Outline Covering Java SE 9

SEPTEMBER, 2018

LOLH

Short Contents

Table of Contents

The Java Language

The Java Language

1 Java SE 9 Introduction

2 Classes

The class is the logical construct upon which the Java language is built because it defines the shape and nature of an object, and therefore forms the basis for object-oriented programming in Java.

2.1 Class Fundamentals

A *class* defines a new data type. Once defined, this new type can be used to create objects of that type. A class is therefore a *template* for an object, and an *object* is an *instance* of a class. *Object* and *instance* are often used interchangeably.

2.1.1 General Form of a Class

When you define a class, you declare its exact form and nature. You do this by specifying the data that it contains and the code that operates on that data. A class is declared by use of the `class` keyword.

```
class classname {
  type instance-variable1;
  type instance-variable2;
  ...
  type instance-variableN;

  type method-name1 (parameter-list {
    body of method
  }

  type method-name2 (parameter-list {
    body of method
  }
  ...
  type method-nameN (parameter-list {
    body of method
  }
}
```

GeneralForm 2.1: Class Declaration — General Form

The data, or variables, defined within a class are called *instance variables*. The code is contained within *methods*. Collectively, the methods and variables defined within a class are called *members* of the class. In most cases, the instance variables are acted upon and accessed by the methods defined for that class. As a general rule, it is the methods that determine how a class' data can be used.

Each instance of the class (that is, each object of the class) contains its own copy of the instance variables. The data for one object is separate and unique from the data for another. Changes to the instance variables of one object have no effect on the instance variables of another.

Java classes do not need to have a '`main()`' method; you only need to specify one if that class is the starting point for the program.

In general, you use the *dot operator* to access both the instance variables and the methods within an object. Although commonly referred to as the dot *operator*, the formal specification for Java categorizes the `.` as a *separator*.

2.2 Declaring Objects

Because a class creates a new data type, you can use this type to declare objects of that type. Obtaining objects of a class is a two-step process.

1. Declare a variable of the class type; this variable does not define an object. Instead, it is simply a variable that can *refer* to an object.
2. Acquire an actual, physical copy of the object and assign it to the variable; you can do this using the `new` operator. The `new` operator dynamically allocates (at run time) memory for an object, and returns a reference to to. This reference is (essentially) the address in memory of the object allocated by `new`. This reference is then stored in the variable. In Java, all class objects must be dynamically allocated.

Example Declaration, Allocation, and Assignment

```
Box mybox; // 1. declare a variable
mybox = new Box(); // 2. allocate a Box object
```

These two declarations can be combined into a single declaration, and usually are:

```
Box mybox = new Box();
```

The `mybox` variable simply holds the memory address of the actual `Box` object. The class name followed by parenthese specifies the *constructor* for the class.

2.3 Methods

General Form of a Method Declaration

type name (*parameter-list*) {
  body of method
}

GeneralForm 2.2: Method Declaration — General Form

type specifies the type of data returned by the method. This can be any valid type, including class types that you create. If the method does not return a value, its return type must be `void`.

name is the name of the method. This can be any legal identifier.

parameter-list is a sequence of type and identifier pairs separated by commas. *Parameters* are essentially variables that receive the value of the arguments passed to the method when it is called. If the method has no parameters, then the parameter list will be empty.

Methods that have a return type other than `void` return a value to the calling routine using a *return statement*:

return *value*

where *value* is the value returned.

2.4 Constructors

Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor.

A *constructor* initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called when the object is created, before the `new` operator completes. Constructors have no return type. It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have fully initialized, usable object immediately.

2.5 The `this` Keyword

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the `this` keyword. `this` can be used inside any method to refer to the *current* object. That is, `this` is always a refernce to the object on which the method was invoked. You can use `this` anywhere a reference to an object of the current class' type is permitted.

2.5.1 Instance Variable Hiding

It is illegal to declare two local variables with the same name inside the same or enclosing scope. However, you can have local variables, including formal parameters to methods, which overlap with the names of the class' instance variables. For these cases, the local variables *hide* the instance variables of the same name.

Because `this` lets you refer directly to the object, you can use it to resolve any namespace collisions that might occur between instance variables and local variables. So, ***`this.width = width`*** is an example of a local variable (***`width`***) hiding an instance variable (also ***`width`***), with `this` allowing an assignment between them.

2.6 A Stack Class

To see a practical application of object-oriented programming, here is one of the archetypal examples of encapsulation: the stack. A *stack* stores data using *first-in, last-out* ordering. That is, a stack is like a stack of plates on a table — the first plate put down on the table is the last plate to be used. Stacks are controlled through two operations traditionally called *push* and *pop*. To put an item on top of the stack, you will use `push`. To take an item off the stack, you will use `pop`. It is easy to encapsulate the entire stack mechanism.

Here is a class called `Stack` that implements a stack for up to ten integers, plus test class called `TestStack`:

Stack.java

{`Stack.java`} ≡

```
class Stack {
  <Stack Instance Variables>
  <Stack Constructor>
  <Stack Instance Methods>
}
```

The following table lists called chunk definition points.

Chunk name	First definition point
<*Stack Constructor*>	See "Stack Constructor Subsection", page 7.
<*Stack Instance Methods*>	See "Stack Instance Methods Subsection", page 7.
<*Stack Instance Variables*>	See "Stack Instance Variables", page 7.

TestStack.java

{TestStack.java} ≡

```
class TestStack {
  <TestStack Main Method>
}
```

The called chunk <*TestStack Main Method*> is first defined at "Stack TestStack Subsection", page 8.

2.6.1 Stack Instance Variables

<*Stack Instance Variables*> ≡

```
int[] stck = new int[10];
int tos;
```

This chunk is called by {Stack.java}; see its first definition at "A Stack Class", page 6.

2.6.2 Stack Constructor Subsection

<*Stack Constructor*> ≡

```
// initialize top-of-stack tos
Stack() {
  tos = -1;
}
```

This chunk is called by the following chunks:

Chunk name	First definition point
{Stack.java}	See "A Stack Class", page 6.
{StackImproved.java}	See "An Improved `Stack` Class", page 13.

2.6.3 Stack Instance Methods Subsection

<*Stack Instance Methods*> ≡

```
<Stack Push>
<Stack Pop>
```

This chunk is called by the following chunks:

Chunk name	First definition point
{Stack.java}	See "A Stack Class", page 6.
{StackImproved.java}	See "An Improved `Stack` Class", page 13.

The following table lists called chunk definition points.

Chunk name	**First definition point**
<*Stack Pop*>	See "Stack Push and Pop Subsubsection", page 8.
<*Stack Push*>	See "Stack Push and Pop Subsubsection", page 8.

2.6.3.1 Stack Push and Pop Subsubsection

<*Stack Push*> ≡

```
// Push an item onto the stack
void push(int item) {
  if (tos == 9)
    System.out.println("Stack is full.");
  else
    stck[++tos] = item;
}
```

This chunk is called by <*Stack Instance Methods*>; see its first definition at "Stack Instance Methods Subsection", page 7.

<*Stack Pop*> ≡

```
// Pop an item from the stack
int pop() {
  if (tos < 0) {
    System.out.println("Stack underflow.");
    return 0;
  } else
    return stck[tos--];
  }
}
```

This chunk is called by <*Stack Instance Methods*>; see its first definition at "Stack Instance Methods Subsection", page 7.

