# Appendix C
## Additional Java Syntax

This appendix briefly covers several features of Java that are not otherwise shown in the textbook. It is not intended to be a complete reference; see the Java Language Specification and Java Tutorial online for more detailed coverage of these language features.

## Primitive Types: `byte`, `short`, `long`, `float`

In this textbook we have focused our attention on four primitive data types: `int`, `double`, `char`, and `boolean`. Java has four additional primitive types that are used in certain special situations. Three of these additional types (`byte`, `short`, and `long`) are variants of `int` that use a different amount of memory, providing a tradeoff between memory consumption and the range of numbers that can be represented. The fourth, `float`, is a variant of `double` that uses half the memory. It can be useful in certain applications where reduced memory usage and faster computation is more important than extremely high numeric accuracy, such as in computer games.

| Type | Description | Range | Usage |
|---|---|---|---|
| `byte` | 8-bit (1-byte) integer | −128 to 127 | Reading data from an input source one byte at a time |
| `short` | 16-bit (2-byte) integer | −32,768 to 32,767 | Saving memory when creating many integers |
| `long` | 64-bit (8-byte) integer | $-2^{63}$ to $(2^{63} - 1)$ | Representing very large integers that may not fit into the range of `int` |
| `float` | 32-bit (4-byte) real number | roughly −3.4E+38 to 3.4E+38 | Saving memory when creating many real numbers |

Many of the operators and much of the syntax that you have used with `int` and `double` work with these types. Two of these types use a suffix letter at the end of their literal values: `F` for float, and `L` for `long`. The following code declares a variable of each type:

```
byte var1 = 63;
short var2 = -16000;
long var3 = 1234567890123456L;
float var4 = 1.2345F;
```

## Ternary Operator ? :

Java has a *ternary operator* that allows you to choose between two expressions based on the value of a `boolean` test. ("Ternary" means "having three sections.") Think of it as an abbreviated form of an `if/else` statement, except that an `if/else` chooses between two blocks of statements, while a ternary expression chooses between two expressions or values:

```
<test> ? <expression1> : <expression2>
```

A ternary expression is most useful when you want to assign a variable one of two values, or when you want to pass one of two values as a parameter or return value. For example:

```
// if d > 10.0, set x to 5;  else set x to 2
double d = ...;
int x = d > 10.0 ? 5 : 2;

// e.g. "I have 3 buddies" or "I have 1 buddy"
int pals = ...;
String msg = "I have " + pals + " " + (pals == 1 ? "buddy" :
                                       "buddies");

// Returns the larger of a and b
public static int max(int a, int b) {
    return (a > b) ? a : b;
}
```

## Exiting a Loop: break and continue

Java has a statement called `break` that will immediately exit a `while`, `do/while`, or `for` loop. One common usage of `break` is to write a loop that performs its exit test in the middle of each iteration rather than at the start or end. The common template is to form what appears to be an infinite loop:

```
while (true) {
    <statement>;
    ...
    if (<test>) {
        break;
    }
    <statement>;
    ...
}
```

Because the `boolean` literal `true` always evaluates to `true`, this loop appears to execute indefinitely. But a test occurs in the middle of the loop. This technique is useful for solving fencepost and sentinel problems. For example, Chapter 5 examines a sentinel problem for summing numbers until −1. This problem can be elegantly solved with `break` as follows:

```
Scanner console = new Scanner(System.in);
int sum = 0;
while (true) {
    System.out.print("next integer (-1 to quit)?");
    int number = console.nextInt();
    if (number == -1) {
        break;
    }
    sum += number;
}
System.out.println("sum = " + sum);
```

The `continue` statement immediately ends the current iteration of a loop. The code execution will return to the loop's header, which will perform the loop's test again (along with the update step, if it is a `for` loop). The following loop uses `continue` to avoid negative integers:

```
for (int i = 0; i < 100; i++) {
    System.out.print("type a nonnegative integer:");
    int number = console.nextInt();
    if (number < 0) {
        continue; // skip this number
    }
    sum += number;
    ...
}
```

Many programmers discourage use of the `break` and `continue` statements because these statements cause the code execution to jump from one place to another, which some people find nonintuitive. Any solution that uses `break` or `continue` can be rewritten to work without them.

It is possible to label statements and to `break` or `continue` executing from a particular label. This can be useful for immediately exiting to particular levels from a set of nested loops. This syntax is outside the scope of this textbook.

