Appendix I

Java Language Definition

This section defines the Java subset used in this text. (Consult *The Java Language Specification*, by J. Gosling, B. Joy, and G. Steele, Addison Wesley Publishing, 1996, for the definition of the full Java language.)

A language is defined by its syntax (its spelling and grammar) and its semantics (what its sentences mean). We will define syntax with the grammatical notation explained in the next section, and we will give semantics by means of English explanation and examples. Also, references will appear to the relevant sections of the text.

Grammatical Notation

The syntax of a computer language are defined by a set of grammar rules or "equations" that define the phrases' precise appearance. An example best introduces this format.

Using an English description, we might define a *number* to appear as a whole-number part followed by an optional fraction part. Both whole-number and fraction parts consist of nonempty sequences of numerals; the fraction part is preceded by a decimal point. These grammar rules formalize the description:

```
NUMBER ::=  WHOLE_PART  FRACTION_PART?
WHOLE_PART ::=  NUMERAL_SEQUENCE
FRACTION_PART ::=  . NUMERAL_SEQUENCE
NUMERAL_SEQUENCE ::=  NUMERAL  NUMERAL*
NUMERAL ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

The words in upper-case letters name the phrase forms. Thus, the first rule tells us that a `NUMBER` phrase consists of a `WHOLE_PART` followed by a `FRACTION_PART`; the question mark states that the `FRACTION_PART` is optional. The second and third rules have obvious readings; note the decimal point in the third rule.

The asterisk in the fourth rule stands for "zero or more"—a `NUMERAL_SEQUENCE` consists of one `NUMERAL` followed by zero or more additional `NUMERAL`s. The vertical bars in the last rule are read as "or." Unless stated otherwise, appearances of `?`, `*`, and `|` in grammar rules will stand for the notions just described.

For example, `123.4` is a `NUMBER` because `123`, the `WHOLE_PART`, is a `NUMERAL_SEQUENCE` and because `.4` is a `FRACTION_PART`, where `4` is a `NUMERAL_SEQUENCE`.

For conciseness, the first three grammar rules can be compressed into just one:

```
NUMBER ::=  NUMERAL_SEQUENCE  [[ . NUMERAL_SEQUENCE ]]?
```

The double-bracket pairs, `[[` and `]]`, enclose a phrase; because of the question mark suffixed to it, the entire phrase is optional. We also use double brackets to enclose phrases that can be repeated zero or more times, e.g.,

```
NUMBER_LIST ::=  NUMBER  [[ , NUMBER ]]*
```

This rule defines the syntax of a list of numbers separated by commas, e.g., `1, 2.34, 56.7` is a `NUMBER_LIST` as is merely `89`.

Double brackets can be used to enclose alternatives, e.g., this grammar rule defines exactly the two words, "cat" and "cut":

```
WORDS ::=  c [[ a | u ]] t
```

As the examples show, spaces within the grammar rules do *not* imply that spaces are required within the phrases defined by the rules. For example, we normally write a `NUMBER`, like `56.7`, with no internal spaces. We follow this convention in the sections that follow.

Java Definition

We now define the Java subset used in this text. For exposition, we present the language one construct at a time. Each construct is defined by

- one or more grammar rules, which define syntax;
- an explanation of the grammar rules and restrictions (e.g., data-typing), if any on the phrases they define;
- one or more examples of Java phrases that are defined by the syntax;
- a summary of the construction's semantics.

Program Unit

```
PROGRAM_UNIT ::=  [[ package IDENTIFIER ; ]]?
                  IMPORT*
                  [[ CLASS | INTERFACE ]]
IMPORT ::=  import PACKAGE_NAME ;
PACKAGE_NAME ::=  IDENTIFIER  [[ . IDENTIFIER ]]* [[ . * ]]? ;
```

A Java program unit begins with an optional package affiliation, followed by zero or more `import` statements, followed by either a class definition or an interface definition. (*Important:* the asterisk in the phrase, `[[ . * ]]`, is indeed an asterisk to be included in the package name; it is *not* indicating a repetition.) An `IDENTIFIER` is a name, as defined in "What is an Identifier?" in Chapter 3. `CLASS` and `INTERFACE` are defined below.

Example:

```
package MyGraphicsFrames;
import java.util.GregorianCalendar;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public abstract class MyFrame extends JFrame
{ ... }
```

Semantics: A package affiliation groups a Java unit with others in the same-named package (see "Packages" in Chapter 9). The `import` statements make available the classes contained in the imported packages (see Figure 4 and "Java Packages" in Chapter 2). When the `. *` suffix is omitted, then the `PACKAGE_NAME` must name a specific class in a specific package, and it is only this one class that is available for use.

Class

```
CLASS ::=  public [[ abstract | final ]]?
           class IDENTIFIER EXTEND? IMPLEMENT*
           { FIELD*
             CONSTRUCTOR*
             METHOD*
           }
EXTEND ::=  extends IDENTIFIER
IMPLEMENT ::=  implements IDENTIFIER_LIST
IDENTIFIER_LIST ::=  IDENTIFIER [[ , IDENTIFIER ]]*
```

As indicated by the grammar rule, the keywords, `abstract` and `final` are exclusive and optional. The identifier mentioned in `EXTEND` must be the name of a class; the identifiers in `IMPLEMENT` must be names of interfaces. `FIELD`, `CONSTRUCTOR`, and `METHOD` are defined in subsequent sections.

Example:

```
public abstract class MyFrame extends Frame implements MyInterface
{ private int i = 2;
    ...
  public MyFrame() { ... }
    ...
  public void f() { i = i + 1; }
  public void paint(Graphics g) { ... }
  public abstract int h(GregorianCalendar g);
    ...
}
```

Semantics: A class defines a "blueprint" from which objects are constructed. Chapter 2 introduces classes; see also "Revised Syntax and Semantics of Classes" at the end of Chapter 5.

An abstract class lacks codings for some of its methods, and objects cannot be constructed directly from an abstract class (see "Abstract Classes" in Chapter 9)—another class must be written that extends the abstract class. A final class cannot be extended (see "final Components" in Chapter 9). A class can extend at most one other class but it can implement multiple interfaces.

Interface

```
INTERFACE ::=  public interface IDENTIFIER EXTEND?
               {  [[ public abstract? RETURN_TYPE METHOD_HEADER ; ]]* }
```

An interface contains zero or more header lines of methods. `RETURN_TYPE` and `METHOD_HEADER` are defined below in the sections, "Type" and "Constructor," respectively.

Example:

```
public interface MyInterface
{ public void f();
  public int h(GregorianCalendar g);
}
```

Semantics: An interface lists method names that a class might possess; a class that does so is said to *implement* the interface. See "Interfaces" in Chapter 9 for details and examples.

Field

```
FIELD ::=  VISIBILITY static? final? DECLARATION
VISIBILITY ::=  public | private | protected | (nothing)
```

`DECLARATION` is defined in a later section.

Examples:

```
private int i = 2;
public static final double ONE = 1.0;
private static boolean b;
private GregorianCalendar today = new GregorianCalendar();
```

Semantics: A field is a variable declaration that exists independently of the methods that use it. (See "Objects with State: Field Variables" and "Scope of Field Declarations" in Chapter 4.) A final field cannot not be altered in value once it is initialized. A static field can be used by static methods, like `main`. As illustrated in "Case Study: Card Games," in Chapter 8, a public static final field defines a constant name that can be used throughout a program.

The forms of visibility are

- `public`: the component can be referenced by all components of all classes in the program
- `private`: the component can be referenced by only those components that are defined in the same class as this component
- `protected`: the component can be referenced by components in subclasses of the class in which this component appears *and* by components in classes in the same package as the one in which this component appears
- (nothing): the component can be referenced by components in classes in the same package as the one in which this component appears

Public and private visibilities are used extensively in examples throughout Chapter 5. Fields are almost always labelled `private`. Protected visibility is used occasionally with subclasses—see "Subclasses and Method Overriding" and Figure 15 in Chapter 9 for an example. The last visibility, called *package visibility*, is not used in this text.

Constructor

```
CONSTRUCTOR ::=  public METHOD_HEADER
                 { [[ super ( ARGUMENT_LIST ) ; ]]?
                   STATEMENT*
                 }
METHOD_HEADER ::=  IDENTIFIER ( FORMALPARAM_LIST? )
FORMALPARAM_LIST ::=  FORMALPARAM [[ , FORMALPARAM ]]*
FORMALPARAM ::=  TYPE IDENTIFIER
```

A constructor method's name must be the same as the class in which the constructor is included. A class can have more than one constructor method, provided that each constructor is uniquely identified by the quantity and data types of its formal parameters. (Such a situation is called *overloading* the constructor.) The formal parameters in the `METHOD_HEADER` must be distinctly named.

If the class in which the constructor appears extends another class, then the first statement of the constructor can be an invocation of the constructor in the super class—the invocation is called `super`. `STATEMENT` is defined in a later section.

`ARGUMENT_LIST` is defined in the section, "Invocation." `TYPE` is defined below.

Examples:

```
public MyFrame()
{ setSize(200, 100);
  setVisible(true);
}

public MyFrame(int width, String title)
```

```
{ setTitle(title);
  setSize(width, width/2);
  setVisible(true);
}

public MyCustomButton(String label)
{ super(label);
  setBackground(Color.red);
}
```

Semantics: When an object is constructed from a class, the object is constructed in computer storage and the class's constructor method is executed. Chapter 5 introduces and uses extensively constructor methods; see "Revised Syntax and Semantics of Classes" at the end of that chapter for technical details. The semantics of executing a constructor method is the same as the semantics of executing a method, and the section on methods that follows should be studied.

Type

```
TYPE ::=  PRIMITIVE_TYPE | REFERENCE_TYPE
PRIMITIVE_TYPE ::=  boolean | byte | int | long | char | float | double
REFERENCE_TYPE ::=  IDENTIFIER | TYPE[]
RETURN_TYPE ::=  TYPE | void
```

Note that type names like `String`, `GregorianCalendar`, etc., are in fact identifiers.

Examples:

```
double[][]
MyFrame
MyInterface
```

Semantics: As explained in "Data Type Checking" in Chapter 3, types in Java are classified as *primitive* or *reference* types. Elements of the former are atomic, non-object values; the latter are objects. Every class and interface defines a data type; objects constructed from a class are the elements of that class's type. If a class implements an interface, the class's objects also belong to the interface's type.

Types are related by a subtyping relation; this is explained in Appendix II. A `RETURN_TYPE` is the type of value that a method may return as its result; `void` means "no result returned."

Method

