num) {
                return i;
            }
        }
        return -1;
    }
}
```

5.5 The `if-else` Statement

- The `if-else` statement defines a block of code that is to be executed based on some boolean condition.

```
if (boolean-expression) {
    // Run if BE is true
} else if (boolean-expression2) {
    // Run if BE is false and BE2 is true
} else {
    // Run if both BE and BE2 are false
}
```

- The `else if` part is optional and can appear many times between the `if` and `else` parts.
- The `else` part is optional but can appear only once after `if` and all/any `else if` parts.

```
public class LetterGrade {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("Usage: LetterGrade <numeric-score>");
            return;
        }
        int score = Integer.parseInt(args[0]);
        char grade;

        if (score >= 90) {
            grade = 'A';
        } else if (score >= 80) {
            grade = 'B';
        } else if (score >= 70) {
            grade = 'C';
        } else if (score >= 60) {
            grade = 'D';
        } else {
            grade = 'F';
        }

        System.out.println("Grade = " + grade);
    }
}
```

Note
In this example, the numerical score is parsed from a string (the first and only command-line argument) into an integer data type.

5.6 The `switch` Statement

- The `switch` statement is a special-purpose `if-else-if-else` construct that is easier and shorter to write:

```
switch(expression) {
    case const1:
        /* do X */
        break;
    case const2:
        /* do Y */
        break;
    default:
        /* do something else */
}
```

  - The `switch` expression must be an integral type *other* than `long`: `byte`, `short`, `char`, or `int`.
  - `case` values must be constants (not variables).
  - `case` values are tested for equality only (==).

```
public class MonthFromNumber {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("Usage: MonthFromNumber <month>");
```

```
            return;
        }
        int month = Integer.parseInt(args[0]);
        switch (month) {
            case 1:  System.out.println("January"); break;
            case 2:  System.out.println("February"); break;
            case 3:  System.out.println("March"); break;
            case 4:  System.out.println("April"); break;
            case 5:  System.out.println("May"); break;
            case 6:  System.out.println("June"); break;
            case 7:  System.out.println("July"); break;
            case 8:  System.out.println("August"); break;
            case 9:  System.out.println("September"); break;
            case 10: System.out.println("October"); break;
            case 11: System.out.println("November"); break;
            case 12: System.out.println("December"); break;
            default: System.out.println("Invalid month: " + month);
        }
    }
}
```

Note
You can also switch on enumerated values defined by `enum`, which is discussed in Typesafe Enums.

5.7 The `switch` Statement (cont.)

- A `case` is an entry point into a `switch` statement, whereas a `break` acts as an exit point.
  - Typically each `case` has a trailing `break`.
  - Without a `break`, execution automatically "falls through" to the statements in the following `case`.

    Tip
    If you intentionally fall through to the following `case`, include a comment to alert maintenance programmers of your intent. Otherwise, they might think you simply forgot the `break` and add it in later.

- The optional `default` case cannot `break` and is executed if no other `cases` match and `break` prior to it.
- To exit both the `switch` statement as well as the method in which it is defined, use a `return` in place of a `break`.

```
public class DaysInMonth {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.err.println("Usage: DaysInMonth <year> <month>");
            return;
        }
        int year = Integer.parseInt(args[0]);
        int month = Integer.parseInt(args[1]);
        int numDays = 0;

        switch (month) {
            case 1:
            case 3:
```

```
            case 5:
            case 7:
            case 8:
            case 10:
            case 12:
                numDays = 31;
                break;
            case 4:
            case 6:
            case 9:
            case 11:
                numDays = 30;
                break;
            case 2:
                numDays =  ( ( (year % 4 == 0) && !(year % 100 == 0))
                           || (year % 400 == 0) )? 29 : 28;
                break;
            default:
                System.err.println("Invalid month " + month);
                return;
        }
        System.out.println("Number of Days = " + numDays);
    }
}
```

5.8 The `while` Loop Statement

- The `while` loop statement executes a block of statements as long as a condition remains true:

```
while (boolean-expression) {
    // Do something repeatedly
    // Update condition
}
```

- To ensure the loop body runs at least once, a `do-while` variant executes the statement block *before* evaluating the condition:

```
do {
    // Do something at least once
} while (boolean-expression);
```

Note
Remember to update the condition to avoid infinite loops.

```
public class WhileArguments {
    public static void main (String[] args) {
        int i = 0;
        while (i < args.length) {
            System.out.println(args[i]);
            i += 1;
        }
    }
}
```

5.9 The `for` Loop Statement

- The `for` loop statement is similar to the `while` loop, but allows compact initialization of an iterator variable and its increment/decrement:

```
for (init; condition; update) {
    // Do something
}
```

  - The initialization statement runs before anything else.
  - The condition is evaluated before each repetition of the loop body.
  - The update statement is run after each body repetition. This can consist of multiple expression statements separated by commas (`,`).

Note
You can declare a local variable in the `for` initialization statement, in which case its scope is the loop body.

```
for (int i = 1; i <= 10; i++) {
    // i is local to this scope
}
```

```
public class ForArguments {
    public static void main(String[] args) {
        for (int i = 0; i < args.length; i++) {
            System.out.println(args[i]);
        }
    }
}
```

Tip
You can omit the initialization and/or update statements from the `for` loop, but you must still include the `;` separator characters:

```
// x has been declared and initialized elsewhere
for (; x < 5; x++) System.out.println(x);
```

5.10 Using `for` to Iterate over Arrays and Collections

- Java 5 also introduced an advanced `for` syntax known as `for-each`.
  - It is designed specifically for iterating over collections of data, whether whose collections are arrays, or some predefined collection classes (discussed in the Java Collections Framework module).
  - The `for-each` statement has the following syntax:

```
for (Type element: collectionOfType) {
    // Do something with element
}
```

  - For example:

```
public class ForEachArguments {
    public static void main(String[] args) {
        for (String arg: args) {
            System.out.println(arg);
        }
    }
}
```

5.11 The `break` Statement

- In addition to defining the exit points from cases in a `switch` statement, the `break` statement stops the execution of a loop:

```
while (someCondition) {
    // Do something
    if (someOtherCondition) {
        break;
    }
    // Do something else
}
```

```
public class SearchForNumber {
    public static void main (String[] args) {
        int[] nums = {1, 5, 4, 43, -2, 6, 4, 9 };
        int search = 4;
        for (int i = 0; i < nums.length; i++) {
            if (nums[i] == search) {
                System.out.println("Found " + search + " at position " + i);
                break;
            }
        }
    }
}
```

5.12 The `continue` Statement

- Skips one execution of a loop's body
- With `continue`, this:

```
while (someCondition) {
    if (someOtherCondition) {
        continue;
    }
    // Do something
}
```

works the same as this:

```
while (someCondition) {
    if (!someOtherCondition) {
        // Do something
    }
}
```

```
public class SkipOddNumbers {
    public static void main(String[] args) {
        int[] nums = { 0, 3, 4, 5, 7, 2, 6, 9, 8, 7, 1 };
        for (int i = 0; i < nums.length; i++) {
            if (nums[i] % 2 != 0) {
                continue;
            }
            System.out.println(nums[i] + " is even");
        }
    }
}
```

5.13 Nested Loops and Labels

- Looping structures can be nested.
  - By default, a `break` or `continue` statement affects the innermost loop in which it is located.
- You can apply an optional *label* to a looping structure.
  - The label is an identifier followed by a `:` placed before the looping structure.
- Both the `break` and the `continue` statement accept an optional label argument.
  - In that case, the `break` or `continue` statement affects the looping structure with that label.

```
OUTER:
for (int i = 1; i <= 10; i++) {
    for (int j = 1; j <= 10; j++) {
        if (i == j) continue OUTER;
    }
}
```

```
public class SearchForNumber2D {
    public static void main (String[] args) {
        int[][] nums = { {1, 3, 7, 5},
                         {5, 8, 4, 6},
                         {7, 4, 2, 9} };
        int search = 4;
        foundNumber:
        for (int i = 0; i < nums.length; i++) {
            for (int j = 0; j < nums[i].length; j++) {
                if (nums[i][j] == search) {
                    System.out.println(
                        "Found " + search + " at position " + i + "," + j);
                    break foundNumber;
                }
            }
        }
    }
}
```

Chapter 6

Object Oriented Programming in Java

Introduction to Object Oriented Programming in Java

6.1 What is Object Oriented Programming (OOP)?

- A software design method that models the characteristics of real or abstract objects using software classes and objects.
- Characteristics of objects:
  - State (what the objects have)
  - Behavior (what the objects do)
  - Identity (what makes them unique)
- Definition: an object is a software bundle of related *fields* (variables) and *methods*.
- In OOP, a program is a collection of objects that act on one another (vs. procedures).

For example, a car is an object. Its state includes current:

- Speed
- RPM
- Gear
- Direction
- Fuel level
- Engine temperature

Its behaviors include:

- Change Gear
- Go faster/slower
- Go in reverse
- Stop

- Shut-off

Its identity is:

- VIN
- License Plate

6.2 Why OOP?

- *Modularity* — Separating entities into separate logical units makes them easier to code, understand, analyze, test, and maintain.
- *Data hiding* (*encapsulation*) — The implementation of an object's private data and actions can change without affecting other objects that depend on it.
- Code reuse through:
  - *Composition* — Objects can contain other objects
  - *Inheritance* — Objects can inherit state and behavior of other objects
- Easier design due to natural modeling

Even though OOP takes some getting used to, its main benefit is to make it easier to solve real-world problems by modeling natural objects in software objects. The OO thought process is more intuitive than procedural, especially for tackling complex problems.

Although a lot of great software is implemented in procedural languages like C, OO languages typically scale better for taking on medium to large software projects.

6.3 Class vs. Object

- A *class* is a template or blueprint for how to build an object.
  - A class is a prototype that defines state placeholders and behavior common to all objects of its kind.
  - Each object is a member of a single class — there is no multiple inheritance in Java.
- An *object* is an instance of a particular class.
  - There are typically many object instances for any one given class (or type).
  - Each object of a given class has the same built-in behavior but possibly a different state (data).
  - Objects are *instantiated* (created).

For example, each car starts of with a design that defines its features and properties.

It is the design that is used to build a car of a particular type or class.

When the physical cars roll off the assembly line, those cars are *instances* (concrete objects) of that class.

Many people can have a 2007 BMW 335i, but there is typically only one design for that particular class of cars.

As we will see later, classification of objects is a powerful idea, especially when it comes to inheritance — or classification hierarchy.

6.4 Classes in Java

- Everything in Java is defined in a class.
- In its simplest form, a class just defines a collection of data (like a record or a C `struct`). For example:

```
class Employee {
    String name;
    String ssn;
    String emailAddress;
    int yearOfBirth;
}
```

- The order of data fields and methods in a class is not significant.

If you recall, each class must be saved in a file that matches its name, for example: `Employee.java`

There are a few exceptions to this rule (for non-`public` classes), but the accepted convention is to have one class defined per source file.

Note that in Java, `Strings` are also classes rather than being implemented as primitive types.

Unlike local variables, the state variables (known as *fields*) of objects do not have to be explicitly initialized. Primitive fields (such as `yearOfBirth`) are automatically set to primitive defaults (`0` in this case), whereas objects (`name`, `ssn`, `emailAddress`) are automatically set to `null` — meaning that they do not point to any object.

6.5 Objects in Java

- To create an object (instance) of a particular class, use the `new` operator, followed by an invocation of a *constructor* for that class, such as:

  ```
  new MyClass()
  ```

  - The constructor method initializes the state of the new object.
  - The `new` operator returns a *reference* to the newly created object.
- As with primitives, the variable type must be compatible with the value type when using object references, as in:

  ```
  Employee e = new Employee();
  ```

- To access member data or methods of an object, use the dot (`.`) notation: *`variable.field`* or *`variable.method`*`()`

We'll explore all of these concepts in more depth in this module.

Consider this simple example of creating and using instances of the `Employee` class:

```
public class EmployeeDemo {
    public static void main(String[] args) {
        Employee e1 = new Employee();
        e1.name = "John";
        e1.ssn = "555-12-345";
        e1.emailAddress = "john@company.com";

        Employee e2 = new Employee();
        e2.name = "Tom";
        e2.ssn = "456-78-901";
        e2.yearOfBirth = 1974;

        System.out.println("Name: " + e1.name);
        System.out.println("SSN: " + e1.ssn);
        System.out.println("Email Address: " + e1.emailAddress);
        System.out.println("Year Of Birth: " + e1.yearOfBirth);

        System.out.println("Name: " + e2.name);
        System.out.println("SSN: " + e2.ssn);
        System.out.println("Email Address: " + e2.emailAddress);
        System.out.println("Year Of Birth: " + e2.yearOfBirth);

    }
}
```

Running this code produces:

```
Name: John
SSN: 555-12-345
Email Address: john@company.com
Year Of Birth: 0
Name: Tom
SSN: 456-78-901
Email Address: null
Year Of Birth: 1974
```

6.6 Java Memory Model

- Java variables do not contain the actual objects, they contain *references* to the objects.
  - The actual objects are stored in an area of memory known as the *heap*.
  - Local variables referencing those objects are stored on the stack.
  - More than one variable can hold a reference to the same object.

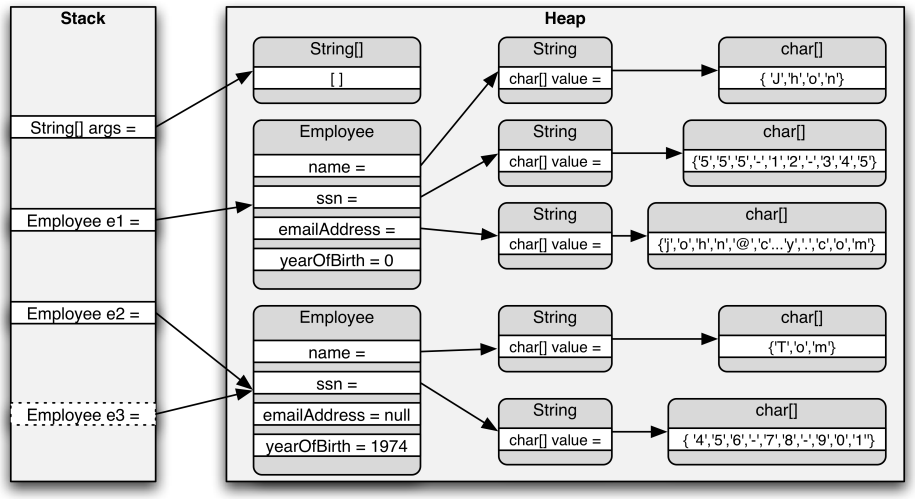


Figure 6.1: Java Memory Model

As previously mentioned, the *stack* is the area of memory where local variables (including method parameters) are stored. When it comes to object variables, these are merely *references* (pointers) to the actual objects on the heap.

Every time an object is instantiated, a chunk of *heap memory* is set aside to hold the data (state) of that object. Since objects can contain other objects, some of this data can in fact hold references to those nested objects.

In Java:

- Object references can either point to an actual object of a compatible type, or be set to `null` (`0` is not the same as `null`).
- It is not possible to instantiate objects on the stack. Only local variables (primitives and object references) can live on the stack, and everything else is stored on the heap, including classes and static data.

6.7 Accessing Objects through References

```
Employee e1 = new Employee();
Employee e2 = new Employee();

// e1 and e2 refer to two independent Employee objects on the heap

Employee e3 = e1;

// e1 and e3 refer to the *same* Employee object

e3 = e2;

// Now e2 and e3 refer to the same Employee object

e1 = null;

// e1 no longer refers to any object. Additionally, there are no references
// left to the Employee object previously referred to by e1. That "orphaned"
// object is now eligible for garbage collection.
```

Note
The statement `Employee e3 = e2;` sets `e3` to point to the same physical object as `e2`. It does **not** duplicate the object. Changes to `e3` are reflected in `e2` and vice-versa.

6.8 Garbage Collection

- Unlike some OO languages, Java does not support an explicit *destructor* method to delete an object from memory.
  - Instead, unused objects are deleted by a process known as *garbage collection*.
- The JVM automatically runs garbage collection periodically. Garbage collection:
  - Identifies objects no longer in use (no references)
  - Finalizes those objects (deconstructs them)
  - Frees up memory used by destroyed objects
  - Defragments memory
- Garbage collection introduces overhead, and can have a major affect on Java application performance.
  - The goal is to avoid how often and how long GC runs.
  - Programmatically, try to avoid unnecessary object creation and deletion.
  - Most JVMs have tuning parameters that affect GC performance.

Benefits of garbage collection:

- Frees up programmers from having to manage memory. Manually identifying unused objects (as in a language such as C++) is not a trivial task, especially when programs get so complex that the responsibility of object destruction and memory deallocation becomes vague.
- Ensures integrity of programs:
  - Prevents memory leaks — each object is tracked down and disposed off as soon as it is no longer used.
  - Prevents deallocation of objects that are still in use or have already been released. In Java it is impossible to explicitly deallocate an object or use one that has already been deallocated. In a language such as C++ dereferencing null pointers or double-freeing objects typically crashes the program.

Through Java command-line switches (`java -X`), you can:

- Set minimum amount of memory (e.g. `-Xmn`)
- Set maximum amount of memory (e.g. `-Xmx`, `-Xss`)
- Tune GC and memory integrity (e.g. `-XX:+UseParallelGC`)

For more information, see: http://java.sun.com/docs/hotspot/VMOptions.html and http://www.petefreitag.com/articles/gctuning/-

6.9 Methods in Java

- A *method* is a set of instructions that defines a particular behavior.
  - A method is also known as a *function* or a *procedure* in procedural languages.
- Java allows procedural programming through its *static* methods (e.g. the `main()` method).
  - A static method belongs to a class independent of any of the class's instances.
- It would be possible to implement an entire program through static methods, which call each other procedurally, but that is not OOP.

```
public class EmployeeDemo {
    public static void main(String[] args) {
        Employee e1 = new Employee();
        e1.name = "John";
        e1.ssn = "555-12-345";
        e1.emailAddress = "john@company.com";

        Employee e2 = new Employee();
        e2.name = "Tom";
        e2.ssn = "456-78-901";
        e2.yearOfBirth = 1974;

        printEmployee(e1);
        printEmployee(e2);
    }

    static void printEmployee(Employee e) {
        System.out.println("Name: " + e.name);
        System.out.println("SSN: " + e.ssn);
        System.out.println("Email Address: " + e.emailAddress);
        System.out.println("Year Of Birth: " + e.yearOfBirth);
    }
}
```

Running this code produces the same output as before:

```
Name: John
SSN: 555-12-345
Email Address: john@company.com
Year Of Birth: 0
Name: Tom
SSN: 456-78-901
Email Address: null
Year Of Birth: 1974
```

6.10 Methods in Java (cont.)

- In true OOP, we combine an object's state and behavior together.
  - For example, rather than having external code access the individual fields of an Employee object and print the values, an Employee object could know how to print itself:

```
class Employee {
    String name;
    String ssn;
    String emailAddress;
    int yearOfBirth;

    void print() {
        System.out.println("Name: " + name);
        System.out.println("SSN: " + ssn);
        System.out.println("Email Address: " + emailAddress);
        System.out.println("Year Of Birth: " + yearOfBirth);
    }
}
```

```
public class EmployeeDemo {
    public static void main(String[] args) {
        Employee e1 = new Employee();
        e1.name = "John";
        e1.ssn = "555-12-345";
        e1.emailAddress = "john@company.com";

        Employee e2 = new Employee();
        e2.name = "Tom";
        e2.ssn = "456-78-901";
        e2.yearOfBirth = 1974;

        e1.print();
        e2.print();
    }
}
```

Running this code produces the same output as before:

```
Name: John
SSN: 555-12-345
Email Address: john@company.com
Year Of Birth: 0
Name: Tom
SSN: 456-78-901
Email Address: null
Year Of Birth: 1974
```

6.11 Method Declarations

Each method has a *declaration* of the following format:

```
modifiers returnType name(params) throws-clause { body }
```

modifiers
: `public`, `private`, `protected`, `static`, `final`, `abstract`, `native`, `synchronized`

returnType
: A primitive type, object type, or `void` (no return value)

name
The name of the method

params
paramType paramName, ...

throws-clause
`throws` *ExceptionType*, ...

body
The method's code, including the declaration of local variables, enclosed in braces

---

Note that abstract methods do not have a body (more on this later).

Here are some examples of method declarations:

```
public static void print(Employee e) { ... }
public void print() { ... }
public double sqrt(double n) { ... }
public int max(int x, int y) { ... }
public synchronized add(Employee e) throws DuplicateEntryException { ... }
public int read() throws IOException { ... }
public void println(Object o) { ... }
protected void finalize() throws Throwable { ... }
public native void write(byte[] buffer, int offset, int length) throws IOException { ... }
public boolean equals(Object o) { ... }
private void process(MyObject o) { ... }
void run() { ... }
```

6.12 Method Signatures

- The *signature* of a method consists of:
  - The method name
  - The parameter list (that is, the parameter types and their order)
- The signature does **not** include:
  - The parameter names
  - The return type
- Each method defined in a class must have a unique signature.
- Methods with the same name but different signatures are said to be *overloaded*.

We'll discuss examples of overloading constructors and other methods later in this module.

6.13 Invoking Methods

- Use the dot (`.`) notation to invoke a method on an object: *`objectRef.method(params)`*
- Parameters passed into methods are always copied ("pass-by-value").
  - Changes made to parameter variables within the methods do no affect the caller.
  - Object references are also copied, but they still point to the *same object*.

For example, we add the following method to our Employee class:

```
void setYearOfBirth(int year) {
    yearOfBirth = year;
    year = -1;      // modify local variable copy
}
```

We invoke it from `EmployeeDemo.main()` as:

```
int y = 1974;
e2.setYearOfBirth(y);
System.out.println(e2.yearOfBirth); // prints 1974
System.out.println(y);              // prints 1974
```

On the other hand, we add this method to our EmployeeDemo class:

```
static void printYearOfBirth(Employee e) {
    System.out.println(e.yearOfBirth);
    e.yearOfBirth = -1; // modify object's copy
}
```

We invoke it from `EmployeeDemo.main()` as:

```
printYearOfBirth(e2);               // prints 1974
System.out.println(e2.yearOfBirth); // prints -1
```

6.14 Static vs. Instance Data Fields

- Static (or class) data fields
  - Unique to the entire class
  - Shared by all instances (objects) of that class
  - Accessible using *`ClassName.fieldName`*
  - The class name is optional within static and instance methods of the class, unless a local variable of the same name exists in that scope
  - Subject to the declared access mode, accessible from outside the class using the same syntax
- Instance (object) data fields
  - Unique to each instance (object) of that class (that is, each object has its own set of instance fields)
  - Accessible within instance methods and constructors using `this.`*`fieldName`*
  - The `this.` qualifier is optional, unless a local variable of the same name exists in that scope
  - Subject to the declared access mode, accessible from outside the class from an object reference using *`objectRef.fieldName`*

Say we add the following to our Employee class:

```
static int vacationDays = 10;
```

and we print this in the Employee's `print()` method:

```
System.out.println("Vacation Days: " + vacationDays);
```

In the EmployeeDemo's `main()` method, we change `vacationDays` to 15:

```
Employee.vacationDays = 15;
```

Now, `e1.print()` and `e2.print()` will both show the vacation days set to 15. This is because both `e1` and `e2` (and any other Employee object) share the static `vacationDays` integer field.

The field `vacationDays` is part of the Employee class, and this is also stored on the heap, where it is shared by all objects of that class.

Static fields that are not protected (which we will soon learn how to do) are almost like global variables — accessible to anyone.

Note that it is possible to access static fields through instance variables (e.g., `e1.vacationDays = 15;` will have the same effect), however this is discouraged. You should always access static fields by *`ClassName.staticFieldName`*, unless you are within the same class, in which case you can just say *`staticFieldName`*.

6.15 Static vs. Instance Methods

- Static methods can access only static data and invoke other static methods.
  - Often serve as helper procedures/functions
  - Use when the desire is to provide a utility or access to class data only
- Instance methods can access both instance and static data and methods.
  - Implement behavior for individual objects
  - Use when access to instance data/methods is required
- An example of static method use is Java's Math class.
  - All of its functionality is provided as static methods implementing mathematical functions (e.g., `Math.sin()`).
  - The Math class is designed so that you don't (and can't) create actual Math instances.
- Static methods also are used to implement *factory methods* for creating objects, a technique discussed later in this class.

```
class Employee {
    String name;
    String ssn;
    String emailAddress;
    int yearOfBirth;
    int extraVacationDays = 0;
    static int baseVacationDays = 10;
```

```
    Employee(String name, String ssn) {
        this.name = name;
        this.ssn = ssn;
    }

    static void setBaseVacationDays(int days) {
        baseVacationDays = days < 10? 10 : days;
    }

    static int getBaseVacationDays() {
        return baseVacationDays;
    }

    void setExtraVacationDays(int days) {
        extraVacationDays = days < 0? 0 : days;
    }

    int getExtraVacationDays() {
        return extraVacationDays;
    }

    void setYearOfBirth(int year) {
        yearOfBirth = year;
    }

    int getVacationDays() {
        return baseVacationDays + extraVacationDays;
    }

    void print() {
        System.out.println("Name: " + name);
        System.out.println("SSN: " + ssn);
        System.out.println("Email Address: " + emailAddress);
        System.out.println("Year Of Birth: " + yearOfBirth);
        System.out.println("Vacation Days: " + getVacationDays());
    }
}
```

To change the company vacation policy, do `Employee.setBaseVacationDays(15);`

To give one employee extra vacation, do `e2.setExtraVacationDays(5);`

6.16 Method Overloading

- A class can provide multiple definitions of the same method. This is known as *overloading*.
- Overloaded methods *must* have distinct signatures:
  - The parameter type list must be different, either different number or different order.
  - Only parameter *types* determine the signature, not parameter *names*.
  - The return type is not considered part of the signature.

We can overload the print method in our Employee class to support providing header and footer to be printed:

```
public class Employee {
    ...
```

```
    public void print(String header, String footer) {
        if (header != null) {
            System.out.println(header);
        }
        System.out.println("Name: " + name);
        System.out.println("SSN: " + ssn);
        System.out.println("Email Address: " + emailAddress);
        System.out.println("Year Of Birth: " + yearOfBirth);
        System.out.println("Vacation Days: " + getVacationDays());
        if (footer != null) {
            System.out.println(footer);
        }
    }

    public void print(String header) {
        print(header, null);
    }

    public void print() {
        print(null);
    }
}
```

In our `EmployeeDemo.main()` we can then do:

```
e1.print("COOL EMPLOYEE");
e2.print("START OF EMPLOYEE", "END OF EMPLOYEE");
```

6.17 Variable Argument Length Methods

- Java 5 introduced syntax supporting methods with a variable number of argument (also known as *varargs*).
  - The last parameter in the method declaration must have the format *`Type...  varName`* (literally three periods following the type).
  - The arguments corresponding to the parameter are presented as an *array* of that type.
  - You can also invoke the method with an explicit array of that type as the argument.
  - If no arguments are provided corresponding to the parameter, the result is an array of length 0.
- There can be at most one varargs parameter in a method declaration.
- The varargs parameter must be the last parameter in the method declaration.

For example, consider the following implementation of a `max()` method:

```
public int max(int... values) {
    int max = Integer.MIN_VALUE;
    for (int i: values) {
        if (i > max) max = i;
    }
    return max;
}
```

You can then invoke the `max()` method with any of the following:

```
max(1, -2, 3, -4);
max(1);
max();
int[] myValues = {5, -7, 26, -13, 42, 361};
max(myValues);
```

6.18 Constructors

- Constructors are like special methods that are called implicitly as soon as an object is instantiated (i.e. on `new ClassName()`).
  - Constructors have no return type (not even `void`).
  - The constructor name must match the class name.
- If you don't define an explicit constructor, Java assumes a default constructor
  - The default constructor accepts no arguments.
  - The default constructor automatically invokes its base class constructor with no arguments, as discussed later in this module.
- You can provide one or more explicit constructors to:
  - Simplify object initialization (one line of code to create and initialize the object)
  - Enforce the state of objects (require parameters in the constructor)
  - Invoke the base class constructor with arguments, as discussed later in this module.
- Adding *any* explicit constructor disables the implicit (no argument) constructor.

We can add a constructor to our Employee class to allow/require that it be constructed with a name and a social security number:

```
class Employee {
    String name;
    String ssn;
    ...
    Employee(String name, String ssn) {
        this.name = name; // "this." helps distinguish between
        this.ssn = ssn; // instance and parameter variables
    }
    ...
}
```

Then we can modify `EmployeeDemo.main()` to call the specified constructor:

```
public class EmployeeDemo {
    public static void main(String[] args) {
        Employee e1 = new Employee("John", "555-12-345");
        e1.emailAddress = "john@company.com";
        Employee e2 = new Employee("Tom", "456-78-901");
        e2.setYearOfBirth(1974);
        ...
    }
}
```

6.19 Constructors (cont.)

- As with methods, constructors can be overloaded.
- Each constructor must have a unique signature.
  - The parameter type list must be different, either different number or different order.
  - Only parameter *types* determine the signature, not parameter *names*.
- One constructor can invoke another by invoking `this(param1, param2, ...)` as the *first* line of its implementation.

It is no longer possible to do `Employee e = new Employee();` because there is no constructor that takes no parameters.

We could add additional constructors to our class Employee:

```
Employee(String ssn) { // employees must have at least a SSN
    this.ssn = ssn;
}

Employee(String name, String ssn) {
    this(ssn);
    this.name = name;
}

Employee(String name, String ssn, String emailAddress) {
    this(name, ssn);
    this.emailAddress = emailAddress;
}

Employee(String ssn, int yearOfBirth) {
    this(ssn);
    this.yearOfBirth = yearOfBirth;
}
```

Now we can construct Employee objects in different ways:

```
Employee e1 = new Employee("John", "555-12-345", "john@company.com");
Employee e2 = new Employee("456-78-901", 1974);
e2.name = "Tom";
```

6.20 Constants

- "Constant" fields are defined using the `final` keyword, indicating their values can be assigned only once.
  - Final instance fields must be initialized by the end of object construction.
  - Final static fields must be initialized by the end of class initialization.
  - Final local variables must be initialized only once before they are used.
  - Final method parameters are initialized on the method call.

Important
Declaring a reference variable as `final` means only that once initialized to refer to an object, it can't be changed to refer to another object. It does **not** imply that the state of the object referenced cannot be changed.

- Final static field can be initialized through direct assignment or by using a *static initializer*.
  - A static initializer consists of the keyword `static` followed by a block, for example:

```
private static int[] values = new int[10];
static {
    for (int i = 0; i < values.length; i++) {
        values[i] = (int) (100.0 * Math.random());
    }
}
```

If we declare `ssn` as final, we then must either assign it right away or initialize it in all constructors. This can also be done indirectly via constructor-to-constructor calls. Once initialized, final instance fields (e.g. `ssn`) can no longer be changed.