## The `switch` Statement

The `switch` statement is a control structure that is similar to the `if/else` statement. It chooses one of many paths ("cases") to execute on the basis of the value of a given variable or expression. It uses the following syntax:

```
switch (<expression>) {
    case <value>:
        <statements>;
        break;
    case <value>:
        <statements>;
        break;
    ...
    default:                // optional
        <statements>;
        break;
}
```

The <expression> used in a `switch` statement must be an integral type (`byte`, `short`, `char`, or `int`) or an enumerated type (`enum`, discussed later in this appendix). In Java version 7 and up, you may also switch on a `String` value.

The benefit of the `switch` syntax is that you do not need to repeat the `else if` syntax or the variable's name for each path; you simply write `case` plus the next value to test. If the value does not match any of the cases, none of them is executed. The following code prints a runner's medal; if the runner was not in first through third place, no message is printed.

```
Scanner console = new Scanner(System.in);
System.out.print("In what place did you finish the race? ");
int place = console.nextInt();

switch (place) {
    case 1:
        System.out.println("You won the gold medal!!!");
        break;
    case 2:
        System.out.println("You earned a silver medal!");
        break;
    case 3:
        System.out.println("You got a bronze medal.");
        break;
}
```

The optional `default` case holds code to execute if the expression's value does not match any of the other cases, as shown in the code that follows.

The tricky part of `switch` is remembering to write a `break` statement at the end of each case. If you omit the `break` at the end of a case, the code "falls through" into the next case and also executes its code. A programmer may cause this to happen intentionally, as in the following code, but often it is done by accident and leads to bugs:

```
switch (place) {
    case 1:  // give the same response for values 1–3
```

```
    case 2:
    case 3:
        System.out.println("You won a medal!!!");
        break;
    default:
        System.out.println("You did not win a medal. Sorry.");
        break;
}
```

Many programmers avoid the `switch` statement because it is so easy to produce a bug such as that just described. In older programming languages the `switch` statement was a more efficient way to execute the `if/else` statement, but this benefit is not noticeable in Java.

## The `try/catch` Statement

The `try/catch` statement "tries" to execute a given block of code (called the "`try` block"). The statement also specifies a second "`catch` block" of code that should be executed if any code in the "`try` block" generates an exception of a particular type. It uses the following syntax:

```
try {
    <statements>;
} catch (<type> <name>) {
    <statements>;
}
```

For example, the following code attempts to read an input file and prints an error message if the operation fails:

```
try {
    Scanner input = new Scanner(new File("input.txt"));
    while (input.hasNextLine()) {
        System.out.println(input.nextLine());
    }
} catch (FileNotFoundException e) {
    System.out.println("Error reading file: " + e);
}
```

If you wrap all potentially unsafe operations in a method with the `try/catch` syntax, you do not need to use a `throws` clause on that method's header. For example, you do not need to declare that your `main` method throws a `FileNotFoundException` if you handle it yourself using a `try/catch` block.

Some variations of the `try/catch` syntax are not shown here. It is possible to have multiple `catch` blocks for the same `try` block, to handle multiple kinds of

exceptions. It is also possible to add another block called a "`finally` block" that contains code to execute in all cases, whether an error occurs or not. This technique can be useful to consolidate cleanup code that should be run after the `try` block finishes. These syntax variations are outside the scope of this textbook.

## The `assert` Statement

In Chapters 4 and 5 we discussed preconditions, postconditions, and logical assertions. Sometimes programmers want to test logical assertions (Boolean expressions) as sanity checks in their own code. As the code runs, it will check each assertion that it reaches. If the assertion's expression is not true, the program will halt with an error.

Java supports the testing of assertions with its `assert` statement, which uses the following syntax:

```
assert <boolean test>;
```

For example, to test that a variable `x` is nonnegative, you could write the following line of code:

```
assert x >= 0;
```

In general, we expect these assertions to succeed. When an assertion fails, it signals a problem. It means that the program has a logic error that is preventing the assumptions from holding true.

Testing of assertions can be expensive, so Java lets you control whether this feature is enabled or disabled. You can enable assertion checking in your Java editor while you are developing and testing a program to make sure it works properly. Then you can disable it when you're fairly confident that the program works and you want to speed it up. By default, assertion checking is disabled.

## Enumerations: `enum`

Since Chapter 2 we have seen the usefulness of class constants. Sometimes we want to create a type that has only a small number of predefined constant values. For example, suppose we are writing a program for a card game. Each card has a suit: Clubs, Diamonds, Hearts, or Spades. We could represent these values as integers (0 through 3) or strings, but these are clumsy solutions because the range of integers and strings is large, so it may be possible to slip in an invalid suit value.

