APPENDIX E

JAVA SYNTAX SUMMARY

In this syntax summary, we use a monospaced font for actual Java reserved words and tokens such as `while`. An italic font denotes language constructs such as *condition* or *variable*. Items enclosed in brackets `[ ]` are optional. Items separated by vertical bars | are alternatives. Do not include the brackets or vertical bars in your code!

The summary reflects the parts of the Java language that were covered in this book. For a full overview of the Java syntax, see `http://download.oracle.com/javase/7/docs/api/`.

Please be careful to distinguish an ellipsis . . . from the ... token. The latter appears twice in this appendix in the "variable parameters" discussion in the "Methods" section.

Types

A type is a primitive type or a reference type. The primitive types are

- The numeric types `int`, `long`, `short`, `char`, `byte`, `float`, `double`
- The `boolean` type

The reference types are

- Classes such as `String` or `Employee`
- Enumeration types such as `enum Sex { FEMALE, MALE }`
- Interfaces such as `Comparable`
- Array types such as `Employee[]` or `int[][]`

Variables

Local variable declarations have the form

`[final]` *Type variableName* `[=` *initializer*`];`

Examples:

```
int n;
double x = 0;
String harry = "Harry Handsome";
Rectangle box = new Rectangle(5, 10, 20, 30);
int[] a = { 1, 4, 9, 16, 25 };
```

The variable name consists only of letters, numbers, and underscores. It must begin with a letter or underscore. Names are case-sensitive: `totalscore`, `TOTALSCORE`, and `totalScore` are three different variables.

The scope of a local variable extends from the point of its definition to the end of the enclosing block.

A variable that is declared as `final` can have its value set only once.

Instance variables will be discussed under "Classes".

Expressions

An *expression* is a variable, a method call, or a combination of subexpressions joined by operators. Examples are:

```
x
Math.sin(x)
x + Math.sin(x)
x * (1 + Math.sin(x))
x++
x == y
x == y && (z > 0 || w > 0)
p.x
e.getSalary()
v[i]
```

Operators can be *unary*, *binary*, or *ternary*. A unary operator acts on a single expression, such as `x++`. A binary operator combines two expressions, such as `x + y`. A ternary operator combines three expressions. Java has one ternary operator, `? :` (see Special Topic 5.1).

Unary operators can be *prefix* or *postfix*. A prefix operator is written before the expression on which it operates, as in `-x`. A postfix operator is written after the expression on which it operates, such as `x++`.

Operators are ranked by *precedence* levels. Operators with a higher precedence bind more strongly than operators with a lower precedence. For example, `*` has a higher precedence than `+`, so `x + y * z` is the same as `x + (y * z)`, even though the `+` comes first.

Most operators are *left-associative*. That is, operators of the same precedence are evaluated from the left to the right. For example, `x - y + z` is interpreted as `(x - y) + z`, not `x - (y + z)`. The exceptions are the unary prefix operators and the assignment operator which are right-associative. For example, `z = y = Math.sin(x)` means the same as `z = (y = Math.sin(x))`.

Appendix B has a list of all Java operators.

Classes

The syntax for a *class* is

```
[public] [abstract|final] class ClassName
      [extends SuperClassName]
      [implements InterfaceName1, InterfaceName2, . . .]
{
   feature1
   feature2
   . . .
}
```

Each *feature* is either a declaration of the form

modifiers constructor|method|instance variable|class

or an initialization block

[static] { *body* }

See the section "Constructors" for more information about initialization blocks.

Potential *modifiers* include `public`, `private`, `protected`, `static`, and `final`.

An *instance variable* declaration has the form

Type variableName [= *initializer*];

A *constructor* has the form

ClassName(*parameter*$_1$, *parameter*$_2$, . . .)
    [throws *ExceptionType*$_1$, *ExceptionType*$_2$, . . .]
{
   body
}

A *method* has the form

Type methodName(*parameter*$_1$, *parameter*$_2$, . . .)
    [throws *ExceptionType*$_1$, *ExceptionType*$_2$, . . .]
{
   body
}

An *abstract method* has the form

abstract *Type methodName*(*parameter*$_1$, *parameter*$_2$, . . .);

Here is an example:

```
public class Point
{
   private double x;  // Instance variable
   private double y;

   public Point()  // Constructor with no arguments
   {
      x = 0; y = 0;
   }

   public Point(double xx, double yy)  // Constructor
   {
      x = xx; y = yy;
   }

   public double getX()  // Method
   {
      return x;
   }

   public double getY()  // Method
   {
      return y;
   }
}
```

A class can have both instance variables and `static` variables. Each object of the class has a separate copy of the instance variables. There is only a one per-class copy of the `static` variables.

A class that is declared as `abstract` cannot be instantiated. That is, you cannot construct objects of that class.

A class that is declared as `final` cannot be extended.

Interfaces

The syntax for an interface is

```
[public] interface InterfaceName
      [extends InterfaceName1, InterfaceName2, . . .]
{
   feature1
   feature2
   . . .
}
```

Each feature has the form

modifiers method|*instance variable*

Potential modifiers are `public`, `static`, `final`. However, modifiers are never necessary because methods are automatically `public` and instance variables are automatically `public static final`.

An instance variable declaration has the form

Type variableName = *initializer*;

A method declaration has the form

Type methodName(*parameter*$_1$, *parameter*$_2$, . . .);

Here is an example:

```
public interface Measurable
{
   int CM_PER_INCH = 2.54;

   int getMeasure();
}
```

Enumeration Types

The syntax for an enumeration type is

```
[public] enum EnumerationTypeName
{
   constant1, constant2, . . .;
   feature1
   feature2
   . . .
}
```

Each constant is a constant name, followed by optional construction parameters.

constantName[(*parameter*$_1$, *parameter*$_2$, . . .)]

The semicolon after the constants is only required if the enumeration declares additional features. An enumeration can have the same features as a class. Each feature has the form

modifiers method|instance variable

Potential modifiers are `public`, `static`, `final`.

Here are two examples:

```
public enum Suit { HEARTS, DIAMONDS, SPADES, CLUBS };
public enum Card
{
   TWO(2), THREE(3), FOUR(4), FIVE(5), SIX(6),
         SEVEN(7), EIGHT(8), NINE(9), TEN(10),
         JACK(10), QUEEN(10), KING(10), ACE(11);
   private int value;

   public void Card(int aValue) { value = aValue; }
   public int getValue() { return value; }
}
```

Methods

A method definition has the form

modifiers Type methodName(*parameter*$_1$, *parameter*$_2$, . . ., *parameter*$_n$)
    [`throws` *ExceptionType*$_1$, *ExceptionType*$_2$, . . .]
{
   body
}

The return type *Type* is any Java type, or the special type `void` to indicate that the method returns no value.

Each *parameter variable* has the form

[`final`] *Type parameterName*

A method has variable parameters if the last parameter variable has the special form

Type`...` *parameterName*

Such a method can be called with a sequence of values of the given type of any length. The parameter variable with the given name is an array of the given type that holds the arguments. For example, the method

```
public static double sum(double... values)
{
   double s = 0;
   for (double v : values) { s = s + v; }
   return s;
}
```

can be called as

```
double result = sum(1, -2.5, 3.14);
```

In Java, all parameters are passed by *value*. Each parameter variable is a local variable whose scope extends to the end of the method body. It is initialized with a copy of the value supplied in the call. That value may be a primitive type or a reference type. If it

is a reference type, invoking a mutator on the reference will modify the object whose reference has been passed to the method.

Changing the value of the parameter variable has no effect outside the method. Tagging the parameter variable as `final` disallows such a change altogether. This is commonly done to allow access to the parameter variable from an inner class declared in the method.