```
METHOD ::=  VISIBILITY  [[ abstract | static ]]?  final?  RETURN_TYPE
            METHOD_HEADER  METHOD_BODY
METHOD_BODY ::=  { STATEMENT* }  |  ;
```

The method is labelled `abstract` if and only if the method's body consists of just a semicolon. A static method is invoked by other static methods, e.g., `main`. A final method cannot be overridden (see below). A method's formal parameters must have distinct names.

Examples:

```
private double sum(int i, double j)
{ return i + j; }

public abstract int h(GregorianCalendar g);

public static void main(String[] args)
{ new MyFrame(300, "My Test Frame"); }
```

Semantics: A method is *invoked* by a client, which supplies arguments to the formal parameters in the method's header. The arguments bind to the formal parameter names, and the statements in the method's body execute. See Chapter 5 for extensive examples and see "Formal Description of Methods" in Chapter 5 for a precise description of invocation, binding, and execution.

A class can have multiple methods with the same name, provided that each method is uniquely identified by the quantity and data types of its formal parameters. (Such a situation is called *overloading* the method name; see "Method Overloading" and "Semantics of Overloading" in Chapter 9 for details.)

If a class extends a superclass, then the class can contain a method whose name and formal parameters are identical to those of a method in the superclass. This is called *overriding* the method. See "Subclasses and Method Overriding" and "Semantics of Overriding" in Chapter 9 for extensive explanation.

Statement

```
STATMENT ::=  DECLARATION
              | RETURN
              | IF | SWITCH
              | WHILE | DOWHILE | FOR
              | TRY_CATCH
              | THROW
              | STATEMENT_EXPRESSION ;
              | { STATEMENT* }
              | ;
STATEMENT_EXPRESSION ::=  OBJECT_CONSTRUCTION | INVOCATION | ASSIGNMENT
```

The various statement forms are defined in the sections that immediately follow.

Examples and Semantics: Statements are executed in sequence, from "left to right." *Statement expressions* are statements that return results; a statement expression can be used where an expression is expected. Examples are

```
GregorianCalendar g = new GregorianCalendar();  // object construction
System.out.println(Math.abs(-2) + 3);  // invocation
int i;  int j = (i = 3) + 1;  // assignment: set i to 3 and then j to 4
```

The last usage of a statement expression is *not* encouraged.

Declaration

```
DECLARATION ::=  TYPE IDENTIFIER [[ = INITIAL_EXPRESSION ]]? ;
```

`INITIAL_EXPRESSION` is defined in "Expression"; its data type must be a subtype of the `TYPE` in the declaration.

Examples:

```
int i = 2 + 3;
GregorianCalendar[] days_in_this_month;
```

Semantics: A declaration creates a cell, named by the identifier, to hold values of the indicated data type. The cell receives the value computed from the initial expression (if any). The sections, "Declarations and Variables" "Local Variables and Scope," in Chapter 3 provide examples and details.

Return

```
RETURN ::=  return EXPRESSION ;
```

The statement can appear only in the body of a method that has a non-`void` return type stated in its header line; the data type of `EXPRESSION` must be a subtype of the return type. (See Section "Type" for the definition of `RETURN_TYPE`.)

Examples:

```
return 2 * sum(3, 4);
return new int[3];
return i;
```

Semantics: The return-statement calculates a value, which is immediately returned as the result of a method. See "Results from Methods" in Chapter 5.

Conditional

```
IF ::=  if ( EXPRESSION ) { STATEMENT* } [[ else { STATEMENT* } ]]?
SWITCH ::=  switch ( EXPRESSION )
            { [[ case EXPRESSION : { STATEMENT*  break ; } ]]*
              default : { STATEMENT* }
            }
```

The expression part of the if-statement must have data type `boolean`. The first expression of the switch-statement must have data type `integer`, `char`, or `byte`.

The second expression of the switch-statement must be an `integer`-, `char`-, or `byte`-typed *constant expression*, that is, an integer expression that is constructed from numeric constants, arithmetic operations, and fields that are static and final. Here is an example that uses a convoluted numeric constant:

```
public class Test
{ private static final int k = 2;

 public static void main(String[] args)
  { int x = 0;
    switch (x + 1)
           { case ((int)(k / 3.3) + 1) : { break; }
             default : { }
           }
  }
}
```

The Java compiler calculates the integer represented by the constant expression and replaces the expression by the integer. (Here, `(int)(k / 3.3) + 1` is replaced by 1.) In practice, constant expressions are merely integer and character constants, e.g., `5` and `'a'`.

Additional Example:

```
if ( i > 0  &&  i != 2 )
     { j = sum(i, 2.2); }
else { if ( b )
          { j = 0; }
     }
```

Semantics: The appropriate arm of an if- or switch-statement is selected based upon the value of the test expression. Chapter 6 presents numerous examples of if-statements; see "The Switch Statement" at the end of that chapter for examples of the switch-statement.

Iteration

```
WHILE ::=  while ( EXPRESSION ) { STATEMENT* }
DOWHILE ::=  do { STATEMENT* } while ( EXPRESSION ) ;
FOR ::=  for ( [[ DECLARATION | ASSIGNMENT ; ]]
               EXPRESSION ;
               STATEMENT_EXPRESSION )
             { STATEMENT* }
```

The test expression for each of the three loops must be typed `boolean`. Although the for-loop accepts a `STATEMENT_EXPRESSION` as the third part of its header line, in practice this is an assignment statement. Alas, there is no restriction regarding the variables declared, referenced, and assigned in the three parts of the for-loop's header line.

Examples:

```
for ( int i = 0;  i != r.length;  i = i + 1 )
    { while ( r[i] > 0 )
            { r[i] = r[i] - 2; }
    }
```

Semantics: As long as the test expression evaluates to true, the body of a loop repeats. Chapter 7 displays numerous examples. The do-while-loop is a minor variation of the while-loop—the body executes once before the loop's test is computed; see "The do-while loop" in Chapter 7.

The for-loop executes its declaration-assignment on entry; next, the test expression is computed, and as long as the text computes to true, the loop's body is executed, followed by the statement-expression in the loop's header line. For loops are introduced in "The for-loop" in Chapter 7 and are extensively used with arrays in Chapter 8.

Exception Handler

```
TRY_CATCH ::=  try { STATEMENT* } CATCH_LIST
CATCH_LIST ::= CATCH CATCH*
CATCH ::=  catch ( TYPE IDENTIFIER ) { STATEMENT* }
THROW ::=  throw EXPRESSION ;
```

The data type listed in the `CATCH` clause of an exception handler and the data type of the `EXPRESSION` in the `THROW` statement must be a subtype of `Throwable`.

Example:

```
try { if ( i != 0 )
           { j = j / i; }
      else { throw new RuntimeException("i is zero"); }
    }
catch (RuntimeException e)
      { System.out.println(e.toString());
        j = 0;
      }
```

Semantics: The statements in the `try` clause of an exception handler are executed as usual, unless a `throw` statement is executed. The `throw` aborts usual execution and forces an immediate search of the clauses in the `CATCH_LIST` for the first `catch`

clause whose data type is a subtype of the type of the exception object listed in the `throw` statement. If a matching `catch` clause is found, the exception binds to its formal parameter, the clause's statements are executed. If no matching `catch` clause is found, search for a matching clause proceeds at the next enclosing exception handler, *as found by search the prior sequence of statements that were executed.*

See "Exceptions Are Objects" in Chapter 11 for details.

Object Construction

```
OBJECT_CONSTRUCTION ::=  new IDENTIFIER ( ARGUMENT_LIST? )
                       | new ARRAY_ELEMENT_TYPE DIMENSIONS
ARGUMENT_LIST ::=  EXPRESSION [[ , EXPRESSION ]]*
ARRAY_ELEMENT_TYPE ::=  PRIMITIVE_TYPE | IDENTIFIER
DIMENSIONS ::=  [ EXPRESSION ]  [[  [ EXPRESSION ]  ]]* [[  []  ]]*
```

Objects are constructed in two forms: individual objects and array objects. For constructing the former, the identifier must be a class name, and its argument list must contain the same quantity of parameters as (one of) the constructor methods included in the class; the data type of each argument must be a subtype of the data type listed with the corresponding formal parameter in the constructor method.

Examples:

```
new GregorianCalendar()
new MyFrame(300, "")
```

When array objects are constructed, the data type of the array's individual elements is listed first, followed by the dimensions. The individual elements can be of primitive type or a type defined by a class name. The dimensions are indicated by pairs of brackets, where the size of each dimension is indicated by an expression embedded within a bracket pair. The quantity of at least the first dimension must be indicated.

Examples:

```
new int[4]
new GregorianCalendar[12][]
new double[size][size * size][][]
```

Semantics: The statement creates an object from `class IDENTIFIER` (or an array that holds values of the array-element type). Examples pervade the text—see "An Application that Creates an Object" in Chapter 2 for a simple one, see "Revised Syntax and Semantics of Classes" in Chapter 5 for details about object creation, and see "Formal Description of Arrays" at the end of Chapter 8 for examples of array object construction. The class's constructor method whose formal parameters best match the argument list is executed; see "Formal Description of Methods" in Chapter 5 for details about the matching process.

Method Invocation

```
INVOCATION ::=  [[ RECEIVER . ]]? IDENTIFIER ( ARGUMENT_LIST? )
RECEIVER ::=  this | super
            | IDENTIFIER
            | RECEIVER [ EXPRESSION ]
            | RECEIVER . IDENTIFIER
            | STATEMENT_EXPRESSION
            | ( RECEIVER )
```

The identifier mentioned in the `INVOCATION` rule is a method name; the identifier in `RECEIVER` can be either a variable name whose data type is a reference type (that is, the variable holds the address of an object) or it can be the name of a class, which occurs when a static method is invoked; see below.

The argument list must contain the same quantity of parameters as the number of formal parameters listed in the header line of the invoked method, `IDENTIFIER`. The data type of each argument must be a subtype of the data type listed with the corresponding formal parameter.

Examples and Semantics: To send a message (that is, to invoke a method), the (address of the) object where the method lives must be determined—the object is the "receiver" of the message. If the `RECEIVER` is omitted from the invocation, then the receiver is the same object where the invocation is situated. Private methods are usually invoked this way:

```
public class C
{ private static double sum(int i, double j)
  { return i + j; }

  public static void main(String[] args)
  { System.out.println( sum(2, 3.4) ); }
}
```

Public methods can be invoked this way, also:

```
import javax.swing.*;
public class MyFrame extends JFrame
{ public MyFrame()
  { setTitle("MyFrame");
    f("MyFrame");
    this.f("MyFrame");
  }

  public void f(String label)
  { this.setTitle(label); }

   ...
}
```

All invocations in the example ultimately invoke `setTitle` in the superclass, `JFrame`; see "Customizing Frames with Inheritance" in Chapter 4. The semantics of `this` is the same as an omitted `RECEIVER`.

`super` asserts that the receiver should be the same object as the invocation's but the invoked method must be selected from the superclass from which the object is built:

```
import javax.swing.*;
public class MyFrame extends JFrame
{ public MyFrame()
  { super.setTitle("MyFrame"); }

  public setTitle(String s)
  { }
}
```

invokes the `setTitle` method located in `class JFrame` and ignores the one in `MyFrame`. See Figure 16, Chapter 9, for a significant example.

When an identifier defines the receiver, the identifier is usually a variable that holds the address of an object, as `g` does here:

```
GregorianCalendar g = new GregorianCalendar();
... g.getDate(); ...
```

But the identifier can also be a class name when the invoked method is static. Here is an example:

```
public class C
{ public static int one() { return 1; } }

public class D
{ public static void main(String[] args)
  { System.out.println( 2 * C.one() ); }
}
```

Table 9 of Chapter 3 lists a variety of static methods that are invoked with the receiver, `Math`, e.g., `Math.sqrt(2)`.

A receiver can be indexed if it names an array object, e.g.,

```
GregorianCalendar[] dates = new GregorianCalendar[3];
  ...
System.out.println( dates[0].getTime() );
```

Similarly, an object that contains a public field that holds an object can be indexed by an identifier:

```
public class C
{ public GregorianCalendar g = new GregorianCalendar();
   ...
}

public class D
{  ...
  C x = new C();
  ...  x.g.getTime() ...
}
```

Finally, a statement expression can define a receiver. This happens when an object is constructed anew, e.g., `new GregorianCalendar().getTime()`, and when a method invocation returns an address of an object as its result:

```
public class C
{ public GregorianCalendar f() { return new GregorianCalendar(); }
   ...
}

public class D
{ ... (new C().f()).getTime() ...  }
```

Here, `new C().f()` is the invocation that defines the receiver for the invocation of `getTime`.

Once the receiver is established, the method within the receiver is selected. (Note the impact of `super` on this selection.) Next, the arguments are evaluated, bound to the formal parameters of the selected method, and the selected method executes. Precise descriptions of these steps are found in "Formal Description of Methods" in Chapter 5, "Semantics of Overloading" in Chapter 9, and "Semantics of Overriding" also in Chapter 9.

Examples of method invocation abound in the text; Chapter 5 is devoted to this one topic.

Assignment

```
ASSIGNMENT :=  VARIABLE = EXPRESSION
VARIABLE ::=  IDENTIFIER
              | RECEIVER [ EXPRESSION ]
              | RECEIVER . IDENTIFIER
```

The data type of the assignment's expression must be a subtype of the variable's data type.

Examples and Semantics: An assignment places the value of its right-hand side expression into the cell named by the left-hand side variable. Variables are usually identifiers, e.g., `x` in `int x; x = 0`.

A variable can be the element of an array, as is `r[2]` in `int[] r = new int[3]; r[2] = 0`. In this case, the address of the array object, `r` is first computed as a `RECEIVER`, and the address is used with the index. (See the explanation of `RECEIVER` in the previous section.)

If an object possesses public fields, the variable part of an assignment can be a reference to one of those fields:

```
public class C
{ public int x; }

public class D
{ ...  C c = new C();
       c.x = 1;  ...
}
```

Expression

```
EXPRESSION ::=  LITERAL
                | VARIABLE
                | EXPRESSION INFIX_OPERATOR EXPRESSION
                | PREFIX_OPERATOR EXPRESSION
                | ( EXPRESSION )
                | ( TYPE ) EXPRESSION
                | EXPRESSION instanceof REFERENCE_TYPE
                | this | null
                | STATEMENT_EXPRESSION
LITERAL ::=  BOOLEAN_LITERAL
             | INTEGER_LITERAL | LONG_LITERAL
             | FLOAT_LITERAL | DOUBLE_LITERAL
             | CHAR_LITERAL | STRING_LITERAL
INFIX_OPERATOR ::=   +  |  -  |  *  |  /  |  %
                     |  <  |  >  |  <=  |  >=  |  ==  |  !=
                     |  ||  |  &&
PREFIX_OPERATOR ::=  -  |  !
INITIAL_EXPRESSION ::=  EXPRESSION
                        | { [[ INITIAL_EXPRESSION_LIST ]]? }
INITIAL_EXPRESSION_LIST ::=  INITIAL_EXPRESSION [[ , INITIAL_EXPRESSION ]]*
```

As always, infix and prefix operators require arguments whose data types are acceptable to the operators, e.g., the multiplication operator, `*`, requires arguments of numeric data type. Literals are constants, e.g., `3` and `true`; see Table 2 of Chapter 3.

`INITIAL_EXPRESSION`s are used to initialize variable declarations; the set expressions are used to initialize array variables, e.g., `int[][] r = { {0,1}, {1,3,5} {},`

{1}}, constructs and initializes an array with 4 rows where each row has a different number of columns.

Semantics and Examples: As described in Chapter 3, expressions are computed from left to right, producing a result that can be a primitive value or the address of an object. Consider each of the clauses of the syntax definition for `EXPRESSION`:

- A `LITERAL`, e.g., `3`, `false`, or `"abc"`, computes to itself. (Exception: a `STRING_LITERAL` computes to the address of an object that holds a sequence of characters, but one can pretend the object and the character sequence are the "same.")

- A variable, e.g., `r`, computes to the value held in the variable's cell. This value can be a primitive value or (the address of) an object.

- An infix expression, e.g., `(-2 > 3 + (4 * 5)) || (r[i] == sum(j, j))` computes its operands from left to right, and operates on the two results, producing an answer; a prefix expression computes its operand's value and performs its operation.

- An expression can be parenthesized to indicate the order in which infix and prefix operations should be performed in compound expressions

- `(TYPE)EXPRESSSION`, a cast, computes its expression's value and attempts to perform the cast of the value into the indicated data type. (If the value is a primitive type, the cast may well change the internal representation of the value, e.g., `(int)(5/2)` forces 2.5 to 2. A cast on a value of reference type—an address—does not change the value itself but only the data type of the value, as it is understood by the Java compiler, e.g, `MyFrame f = new MyFrame(); (JFrame)f` leaves the underlying object, `f`, unaltered but treats it as a mere `JFrame`.

- `EXPRESSION instanceof REFERENCE_TYPE` computes the value of its expression part, which will be (the address of) an object. The data type embedded in the object is checked to see if it is a subtype of the `REFERENCE_TYPE`. For example, the conditional's test expression in