2.6.4 Stack TestStack Subsection

<*TestStack Main Method*> ≡

```
public static void main(String[] args) {
  Stack mystack1 = new Stack();
  Stack mystack2 = new Stack();

  // push some numbers onto the stack
  for (int i = 0; i < 10; i++)
    mystack1.push(i);
  for (int i = 10; i < 20; i++)
    mystack2.push(i);

  // pop those numbers off the stack
  System.out.println("Stack in mystack1:");
  for (int i = 0; i < 10; i++)
```

```
      System.out.println(mystack1.pop());

    System.out.println("Stack in mystack2:");
    for (int i + 0; i < 10; i++)
      System.out.println(mystack2.pop());
  }
```

This chunk is called by {`TestStack.java`}; see its first definition at "A Stack Class", page 7.

3 Methods and Classes

This chapter examines several topics relating to methods and classes, including

- overloading
- parameter passing
- recursion
- access control
- keywords `static` and `final`
- `String` class
- Arrays
- nested and inner classes
- command-line arguments and varargs

3.1 Overloading Methods

It is possible to define two or more methods within the same class that share the same name as long as their parameter declarations are different. When this is the case, the methods are said to be *overloaded*, and the process is referred to as *method overloading*. Method overloading is one of the ways that Java supports polymorphism.

When an overloaded method is invoked, Java uses the type and number of arguments as its guid to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type or number of their parameters. While overloaded methods may have different return types, thei return type alone is inusfficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

The match between arguments and parameters need not always be exact. In some cases,Java's automatic type conversions can play a role in overload resolution. For example, if there is a method with one `double` parameter, and that method is invoked with a single `int` argument, then, when no exact match is found, Java will automatically convert the integer into a `double`, and this conversion will be used to resolve the call. Java will employ automatic type conversion only if no exact match is found.

Method overloading supports polymorphism because it is one way that Java implements the *one interface, multiple methods* paradigm. That is, Java does not need to rename each similar method just because it has a slightly different parameter requirements. The value of overloading is that it allows related methods to be accessed by use of a common name, representing the *general action* that is being performed, and leaves to the compiler the choice of the right *specific* version for a particular circumstance. The programmer need only remember the general operation being performed. Through the application of polymorphism, several names have been reduced to one. Overloading can help manage greater complexity.

3.1.1 Overloading Constructors

You can also overload constructor methods.

3.2 Objects as Parameters

It is both correct and common to pass objects to methods as well as primitive types. One of the most common uses of object parameters involves constructors. Frequently you will want to construct a new object so that it is initially the same as some existing object. To do this, you must define a constructor that takes an object of its class as a parameter. Providng many forms of constructors is usually required to allow objects to be constructed in a convenient and efficient manner.

3.3 Argument Passing

In general, there are two ways that a computer language can pass an argument to a subroutine:

1. call-by-value
2. call-by-reference

Java uses call-by-value to pass all arguments, although the precise effect differs between whether a primitive type or a reference type is passed.

When you pass a primitive type to a method, it is passed by value. Thus, a copy of the argument is made, and what occurs to the parameter that receives the argument has no effect outside the method.

When you pass an object to a method, the situation changes; objects are passed by what is effectively call-by-reference. When you pass a variable of a class type, you pass a reference to the method and the parameter receiving it will refer to the same object. This effectively means that objects act as if they are passed to methods by use of call-by-reference. Changes to the object inside the method *do* affect the object used as an argument. However, when an object reference is passed to a method, the reference itself is passed by use of call-by-value; therefore, that reference will continue to refer to the object, even though the object itself may be modified.

3.4 Returning Objects

A method can return any type of data, including class types that you create.

Since all objects are dynamically allocated using `new`, you don't need to worry about an object going out-of-scope because the method in which it was created terminates. The object will continue to exist as long as there is a reference to it somewhere in your program. When there are no references to it, the object will be reclaimed the next time garbage collection takes place.

3.5 Recursion

Recursion is the process of defining something in terms of itself. In programming, it is also what allows a method to call itself. A method that calls itself is said to be *recursive*.

When a method calls itself, new local variables and parameters are allocated storage on the stack, and the method code is executed with these new variables from the start. As each recursive call returns, the old local variables and parameters are removed from the stack, and execution resumes at the point of the call inside the method.

Recursive versions of many routines may execute a bit slower than the iterative equivalent because of the added overhead of the additional method calls. A large number of recursive calls to a method could cause a stack overrun. Because storage for parameters and local varibles is on the stack and each new call creates a new copy of these variables, it is possible that the stack could be exhausted. If this occurs, the Java run-time system will cause an exception.

The main advantage to recursive methods is that they can be used to create clearer and simpler versions of several algorithms than can their iterative relatives. For example, the QuickSort sorting algorithm is quite difficult to implement in an iterative way. Also, some types of AI-related algorithms are most easily implemented using recursive solutions.

3.6 Access Control

Encapsulation provides another important attribute besides linking data with code: *access control*. Through encapsulation, you can control what parts of a program can access the members of a class. By controlling access, you can prevent misuse. Thus, when correctly implemented, a class creates a *black box* which may be used, but the inner workings of which are not open to tampering. The classes introduced earlier do not completely meet this goal. For example, the `Stack` class provides the methods `push()` and `pop()` as a controlled interface to the stack, this interface is not enforced — it is possible for another part of the program to bypass these methods and access the stack directly. This could lead to trouble.

How a member can be accessed is determined by the *access modifier* attached to its declaration. Java supplies a rich set of access modifiers. Some aspects of access control are related mostly to inheritance or packages (and now modules). Those ideas will be discussed later. Here, let's examine access control as it relates to a single class.

Access Modifiers

Java's access modifiers are:

- public
- private
- protected (applies only to inheritance)
- default access level

`public` vs `private` Access

When a member of a class is modified by `public`, then that member can be accessed by any other code. When a member of a class is specified as `private`, then that member can only be accessed by other members of its class. Thus, the method `main()` is always preceded by the `public` modifier. It must be called by code that is outside the program — the Java run-time system.

Default Access — No Access Modifier

When no access modifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package. In the classes developed so far, all members of a class have used the `default` access mode. However, this is typically

not what you will want to be the case. Usually, you will want to restrict access to the data members of a class — allowing access only through methods. There will also be times when you will want to define methods that are private to a class.

Access Modifier Syntax

An access modifier precedes the rest of a member's type specification. That is, it must begin a member's declaration statement. As an example:

```
public int i;
private double j;

private int myMethod(int a, char b) {
  ...
}
```

Access Control and Inheritance

Consult the chapter on Chapter 4 "Inheritance", page 18, for more on the topic of access control in relation to inheritance.

3.6.1 An Improved `Stack` Class

StackImproved.java

Compare this code with that of Section 2.6 "A Stack Class", page 6.[1]

{`StackImproved.java`} ≡

```
class StackImproved {
  <Stack Private Instance Variables>
  <Stack Constructor>
  <Stack Instance Methods>
}
```

The following table lists called chunk definition points.

Chunk name	**First definition point**
<*Stack Constructor*>	See "Stack Constructor Subsection", page 7.
<*Stack Instance Methods*>	See "Stack Instance Methods Subsection", page 7.
<*Stack Private Instance Variables*>	See "An Improved `Stack` Class", page 13.

Stack Private Instance Variables

<*Stack Private Instance Variables*> ≡

```
/* Now, both stck and tos are private.  This means
   that they cannot be accidentally or maliciously
   altered in a way that would be harmful to the stack.
 */

private int[] stck = new int[10];
```

[1] Notice how all of the prior code except what is changed can easily be reused using TexiWebJr's modular system.

```
private int tos;
```

This chunk is called by {`StackImproved.java`}; see its first definition at "An Improved `Stack` Class", page 13.