Java distinguishes between *instance* methods and *static* methods. Instance methods have a special parameter, the *implicit* parameter, supplied in the method call with the syntax

implicitParameterValue.*methodName*(*parameterValue*$_1$, *parameterValue*$_2$, . . .)

Example:

```
harry.setSalary(30000)
```

The type of the implicit parameter must be the same as the type of the class containing the method definition. A static method does not have an implicit parameter.

In the method body, the `this` variable is initialized with a copy of the implicit parameter value. Using an instance variable name without qualification means to access the instance variable of the implicit parameter. For example,

```
public void setSalary(double s)
{
   salary = s; // i.e., this.salary = s
}
```

By default, Java uses *dynamic method lookup*. The virtual machine determines the class to which the implicit parameter object belongs and invokes the method declared in that class. However, if a method is invoked on the special variable `super`, then the method declared in the superclass is invoked on `this`. For example,

```
public class MyPanel extends JPanel
{
   . . .
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      // Calls JPanel.paintComponent
      . . .
   }
   . . .
}
```

The `return` statement causes a method to exit immediately. If the method type is not `void`, you must return a value. The syntax is

```
return [value];
```

For example,

```
public double getSalary()
{
   return salary;
}
```

A method can call itself. Such a method is called *recursive:*

```
public static int factorial(int n)
{
   if (n <= 1) { return 1; }
   return n * factorial(n - 1);
}
```

Constructors

A constructor definition has the form

modifiers *ClassName*(*parameter*$_1$, *parameter*$_2$, . . .)
    [throws *ExceptionType*$_1$, *ExceptionType*$_2$, . . .]
{
   body
}

You invoke a constructor to allocate and construct a new object with a `new` expression

`new` *ClassName*(*parameterValue*$_1$, *parameterValue*$_2$, . . .)

A constructor can call the body of another constructor of the same class with the syntax

`this`(*parameterValue*$_1$, *parameterValue*$_2$, . . .)

For example,

```
public Employee()
{
   this("", 0);
}
```

It can call a constructor of its superclass with the syntax

`super`(*parameterValue*$_1$, *parameterValue*$_2$, . . .)

The call to `this` or `super` must be the first statement in the constructor.

Arrays are constructed with the syntax

`new` *ArrayType* `[ = {` *initializer*$_1$, *initializer*$_2$, . . . `}]`

For example,

```
new int[] = { 1, 4, 9, 16, 25 }
```

When an object is constructed, the following actions take place:

- All instance variables are initialized with `0`, `false`, or `null`.
- The initializers and initialization blocks are executed in the order in which they are declared.
- The body of the constructor is invoked.

When a class is loaded, the following actions take place:

- All static variables are initialized with `0`, `false`, or `null`.
- The initializers of static variables and static initialization blocks are executed in the order in which they are declared.

Statements

A *statement* is one of the following:

- An expression followed by a semicolon
- A branch or loop statement
- A `return` statement

- A `throw` statement
- A block, that is, a group of variable declarations and statements enclosed in braces `{. . .}`
- A `try` block

Java has two branch statements (`if` and `switch`), three loop statements (`while`, `for`, and `do`), and two mechanisms for nonlinear control flow (`break` and `continue`).

The `if` statement has the form

`if (`*condition*`)` *statement*$_1$ `[else` *statement*$_2$`]`

If the *condition* is true, then the first *statement* is executed. Otherwise, the second *statement* is executed.

The `switch` statement has the form

```
switch (expression)
{
   group1
   group2
   . . .
     [default:
      statement1
      statement2
      . . .]
}
```

Where each group has the form

```
case constant1:
case constant2:
. . .
   statement1
   statement2
   . . .
```

The *expression* must be an integer, enumeration type, or string. Depending on its value, control is transferred to the first statement following the matching `case` label, or to the first statement following the `default` label if none of the `case` labels match. Execution continues with the next statement until a `break` or `return` statement is encountered, an exception is thrown, or the end of the `switch` is reached. Execution skips over any `case` labels.

The `while` loop has the form

`while (`*condition*`)` *statement*

The *statement* is executed while the *condition* is true.

The `for` loop has the form

```
for (initExpression|variableDeclaration;
      condition;
      updateExpression1, updateExpression2, . . .)
   statement
```

The initialization expression or the variable declaration are executed once. While the *condition* remains true, the loop *statement* and the *updateExpressions* are executed.

Examples:

```
for (i = 0; i < 10; i++)
{
   sum = sum + i;
}

for (int i = 0, j = 9; i < 10; i++, j--)
{
   a[j] = b[i];
}
```

The enhanced `for` loop has the form

```
for (Type variable : array|iterableObject)
   statement
```

When this loop traverses an array, it is equivalent to

```
for (int i = 0; i < array.length; i++)
{
   Type variable = array[i];
   statement
}
```

Otherwise, the *iterableObject* must belong to a class that implements the `Iterable` interface. Then the loop is equivalent to

```
Iterator i = iterableObject.iterator();
while (i.hasNext())
{
   Type variable = i.next();
   statement
}
```

The `do` loop has the form

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

The *statement* is repeatedly executed until the *condition* is no longer true. In contrast to a `while` loop, the statement of a `do` loop is executed at least once.

The `break` statement exits the innermost enclosing `while`, `do`, `for`, or `switch` statement (not counting `if` or block statements).

Any statement (including `if` and block statements) can be tagged with a label:

```
label: statement
```

The labeled `break` statement

```
break label;
```

exits the labeled statement.

The `continue` statement skips past the end of the *statement* part of a `while`, `do`, or `for` loop. In the case of the `while` or `do` loop, the loop *condition* is executed next. In the case of the `for` loop, the *updateExpressions* are executed next.

The labeled `continue` statement

```
continue label;
```

skips past the end of the *statement* part of a `while`, `do`, or `for` loop with the matching label.

Exceptions

The `throw` statement

```
throw expression;
```

abruptly terminates the current method and resumes control inside the innermost matching `catch` clause of a surrounding `try` block. The *expression* must evaluate to a reference to an object of a subclass of `Throwable`.

The `try` statement has the form

```
try tryBlock
[catch (ExceptionType1 exceptionVariable1) catchBlock1
catch (ExceptionType2 exceptionVariable2) catchBlock2
. . .]
[finally finallyBlock]
```

- The `try` statement must have at least one `catch` or `finally` clause.
- All blocks are block statements in the usual sense, that is, `{ . . . }`-delimited statement sequences.

The statements in the *tryBlock* are executed. If one of them throws an exception object whose type is a subtype of one of the types in the `catch` clauses, then its *catchBlock* is executed. As soon as the catch block is entered, that exception is handled.

If the *tryBlock* exits for any reason at all (because all of its statements executed completely; because one of its statements was a `break`, `continue`, or `return` statement; or because an exception was thrown), then the *finallyBlock* is executed.

If the *finallyBlock* was entered because an exception was thrown and it itself throws another exception, then that exception masks the prior exception.

Packages

A class can be placed in a package by putting the package declaration

```
package packageName;
```

as the first non-`import` declaration of the source file.

A package name has the form

```
identifier1.identifier2. . . .
```

For example,

```
java.util
com.horstmann.bigjava
```

A fully qualified name of a class is

```
packageName.ClassName
```

Classes can always be referenced by their fully qualified class names. However, this can be inconvenient. For that reason, you can reference imported classes by just their *ClassName*. All classes in the package `java.lang` and in the package of the current source file are always imported.

To import additional classes, use an `import` directive

`import` *packageName*`.`*ClassName*`;`

or

`import` *packageName*`.*;`

The second version imports all classes in the package.