```
class Employee {
    final String ssn;
    ...
    Employee(String ssn) {
        this.ssn = ssn;
    }
    ...
}
```

Local variables can also be set as final, to indicate that they should not be changed once set:

```
public class EmployeeDemo {
    public static void main(String[] args) {
        final Employee e1 = new Employee(...);
        final Employee e2 = new Employee("456-78-901", 1974);
        final Employee e3;
        e3 = e2;
        ...
    }
}
```

6.21 Encapsulation

- The principle of *encapsulation* is that all of an object's data is contained and hidden in the object and access to it restricted to methods of that class.
  - Code outside of the object cannot (or at least should not) directly access object fields.
  - All access to an object's fields takes place through its methods.
- Encapsulation allows you to:
  - Change the way in which the data is actually stored without affecting the code that interacts with the object
  - Validate requested changes to data
  - Ensure the consistency of data — for example preventing related fields from being changed independently, which could leave the object in an inconsistent or invalid state
  - Protect the data from unauthorized access
  - Perform actions such as notifications in response to data access and modification

---

More generally, encapsulation is a technique for *isolating change*. By hiding internal data structures and processes and publishing only well-defined methods for accessing your objects,

6.22 Access Modifiers: Enforcing Encapsulation

- *Access modifiers* are Java keywords you include in a declaration to control access.
- You can apply access modifiers to:
  - Instance and static fields
  - Instance and static methods
  - Constructors
  - Classes
  - Interfaces (discussed later in this module)
- Two access modifiers provided by Java are:

`private`
: visible only within the same class

`public`
: visible everywhere

Note
There are two additional access levels that we'll discuss in the next module.

6.23 Accessors (Getters) and Mutators (Setters)

- A common model for designing data access is the use of *accessor* and *mutator* methods.
- A mutator — also known as a *setter* — changes some property of an object.
  - By convention, mutators are usually named `set`*`PropertyName`*.
- An accessor — also known as a *getter* — returns some property of an object.
  - By convention, accessors are usually named `get`*`PropertyName`*.
  - One exception is that accessors that return a `boolean` value are commonly named `is`*`PropertyName`*.
- Accessors and mutators often are declared `public` and used to access the property outside the object.
  - Using accessors and mutators from code within the object also can be beneficial for side-effects such as validation, notification, etc.
  - You can omit implementing a mutator — or mark it `private` — to implement *immutable* (unchangeable) object properties.

We can update our Employee class to use access modifiers, accessors, and mutators:

```
public class Employee {
    private String name;
    private final String ssn;
    ...
    public void setName(String name) {
        if (name != null && name.length() > 0) {
            this.name = name;
        }
    }
```

```
    public String getName() {
        return this.name;
    }

    public String getSsn() {
        return this.ssn;
    }
    ...
}
```

Now, to set the name on an employee in `EmployeeDemo.main()` (i.e., from the outside), you must call:

```
e2.setName("Tom");
```

as opposed to:

```
e2.name = "Tom"; // won't compile, name is hidden externally
```

6.24 Inheritance

- *Inheritance* allows you to define a class based on the definition of another class.
  - The class it inherits from is called a *base class* or a *parent class*.
  - The derived class is called a *subclass* or *child class*.
- Subject to any access modifiers, which we'll discuss later, the subclass gets access to the fields and methods defined by the base class.
  - The subclass can add its own set of fields and methods to the set it inherits from its parent.
- Inheritance simplifies modeling of real-world hierarchies through generalization of common features.
  - Common features and functionality is implemented in the base classes, facilitating code reuse.
  - Subclasses can extended, specialize, and override base class functionality.

Inheritance provides a means to create specializations of existing classes. This is referred to as *sub-typing*. Each subclass provides specialized state and/or behavior in addition to the state and behavior inherited by the parent class.

For example, a manager is also an employee but it has a responsibility over a department, whereas a generic employee does not.

6.25 Inheritance, Composition, and Aggregation

- Complex class structures can be built through inheritance, composition, and aggregation.
- Inheritance establishes an "is-a" relationship between classes.
  - The subclass has the same features and functionality as its base class, with some extensions.
  - A Car **is-a** Vehicle. An Apple **is-a** Food.
- Composition and aggregation are the construction of complex classes that incorporate other objects.

  - They establish a "has-a" relationship between classes.
  - A Car **has-a** Engine. A Customer **has-a** CreditCard.
- In *composition*, the component object exists solely for the use of its composite object.
  - If the composite object is destroyed, the component object is destroyed as well.
  - For example, an Employee **has-a** name, implemented as a String object. There is no need to retain the String object once the Employee object has been destroyed.
- In *aggregation*, the component object can (but isn't required to) have an existence independent of its use in the aggregate object.
  - Depending on the structure of the aggregate, destroying the aggregate may or may not destroy the component.
  - For example, let's say a Customer **has-a** BankAccount object. However, the BankAccount might represent a joint account owned by multiple Customers. In that case, it might not be appropriate to delete the BankAccount object just because one Customer object is deleted from the system.

Notice that composition and aggregation represent another aspect of encapsulation. For example, consider a Customer class that includes a customer's birthdate as a property. We could define a Customer class in such a way that it duplicates all of the fields and methods from and existing class like Date, but duplicating code is almost always a bad idea. What if the data or methods associated with a Date were to change in the future? We would also have to change the definitions for our Customer class. Whenever a set of data or methods are used in more than one place, we should look for opportunities to encapsulate the data or methods so that we can reuse the code and isolate any changes we might need to make in the future.

Additionally when designing a class structure with composition or aggregation, we should keep in mind the principle of encapsulation when deciding where to implement functionality. Consider implementing a method on Customer that would return the customer's age. Obviously, this depends on the birth date of the customer stored in the Date object. But should we have code in Customer that calculates the elapsed time since the birth date? It would require the Customer class to know some very Date-specific manipulation. In this case, the code would be *tightly coupled* –- a change to Date is more likely to require a change to Customer as well. It seems more appropriate for Date objects to know how to calculate the elapsed time between two instances. The Customer object could then *delegate* the age request to the Date object in an appropriate manner. This makes the classes *loosely coupled* –- so that a change to Date is unlikely to require a change to Customer.

6.26 Inheritance in Java

- You define a subclass in Java using the `extends` keyword followed by the base class name. For example:

```
class Car extends Vehicle { // ... }
```

  - In Java, a class can extend at most one base class. That is, multiple inheritance is not supported.
  - If you don't explicitly extend a base class, the class inherits from Java's Object class, discussed later in this module.
- Java supports multiple *levels* of inheritance.
  - For example, Child can extend Parent, which in turn extends GrandParent, and so on.

```
class A {
    String a = null;
    void doA() {
        System.out.println("A says " + a);
    }
}
```

```
class B extends A {
    String b = null;
    void doB() {
        System.out.println("B says " + b);
    }
}
class C extends B {
    String c = null;
    void doA() {
        System.out.println("Who cares what A says");
    }
    void doB() {
        System.out.println("Who cares what B says");
    }
    void doC() {
        System.out.println("C says " + a + " " + b + " " + c);
    }
}
public class ABCDemo {
    public static void main(String[] args) {
        A a = new A();
        B b = new B();
        C c = new C();

        a.a = "AAA";
        b.a = "B's A";
        b.b = "BBB";
        c.a = "Who cares";
        c.b = "Whatever";
        c.c = "CCC";

        a.doA();
        b.doB();
        c.doA();
        c.doB();
        c.doC();
    }
}
```

The output of running ABCDemo is:

```
A says AAA
B says B's AAAs, BBB
Who cares what A says
Who cares what B says
C says Who cares Whatever CCC
```

6.27 Invoking Base Class Constructors

- By default, Java automatically invokes the base class's constructor with **no** arguments before invoking the subclass's constructor.
  - This might not be desirable, especially if the base class doesn't have a no-argument constructor. (You get a compilation error in that case.)
- You can explicitly invoke a base class constructor with any arguments you want with the syntax `super(arg1, arg2, ...)`. For example:

```
class Subclass extends ParentClass {
    public Subclass(String name, int age) {
        super(name);
        // Additional Subclass initialization...
    }
}
```

– The call to `super()` must be the **first** statement in a subclass constructor.

Note
A single constructor cannot invoke both `super()` and `this()`. However, a constructor can use `this()` to invoke an overloaded constructor, which in turn invokes `super()`.

6.28 Overriding vs. Overloading

- The subclass can *override* its parent class definition of fields and methods, replacing them with its own definitions and implementations.
  - To successfully override the base class method definition, the subclass method must have the same signature.
  - If the subclass defines a method with the same name as one in the base class but a different signature, the method is overloaded not overridden.
- A subclass can explicitly invoked an ancestor class's implementation of a method by prefixing `super.` to the method call. For example:

```
class Subclass extends ParentClass {
    public String getDescription() {
        String parentDesc = super.getDescription();
        return "My description\n" + parentDesc;
    }
}
```

Consider defining a Manager class as a subclass of Employee:

```
public class Manager extends Employee {
    private String responsibility;

    public Manager(String name, String ssn, String responsibility) {
        super(name, ssn);
        this.responsibility = responsibility;
    }

    public void setResponsibility(String responsibility) {
        this.responsibility = responsibility;
    }

    public String getResponsibility() {
        return this.responsibility;
    }

    public void print(String header, String footer) {
        super.print(header, null);
        System.out.println("Responsibility: " + responsibility);
```

```
        if (footer != null) {
            System.out.println(footer);
        }
    }
}
```

Now with code like this:

```
public class EmployeeDemo {
    public static void main(String[] args) {
        ...
        Manager m1 = new Manager("Bob", "345-11-987", "Development");
        Employee.setBaseVacationDays(15);
        m1.setExtraVacationDays(10);
        ...
        m1.print("BIG BOSS");
    }
}
```

The output is:

```
BIG BOSS
Name: Bob
SSN: 345-11-987
Email Address: null
Year Of Birth: 0
Vacation Days: 25
Responsibility: Development
```

Note that the Manager class must invoke one of super's constructors in order to be a valid Employee. Also observe that we can invoke a method like `setExtraVacationDays()` that is defined in Employee on our Manager instance `m1`.

6.29 Polymorphism

- *Polymorphism* is the ability for an object of one type to be treated as though it were another type.
- In Java, inheritance provides us one kind of polymorphism.
  - An object of a subclass can be treated as though it were an object of its parent class, or any of its ancestor classes. This is also known as *upcasting*.
  - For example, if Manager is a subclass of Employee:

    ```
    Employee e = new Manager(...);
    ```

  - Or if a method accepts a reference to an Employee object:

    ```
    public void giveRaise(Employee e) { // ... }
    // ...
    Manager m = new Manager(...);
    giveRaise(m);
    ```

Why is polymorphism useful? It allows us to create more generalized programs that can be extended more easily.

Consider an online shopping application. You might need to accept multiple payment methods, such as credit cards, debit card, direct bank debit through ACH, etc. Each payment method might be implemented as a separate class because of differences in the way you need to process credits, debits, etc.

If you were to handle each object type explicitly, the application would be very complex to write. It would require `if-else` statements everywhere to test for the different types of payment methods, and overloaded methods to pass different payment type objects as arguments.

On the other hand, if you define a base class like PaymentMethod and then derive subclasses for each type, then it doesn't matter if you've instantiated a CreditCard object or a DebitCard object, you can treat it as a PaymentMethod object.

6.30 More on Upcasting

- Once you have upcast an object reference, you can access only the fields and methods declared by the base class.
  - For example, if Manager is a subclass of Employee:

    ```
    Employee e = new Manager(...);
    ```
  - Now using `e` you can access only the fields and methods declared by the Employee class.
- However, if you invoke a method on `e` that is defined in Employee but overridden in Manager, the Manager version is executed.
  - For example:

    ```
    public class A {
        public void print() {
            System.out.println("Hello from class A");
        }
    }
    public class B extends A {
        public void print() {
            System.out.println("Hello from class B");
        }
    }
    // ...
    A obj = new B();
    obj.print();
    ```

    In the case, the output is "Hello from class B".

From within a subclass, you can explicitly invoke a base class's version of a method by using the `super.` prefix on the method call.

6.31 Downcasting

- An upcast reference can be *downcast* to a subclass through explicit casting. For example:

  ```
  Employee e = new Manager(...);
  // ...
  Manager m = (Manager) e;
  ```
  - The object referenced must actually be a member of the downcast type, or else a `ClassCastException` run-time exception occurs.
- You can test if an object is a member of a specific type using the `instanceof` operator, for example:

  ```
  if (obj instanceof Manager) { // We've got a Manager object }
  ```

```
public class EmployeeDemo {
    public static void main(String[] args) {
        final Employee e1 = new Employee("John", "555-12-345", "john@company.com");
        final Employee e2 = new Employee("456-78-901", 1974);
        e2.setName("Tom");
        Employee em = new Manager("Bob", "345-11-987", "Development");

        Employee.setBaseVacationDays(15);
        e2.setExtraVacationDays(5);
        em.setExtraVacationDays(10);

        if (em instanceof Manager) {
            Manager m = (Manager) em;
            m.setResponsibility("Operations");
        }

        e1.print("COOL EMPLOYEE");
        e2.print("START OF EMPLOYEE", "END OF EMPLOYEE");
        em.print("BIG BOSS");
    }
}
```

This would print:

```
BIG BOSS
Name: Bob
SSN: 345-11-987
Email Address: null
Year Of Birth: 0
Vacation Days: 25
Responsibility: Operations
```

6.32 Abstract Classes and Methods

- An *abstract class* is a class designed solely for subclassing.
  - You can't create actual instances of the abstract class. You get a compilation error if you attempt to do so.
  - You design abstract classes to implement common sets of behavior, which are then shared by the *concrete* (instantiable) classes you derive from them.
  - You declare a class as abstract with the `abstract` modifier:

    ```
    public abstract class PaymentMethod { // ... }
    ```

- An *abstract method* is a method with no body.
  - It declares a method signature and return type that a concrete subclass must implement.
  - You declare a method as abstract with the `abstract` modifier and a semicolon terminator:

    ```
    public abstract boolean approveCharge(float amount);
    ```

  - If a class has any abstract methods declared, the class itself must also be declared as abstract.

For example, we could create a basic triangle class:

```
public abstract class Triangle implements Shape {
  public abstract double getA();
  public abstract double getB();
  public abstract double getC();
  public double getPerimeter() {
    return getA() + getB() + getC();
  }
  // getArea() is also abstract since it is not implemented
}
```

Now we can create concrete triangle classes based on their geometric properties. For example:

```
public class RightAngledTriangle extends Triangle {
  private double a, b, c;
  public RightAngledTriangle(double a, double b) {
    this.a = a;
    this.b = b;
    this.c = Math.sqrt(Math.pow(a, 2) + Math.pow(b, 2));
  }
  public double getA() { return a; }
  public double getB() { return b; }
  public double getC() { return c; }
  public double getArea() { return (a * b) / 2; }
}
```

6.33 Interfaces

- An *interface* defines a set of methods, without actually defining their implementation.
  - A class can then *implement* the interface, providing actual definitions for the interface methods.
- In essence, an interface serves as a "contract" defining a set of capabilities through method signatures and return types.
  - By implementing the interface, a class "advertises" that it provides the functionality required by the interface, and agrees to follow that contract for interaction.

The concept of an interface is the cornerstone of object oriented (or modular) programming. Like the rest of OOP, interfaces are modeled after real world concepts/entities.

For example, one must have a driver's license to drive a car, regardless for what kind of a car that is (i.e., make, model, year, engine size, color, style, features etc.).

However, the car must be able to perform certain operations:

- Go forward
- Slowdown/stop (break light)
- Go in reverse
- Turn left (signal light)
- Turn right (signal light)
- Etc.

These operations are defined by an interface (contract) that defines what a car is from the perspective of **how it is used**. The interface does not concern itself with how the car is **implemented**. That is left up to the car manufacturers.

6.34 Defining a Java Interface

- Use the `interface` keyword to define an interface in Java.
  - The naming convention for Java interfaces is the same as for classes: CamelCase with an initial capital letter.
  - The interface definition consists of public abstract method declarations. For example:

```
public interface Shape {
    double getArea();
    double getPerimeter();
}
```

- All methods declared by an interface are implicitly `public abstract` methods.
  - You can omit either or both of the `public` and `static` keywords.
  - You must include the semicolon terminator after the method declaration.

Rarely, a Java interface might also declare and initialize `public static final` fields for use by subclasses that implement the interface.

- Any such field must be declared **and** initialized by the interface.
- You can omit any or all of the `public`, `static`, and `final` keywords in the declaration.

6.35 Implementing a Java Interface

- You define a class that implements a Java interface using the `implements` keyword followed by the interface name. For example:

```
class Circle implements Shape { // ... }
```

- A concrete class must then provide implementations for all methods declared by the interface.
  - Omitting any method declared by the interface, or not following the same method signatures and return types, results in a compilation error.
- An abstract class can omit implementing some or all of the methods required by an interface.
  - In that case concrete subclasses of that base class must implement the methods.
- A Java class can implement as many interfaces as needed.
  - Simply provide a comma-separated list of interface names following the `implements` keyword. For example:

```
class ColorCircle implements Shape, Color { // ... }
```

- A Java class can extend a base class **and** implement one or more interfaces.
  - In the declaration, provide the `extends` keyword and the base class name, followed by the `implements` keywords and the interface name(s). For example:

```
class Car extends Vehicle implements Possession { // ... }
```

```
public class Circle implements Shape {

    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public double getRadius() {
        return radius;
    }

    public double getArea() {
        return Math.PI * Math.pow(this.radius, 2);
    }

    public double getPerimeter() {
        return Math.PI * this.radius * 2;
    }
}

/**
 * Rectange shape with a width and a height.
 * @author sasa
 * @version 1.0
 */
public class Rectangle implements Shape {

    private double width;
    private double height;

    /**
     * Constructor.
     * @param width the width of this rectangle.
     * @param height the height of this rectangle.
     */
    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    /**
     * Gets the width.
     * @return the width.
     */
    public double getWidth() {
        return width;
    }

    /**
     * Gets the height.
     * @return the height.
     */
    public double getHeight() {
        return height;
    }

    public double getArea() {
        return this.width * this.height;
    }
```

```
    public double getPerimeter() {
        return 2 * (this.width + this.height);
    }
}

public class Square extends Rectangle {
    public Square(double side) {
        super(side, side);
    }
}
```

6.36 Polymorphism through Interfaces

- Interfaces provide another kind of polymorphism in Java.
  - An object implementing an interface can be assigned to a reference variable typed to the interface.
  - For example, if Circle implements the Shape interface:

    ```
    Shape s = new Circle(2);
    ```

  - Or you could define a method with an interface type for a parameter:

    ```
    public class ShapePrinter {
        public void print(Shape shape) {
            System.out.println("AREA: " + shape.getArea());
            System.out.println("PERIMETER: " + shape.getPerimeter());
        }
    }
    ```

- When an object reference is upcast to an interface, you can invoke only those methods declared by the interface.

The following illustrates our shapes and ShapePrinter classes:

```
public class ShapeDemo {
    public static void main(String[] args) {
        Circle c = new Circle(5.0);
        Rectangle r = new Rectangle(3, 4);
        Square s = new Square(6);

        ShapePrinter printer = new ShapePrinter();
        printer.print(c);
        printer.print(r);
        printer.print(s);

    }
}
```

This would print:

```
AREA: 78.53981633974483
CIRCUMFERENCE: 31.41592653589793
AREA: 12.0
CIRCUMFERENCE: 14.0
AREA: 36.0
CIRCUMFERENCE: 24.0
```

6.37 `Object`: Java's Ultimate Superclass

- Every class in Java ultimately has the Object class as an ancestor.
  - The Object class implements basic functionality required by all classes.
  - Often you'll want to override some of these methods to better support your custom classes.
- Behaviors implemented by Object include:
  - Equality testing and hash code calculation
  - String conversion
  - Cloning
  - Class introspection
  - Thread synchronization
  - Finalization (deconstruction)

6.38 Overriding `Object.toString()`

- The `Object.toString()` method returns a String representation of the object.
- The `toString()` method is invoked automatically:
  - When you pass an object as an argument to `System.out.println(obj)` and some other Java utility methods
  - When you provide an object as a String concatenation operand
- The default implementation returns the object's class name followed by its hash code.
  - You'll usually want to override this method to return a more meaningful value.

For example:

```
public class Circle implements Shape {
    // ...
    public String toString() {
        return "Circle with radius of " + this.radius;
    }
}
```

6.39 Object Equality

- When applied to object references, the equality operator (==) returns `true` only if the references are to the same object. For example:

```
Circle a = new Circle(2);
Circle b = new Circle(2);
Circle c = a;
if { a == b } { // false }
if { a == c } { // true }
```

One exception to this equality principle is that Java supports *string interning* to save memory (and speed up testing for equality). When the `intern()` method is invoked on a String, a lookup is performed on a table of interned Strings. If a String object with the same content is already in the table, a reference to the String in the table is returned. Otherwise, the String is added to the table and a reference to it is returned. The result is that after interning, all Strings with the same content will point to the same object.

String interning is performed on String literals automatically during compilation. At run-time you can invoke `intern()` on any String object that you want to add to the intern pool.

6.40 Object Equivalence

- The Object class provides an `equals()` method that you can override to determine if two objects are equivalent.
  - The default implementation by Object is a simple == test.
  - You should include all "significant" fields for your class when overriding `equals()`.
  - We could define a simple version of `Circle.equals` as follows:

```
public booleans equals(Object obj) {
    if (obj == this) {
        // We're being compared to ourself
        return true;
    }
    else if (obj == null || obj.getClass() != this.getClass()) {
        // We can only compare to another Circle object
        return false;
    }
    else {
        // Compare the only significant field
        Circle c = (Circle) obj;
        return c.radius == this.radius;
    }
}
```

A somewhat more complex example of an equality test is shown in this class:

```
public class Person {
    private String name;
    private final int yearOfBirth;
    private String emailAddress;

    public Person(String name, int yearOfBirth) {
        this.name = name;
        this.yearOfBirth = yearOfBirth;
    }

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }
```

```
    public int getYearOfBirth() {
        return this.yearOfBirth;
    }

    public String getEmailAddress() {
        return this.emailAddress;
    }

    public void setEmailAddress() {
        this.emailAddress = emailAddress;
    }

    public boolean equals(Object o) {
        if (o == this) {
            // we are being compared to ourself
            return true;
        } else if (o == null || o.getClass() != this.getClass()) {
            // can only compare to another person
            return false;
        } else {
            Person p = (Person) o; // cast to our type
            // compare significant fields
            return p.name.equals(this.name) &&
                p.yearOfBirth == this.yearOfBirth;
        }
    }

    public int hashCode() {
        // compute based on significant fields
        return 3 * this.name.hashCode() + 5 * this.yearOfBirth;
    }
}
```

6.41 Object Equivalence (cont.)

- Overriding `equals()` requires care. Your equality test must exhibit the following properties:
  - Symmetry: For two references, `a` and `b`, `a.equals(b)` if and only if `b.equals(a)`
  - Reflexivity: For all non-`null` references, `a.equals(a)`
  - Transitivity: If `a.equals(b)` and `b.equals(c)`, then `a.equals(c)`
  - Consistency with `hashCode()`: Two equal objects must have the same `hashCode()` value

---

Note
The `hashcode()` method is used by many Java Collections Framework classes like HashMap to identify objects in the collection. If you override the `equals()` method in your class, you **must** override `hashcode()` as well.

Tip
The Apache Commons Lang library (http://commons.apache.org/lang) includes two helper classes, EqualsBuilder and HashCodeBuilder, that can greatly simplify creating proper `equals()` and `hashcode()` implementations.

A complete discussion of the issues implementing equality tests and hashcodes is beyond the scope of this course. For more information on the issues to be aware of, refer to:

- This Stack Overflow article: http://stackoverflow.com/questions/27581/overriding-equals-and-hashcode-in-java
- The article "Java theory and practice: Hashing it out" on IBM developerWorks: http://www.ibm.com/developerworks/java/-library/j-jtp05273.html
- The book *Effective Java, 2nd ed.*, by Joshua Bloch

Chapter 7

Packaging

- Packaging Java Code
- Using Packaged Java Libraries

7.1 Why is Packaging Needed?

- Packaging protects against name collisions.
  - With thousands of Java libraries available, it's inevitable that class and interface names will be duplicated.
  - Packages allow you to use two different classes or interfaces with the same name.
- Packages hide implementation that spans multiple classes (multi-class encapsulation).
  - A set of classes might need access to each other, but still be hidden from the outside world.
  - Java packages support access levels other than `public` and `private` to support information sharing and class visibility.
- Packages help organize source code.
  - Having all source files in the same directory becomes hard to manage as the projects grow.
  - Packages allow you to group together related classes and interfaces into common directories.

Note
In addition to classes and interfaces, a Java package can contain definitions of *enumerations* and *annotations*. These Java constructs are discussed later in this class.

7.2 Packages in Java

- A Java package name consists of a set of name components separated by periods (`.`).
  - Each name component must be a valid Java identifier.
  - A component name must not start with an upper-case letter.

    Tip
    Best practice is for each name component to be all lower-case.

- The package name corresponds to a directory structure on disk where the classes are stored.
  - For example, the class and interface files for `org.xml.sax` are located in the directory `org/xml/sax`, relative to the directory where Java looks for classes (we'll discuss this later in this module).
- To add a class or interface to a package:
  - Add `package myPackageName;` as the first Java statement of the source file
  - In your development directory, store the source file in a directory structure corresponding to your package name.

Tip
Java does not require you to store your `.java` source files in in package-based directories, but it is a common convention. However, once the source files are compiled, the generated `.class` files are (and must be) stored in the package-based directories.
It's also common to have separate, parallel directory hierarchies for your `.java` and `.class` files reflecting the package structure. Having your source files separate from your class files is convenient for maintaining the source code in a source code control system without having to explicitly filter out the generated class files. Having the class files in their own independent directory hierarchy is also useful for generating JAR files, as discussed later in this module.
Most modern IDEs can create and manage your source and class file directory hierarchies automatically for you.

Caution
If you don't explicitly specify a package, your classes and interfaces end up in an unnamed packaged, also known as the *default package*. Best practice is not to use the default package for any production code.

7.3 Sub-Packages in Java

- You can construct hierarchical package names, for example `org.xml.sax` and `org.xml.sax.helpers`.
  - However, Java treats these as two **independent packages**, one with class files in `org/xml/sax` and the other with class files in `org/xml/helpers`.
  - The "information sharing" features of Java packages do not apply across these independent packages.
  - Still, hierarchical package names are useful for grouping together related sets of classes.

7.4 Package Naming Conventions

- To prevent package name collisions, the convention is for organizations to use their reversed Internet domain names to begin their package names.
  - For example, `com.marakana`
  - If the Internet domain name contains an invalid character, such as a hyphen, the convention is to replace the invalid character with an underscore. For example, `com.avia_training`
  - If a domain name component starts with a digit or consists of a reserved Java keyword, the convention is to add an underscore to the component name.
- Organizations often create sub-packages to reflect business units, product lines, etc.

  - For example, `com.marakana.android`

- Package names starting with `java.` and `javax.` are reserved for Java libraries.

Java does not do any explicit protection of package namespaces. This means that it is possible for an organization ABC to create a class in package `com.xyz` even though the domain `xyz.com` belongs to an organization called XYZ. However in practice organizations do not abuse each other's namespaces.

7.5 Using Package Members: Qualified Names

- The *fully-qualified name* for a class or interface is the package name followed by the class/interface name, separated by a period (`.`).
  - For example, `com.marakana.utils.MyClass`
- Code outside the package can reference public classes and interfaces of a package using their fully-qualified names.
- Code within the package can reference classes and interfaces of the package by their simple names, without package qualification.
- To execute a Java application whose class is contained within a package, you must use its fully-qualified name.
  - For example, to run `ShapeDemo.main()` in the package `shape` you must invoke it as:

    ```
    java shape.ShapeDemo
    ```

7.6 Importing Package Members

- Instead of using fully-qualified names for classes and interfaces, you can *import* the package member.
  - To import a specific member from a package, include an `import` statement specifying the fully-qualified name of that member. For example:

    ```
    import com.marakana.utils.MyClass;
    ```

    You can then use the simple (unqualified) name of the member throughout the rest of the source file.
  - You can import **all** of the members of a package by importing the package with an asterisk (`*`) wildcard. For example:

    ```
    import com.marakana.utils.*;
    ```

    You can then use the simple (unqualified) name of all the package members throughout the rest of the source file.

    Important
    Using the wildcard `import` does **not** import any sub-packages or their members.

- The `import` statements must appear after any `package` statement and before all class/interface definitions in your file.

Note
The `java.lang` package is imported automatically into all source files.

In case there are two or more classes with the same name but defined in different imported (or assumed) packages, then those classes must be referenced by their fully-qualified-class-name (FQCN).

For example, you could define:

`com.mycompany.calc.AdditionOperation`,

but someone else could implement:

`com.othercompany.calculator.AdditionOperation`.

If you wanted to use both addition operations from a class defined in `com.mycompany.calc` package, then doing