Now both `stck`, which holds the stack, and `tos`, which is the index of the top of the stack, are specified as `private`. This means that they cannot be accessed or altered except through `push()` and `pop()`. Making `tos` private, for example, prevents other parts of your program from inadvertently setting it to a value that is beyond the end of the `stck` array. In other words, the following code, added to the end of the `TestStack.java` program (see "Stack TestStack Subsection", page 8), would be illegal and the program would not compile:

```
mystack1.tos = -2;
mystack2.stck[3] = 100;
```

3.7 `static` Keyword

There will be times when you want to define a class member that will be used independently of any object of that class. Normally, a class member must be accessed in conjunction with an object of its class. However, it is possible to create a member that can be used by itself without reference to a specific instance. To create such a member, precede its declaration with the keyword `static`. When a member is declared `static`, it can be accessed before any objects of its class are created, and without reference to any object.

You can declare both methods and variables to be `static`. Instance variables declared as `static` are essentially global variables. When objects of its class are declared, no copy of a `static` variable is made. Instead, all instances of the class share the same `static` variable.

Restrictions on `static` Methods

Methods declared as `static` have several restrictions:

- they can only directly call other `static` methods of their class;
- they can only directly access `static` variables of their class;
- they cannot refer to `this` or `super` in any way;

`static` Block

If you need to do computation in order to initialize your `static` variables, you can declare a `static` block that gets executed exactly once, when the class is first loaded (*static initialization block*).

```
class UseStatic {
  static int a = 3;
  static int b;

  static {
    b = a * 4;
  }
}
```

As soon as the `UseStatic` class is loaded, all of the `static` statements are run. First, `a` is set to '`3`', then the `static` block executes and initializes `b` to '`a * 4`' or '`12`'. Then `main()` is called (not shown).

Use of `static` Members Outside Their Class

Outside of the class in which they are defined, `static` methods and variables can be used independently of any object. To do so, you need only specify the name of their class followed by the dot operator: *classname.method()*. *classname* is the name of the class in which the `static` method is declared. A `static` variable can be accessed in the same way. This is how Java implements a controlled version of global methods and global variables.

3.8 `final` Keyword

A field can be declared as `final`. Doing so prevents its contents from being modified, making it, esentially, a constant. This means that you must initialize a `final` field when it is declared. You can do this in one of two ways: when it is declared, or within a constructor.

In addition to fields, both method parameters and local variables can be declared as `final`. Declaring a parameter as `final` prevents it from being changed within the method. Declaring a local variable `final` prevents it from being assigned a value more than once.

The keyword `final` can also be applied to methods, but its meaning is different than when applied to variables. This usage of `final` is described in the next chapter (see Chapter 4 "Inheritance", page 18).

3.9 Arrays Revisited

Arrays are implemented as objects. Because of this, there is a special array attribute that you will want to take advantage of. Specifically, the size of an array—that is, the number of elements that an array can hold—is found in its `length` instance variable. All arrays have this variable, and it will always hold the size of the array. Keep in mind that the value of `length` has nothing to do with the number of elements that are actually in use. It only reflects the number of elements taht the array is designed to hold.

3.10 Nested and Inner Classes

It is possible to define a class within another class; such classes are known as *nested classes*. The scope of a nested class is bounded by the scope of its enclosing class. A nested class does not exist independently of its enclosing class. A nested class has access to the members, including private members, of the enclosing class. However,the enclosing class does not have access to the members of the nested class. A nested class that is declared directly within its enclosing class scope is a member of its enclosing class. It is also possible to declare a nested class that is local to a block.

Static Nested Class

There are two types of nested class: *static* and *inner*. A static nested class is one that has the `static` modifer applied. Because it is static, it must access the non-static members of its enclosing class through an object. That is, it cannot refer to non-static members of its enclosing class directly. Static nested classes are seldom used.

Inner Class

The most important type of nested class is the *inner* class. An inner class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.

An instance of an inner class can be created only in the context of its enclosing class. The Java compiler will report an error otherwise. In general, an inner class instance is often creaed by code within its enclosing scope.

It is possible to define inner classes within any block scope, including within the block defined by a method or even within the body of a `for` loop.

Handling Events

While nested classes are not applicable to all situations, they are particularly helpful when handling events. See Chapter 18 "Event Handling", page 126. There are also *anonymous inner classes*, inner classes that don't have a name.

3.11 The `String` Class

Every string you create is an object of type `String`. Even string constants are `String` objects. For example, in the statement *`System.out.println("This is a String, too");`*, the quote is a `String` object.

Objects of type `String` are immutable; once a `String` object is created, its contents cannot be altered. Java defines peer classes of `String`, called `StringBuffer` and `StringBuilder`, which allow strings to be altered, so all of the normal string manipulations are still available.

Constructing `String` Objects and Concatenating `Strings`

Strings can be constructed in a variable of ways. The easiest is to use a statement:

```
String myString = "this is a test";
```

Java defines one operator for `String` objects: `+`. It is used to concatenate two strings.

```
String myString = "I" + " like " + "Java.";
```

`String` Methods

The `String` class contains several methods that you can use.

- boolean `equals(`*`secondStr`*`)`
- int `length()`
- char `charAt(`*`index`*`)`

3.12 Using Command-Line Arguments

Sometimes you will want to pass information into a program when you run it. This is accomplished by passing *command-line arguments* to `main()`. A command-line argument is the informatino that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program, access the `String`

arguments passed to the `args` parameter of `main()`. The first command-line argument is stored at `args[0]`, the second at `args[1]`, and so on. All command-line arguments are passed as strings. You must convert numeric values to their internal forms manually. See Chapter 12 "Exploring `java.lang`", page 84.

3.13 Varargs: Variable-Length Arguments

Beginning with JDK 5, Java has included a feature that simplifies the creation of methods that need to take a variable number of arguments. This feature is called *varargs* and it is short for *variable-length arguments*. A method that takes a variable number of arguments is called a *variable-arity method*, or simply *varargs method*.

A variable-length argument is specified by three period (. . .). For example: `static void vaTest (int ... v) {`. This syntax tells the compiler that `vaTest()` can be called with zero or more arguments. As a result, `v` is implicitly declared as an array of type `int[]`. Thus, inside `vaTest()`, `v` is accessed using the normal array syntax.

A method can have *normal* parameters along with a variable-length parameter, but the variable-length parameter must be the final parameter declared by the method. Further, there can be only one varargs parameter.

```
int doIt(int a, int b, double c, int ... vals) {
```

After the first three arguments, any remaining arguments are passed to `vals`.

Overloading Vararg Methods

You can overload a method that takes a variable-length argument (i.e., it can be given a different type, or additional parameters can be included, or a non varargs parameter).

Note that unexpected errors can result when overloading a method that takes a variable-length argument. These errors involve ambiguity because it is possible to create an ambiguous call to an overloaded varargs method. In such a case, the program will not compile. While each individual method declaration might be valid, the call might yet be ambiguous.

4 Inheritance

Inheritance is a cornerstone of object-oriented programming because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to them.

A class that is inherited is called a *superclass*. The class that does the inheriting is called a *subclass*. A subclass is a specialized version of a subclass. It inherits all of the members defined by the superclass and adds its own, unique elements.

4.1 Inheritance Basics

To *inherit* a class, incorporate the definition of one class into another by using the `extends` keyword.

```
class A {...}
class B extends A {...}
```

A subclass will include all of the members of its superclass. The subclass can directly reference all of the members of the superclass as well. Subclasses can be superclasses of other subclasses.