Generic Types and Methods

A generic type is declared with one or more type parameters, placed after the type name:

modifiers `class|interface` *TypeName*`<`*typeParameter*$_1$`,` *typeParameter*$_2$`, . . .>`

Similarly, a generic method is declared with one or more type parameters, placed before the method's return type:

modifiers `<`*typeParameter*$_1$`,` *typeParameter*$_2$`, . . .>` *returnType methodName*

Each type parameter has the form

typeParameterName `[extends` *bound*$_1$ `&` *bound*$_2$ `& . . .]`

For example,

```
public class BinarySearchTree<T extends Comparable>
public interface Comparator<T>
public <T extends Comparable & Cloneable> T cloneMin(T[] values)
```

Type parameters can be used in the definition of the generic type or method as if they were regular types. They can be replaced with any types that match the bounds. For example, the `BinarySearchTree<String>` type substitutes the `String` type for the type parameter `T`.

Type parameters can also be replaced with *wildcard types*. A wildcard type has the form

`? [super|extends` *Type*`]`

It denotes a specific type that is unknown at the time that it is declared. For example, `Comparable<? super Rectangle>` is a type `Comparable<S>` for a specific type `S`, which can be `Rectangle` or a supertype such as `RectangularShape` or `Shape`.

Comments

There are three kinds of comments:

```
/* comment */
// one-line-comment
/** documentationComment */
```

The one-line comment extends to the end of the line. The other comments can span multiple lines and extend to the `*/` delimiter.

Documentation comments are further explained in Appendix F.

APPENDIX F

TOOL SUMMARY

In this summary, we use a monospaced font for actual commands such as `javac`. An italic font denotes descriptions of tool command components such as *options*. Items enclosed in brackets [. . .] are optional. Items separated by vertical bars | are alternatives. Do not include the brackets or vertical bars when typing the commands.

The Java Compiler

`javac` [*options*] *sourceFile*$_1$|`@`*fileList*$_1$ *sourceFile*$_2$|`@`*fileList*$_2$ . . .

A file list is a text file that contains one file name per line. For example,

Greeting.list

```
1  Greeting.java
2  GreetingTester.java
```

Then you can compile all files with the command

```
javac @Greeting.list
```

The Java compiler options are summarized in Table 1.

Table 1 Common Compiler Options

Option	Description
`-classpath` *locations* or `-cp` *locations*	The compiler is to look for classes on this path, overriding the `CLASSPATH` environment variable. If neither is specified, the current directory is used. Each *location* is a directory, JAR file, or ZIP file. Locations are separated by a platform-dependent separator (`:` on Unix, `;` on Windows).
`-sourcepath` *locations*	The compiler is to look for source files on this path. If not specified, source files are searched in the class path.
`-d` *directory*	The compiler places files into the specified directory.
`-g`	Generate debugging information.
`-verbose`	Include information about all classes that are being compiled (useful for troubleshooting).
`-deprecation`	Give detailed information about the usage of deprecated messages.
`-Xlint:`*errorType*	Carry out additional error checking. If you get warnings about unchecked conversions, compile with the `-Xlint:unchecked` option.

The Java Virtual Machine Launcher

The following command loads the given class and starts its `main` method, passing it an array containing the provided command line arguments:

```
java [options] ClassName [argument1 argument2 . . . ]
```

The following command loads the main class of the given JAR file and starts its `main` method, passing it an array containing the provided command line arguments:

```
java [options] -jar jarFileName [argument1 argument2 . . . ]
```

The Java virtual machine options are summarized in Table 2.

Table 2 Common Virtual Machine Launcher Options

Option	Description
`-classpath` *locations* or `-cp` *locations*	Look for classes on this path, overriding the `CLASSPATH` environment variable. If neither is specified, the current directory is used. Each *location* is a directory, JAR file, or ZIP file. Locations are separated by a platform-dependent separator (: on Unix, ; on Windows).
`-verbose`	Trace class loading
`-D`*property*`=`*value*	Set a system property that you can retrieve with the `System.getProperties` method.

The JAR Tool

To combine one or more files into a JAR (Java Archive) file, use the command

```
jar cvf jarFile file1 file2 . . .
```

The resulting JAR file can be included in a class path.

To build a program that can be launched with `java -jar`, you must create a *manifest file*, such as

myprog.mf

```
1  Main-Class: com/horstmann/MyProg
```

The manifest must specify the path name of the class file that launches the application, but with the `.class` extension removed. Then build the JAR file as

```
jar cvfm jarFile manifestFile file1 file2 . . .
```

You can also use JAR as a replacement for a ZIP utility, simply to compress and bundle a set of files for any purpose. Then you may want to suppress the generation of the JAR manifest, with the command

```
jar cvfM jarFile file1 file2 . . .
```

To extract the contents of a JAR file into the current directory, use

```
jar xvf jarFile
```

To see the files contained in a JAR file without extracting the files, use

```
jar tvf jarFile
```

The javadoc Tool

To extract documentation comments (summarized in the following section), run the `javadoc` program:

```
javadoc [options] sourceFile1|packageName1|@fileList1
        sourceFile2|packageName2|@fileList2 . . .
```

Commonly used options are summarized in Table 3. See the documentation of the `javac` command in the first section of this appendix for an explanation of file lists.

To document all files in the current directory, use (all on one line)

```
javadoc -link http://download.oracle.com/javase/7/docs/api -d docdir *.java
```

Table 3 Common javadoc Command Line Options

Option	Description
`-link` *URL*	Link to another set of `javadoc` files. You should include a link to the standard library documentation, either locally or at `http://download.oracle.com/javase/7/docs/api`.
`-d` *directory*	Store the output in *directory*. This is a useful option, because it keeps your current directory from being cluttered up with `javadoc` files.
`-classpath` *locations*	Look for classes on the specified paths, overriding the `CLASSPATH` environment variable. If neither is specified, the current directory is used. Each *location* is a directory, JAR file, or ZIP file. Locations are separated by a platform-dependent separator (: Unix, ; Windows).
`-sourcepath` *locations*	Look for source files on the specified paths. If not specified, source files are searched in the class path.
`-author, -version`	Include author, version information in the documentation. This information is omitted by default.

Documentation Comments

A documentation comment is delimited by `/**` and `*/`. You can comment

- Classes
- Methods
- Instance variables

Each comment is placed *immediately above* the feature it documents.

Each `/** . . . */` documentation comment contains introductory text followed by tagged documentation. A tag starts with an `@` character, such as `@author` or `@param`. Tags are summarized in Table 4. The *first sentence* of the introductory text should be a summary statement. The `javadoc` utility automatically generates summary pages that extract these sentences.

You can use HTML tags such as `em` for emphasis, `code` for a monospaced font, `img` for images, `ul` for bulleted lists, and so on.

Here is a typical example. The summary sentence (in color) will be included with the method summary.

```
/**
   Withdraws money from the bank account. Increments the
   transaction count.
   @param amount the amount to withdraw
   @return the balance after the withdrawal
   @throws IllegalArgumentException if the balance is not sufficient
*/
public double withdraw(double amount)
{
   if (balance - amount < minimumBalance)
   {
      throw new IllegalArgumentException();
   }
   balance = balance - amount;
   transactions++;
   return balance;
}
```

Table 4 Common javadoc Tags

Tag	Description
`@param` *parameter explanation*	A parameter of a method. Use a separate tag for each parameter.
`@return` *explanation*	The return value of a method.
`@throws` *exceptionType explanation*	An exception that a method may throw. Use a separate tag for each exception.
`@deprecated`	A feature that remains for compatibility but that should not be used for new code.
`@see` *packageName.ClassName* `@see` *packageName.ClassName* *#methodName*(*Type*$_1$, *Type*$_2$, . . .) `@see` *packageName.ClassName#variableName*	A reference to a related documentation entry.
`@author`	The author of a class or interface. Use a separate tag for each author.
`@version`	The version of a class or interface.