  ```
  Frame f = new MyFrame();
  if ( f instance MyFrame ) { ... }
  ```

  computes to true, because the data type within the object is `MyFrame` even as the Java compiler uses type `JFrame` for the values held in cell `f`.

- `this` computes to (the address of) the very object in which the expression is embedded; `null` is the "no value" value.

- Statement expressions are computed just like statements, but they return values that are their "results":

- The value of an `OBJECT_CONSTRUCTION` is (the address of) the newly constructed object.
- The value of an `INVOCATION` is the result returned by the invoked method.
- The value of an `ASSIGNMENT` is the value of the assignment's right-hand side expression, e.g., the value of `i = 2 * 3` is 6.

Literal and Identifier

The forms of literals are listed in Table 2, Chapter 3. Identifiers are defined by the section, "What is an Identifier?," at the end of Chapter 3.

Appendix II

Typing and Subtyping

There are two forms of data types in the Java language:

- *primitive types*, such as `int`, `double`, and `boolean`
- *reference (object) types*, such as `GregorianCalendar`, `int[]`, and `String`

Every `class` and `interface` definition defines a reference data type, e.g., `public class MyFrame extends JFrame` defines type `MyFrame`, and `public interface UpdateableView` defines type `UpdateableView`.

The Java compiler calculates data types to enforce compatibility in expression calculation, assignment, parameter binding, and results returned from methods. For example, given

```
int i = 200;
JFrame f = new JFrame();
f.setSize(i + 1, i / 2);
```

the compiler calculates data types of the numerals, variables, and expressions to validate *in advance of these statements' execution* that they will behave correctly:

- The initialization of `int` variable, `i`, will execute correctly because `200` has type `int`.
- When a `new JFrame()` object is constructed, its address can be saved in `f`'s cell, because the cell is prepared to hold addresses of `JFrame`s.
- Whatever value is held in `f`'s cell will be the address of a `JFrame` object and will possess a method named `setSize`, therefore the `setSize` message can be correctly sent to the object named by `f`.
- Whatever their computed values, `i + 1` and `i / 2` will be integers that are acceptable arguments to `setSize`.

Because of the compiler's work, none of the above checks must be repeated when the program executes.

Subtypes

The Java compiler allows some flexibility in its data type calculation—given a context where a value of data type, `T1`, is required, the Java compiler will allow a value whose data type, `T2`, is a *subtype* of `T1`. We write `T2 <= T1`. One simple example is

```
int i = 3;
double d = i * 2.5;
```

Although `i` is `int`-typed, it is allowed in the context, `double d = __ * 2.5`, because `int <= double`, that is, `int` is a subtype of `double`.

Another context is the binding of actual parameters to formal parameters: A method that expects an argument of type `T1` will execute correctly if it receives an argument of type `T2` such that `T2 <= T1` holds true. An example is

```
private void f(double x) { ... }
 ...
f(3);
```

Within the context, `f(__)`, a `double` is required, but `3` suffices because its data type is `int` and `int <= double`.

Subtypes prove crucial to object usage, because data types are embedded into objects when objects are constructed during program execution. Consider

```
Component f = new JFrame();
 ...
if ( f instanceof JFrame )
   { ((JFrame)f).setVisible(true); }
```

the `JFrame` object constructed in the first statement has an internal data type of `JFrame`, even as the Java compiler has calculated that `f` holds values of type `Component`. The assignment is approved by the compiler because `JFrame <= Component`; see Chapter 10.

When the Java compiler calculates data types, it does *not* execute the statements themselves, and when the compiler examines the above if-statement, it knows merely that `f` has data type `Component`; it cannot answer whether the object held by `f` is truly a `JFrame`.

Later, when the program is started and the if-statement executes, the `instanceof` operation examines the data type embedded within the object named by `f` to calculate the answer to the test. If the answer is true, the cast operation lets `f` be used as a `JFrame`-typed object.

Subtypes of Primitive Types

The primitive types used in this text are `boolean`, `char`, `byte`, `int`, `long`, `float`, and `double`; see Chapter 3. The numeric types in the above list are related by subtyping as follows:

```
byte  <=  int  <=  long  <=  float  <=  double
```

where `T1 <= T2` if `T1` lies no further to the right in the ordering than does `T2`. For example, `byte <= int`, `byte <= long`, and `int <= int`.

Subtypes of Reference Types

Reference types are related by subtyping according to the following rules. Given arbitrary reference types, `T1` and `T2`, we say that `T1 <= T2` when one of the following rules holds true:

- `T1` is the same name as `T2`
- `T1` is defined by `class T1 extends T2` or by `class T1 implements T2` or by `interface T1 extends T2`
- `T1` is an array type, `T11[]`, `T2` is an array type, `T21[]`, and we can use these rules to prove that `T11 <= T21`
- There is another type, `T3`, and we can use these rules to prove that `T1 <= T3` and `T3 <= T2`
- `T2` is `Object`

For example, from these definitions,

```
public interface I1 { ... }

public interface I2 extends I1 { ... }

public class C implements I1 { ... }

public class D extends C implements I2 { ... }
```

we can conclude, say, `D[] <= I1[]`, because

- `class D extends C`, hence, `D <= C`
- `class C implements I1`, hence, `C <= I1`
- `D <= I1`, because of the previous two clauses
- `D[] <= I1[]`, because of the previous clause

Yet a second way of concluding the same fact is by noting that `class D implements I2`, `interface I2 extends I1`, hence, `D <= I1` and then `D[] <= I1[]`.

Appendix III

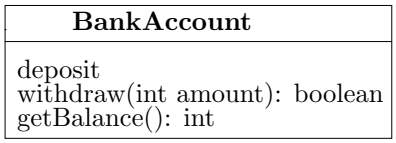
Class Diagrams and Hierarchies

Programs are designed and documented with *class diagrams*. A class diagram consists of a collection of components (names of classes and interfaces) connected by lines and arrows that represent collaborations and dependencies. The class diagrams used in this text are drawn in a subset of the diagram language of UML (*Universal Modelling Language*; see *UML Distilled*, by M. Fowler, Addison-Wesley, 1999, for a concise introduction.) Here is a summary of the diagram language.

Classes

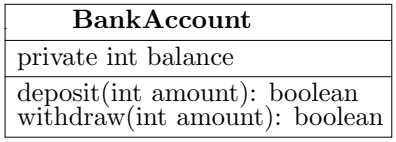
In its simplest representation, a class is drawn as a rectangle containing the class's name.

A class can be presented with some or all of its public methods:



A method can be specified with just its name, e.g., `deposit`, or it can be listed with its formal parameters and its result type, e.g., `withdraw(int amount): boolean`. Note that the result type is listed at the *right* of the method's specification, preceded by a colon. Usually, constructor methods are not listed with the class, but they can appear if the constructor or its parameters are important to understanding the class.

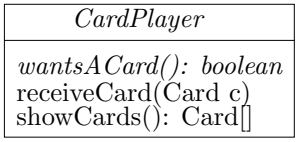
A class's attributes can be listed as well:



When an attribute is specified, one can assume that the class will have an accessor method for the attribute, e.g., `public int balanceOf() { return balance; }`. See Figure 12, Chapter 6, for this example.

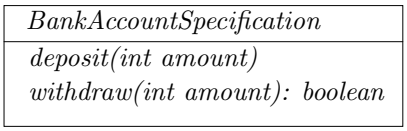
Objects can be that already exist prior to the execution of a program can be included in a class diagram as well:

An abstract class is specified by writing in italics the class's name and those methods that are abstract. (If you write the class diagram by hand, write the word, "abstract" in front of those words that should be in italics.)



¡/pre¿ See Figure 18, Chapter 9, for this example.

An interface is drawn like an abstract class—the name and all methods appear in italics:

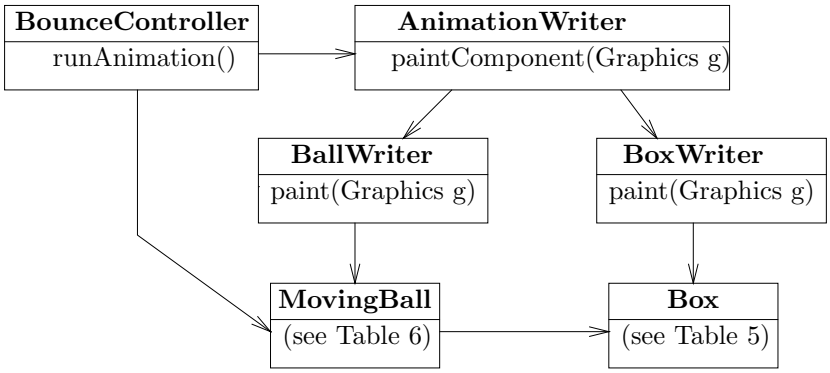


Such an arrangement might appear at an early stage of design, before specifics about methods and dependencies are decided.

An arrow from one component to another means that the first component *depends* on the second. By "depends," we mean some or all of the following:

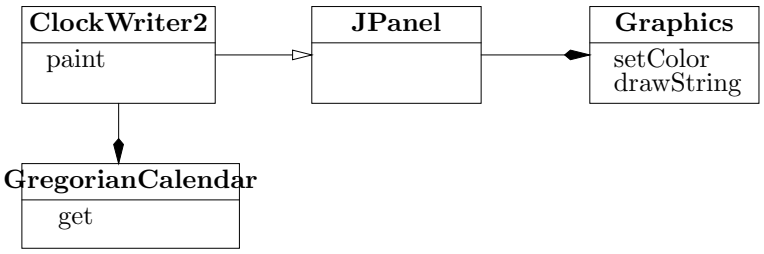
- the first component sends messages to the second
- the first component owns or holds the address (a "handle") to the second in a private field variable
- the first component cannot compile without the second

Here is an example from Chapter 7:



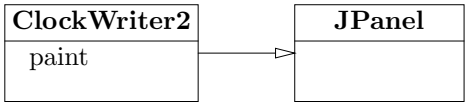
A stronger form of dependence is called *composition*, which defines the "has-a"

relationship:



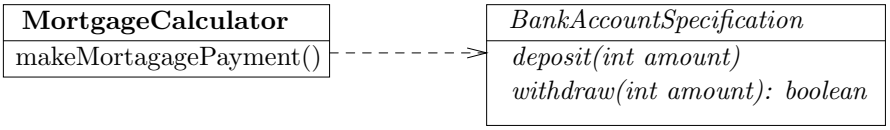
The large, black arrowheads in the diagram assert that every `JPanel` object will have its own unique `Graphics` object, and every `ClockWriter2` will have its own `GregorianCalendar`.

When one class extends another by means of inheritance, an arrow with a head is drawn from the subclass to the superclass:



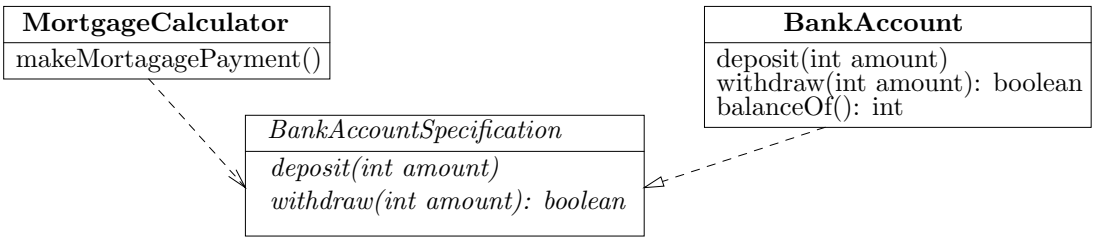
The arrow indicates that an object created from the subclass will contain the structure inherited from the superclass; in particular, the subclass's methods include methods listed in the superclass. Here, `ClockWriter2` is a `JPanel` extended by additional methods.

When a component depends on (refers to) an interface, a dotted arrow is drawn from the component to the interface:



Here, the coding of `MortgageCalculator` depends on `interface BankAccountSpecification`; see Chapter 9 for this example.

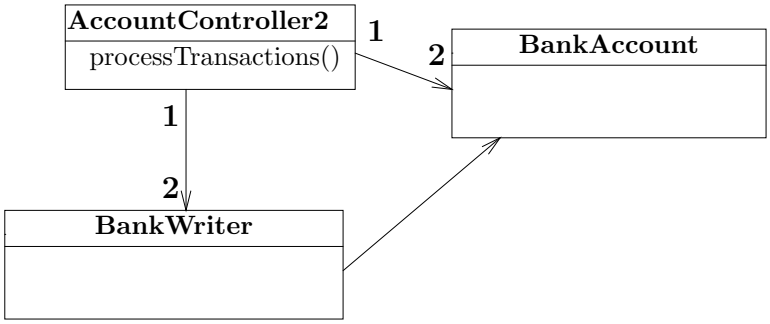
A dotted arrow with an arrow head is drawn from a class to an interface when the former implements the latter:



Here, `class BankAccount implements BankAccountSpecificaiton`.

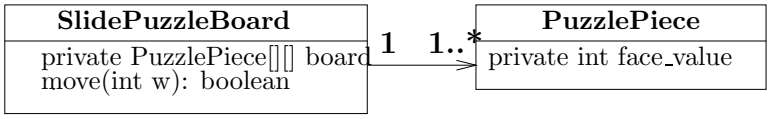
Annotations

The usual dependency between components is *one-to-one*, that is, if an arrow is drawn from component `A` to component `B`, it suggests that every one `A` object depends on one `B` object. To make explicit other quantities of dependency, we annotate the arrow with integers. For example, Figure 17 of Chapter 6 displayed the class diagram of an application whose controller manage two bank accounts and their output views:



The annotations on the arrow state that each one `AccountController` object depends upon (uses) two distinct `BankAccount` objects.

If there is a one-to-many dependency, but the exact quantity of the "many" is determined only when the application is built, an asterisk can be used in place of the integer. For example, the slide puzzle application in Chapter 8 is designed to construct slide puzzles with a varying quantity of movable puzzle pieces. Therefore, the puzzle board uses at least one and possibly many puzzle pieces:



Class Hierarchies

A package of classes that are related by inheritance is depicted in a form of class diagram that uses indentation in place of large arrow heads. For example, if `class Window extends Container`, `JFrame extends Window`, and `JApplet extends Container`, we might display the classes in this hierarchy:

```
Container
 |
 +-Window
 |  |
 |  +-JFrame
 |
 +-Japplet
```

Figure 5, Chapter 10, and the API Web pages for the Java packages display such class hierarchies.