```
import com.othercompany.calculator.AdditionOperation;
```

would make AdditionOperation ambiguous. Instead of importing it, you would reference it by its FQCN.

Caution
Even though importing an entire package using the asterisk (`*`) notation is convenient, it is not generally recommended — especially when multiple packages are imported. It makes it harder to track down the classes if you do not know exactly which package they come from. Modern IDEs help with resolving classes (as well as with organizing imports), but you should not assume that everyone will always have an IDE handy.

7.7 Static Imports

- Prior to Java 5, you always had to access static fields and methods qualified by their class name.
  - For example:

```
double c= Math.PI * Math.pow(r, 2.0);
```

- Java 5 introduced `import static` to import static fields and methods.
  - For example:

```
import static java.lang.Math.PI;    // A single field
import static java.lang.Math.*;     // All static fields and methods
```

  - Once imported, you can use them as if they were defined locally:

```
double c= PI * pow(r, 2.0);
```

7.8 Access Modifiers and Packages

- The the last module introduced the concept of the `public` and `private` *access modifiers* that control access to fields, methods, constructors, classes, and interfaces.
  - There are two additional access levels:

Access Modifier	Description
`public`	Accessible from any class
`protected`	Accessible from all classes in the **same package** or any **child classes** regardless of the package
default (no modifier)	Accessible only from the classes in the **same package** (also known as *friendly*)
`private`	Accessible only from within the same class (or any nested classes)

7.9 The Class Path

- The *class path* is the path that the Java runtime environment searches for classes and other resource files.
- By default, Java looks for classes in only the present working directory (`.`).
- To access classes at runtime outside the present working directory, there are three options:
  - Set an environmental variable named `CLASSPATH` with a colon-separated list of class directories, JAR files, and/or Zip files. (This should be a semicolon-separated list on Windows.)
  - Use use the `-classpath` option (`-cp` is an alias) with `java` and `javac` to override the class path with a colon/semicolon-separated list of class directories, JAR files, and/or Zip files.
  - Copy classes to `$JAVA_HOME/lib/ext/` (this is typically used only for Java extensions).

Important
The class path should include only the root directory of a package directory hierarchy, not the individual package directories.

For example, to run `calc.CalculatorDemo` located in `/home/me/classes/calc/CalculatorDemo.class`, you could do:

```
cd /home/me/classes
java calc.CalculatorDemo
```

or (from any directory):

```
export CLASSPATH=/home/me/classes
java calc.CalculatorDemo
```

or (from any directory):

```
java -classpath /home/me/classes calc.CalculatorDemo
```

or (from any directory):

```
cp -r /home/me/classes/calc $JAVA_HOME/lib/ext/.
java calc.CalculatorDemo
```

It is important to notice the distinction between class path locations and package directories. Package-based directories should never be included within the class path and vice-versa.

This means that these would not work:

```
java -classpath /home/me/classes/calc CalculatorDemo
java -classpath /home/me classes.calc.CalculatorDemo
```

7.10 Java Archive (JAR) Files

- A Java Archive (JAR) file packages multiple classes in a single compressed file.
  - Based on the ZIP file format
  - Easier and more efficient management of binaries
  - Also known as Java library (.jar extension)
  - Must preserve package directory structure
- A JAR file can include additional meta-data in a manifest file at the pathname `META-INF/MANIFEST.MF`.
  - The manifest is a text file containing `name: value` formatted entries.
  - For example, you can make an executable JAR by including a `Main-Class` entry whose value is the fully-qualified name of a class containing a `main()` method. You could then execute the JAR as:

    ```
    java -jar <file>.jar
    ```

JAR files internally must look like package directories. For example, a class `com.myco.calc.CalculatorDemo` would be stored within `calculator.jar` file just like it would be stored on disk: `com/myco/calc/CalculatorDemo.class`.

JAR files are referenced in the `CLASSPATH` by their actual name, not by the name of the directory in which they are contained.

To run `com.myco.calc.CalculatorDemo` you would do:

```
java -classpath calculator.jar com.myco.calc.CalculatorDemo
```

To make `calculator.jar` executable:

1. Create a file: `META-INF/MANIFEST.MF`
2. Add a line:

   ```
   Main-Class: com.myco.calc.CalculatorDemo
   ```

3. Include `META-INF/MANIFEST.MF` in `calculator.jar`
4. Run with

   ```
   java -jar calculator.jar
   ```

7.11 The `jar` Command-Line Tool Examples

- Create a JAR file

```
cd classes; jar -cvf shape.jar shape
jar -cvf shape.jar -C classes shape
```

- View the contents of a JAR file

```
jar -tvf shape.jar
```

- Extract the contents of JAR file

```
jar -xvf shape.jar
jar -xvf shape.jar META-INF/MANIFEST.MF
```

- Update the JAR file (e.g., add to the manifest)

```
jar -uvmf MainClass.txt shape.jar
```

You will find that the `jar` command has a usage similar to the Unix `tar` command:

```
Usage: jar {ctxui}[vfm0Me] [jar-file] [manifest-file] [entry-point] [-C dir] files ...
Options:
    -c  create new archive
    -t  list table of contents for archive
    -x  extract named (or all) files from archive
    -u  update existing archive
    -v  generate verbose output on standard output
    -f  specify archive file name
    -m  include manifest information from specified manifest file
    -e  specify application entry point for stand-alone application
        bundled into an executable jar file
    -0  store only; use no ZIP compression
    -M  do not create a manifest file for the entries
    -i  generate index information for the specified jar files
    -C  change to the specified directory and include the following file
If any file is a directory then it is processed recursively.
The manifest file name, the archive file name and the entry point name are
specified in the same order as the 'm', 'f' and 'e' flags.

Example 1: to archive two class files into an archive called classes.jar:
       jar cvf classes.jar Foo.class Bar.class
Example 2: use an existing manifest file 'mymanifest' and archive all the
           files in the foo/ directory into 'classes.jar':
       jar cvfm classes.jar mymanifest -C foo/ .
```

Note
You can use any utility with support for ZIP files (e.g. WinZip) to create, view, extract, and manage your JAR files.

Chapter 8

JavaDoc

Documenting Java Code

8.1 What is JavaDoc?

- *Doc comments* (also known informally as *Javadoc comments*, although this technically violates trademark usage) document your APIs for other programmers who use your classes and for maintenance programmers.
- Doc comments standardize the way source code is documented.
- Documentation is kept in the source file, so it's easier for developers to keep it in sync with the code.
- You can document packages, classes, constructors, fields, and methods.

Doc comments are a very important feature of Java, because it defines a common way for developers to communicate about their code.

It makes using third-party classes (including the Java Library itself) much easier because everything is documented the same way.

8.2 Reading Doc Comments (Java API)

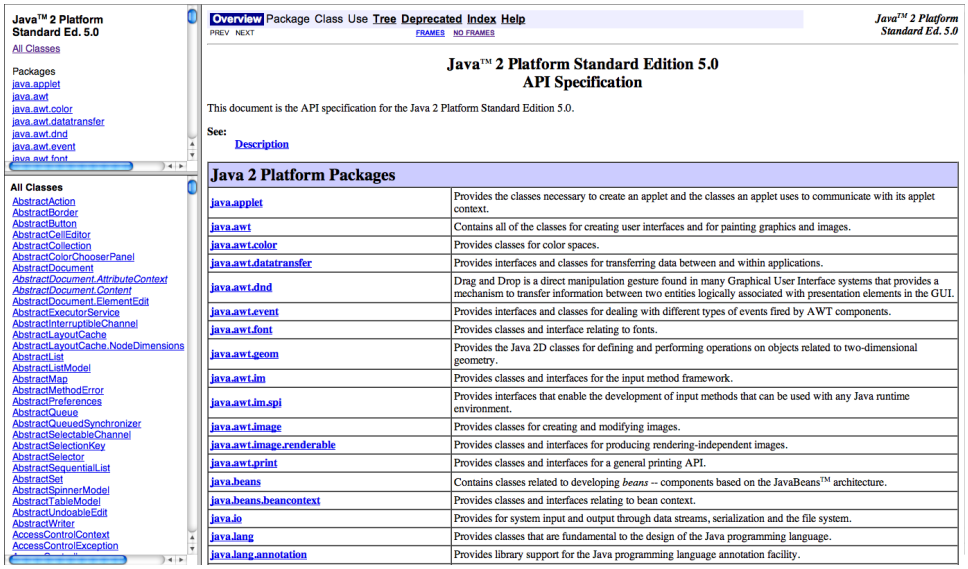


Figure 8.1: JavaDoc Page For Java 1.5

You can browse the Java API documentation online at: http://download.oracle.com/javase/6/docs/api/
Alternatively, you can download the documentation from http://www.oracle.com/technetwork/java/javase/downloads/index.html and browse it locally.

- The top-left frame defines all available packages.
- Clicking on a package name in the top-left frame causes the bottom-left frame to display all interfaces, classes, enumerations, exceptions, errors, and annotations in that package.
- Clicking on the package name in the bottom-left frame (at top) causes the right-hand frame to display the information for that package.
- Clicking on an interface, class, enumeration, exception, error, or annotation in the bottom-left frame, causes the right-hand frame to display the information about that entity. Links within the right-hand frame usually jump to anchors within the same page.

8.3 Defining JavaDoc

- Use `/** */` comments right **before** the entities that are to be documented.
  - You can have whitespace between the doc comment and the declaration, but no other code. For example, don't put `import` statements between your doc comment and a class declaration.
  - If a doc comment line begins with a `*` preceded by optional whitespace, those characters are ignored.
  - As of Java 1.4, leading whitespace is preserved if a line does not begin with a `*` character. This allows you to include formatted code fragments (wrapped with HTML `<pre>` tags) in your documentation.
- A doc comment consists of an optional main description followed by an optional tag section.
- A doc comment can contain HTML markup, but keep it simple (as in, keep it `<em>simple</em>`).
- The first sentence of the main description (ending in a period followed by a space, tab, line terminator, or first block tag) is used as the summary description for the declared entity.

8.4 Doc Comment Tags

- *Block tags* have the format `@tag description`. There are many block tags available, but the more commonly used ones are:

  `@author` *`name`*
  : Author Name (class/interface only)

  `@version` *`major.minor.patch`*
  : Version number (class/interface only)

  `@param` *`name description`*
  : Description of parameter (method only)

  `@return` *`description`*
  : Description (method only)

  `@throws` *`Throwable description`*
  : Description of exception (exceptions are discussed in the next module)

  `@deprecated` *`explanation`*
  : Explanation (method only)

  `@see` *`package.class#member label`*
  : A hyperlink to a reference package/class/member or field. Or `simple text` for a "See Also" reference

- Java also supports various *inline tags* of the form `{@tag text_content}` to handle special text inline. The most important ones are:

  `{@literal` *`text`*`}`
  : Treat the text as literal, suppressing the processing of HTML markup and nested doc comment tags

  `{@code` *`text`*`}`
  : Treat the text as literal, suppressing the processing of HTML markup and nested doc comment tags, and render the text in a code font

---

Use doc comments! At least document everything that is marked `public` or `protected`.

This is an example of how you could document an interface:

```
/**
 * Simple calculator operation.
 * @author <a href="mailto:me@my.com">Me</a>
 * @version 1.0
 */
public interface Operation {
    /**
     * Perform a single calculation.
     * @param operand the operand to use for calculation.
     */
    public void calculate(double operand);

    /**
     * Get the current result.
     * @return the current result. If no calculations were
     *         performed the result is undefined.
     */
    public double getResult();
}
```

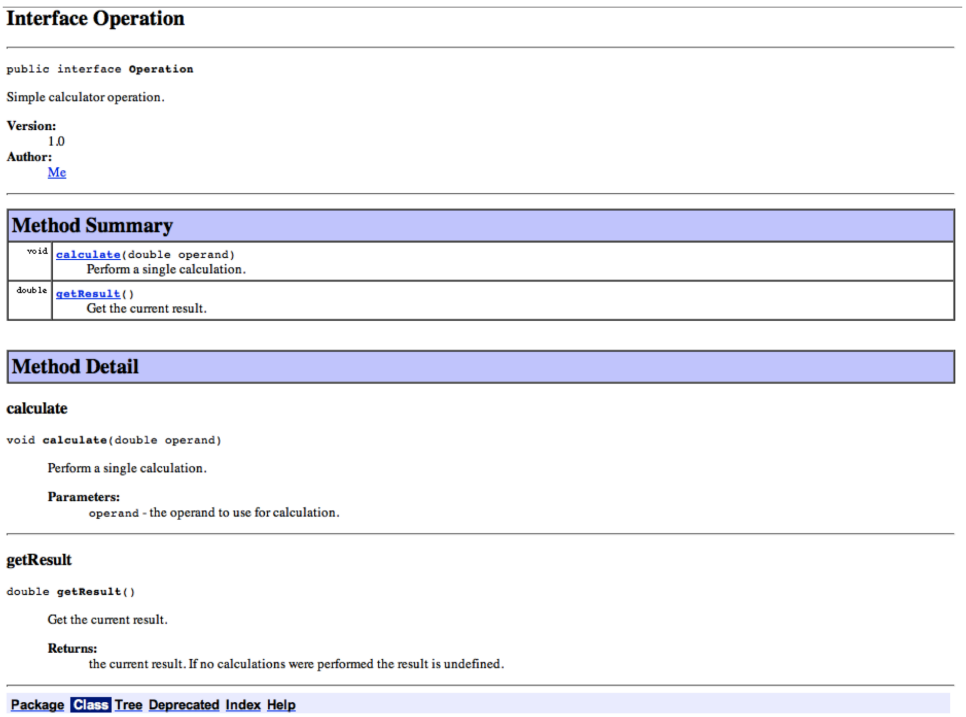
For the full doc comments HOWTO (including a complete list of doc comment tags), please see: http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

8.5 Generating Doc Comment Output

- To generate HTML formatted documentation, run the `javadoc` command-line tool on your source files to generate the HTML API files (including indexes):

```
javadoc -d /path/to/output /path/to/*.java
```

- You can also specify one or more package names as arguments, in which case `javadoc` generates output for all classes and interfaces in those packages.
  - Open the generated */path/to/output/*`index.html` file in your browser.



Interface Operation

`public interface Operation`

Simple calculator operation.

Version:
1.0
Author:
Me

Method Summary

void	`calculate(double operand)` Perform a single calculation.
double	`getResult()` Get the current result.

Method Detail

calculate

`void calculate(double operand)`

Perform a single calculation.

Parameters:
operand - the operand to use for calculation.

getResult

`double getResult()`

Get the current result.

Returns:
the current result. If no calculations were performed the result is undefined.

Package Class Tree Deprecated Index Help

Figure 8.2: Doc Comment Generation Example

Chapter 9

Exceptions

Exception Handling in Java

9.1 What are Exceptions?

- *Exceptions* are events that occur during the execution of programs that disrupt the normal flow of instructions (e.g. divide by zero, array access out of bound, etc.).
- In Java, an exception is an **object** that wraps an error event that occurred within a method and contains:
  - Information about the error including its type
  - The state of the program when the error occurred
  - Optionally, other custom information
- Exception objects can be *thrown* and *caught*.

Exceptions are used to indicate many different types of error conditions.

- JVM Errors:
  - OutOfMemoryError
  - StackOverflowError
  - LinkageError

System errors:

- FileNotFoundException
- IOException
- SocketTimeoutException
  - Programming errors:
- NullPointerException
- ArrayIndexOutOfBoundsException
- ArithmeticException

9.2 Why Use Exceptions?

- Exceptions separate error handling code from regular code.
  - Benefit: Cleaner algorithms, less clutter
- Exceptions propagate errors up the call stack.
  - Benefit: Nested methods do not have to explicitly catch-and-forward errors (less work, more reliable)
- Exception classes group and differentiate error types.
  - You can group errors by their generalize parent class, **or**
  - Differentiate errors by their actual class
- Exceptions standardize error handling.

In a traditional programming language like C, an error condition is usually indicated using an integer return value (e.g. -1 for out of memory). However, this practice:

- Requires standardization on error codes (hard for large projects)
- Makes functions hard to use because they must return actual values by reference
- Requires programmers to check return error codes in every nested function call. This leads to cluttered source code and hard-to-follow logic
- Does not allow for the state of the program to be easily captured on an error condition for later examination
- Cannot be enforced, and programmers often ignore error conditions, which leads to security or stability issues in programs

9.3 Built-In Exceptions

- Many exceptions and errors are automatically generated by the Java virtual machine.
- Errors, generally abnormal situations in the JVM, such as:
  - Running out of memory
  - Infinite recursion
  - Inability to link to another class
- Runtime exceptions, generally a result of programming errors, such as:
  - Dereferencing a `null` reference
  - Trying to read outside the bounds of an array
  - Dividing an integer value by zero

```
public class ExceptionDemo {

    public static void main (String[] args) {
        System.out.println(divideArray(args));
    }

    private static int divideArray(String[] array) {
        String s1 = array[0];
        String s2 = array[1];
        return divideStrings(s1, s2);
    }

    private static int divideStrings(String s1, String s2) {
        int i1 = Integer.parseInt(s1);
        int i2 = Integer.parseInt(s2);
        return divideInts(i1, i2);
    }

    private static int divideInts(int i1, int i2) {
        return i1 / i2;
    }
}
```

Compile and try executing the program as follows:

```
java ExceptionDemo 100 4
java ExceptionDemo 100 0
java ExceptionDemo 100 four
java ExceptionDemo 100
```

Note that we could have overloaded all `divide*()` methods (since they have different parameters), but this way the generated *stack traces* are easier to follow.

9.4 Exception Types

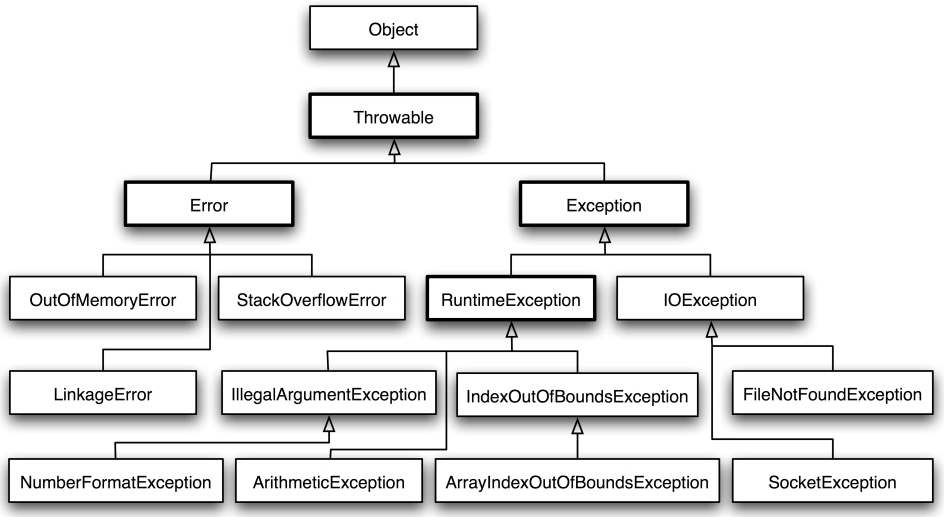


Figure 9.1: A Sample of Exception Types

All exceptions and errors extend from a common `java.lang.Throwable` parent class. Only Throwable objects can be thrown and caught.

Other classes that have special meaning in Java are `java.lang.Error` (JVM Error), `java.lang.Exception` (System Error), and `java.lang.RuntimeException` (Programming Error).

```
public class java.lang.Throwable extends Object
            implements java.io.Serializable {
    public Throwable();
    public Throwable(String msg);
    public Throwable(String msg, Throwable cause);
    public Throwable(Throwable cause);
    public String getMessage();
    public String getLocalizedMessage();
    public Throwable getCause();
    public Throwable initCause(Throwable cause);
    public String toString();
    public void printStackTrace();
    public void printStackTrace(java.io.PrintStream);
    public void printStackTrace(java.io.PrintWriter);
    public Throwable fillInStackTrace();
    public StackTraceElement[] getStackTrace();
    public void setStackTrace(StackTraceElement[] stackTrace);
}
```

9.5 Checked vs. Unchecked Exceptions

- Errors and RuntimeExceptions are *unchecked* — that is, the compiler does not enforce (check) that you handle them explicitly.
- Methods do not have to declare that they `throw` them (in the method signatures).
- It is assumed that the application cannot do anything to recover from these exceptions (at runtime).
- All other Exceptions are *checked* — that is, the compiler enforces that you handle them explicitly.
- Methods that generate checked exceptions must declare that they `throw` them.
- Methods that invoke other methods that throw checked exceptions must either handle them (they can be reasonably expected to recover) or let them propagate by declaring that they `throw` them.

There is a lot of controversy around checked vs. unchecked exceptions.

Checked exceptions give API designers the power to force programmers to deal with the exceptions. API designers expect programmers to be able to reasonably recover from those exceptions, even if that just means logging the exceptions and/or returning error messages to the users.

Unchecked exceptions give programmers the power to ignore exceptions that they cannot recover from, and only handle the ones they can. This leads to less clutter. However, many programmers simply ignore unchecked exceptions, because they are by default un-recoverable. Turning all exceptions into the unchecked type would likely lead to poor overall error handling.

It is interesting to note that Microsoft's C# programming language (largely based on Java) does not have a concept of checked exceptions. All exception handling is purely optional.

9.6 Exception Lifecycle

- After an exception object is created, it is handed off to the runtime system (*thrown*).
- The runtime system attempts to find a handler for the exception by backtracking the ordered list of methods that had been called.
  - This is known as the *call stack*.
- If a handler is found, the exception is *caught*.
  - It is handled, or possibly re-thrown.
- If the handler is not found (the runtime backtracks all the way to the `main()` method), the exception stack trace is printed to the standard error channel (`stderr`) and the application aborts execution.

For example:

```
java ExceptionDemo 100 0

Exception in thread "main" java.lang.ArithmeticException: / by zero
at ExceptionDemo.divideInts(ExceptionDemo.java:21)
at ExceptionDemo.divideStrings(ExceptionDemo.java:17)
at ExceptionDemo.divideArray(ExceptionDemo.java:10)
at ExceptionDemo.main(ExceptionDemo.java:4)
```

Looking at the list bottom-up, we see the methods that are being called from `main()` all the way up to the one that resulted in the exception condition. Next to each method is the line number where that method calls into the next method, or in the case of the last one, throws the exception.

9.7 Handling Exceptions

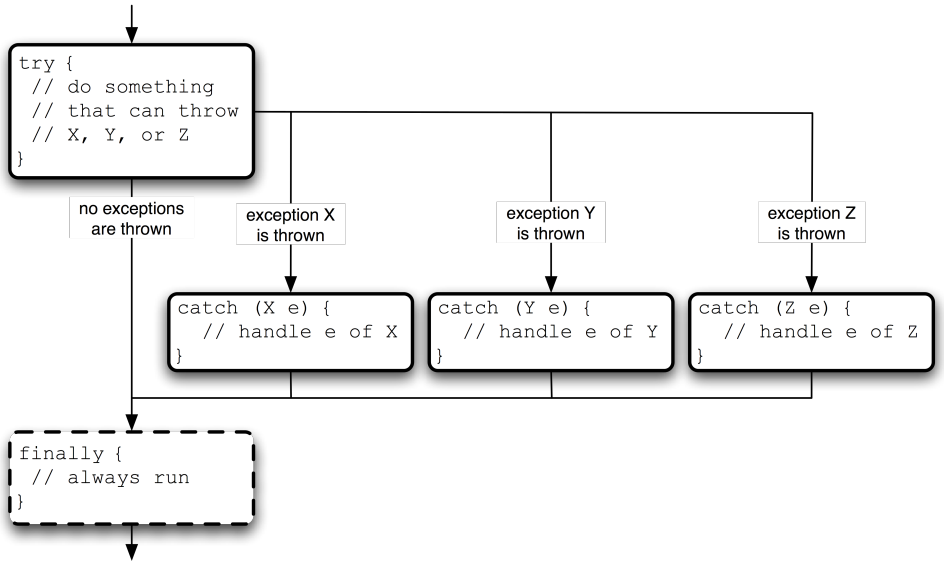


Figure 9.2: The `try-catch-finally` Control Structure

```
public class ExceptionDemo {

    public static void main (String[] args) {
       divideSafely(args);
    }

    private static void divideSafely(String[] array) {
        try {
            System.out.println(divideArray(array));
        } catch (ArrayIndexOutOfBoundsException e) {
            System.err.println("Usage: ExceptionDemo <num1> <num2>");
        } catch (NumberFormatException e) {
            System.err.println("Args must be integers");
        } catch (ArithmeticException e) {
            System.err.println("Cannot divide by zero");
        }
    }

    private static int divideArray(String[] array) {
        String s1 = array[0];
        String s2 = array[1];
        return divideStrings(s1, s2);
    }

    private static int divideStrings(String s1, String s2) {
        int i1 = Integer.parseInt(s1);
        int i2 = Integer.parseInt(s2);
        return divideInts(i1, i2);
    }

    private static int divideInts(int i1, int i2) {
        return i1 / i2;
    }
}
```

9.8 Handling Exceptions (cont.)

- The `try-catch-finally` structure:

```
try {
    // Code bock
}
catch (ExceptionType1 e1) {
    // Handle ExceptionType1 exceptions
}
catch (ExceptionType2 e2) {
    // Handle ExceptionType2 exceptions
}
// ...
finally {
    // Code always executed after the
    // try and any catch block
}
```

  - Both `catch` and `finally` blocks are optional, but at least one must follow a `try`.
  - The `try-catch-finally` structure can be nested in `try`, `catch`, or `finally` blocks.

  - The `finally` block is used to clean up resources, particularly in the context of I/O.
  - If you omit the `catch` block, the `finally` block is executed before the exception is propagated.
- Exceptions can be caught at any level.
- If they are not caught, they are said to *propagate* to the next method.