APPENDIX G

NUMBER SYSTEMS

Binary Numbers

Decimal notation represents numbers as powers of 10, for example

$$1729_{\text{decimal}} = 1\times 10^3 + 7\times 10^2 + 2\times 10^1 + 9\times 10^0$$

There is no particular reason for the choice of 10, except that several historical number systems were derived from people's counting with their fingers. Other number systems, using a base of 12, 20, or 60, have been used by various cultures throughout human history. However, computers use a number system with base 2 because it is far easier to build electronic components that work with two values, which can be represented by a current being either off or on, than it would be to represent 10 different values of electrical signals. A number written in base 2 is also called a *binary* number.

For example,

$$1101_{\text{binary}} = 1\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0 = 8 + 4 + 1 = 13$$

For digits after the "decimal" point, use negative powers of 2.

$$\begin{aligned} 1.101_{\text{binary}} &= 1\times 2^0 + 1\times 2^{-1} + 0\times 2^{-2} + 1\times 2^{-3} \\ &= 1 + \frac{1}{2} + \frac{1}{8} \\ &= 1 + 0.5 + 0.125 = 1.625 \end{aligned}$$

In general, to convert a binary number into its decimal equivalent, simply evaluate the powers of 2 corresponding to digits with value 1, and add them up. Table 1 shows the first powers of 2.

To convert a decimal integer into its binary equivalent, keep dividing the integer by 2, keeping track of the remainders. Stop when the number is 0. Then write the remainders as a binary number, starting with the *last* one. For example,

$$\begin{aligned} 100 \div 2 &= 50 \text{ remainder } 0 \\ 50 \div 2 &= 25 \text{ remainder } 0 \\ 25 \div 2 &= 12 \text{ remainder } 1 \\ 12 \div 2 &= 6 \text{ remainder } 0 \\ 6 \div 2 &= 3 \text{ remainder } 0 \\ 3 \div 2 &= 1 \text{ remainder } 1 \\ 1 \div 2 &= 0 \text{ remainder } 1 \end{aligned}$$

Therefore, $100_{\text{decimal}} = 1100100_{\text{binary}}$.

Conversely, to convert a fractional number less than 1 to its binary format, keep multiplying by 2. If the result is greater than 1, subtract 1. Stop when the number is 0. Then use the digits before the decimal points as the binary digits of the fractional part, starting with the *first* one. For example,

$$0.35 \cdot 2 = 0.7$$
$$0.7 \cdot 2 = 1.4$$
$$0.4 \cdot 2 = 0.8$$
$$0.8 \cdot 2 = 1.6$$
$$0.6 \cdot 2 = 1.2$$
$$0.2 \cdot 2 = 0.4$$

Here the pattern repeats. That is, the binary representation of 0.35 is 0.01 0110 0110 0110 . . .

To convert any floating-point number into binary, convert the whole part and the fractional part separately.

Table 1 Powers of Two

Power	Decimal Value
2^0	1
2^1	2
2^2	4
2^3	8
2^4	16
2^5	32
2^6	64
2^7	128
2^8	256
2^9	512
2^{10}	1,024
2^{11}	2,048
2^{12}	4,096
2^{13}	8,192
2^{14}	16,384
2^{15}	32,768
2^{16}	65,536

Overflow and Roundoff Errors

In Java, an `int` value is an integer that is 32 bits long. When combining two such values, it is possible that the result does not fit into 32 bits. In that case, only the last 32 bits of the results are used, yielding an incorrect answer. For example,

```
int fiftyMillion = 50000000;
System.out.println(100 * fiftyMillion); // Expected: 5000000000
```

displays 705032704.

To see why this curious value is the result, one can carry out the long multiplication by hand:

```
1 1 0 0 1 0 0 * 1 0 1 1 1 1 1 0 1 0 1 1 1 1 0 0 0 0 1 0 0 0 0 0 0 0

1 0 1 1 1 1 1 0 1 0 1 1 1 1 0 0 0 0 1 0 0 0 0 0 0 0
  1 0 1 1 1 1 1 0 1 0 1 1 1 1 0 0 0 0 1 0 0 0 0 0 0 0
    0
      0
        1 0 1 1 1 1 1 0 1 0 1 1 1 1 0 0 0 0 1 0 0 0 0 0 0 0
          0
            0
_____________________________________________________________
1 0 0 1 0 1 0 1 0 0 0 0 0 0 1 0 1 1 1 1 1 0 0 1 0 0 0 0 0 0 0 0
```

The result has 33 bits. However, you can't fit a 33-bit result into a 32-bit `int`, and the top bit is discarded. The last 32 bits are the binary representation of 705032704. (Note that the top bit is 2^{32} = 4294967296, and the two values add up to 5000000000, the correct result.)

With floating-point numbers, you can encounter another type of error: roundoff error. Consider this example:

```
double price = 4.35;
double quantity = 100;
double total = price * quantity; // Should be 100 * 4.35 = 435
System.out.println(total); // Prints 434.99999999999999
```

To see why the error occurs, carry out the long multiplication:

```
1 1 0 0 1 0 0 * 1 0 0.0 1|0 1 1 0|0 1 1 0|0 1 1 0 ...

1 0 0.0 1|0 1 1 0|0 1 1 0|0 1 1 0 ...
  1 0 0.0 1|0 1 1 0|0 1 1 0|0 1 1 ...
    0
      0
        1 0 0.0 1|0 1 1 0|0 1 1 0 ...
          0
            0
_____________________________________
1 1 0 1 1 0 0 1 0.1 1 1 1 1 1 1 1 ...
```

That is, the result is 434, followed by an infinite number of 1s. The fractional part of the product is the binary equivalent of an infinite decimal fraction 0.999999 . . . , which is equal to 1. But the CPU can store only a finite number of 1s, and it discards some of them when converting the result to a decimal number.

Two's Complement Integers

To represent negative integers, there are two common representations, called "signed magnitude" and "two's complement". Signed magnitude notation is simple: use the leftmost bit for the sign (0 = positive, 1 = negative). For example, when using 8-bit numbers,

$$-13 = 10001101_{\text{signed magnitude}}$$

However, building circuitry for adding numbers gets a bit more complicated when one has to take a sign bit into account. The two's complement representation solves this problem. To form the two's complement of a number,

- Flip all bits.
- Then add 1.

For example, to compute –13 as an 8-bit value, first flip all bits of 00001101 to get 11110010. Then add 1:

$$-13 = 11110011_{\text{two's complement}}$$

Now no special circuitry is required for adding two numbers. Simply follow the normal rule for addition, with a carry to the next position if the sum of the digits and the prior carry is 2 or 3. For example,

```
      1 1111 111
+13     0000 1101
-13     1111 0011
      ------------
    1 0000 0000
```

But only the last 8 bits count, so +13 and –13 add up to 0, as they should.

In particular, –1 has two's complement representation 1111 . . . 1111, with all bits set.

The leftmost bit of a two's complement number is 0 if the number is positive or zero, 1 if it is negative.

Two's complement notation with a given number of bits can represent one more negative number than positive numbers. For example, the 8-bit two's complement numbers range from –128 to +127.

This phenomenon is an occasional cause for a programming error. For example, consider the following code:

```
short b = ...;
if (b < 0) { b = (byte) -b; }
```

This code does not guarantee that `b` is nonnegative afterwards. If `b` happens to be –128, then computing its negative again yields –128. (Try it out—take 10000000, flip all bits, and add 1.)