General Form of a Subclass Inheriting a Superclass

class *subclass-name* extends *superclass-name* {
  body of class
}

GeneralForm 4.1: Subclass General Form

A subclass can have only one superclass. Java does not support the inheritance of multiple superclasses into a single subclass.

4.1.1 Member Access and Inheritance

Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as `private`. A class member that has been declared as `private` will remain private to its class. It is not accessible by any code outside its class, including subclasses.

A major advantage of inheritance is that once you have created a superclass that defines the attributes commoin to a set of objects, it can be used to create any number of more specific subclasses. Each subclass can precisely tailor its own classification.

4.1.2 A Superclass Variable Can Reference a Subclass Object

A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass.

It is important to understand that itis the *type of the reference variable* — not the type of the object that it refers to — that determines what members can be accessed. That is, when a reference to a subclass object is assigned to a superclass reference variable, you will have access *only* to those parts of the object defined by the superclass. The superclass has no knowledge of what a subclass adds to it.

4.2 Using `super`

Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword `super`. `super` has two general forms. The first calls the superclass' constructor. The second is used to access a member of the superclass that has been hidden by a member of a subclass.

4.2.1 Using `super` to Call Superclass Constructors

A subclass can call a constructor defined by its superclass by use of the following form of `super`:

super(*arg-list*);

GeneralForm 4.2: `super` Calling a Constructor

arg-list specifies any arguments needed by the constructor in the superclass. `super()` must always be the first statement executed inside a subclass' constructor. `super()` can be called using any form defined by the superclass.

4.2.2 `super` Referencing Superclass

The second form of `super` acts somewhat like `this`, except that it always refers to the superclass of the subclass in which it is used.

super.*member*

GeneralForm 4.3: `super` Referencing its Superclass

member can be either a method or an instance variable. This form of `super` is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

```
i = super.i;
```

`super` allows access to the `i` defined in the superclass. `super` can also be used to call methods that are hidden by a subclass.

4.3 Creating a Multilevel Hierarchy

You can buidl hierarchies that contain as many layers of inheritance as you like. It is acceptable to use a subclass as a superclass of antoher. Each subclass inherits all of tghe traits found in all of its superclasses.

`super` always refers to the constructor in the closest superclass.

While an entire class hierarchy can be created in a single file, the individual classes (superclasses and subclasses) can be placed into their own files and compiled separately. Using separate files is the norm, not the exception, in creating class hierarchies.

4.4 When Constructors are Executed

In a class hierarchy, constructors complete their execution in order of derivation, from superclass to subclass.

4.5 Method Overriding

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass. When an overriden method is called from within its subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

If you wish to access the superclass version of an overridden method, you can so by using `super`.

Method overriding occurs *only* when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded (no name hiding takes place).

4.6 Dynamic Method Dispatch

Method overriding forms the basis for one of Java's most powerful concepts: *dynamic method dispatch*. This is a meachanism by which a call to an overrident method is resolved at run time, rather than compile time. This is important because this is how Java implements run-time polymorphism.

A superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overriden methods at run time. When an overriden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called. In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed. Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

4.6.1 Why Overridden Methods?

Overridden methods allow Java to support run-time polymorphism. Polymorphism is essential to object-oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods. Overridden methods are another way that Java implements the "one interface, multiple methods" aspect of polymorphism.

Successfully applying polymorphism is understanding that the superclasses and subclasses form a hierarchy which moves from lesser to greater specialization. Used correctly, the superclass provides all elements that a subclass can use directly. It also defines those methods that the derived class must implement on its own. This allows the subclass the flexibility to define its own methods, yet still enforces a consistent interface. Thus, by combining inheritance with overridden methods, a superclass can define the general form of the methods that will be used by all of its subclasses.

Dynamic, run-time polymorphism is one of the most powerful mechanisms that object-oriented design brings to bear on code reuse and robustness. The ability of existing code libraries to call methods on instances of new classes without recompiling while maintaining a clean abstract interface is a profoundly powerful tool.

4.6.2 Applying

Let's look at a practical example that uses method overriding. The following program creates a superclass called `Figure` that stores the dimensions of a two-dimensional object. It also defines a method called `area()` that computes the area of an object. The program derives two subclasses from `Figure`. The first is `Rectangle` and the second is `Triangle`. Each of these subclasses overrides `area()` so that it returns the area of a rectangle and a triangle respectively.

{`FindAreas.java` } ≡

```
<FindAreas SuperClass Figure >
<FindAreas SubClass Rectangle >
<FindAreas SubClass Triangle >
<FindAreas Main Class >
```

The following table lists called chunk definition points.

Chunk name	**First definition point**
<*FindAreas Main Class* >	See "FindAreas Main Class Section", page 24.
<*FindAreas SubClass Rectangle* >	See "FindAreas SubClass Rectangle Section", page 23.
<*FindAreas SubClass Triangle* >	See "FindAreas SubClass Triangle Section", page 23.
<*FindAreas SuperClass Figure* >	See "FindAreas Superclass Figure Section", page 21.

Output

The output from the program should be:

```
Inside Area for Rectangle.
Area is 45
Inside Area for Triangle.
Area is 40
Area for Figure is undefined.
Area is 0
```

Through the dual mechanisms of inheritance and run-time polymorphism, it is possible to define one consistent interface that is used by several different, yet related, types of objects. In this case, if an object is derived from `Figure`, then its area can be obtained by calling `area()`. The interface to this operation is the same no matter what type is being used.

4.6.2.1 FindAreas Superclass Figure Section

<*FindAreas SuperClass Figure* > ≡

```
class Figure {
  <Figure Instance Variable Declarations >
  <Figure Constructor >
  <Figure Area Method Declaration >
}
```

This chunk is called by {`FindAreas.java` }; see its first definition at "Applying", page 21.

The following table lists called chunk definition points.

Chunk name	First definition point
<*Figure Area Method Declaration* >	See "FindAreas Superclass Figure Section", page 22.
<*Figure Constructor* >	See "FindAreas Superclass Figure Section", page 22.
<*Figure Instance Variable Declarations* >	See "FindAreas Superclass Figure Section", page 22.

Figure Instance Variable Declarations

<*Figure Instance Variable Declarations* > ≡

```
double dim1;
double dim2;
```

This chunk is called by the following chunks:

Chunk name	First definition point
<*AbstractAreas Abstract Class Figure* >	See "AbstractAreas Abstract Class Figure Section", page 27.
<*FindAreas SuperClass Figure* >	See "FindAreas Superclass Figure Section", page 21.

Figure Constructor

<*Figure Constructor* > ≡

```
Figure (double 1, double b) {
  dim1 = a;
  dim2 = b;
}
```

This chunk is called by the following chunks:

Chunk name	First definition point
<*AbstractAreas Abstract Class Figure* >	See "AbstractAreas Abstract Class Figure Section", page 27.
<*FindAreas SuperClass Figure* >	See "FindAreas Superclass Figure Section", page 21.

Figure Area Method Declaration

It will be this method that will be overridden by the two subclasses; while this method will not produce any output, each of the subclasses will provide a formula for their own area and output that number, even though the same method (`area()`) is being called in each case from the same variable.

<*Figure Area Method Declaration* > ≡

```
double area() {
  System.out.println("Area for Figure is undefined.");
  return 0;
}
```

This chunk is called by <*FindAreas SuperClass Figure* >; see its first definition at "FindAreas Superclass Figure Section", page 21.