```
public class ExceptionDemo {

    public static void main (String[] args) {
       divideSafely(args);
    }

    private static void divideSafely(String[] array) {
        try {
            System.out.println(divideArray(array));
        } catch (ArrayIndexOutOfBoundsException e) {
            System.err.println("Usage: ExceptionDemo <num1> <num2>");
        }
    }

    private static int divideArray(String[] array) {
        String s1 = array[0];
        String s2 = array[1];
        return divideStrings(s1, s2);
    }

    private static int divideStrings(String s1, String s2) {
        try {
            int i1 = Integer.parseInt(s1);
            int i2 = Integer.parseInt(s2);
            return divideInts(i1, i2);
        } catch (NumberFormatException e) {
            return 0;
        }
    }

    private static int divideInts(int i1, int i2) {
        try {
            return i1 / i2;
        } catch (ArithmeticException e) {
            return 0;
        }
    }
}
```

9.9 Grouping Exceptions

- Exceptions can be caught based on their generic type. For example:

```
catch (RuntimeException e) {
    // Handle all runtime errors
}
```

Caution
Be careful about being too generic in the types of exceptions you catch. You might end up catching an exception you didn't know would be thrown and end up "hiding" programming errors you weren't aware of.

- `catch` statements are like repeated `else-if`s.
  - At most one `catch` bock is executed.
  - Be sure to catch generic exceptions after the specific ones.

```
public class ExceptionDemo {

    public static void main (String[] args) {
       divideSafely(args);
    }

    private static void divideSafely(String[] array) {
        try {
            System.out.println(divideArray(array));
        } catch (ArrayIndexOutOfBoundsException e) {
            System.err.println("Usage: ExceptionDemo <num1> <num2>");
        } catch (RuntimeException e) {
            System.err.println("Error result: 0");
        }
    }

    private static int divideArray(String[] array) {
        String s1 = array[0];
        String s2 = array[1];
        return divideStrings(s1, s2);
    }

    private static int divideStrings(String s1, String s2) {
        int i1 = Integer.parseInt(s1);
        int i2 = Integer.parseInt(s2);
        return divideInts(i1, i2);
    }

    private static int divideInts(int i1, int i2) {
        return i1 / i2;
    }
}
```

9.10 Throwing Exception

- A method that generates an unhandled exception is said to *throw* an exception. That can happen because:
  - It generates an exception to signal an exceptional condition, or
  - A method it calls throws an exception
- Declare that your method throws the exception using the `throws` clause.
  - This is required only for checked exceptions.

- Recommended: Document (in JavaDoc) that the method `@throws` the exception and under what circumstances.
- To generate an exceptional condition:
  - Create an exception of the appropriate type. Use predefined exception types if possible, or create your own type if appropriate.
  - Set the error message on the exception, if applicable.
  - Execute a `throw` statement, providing the created exception as an argument.

Some of the more commonly used exception types are:

NullPointerException
: Parameter value is `null` where prohibited

IllegalArgumentException
: Non-`null` parameter value is inappropriate

IllegalStateException
: Object state is inappropriate for method invocation

IndexOutOfBoundsException
: Index parameter is out of range

UnsupportedOperationException
: Object does not support the method

For example:

```
/**
* Circle shape
*/
public class Circle implements Shape {
    private final double radius;
    /**
    * Constructor.
    * @param radius the radius of the circle.
    * @throws IllegalArgumentException if radius is negative.
    */
    public Circle(double radius) {
        if (radius < 0.0) {
            throw new IllegalArgumentException("Radius " + radius
                + " cannot be negative");
        }
        this.radius = radius;
    }
    ...
}
```

9.11 Creating an Exception Class

- If you wish to define your own exception:
  - Pick a self-describing `*Exception` class name.
  - Decide if the exception should be checked or unchecked.

Checked
extends Exception

Unchecked
extends RuntimeException

- Define constructor(s) that call into super's constructor(s), taking message and/or cause parameters.
- Consider adding custom fields, accessors, and mutators to allow programmatic introspection of a thrown exception, rather than requiring code to parse an error message.

```
/**
 * Exception thrown to indicate that there is insufficient
 * balance in a bank account.
 * @author me
 * @version 1.0
 */
public class InsufficientBalanceException extends Exception {
    private final double available;
    private final double required;

    /**
     * Constructor.
     * @param available available account balance
     * @param required required account balance
     */
    public InsufficientBalanceException(double available, double required) {
        super("Available $"+available+" but required $"+required);
        this.available = available;
        this.required = required;
    }

    /**
     * Get available account balance
     * @return available account balance
     */
    public double getAvailable() {
        return available;
    }

    /**
     * Get required account balance
     * @return required account balance
     */
    public double getRequired() {
        return required;
    }

    /**
     * Get the difference between required and available account balances
     * @return required - available
     */
    public double getDifference() {
        return required - available;
    }
}
```

9.12 Nesting Exceptions

- A Throwable object can nest another Throwable object as its *cause*.
- To make APIs independent of the actual implementation, many methods throw generic exceptions. For example:
  - Implementation exception: SQLException
  - API exception: DataAccessException
  - API exception nests implementation exception
  - To get the original implementation exception, do `apiException.getCause()`

```
public Customer getCustomer(String id) throws DataAccessException {
    try {
        Connection con = this.connectionFactory.getConnection();
        try {
            String sql
                = "SELECT * FROM Customers WHERE CustomerID='"
                + id + "'";
            PreparedStatement stmt = con.prepareStatement(sql);
            try {
                ResultSet result = stmt.executeQuery(sql);
                try {
                    return result.next() ? this.readCustomer(result) : null;
                } finally {
                    result.close();
                }
            } finally {
                stmt.close();
            }
        } finally {
            con.close();
        }
    } catch (SQLException e) {
        throw new DataAccessException("Failed to get customer ["
                        + id + "]: " + e.getMessage(), e);
    }
}
```

Observe that the `getCustomer()` method signature does not say anything about customers being retrieved from a SQL-based RDBMS. This allows us to provide an alternative implementation, say based on a LDAP directory server or XML files.

Chapter 10

The `java.lang` Package

Exploring the Main Java Library

10.1 Primitive Wrappers

- The `java.lang` package includes a class for each Java primitive type:
  - Boolean, Byte, Short, *Character*, *Integer*, Float, Long, Double, *Void*
- Used for:
  - Storing primitives in Object based collections
  - Parsing/decoding primitives from Strings, for example:

    ```
    int value = Integer.parseInt(str);
    ```

  - Converting/encoding primitives to Strings
  - Range definitions (e.g. `Byte.MAX_VALUE`)
  - Toolbox of convenience methods — many of them `static` (e.g., `isLetter()`, `toLowerCase()`, `reverseBytes()`, `isInfinite()`, `isNaN()`)

Any object in Java can be assigned to type `java.lang.Object`. This means that generic collections can be implemented that deal with `java.lang.Objects`, and not care about the actual types. Unfortunately, this does not work for primitives, because they are not Objects. But with their wrapper classes, primitives can easily move to/from Objects.

For example:

```
int i = 5;
List list = new LinkedList();
list.add(new Integer(i));
Integer intObj = (Integer) list.get(0);
int j = intObj.intValue();
System.out.println(i == j); // prints true
```

Note
Since Java 5, Java supports implicit wrapping/unwrapping of primitives as needed. This compiler-feature is called auto-boxing/auto-unboxing. The following is legal:

```
Integer j = 5 + 10;
int k = new Integer(5) + new Integer(10);
Float f = new Float(10) * new Float (3);
listOfLongs.add(123123123L);
long l = (Long) listOfLongs.get(0);
```

Warning
This auto-boxing feature can lead to legal but very inefficient code. For example, in the following loop, the index variable is accidentally declared as a `Long` rather than a `long`, leading to the creation of a billion extraneous objects compared to using the primitive type:

```
for (Long i = 0; i <= 1000000000; i++) {
    // This will take a very long time to execute
}
```

10.2 String and StringBuilder

- String
  - Automatically created with double quotes
  - Pooled (interned) to save memory
  - Immutable — once created cannot change
  - Concatenated using the expensive + operator
  - Many methods for string manipulation/searching
- StringBuilder and StringBuffer
  - Mutable — can quickly grow and shrink as needed
  - Few methods modifying the buffer: `append()`, `insert()`, `delete()`, `replace()`
  - Use `toString()` if you need to get a String representation of the object's content
  - StringBuffer is synchronized for thread-safe access in multithreaded applications
  - Use StringBuilder for better performance unless you plan to access the object from multiple threads

---

In a single-threaded context (as is generally the case), StringBuilder is faster than StringBuffer. But both are 1000+ times faster than repeated String concatenation.

```
public class StringConcatenationTest {
    public static void main (String args[]) {
        final int count = Integer.parseInt(args[0]);
        String s = "";
        long time = System.currentTimeMillis();
        for (int i = 0; i < count; i++) {
            s += ".";
        }
        time = System.currentTimeMillis() - time;
        System.out.println(time);
    }
}
```

```
public class StringBuilderConcatenationTest {
    public static void main (String args[]) {
        final int count = Integer.parseInt(args[0]);
        StringBuilder sb = new StringBuilder();
        long time = System.currentTimeMillis();
        for (int i = 0; i < count; i++) {
            sb.append(".");
        }
        sb.trimToSize();
        time = System.currentTimeMillis() - time;
        System.out.println(time);
    }
}
```

10.3 The Math Class

- The Math class contains static methods implementing math functions, such as:
  - Trigonometry: `sin()`, `cos()`, `tan()`, etc.
  - Rounding: `round()`, `floor()`, `ceil()`, `rint()`
  - Logarithmic: `log()`, `log10()`
  - Exponentiation/root: `sqrt()`, `cbrt()`, `pow()`, `exp()`
  - Utility: `abs()`, `min()`, `max()`, `random()`, `hypot()`, `toDegrees()`, `toRadians()`
- Match also includes constants for PI and E.

In addition to `java.lang.Math`, Java ships with `java.lang.StrictMath`, which has the same functions as Math. What's different with StrictMath is that its functions produce the same results as certain published algorithms, namely the C-based `netlib` and "Freely Distributable Math Library".

Java also comes with `java.math` package which includes:

- BigDecimal - Immutable arbitrary-precision signed decimal number.
- BigInteger - Immutable arbitrary-precision integer.

These two classes have their own methods for various operations, including arithmetic and bitwise.

10.4 The System Class

- The System class provides access to your system and the JVM, including:
  - A context for `stdin`, `stdout`, `stderr` streams
  - Environmental variables and JVM properties
  - System time in milliseconds or nanoseconds
  - Loading external [native] libraries
  - Terminating the JVM
  - Efficient array copying

The `java.lang.System` class gives access to the JVM system, although it is used most often for printing to `stdout`:

```
System.out.println("Hello World");
```

You should avoid invoking:

- `System.exit(int status)` — This method forces termination of all threads in the JVM. According the Sun's Portability Cookbook:

  > `System.exit()` should be reserved for a catastrophic error exit, or for cases when a program is intended for use as a utility in a command script that may depend on the program's exit code.

- `System.gc()` — There is no guarantee that the JVM will run the garbage collection in response to this method. Let the JVM manage garbage collection.
- `System.runFinalization()` - Do not depend on object finalization. It is not guaranteed to run before your application terminates.

System properties give you information about the JVM and the context in which it is running (e.g., the user, present working directory, etc.). Properties should not be used for global variables, but if you do set your own properties, do so in your own namespace. Properties are standardized across platforms.

Environmental variables are system dependent (configured by the shell), so they are generally not portable across platforms.

Chapter 11

Input/Output

Exploring the Java I/O Library

11.1 Representing File Paths

- The `java.io.File` class is for manipulating file *paths*.
  - The path may or may not refer to an actual on-disk file or directory.
  - Methods on the File class allow you to manipulate the path and perform file system operations.
  - The File class is **not** used to read or write file contents.
- The File constructor is overloaded, allowing you to create a File object from:
  - A single String representing a path
  - A String or File representing a parent directory path and a second String argument representing a child directory or file
- The path used to create a File object can be absolute or relative to the present working directory.
- Like String objects, File objects are *immutable*.
  - Once you create one, you cannot modify the path it represents.

How does Java handle platform-specific directory delimiter characters? According to the File reference page:

> When an abstract pathname is converted into a pathname string, each name is separated from the next by a single copy of the default *separator character*. The default name-separator character is defined by the system property `file.separator`, and is made available in the public static fields `separator` and `separatorChar` of this class. When a pathname string is converted into an abstract pathname, the names within it may be separated by the default name-separator character or by any other name-separator character that is supported by the underlying system.

11.2 Managing File Paths

- A File object provides methods for querying and manipulating the file system.
  - Methods that query the file system include: `canRead()`, `canWrite()`, `exists()`, `isDirectory()`, `isFile()`, `isHidden()`, `getAbsolutePath()`, `lastModified()`, `length()`, `listFiles()`, `listRoots()`
  - Methods that modify the file system include: `createNewFile()`, `mkdir()`, `renameTo()`, `delete()`, `deleteOnExit()`, `setReadOnly()`, `setLastModified()`

```
package io;

import java.io.File;

public class Remove {
    public static void main(String[] args) {
        if (args.length == 0) {
            System.err.println("Usage: Remove <file|directory>...");
        }
        for (int i = 0; i < args.length; i++) {
            remove(new File(args[i]));
        }
    }
    private static void remove(File file) {
        if (!file.exists()) {
            System.err.println("No such file or directory: "
                    + file.getAbsolutePath());
        } else if (file.isDirectory() && file.list().length > 0) {
            System.err.println("Directory contains files: "
                    + file.getAbsolutePath());
        }
        else if (!file.delete()) {
            System.err.println("Could not remove: " + file.getAbsolutePath());
        }
    }
}
```

11.3 Input/Output Class Hierarchy

- The `java.io` package includes a large number of classes for input and output.
- A set of four abstract base classes provide the core functionality for most of these classes:
  - Byte-oriented (binary) data: InputStream and OutputStream
  - Character-oriented data: Reader and Writer
- The methods for reading data from an InputStream and a Reader are very similar, as are the methods for writing data to an OutputStream or Writer.
- Many of the other I/O classes follow the *Decorator Pattern*, attaching additional responsibilities to an object dynamically.
  - Most of these classes have constructors that take a Reader (or Writer, or InputStream, or OutputSteam) object as an argument, and return an object that's another Reader (or Writer, or InputStream, or OutputSteam).
  - Each class just provides some additional functionality (such as buffering, compression, etc.) on top of the basic I/O functionality as defined by the abstract base class.

The key to the Decorator Pattern (and the I/O class hierarchy) is *polymorphism*. A FileReader "is-a" Reader, a BufferedReader "is-a" Reader, a LineNumberReader "is-a" Reader, etc.

11.4 Byte Streams

- InputStream — abstract *byte* reader
  - `read()` a single byte, or into a byte array
  - `skip()` bytes
  - `mark()` and `reset()` position
  - `close()` the stream (do so in `finally`)
- OutputStream — abstract *byte* writer
  - `write()` a single byte, or from a byte array
  - `flush()` the stream (in case it is buffered)
  - `close()` the stream (do so in `finally`)

Note
The no-argument `InputStream.read()` method returns an `int` value. Only the low-order 8 bits contains the byte of data. The method returns -1 on reaching the end of the stream. Symmetrically, `OutputStream.write(int)` accepts an `int` as an argument, but only the low-order 8 bits are written.

To read/write a single byte at a time, you could use the following loop:

```
int b = 0;
while ((b == fis.read()) != -1) {
    System.out.write(b);
}
```

However, using a byte array as a buffer is more efficient:

```
private static void read(File file) {
    try {
        FileInputStream fis = new FileInputStream(file);
        try {
            byte[] buffer = new byte[1024];
            int len = 0;
            while ((len = fis.read(buffer)) > 0) {
                System.out.write(buffer, 0, len);
            }
        } finally {
            fis.close();
        }
    } catch (FileNotFoundException e) {
        System.err.println("No such file exists: " + file.getAbsolutePath());
    } catch (IOException e) {
        System.err.println("Cannot read file: "
                + file.getAbsolutePath() + ": " + e.getMessage());
    }
}
```

11.5 Character Streams

- Reader — abstract *character* reader
  - `read()` a single char, or into a char array
  - `skip()` chars
  - `mark()` and `reset()` position
  - `close()` the stream (do so in `finally`)
- Writer — abstract *character* writer
  - `write()` a single char, or from a char array
  - `flush()` the stream (in case it is buffered)
  - `close()` the stream (do so in `finally`)

Note
The no-argument `Reader.read()` method returns an `int` value. Only the low-order 16 bits contains the char of data. The method returns -1 on reaching the end of the stream. Symmetrically, `Writer.write(int)` accepts an `int` as an argument, but only the low-order 16 bits are written.

```
private static void readDefault(File file) {
    try {
        FileReader reader = new FileReader(file);
        try {
            int ch;
            while ((ch = reader.read()) != -1) {
                System.out.print((char) ch);
            }
            System.out.flush();
        } finally {
            reader.close();
        }
    } catch (FileNotFoundException e) {
        System.err.println("No such file exists: " + file.getAbsolutePath());
    } catch (IOException e) {
        System.err.println("Cannot read file: "
                + file.getAbsolutePath() + ": " + e.getMessage());
    }
}
```

11.6 Exception Handling in Java I/O

- Almost all I/O class constructors and methods can throw a checked exception.
- The base class for I/O exceptions is `java.io.IOException`.
  - Some methods throw a subclass of IOException.
  - For example, the constructors for FileInputStream, FileOutputStream, and FileReader can throw FileNotFoundException.
- Your `close()` call should always occur in a `finally` block, to ensure that your I/O channel is closed properly even if an exception occurs.

Note
The `close()` method in almost all I/O classes can also throw an IOException, which must be handled. This leads to the frequent use of nested `try-catch-finally` blocks when performing I/O.

11.7 Java File I/O Classes

- The `java.io` package includes a set of concrete classes for file-based I/O:
  - FileInputStream, FileOutputStream, FileReader, FileWriter
- Overloaded constructors for each class allow you to specify a file path as either a File object or a String.
  - If these constructors can't open the file, all except FileWriter throw a FileNotFoundException.
  - FileWriter throws an IOException.
- The FileOutputStream and FileWriter constructors create a file if one doesn't already exist, unless the operating system prohibits it (e.g., bad path, no permission, etc.).
- By default, writes to the file overwrite existing content starting from the beginning of the file.
  - You can provide an optional boolean argument with a value of `true` to the FileOutputStream and FileWriter constructors to cause writes to append to the end of any existing content.

The FileWriter and FileReader classes automatically use the system encoding for character translation. If you need to use a different character set encoding, you need to open the file with FileInputStream or FileOutputStream, and then use an InputStreamReader or OutputStreamWriter as an adapter, and specify the appropriate character set in the constructor.

```
private static void readUTF8(File file) {
    try {
        FileInputStream fis = new FileInputStream(file);
        InputStreamReader reader = new InputStreamReader(fis, "UTF-8");
        try {
            OutputStreamWriter writer = new OutputStreamWriter(System.out, "UTF-8");
            int ch;
            while ((ch = reader.read()) != -1) {
                writer.write(ch);
            }
            writer.flush();
        } finally {
            reader.close();
        }
    } catch (FileNotFoundException e) {
        System.err.println("No such file exists: " + file.getAbsolutePath());
    } catch (IOException e) {
        System.err.println("Cannot read file: "
                + file.getAbsolutePath() + ": " + e.getMessage());
    }
}
```

11.8 Easy Text Output: PrintWriter/PrintStream

- The PrintWriter and PrintStream classes are designed to simplify common text output tasks.
  - The `print()` method is overloaded to print a String representation of all Java primitive types, and to automatically print the `toString()` representation of all Objects.
  - The `println()` method works in the same way as `print()`, but add a platform-specific line terminator.
  - The `format()` method and its alias the `printf()` method print a formatted representation of one or more Objects as specified by a format String. The operation is based on the C language `printf()` function.
  - The class methods never throw an IOException. Instead, exceptional situations merely set an internal flag that can be tested via the `checkError()` method.

Note
The `System.out` and `System.err` objects are instances of PrintStream.

- The PrintWriter and PrintStream constructors are overloaded. In addition to versions that accept an existing InputStream or Writer to "wrap", there are versions that accept a File object or a String representing a file path to specify an output file.

11.9 Reading from the Terminal

- For historical reasons, the `System.in` object is an instance of an InputStream, not a Writer.
  - Using `read()` in this case reads a *byte* at a time, not necessarily a complete *character*.
- You can use the `java.io.InputStreamReader` as an adapter.
  - By default, it reads bytes and decodes them into characters using the system's default character set encoding.
  - Overloaded versions of the InputStreamReader constructor allow you to specify a difference encoding.
  - You could then use `read()` to read a character at a time.
- Often it's more convenient to perform line-oriented input.
- The `java.io.BufferedReader` class buffers characters to provide more efficient reading from a character stream.
  - It includes a `readLine()` method that returns a complete line of text, stripped of its end-of-line terminator, as a String object.
- Combining the InputStreamReader and the BufferedReader is often a convenient way to read from the terminal:

```
BufferedReader input = new BufferedReader(
    new InputStreamReader(System.in) );
String line = input.readLine();
```

The following shows a simple example of line-oriented input from the terminal:

```
package com.marakana.demo;

import java.io.*;

public class HelloYou {

    public static void main(String[] args) {
```

```
        System.out.print("What's your name? ");

        /*
         * Set up so that we can do line-oriented character reads
         * from the standard input stream.
         */

        BufferedReader input = new BufferedReader(
            new InputStreamReader(System.in) );

        try {
            String name = input.readLine();
            System.out.println("Hello, " + name + "!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

}
```

11.10 Filtered Streams

- Many other pre-defined I/O "filter" classes are available in Java, including:
  - Buffering (BufferedReader/Writer/InputStream/OutputStream)
  - Checksum handling (java.util.zip.CheckedInputStream/OutputStream)
  - Compression (java.util.zip.GZIPInputStream/OutputStream and java.util.zip.ZipInputStream/OutputStream)
- You can define your own custom I/O filters by subclassing FilterReader/Writer/InputStream/OutputStream.

```
package io;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.zip.GZIPOutputStream;

/**
 * This program compresses one or more input files using gzip compression
 * algorithm. Source files are deleted and compressed files get .gz extension.
 */
public class Gzip {

    public static void main(String[] args) {
        if (args.length == 0) {
            System.err.println("Usage: Gzip <file>...");
        }
        for (int i = 0; i < args.length; i++) {
            gzip(new File(args[i]));
        }
    }
```

```
    private static void gzip(File file) {
        if (!file.exists()) {
            System.err.println("No such file: " + file.getAbsolutePath());
        } else if (file.isDirectory()) {
            System.err.println("Cannot compress directory: "
                    + file.getAbsolutePath());
        } else {
            File outFile = new File(file.getAbsoluteFile() + ".gz");
            try {
                InputStream in = new FileInputStream(file);
                try {
                    OutputStream out = new FileOutputStream(outFile);
                    try {
                        out = new GZIPOutputStream(out);
                        byte[] buffer = new byte[1024];
                        int len = 0;
                        while ((len = in.read(buffer)) > 0) {
                            out.write(buffer, 0, len);
                        }
                    } finally {
                        out.close();
                    }
                } finally {
                    in.close();
                }
                file.delete(); // delete original
            } catch (IOException e) {
                if (outFile.exists()) {
                    outFile.delete();
                }
                System.err.println("Failed to compress "
                        + file.getAbsolutePath() + ": " + e.getMessage());
            }
        }
    }
}
```

11.11 Object Serialization

- Java supports the automatic conversion of objects to and from byte streams.
  - Including complete object graphs.
- For an object (class) to be serializable, the class must:
  - Implement the java.io.Serializable interface, a *marker interface* with no required methods
  - Contain instance fields that are serializable — primitives or other Serializable types — except for any fields marked as `transient`
  - Have a no-argument constructor
  - (Optional but recommended) Implement a static final long field named `serialVersionUID` as a "version number" to identify changes to the class implementation that are incompatible with serialized objects of previous versions of the class.
- You can then serialize and deserialize objects with the following filter classes:
  - ObjectOutputStream — Serialize an object to an underlying OutputStream with the `writeObject()` method.
  - ObjectInputStream — Deserialize an object from an underlying InputStream with the `readObject()` method.

A simple example, assuming that Employee implements java.io.Serializable:

```
import java.io.ObjectOutputStream;
import java.io.IOException;
import java.io.FileOutputStream;

public class StoreEmployee {
    public static void main (String[] args) throws IOException {
        if (args.length != 6) {
            System.err.println("Usage: StoreEmployee <name> <ssn> <email> <yob> <extra> < ↩
                file>");
            return;
        }
        Employee e = new Employee(args[0], args[1], args[2]);
        e.setYearOfBirth(Integer.parseInt(args[3]));
        e.setExtraVacationDays(Integer.parseInt(args[4]));

        ObjectOutputStream oos = new ObjectOutputStream(
            new FileOutputStream(args[5]));
        try {
            oos.writeObject(e);
        } finally {
            oos.close();
        }
    }
}

import java.io.FileInputStream;
import java.io.ObjectInputStream;

public class LoadEmployee {
    public static void main(String[] args) throws Exception {
        if (args.length != 1) {
            System.err.println("Usage: LoadEmployee <file>");
            return;
        }
        ObjectInputStream ois = new ObjectInputStream(
            new FileInputStream(args[0]));
        try {
            Employee e = (Employee) ois.readObject();
            e.print();
        } finally {
            ois.close();
        }
    }

}
```

Note
Object serialization support can become quite complex, especially if a class supports internal references to I/O streams, databases, etc. A full discussion is beyond the scope of this class.

Chapter 12

java.net

Exploring Java Networking Library

12.1 java.net.InetAddress

- Represents an IP address
  - Supports IPv4 and IPv6
  - Supports unicast and multicast
- Hostname resolution (DNS lookup)
  - Supports caching (positive and negative)
- `java.net.NetworkInterface`
  - Represents system network interface
  - Lookup local *listening* `InetAddress`es

```
import java.net.*;
import java.util.*;

public class NICs {
  public static void main(String[] args) throws Exception {
    for (Enumeration nis = NetworkInterface.getNetworkInterfaces();
      nis.hasMoreElements();) {
      NetworkInterface ni = (NetworkInterface) nis.nextElement();
      System.out.println(ni.getDisplayName());
      for (Enumeration ias = ni.getInetAddresses();
        ias.hasMoreElements();) {
        InetAddress ia = (InetAddress) ias.nextElement();
        System.out.println("\t" + ia.getHostAddress());
      }
    }
  }
}

public class NsLookup {
  public static void main(String[] args) {
```

```
    try {
      InetAddress ia = InetAddress.getByName(args[0]);
      System.out.print(ia.getHostName());
      System.out.println(": " + ia.getHostAddress());
     } catch (UnknownHostException e) {
      System.err.println("Unknown " + args[0]);
    }
  }
}
```

12.2 URL/Connections

- `java.net.URL`
  - **U**niform **R**esource **L**ocator
  - Elements: scheme/protocol, host, port, authority, userInfo, path, query, fragment/ref
  - Factory for `URLConnections`
- `java.net.URLConnection`
  - Communication link between the app and the URL
  - Built-in support for HTTP and JAR connections
  - Read/write from/to connection using I/O streams

```
import java.io.*;
import java.net.*;

public class WebClient {
  public static void main(String[] args) throws Exception {
    if (args.length != 1) {
      System.err.println("Usage: WebClient <url>");
      return;
    }
    URL url = new URL(args[0]);
    URLConnection connection = url.openConnection();
    connection.connect();
    InputStream in = connection.getInputStream();
    try {
      byte[] buf = new byte[1024];
      int nread = 0;
      while ((nread = in.read(buf)) > 0) {
        System.out.write(buf, 0, nread);
      }
      System.out.flush();
    } finally {
      in.close();
    }
  }
}
```

12.3 TCP Sockets

- `java.net.Socket`
  - End-point for communication between two hosts
  - Options: `TCP_NODELAY`, `SO_KEEPALIVE`, `SO_TIMEOUT`, `SO_LINGER`, `SO_SNDBUF`, etc.
  - Read/write using I/O byte streams
- `java.net.ServerSocket`
  - Accepts incoming socket connections
  - Once accepted use `Socket` for communication
  - Supports OS queuing (backlog)
  - Supports blocking and non-blocking modes