IEEE Floating-Point Numbers

The Institute for Electrical and Electronics Engineering (IEEE) defines standards for floating-point representations in the IEEE-754 standard. Figure 1 shows how single-precision (`float`) and double-precision (`double`) values are decomposed into

- A sign bit
- An exponent
- A mantissa

Floating-point numbers use scientific notation, in which a number is represented as

$$b_0.b_1b_2b_3\ldots \times 2^e$$

In this representation, e is the exponent, and the digits $b_0.b_1b_2b_3\ldots$ form the mantissa. The *normalized* representation is the one where $b_0 \neq 0$. For example,

$$100_{\text{decimal}} = 1100100_{\text{binary}} = 1.100100_{\text{binary}} \times 2^6$$

In the binary number system, because the first bit of a normalized representation must be 1, it is not actually stored in the mantissa. Therefore, you always need to add it on to represent the actual value. For example, the mantissa 1.100100 is stored as 100100.

The exponent part of the IEEE representation uses neither signed magnitude nor two's complement representation. Instead, a *bias* is added to the actual exponent. The bias is 127 for single-precision numbers and 1023 for double-precision numbers. For example, the exponent $e = 6$ would be stored as 133 in a single-precision number.

Thus,

$$100_{\text{decimal}} = 0\ 10000101\ 10010000000000000000000_{\text{single-precision IEEE}}$$

In addition, there are several special values. Among them are:

- *Zero:* biased exponent = 0, mantissa = 0.
- *Infinity:* biased exponent = 11. . .1, mantissa = ±0.
- *NaN* (not a number): biased exponent = 11 . . . 1, mantissa ≠ ±0.

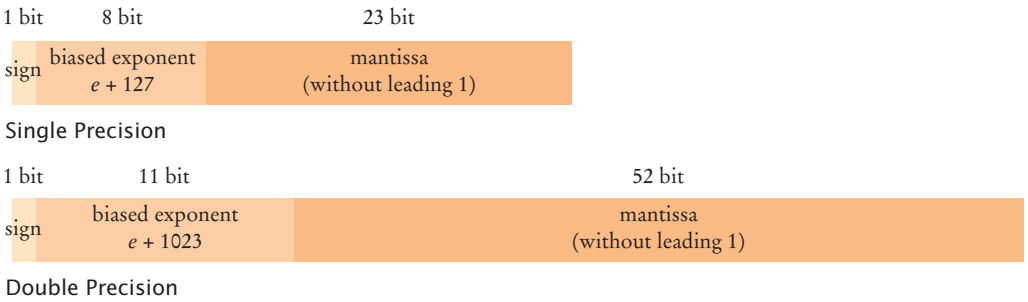


Figure 1 IEEE Floating-Point Representation

Hexadecimal Numbers

Because binary numbers can be hard to read for humans, programmers often use the hexadecimal number system, with base 16. The digits are denoted as 0, 1, …, 9, A, B, C, D, E, F. (See Table 2.)

Four binary digits correspond to one hexadecimal digit. That makes it easy to convert between binary and hexadecimal values. For example,

$$11|1011|0001_{\text{binary}} = 3B1_{\text{hexadecimal}}$$

In Java, hexadecimal numbers are used for Unicode character values, such as `\u03B1` (the Greek lowercase letter alpha). Hexadecimal integers are denoted with a `0x` prefix, such as `0x3B1`.

Table 2 Hexadecimal Digits

Hexadecimal	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Bit and Shift Operations

There are four bit operations in Java: the unary negation (~) and the binary *and* (&), *or* (|), and *exclusive or* (^), often called *xor*.

Tables 3 and 4 show the truth tables for the bit operations in Java. When a bit operation is applied to integer values, the operation is carried out on corresponding bits.

For example, suppose you want to compute `46 & 13`. First convert both values to binary. $46_{decimal} = 101110_{binary}$ (actually `00000000000000000000000000101110` as a 32-bit integer), and $13_{decimal} = 1101_{binary}$. Now combine corresponding bits:

```
  0.....0101110
& 0.....0001101
  0.....0001100
```

The answer is $1100_{binary} = 12_{decimal}$.

You sometimes see the `|` operator being used to combine two bit patterns. For example, the symbolic constant `BOLD` is the value 1, and the symbolic constant `ITALIC` is 2. The binary *or* combination `BOLD | ITALIC` has both the bold and the italic bit set:

```
  0.....0000001
| 0.....0000010
  0.....0000011
```

Don't confuse the `&` and `|` bit operators with the `&&` and `||` operators. The latter work only on `boolean` values, not on bits of numbers.

Table 3 The Unary Negation Operation

a	~a
0	1
1	0

Table 4 The Binary And, Or, and Xor Operations

a	b	a & b	a | b	a ^ b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Besides the operations that work on individual bits, there are three *shift* operations that take the bit pattern of a number and shift it to the left or right by a given number of positions. There are three shift operations: left shift (<<), right shift with sign extension (>>), and right shift with zero extension (>>>).

The left shift moves all bits to the left, filling in zeroes in the least significant bits. Shifting to the left by n bits yields the same result as multiplication by 2^n. The right shift with sign extension moves all bits to the right, propagating the sign bit. There-

fore, the result is the same as integer division by 2^n, both for positive and negative values. Finally, the right shift with zero extension moves all bits to the right, filling in zeroes in the most significant bits. (See Figure 2.)

Note that the right-hand-side value of the shift operators is reduced modulo 32 (for `int` values) or 64 (for `long` values) to determine the actual number of bits to shift.

For example, `1 << 35` is the same as `1 << 3`. Actually shifting 1 by 35 bits to the left would make no sense—the result would be `0`.

The expression

```
1 << n
```

yields a bit pattern in which the `n`th bit is set (where the `0` bit is the least significant bit).

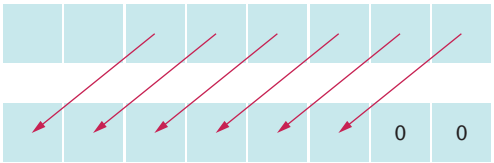
To set the `n`th bit of a number, carry out the operation

```
x = x | 1 << n
```

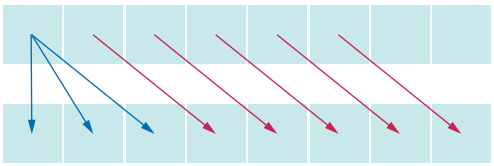
To check whether the `n`th bit is set, execute the test

```
if ((x & 1 << n) != 0) . . .
```

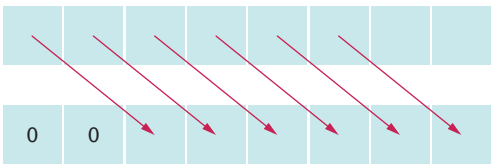
Note that the parentheses around the `&` are required—the `&` operator has a lower precedence than the relational operators.



Left shift (<<)



Right shift with sign extension (>>)



Right shift with zero extension (>>>)

Figure 2
The Shift Operations

APPENDIX H

UML SUMMARY

In this book, we use a very restricted subset of the UML notation. This appendix lists the components of the subset.

CRC Cards

CRC cards are used to describe in an informal fashion the responsibilities and collaborators for a class. Figure 1 shows a typical CRC card.

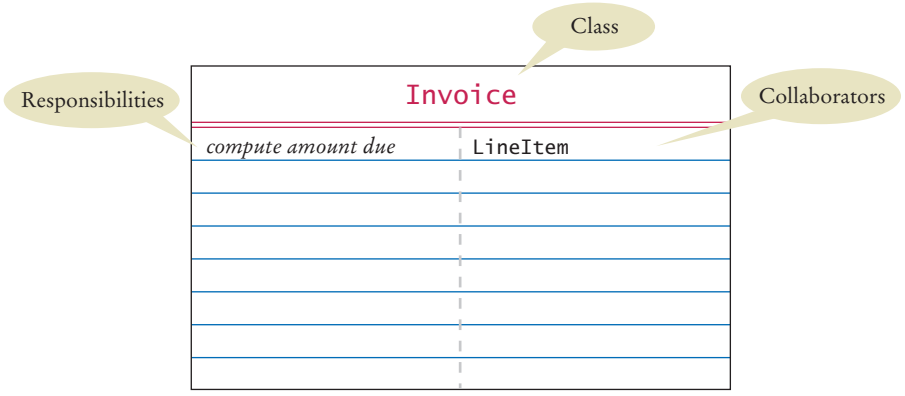


Figure 1 Typical CRC Card

UML Diagrams

Figure 2 shows the UML notation for classes and interfaces. You can optionally supply attributes and methods in a class diagram, as in Figure 3.