4.6.2.2 FindAreas SubClass Rectangle Section

<*FindAreas SubClass Rectangle* > ≡

```
class Rectangle extends Figure {
  <Rectangle Constructor >
  <Rectangle Area Method Declaration >
}
```

This chunk is called by the following chunks:

Chunk name	First definition point
{`AbstractAreas.java` }	See "Improved Figure Class", page 27.
{`FindAreas.java` }	See "Applying", page 21.

The following table lists called chunk definition points.

Chunk name	First definition point
<*Rectangle Area Method Declaration* >	See "FindAreas SubClass Rectangle Section", page 23.
<*Rectangle Constructor* >	See "FindAreas SubClass Rectangle Section", page 23.

Rectangle Constructor

<*Rectangle Constructor* > ≡

```
Rectangle (double a, double b) {
  super(a, b);
}
```

This chunk is called by <*FindAreas SubClass Rectangle* >; see its first definition at "FindAreas SubClass Rectangle Section", page 23.

Rectangle Area Method Declaration

<*Rectangle Area Method Declaration* > ≡

```
// override area for rectangle
double area() {
  System.out.println("Inside Area for Rectangle.";
  return dim1 * dim2;
}
```

This chunk is called by <*FindAreas SubClass Rectangle* >; see its first definition at "FindAreas SubClass Rectangle Section", page 23.

4.6.2.3 FindAreas SubClass Triangle Section

<*FindAreas SubClass Triangle* > ≡

```
class Triangle extends Figure {
  <Triangle Constructor >
  <Triangle Area Method Declaration >
}
```

This chunk is called by the following chunks:

Chunk name	**First definition point**
{`AbstractAreas.java` }	See "Improved Figure Class", page 27.
{`FindAreas.java` }	See "Applying", page 21.

The following table lists called chunk definition points.

Chunk name	**First definition point**
<*Triangle Area Method Declaration* >	See "FindAreas SubClass Triangle Section", page 24.
<*Triangle Constructor* >	See "FindAreas SubClass Triangle Section", page 24.

Triangle Constructor

<*Triangle Constructor* > ≡

```
Triangle (double a, double b) {
  super(a, b);
}
```

This chunk is called by <*FindAreas SubClass Triangle* >; see its first definition at "FindAreas SubClass Triangle Section", page 23.

Triangle Area Method Declaration

<*Triangle Area Method Declaration* > ≡

```
// override area for right triangle
double area () {
  System.out.println("Inside Area for Triangle.");
  return dim1 * dim2 / 2;
}
```

This chunk is called by <*FindAreas SubClass Triangle* >; see its first definition at "FindAreas SubClass Triangle Section", page 23.

4.6.2.4 FindAreas Main Class Section

<*FindAreas Main Class* > ≡

```
class FindAreas {
  <FindAreas Main Method Declaration >
}
```

This chunk is called by {`FindAreas.java` }; see its first definition at "Applying", page 21.

The called chunk <*FindAreas Main Method Declaration* > is first defined at "FindAreas Main Class Section", page 24.

FindAreas Main Method Declaration

<*FindAreas Main Method Declaration* > ≡

```
public static void main (String[] args[]) {
  <Create Basic Figure Objects >
  <Create Basic Figure Reference Variable >
  <Call Overridden Methods One By One >
```

```
}
```

This chunk is called by <*FindAreas Main Class* >; see its first definition at "FindAreas Main Class Section", page 24.

The following table lists called chunk definition points.

Chunk name	First definition point
<*Call Overridden Methods One By One* >	See "FindAreas Main Class Section", page 25.
<*Create Basic Figure Objects* >	See "FindAreas Main Class Section", page 25.
<*Create Basic Figure Reference Variable* >	See "FindAreas Main Class Section", page 25.

Create Basic Figure Objects

<*Create Basic Figure Objects* > ≡

```
Figure f = new Figure(10, 10);
Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);
```

This chunk is called by <*FindAreas Main Method Declaration* >; see its first definition at "FindAreas Main Class Section", page 24.

Create Basic Figure Reference Variable

This superclass reference variable `Figure figref` will hold, alternately, references to each of the classes and will call the method `area()` on each, producing a different result each time. This is the essence of method overriding and dynamic method dispatch.

<*Create Basic Figure Reference Variable* > ≡

```
Figure figref;
```

This chunk is called by the following chunks:

Chunk name	First definition point
<*AbstractAreas Main Method Declaration* >	See "Abstract Main Class", page 28.
<*FindAreas Main Method Declaration* >	See "FindAreas Main Class Section", page 24.

Call Overridden Methods One By One

<*Call Overridden Methods One By One* > ≡

```
figref = r;
System.out.println("Area is " + figref.area());

figref = t;
System.out.println("Area is " + figref.area());

figref = f;
System.out.println("Area is " + figref.area());
```

This chunk is called by <*FindAreas Main Method Declaration* >; see its first definition at "FindAreas Main Class Section", page 24.

4.7 Using Abstract Classes

There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes you will wnat to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement. One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method. This is the case with `Figure` in the preceding example. The definition of `area()` is simply a placeholder. It will not compute and display the area of any type of object.

It is not uncommon for a method to have no meaningful definition in the context of its superclass. Java's solution to this problem is the *abstract method*.

You can require that certain methods be overridden by subclasses by specifying the `abstract` type modifier. These methods are sometimes referred to as *subclasser responsibility* because they have no implementation specified in the superclass. Thus, a subclass must override them — it cannot simply use the version defined in the superclass.

To declare an abstract method, use the general form:

abstract *type name* (*parameter-list*);

GeneralForm 4.4: Abstract Method Declaration—General Form

No method body is present.

Any class that contains one or more abstract methods must also be declared abstract. To declare a class abstract, simply use the `abstract` keyword in front of the `class` keyword at the beginning of the class declaration. There can be no objects of an abstract class. That is, an abstract class cannot be directlyi instantiated with the `new` operator. You cannot declare abstract constructors or abstract static methods. Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be declared `abstract` itself. Abstract classes can include fully implemented methods.

Abstract Classes Can Be Reference Variables

Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach to run-time polymorphism is implemented thruogh the use of superclass references. Thus, it must be possible to create a reference to an asbtract class so that it can be used to point to a subclass object.

4.7.1 Improved Figure Class

Using the abstract class, you can improve the `Figure` class. Since there is no meaningful concept of area for an undefined two-dimensional figure, the following version of the program declares `area()` as abstract inside `Figure`. This means that all classes derived from `Figure` must override `area()`.

{`AbstractAreas.java` } ≡

<*AbstractAreas Abstract Class Figure* >
<*FindAreas SubClass Rectangle* >
<*FindAreas SubClass Triangle* >
<*AbstractAreas Main Class* >

The following table lists called chunk definition points.

Chunk name	First definition point
<*AbstractAreas Abstract Class Figure* >	See "AbstractAreas Abstract Class Figure Section", page 27.
<*AbstractAreas Main Class* >	See "Abstract Main Class", page 27.
<*FindAreas SubClass Rectangle* >	See "FindAreas SubClass Rectangle Section", page 23.
<*FindAreas SubClass Triangle* >	See "FindAreas SubClass Triangle Section", page 23.

4.7.1.1 AbstractAreas Abstract Class Figure Section

Notice that much of this class stays the same as the original `Figure` code, but includes two `abstract` declarations, one for the class, and one for the `area()` method declaration.