```
import java.io.InputStream;
import java.io.OutputStream;
import java.io.IOException;
import java.net.Socket;
import java.net.ServerSocket;

public class SingleThreadedEchoServer {
    public static void main(String[] args) throws IOException {
        int port = Integer.parseInt(args[0]);
        ServerSocket serverSocket = new ServerSocket(port);
        System.out.println("Listening on port " + port);
        System.out.println("(Control-C to terminate)");
        while(true) {
            Socket clientSocket = serverSocket.accept(); // blocks
            try {
                InputStream in = clientSocket.getInputStream();
                OutputStream out = clientSocket.getOutputStream();
                int b;
                while ((b = in.read()) != -1) {
                    out.write(b);
                }
                out.flush();
            } finally {
                clientSocket.close();
            }
        }
    }
}
```

Note
UDP sockets are supported through `java.net.DatagramSocket`.

Chapter 13

Java Collections and Generics

- The Java Collections Framework
- Java Generics

13.1 The Java Collections Framework

- Java arrays have limitations.
  - They cannot dynamically shrink and grow.
  - They have limited type safety.
  - Implementing efficient, complex data structures from scratch would be difficult.
- The Java Collections Framework is a set of classes and interfaces implementing complex collection data structures.
  - A *collection* is an object that represents a group of objects.
- The Java Collections Framework provides many benefits:
  - Reduces programming effort (already there)
  - Increases performance (tested and optimized)
  - Part of the core API (available, easy to learn)
  - Promotes software reuse (standard interface)
  - Easy to design APIs based on generic collections

The Java Collections Framework consists of:

Collection interfaces
: Collection, List, Set, Map, etc.

General-purpose implementations
: ArrayList, LinkedList, HashSet, HashMap, etc.

Special-purpose implementations
: designed for performance characteristics, usage restrictions, or behavior

Concurrent implementations
: designed for use in high-concurrency contexts

Wrapper implementations
adding synchronization, immutability, etc.

Abstract implementations
building blocks for custom implementations

Algorithms
static methods that perform useful functions, such as sorting or randomizing a collection

Infrastructure
interfaces to support the collections

Array utilities
static methods that perform useful functions on arrays, such as sorting, initializing, or converting to collections

More information on the Java Collections Framework is available at: http://download.oracle.com/javase/tutorial/collections/-index.html

13.2 The Collection Interface

- `java.util.Collection` is the root interface in the collections hierarchy.
  - It represents a group of Objects.
  - Primitive types (e.g., `int`) must be boxed (e.g., Integer) for inclusion in a collection.
  - More specific collection interfaces (e.g., List) extend this interface.
- The Collection interface includes a variety of methods:
  - Adding objects to the collection: `add(E)`, `addAll(Collection)`
  - Testing size and membership: `size()`, `isEmpty()`, `contains(E)`, `containsAll(Collection)`
  - Iterating over members: `iterator()`
  - Removing members: `remove(E)`, `removeAll(Collection)`, `clear()`, `retainAll(Collection)`
  - Generating array representations: `toArray()`, `toArray(T[])`

The Collection interface does not say anything about:

- the order of elements
- whether they can be duplicated
- whether they can be `null`
- the types of elements they can contain

This interface is typically used to pass collections around and manipulate them in the most generic way.

13.3 Iterating Over a Collection

- An *iterator* is an object that iterates over the objects in a collection.
- `java.util.Iterator` is an interface specifying the capabilities of an iterator.
  - Invoking the `iterator()` method on a collection returns an iterator object that implements Iterator and knows how to step through the objects in the underlying collection.
- The Iterator interface specifies the following methods:
  - `hasNext()` - Returns `true` if there are more elements in the collection; `false` otherwise
  - `next()` - Returns the next element
  - `remove()` - Removes from the collection the last element returned by the iterator

---

For example, to print all elements in a collection:

```
private static void print(Collection c) {
    Iterator i = c.iterator();
    while (i.hasNext()) {
        Object o = i.next();
        System.out.println(o);
    }
}
```

This can also be written as:

```
private static void print(Collection c) {
    for (Iterator i = c.iterator(); i.hasNext(); ) {
        System.out.println(i.next());
    }
}
```

And in Java 5, the Iterator can be used implicitly thanks to `for-each`:

```
private static void print(Collection c) {
    for (Object o : c) {
        System.out.println(o);
    }
}
```

13.4 The List Interface

- The `java.util.List` Interface extends the Collection interface.
- The List interface declares methods for managing an ordered collection of object (a *sequence*). You can:
  - Control where each element is inserted in the list
  - Access elements by their integer index (position in the list)
  - Search for elements in the list
  - Insert duplicate elements and `null` values, in most List implementations

Tip
Especially if you're dynamically resizing a collection of object, favor using a List over a Java array.

- Some additional methods provided by List include:
  - `add(int index, E element)` — Insert an element at a specific location (without an index argument, new elements are appended to the end)
  - `get(int index)` — Return an element at the specified location
  - `remove(int index)` — Remove an element at the specified location
  - `set(int index, E element)` — Replace the element at the specified location
  - `subList(int fromIndex, int toIndex)` — Returns as a List a *modifiable view* of the specified portion of the list (that is, changes to the view actually affect the underlying list)

13.5 Classes Implementing List

- Java provides several classes that implement the List interface.
- Two are used most commonly:

`java.util.ArrayList`
An ArrayList is usually your best bet for a List if the values remain fairly static once you've created the list. It's more efficient than a LinkedList for random access.

`java.util.LinkedList`
A LinkedList provides better performance than an ArrayList if you're frequently inserting and deleting elements, especially from the middle of the collection. But it's slower than an ArrayList for random-access.

In the following example, notice the use of the `java.util.Collections.sort()` static method, which does an in-place sort of the elements in a List:

```
package util;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;

/**
 * This program reads arguments passed to it from the command-line
 * or the lines from STDIN, sorts them, and writes the result to STDOUT.
 */
public class Sort {
    public static void main(String[] args) throws IOException {
        if (args.length > 0) {
            Arrays.sort(args);
            for (int i = 0; i < args.length; i++) {
                System.out.println(args[i]);
            }
        } else {
            List lines = new ArrayList();
```

```
            BufferedReader reader = new BufferedReader(new InputStreamReader(
                    System.in));
            String line = null;
            while ((line = reader.readLine()) != null) {
                lines.add(line);
            }
            Collections.sort(lines);
            for (Iterator i = lines.iterator(); i.hasNext();) {
                System.out.println(i.next());
            }
        }
    }
}
```

13.6 The Set and SortedSet Interfaces

- The `java.util.Set` Interface extends the Collection interface.
- The Set interface declares methods for managing a collection of objects that contain no duplicates.
  - The basic Set interface makes no guarantee of the order of elements.
  - A Set can contain at most one `null` element.
  - Mutable elements should not be changed while in a Set.
- The Set interface adds no methods beyond those of the Collection interface.
  - The Set interface simply enforces behavior of the collection.
- The `java.util.SortedSet` interface extends Set so that elements are automatically ordered.
  - A SortedSet Iterator traverses the Set in order.
  - A SortedSet also adds the methods `first()`, `last()`, `headSet()`, `tailSet()`, and `subSet()`

13.7 Classes Implementing Set

- Java provides several classes that implement the Set interface, including:

+ `java.util.HashSet`:: A hash table implementation of the Set interface. The best all-around implementation of the Set interface, providing fast lookup and updates.

`java.util.LinkedHashSet`
: A hash table and linked list implementation of the Set interface. It maintains the insertion order of elements for iteration, and runs nearly as fast as a HashSet.

`java.util.TreeSet`
: A red-black tree implementation of the SortedSet interface, maintaining the collection in sorted order, but slower for lookups and updates.

```
package util;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

/**
 * This program computes a unique set of elements passed to it either from
 * command line (as arguments) or from STDIN (as lines).
 *
 * This demonstrates the use of Sets.
 */
public class Unique {

    public static void main(String[] args) throws IOException {
        Set unique = new HashSet(); // replace with TreeSet to get them sorted
        if (args.length > 0) {
            unique.addAll(Arrays.asList(args));
        } else {
            BufferedReader reader = new BufferedReader(new InputStreamReader(
                    System.in));
            String line = null;
            while ((line = reader.readLine()) != null) {
                unique.add(line);
            }
        }
        for (Iterator i = unique.iterator(); i.hasNext();) {
            System.out.println(i.next());
        }
    }
}
```

13.8 The Queue Interface

- The `java.util.Queue` interface is a collection designed for holding elements prior to processing.
  - Besides basic Collection operations, queues provide additional insertion, extraction, and inspection operations.
  - Queues can implement FIFO (queue), LIFO (stack), and priority ordering.
  - Queues generally do not accept `null` elements.
- Queue operations include:
  - Inserting elements: `add()` and `offer()`
  - Removing and returning elements: `remove()` and `poll()`
  - Returning elements without removal: `element()` and `peek()`
- The `java.util.concurrent.BlockingQueue` interfaces declare additional blocking `put()` and `take()` methods, designed for concurrent access by multiple threads.

General-purpose Queue implementations:

- `java.util.LinkedList`
- `java.util.PriorityQueue`

Concurrency-specific BlockingQueue implementations:

- `java.util.concurrent.ArrayBlockingQueue`
- `java.util.concurrent.ConcurrentLinkedQueue`
- `java.util.concurrent.DelayQueue`
- `java.util.concurrent.LinkedBlockingQueue`
- `java.util.concurrent.PriorityBlockingQueue`
- `java.util.concurrent.SynchronousQueue`

13.9 The Map Interface

- The `java.util.Map` interface does **not** extend the Collection interface!
- A Map is a collection that maps key objects to value objects.
  - A Map cannot contain duplicate keys
  - Each key can map to at most one value.
- The Map interface methods include:
  - Adding key-value pairs to the collection: `put(K, V)`, `putAll(Map)`
  - Retrieving a value by its key: `get(K)`
  - Testing size: `size()`, `isEmpty()`
  - Testing membership: `containsKey(K)`, `containsValue(V)`
  - Removing members: `remove(K)`, `clear()`
- The `java.util.SortedMap` interface extends Map so that elements are automatically ordered by key.
  - Iterating over a SortedMap iterates over the elements in key order.
  - A SortedMap also adds the methods `firstKey()`, `lastKey()`, `headMap()`, `tailMap()`, and `subMap()`

13.10 Retrieving Map Views

- You can also retrieve collections as *modifiable views* from the Map representing the keys (`keySet()`), the values (`values()`), and a Set of key-value pairs (`entrySet()`).
  - These collections are used typically to iterate over the Map elements.
  - These collections are backed by the Map, so changes to the Map are reflected in the collection views and vice-versa.
- Iterating over Map elements is done typically with a Set of objects implementing the `java.util.Map.Entry` interface.
  - You retrieve the Set of Map.Entry objects by invoking `Map.entrySet()`.
  - To iterate over the Map elements using an explicit Iterator:

```
for (Iterator i = map.entrySet().iterator; i.hasNext(); ) {
    Map.Entry entry = (Map.Entry) i.next();
    System.out.println( entry.getKey() + ":\t" + entry.getValue() );
}
```

– To iterate over the Map elements using the `for-each` syntax:

```
for ( Map.Entry entry: map.entrySet() ) {
    System.out.println(entry.getKey() + ":\t" + entry.getValue());
}
```

13.11 Classes Implementing Map

- Java provides several classes that implement the Map interface, including:

+ `java.util.HashMap`:: A hash table implementation of the Map interface. The best all-around implementation of the Map interface, providing fast lookup and updates.

`java.util.LinkedHashMap`
: A hash table and linked list implementation of the Map interface. It maintains the insertion order of elements for iteration, and runs nearly as fast as a HashMap.

`java.util.TreeMap`
: A red-black tree implementation of the SortedMap interface, maintaining the collection in sorted order, but slower for lookups and updates.

```
package util;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

/**
 * This program computes the frequency of words passed to it either as command
 * line arguments or from STDIN. The result is written back to STDOUT.
 *
 * This demonstrates the use of Maps.
 */
public class WordFrequency {

    public static void main(String[] args) throws IOException {
        // replace with TreeMap to get them sorted by name
        Map wordMap = new HashMap();
        if (args.length > 0) {
            for (int i = 0; i < args.length; i++) {
                countWord(wordMap, args[i]);
            }
        } else {
            getWordFrequency(System.in, wordMap);
        }
```

```
        for (Iterator i = wordMap.entrySet().iterator(); i.hasNext();) {
            Map.Entry entry = (Map.Entry) i.next();
            System.out.println(entry.getKey() + " :\t" + entry.getValue());
        }

    }

    private static void getWordFrequency(InputStream in, Map wordMap)
            throws IOException {
        BufferedReader reader = new BufferedReader(new InputStreamReader(in));
        int ch = -1;
        StringBuffer word = new StringBuffer();
        while ((ch = reader.read()) != -1) {
            if (Character.isWhitespace(ch)) {
                if (word.length() > 0) {
                    countWord(wordMap, word.toString());
                    word = new StringBuffer();
                }
            } else {
                word.append((char) ch);
            }
        }
        if (word.length() > 0) {
            countWord(wordMap, word.toString());
        }
    }

    private static void countWord(Map wordMap, String word) {
        Integer count = (Integer) wordMap.get(word);
        if (count == null) {
            count = new Integer(1);
        } else {
            count = new Integer(count.intValue() + 1);
        }
        wordMap.put(word, count);
    }
}
```

13.12 The Collections Class

- The `java.util.Collections` class (note the final "s") consists exclusively of static methods that operate on or return collections. Features include:
  - Taking a collection and returning an unmodifiable view of the collection
  - Taking a collection and returning a synchronized view of the collection for thread-safe use
  - Returning the minimum or maximum element from a collection
  - Sorting, reversing, rotating, and randomly shuffling List elements
- Several methods of the Collections class require objects in the collection to be comparable.
  - The object class must implement the `java.lang.Comparable` interface.
  - The object class must provide an implementation of the `compareTo()` method, which a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Note
There are several requirements for proper implementation of the `compareTo()` method. See the reference documentation for more information. The book *Effective Java, 2nd ed.*, by Joshua Bloch also has excellent information on the requirements and pitfalls of proper implementation of the `compareTo()` method.

13.13 Type Safety in Java Collections

- The Java Collections Framework was designed to handle objects of any type.
  - In Java 1.4 and earlier they used Object as the type for any object added to the collection.
  - You had to explicitly cast the objects to the desired type when you used them or else you would get compile-time errors.

    ```
    Employee e = (Employee) list.get(0);
    ```

  - Worse yet, if you were dealing with a collection of objects, say of type Dog, and then accidentally added an object of an incompatible type, say a Cat, your code could eventually try to cast the object to the incompatible type, resulting in a runtime exception.

Java arrays actually have a similar polymorphism problem that can result in runtime exceptions:

```
package com.markana.demo;

abstract class Animal {
    abstract public void speak();

    public void identify() {
        System.out.println("I'm an animal.");
    }
}

class Dog extends Animal {
    @Override
    public void speak() {
        System.out.println("woof");
    }

    @Override
    public void identify() {
        System.out.println("I'm a Dog.");
    }
}

class Cat extends Animal {
    @Override
    public void speak() {
        System.out.println("meow");
    }

    @Override
    public void identify() {
        System.out.println("I'm a Cat.");
    }
}

package com.markana.demo;
```

```
/** Demonstration that arrays are not completely type-safe
 * @author Ken Jones
 *
 */
public class ArrayTypeError {

    public static void main(String[] args) {
        // Create an array of three anonymous dogs
        Dog[] kennel = { new Dog(), new Dog(), new Dog()};

        // Let them all speak
        for (Dog d: kennel) d.speak();

        // Dogs are Objects, so this should work
        Object[] things = kennel;

        /* A Cat is an Object, so we should be able to add one to the
         * things array. Note that the following does NOT cause a
         * compiler error! Instead it throws a RUNTIME exception,
         * ArrayStoreException.
         */
        things[0] = new Cat();
    }

}
```

13.14 Java Generics

- Java Generics, introduced in Java 5, provide stronger type safety.
- Generics allow *types* to be passed as *parameters* to class, interface, and method declarations. For example:

```
List<Employee> emps = new ArrayList<Employee>();
```

- The `<Employee>` in this example is a *type parameter*.
  - With the type parameter, the *compiler* ensures that we use the collection with objects of a compatible type only.
  - Another benefit is that we won't need to cast the objects we get from the collection:

    ```
    Employee e = emps.get(0);
    ```

  - Object type errors are now detected at compile time, rather than throwing casting exceptions at runtime.

13.15 Generics and Polymorphism

- What can be confusing about generics when you start to use them is that collections of a type are **not** polymorphic on the *type*.
  - That is, you can **not** assign a `List<String>` to a reference variable of type `List<Object>` (and by extension, pass a `List<String>` as an argument to a method whose parameter is type `List<Object>`); it results in a compiler error.
  - Why? If allowed, we could then add objects of an incompatible type to the collection through the more "generic" typed reference variable.
- So if you define a `printCollection()` to accept a parameter typed `List<Person>`, you can pass only `List<Person>` collections as arguments.

- Even if Employee is a subclass of Person, a `List<Employee>` can't be assigned to a `List<Person>`.

Here's an illustration of how type parameters are not polymorphic for collections:

```
package com.markana.demo;

import java.util.*;

public class GenericsTypeError {
    public static void main( String[] args) {
        // Create a List of Dog objects
        List<Dog> kennel = new ArrayList<Dog>();

        // Adding a Dog is no problem
        kennel.add( new Dog() );

        // The following line results in a compiler error
        List<Object> objs = kennel;

        // Because if it were allowed, we could do this
        objs.add( new Cat() );

        // And now we've got a Cat in our List of Dogs
    }
}
```

13.16 Type Wildcards

- The `?` type parameter *wildcard* is interpreted as "type unknown."
  - So declaring a variable as `List<?>` means that you can assign a List of any type of object to the reference variable.
  - However, once assigned to the variable, you can't make any assumptions about the type of the objects in the list.
- So defining your method as `printCollection(List<?> persons)` means that it can accept a List of any type.
  - But when invoked, you can't make any assumptions as to the type of objects that the list contains.
  - Remember, `?` is "type unknown".
  - At best, you know that everything can be treated as an Object.

```
package com.markana.demo;

import java.util.*;

public class GenericsWildcardExample1 {

    public static void main( String[] args) {
        // Create a List of Dog objects
        List<Dog> kennel = new ArrayList<Dog>();

        // Adding a Dog is no problem
        kennel.add( new Dog() );
```

```
        // The following line compiles without error
        List<?> objs = kennel;

        /*
         * But now we can't make any assumptions about the type of
         * objects in the objs List. In fact, the only thing that
         * we can safely do with them is treat them as Objects.
         * For example, implicitly invoking toString() on them.
         */

        for (Object o: objs) {
            System.out.println("String representation: " + o);
        }

    }
}
```

13.17 Qualified Type Wildcards

- Declaring a variable or parameter as `List<? extends Person>`, says that the list can be of all Person objects, or all objects can be of (the same) subclass of Person.
  - So you can access any existing object in the List as a Person.
  - However, you can't **add** new objects to the List.
  - The list might contain all Person objects... or Employee objects, or Customer objects, or objects of some other subclass of Person. You'd be in trouble if you could add a Customer to a list of Employees.
- Another type wildcard qualifier is `super`.
  - `List<? super Employee>` means a List of objects of type Employee or some supertype of Employee.
  - So the type is "unknown," but in this case it could be Employee, Person, or Object.
  - Because you don't know which, for "read" access the best you can do is use only Object methods.
  - But you can add new Employee objects to the list, because polymorphism allows the Employee to be treated as a Person or Object as well.
- Both `extends` and `super` can be combined with the wildcard type.

```
package com.markana.demo;

import java.util.*;

public class GenericsWildcardExample2 {

    public static void main( String[] args) {
        // Create a List of Dog objects
        List<Dog> kennel = new ArrayList<Dog>();

        // Adding a Dog is no problem
        kennel.add( new Dog() );

        /*
         * We can assign to objs a reference to any List as long
         * as it contains objects of type Animal or some subclass
         * of Animal.
```

```
         */

        List<? extends Animal> objs = kennel;

        /*
         * Now we know that the objects in the objs List are
         * all Animals or all the same subclass of Animal. So
         * we can safely access the existing objects as Animals.
         * For example, invoking identify() on them.
         */

        for (Animal o: objs) {
            o.identify();
        }

        /*
         * However, it would be a compilation error to try to
         * add new objects to the list through objs. We don't know
         * what type of objects the List contains. They might be
         * all Dogs, or all Cats, or all "generic" Animals.
         */
    }
}
```

13.18 Generic Methods

- A *generic method* is one implemented to work with a variety of types.
  - The method definition contains one or more type parameters whose values are determined when the method is **invoked**.
  - Type parameters are listed within <> after any access modifiers and before the method return type.
  - You can use any identifier for a type parameter, but single letters like `<T>` are used most often.
- For example:

```
static <T> void addToCollection(T p, Collection<T> c) {
    c.add(p);
}
```

  - In this example, the object `p` and the objects in the collection `c` must all be the same type.
- You can use type wildcards, `extends`, and `super` in generic method definitions as well.

```
package com.markana.demo;

import java.util.*;

public class GenericsWildcardExample3 {

    public static <T> void add1( T obj, Collection<? super T> c) {
        c.add(obj);
    }

    public static <U, T extends U> void add2( T obj, Collection<U> c) {
        c.add(obj);
    }
```

```
    public static void main( String[] args) {

        // Create a List of Cat and Dog objects
        List<Animal> menagerie = new ArrayList<Animal>();

        // Add a Cat and a Dog
        menagerie.add( new Cat() );
        menagerie.add( new Dog() );

        // And now let's try using our generic methods to add objects
        add1( new Cat(), menagerie);
        add2( new Dog(), menagerie);

        for (Animal o: menagerie) {
            o.identify();
        }

    }
}
```

Chapter 14

Threading

14.1 Runnable And Thread

- To run an object in a thread:
  - Implement `Runnable`'s `run()` and `start()` in a new `Thread` (preferred)
  - Extend `Thread`, override `run()`, instantiate, and `start()`
- Threads can: `join`, `interrupt`, `sleep`, `yield`, be prioritized, stack-dumped, be enumerated, grouped, etc.

```
public class ThreadDemo {
  public static void main(String[] args) throws InterruptedException {
    int numThreads = Integer.parseInt(args[0]);
    int count = Integer.parseInt(args[1]);
    Thread[] threads = new Thread[numThreads];
    System.out.println("Creating threads");
    for (int i = 0; i < threads.length; i++) {
      threads[i] = new Thread(new Runner("Runner " + i, count));
    }
    System.out.println("Starting threads");
    for (int i = 0; i < threads.length; i++) { threads[i].start(); }
    System.out.println("Waiting for threads");
    for (int i = 0; i < threads.length; i++) { threads[i].join(); }
    System.out.println("Done");
  }
}
public class Runner implements Runnable {
  private final String name;
  private final int count;
  public Runner(String name, int count) {
    this.name = name; this.count = count;
  }
  public void run() {
    for (int i = 0; i < count; i++) {
      System.out.println(name + "=" + i);
      Thread.yield();
    }
  }
}
```

14.2 Thread Synchronization

- Use `synchronized` keyword to protect critical sections from concurrent access
  - Do not allow data/structures to be modified by multiple threads simultaneously
  - Can be used on methods or blocks of code
- Synchronization occurs on an object that acts as a *monitor* for the running threads
  - Usually, this is the object that is to be protected
  - Can be any other object (even generic instances of `Object`): `Object lock = new Object();`

```
public class Counter {
  private int count = 0;
  public void setCount(int count) { this.count = count; }
  public int getCount() { return this.count; }
}
public class Runner implements Runnable {
  private final int maxCount;
  private final Counter counter;
  public Runner(int maxCount, Counter counter) {
    this.maxCount = maxCount; this.counter = counter;
  }
  public void run() {
    for (int i = 0; i < this.maxCount; i++) {
      int currentCount;
      synchronized(this.counter) {
        currentCount = this.counter.getCount();
        currentCount++;
        this.counter.setCount(currentCount);
      }
      System.out.println(Thread.currentThread().getName()
        + " at count " + currentCount);
    }
  }
}
```

A better approach would be to add a synchronized count method on the Counter class so that Runners would not have to worry about synchronization.

```
public synchronized int count() {
  return ++this.count;
}
```

14.3 More Thread Synchronization

- Use `Object.wait()`, `Object.notify()`, and `Object.notifyAll()` to schedule threads (wait on each other)
  - Solves the classic Consumer-Producer problem
- Methods `wait()`, `notify()`, and `notifyAll()` must be called in `synchronized` blocks
- Always do checks around `wait()` using `while` or `for` loops rather than simple `if` conditions
- Better yet, use Java 5's `java.util.concurrent` advanced thread-safe data structures and controls

```
public class Queue {
  private static class Element {
    final Object value;
    Element next;
    Element(Object value) {this.value = value;}
  }
  private Element first, last;
  private int curSize, maxSize;

  public Queue(int maxSize) { this.maxSize = maxSize; }

  public synchronized void put(Object o) throws InterruptedException {
    while (this.curSize == this.maxSize) { this.wait(); }
    if (this.first == null) {
      this.first = (this.last = new Element(o));
    } else {
      this.last = (this.last.next = new Element(o));
    }
    this.curSize++;
    this.notifyAll();
  }
  public synchronized Object get() throws InterruptedException {
    while (this.curSize == 0) { this.wait(); }
    Object o = this.first.value;
    this.first = this.first.next;
    this.curSize--;
    this.notifyAll();
    return o;
  }
}
```

14.4 Java 5 Concurrency Features

- The Java 5's `java.util.concurrent` package provides a powerful, extensible framework of high-performance, scalable, thread-safe building blocks for developing concurrent classes and applications
  - Thread pools and custom execution frameworks
  - Queues and related thread-safe collections
  - Atomic variables
  - Special-purpose locks, barriers, semaphores and condition variables
- Better than `synchronized`, `wait()`, `notify()`, ...

- Its packages free the programmer from having to write these utilities by hand, like Collections did for data structures
- The packages also provide low-level primitives for advanced concurrent programming
- Built-in concurrency primitives : `wait()`, `notify()`, `synchronized` are too hard:
  - Hard to use
  - Easy to make errors
  - Low level for most applications

  - Poor performance if used incorrectly
  - Too much wheel-reinventing (not standardized)
- Concurrency APIs in Java 5
  - `java.util.concurrent` - Utility classes and frameworks commonly used in concurrent programming - e.g. thread pools
  - `java.util.concurrent.atomic` - Classes that support lock-free and thread-safe operations on single variables - e.g. atomic `i++`
  - `java.util.concurrent.locks` - Framework for locking and waiting for conditions that is distinct from built-in synchronization and monitors

14.5 Thread Scheduling Pre-Java 5

```
public class NetworkedService {
  public void service(int port) throws Exception {
    ServerSocket ss = new ServerSocket(port);
    while(true) {
      new Thread(new Handler(ss.accept())).start(); //❶
    }
  }
  private static class Handler implements Runnable {
    private final Socket s;
    public Handler(Socket s) { this.s = s; }
    public void run() {
      //❷
    }
  }
}
```

❶ New thread every time - poor resource management

❷ Read requests, access registry, return response

Note
With no access to a standard thread-pool library, we end up creating Threads on demand. Works well for very simple cases, but does not scale and it is not resource-efficient.

14.6 Thread Scheduling In Java 5

```
public class NetworkedService {
  public void service(int port) throws Exception {
    ServerSocket ss = new ServerSocket(port);
    ExecutorService pool = Executors.newCachedThreadPool(); //❶

    while(true) {
      pool.execute(new Handler(ss.accept())); //❷
    }
  }
  private static class Handler implements Runnable {
    //❸
  }
}
```

1. Pick a pool scheduling strategy that fits our needs
2. Let the executor service manage the threads as needed
3. Same implementation as before

- Java 5's Scheduling framework standardizes invocation, scheduling, execution, and control of asynchronous tasks
- We can use its thread-pool executor service to provide better resource utilization
- Executor interface decouples task submission from the mechanics of how it will be run:

```
public interface Executor {
  public void execute(Runnable command);
}
```

- Very easy to build different execution strategies:

```
public class InThreadExecutor implements Executor {
  public void execute(Runnable r) {
    r.run();
  }
}

public class ThreadPerTaskExecutor implements Executor {
  public void execute(Runnable r) {
    new Thread(r).start();
  }
}
```

14.7 Java 5's `ExecutorService`

- `ExecutorService` builds upon `Executor` to provide lifecycle management and support for tracking progress of asynchronous tasks
- Constructed through through factory methods on `Executors` class
- `ScheduledExecutorService` provides a framework for executing deferred and recurring tasks that can be cancelled
  - Like `Timer` but supports pooling

```
public interface ExecutorService extends Executor {
  /** Stop accepting new tasks */
  public void shutdown();
  /** Stop accepting new tasks, cancel tasks
      currently waiting, and interrupt active tasks */
  public List<Runnable> shutdownNow();
  public boolean isShutdown();
  public boolean isTerminated();
  public boolean awaitTermination(long timeout, TimeUnit unit);
  public void execute(Runnable c);
  public <T> Future<T> submit(Callable<T> task);
  ...
}
```