In such situations, it can be useful to create an *enumerated type*, which is a simple type that has only a small number of constant values.

```
public enum <name> {
    <name>, <name>, ..., <name>
}
```

For example, to create an enumerated type for card suits, you could write the following code in a file named `Suit.java`:

```
public enum Suit {
    CLUBS, DIAMONDS, HEARTS, SPADES
}
```

The following client code uses the enumerated type:

```
// 9 of Diamonds
int myCardRank = 9;
Suit myCardSuit = Suit.DIAMONDS;
```

It would also be possible to create a `Card` class that has a field for the card's rank and a field of type `Suit` for the card's suit. This way an invalid suit can never be passed; it must be one of the four constants declared. You can test whether a variable of an enumerated type stores a particular value with the `==` and `!=` operators:

```
if (myCardSuit == Suit.CLUBS) {
    // then this card is a Club
    ...
}
```

The `enum` concept is borrowed from past programming languages such as C, in which enumerated constants were actually represented as 0-based integers on the basis of the order in which they were declared. If you want to get an integer equivalent value for a Java `enum` value, call its `ordinal` method. For example, the call of `myCardSuit.ordinal()` returns 1 because `DIAMONDS` is the second constant declared. Every `enum` type also has a `values` method that returns all possible values in the enumeration as an array. For example, the call of `Suit.values()` returns the following array:

```
{Suit.CLUBS, Suit.DIAMONDS, Suit.HEARTS, Suit.SPADES}
```

## Packages

Since Chapter 3 we have used the `import` statement to make use of classes from the Java class libraries. The classes in the libraries are organized into groups called *packages*. Packages are valuable when you are working with large numbers of classes. They give Java multiple *namespaces*, meaning that there may be two classes with the same name so long as they are in different packages. Packages also help physically separate your code files into different directories, which makes it easier to organize a large Java project.

Every Java class is part of a package. If the class does not specify which package it belongs to, it is part of a nameless default package. To specify that the class belongs to some other package, place a package declaration statement as the first

statement at the top of the file, even before any `import` statements or the class's header. A package declaration has the following syntax:

```
package <name>;
```

For example, to specify that the class `CardGame` belongs to the `homework4` package, you would write the following statement at the top of the `CardGame.java` file:

```
package homework4;

import java.io.*;
import java.util.*;

// This class represents the main card game logic.
public class CardGame {
    ...
}
```

Packages may be nested, indicated by dots. The packages in the Java class libraries are nested at least two levels deep, such as `java.util` or `java.awt.event`. For example, to specify that a graphical user interface file for Homework 4 belongs to the `gui` subpackage within the `homework4` package, you would write the following statement:

```
package homework4.gui;
...
```

Packages are reflected by the directory structure of the files in your Java project. For example, if a file claims to belong to the `homework4` package, it must be placed in the `homework4/` directory relative to the root directory of your project. A file in the `homework.gui` package must be in the `homework4/gui` directory relative to the project's root.

If you write classes that are part of different packages and a class from one package wants to use a class from another, you must use an import statement for the compiler to find the classes. Remember that packages are not nested, so importing the `homework4` package does not automatically import `homework4.gui` and vice versa.

```
package general;

import homework4.*;
import homework4.gui.*;

...
```

Packages are generally not necessary for small projects, though some editors add them to the top of all files automatically. Users of basic Java IDEs or text editors

often avoid packages because they can make it harder to compile the entire project successfully. More sophisticated Java editors such as Eclipse or NetBeans handle packages better and usually provide a one-click button for compiling and executing all of the packages of a project.

## Protected and Default Access

In Chapter 8's discussion of encapsulation we learned about two kinds of access: `private` (visible only to one class) and `public` (visible to all classes). In general, you should follow the convention shown in this book of declaring all fields `private` and most methods `public` (other than internal helper methods). But Java has two other levels of access:

- `protected` access:

  Visible to this class and all of its subclasses, as well as to all classes in the same package as this class.
- default (package) access:

  Visible to all classes in the same package as this class.

Some programmers prefer to declare their fields as `protected` when writing classes that are part of an inheritance hierarchy. This declaration provides a relaxed level of encapsulation, because subclasses can directly access protected fields rather than having to go through accessor or mutator methods. For example, if a subclass `DividendStock` chose to extend the following `Stock` class, it would be able to directly get or set its `symbol` or `shares` field values:

```
public class Stock {
    protected String symbol;
    protected int shares;
    ...
}
```

The downside is that any class in the same package can also access the fields, which is generally discouraged. A compromise recommended by Joshua Bloch, author of *Effective Java*, is to give subclasses access to private data through protected methods instead of protected fields. This gives the class author the freedom to change the implementation of the class later if necessary.

Default access is given to a field when it has no access modifier in front of its declaration, such as in the initial versions of our `Point` class in Chapter 8 (prior to properly encapsulating it). A field or method with default access can be directly accessed by any class in the same package. We generally discourage the use of default access in most situations, since it can needlessly violate the encapsulation of the object.