<*AbstractAreas Abstract Class Figure* > ≡

```
abstract class Figure {
  <Figure Instance Variable Declarations >
  <Figure Constructor >
  <AbstractAreas Abstract Area Method Declaration >
}
```

This chunk is called by {`AbstractAreas.java` }; see its first definition at "Improved Figure Class", page 27.

The following table lists called chunk definition points.

Chunk name	First definition point
<*AbstractAreas Abstract Area Method Declaration* >	See "AbstractAreas Abstract Class Figure Section", page 27.
<*Figure Constructor* >	See "FindAreas Superclass Figure Section", page 22.
<*Figure Instance Variable Declarations* >	See "FindAreas Superclass Figure Section", page 22.

AbstractAreas Abstract Area Method Declaration

<*AbstractAreas Abstract Area Method Declaration* > ≡

```
// areas is now an abstract method
abstract double area ();
```

This chunk is called by <*AbstractAreas Abstract Class Figure* >; see its first definition at "AbstractAreas Abstract Class Figure Section", page 27.

4.7.1.2 Abstract Main Class

<*AbstractAreas Main Class* > ≡

```
class AbstractAreas {
  <AbstractAreas Main Method Declaration >
```

```
}
```

This chunk is called by {`AbstractAreas.java` }; see its first definition at "Improved Figure Class", page 27.

The called chunk <*AbstractAreas Main Method Declaration* > is first defined at "Abstract Main Class", page 28.

AbstractAreas Main Method Declaration

<*AbstractAreas Main Method Declaration* > ≡

```
public static void main (String[] args) {
  <Create Basic Figure Objects Except Figure >
  <Create Basic Figure Reference Variable >
  <Call Overridden Methods One By One Except Figure >
}
```

This chunk is called by <*AbstractAreas Main Class* >; see its first definition at "Abstract Main Class", page 27.

The following table lists called chunk definition points.

Chunk name	**First definition point**
<*Call Overridden Methods One By One Except Figure* >	See "Abstract Main Class", page 28.
<*Create Basic Figure Objects Except Figure* >	See "Abstract Main Class", page 28.
<*Create Basic Figure Reference Variable* >	See "FindAreas Main Class Section", page 25.

Create Basic Figure Objects Except Figure

The only difference here is that because the superclass Figure is now abstract, it cannot be instantiated using `new`. It can, however, be used as a reference variable, and so the declaration ***`Figure figref;`*** is still valid and does not change from the prior implementation. **This is the essence of run-time polymorphism and dynamic method dispatch.**

<*Create Basic Figure Objects Except Figure* > ≡

```
// abstract class Figure cannot be instantiated
// Figure f = new Figure (10, 10);
Rectangle r = new Rectangle (9, 5);
Triangle t = new Triangle (10, 8);
```

This chunk is called by <*AbstractAreas Main Method Declaration* >; see its first definition at "Abstract Main Class", page 28.

Call Overridden Methods One By One Except Figure

The only difference here is that, because there is no `Figure` object, it cannot be referenced.

<*Call Overridden Methods One By One Except Figure* > ≡

```
figref = r;
System.out.println("Area is " + figref.area());
```

```
figref = t;
System.out.println("Area is " + figref.aread());

// there is no Figure object, so this will not work.
// figref = f;
```

This chunk is called by <*AbstractAreas Main Method Declaration* >; see its first definition at "Abstract Main Class", page 28.

4.8 Using `final` with Inheritance

The keyword `final` has three uses.

1. create the equivalent of a name constant.
2. prevent overriding
3. prevent inheritance

4.8.1 Using `final` to Prevent Overriding

There will be times when you want to prevent overriding from occurring. To disallow a method from being overridden, specify `final` as a modifier at the start of its declaration. Methods declared as `final` cannot be overridden.

Methods declared as `final` can sometimes provide a performance enhancement. The compiler is free to *inline* calls to them because it knows they will not be overridden by a subclass. Inlining is an option only with `final` methods. Normally, Java resolves calls to methods dynamically, at run time. This is called *late binding*. However, since `final` methods cannot be overridden, a call to one can be resolved at compile time. This is called *early binding*.

4.8.2 Using `final` to Prevent Inheritance

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with `final`. Declaring a class as `final` implicitly declares all of its methods as `final` also.

4.9 The Object Class

There is one special class, `Object`, defined by Java. All other classes are subclasses of `Object`. That is, `Object` is a superclass of all other classes. This means that a reference variable of type `Object` can refer to an object of any other class. Also, since arrays are implemented as classes, a variable of type `Object` can also refer to any array.

`Object` Methods

`Object` defines the following methods; this means they are available in every object.

`Object clone()`
: Creates a new object that is the same as the object being cloned.

`boolean equals(Object object)`
: Determines whether one object is equal to another.

`void finalize()`
: Called before an unused object is recycled. (Deprecated by JDK 9).

`Class<?> getClass()`
: Obtains the class of an object at run time.

`int hashCode()`
: Returns the hash code associated with the invoking object.

`void notify()`
: Resumes execution of a thread waiting on the invoking object.

`void notifyAll()`
: Resumes execution of all threads waiting on the invoking object.

`String toString()`
: Returns a string that describes the object.

`void wait()`
`void wait(long` *`milliseconds`*`)`
`void wait(long` *`millisconds`*`, int` *`nanoseconds`*`)`
: Waits on another thread of execution

The methods

- `getClass()`
- `notify()`
- `notifyAll()`
- `wait()`

are declared as `final`. You may override the others.

However, notice two methods now:

`equals()` compares two objects; returns `true` if the objects are equal, and `false` if not; the precise definition of equality can vary, depending on the type of objects being compared.

`toString()`
: returns a string that contains a description of the object on which it is called; this method is automatically called when an object is output using `println()`; many classes override this method; doing so allows them to tailor a description specifically for the types of objects that they create.

5 Packages

Packages are containers for classes. They are used to keep the class namespace compartmentalized, i.e., to prevent collisions between file names. Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.

5.1 Introduction to Packages

Java provides a mechanism for partitioning the class namespace into manageble chunks: the *PACKAGE*. The package is both a naming and a visibility control mechanism. In other words, you can use the package mechanism to define classes inside a package that are not accessible by code outside the package; and you can define class members that are exposed only to other members of the same package.

5.2 Defining Packages

To create a package ("define" a package), include the `package` command as the first statement in a Java source file. Thereafter, any classes declared within that file will belong to the specified package. The `package` statement defines a namespace in which classes are stored. Without the `package` statement, classes are put into the `default` package (which has no name).

General Form of `package` statement

package *pkg*

GeneralForm 5.1: Package Statement — General Form

pkg is the name of the package. For example:

```
package mypackage;
```

File System Directories

Java uses the file system directories to store packages. Therefore, the `.class` files for any classes you declare to be part of `mypackage` must be stored in a directory called `mypackage`. The directory name must match the package name exactly.

More than one file can include the same `package` statement. The `package` statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package. Most real-world packages are spread across many files.

Hierarchy of Packages

You can create a hierarch of packages. To do so, separate each package name form the one above it by use of a period. The general form of a multileveled package statement is:

package *pkg1[.pkg2[.pkg3]]*

GeneralForm 5.2: Package Statement — Multilevel Form

A package hierarchy must be reflected in the file system of your Java development system. For example a package declared as:

```
package a.b.c;
```

needs to be stored in directory `a/b/c`.

Be sure to choose package names carefully; you cannot rename a package without renaming the directory in which the classes are stored.

5.3 Finding Packages and CLASSPATH

Packages are mirrored by directories. How does the Java run-time system know where to look for packages?

'cwd' By default, the Java run-time system uses the currect working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found.