```
public class Executors {
  public static ExecutorService newSingleThreadedExecutor();
  public static ExecutorService newFixedThreadPool(int n);
  public static ExecutorService newCachedThreadPool();
  ...
}
```

14.8 Getting Results From Threads Pre Java 5

- Getting results from operations that run in separate threads requires a lot of boiler-plate code
- We need a custom `Runnable` that remembers its result (or exception) as its state and we explicit access to it
- What if we needed to wait on the result? We need to `join()` the thread (boring)

```
public class Handler implements Runnable {
  private Exception e = null;
  private Long result = null;
  public synchronized void run() {
    long t = System.currentTimeMillis();
    try {
      // handle request
      this.result = System.currentTimeMillis() - t;
    } catch (Exception e) {
      this.e = e;
    } finally {
      this.notifyAll();
    }
  }
  public synchronized long getResult() throws Exception {
    while (result == null && e == null) {
      this.wait();
    }
    if (e != null) {
      throw e;
    } else {
      return result;
    }
  }
}
```

14.9 Getting Results From Threads In Java 5

- Java 5 provides a new interface to define a *runnable* task that returns a result and may throw an exception:

```
public interface Callable<V> {
  public V call() throws Exception;
}
```

- `Future` - the result of an async action
  - An easy way to get the task status, block and wait for its result, or cancel it

---

Our Handler becomes much simpler with a Callable:

```
public class Handler implements Callable<Long> {
  public Long call() throws Exception {
    long t = System.currentTimeMillis(); //❶
    return System.currentTimeMillis() - t;
  }
}
```

❶ Handle request that can throw an Exception

Futures make it easy to define asynchronous operations:

```
public interface Future<V> {
  public V get() throws InterruptedExeption,ExecutionException;
  public V get(long timeout, TimeUnit unit);
  public boolean cancel(boolean mayInterrupt);
  public boolean isCanceled();
  public boolean isDone();
}
```

And `ExecutorService.submit(Callable<T> task)` conveniently returns `Future<T>` for that task. We just have to call `Future.get()` to wait on the result.

14.10 Thread Sync Pre-Java 5

- Java's `synchronized` construct forces all lock acquisition and release to occur in a block-structured way
  - Cannot easily implement chain-locking
  - Often used inefficiently
- Using `Object.wait()` and `Object.notify()` for thread sync is extremely hard to do well

---

```
public class BlockingMap<K,V> {
  private final Map<K,V> map = new HashMap<K,V>();

  // only one get() a time - poor read performance
  public synchronized V get(K key) {
    return map.get(key);
  }
  public synchronized V require(K key) {
    V value = null;
    while((value = map.get(key)) == null) {
      try {
        this.wait();
      }
      catch (InterruptedException e) {
        break;
      }
    }
    return value;
  }
  public synchronized void set(K key, V value) {
```

```
    map.put(key, value);
    this.notifyAll();
  }
}
```

14.11 Thread Sync In Java 5 With Locks

- Locks and condition variables provide an alternative to implicit object monitors
- `ReadWriteLock` allows concurrent read access to a shared resource
- Possible to provide deadlock detection and non-reentrant usage semantics
- Support for guaranteed fair-ordering
  - Access to longest-waiting thread
- But with increased flexibility, comes additional responsibility (use `try-finally` to unlock)

```
public class BlockingMap<K,V> {
  private Map<K,V> map = ...
  private ReadWriteLock lock = new ReentrantReadWriteLock();
  private Condition vAdded = lock.writeLock().newCondition();
  public V get(K key) {
    lock.readLock().lock();
    try {
      return map.get(key);
    } finally {
      lock.readLock().unlock();
    }
  }
  public V require(K key) {
    V value = get(key); // try cheap first
    if (value == null) {
      lock.writeLock().lock();
      try {
        while((value = map.get(key)) == null) {
          vAdded.await();
        }
      } catch (InterruptedException giveup) {
        ...
      } finally {
        lock.writeLock().unlock();
      }
    }
    return value;
  }
  public void set(String key, String value) {
    lock.writeLock().lock();
    try {
      map.put(key, value);
      vAdded.signalAll();
    } finally {
      lock.writeLock().unlock();
    }
  }
}
```

14.12 Benefits Of Java 5 Locks

- We can do chain-locking (e.g. support for data traversal algorithms)
- Non-blocking lock attempts: `aLock.tryLock();`
- Interrupted lock attempts: `aLock.lockInterruptibly();`
- Timing out lock attempts: `aLock.tryLock(5, TimeUnit.SECONDS);`

```
if (lock.tryLock()) {
  try {
    // do work that requires a lock
  } finally {
    lock.unlock();
  }
} else {
  // the lock was not available
}

try {
  lock.lockInterruptibly();
  try {
    // do work that requires a lock
  } finally {
    lock.unlock();
  }
} catch (InterruptedException e) {
  // interrupted while waiting
}

try {
  if (lock.tryLock(10, TimeUnit.SECONDS)) {
    try {
      // do work that requires a lock
    } finally {
      lock.unlock();
    }
  }
} catch (InterruptedException e) {
  // interrupted while waiting
}
```

14.13 Java 5 Conditions

- Conditions provide means for one thread to suspend its execution until notified by another
- What locks are to `synchronized`, Conditions are to the `Object` class's monitor methods (`wait`, `notify`, and `notifyAll`)
- Support for multiple conditions per lock, uninterruptible conditions, time-wait conditions, absolute time waits

---

- Support for uninterruptible conditions: `Condition.awaitUninterruptibly()`
- Timed waits on conditions tell you why it returned: `Condition.await(long time, TimeUnit unit)` returns `false` if time elapsed
- Support for absolute time waits: `Condition.awaitUntil(Date deadline)`

14.14 Atomics In Java 5

- Prior to Java 5, operations such as `i++` had to be wrapped in `synchronized` if thread safety was important
- Now we have *atomic* "primitives"
  - Easier and more efficient
  - Can be used in arrays/collections
  - Building blocks for non-blocking data structures

```
public class AtomicInteger extends Number {
  public int incrementAndGet() {...}
  public int decrementAndGet() {...}
  public int getAndIncrement() {...}
  public int getAndDecrement() {...}
  ...
}
public class AtomicLong extends Number {
  public boolean compareAndSet(long expected,
    long update) {...}
  public long addAndGet(long delta) {...}
  ...
}
public class AtomicBoolean {
  public boolean getAndSet(boolean newValue) {...}
  ...
}
```

14.15 Other Java 5 Concurrency Features

- Concurrent collections offer thread-safe iteration and better performance than `HashTable` and `Vector`
  - `ConcurrentHashMap`
  - `CopyOnWriteArrayList`
- General purpose synchronizers: `Latch`, `Semaphore`, `Barrier`, `Exchanger`
- Nanosecond-granularity timing: `aLock.tryLock(250, TimeUnit.NANOSECONDS)`

- `java.util.concurrent.Semaphore` - Counting semaphore maintains a set of permits and is generally used to restrict number of threads that can access a resource
- `java.util.concurrent.CyclicBarrier` - Aids a set of threads to wait for each other to reach a common “barrier” point
- `java.util.concurrent.CountDownLatch` – Allows a set of threads to wait for some number of operations performed by other thread(s) to complete
- `java.util.concurrent.Exchanger<V>` - A synchronization point at which a pair of threads can exchange objects

Chapter 15

Additional Java Features

15.1 The Date Class

- The `java.util.Date` class represents a specific instant in time.
  - The time is measured as a signed `long` representing the number of milliseconds since the *epoch*.
  - In Java, the epoch is defined as January 1, 1970, 00:00:00 UTC (GMT).
- Invoking the Date constructor with:
  - A long value initializes the object to represent that time
  - No arguments initializes the object to represent the object's creation time
- The Date class includes methods for:
  - Retrieving the time: `getTime()`
  - Changing the object's time: `setTime(long)`
  - Comparing Date objects: `after(Date)`, `before(Date)`, `compareTo(Date)`, `equals(Object)`
  - Generating a String representation in a fixed format: `toString()`

Caution
Most other Date methods are deprecated. They supported time and date manipulation, parsing, and formatting, but they have been superseded by newer classes.

The `System.currentTimeMillis()` returns the current epoch-based time as a `long`.

15.2 The Calendar Class

- The `java.util.Calendar` class is used to represent a specific instance in time. It supports methods for:
  - Setting individual time and date fields of the object (month, day, hour, minute, etc.)
  - Performing time and date arithmetic
  - Performing comparisons with other Calendar objects

  - Retrieving the represented time as a Date object or an epoch-based `long`

- The Calendar class is an abstract class, to support concrete implementations for different calendar styles and languages.
  - The `java.util.GregorianCalendar` class is a concrete implementation of the Gregorian calendar used by most of the world.
  - Invoking the static `Calendar.getInstance()` method returns a localized Calendar object whose calendar fields have been initialized with the current date and time.

15.3 The TimeZone Class

- The `java.util.TimeZone` class represents a time zone offset, and also figures out daylight savings time.
- Typical use is to invoke the static `TimeZone.getTimeZone()` method to return a TimeZone object based on the system's time zone setting.
  - You can also invoke `TimeZone.getTimeZone()` with a String time zone ID. For example:

```
TimeZone tz = TimeZone.getTimeZone("America/Los_Angeles");
```

```
package util;

import java.util.Date;
import java.util.Calendar;
import java.util.TimeZone;

public class Now {
    public static void main(String[] args) {
        Date date = new Date();
        System.out.println(date);
        System.out.println(date.getTime() + " ms since the epoch");

        Calendar c = Calendar.getInstance();
        System.out.println("Year:    " + c.get(Calendar.YEAR));
        System.out.println("Month:   " + c.get(Calendar.MONTH));
        System.out.println("Day:     " + c.get(Calendar.DAY_OF_MONTH));
        System.out.println("WeekDay:" + c.get(Calendar.DAY_OF_WEEK));
        System.out.println("Hour:    " + c.get(Calendar.HOUR_OF_DAY));
        System.out.println("Minute:  " + c.get(Calendar.MINUTE));
        System.out.println("Second:  " + c.get(Calendar.SECOND));
        System.out.println("Millis:  " + c.get(Calendar.MILLISECOND));

        TimeZone timeZone = c.getTimeZone();
        System.out.println("TZ ID:   " + timeZone.getID());
        System.out.println("TZ Name:" + timeZone.getDisplayName());
        System.out.println("TZ Off:  " + timeZone.getRawOffset());
    }
}
```

15.4 Formatting and Parsing Dates

- `java.text.DateFormat` is an abstract class for date/time formatting subclasses which formats and parses dates or time in a language-independent manner.

  - The date/time formatting subclass handles formatting and parsing dates and times.

- DateFormat includes several static methods for obtaining default date/time formatters based on the default or a specified locale.
  - For example, to get a date/time formatter with the default formatting style for the default locale:

    ```
    DateFormat df = DateFormat.getDateTimeInstance();
    ```

  - To get a formatter for the long-style date in the French locale:

    ```
    DateFormat df = DateFormat.getDateInstance(
            DateFormat.LONG, Locale.FRANCE);
    ```

- For a given instance of a DateFormat object:
  - The `format(Date)` method generates a String representation of a Date object
  - The `parse(String)` method generates a Date object by parsing a date/time string
- `java.text.SimpleDateFormat` is a concrete subclass of DateFormat.
  - It allows you to define you own patterns for date/time formatting and parsing.

```
package com.marakana.demo;

import java.util.Date;
import java.text.*;

public class DateExample {

    public static void main(String[] args) {
        Date now = new Date();
        DateFormat df;

        // Get the default date and time format
        df = DateFormat.getDateTimeInstance();
        System.out.println(df.format(now));

        // Get the default "short" date and time format
        df = DateFormat.getInstance();
        System.out.println(df.format(now));

        // Get the default "medium" date format
        df = DateFormat.getDateInstance(DateFormat.MEDIUM);
        System.out.println(df.format(now));

        // Get the default "long" time format
        df = DateFormat.getTimeInstance(DateFormat.LONG);
        System.out.println(df.format(now));

        // An example of parsing a time and date string into a Date
        try {
            df = new SimpleDateFormat("MMM d, y");
            Date birthday = df.parse("August 29, 1965");
            df = DateFormat.getDateInstance(DateFormat.MEDIUM);
            System.out.println(df.format(birthday));
        } catch (ParseException e) {
            // Couldn't parse the date string
            e.printStackTrace();
        }
```

```
    }
}
```

15.5 Formatting and Parsing Dates

- `java.text.NumberFormat` is an abstract class for number formatting subclasses which formats and parses numbers in a language-independent manner.
  - The number formatting subclass handles formatting and parsing numbers.
  - Your code can be completely independent of the locale conventions for decimal points, thousands-separators, currency symbols, or even the particular decimal digits used.
- NumberFormat includes several static methods for obtaining default numbers formatters based on the default or a specified locale.
  - For example, to get a number formatter with the default formatting style for the default locale:

    ```
    NumberFormat nf = NumberFormat.getInstance();
    ```

  - To get a currency formatter for the Japanese locale:

    ```
    NumberFormat nf
        = NumberFormat.getCurrencyInstance(Locale.JAPAN);
    ```

- For a given instance of a NumberFormat object:
  - The `format()` method generates a String representation of a number
  - The `parse()` method generates a Number object by parsing a numeric-formatted string

---

```
package com.marakana.demo;

import java.text.*;
import java.util.Locale;

public class FormatNumbers {

    public static void main(String[] args) {
        // Print out a number using the localized number, integer, currency,
         // and percent format for each locale
         Locale[] locales = NumberFormat.getAvailableLocales();
         double myNumber = -123456.78;
         NumberFormat form;
         for (int j=0; j<4; ++j) {
             System.out.println("FORMAT");
             for (Locale l: locales) {
                 if (l.getCountry().length() == 0) {
                    continue; // Skip language-only locales
                 }
                 System.out.print(l.getDisplayName());
                 switch (j) {
                 case 0:
                     form = NumberFormat.getInstance(l); break;
```

```
                case 1:
                    form = NumberFormat.getIntegerInstance(l); break;
                case 2:
                    form = NumberFormat.getCurrencyInstance(l); break;
                default:
                    form = NumberFormat.getPercentInstance(l); break;
                }
                if (form instanceof DecimalFormat) {
                    System.out.print(":\t" + ((DecimalFormat) form).toPattern());
                }
                System.out.print(" ->\t" + form.format(myNumber));
                try {
                    System.out.println(" ->\t" + form.parse(form.format(myNumber)));
                } catch (ParseException e) {}
            }
        }
    }

}
```

15.6 Typesafe Enums

- Prior to Java 5, it was common to use `public final static int`s for enumerated values (*enums*). But there are drawbacks:
  - Not typesafe — It's just an `int`, and nothing prevents you from passing in another `int` where an enum was expected.
  - No namespace — You need to invent prefixes to avoid name collisions with other `int` enum types.
  - Brittle — The `int` constants are compiled into client code, so the behavior is undefined if the enum order changes or new constants are added.
  - Uninformative printed values — Because they are just `int`s, if you print one out all you get is a number.
  - No easy way to guarantee iteration over all enum values.

```
public class ChessPiece {
    public static final int COLOR_WHITE = 0;
    public static final int COLOR_BLACK = 1;
    public static final int FIGURE_KING = 0;
    public static final int FIGURE_QUEEN = 1;
    public static final int FIGURE_ROOK = 2;
    public static final int FIGURE_BISHOP = 3;
    public static final int FIGURE_KNIGHT = 4;
    public static final int FIGURE_PAWN = 5;

    private final int color;
    private final int figure;

    public ChessPiece(int color, int figure) {
        this.color = color;
        this.figure = figure;
    }

    public int getColor() { return this.color; }
    public int getFigure() { return this.figure; }

    public String toString() {
```

```
        /* switch statements to convert int color and figure to string */
    }
}
```

But the following invalid initialization still compiles!

```
new ChessPiece(ChessPiece.FIGURE_QUEEN, ChessPiece.FIGURE_KING);
```

You can even combine `int` enums in unnatural ways:

```
int combined = ChessPiece.COLOR_BLACK + ChessPiece.FIGURE_PAWN;
```

15.7 Typesafe Enums (cont.)

- Java 5 introduced the `enum` keyword for defining typesafe enumerated values. For example:

```
enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY }
```

- Java 5 enums are full-fledged classes, providing many benefits:
  - They are typesafe, with compile-time error checking.
  - The declared `enum` provides a natural namespace.
  - Enum definitions are loaded at runtime, so enums can evolve and change without breaking existing code.
  - You can `switch` on `enum` values.
  - They can have fields and methods to add state and behavior.
  - They have a default `toString()` that returns the enum value as a String — but you can override it if you want!
  - The enum type has a default `values()` method that returns an array of all the enum values. For example:

    ```
    for (Day d: Day.values()) System.out.println(d);
    ```

  - They can implement interfaces
  - They automatically implement the Comparable and Serializable interfaces.
  - They can be used in collections as objects more efficiently than `int` values — no boxing/unboxing required!

Tip
Use `enum`s any time you need a fixed set of constants. For example: choices on a menu, rounding modes, command line flags, etc.

```
public class ChessPiece {
  public static enum Figure {KING, QUEEN, ROOK, BISHOP, KNIGHT, PAWN};
  public static enum Color { WHITE, BLACK };

  private final Color color;
  private final Figure figure;

  public ChessPiece(Color color, Figure figure) {
    this.color = color;
    this.figure = figure;
  }
```

```
  public Figure getFigure() {
    return this.figure;
  }

  public Color getColor() {
    return this.color;
  }

  public String toString() {
    return this.color + " " + this.figure;
  }
}
```

15.8 EnumSet and EnumMap

- The `java.util` package contains special-purpose Set and Map implementations called EnumSet and EnumMap.
- EnumSet is a high-performance Set implementation for enums.
  - All of the members of an EnumSet must be of the same enum type.
  - EnumSets support iteration over ranges of enum types. For example:

```
for (Day d : EnumSet.range(Day.MONDAY, Day.FRIDAY))
    System.out.println(d);
```

  - EnumSets can replace traditional bit flags:

```
EnumSet.of(Style.BOLD, Style.ITALIC)
```

- EnumMap is a high-performance Map implementation for use with enum keys.
  - Enum maps combine the richness and safety of the Map interface with speed approaching that of an array.
  - If you want to map an enum to a value, you should always use an EnumMap in preference to an array.

15.9 Annotations

- Java 5 introduced *annotations* to the language, which provide metadata about a program that is not part of the program itself.
  - They have no direct effect on the operation of the code they annotate.
- Annotations have a number of uses, including:

Information for the compiler
: Annotations can be used by the compiler to detect errors or suppress warnings.

Compile-time and deployment-time processing
: Software tools can process annotation information to generate code, XML files, etc.

Runtime processing
: Libraries and frameworks can use annotations conditionally process classes and methods, and to *inject* objects into annotated classes and methods.

The Java annotation facility consists of:

- Syntax for declaring annotation types

- Syntax for annotating declarations
- APIs for reading annotations reflectively
- Class file representation for annotations
- Annotation Processing Tool (apt)

15.10 Using Annotations

- Annotations can be applied to a program's declarations of classes, fields, methods, and other program elements.
- The annotation appears first, usually on its own line, and may include *elements* with named or unnamed values:

```
@Unfinished(
   owner = "Zaphod Beeblebrox",
   date = "5/11/2001"
)
class HeartOfGold { }

@SuppressWarnings(value = "unchecked")
void myMethod() { }
```

- If there is just one element named "value", then the name may be omitted:

```
@SuppressWarnings("unchecked")
void myMethod() { }
```

- If an annotation has no elements, the parentheses may be omitted:

```
@Override
void myMethod() { }
```

Annotation elements must be compile-time constant values.

Annotation elements may be primitive types, Strings, Classes, enums, annotations, or an array of a permitted type.

An array of values is supplied within braces, as in:

```
@SuppressWarnings({"unchecked","deprecation"})
```

15.11 Standard Java Annotations

The `java.lang` package defines three standard annotations:

`@Override`
: Indicates that a method declaration is intended to override a method declaration in a base class. Compilers generate an error message if the method does not actually override a base class method (e.g., misspelled, wrong signature, etc.).

`@Deprecated`
: Marks a feature that programmers should not use (e.g., replaced by a newer method, dangerous, etc.). Compilers generate a warning when code uses a deprecated feature.

`@SuppressWarnings`
Suppresses a compiler warning on the annotated element (and those contained within the element). You must provide a `value` element with a single String or an array of String values indicating the warnings to suppress. Supported values include:

- `all` — all warnings
- `deprecation` — warnings relative to deprecation
- `fallthrough` — warnings relative to missing breaks in switch statements
- `hiding` — warnings relative to locals that hide variable
- `serial` — warnings relative to missing serialVersionUID field for a serializable class
- `unchecked` — warnings relative to unchecked operations
- `unused` — warnings relative to unused code

Chapter 16

Java Native Interface (JNI)

16.1 JNI Overview

- An interface that allows Java to interact with code written in another language
- Motivation for JNI
  - Code reusability
    - Reuse existing/legacy code with Java (mostly C/C++)
  - Performance
    - Native code used to be up to 20 times faster than Java, when running in interpreted mode
    - Modern JIT compilers (HotSpot) make this a moot point
  - Allow Java to tap into low level O/S, H/W routines
- JNI code is not portable!

Note
JNI can also be used to invoke Java code from within natively-written applications - such as those written in C/C++.
In fact, the java command-line utility is an example of one such application, that launches Java code in a Java Virtual Machine.

16.2 JNI Components

- `javah` - JDK tool that builds C-style header files from a given Java class that includes `native` methods
  - Adapts Java method signatures to native function prototypes
- `jni.h` - C/C++ header file included with the JDK that maps Java types to their native counterparts
  - `javah` automatically includes this file in the application header files

16.3 JNI Development (Java)

- Create a Java class with native method(s): `public native void sayHi(String who, int times);`
- Load the library which implements the method: `System.loadLibrary("HelloImpl");`
- Invoke the native method from Java

For example, our Java code could look like this:

```
package com.marakana.jniexamples;

public class Hello {
  public native void sayHi(String who, int times); //❶

  static { System.loadLibrary("HelloImpl"); } //❷

  public static void main (String[] args) {
    Hello hello = new Hello();
    hello.sayHi(args[0], Integer.parseInt(args[1])); //❸
  }
}
```

❶, ❸ The method `sayHi` will be implemented in C/C++ in separate file(s), which will be compiled into a library.

❷ The library filename will be called `libHelloImpl.so` (on Unix), `HelloImpl.dll` (on Windows) and `libHelloImpl.j` (Mac OSX), but when loaded in Java, the library has to be loaded as `HelloImpl`.

16.4 JNI Development (C)

- We use the JDK `javah` utility to generate the header file `package_name_classname.h` with a function prototype for the `sayHi` method:
  `javac -d ./classes/ ./src/com/marakana/jniexamples/Hello.java`
  Then in the classes directory run: `javah -jni com.marakana.jniexamples.Hello` to generate the header file `com_marakana_jniexamples_Hello.h`
- We then create `com_marakana_jniexamples_Hello.c` to implement the `Java_com_marakana_jniexamples_Hello` function

The file `com_marakana_jniexamples_Hello.h` looks like:

```
...
#include <jni.h>
...
JNIEXPORT void JNICALL Java_com_marakana_jniexamples_Hello_sayHi
  (JNIEnv *, jobject, jstring, jint);
...
```

The file Hello.c looks like:

```
#include <stdio.h>
#include "com_marakana_jniexamples_Hello.h"

JNIEXPORT void JNICALL Java_com_marakana_jniexamples_Hello_sayHi(JNIEnv *env, jobject obj,  ↩
    jstring who, jint times) {
  jint i;
  jboolean iscopy;
  const char *name;
  name = (*env)->GetStringUTFChars(env, who, &iscopy);
  for (i = 0; i < times; i++) {
    printf("Hello %s\n", name);
  }
}
```

16.5 JNI Development (Compile)

- We are now ready to compile our program and run it
  - The compilation is system-dependent
- This will create `libHelloImpl.so`, `HelloImpl.dll`, `libHelloImpl.jnilib` (depending on the O/S)
- Set `LD_LIBRARY_PATH` to point to the directory where the compiled library is stored
- Run your Java application

For example, to compile `com_marakana_jniexamples_Hello.c` in the "classes" directory (if your `.h` file and `.c` file are there) on Linux do:

```
gcc -o libHelloImpl.so -lc -shared \
    -I/usr/local/jdk1.6.0_03/include \
    -I/usr/local/jdk1.6.0_03/include/linux com_marakana_jniexamples_Hello.c
```

On Mac OSX :

```
gcc -o libHelloImpl.jnilib -lc -shared \
    -I/System/Library/Frameworks/JavaVM.framework/Headers com_marakana_jniexamples_Hello.c
```

Then set the `LD_LIBRARY_PATH` to the current working directory:

```
export LD_LIBRARY_PATH=.
```

Finally, run your application in the directory where your compiled classes are stored ("classes" for example):

```
java com.marakana.jniexamples.Hello Student 5
Hello Student
Hello Student
Hello Student
Hello Student
Hello Student
```

Note
Common mistakes resulting in `java.lang.UnsatisfiedLinkError` usually come from incorrect naming of the shared library (O/S-dependent), the library not being in the search path, or wrong library being loaded by Java code.

16.6 Type Conversion

- In many cases, programmers need to pass arguments to native methods and they do also want to receive results from native method calls
- Two kind of types in Java:
  - Primitive types such as `int`, `float`, `char`, etc
  - Reference types such as classes, instances, arrays and strings (instances of `java.lang.String` class)
- However, primitive and reference types are treated differently in JNI
  - Mapping for primitive types in JNI is simple

Table 16.1: JNI data type mapping in variables:

Java Language Type	Native Type	Description
boolean	jboolean	8 bits, unsigned
byte	jbyte	8 bits, signed
char	jchar	16 bits, unsigned
double	jdouble	64 bits
float	jfloat	32 bits
int	jint	32 bits, signed
long	jlong	64 bits, signed
short	jshort	16 bits, signed
void	void	N/A

  - Mapping for objects is more complex. Here we will focus only on strings and arrays but before we dig into that let us talk about the native methods arguments

---

- JNI passes objects to native methods as opaque references
- Opaque references are C pointer types that refer to internal data structures in the JVM
- Let us consider the following Java class:

```
package com.marakana.jniexamples;

public class HelloName {
  public static native void sayHelloName(String name);

  static { System.loadLibrary("helloname"); }

  public static void main (String[] args) {
    HelloName hello = new HelloName();
    String name = "John";
    hello.sayHelloName(name);
  }
}
```

- The `.h` file would look like this:

```
...
#include <jni.h>
```

```
...
JNIEXPORT void JNICALL Java_com_marakana_jniexamples_HelloName_sayHelloName
  (JNIEnv *, jclass, jstring);
...
```

- A `.c` file like this one would not produce the expected result:

```
#include <stdio.h>
#include "com_marakana_jniexamples_HelloName.h"


JNIEXPORT void JNICALL Java_com_marakana_jniexamples_HelloName_sayHelloName(JNIEnv *env,  ↩
    jclass class, jstring name){
    printf("Hello %s", name);
}
```

16.7 Native Method Arguments

- All native method implementation accepts two standard parameters:
  - `JNIEnv *env`: Is a pointer that points to another pointer pointing to a function table (array of pointer). Each entry in this function table points to a JNI function. These are the functions we are going to use for type conversion
  - The second argument is different depending on whether the native method is a static method or an instance method
    - Instance method: It will be a `jobject` argument which is a reference to the object on which the method is invoked
    - Static method: It will be a `jclass` argument which is a reference to the class in which the method is define

16.8 String Conversion

- We just talked about the `JNIEnv *env` that will be the argument to use where we will find the type conversion methods
- There a lot of methods related to strings:
  - Some are to convert `java.lang.String` to `C` string: `GetStringChars` (Unicode format), `GetStringUTFChars` (UTF-8 format)
  - Some are to convert `java.lang.String` to `C` string: `NewString` (Unicode format), `NewStringUTF` (UTF-8 format)
  - Some are to release memory on C string: `ReleaseStringChars`, `ReleaseStringUTFChars`

Note
Details about these methods can be found at
http://download.oracle.com/javase/6/docs/technotes/guides/jni/spec/functions.html

---

- If you remember the previous example, we had a native method where we wanted to display "Hello *name*":

```
#include <stdio.h>
#include "com_marakana_jniexamples_HelloName.h"

JNIEXPORT void JNICALL Java_com_marakana_jniexamples_HelloName_sayHelloName(JNIEnv *env,  ↩
    jclass class, jstring name) {
    printf("Hello %s", name); //❶
}
```

❶ This example would not work since the `jstring` type represents strings in the Java virtual machine. This is different from the C string type (`char *`)

- Here is what you would do, using UTF-8 string for instance:

```
#include <stdio.h>
#include "com_marakana_jniexamples_HelloName.h"

JNIEXPORT void JNICALL Java_com_marakana_jniexamples_HelloName_sayHelloName(JNIEnv *env,  ↩
    jclass class, jstring name){
    const jbyte *str;
    str = (*env)->GetStringUTFChars(env, name, NULL); //❶
    printf("Hello %s\n", str);
    (*env)->ReleaseStringUTFChars(env, name, str); //❷
}
```

❶ This returns a pointer to an array of bytes representing the string in UTF-8 encoding (without making a copy)

❷ When we are not making a copy of the string, calling `ReleaseStringUTFChars` prevents the memory area used by the string to stay "pinned". If the data was copied, we need to call `ReleaseStringUTFChars` to free the memory which is not used anymore

- Here is another example where we would construct and return a `java.lang.String` instance:

```
#include <stdio.h>
#include "com_marakana_jniexamples_GetName.h"

JNIEXPORT jstring JNICALL Java_com_marakana_jniexamples_ReturnName_GetName(JNIEnv *env,  ↩
    jclass class) {
    char buffer[20];
    scanf("%s", buffer);
    return (*env)->NewStringUTF(env, buffer);
}
```

16.9 Array Conversion

- Here we are going to focus on primitive arrays only since they are different from objects arrays in JNI
- Arrays are represented in JNI by the `jarray` reference type and its "subtypes" such as `jintArray` ⇒ A `jarray` is not a C array!
- Again we will use the `JNIEnv *env` parameter to access the type conversion methods
  - `Get<Type>ArrayRegion`: Copies the contents of primitive arrays to a preallocated C buffer. Good to use when the size of the array is known
  - `Get<Type>ArrayElements`: Gets a pointer to the content of the primitive array
  - `New<Type>Array`: To create an array specifying a length

- We are going to see an example of how to read a Java primitive array in the native world
- First, this would be your Java program:

```
package com.marakana.jniexamples;

public class ArrayReader {
    private static native int sumArray(int[] arr); //❶
```

```
    public static void main(String[] args) {
        //Array declaration
        int arr[] = new int[10];
        //Fill the array
        for (int i = 0; i < 10; i++) {
            arr[i] = i;
        }
        ArrayReader reader = new ArrayReader();
        //Call native method
        int result = reader.sumArray(arr); //❷
        System.out.println("The sum of every element in the array is " + Integer.toString( ↩
            result));
    }
    static {
        System.loadLibrary("arrayreader");
    }
}
```

❶, ❷ This method will return the sum of each element in the array

- After running `javah`, create your `.c` file that would look like this:

```
#include <stdio.h>
#include "com_marakana_jniexamples_ArrayReader.h"

JNIEXPORT jint JNICALL Java_com_marakana_jniexamples_ArrayReader_sumArray(JNIEnv *env,  ↩
    jclass class, jintArray array) {
    jint *native_array;
    jint i, result = 0;
    native_array = (*env)->GetIntArrayElements(env, array, NULL); /* ❶ */
    if (native_array == NULL) {
        return 0;
    }
    for (i=0; i<10; i++) {
        result += native_array[i];
    }
    (*env)->ReleaseIntArrayElements(env, array, native_array, 0);
    return result;
}
```

❶ We could also have used `GetIntArrayRegion` since we exactly know the size of the array

16.10 Throwing Exceptions In The Native World

- We are about to see how to throw an exception from the native world
- Throwing an exception from the native world involves the following steps:
  - Find the exception class that you want to throw
  - Throw the exception
  - Delete the local reference to the exception class
- We could imagine a utility function like this one:

```
void ThrowExceptionByClassName(JNIEnv *env, const char *name, const char *message) {
  jclass class = (*env)->FindClass(env, name); //❶
  if (class != NULL) {
      (*env)->ThrowNew(env, class, message); //❷
```

```
        (*env)->DeleteLocalRef(env, class); //❸
    }
}
```

❶ Find exception class by its name

❷ Throw the exception using the class reference we got before and the message for the exception

❸ Delete local reference to the exception class

- Here would be how to use this utility method:

```
ThrowExceptionByClassName(env,"java/lang/IllegalArgumentException","This exception is ↩
    thrown from C code");
```

- While an exception is pending, we can only call the following JNI functions:
  - `DeleteGlobalRef`
  - `DeleteLocalRef`
  - `DeleteWeakGlobalRef`
  - `ExceptionCheck`
  - `ExceptionClear`
  - `ExceptionDescribe`
  - `ExceptionOccurred`
  - `MonitorExit`
  - `PopLocalFrame`
  - `PushLocalFrame`
  - `Release<PrimitiveType>ArrayElements`
  - `ReleasePrimitiveArrayCritical`
  - `ReleaseStringChars`
  - `ReleaseStringCritical`
  - `ReleaseStringUTFChars`
- Calling methods on Java objects (e.g. via `CallObjectMethod`) can fail with an exception, but since exceptions don't automatically abort our funciton call, we must explicitly check for existance of a pending exception with a `ExceptionCheck` - otherwise, the return value is undefined

16.11 Access Properties And Methods From Native Code

- You might want to modify some properties or call methods of the instance calling the native code
- It always starts with this operation: Getting a reference to the object class by calling the `GetObjectClass` method
- We are then going to get instance field id or an instance method id from the class reference using `GetFieldID` or `GetMethodID` methods
- For the rest, it differs depending on whether we are accessing a field or a method

- From this Java class, we will see how to call its methods or access its properties in the native code:

```
package com.marakana.jniexamples;

public class InstanceAccess {
    public String name; //❶

    public void setName(String name) { //❷
        this.name = name;
    }

    //Native method
    public native void propertyAccess(); //❸
    public native void methodAccess(); //❹

    public static void main(String args[]) {
        InstanceAccess instanceAccessor = new InstanceAccess();
        //Set the initial value of the name property
        instanceAccessor.setName("Jack");
        System.out.println("Java: value of name = \""+ instanceAccessor.name +"\"");
        //Call the propetyAccess() method
        System.out.println("Java: calling propertyAccess() method...");
        instanceAccessor.propertyAccess(); //❺
        //Value of name after calling the propertyAccess() method
        System.out.println("Java: value of name after calling propertyAccess() = \""+  ↩
            instanceAccessor.name +"\"");
        //Call the methodAccess() method
        System.out.println("Java: calling methodAccess() method...");
        instanceAccessor.methodAccess(); //❻
        System.out.println("Java: value of name after calling methodAccess() = \""+  ↩
            instanceAccessor.name +"\"");
    }

    //Load library
    static {
        System.loadLibrary("instanceaccess");
    }
}
```

❶ Name property that we are going to modify along this code execution

❷ This method will be called by the native code to modify the name property

❸, ❺ This native method modifies the name property by directly accessing the property

❹, ❻ This native method modifies the name property by calling the Java method `setName()`

- This would be our C code for native execution:

```
#include <stdio.h>
#include "com_marakana_jniexamples_InstanceAccess.h"


JNIEXPORT void JNICALL Java_com_marakana_jniexamples_InstanceAccess_propertyAccess(JNIEnv  ↩
    *env, jobject object){
    jfieldID fieldId;
    jstring jstr;
    const char *cString;

    /* Getting a reference to object class */
    jclass class = (*env)->GetObjectClass(env, object); /* ❶ */

    /* Getting the field id in the class */
    fieldId = (*env)->GetFieldID(env, class, "name", "Ljava/lang/String;"); /* ❷ */
    if (fieldId == NULL) {
```

```
        return; /* Error while getting field id */
    }

    /* Getting a jstring */
    jstr = (*env)->GetObjectField(env, object, fieldId); /* ❸ */

    /* From that jstring we are getting a C string: char* */
    cString = (*env)->GetStringUTFChars(env, jstr, NULL); /* ❹ */
    if (cString == NULL) {
        return; /* Out of memory */
    }
    printf("C: value of name before property modification = \"%s\"\n", cString);
    (*env)->ReleaseStringUTFChars(env, jstr, cString);

    /* Creating a new string containing the new name */
    jstr = (*env)->NewStringUTF(env, "Brian"); /* ❺ */
    if (jstr == NULL) {
        return; /* Out of memory */
    }
    /* Overwrite the value of the name property */
    (*env)->SetObjectField(env, object, fieldId, jstr); /* ❻ */
}


JNIEXPORT void JNICALL Java_com_marakana_jniexamples_InstanceAccess_methodAccess(JNIEnv * ↩
    env, jobject object){
    jclass class = (*env)->GetObjectClass(env, object); /* ❼ */
    jmethodID methodId = (*env)->GetMethodID(env, class, "setName", "(Ljava/lang/String;)V ↩
        "); /* ❽ */
    jstring jstr;
    if (methodId == NULL) {
        return; /* method not found */
    }
    /* Creating a new string containing the new name */
    jstr = (*env)->NewStringUTF(env, "Nick"); /* ❾ */
    (*env)->CallVoidMethod(env, object, methodId, jstr); /* ❿ */
}
```

❶, ❼ This is getting a reference to the object class

❷ Gets a field Id from the object class, specifying the property to get and the internal type. you can find information on the jni type there: http://download.oracle.com/javase/6/docs/technotes/guides/jni/spec/types.html

❸ This will return the value of the property in the native type: here a jstring

❹ We need to convert the jstring to a C string

❺ This creates a new java.lang.String that is going be use to change the value of the property

❻ This sets the property to its new value

❽ Gets a method id from the object class previously obtained, specifying the name of the method along with its signature. There is a very useful java tool that you can use to get the signature of a method: `javap -s -p ClassName` for instance `javap -s -p InstanceAccess`

❾ This creates a new java.lang.String that we are going to use as an argument when calling the java method from native code

❿ Calling `CallVoidMethod` since the Java method return type is `void` and we are passing the previously created `jstring` as a parameter

Chapter 17

`java.sql`

Java Database Connectivity API (JDBC)

17.1 JDBC Overview

- Industry standard for database-independent connectivity between Java applications and wide range of relational databases (RDBMS)
- Technology split into:
  - API: Set of interfaces independent of the RDBMS
  - Driver: RDBMS-specific implementation of API interfaces (e.g. Oracle, DB2, MySQL, etc.)
- Even though JDBC is RDBMS independent, SQL dialects (syntax) are generally not

---

Just like Java aims for "write once, run anywhere", JDBC strives for "write once, run with any database".

17.2 JDBC Drivers

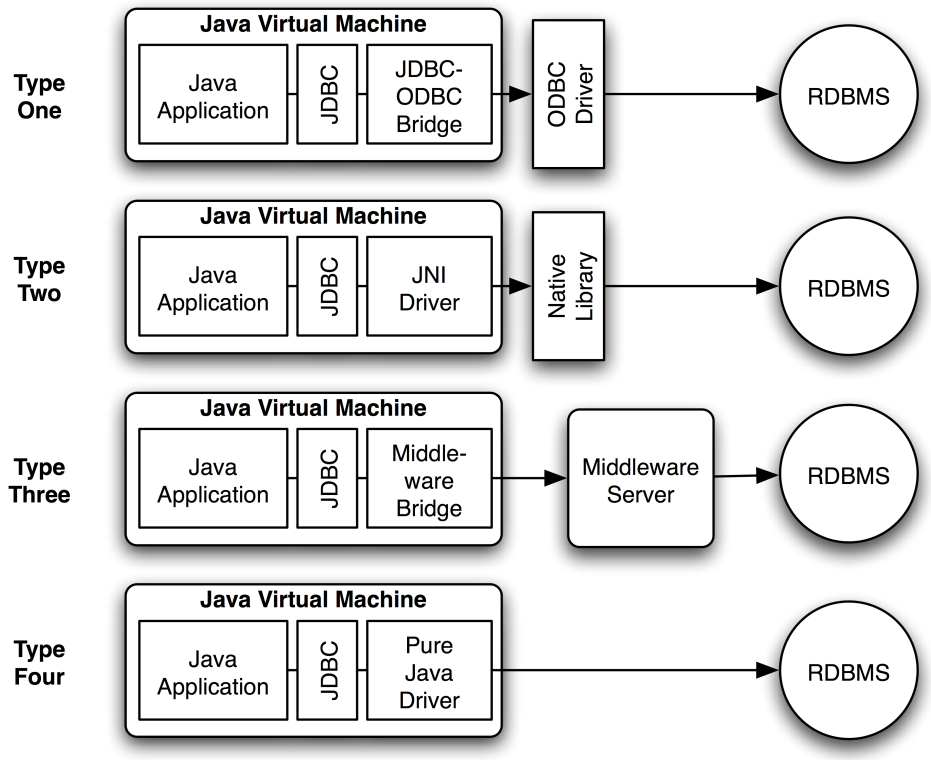


Figure 17.1: JDBC Drivers

- **Type One**
  - JDBC-ODBC Bridge plus ODBC driver
  - Depends on support for ODBC
  - Not portable
- **Type Two**
  - JNI/Native Java driver
  - Requires DB native library
  - Not portable
- **Type Three**
  - Pure Java to Middleware driver
  - Depends on Middleware server
  - Driver is portable, but the middleware server might not be
- **Type Four**
  - Pure Java Driver
  - Talks directly to the RDBMS
  - Portable

17.3 Getting a JDBC `Connection`

- Make sure JDBC driver is in `CLASSPATH`
- Register JDBC driver
  - `Class.forName("my.sql.Driver");`
  - Run with `-Djdbc.drivers=my.sql.Driver`
  - `System.setProperty("jdbc.drivers", "my.sql.Driver");`
- Construct Driver URL
  - `jdbc:sub-protocol://database-locator`
- `Connection connection = DriverManager.getConnection(url);`

For example, to run your application against a MySQL RDBMS, you would run JVM with (all on one line):

+

```
java -classpath .:mysql-connector-java-5.1.5-bin.jar -Djdbc.drivers=com.mysql.jdbc.Driver ↩
    my.MainClass args
```

The driver URL would be set to something like:

```
String url = "jdbc:mysql://server:3306/db?user=me&password=mysecret";
```

Finally, to get a JDBC Connection, do:

```
Connection con = DriverManager.getConnection(url);
```

Note that you can also pass the username and password at run time:

```
Connection con = DriverManager.getConnection("jdbc:mysql://server:3306/db", "me", "mysecret ↩
    ");
```

In a J2EE application server, JDBC connections can also be obtained using a `javax.sql.DataSource` obtained from a JNDI `javax.naming.Context`:

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup(”java:comp/env/jdbc/MyDS");
Connection con = ds.getConnection();
```

17.4 Preparing a `Statement`

- `java.sql.Statement`
  - Used for executing static SQL statement and returning the results it produces
  - Only one result per statement at any given time
- `java.sql.PreparedStatement`
  - Precompiled SQL statement
  - Efficiently executed multiple times
  - Supports insertion + conversion + escaping of Java parameters into the statement

- Release associated resources with `close()`

```
Connection con = DriverManager.getConnection(url);
try {
  Statement stmt = con.createStatement();
  try {
    ResultSet result = stmt.executeQuery("SELECT * FROM Customers");
    try {
      // process result set
    } finally {
      result.close();
    }
  } finally {
    stmt.close();
  }
} finally {
  con.close();
}
Connection con = DriverManager.getConnection(url);
try {
  PreparedStatement stmt = con.prepareStatement(
      "UPDATE Employees SET ExtraVactaion = ? WHERE SSN = ?");
  try {
    stmt.setInt(1, 10); // updates the first ?
    stmt.setString(2, "555-12-1234"); // updates the second ?
    stmt.executeUpdate();
  } finally {
    stmt.close();
  }
} finally {
  con.close();
}
```

17.5 Processing `ResultSet`

- A table of data representing database result in response to a `executeQuery()`
- Maintains a cursor pointing to the current row of data
  - Call `boolean next()` advance the cursor
- Can be made scrollable and/or updatable
- Provides getter methods for reading the current row (by column name or index)
  - Type sensitive: `getString`, `getInt`, `getDate`
  - Index starts at 1; names are case insensitive

```
Connection con = DriverManager.getConnection(url);
try {
  Statement stmt = con.createStatement();
  try {
    ResultSet resultSet = stmt.executeQuery(
```

```
      "SELECT CustomerID, CompanyName, ContactName FROM Customers");
    try {
      // for all rows
      while (resultSet.next()) {
        // print the columns of the current row (indexed by name)
        System.out.print(resultSet.getString("CustomerID"));
        System.out.print(", ");
        System.out.print(resultSet.getString("CompanyName"));
        System.out.print(", ");
        System.out.print(resultSet.getString("ContactName"));
        System.out.println();
      }
    } finally {
      resultSet.close();
    }
  } finally {
    stmt.close();
  }
} finally {
  con.close();
}
```

From JavaDoc: In general, using the column index is more efficient. For maximum portability, result set columns within each row should be read in left-to-right order, and each column should be read only once.

17.6 Using `ResultSetMetaData`

- Used to get information about the types and properties of the columns in a `ResultSet` object:
  - `getColumnCount`
  - `getColumnDisplaySize`
  - `getColumnLabel`, `getColumnName`
  - `getColumnType`, `getColumnTypeName`
  - `isAutoIncrement`, `isCaseSensitive`, `isNullable`, `isReadOnly`, `isSearchable`, `isSigned`, `isWritable`

```
String customerID = "someid";
Connection con = DriverManager.getConnection(url);
try {
  // prepare the statement
  PreparedStatement stmt = con.prepareStatement(
    "SELECT * FROM Customers WHERE CustomerID=?");
  try {
    stmt.setString(1, customerID);
    ResultSet resultSet = stmt.executeQuery();
    try {
      if (resultSet.next()) {
        ResultSetMetaData metaData = resultSet.getMetaData();
        for (int c = 1; c <= metaData.getColumnCount(); c++) {
          System.out.print(metaData.getColumnName(c));
          System.out.print(": ");
          System.out.println(resultSet.getObject(c));
        }
      } else { // customer not found
        System.err.println(customerID + " not found");
      }
```

```
    } finally {
      resultSet.close(); // release resultSet
    }
  } finally {
    stmt.close(); // release the statement
  }
} finally {
  con.close(); // release the connection
}
```

17.7 Updates

- Update actions (`INSERT`, `UPDATE`, `DELETE`) are triggered through

```
int result = stmt.executeUpdate();
```

- The integer result is set to:
  - the row count returned by `INSERT`, `UPDATE`, `DELETE` statements
  - 0 for SQL statements that return nothing

```
Connection con = DriverManager.getConnection(url);
try {
  PreparedStatement stmt = con.prepareStatement(
    "INSERT INTO Employee VALUES (?, ?, ?, ?, ?)");
  try {
    stmt.setString(1, "John");
    stmt.setString(2, "555-12-345");
    stmt.setString(3, "john@company.com");
    stmt.setInt(4, 1974);
    stmt.setInt(5, 7);
    int result = stmt.executeUpdate();
    if (result == 0) {
      System.out.println("Employee not added.");
    } else {
      System.out.println("Employee added.");
    }
  } finally {
    stmt.close(); // release the statement
  }
} finally {
  con.close(); // release the connection
}
```

Chapter 18

XML Processing

Parsing XML with Java

18.1 Parsing XML with Java

- XML parsers for Java developers
  - Core XML functionality spread across packages: `javax.xml`, `org.w3c.dom`, and `org.xml.sax`
  - IBM's XML for Java EA2
  - NanoXML – a lightweight XML parser
  - JDOM - simplifies XML generation and parsing
- Toolkits and libraries for XML processing fall into two categories
  - Event-driven processors
  - Object model construction processors

18.2 Event-Driven Approach

- A parser reads the data and notifies specialized handlers (call-backs) that undertake some actions when different parts of the XML document are encountered:
  - Start/end of document
  - Start/end of element
  - Text data
  - Processing instruction, comment, etc.
- SAX (**S**imple **A**PI for **X**ML) – a standard event-driven XML API

For example, consider the following XML document:

```
<?xml version="1.0"?>
<dvd>
  <title>
      Matrix, The
  </title>
</dvd>
```

- The following events are triggered:
  1. Start document
  2. Start element (dvd)
  3. Characters (line feed and spaces)
  4. Start element (title)
  5. Characters (Matrix, The)
  6. End element (title)
  7. Characters (line feed)
  8. End element (dvd)
  9. End document

18.3 Overview of SAX

- Initially developed for Java
- XML is processed sequentially (event-based)
  - Small memory foot-print
  - Potentially more work for the developer who may need to keep track of the document processing state
- SAX is read-only and it cannot be used to change a document
- DOM parsers may use SAX in order to build internal models
- SAX events are handled by a special object called a content handler that developers implement

---

The way processors notify applications about elements, attributes, character data, processing instructions and entities is parser-specific and can greatly influence the programming style of the XML-related modules.

SAX is important because of the standardization of interfaces and classes that are used during the parsing process.

```
javax.xml.parsers.SAXParserFactory
javax.xml.parsers.SAXParser
org.xml.sax.ContentHandler
org.xml.sax.DTDHandler
org.xml.sax.EntityResolver
org.xml.sax.ErrorHandler
org.xml.sax.helpers.DefaultHandler
org.xml.sax.Attributes
org.xml.sax.SAXException
```

18.4 Parsing XML with SAX

- Create a SAX factory:

```
SAXParserFactory factory = SAXParserFactory.newInstance();
```

- Create a SAX parser:

```
SAXParser parser = factory.newSAXParser();
```

+ * Parse input (`File`, `InputStream`, etc) through a custom instance of a `DefaultHanlder` object:

```
parser.parse(in, myDvdHandler);
```

```
public class MyDvdHandler extends DefaultHandler {
  private Dvd dvd = null;
  private StringBuilder currentText;
  public Dvd getDvd() { return this.dvd; }

  @Override
  public void startElement(String uri, String localName,
    String qName, Attributes attributes) throws SAXException {
    if (qName.equals("dvd")) { this.dvd = new DVD(); }
    this.currentText = null;
  }

  @Override
  public void characters(char[] ch, int start, int length)
    throws SAXException {
    this.currentText.append(ch, start, length);
  }

  @Override
  public void endElement(String uri, String localName,
    String qName) throws SAXException {
    if (qName.equals("title")) {
      this.dvd.setTitle(this.currentText.toString().trim());
    }
  }
}
```

18.5 Object-Model Approach

- Based on the idea of parsing the whole document and constructing an object representation of it in memory
  - A series of parent-child nodes of different types
- DOM (**D**ocument **O**bject **M**odel)
  - Standard, language-independent specification written in OMG's IDL for the constructed object model
- Also known as the tree-based approach

For example, consider the following XML document:

```
<?xml version="1.0"?>
<dvd>
  <title>
      Matrix, The
  </title>
</dvd>
```

The following DOM object tree is constructed:

```
Document
  Element Node "dvd"
    Text Node (white-space)
    Element Node "title"
      Text Node "Matrix, The" (trimmed white-space)
    Text Node (white-space)
```

18.6 Overview of DOM

- DOM defines a series of interfaces for working with XML documents
  - DOM is an object model not a data model, with corresponding actions
  - DOM can be updated in memory (unlike SAX)
- Application manipulates DOM after it is constructed, as opposed to handle XML events during the parsing time
- Slower than SAX (uses more memory) but easier to use (random access to XML data)

- DOM is a language and platform independent interface that allows programs to dynamically access and update the content structure and style of documents
- DOM parser implementations are free to choose whatever internal representation they like, as long as they comply with the DOM interfaces
- Compared to SAX, there is basically a trade-off between memory consumption and fast multiple data accesses after parsing
- DOM factory and parser are defined in `javax.xml.parsers` package as `DocumentBuilderFactory` and `DocumentBuilde`
- DOM interfaces are defined in `org.w3c.dom` package
- Fundamental core interfaces: `Node`, `Document`, `DocumentFragment`, `Element`, `DOMImplementation`, `NodeList`, `NamedNodeMap`, `CharacterData`, `DOMException`, `Attr`, `Text`, and `Comment`
- Extended interfaces: `CDATASection`, `DocumentType`, `EntityReference`, `ProcessingInstruction`

18.7 Parsing XML with DOM

- Create a DOM factory:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
```

- Create a DOM builder:

```
DocumentBuilder builder = factory.newDocumentBuilder();
```

- Parse Document from some XML input:

```
Document document = builder.parse(in); // use document to access parsed DOM
```

```
private Node findNodeByName(Node node, String name) {
  if (name.equals(node.getNodeName())) {
    return node;
  } else {
    for (Node n = node.getFirstChild(); n != null;
              n = n.getNextSibling()) {
      Node found = findNodeByName(n, name);
      if (found != null) {
        return found;
      }
    }
  }
  return null;
}

...
Node dvdNode = findNodeByName(document, "dvd");
if (dvdNode != null) {
  Dvd dvd = new Dvd();
  Node titleNode = findNodeByName(dvdNode, "title");
  if (titleNode != null) {
    dvd.setTitle(titleNode.getTextContent());
  }
}
...
```

Chapter 19

Design Patterns

Design Patterns in Java

19.1 What are Design Patterns?

- The core of solutions to common problems
- All patterns have: name, the problem they solve (in context), solution, consequences
- Types of patterns:
  - Creational - abstraction of instantiation process
  - Structural - composition of classes and objects in formation of larger structures
  - Behavioral - abstraction of algorithms and assignment of responsibility between objects

---

Why reinvent the wheel when you can follow a field-tested blueprint for solving your not-so-unique problem?

Many of the design patterns in use today and described here originate from the famous *Gang of Four* book: "Design Patterns: Elements of Reusable Object-Oriented Software" by Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley Publishing Co., 1995; ISBN: 0201633612)

19.2 Singleton

- Allow only one instance of a given class
  - The class holds a reference to its only instance
  - Lazy (on request) or static initialization (on load)
  - Private constructor to restrict external instantiation
- Provide global point of access to the instance

```
public static ClassName getInstance()
```

- Support multiple instances in the future if the requirements change
  - Update `getInstance` to return multiple (possibly cached) instances if needed

```
public class MySingleton {
  private static final MySingleton instance = new MySingleton();

  public static MySingleton getInstance() {
    return instance;
  }

  private MySingleton() {} // no client can do: new MySingleton()

  public void doX() {...}
  public void doY() {...}
  public String getA() {return ...;}
  public int getB() {return ...;}
}

public class Client {
  public void doSomething() {
    // doing something
    MySingleton.getInstance().doX();
    // doing something else
    int b = MySingleton.getInstance().getB();
    // do something with b
  }
}
```

19.3 Factory Method

- Define an interface for creating an object
- Defer the actual creation to subclasses
- Localizes the knowledge of concrete instances
- Use when clients cannot anticipate the class of objects to create

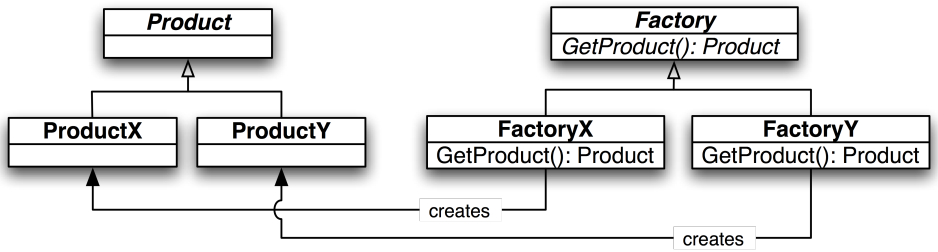


Figure 19.1: Factory Method Pattern

```
public interface Logger {
  public void error(String msg);
  public void debug(String msg);
}
public class ConsoleLogger implements Logger {
```

```
  public void error(String msg) {System.err.println("ERROR: "+ msg);}
  public void debug(String msg) {System.err.println("DEBUG: "+ msg);}
}
public class FileLogger implements Logger {
  private PrintStream out;
  private FileLogger(String file) throws IOException {
    this.out = new PrintStream(new FileOutputStream(file), true);
  }
  public void error(String msg) {out.println("ERROR: "+ msg);}
  public void debug(String msg) {out.println("DEBUG: "+ msg);}
}
public abstract class LoggerFactory {
  public abstract Logger getLogger();
  public static LoggerFactory getFactory(String f) throws Exception {
    return (LoggerFactory ) Class.forName(f).newInstance();
  }
}
public class ConsoleLoggerFactory extends LoggerFactory {
  public Logger getLogger() { return new ConsoleLogger(); }
}
public class FileLoggerFactory extends LoggerFactory { //ignoring expt
  private Logger log = new FileLogger(System.getProperty("log.file"));
  public Logger getLogger() { return log; }
}
```

19.4 Abstract Factory

- Interface for creating family of related objects
- Clients are independent of how the objects are created, composed, and represented
- System is configured with one of multiple families of products

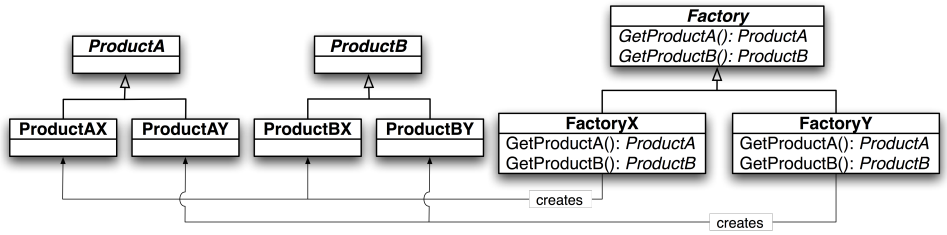


Figure 19.2: Abstract Factory Pattern