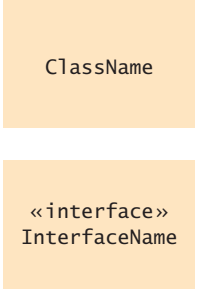


Figure 2
UML Symbols for Classes and Interfaces

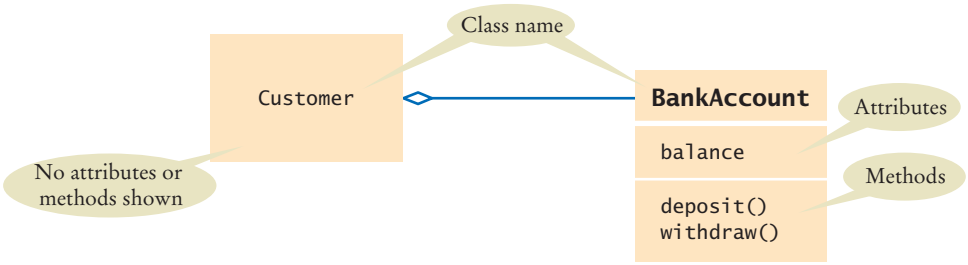


Figure 3
Attributes and Methods in a Class Diagram

Table 1 shows the arrows used to indicate relationships between classes. Multiplicity can be indicated in a diagram, as in Figure 4.

Table 1 UML Relationship Symbols

Relationship	Symbol	Line Style	Arrow Tip
Inheritance		Solid	Triangle
Interface Implementation		Dotted	Triangle
Aggregation		Solid	Diamond
Dependency		Dotted	Open

Figure 4
An Aggregation Relationship with Multiplicities

Dependencies between objects are described by a dependency diagram. Figure 5 is a typical example.

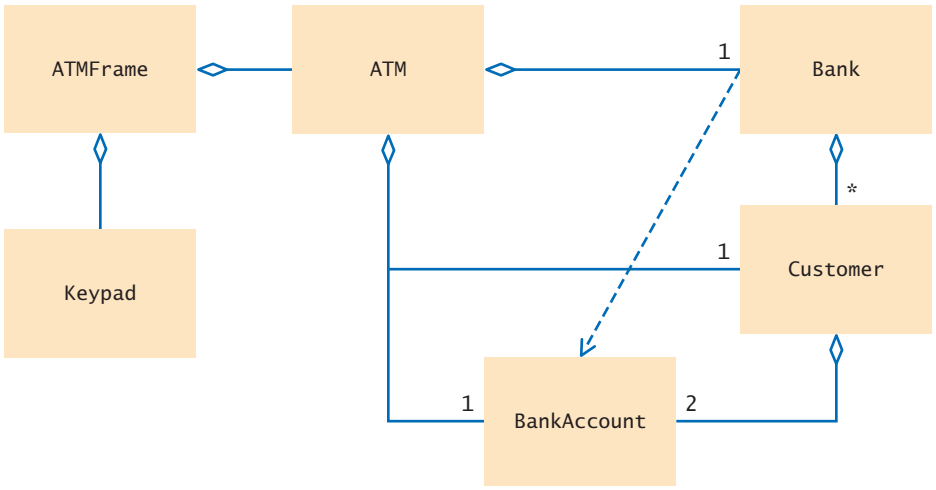


Figure 5
UML Class Diagram for the ATM Simulation

State diagrams are used when an object goes through a discrete set of states that affects its behavior (see Figure 6).

For a complete discussion of the UML notation, see *The Unified Modeling Language User Guide*, by Booch, Rumbaugh, and Jacobson (Addison-Wesley, 2005).

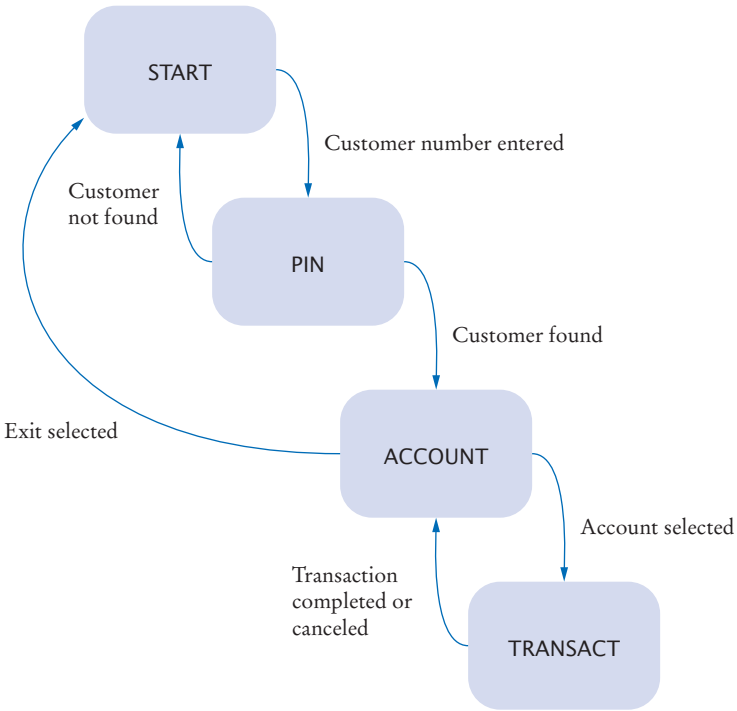


Figure 6 UML State Diagram for the ATM Class

APPENDIX

JAVA LANGUAGE CODING GUIDELINES

Introduction

This coding style guide is a simplified version of one that has been used with good success both in industrial practice and for college courses.

A style guide is a set of mandatory requirements for layout and formatting. Uniform style makes it easier for you to read code from your instructor and classmates. You will really appreciate that if you do a team project. It is also easier for your instructor and your grader to grasp the essence of your programs quickly.

A style guide makes you a more productive programmer because it reduces gratuitous choice. If you don't have to make choices about trivial matters, you can spend your energy on the solution of real problems.

In these guidelines, several constructs are plainly outlawed. That doesn't mean that programmers using them are evil or incompetent. It does mean that the constructs are not essential and can be expressed just as well or even better with other language constructs.

If you already have programming experience, in Java or another language, you may be initially uncomfortable at giving up some fond habits. However, it is a sign of professionalism to set aside personal preferences in minor matters and to compromise for the benefit of your group.

These guidelines are necessarily somewhat dull. They also mention features that you may not yet have seen in class. Here are the most important highlights:

- Tabs are set every three spaces.
- Variable and method names are lowercase, with occasional upperCase characters in the middle.
- Class names start with an Uppercase letter.
- Constant names are UPPERCASE, with an occasional UNDER_SCORE.
- There are spaces after reserved words and surrounding binary operators.
- Braces must line up horizontally or vertically.
- No magic numbers may be used.
- Every method, except for `main` and overridden methods, must have a comment.
- At most 30 lines of code may be used per method.
- No `continue` or `break` is allowed.
- All non-`final` variables must be private.

Note to the instructor: Of course, many programmers and organizations have strong feelings about coding style. If this style guide is incompatible with your own preferences or with local custom, please feel free to modify it.