CLASSPATH
You can specify a directory path or paths by setting the `CLASSPATH` environment variable.

-classpath
You can use the `-classpath` option with `java` and `javac` to specify the path to your classes.

module path
Beginning with JDK 9, a package can be part of a module, and thus found on the `module path`.

Example Finding a Package

Consider the following package specification:

```
package mypack;
```

In order for programs to find `mypack`, the program can be executed from a directory **immediadely above** `mypack`, or the `CLASSPATH` must be set to include the path to `mypack` or the `-classpath` option must specify the path to `mypack` when the program is run via `java`.

When the second or third of the above options is used, the `class path` **must not include** `mypack` itself. It must simply specify the `path` to just above `mypack`. For example, if the path to `mypack` is

```
/MyPrograms/Java/mypack
```

then the class path to `mypack` is

```
/MyPrograms/Java
```

5.4 Packages and Member Access

Packages add another dimension to access control. Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. *Packages* act as containiners for classes and other subordinate packages. *Classes* act as containers for data and code. The class is Java's smallest unit of abstraction. As it relates to the interplay between classes and packages, Java addresses four categories of visibility for class members:

- Subclasses in the same package
- Non-subclasses in the same package

- Subclasses in different packages
- Classes that are neither in the same package nor subclasses

The three access modifiers

- private
- public
- protected

provide a variaty of ways to produce many levels of access required by these categories.

Category	**Private**	**None**	**Protected**	**public**
Same Class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package noni-subclass	No	No	No	Yes

Table 5.1: Package Access Table — Shows all combinations of the access control modifiers

5.5 Importing Packages

Java includes the `import` statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred to directly, using only its name. The `import` statement is a convenience to the programmer and is not technically needed to write a complete Java program.

In a Java source file, `import` statements occur immediately following the `package` statement (if one exists) and before any class definitions. This is the general form of the `import` statement:

import *pkg1*[.*pkg2*].(*classname* | *);

GeneralForm 5.3: Import Statement — General Form

Here, *pkg1* is the name of a top-level package, and *pkg2* is the name of a subordinate package inside the outerpackage separated by a dot (`.`). There is no limit on the depth of a package hierarchy. Finally, you can specify either an explicit `classname` or a star (`*`), which indicates that the Java compiler should import the entire package.

```
import java.util.Date;
imort java.io.*;
```

All of the standard Java SE classes included with Java begin with the name `java`. The basic language functions are stored in a package called `java.lang`. Normally, you have to import every package or class that you want to use, but since Java is useless without much of the functionality in `java.lang`, it is implicitly imported by the compiler for all programs. This is equivalent to the following line being at the top of all your programs:

```
import java.lang.*;
```

The `import` statement is *optional*. Any place you use a class name, you can use its *fully qualified name*, which includes its full package hierarchy.

When a package is imported, only those items within the package declared as `public` will be available to non-subclasses in the importing code.

6 Interfaces

Using the keyword `interface`, you can fully abstract a class' interface from its implementation. That is, using `interface`, you can specify what a class must do, but not how to do it. Interfaces are syntactically similar to classes, but they lack instance variables, and, as a general rule, their methods are declared without any body. Once it is defined, any number of classes can implement an `interface`. Also, one class can implement any number of interfaces. To implement an interfce, a class must provide the complete set of methods required by the interface. Each class is free to determine the details of its own implementation. By providing the `interface` keyword, Java allws you to fully utilize the "one interface, multiple methods" aspect of polymorphism.

Interfaces are designed to support *dynamic method resolution* at run time. Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible. This requirement by itself makes for a static and nonextensible classing environment. Inevitably in a system like this, functionality gets pushed up higher and higher in the class hierarchy so that the mechanisms will be available to more and more subclasses. Interfaces are designed to avoid this problem. *They disconnect the definition of a method or set of methods from the inheritance hierarchy.* Since interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of class hierarchy to implement the same interface. This is where the real power of interfaces is realized.

6.1 Defining Interfaces

An interface is defined much like a class. Here is a simplified general form of an interface definition:

access interface *name* {
  return-type method-name1(*parameter-list*);
  return-type method-name2(*parameter-list*);

  type final-varname1 = *value*
  type final-varname2 = *value*
  . . .
  return-type method-nameN(*parameter-list*);
  type final-varnameN = *value*
}

GeneralForm 6.1: Interface Definition — Simplified General Form

When no access modifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as `public`, the interface can be used by code outside its package. In this case, the interface mst be the only public interface declared in the file, and the file must have the same name as the interface. The methods that are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods. Each class that includes such an interface must implement all of the methods.

Variable Declarations inside Interfaces

As the general form shows, variables can be declared inside interface declarations. They are implicitly `final` and `static`, meaning they cannot be changed by the implementing class. They must also be initialized. All methods and variables are implicitly `public`.

6.2 Implementing Interfaces

Once an interface has been defined, one or more classes can implement that interface. To implement an interface, include the `implements` clause in a class definition, and then create the methods required by the interface. The general form of a class that includes the `implements` clause looks like this:

class *clasname* [extends *superclass*] [implements *interface* [,*interface*. . .] {
  class-body
}

GeneralForm 6.2: Class Implementing Interface — General Form

The methods that implement an interface must be declared `public`. The type signature of the implementing method must match exactly the type signature specified in the `interface` definition.

It is both permissible and common for classes that implement interfaces to define additional members of their own.

6.3 Accessing Implementations Through Interface References

You can declare variables as object references that use an interface rather than a class type. Any instance of any class that implements the declared interface can be referred to by such a variable. When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interfaces. The method to be executed is looked up dynamically at run-time, allowing classes to be created later than the code which calls methods on them. The calling code can dispatch through an interface without having to know anything about the "callee." This process is similar to using a superclass reference to access a subclass object.

6.4 Partial Implementations

If a class includes an interface but does not implement the methds required by that interface, then that class must be declared as `abstract`. Any class that inherits the abstract class must implement the interface or be declared `abstract` itself.

6.5 Nested Interfaces

An interface can be declared a member of a class or another interface. Such an interface is called a *member interface* or a *nested interface*. A nested interface can be declared as `public`, `private`, or `protected`. This differs from a top-level interface, which must either be declared as `public` or use the default access level. When a nexted interface is used

outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is a member. Thus, outside of the class or interface in which a nested interface is declared, its name must be fully qualified.

6.6 Applying Interfaces

See detailed example . . .

6.7 Variables in Interfaces

You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values. When you include that interface in a class (when you "implement" the interface), all of those variable names will be in scope as constants. If an interface contains no methods, then any class that includes such an interface doesn't actually implement anything. It is as if that class were importing this constant fields into the class name space as `final` variables.

6.8 Interfaces Can Be Extended

One interface can inherit another by use of the keyword `extends`. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods required by the interface inheritance chain.

6.9 Default Interface Methods

Prior to JDK 8, an interface could not define any implementation whatsoever. This meant that for all previous versions of Java, the methods specified by an interface were abstract, constaining no body. This is the traditional form of an interface. The release of JDK 8 changed this by adding a new capability to `interface` called the *default method*. A default method lets you define a default implementation for an interface method. It is possible for an interface method to provide a body, rather than being abstract.

A primary motivation for the default method was to provide a means by which interfaces could be expanded without breaking existing code. There must be implementations for all methods defined by an interface. If a new method were added to a popular, widely used interface, then the addition of that method would break existing code because no implementation would be found for that new method. The default method solves this problem by supplying an implementation that willbe used if no other implementation is explicitly provided. Thus, the addition of a default method will not cause preexisting code to break.