```
public interface Restaurant {
    public Appetizer getAppetizer();
    public Entree getEntree();
    public Dessert getDessert();
}

public interface Appetizer { public void eat(); }
public interface Entree { public void eat(); }
public interface Dessert { public void enjoy(); }
```

```
public class AmericanRestaurant implements Restaurant {
    public Appetizer getAppetizer() { return new Oysters(); }
    public Entree getEntree() { return new Steak(); }
    public Dessert getDessert() { return new CheeseCake(); }
}
public class ItalianRestaurant implements Restaurant {
    public Appetizer getAppetizer() { return new Pizzette(); }
    public Entree getEntree() { return new Pasta(); }
    public Dessert getDessert() { return new Gelato(); }
}

public class Oysters implements Appetizer {
    public void eat() {
        System.out.println("Eating Rocky Mountain Oysters");
    }
}

public class Gelato implements Dessert {
    public void enjoy() { System.out.println("Enjoying ice cream"); }
}
```

19.5 Adapter

- Convert the interface of a class into another interface clients expect
- Allow classes to work together despite their incompatible interfaces
- Typically used in development to *glue* components together, especially if 3rd party libraries are used
- Also known as *wrapper*

```
/**
 * Adapts Enumeration interface to Iterator interface.
 * Does not support remove() operation.
 */
public class EnumerationAsIterator implements Iterator {
  private final Enumeration enumeration;

  public EnumerationAsIterator(Enumeration enumeration) {
    this.enumeration = enumeration;
  }

  public boolean hasNext() {
    return this.enumeration.hasMoreElements();
  }

  public Object next() {
    return this.enumeration.nextElement();
  }

  /**
   * Not supported.
   * @throws UnsupportedOperationException if invoked
   */
  public void remove() {
    throw new UnsupportedOperationException("Cannot remove");
  }
}
```

19.6 Composite

- Compose objects into three structures to represent part-whole hierarchies
- Treat individual objects and compositions of objects uniformly

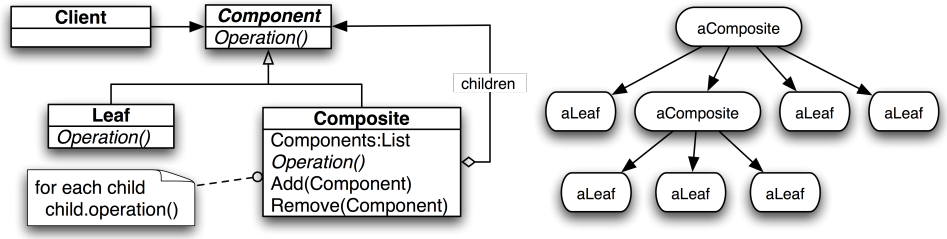


Figure 19.3: Composite Design Pattern

```
public interface Executable { // Component
  public void execute();
}

public class XExecutable implements Executable { // Leaf X
  public void execute() { /* do X */ }
}

public class YExecutable implements Executable { // Leaf Y
  public void execute() { /* do Y */ }
}

public class ExecutableCollection implements Executable { // Composite
  protected Collection executables = new LinkedList();

  public void addExecutable(Executable executable) {
    this.executables.add(executable);
  }

  public void removeExecutable(Executable executable) {
    this.executables.remove(executable);
  }

  public void execute() {
    for (Iterator i = this.executables.iterator(); i.hasNext();) {
      ((Executable) i.next()).execute();
    }
  }
}
```

19.7 Decorator

- Attach additional responsibilities to an object dynamically and transparently (run-time)
- An alternative to sub classing (compile-time)

- Remove responsibilities no longer needed

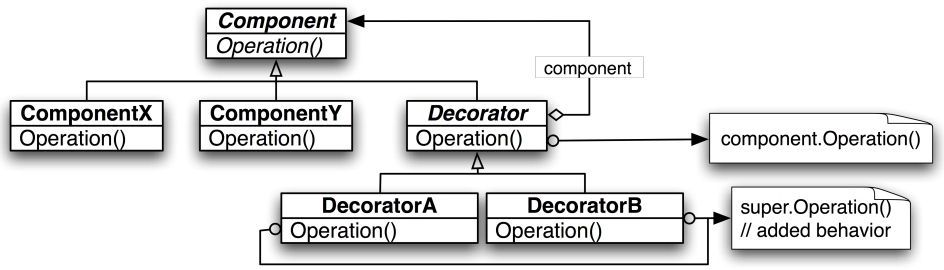


Figure 19.4: Decorator Design Pattern

A typical example of the decorator pattern can be found in `java.io`: `FilterOutputStream`, `FilterInputStream`, `FilterReader`, and `FilterWriter`.

For example, observe how we can decorate an `OutputStream`:

```
OutputStream out = new FileOutputStream("somefile");
out = new BufferedOutputStream(out); // buffer before writing
out = CipherOutputStream(out,        // encrypt all data
         Cipher.getInstance("DES/CBC/PKCS5Padding"));
out = new ZipOutputStream(out);      // compress all data
To write our own decorator, we could do:
public ByteCounterOutputStream extends FilterOutputStream {
  private int counter = 0;
  public ByteCounterOutputStream(OutputStream out) {super(out);}
  public void write(int b) throws IOException {
    super.write(b);
    this.counter++;
  }
  public int getCounter() {return this.counter;}
}
```

We could then wrap out with a byte counter decorator:

```
ByteCounterOutputStream bout = new ByteCounterOutputStream(out);
for (int b = in.read(); b != -1; b = in.read()) {
  bout.write(b);
}
System.out.println("Written "+ bout.getCounter() + " bytes");
```

19.8 Chain of Responsibility

- Decouple senders of requests from receivers
- Allow more than one object to attempt to handle a request
- Pass the request along a chain of receivers until it is handled
- Only one object handles the request
  - `Servlet Filter` and `FilterChain` bend this rule

- Some requests might not be handled

```
public abstract class MessageHandler {
  private MessageHandler next;
  public MessageHandler(MessageHandler next) {
    this.next = next;
  }
  public void handle(Message message) {
    if (this.next != null) {this.next.handle(message); }
  }
}

public class SpamHandler extends MessageHandler {
  public SpamHandler(MessageHandler next) {
    super(next);
  }
  public void handle(Message message) {
    if (isSpam(message)) {
      // handle spam
    } else {
      super.handle(message);
    }
  }
  ...
}
```

If a client was given an instance of a handler (created at run time):

```
MessageHandler handler = new BlackListHandler(new SpamHandler(new ForwardingHandler(new  ↩
    DeliveryHandler(null))));
```

The client would simply do:

```
handler.handle(emailMessage);
```

19.9 Observer / Publish-Subscribe

- Define a one-to-many dependency between objects
- When the observed object (subject) changes state, all dependents get notified and updated automatically
- `java.util.Observable`, `java.util.Observer`

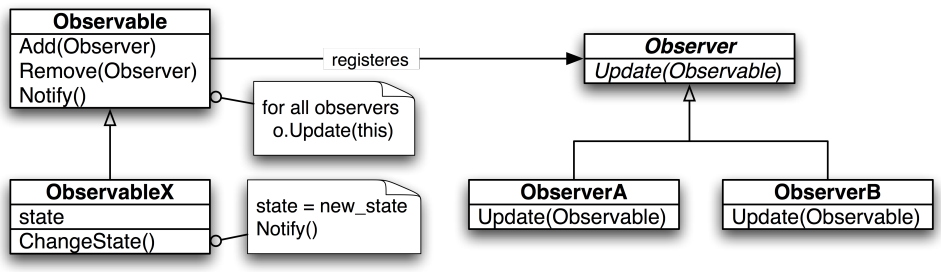


Figure 19.5: Observer Design Pattern

```
public class Employee extends Observable {
  ...
  public void setSalary(double amount) {
    this.salary = amount;
    super.setChanged();
    super.notifyObservers();
  }
}
public class Manager extends Employee implements Observer {
  public void update(Observable o) {
    if (o instanceof Employee) {
      // note employee's salary, vacation days, etc.
    }
  }
}
public class Department implements Observer {
  public void public void update(Observable o) {
   if (o instanceof Employee) {
     // deduct employees salary from department budget
   }
  }
}
public class Accounting implements Observer {
  public void public void update(Observable o) {
   if (o instanceof Employee) {
     // verify that monthly expenses are in line with the forcast
   }
  }
}
```

19.10 Strategy

- Defined family of interchangeable and encapsulated algorithms
- Change algorithms independently of clients that use them
- E.g: `java.util.Comparator`, `java.io.FileFilter`

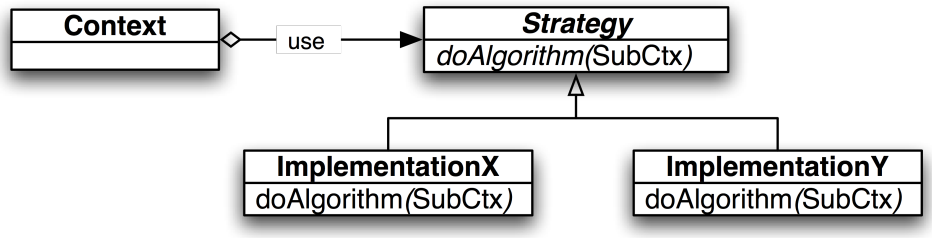


Figure 19.6: Strategy Design Pattern

```
public interface Comparator {
  public int compare(Object o1, Object o2);
}

public class DateComparator implements Comparator {
  public int compare(Object o1, Object o2) {
    return ((Date) o1).compareTo((Date) o2);
  }
}

public class StringIntegerComparator implements Comparator {
  public int compare(Object o1, Object o2) {
    return Integer.parseInt((String) o1) -
           Integer.parseInt((String) o2);
  }
}

public class ReverseComparator implements Comparator {
 private final Comparator c;
 public ReverseComparator(Comparator c) {this.c = c; }
 public int compare(Object o1, Object o2) {
   return c.compare(o2, o1);
 }
}
```

To sort integers represented as strings in descending order, you could do:

```
Arrays.sort(stringArray, new ReverseComparator(new StringIntegerComparator()));
```

The sort algorithm is independent of the comparison strategy.

19.11 Template

- Define a skeleton of an algorithm
- Defer some steps to subclasses
- Redefine other steps in subclasses
- Often a result of refactoring common code

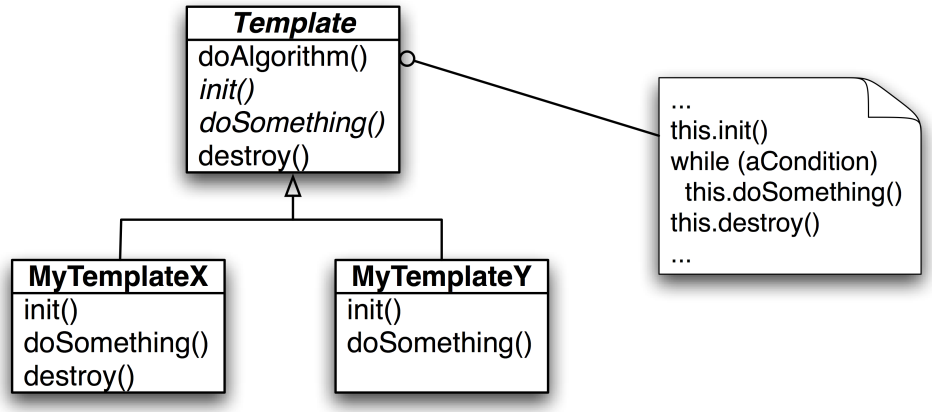


Figure 19.7: Template Design Pattern

```
public interface Calculator {
  public void calculate(double operand);
  public double getResult();
}

public abstract CalculatorTemplate implements Calculator() {
  private double result;
  private boolean initialized = false;
  public final void calculate(double operand) {
    if (this.initialized) {
      this.result = this.doCalculate(this.result, operand);
    } else {
      this.result = operand;
      this.initialized = true;
    }
  }
  public final double getResult() {
    return this.result; // throw exception if !initialized
  }
  protected abstract doCalculate(double o1, double o2);
}

public class AdditionCalculator extends CalculatorTemplate {
  protected doCalculate(double o1, double o1) {return o1 + o2; }
}

public class SubtractionCalculator extends CalculatorTemplate {
  protected doCalculate(double o1, double o1) {return o1 - o2; }
}
```

19.12 Data Access Object

- Abstracts and encapsulates all access to a data source
- Manages the connection to the data source to obtain and store data
- Makes the code independent of the data sources and data vendors (e.g. plain-text, xml, LDAP, MySQL, Oracle, DB2)

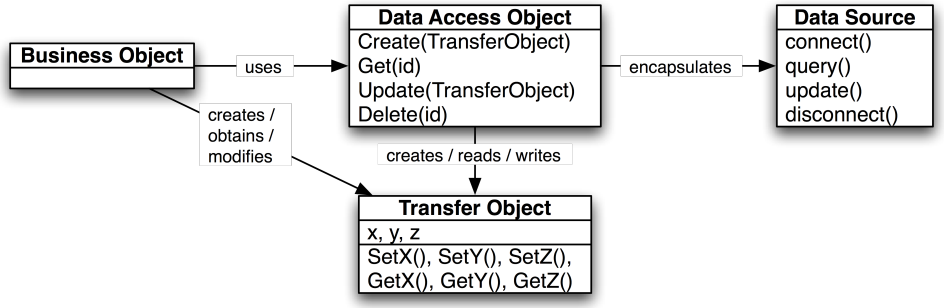


Figure 19.8: DAO Design Pattern

```
public class Customer {
  private final String id;
  private String contactName;
  private String phone;
  public void setId(String id) { this.id = id; }
  public String getId() { return this.id; }
  public void setContactName(String cn) { this.contactName = cn;}
  public String getContactName() { return this.contactName; }
  public void setPhone(String phone) { this.phone = phone; }
  public String getPhone() { return this.phone; }
}

public interface CustomerDAO {
  public void addCustomer(Customer c) throws DataAccessException;
  public Customer getCustomer(String id) throws DataAccessException;
  public List getCustomers() throws DataAccessException;
  public void removeCustomer(String id) throws DataAccessException;
  public void modifyCustomer(Customer c) throws DataAccessException;
}

public class MySqlCustomerDAO implements CustomerDAO {
  public void addCustomer(Customer c) throws DataAccessException {
    Connection con = getConnection();
    PreparedStatement s = con.createPreparedStatement("INSERT ...");
    ...
    s.executeUpdate();
    ...
  }
  ...
}
```

Chapter 20

Java Fundamentals Labs

20.1 Hello World Lab

- Write a program that prints “Hello, World!” to the console.
- Compile and run your program using your IDE.
- Compile and run your program from the command line.
  - Use the `javac` command to compile your source file into a class file.
  - Use the `java` command to run your program.

20.2 Data Types Lab: Arrays and Strings

- Write a program that accepts a command-line argument when run.
- Have your program print ‘`Hello, ” followed by the String argument.
- Compile and run your program from the command line.
- What happens if you don’t provide a command-line argument? What happens if you provide more than one?
- Compile and run your program using your IDE.
  - How do you provide command-line arguments to your application using your IDE?

20.3 Data Types Lab: Basic String Manipulation

- Write a program that accepts a command-line argument when run.
- On separate lines, print:
  - The original String command-line argument
  - The number of characters in the String argument
  - An all uppercase version of the String
  - An all lowercase version of the String
  - The first 4 characters of the String
- What happens if the string contains fewer than 4 characters?

20.4 Operators Lab: Numeric Conversions

- Write a program that accepts two command-line arguments.
- Convert each value to a number.
- Print the result of dividing the first number by the second number.
- What happens if you provide:
  - A non-numeric argument?
  - 0 for the second argument?
  - 0 for both arguments?

20.5 Flow Control: Multiplication Table

- Print a multiplication table ranging from 1 to some maximum value of your choice.
- Extra Credit: Make the maximum value a "named constant" (a `final` value).
- Extra Credit: Also print column and row headers, for example:

```
    1   2   3   4
    === === === ===
1:  1   2   3   4
2:  2   4   6   8
3:  3   6   9   12
4:  4   8   12  16
```

20.6 Flow Control Lab: Loan Amortization

- Write a simple loan amortization program.
- Accept three floating-point values as command-line arguments:
  - An initial loan amount
  - An annual interest rate, expressed as a percentage (e.g., "4.5" for 4.5%)
  - A monthly payment
- Print out a loan amortization schedule, with each line containing:
  - The month number
  - The amount of that month's payment
  - The amount of interest payable for that month
  - An updated balance
- Print a usage error message if the user does not provide exactly three arguments.
- Extra Credit: The last month should reflect a payment amount that gives a 0 balance, not negative.
- Extra Credit: If the amount of the monthly payment results in the loan never being paid off, notify the user rather than printing an infinite loan amortization schedule.

Note
In real life neither `float`s nor `double`s are suitable for calculations on monetary amounts due to floating-point truncation errors. To be on the safe side, you would use `java.math.BigDecimal` (more on this later). But for simplicity, you can use `float` or `double` amounts for this lab.

Tip
For a simple, non-threaded, command-line driven application like this, you might want to use the `System.exit(int)` method to terminate the application in the event of a usage error. For example:

```
System.exit(1);     // Terminate with an exit code of 1
```

20.7 OOP Lab: `CheckingAccount`, Part 1

- Model a checking account by defining a `CheckingAccount` class.
- Each `CheckingAccount` object should contain:
  - A `balance` field, represented by a `double`
  - A `name` field, represented by a `String`
- Write a separate application class that creates two (or more) `CheckingAccount` objects.
  - Set the name and amount fields on each object to values of your choosing.
  - Add and subtract amounts from each object.
  - For each object, print the account name and the final balance.

20.8 OOP Lab: `CheckingAccount`, Part 2

- Revise your `CheckingAccount` class.
  - Implement getters and setters for the `name` field.
  - Implement a getter but **no** setter for the `balance` field.
  - Implement two constructors:
    - One should accept an initial value for the `name` and `balance` fields.
    - The other should be a no-argument constructor that initializes the name to "New Account" and the balance to 0.0.
  - Implement a `credit(double)` method that increases the account balance by the specified amount.
  - Implement a `debit(double)` method that decreases the account balance by the specified amount.
  - Implement a `transferTo(CheckingAccount, double)` method that debits the specified amount from the current object and credits the same amount to the `CheckingAccount` object provided as an argument.
  - Assign appropriate access modifiers to all fields, methods, and constructors.
- Create a new application class that creates at least two `CheckingAccount` objects and exercises all of the methods you've implemented.

Consider the return values of your `credit()`, `debit()`, and `transferTo()` methods. Should they all have a `void` return type? Can you think of useful values they could return?

20.9 OOP Lab: Composition, `CheckingAccounts` and `Customers`

- Each checking account should be associated with a customer.
- Define a `Customer` class:
  - Each `Customer` should have `firstName`, `lastName`, and `ssn` fields, all stored as `String` values.
  - Implement appropriate getter, setter, and constructor methods for your `Customer` class.
- Modify your `CheckingAccount` class:
  - Add a `Customer` field.
  - Implement a `getCustomer()` method. (No `setCustomer()` to change the owner of the account after creation.)
  - Modify the constructors to require a `Customer` object.
- Update your application to test the features you added.
  - Create one or more `Customer` objects.
  - Create one or more `CheckingAccount` objects, making sure to assign a `Customer` object to each `CheckingAccount`.
  - Test the new capabilities of your classes.

20.10 OOP Lab: Inheritance, Part 1

- The bank wants to manage deposit accounts in addition to a checking account.
  - They want a `SavingsAccount` class that has the same features as a `CheckingAccount`, plus the ability to calculate and post interest.
  - They want to manage mixed collections of `CheckingAccount` and `SavingsAccount` objects (e.g., print the balances for all accounts in a collection).
  - They are thinking of adding new product types in the future, like CDs and money market accounts.
  - For any given account in a collection, they want to be able to determine the account type.
- How should you design the inheritance hierarchy to support this?
  - Should you use `CheckingAccount` as the base class for the hierarchy? Or should you create a new base class that `CheckingAccount` and the other account types inherit from?
  - How might you support identifying the type of account?
- Implement the new class hierarchy and test it out with your application.

20.11 OOP Lab: Inheritance, Part 2

- The bank wants an easy way to post interest to savings accounts.
  - For the `SavingsAccount` class only, the bank wants to have a method named `setInterestRate(double)` that accepts a new annual interest rate (e.g., `0.02` for a 2% annual rate) for **all** `SavingsAccount` objects.
  - The bank also wants a `creditInterest()` method that calculates the **monthly** interest for an individual account and automatically credits that amount to the account.

Note
For simplicity, assume that the monthly interest rate is just the annual rate divided by 12.

- Implement this new feature and test it out.
  - Be sure to set the interest rate and credit interest to one or more accounts.
  - Change the interest rate and verify that the new rate is being used by the accounts.

20.12 OOP Lab: Interfaces, Part 1

- The bank would like to manage loans as well as deposit accounts.
  - What features and capabilities of a loan are the same as a deposit account? What is different?
  - How would you design an inheritance hierarchy capable of handling various types of loans (e.g., mortgages, auto loans, personal loans, etc.).
- Can you envision product types other than loans and deposit accounts that the bank might want to manage in the future?
  - How might you use one or more interfaces to manage the different account types?

20.13 OOP Lab: Interfaces, Part 2

- Create a `BankAccount` interface.
  - The interface should declare all the methods currently implemented by the deposit account base class.
  - Modify your deposit account base class to `implement` the `BankAccount` interface.
- Test the `BankAccount` interface.
  - In your test application, declare an array variable of type `BankAccount[]`.
  - Create an array of various deposit accounts and assign it to the array variable.
  - Iterate over the elements in the array variable and print out some common properties, such as name, type, and balance.
- Time permitting, create a loan class hierarchy, where the base class implements the `BankAccount` interface.
  - Test a mixed collection of deposit and loan account.

20.14 OOP Lab: Overriding `java.lang.Object` Methods

- Override the `toString()` method for the `Customer` class, your deposit base class, and any other classes your think appropriate to return reasonable descriptive String representations of object information for each class.
  - Test the `toString()` methods in your banking application.
- Override the `equals()` for the `Customer` class.
  - Determine `Customer` object equality based solely on the `ssn` field.
    - You can use the `String.equals()` method to compare the `ssn` fields.
  - Because you have overridden the `Customer.equals()` method, you should also override the `Customer.hashCode()` method.

    * In this simple case, because the String `ssn` field is the only "significant" field for our equality test, we can safely use the `String.hashCode()` value of the `ssn` field as the hash code for our entire `Customer` object.
- Test your `Customer.equals()` implementation.

> **Tip**
> For simplicity, you might want to create a separate test application that only creates a few `Customer` objects and performs various comparisons using the `==` operator and the `equals()` method.

20.15 Packaging Lab

- Separate all of your code into distinct logical packages, including sub-packages.
  - Use a unique package prefix for your organization.
- Import classes from other packages as needed.
- Hide classes/methods/fields that should not be visible from the outside of your packages.
- Create JAR files for two or more of your packages (including sub-packages).
- Run your code by specifying the class path.
- As a bonus, create an *executable* jar file.

20.16 Doc Comments Lab

- Document your classes using doc comments.
- Generate the HTML API.
- Verify that it matches your source files.
- Create a dummy method, mark it as `@deprecated`, and use it from another class
  - What happens when you recompile?
  - Regenerate the documentation. What's different about your method?
- Experiment with other doc comment tags.

20.17 Exceptions Lab: Throwing Custom Exceptions

- Currently, our banking deposit classes allow debits greater than their current balances.
- Update the `debit()` method to `throw` a `NonSufficientFundsException` if the requested debit exceeds the balance.
- Define `NonSufficientFundsException` as a subclass of `Exception`.
  - Implement at least two constructors: a no-argument constructor, and a constructor that accepts a `String` error message.
- Notice that you now have compilation errors in your banking test application because of the checked exceptions thrown by `debit()`.
  - Modify your banking application to use `try-catch` structures around your `debit()` calls.
  - If a `NonSufficientFundsException` condition occurs, report on the console that the debit was not allowed.

20.18 I/O Lab

- Write a file copy program that not only copies text files, but also changes all alphabetical characters to upper case and strips all white space.
  - Handle all I/O exceptions properly.
  - Try using filtered character streams for maximum reusability.
  - Test by creating a simple text file and running your program against it.

You could implement this program by dividing it up in four separate pieces:

1. A filter that converts alphabetical characters to upper case
2. A filter that strips all white space characters
3. A copier that can copy from one stream to another
4. The main program that glues everything together

20.19 `java.net` Lab

- Create a simple web client using just TCP sockets (i.e. `java.net.Socket`).
- For example, the client can connect to `www.google.com` on port `80` and send the following data:
  `GET /search?q=Java HTTP/1.0\r\n\r\n`
- Read the response from the server and print it out to STDOUT (i.e., `System.out`).

The equivalent behavior could be achieved with the `telnet` command:

```
> telnet www.google.com 80
Trying 72.14.205.104...
Connected to www.l.google.com.
Escape character is '^]'.
GET /search?q=Java HTTP/1.0

HTTP/1.0 200 OK
Cache-Control: private
Content-Type: text/html; charset=ISO-8859-1
Set-Cookie: PREF=ID=59f02b988ad796bc:TM=1198882670:LM=1198882670:S=AMyQ1eoscAqbKBaH;  ↩
    expires=Sun, 27-Dec-2009 22:57:50 GMT; path=/; domain=.google.com
Server: gws
Date: Fri, 28 Dec 2007 22:57:50 GMT
Connection: Close

<html><head><meta http-equiv=content-type content="text/html; charset=ISO-8859-1">
...
```

20.20 Collections Lab

- Create a School class that aggregates all enrolled students in an appropriate data structure.
  - No duplicates
  - Fast access
  - Encapsulated data
- Test sorting students by first/last names, year of birth, and IDs using appropriate Comparators and/or by implementing the Comparable interface.

```
import java.util.Collection;

public class School {
    public void enroll(Student student) {... }
    public void remove(int id) {... }
    public Student getById(int id) {... }
    public int getStudentCount() {... }
    public Collection getAll() {... }
    public Collection getAllByName() {... }
    public Collection getAllByYearOfBirth(int from, int to) {...}
    ...
}
```

20.21 JDBC Lab

- Implement CRUD (Create, Read, Update, Delete) operations for managing Suppliers
  - In addition, provide a list/search operation
  - You can write a separate class for each operation, or do everything in just one class
- Use the provided `suppliers.sql` to create the table schema and load some sample data
- Provide meaningful handling for database exceptions
- Close database resources in a safe manner

Note
Remember that `INSERT`, `UPDATE`, and `DELETE` operations require the use of `executeUpdate` method, whereas `SELECT` is run through `executeQuery`.

To load the `suppliers.sql` file, from MySql client (prompt) do:

```
SOURCE /path/to/suppliers.sql
```

Or simply copy-paste the contents of the file onto the client prompt.

In practice, developers prefer to use OR/M (object-relational mapping) tools/libraries to avoid having to deal with JDBC/SQL directly and abstract away the database-specific SQL dialects. Examples of OR/M include JPA/EJB3 (part of Java EE 5), Hibernate, and TopLink.

Additionally, Data Access Object (DAO) design pattern is often used to group together data access calls through a generic interface, and technology/database-specific implementations.

20.22 XML Parsing Lab

- Take a look at the provided xml-example of parsing RSS feeds using SAX and DOM
  - Observe how the content is parsed from XML to the Java object representation
  - Observe how the Java object representation is converted back to XML
- Following the RSS example, build similar XML parser(s) for the provided address book application

---

The necessary files are part of xml-example Eclipse project.

For example, parse the following XML data (provided):

+

```
<?xml version="1.0" encoding="UTF-8"?>
<address-book>
  <contact>
    <first-name>John</first-name>
    <last-name>Smith</last-name>
    <email>john.smith@email.com</email>
    <phone type="office">415-555-1235</phone>
    <phone type="moble">415-456-1235</phone>
    <phone type="fax">415-456-7000</phone>
    <date-of-birth>1976-06-03</date-of-birth>
  </contact>
  ...
</address-book>
```