Source Files

Each Java program is a collection of one or more source files. The executable program is obtained by compiling these files. Organize the material in each file as follows:

- `package` statement, if appropriate
- `import` statements
- A comment explaining the purpose of this file
- A `public` class
- Other classes, if appropriate

The comment explaining the purpose of this file should be in the format recognized by the `javadoc` utility. Start with a `/**`, and use the `@author` and `@version` tags:

```
/**
   Classes to manipulate widgets.
   Solves CS101 homework assignment #3
   COPYRIGHT (C) 2015 Harry Morgan. All Rights Reserved.
   @author  Harry Morgan
   @version 1.01 2015-02-15
*/
```

Classes

Each class should be preceded by a class comment explaining the purpose of the class.

First list all public features, then all private features.

Within the public and private sections, use the following order:

1. Instance variables
2. Static variables
3. Constructors
4. Instance methods
5. Static methods
6. Inner classes

Leave a blank line after every method.

All non-`final` variables must be private. (However, instance variables of a private inner class may be public.) Methods and final variables can be either public or private, as appropriate.

All features must be tagged `public` or `private`. Do not use the default visibility (that is, package visibility) or the `protected` attribute.

Avoid static variables (except `final` ones) whenever possible. In the rare instance that you need static variables, you are permitted one static variable per class.

Methods

Every method (except for `main`) starts with a comment in `javadoc` format.

```
/**
   Convert calendar date into Julian day.
   Note: This algorithm is from Press et al., Numerical Recipes
   in C, 2nd ed., Cambridge University Press, 1992.
   @param day  day of the date to be converted
   @param month  month of the date to be converted
   @param year  year of the date to be converted
   @return  the Julian day number that begins at noon of the
   given calendar date.
*/
public static int getJulianDayNumber(int day, int month, int year)
{
   . . .
}
```

Parameter variable names must be explicit, especially if they are integers or Boolean:

```
public Employee remove(int d, double s)
   // Huh?
public Employee remove(int department, double severancePay)
   // OK
```

Methods must have at most 30 lines of code. The method signature, comments, blank lines, and lines containing only braces are not included in this count. This rule forces you to break up complex computations into separate methods.

Variables and Constants

Do not define all variables at the beginning of a block:

```
{
   double xold; // Don't
   double xnew;
   boolean done;
   . . .
}
```

Define each variable just before it is used for the first time:

```
{
   . . .
   double xold = Integer.parseInt(input);
   boolean done = false;
   while (!done)
   {
      double xnew = (xold + a / xold) / 2;
      . . .
   }
   . . .
}
```

Do not define two variables on the same line:

```
int dimes = 0, nickels = 0; // Don't
```

Instead, use two separate definitions:

```
int dimes = 0; // OK
int nickels = 0;
```

In Java, constants must be defined with the reserved word `final`. If the constant is used by multiple methods, declare it as `static final`. It is a good idea to define static final variables as `private` if no other class has an interest in them.

Do not use magic numbers! A magic number is a numeric constant embedded in code, without a constant definition. Any number except –1, 0, 1, and 2 is considered magic:

```
if (p.getX() < 300) // Don't
```

Use `final` variables instead:

```
final double WINDOW_WIDTH = 300;
. . .
if (p.getX() < WINDOW_WIDTH) // OK
```

Even the most reasonable cosmic constant is going to change one day. You think there are 365 days per year? Your customers on Mars are going to be pretty unhappy about your silly prejudice. Make a constant

```
public static final int DAYS_PER_YEAR = 365;
```

so that you can easily produce a Martian version without trying to find all the 365s, 364s, 366s, 367s, and so on, in your code.

When declaring array variables, group the `[]` with the type, not the variable.

```
int[] values; // OK
int values[]; // Ugh—this is an ugly holdover from C
```

When using collections, use type parameters and not "raw" types.

```
ArrayList<String> names = new ArrayList<String>(); // OK
ArrayList names = new ArrayList(); // Not OK
```

Control Flow

Statement Bodies

Use braces to enclose the bodies of branch and loop statements, even if they contain only a single statement. For example,

```
if (x < 0)
{
   x++;
}
```

and not

```
if (x < 0)
   x++; // Not OK--no braces
```

The for Statement

Use `for` loops only when a variable runs from somewhere to somewhere with some constant increment/decrement:

```
for (int i = 0; i < a.length; i++)
{
   System.out.println(a[i]);
}
```

Or, even better, use the enhanced `for` loop:

```
for (int e : a)
{
   System.out.println(e);
}
```

Do not use the `for` loop for weird constructs such as

```
for (a = a / 2; count < ITERATIONS; System.out.println(xnew))   // Don't
```

Make such a loop into a `while` loop. That way, the sequence of instructions is much clearer:

```
a = a / 2;
while (count < ITERATIONS) // OK
{
   . . .
   System.out.println(xnew);
}
```

Nonlinear Control Flow

Avoid the `switch` statement, because it is easy to fall through accidentally to an unwanted case. Use `if/else` instead.

Avoid the `break` or `continue` statements. Use another `boolean` variable to control the execution flow.

Exceptions

Do not tag a method with an overly general exception specification:

```
Widget readWidget(Reader in) throws Exception // Bad
```

Instead, specifically declare any checked exceptions that your method may throw:

```
Widget readWidget(Reader in)
      throws IOException, MalformedWidgetException // Good
```

Do not "squelch" exceptions:

```
try
{
   double price = in.readDouble();
}
catch (Exception e)
{ } // Bad
```

Beginners often make this mistake "to keep the compiler happy". If the current method is not appropriate for handling the exception, simply use a `throws` specification and let one of its callers handle it.

Lexical Issues

Naming Conventions

The following rules specify when to use upper- and lowercase letters in identifier names:

- All variable and method names are in lowercase (maybe with an occasional upperCase in the middle); for example, `firstPlayer`.
- All constants are in uppercase (maybe with an occasional UNDER_SCORE); for example, `CLOCK_RADIUS`.
- All class and interface names start with uppercase and are followed by lowercase letters (maybe with an occasional UpperCase letter); for example, `BankTeller`.
- Generic type variables are in uppercase, usually a single letter.

Names must be reasonably long and descriptive. Use `firstPlayer` instead of `fp`. No drppng f vwls. Local variables that are fairly routine can be short (`ch`, `i`) as long as they are really just boring holders for an input character, a loop counter, and so on. Also, do not use `ctr`, `c`, `cntr`, `cnt`, `c2` for variables in your method. Surely these variables all have specific purposes and can be named to remind the reader of them (for example, `current`, `next`, `previous`, `result`, . . .). However, it is customary to use single-letter names, such as `T` or `E` for generic types.

Indentation and White Space

Use tab stops every three columns. That means you will need to change the tab stop setting in your editor!

Use blank lines freely to separate parts of a method that are logically distinct.

Use a blank space around every binary operator:

```
x1 = (-b - Math.sqrt(b * b - 4 * a * c)) / (2 * a);
// Good

x1=(-b-Math.sqrt(b*b-4*a*c))/(2*a);
// Bad
```

Leave a blank space after (and not before) each comma or semicolon. Do not leave a space before or after a parenthesis or bracket in an expression. Leave spaces around the `( . . . )` part of an `if`, `while`, `for`, or `catch` statement.

```
if (x == 0) { y = 0; }

f(a, b[i]);
```

Every line must fit in 80 columns. If you must break a statement, add an indentation level for the continuation:

```
a[n] = ..................................................
     + .................;
```

Start the indented line with an operator (if possible).

Braces

Opening and closing braces must line up, either horizontally or vertically:

```
while (i < n) { System.out.println(a[i]); i++; }

while (i < n)
{
   System.out.println(a[i]);
   i++;
}
```

Some programmers don't line up vertical braces but place the `{` behind the reserved word:

```
while (i < n) { // DON'T
   System.out.println(a[i]);
   i++;
}
```

Doing so makes it hard to check that the braces match.

Unstable Layout

Some programmers take great pride in lining up certain columns in their code:

```
firstRecord = other.firstRecord;
lastRecord  = other.lastRecord;
cutoff      = other.cutoff;
```

This is undeniably neat, but the layout is not stable under change. A new variable name that is longer than the preallotted number of columns requires that you move all entries around:

```
firstRecord = other.firstRecord;
lastRecord  = other.lastRecord;
cutoff      = other.cutoff;
marginalFudgeFactor = other.marginalFudgeFactor;
```

This is just the kind of trap that makes you decide to use a short variable name like `mff` instead. Use a simple layout that is easy to maintain as your programs change.

APPENDIX J

HTML SUMMARY

A Brief Introduction to HTML

A web page is written in a language called HTML (Hypertext Markup Language). Like Java code, HTML code is made up of text that follows certain strict rules. When a browser reads a web page, the browser *interprets* the code and *renders* the page, displaying characters, fonts, paragraphs, tables, and images.

HTML files are made up of text and *tags* that tell the browser how to render the text. Nowadays, there are dozens of HTML tags—see Table 1 for a summary of the most important tags. Fortunately, you need only a few to get started. Most HTML tags come in pairs consisting of an opening tag and a closing tag, and each pair applies to the text between the two tags. Here is a typical example of a tag pair:

```
Java is an <i>object-oriented</i> programming language.
```

The tag pair `<i> </i>` directs the browser to display the text inside the tags in *italics*:

Java is an *object-oriented* programming language.

The closing tag is just like the opening tag, but it is prefixed by a slash (/). For example, bold-faced text is delimited by `<b> </b>`, and a paragraph is delimited by `<p> </p>`.

```
<p><b>Java</b> is an <i>object-oriented</i> programming language.</p>
```

The result is the paragraph

Java is an *object-oriented* programming language.

Another common construct is a bulleted list. For example:

Java is

- object-oriented
- safe
- platform-independent

Here is the HTML code to display it:

```
<p>Java is</p>
<ul><li>object-oriented</li>
<li>safe</li>
<li>platform-independent</li></ul>
```

Each item in the list is delimited by `<li> </li>` (for "list item"), and the whole list is surrounded by `<ul> </ul>` (for "unnumbered list").

Table 1 Selected HTML Tags

Tag	Meaning	Children	Commonly Used Attributes
`html`	HTML document	`head, body`	
`head`	Head of an HTML document	`title`	
`title`	Title of an HTML document		
`body`	Body of an HTML document		
`h1 . . . h6`	Heading level 1 . . . 6		
`p`	Paragraph		
`ul`	Unnumbered list	`li`	
`ol`	Ordered list	`li`	
`dl`	Definition list	`dt, dd`	
`li`	List item		
`dt`	Term to be defined		
`dd`	Definition data		
`table`	Table	`tr`	
`tr`	Table row	`th, td`	
`th`	Table header cell		
`td`	Table cell data		
`a`	Anchor		`href, name`
`img`	Image		`src, width, height`
`pre`	Preformatted text		
`hr`	Horizontal rule		
`br`	Line break		
`i` or `em`	Italic		
`b` or `strong`	Bold		
`tt` or `code`	Typewriter or code font		
`s` or `strike`	Strike through		
`u`	Underline		
`super`	Superscript		
`sub`	Subscript		
`form`	Form		`action, method`

Table 1 Selected HTML Tags

Tag	Meaning	Children	Commonly Used Attributes
`input`	Input field		`type, name, value, size, checked`
`select`	Combo box style selector	`option`	`name`
`option`	Option for selection		
`textarea`	Multiline text area		`name, rows, cols`

As in Java, you can freely use white space (spaces and line breaks) in HTML code to make it easier to read. For example, you can lay out the code for a list as follows:

```
<p>Java is</p>
<ul>
<li>object-oriented</li>
<li>safe</li>
<li>platform-independent</li>
</ul>
```

The browser ignores the white space.

If you omit a tag (such as a `</li>`), most browsers will try to guess the missing tags—sometimes with differing results. It is always best to include all tags.

You can include images in your web pages with the `img` tag. In its simplest form, an image tag has the form

```
<img src="hamster.jpeg"/>
```

This code tells the browser to load and display the image that is stored in the file `hamster.jpeg`. This is a slightly different type of tag. Rather than text inside a tag pair `<img> </img>`, the `img` tag uses an attribute to specify a file name. Attributes have names and values. For example, the `src` attribute has the value `"hamster.jpeg"`. Table 2 contains commonly used attributes.

Table 2 Selected HTML Attributes

Attribute	Description	Commonly Contained in
`name`	Name of form element or anchor	`input, select, textarea, a`
`href`	Hyperlink reference	`a`
`src`	Source (as of an image)	`img`
`code`	Applet code	`applet`
`width, height`	Width, height of image or applet	`img, applet`
`rows, cols`	Rows, columns of text area	`textarea`
`type`	Type of input field, such as `text`, `password`, `checkbox`, `radio`, `submit`, `hidden`	`input`
`value`	Value of input field, or label of submit button	`input`

Table 2 Selected HTML Attributes

Attribute	Description	Commonly Contained in
`size`	Size of text field	`input`
`checked`	Check radio button or checkbox	`input`
`action`	URL of form action	`form`
`method`	`GET` or `POST`	`form`

It is considered polite to use several additional attributes with the `img` tag, namely the *image size* and an *alternate description*:

```
<img src="hamster.jpeg" width="640" height="480"
alt="A photo of Harry, the Horrible Hamster"/>
```

These additional attributes help the browser lay out the page and display a temporary description while gathering the data for the image (or if the browser cannot display images, such as a voice browser for blind users). Users with slow network connections really appreciate this extra effort.

Because there is no closing `</img>` tag, we put a slash `/` before the closing `>`. This is not a requirement of HTML, but it is a requirement of the emerging XHTML standard, the XML-based successor to HTML. See `www.w3c.org/TR/xhtml1` for more information on XHTML.

The most important tag on a web page is the `<a> </a>` tag pair, which makes the enclosed text into a *link* to another file. The links between web pages are what makes the Web into, well, a web. The browser displays a link in a special way (for example, underlined text in blue color). Here is the code for a typical link:

```
<a href="http://horstmann.com">Cay Horstmann</a> is the author of this book.
```

When the viewer of the web page clicks on the words Cay Horstmann, the browser loads the web page located at `horstmann.com`. (The value of the `href` attribute is a *Universal Resource Locator* (URL), which tells the browser where to go. The prefix `http:`, for *Hypertext Transfer Protocol*, tells the browser to fetch the file as a web page. Other protocols allow different actions, such as `ftp:` to download a file, `mailto:` to send e-mail to a user, and `file:` to view a local HTML file.)

Table 3 Selected HTML Entities

Entity	Description	Appearance
`<`	Less than	<
`>`	Greater than	>
`&`	Ampersand	&
`&quot;`	Quotation mark	"
` `	Nonbreaking space	
`&copy;`	Copyright symbol	©

You have noticed that tags are enclosed in angle brackets (less-than and greater-than signs). What if you want to show an angle bracket on a web page? HTML provides the notations `<` and `>` to produce the < and > symbols, respectively. Other codes of this kind produce symbols such as accented letters. The & (ampersand) symbol introduces these codes; to get that symbol itself, use `&`. See Table 3 for a summary.

You may already have created web pages with a web editor that works like a word processor, giving you a WYSIWYG (what you see is what you get) view of your web page. But the tags are still there, and you can see them when you load the HTML file into a text editor. If you are comfortable using a WYSIWYG web editor, you don't need to memorize HTML tags at all. But many programmers and professional web designers prefer to work directly with the tags at least some of the time, because it gives them more control over their pages.