Another motivation for the default method was the desire to specify methods in an interface that are, essentially, optional, depending on how the interface is used.

Interfaces Do No Maintain State and Cannot Be Created

It is important to point out that the addition of default methods does not change a key aspect of `interface`: its inability to maintain state information. An interface still cannot have instance variables, for example. Thus, **the defining difference between an interface and a class is that a class can maintain state information, but an interface cannot.** Furthermore,

it is still not possible to create an instance of an interface by itself. It must be implemented by a class.

6.10 Use Static Methods in an Interface

Another capability added to interface by JDK 8 is the ability to define one or more `static` methods. Like `static` methods in a class, a `static` method defined by an interface can be called independently of any object. Thus, no implementation of the interface is necessary, and no instance of the interface is required, in order to call a `static` method. Instead, a `static` method is called by specifying the interface name, followed by a period, followed by the method name. Here is the general form:

InterfaceName.staticMethodName

GeneralForm 6.3: Interface Static Method, Calling

Notice that this is similar to the way that a `static` method in a class is called. However, `static` interface methods are not inherited by either an implementing class or a subinterface.

6.11 Private Interface Methods

Beginning with JDK 9, an interface can include a private method. A private interface method can be called only by a default method or another private method defined by the same interface. Because a private interface method is specified `private`, it cannot be used by code outside the interface in which it is defined. This restriction includes subinterfaces because a private inteface method is not inherited by a subinterface.

The key benefit of a private interface method is that it lets two or more default methods use a common piece of code, thus avoiding code duplication.

7 I/O

This chapter introduces `java.io`, which supports Java's basic input/output system, including file I/O. Support for I/O comes from Java's core API libraries, not from language keywords. In this chapter the foundation of this subsystem is introduced so that you can see how it fits into the larger context of the Java programming and execution environment. This chapter also looks at the `try-with-resources` statement.

7.1 I/O Basics

Most real applications of Java are not text-based, console programs. Rather, they are either graphically oriented programs that rely on one of Java's graphical user interface (GUI) frameworks, such as Swing, the AWT, or JavaFX, for user interaction, or they are Web applications. Text-based console programs do not constitute an important use for Java in the real world. Java's support for console I/O is limited and somewhat awkword to use. Text-based console I/O is just not that useful in real-world Java programming.

Java does, however, provide strong, flexible support for I/O as it relates to files and networks. Java's I/O system is cohesive and consistent. A general overview of I/O is presented here. A detailed description is found in chapters describing the Java Library: See Chapter 15 "Input/Output — `java.io`", page 98, and See Chapter 16 "NIO", page 124.

7.1.1 Streams

Java programs perform I/O through streams. A *stream* is an abstraction that either produces or consumes information. A stream is linked to a physical device by the Java I/O system. All streams behave in the same manner, even if the actual physical device to which they are linked differ. Thus, the same I/O classes and methods can be applied to different types of devices. This means that an input stream can abstract many different kinds of input; from a disk file, a keyboard, or a network socket. Likewise, an output stream may refer to the console, a disk file, or a network connection. Java implements streams within class hierarchies defined in the `java.io` package.

7.1.2 Byte Streams and Character Streams

Java defines two types of streams:

- byte streams
- character streams

Byte streams provide a convenient means for handling input and output of bytes. Byte streams are used when reading or writing binary data. *Character streams* provide a convenient means for handling input and output of characters. They use Unicode and therefore can be internationalized. In some cases, character streams are more efficient than byte streams.

The original version of Java (Java 1.0) did not include character streams and thus all I/O was byte-oriented. Character streams were added by Java 1.1 and certain byte-oriented classes and methods were deprecated.

At the lowest level, all I/O is still byte-oriented. The character-based streams simply provide a convenient and efficient means for handling characters.

7.1.2.1 The Byte Stream Class

Byte streams are defined by using two class hierarchies. At the top are two abstract classes:

- `InputStream`

- `OutputStream`

Each of these abstract classes has several concrete subclasses that handle the differences among various devices, such as disk files, network connnections, and memory buffers. The byte stream classes in `java.io` are shown in Table 7.1.

To use the stream classes, you must import `java.io`.

`BufferedInputStream`
`BufferedOutputStream`
: Buffered input and output streams

`ByteArrayInputStream`
`ByteArrayOutputStream`
: Input and Output streams that read from and write to a byte array

`DataInputStream`
`DataOutputStream`
: Input and Output streams that contain methods for reading and writing the Java standard data types

`FileInputStream`
`FileOutputStream`
: Input and Output streams that read from and write to a file

`FilterInputStream`
`FilterOutputStream`
: Implements `InputStream` and `OutputStream`

`InputStream`
`OutputStream`
: Abstract classes that describe stream input and output

`ObjectInputStream`
`ObjectOutputStream`
: Input and Output streams for objects

`PipedInputStream`
`PipedOutputStream`
: Input and Output pipe

`PrintStream`
: Output stream that contains `print()` and `println()`

`PushbackInputStream`
: Input stream that allows bytes to be returned to the input stream

`SequenceInputStream`
: Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

Table 7.1: The Byte Stream Classes in `java.io`

The abstract classes `InputStream` and `OutputStream` define several key methods that the other stream classes implement. Two of the most important are:

- `read()`
- `write()`

which respectively read and write bytes of data. Each has a form that is abstract and must be overridden by derived stream classes.

7.1.2.2 The Character Stream Class

Character streams are defined by using two class hierarchies. At the top are two abstract classes:

- `Reader`
- `Writer`.

These abstract classes handle Unicode character streams. Java has several concrete subclasses of these. The character stream classes in `java.io` are shown in Table 7.2.

`BufferedReader` `BufferedWriter`	Buffered input and output character streams
`CharArrayReader` `CharArrayWriter`	Input and Output streams that read and write to and from a character array
`FileReader` `FileWriter`	Input and Output streams that read from and write to a file
`FilterReader` `FilterWriter`	Filtered read and writer
`InputStreamReader` `OutputStreamWriter`	Input and Output streams that translate bytes to characters
`LineNumberReader`	Input stream that counts lines
`PipedReader` `PipedWriter`	Input and Output pipes
`PrintWriter`	Output stream that contains `print()` and `println()`
`PushbackReader`	Input stream that allows characters to be return to the input stream
`Reader` `Writer`	Abstract clases tha describe character stream input and output
`StringReader` `StringWriter`	Input and output streams that read from and write to a string

Table 7.2: The Character Stream I/O Classes in `java.io`

The abstract classes `Reader` and `Writer` define several key methods that the other stream classes implement. Two of the most important methods are:

- `read()`
- `write()`

which read and write characters of data, respectively. Each has a form that is abstract and must be overridden by derived stream classes.

7.1.2.3 The Predefined Streams

The `java.lang` package defines a class called `System`, which encapsulates several aspects of the run-time environment. System contains three predefined stream variables:

1. `in` (standard input), an object of type `InputStream`
2. `out` (standard output), an object of type `PrintStream`
3. `err` (standard error), an object of type `PrintStream`

These fields are declared as `public`, `static`, and `final` within `System`. This means that they can be used by any other part of your program and without reference to a specific `System` object. While these are all byte streams, they can be wrapped within character-based streams, if desired.

7.2 Reading Console Input

For commercial applications, the preferred method of reading console input is to use a character-oriented stream. This makes your program easier to internationalize and maintain.

`System.in` Wrapped in `BufferedReader`

Console input is accomplished by reading from `System.in` To obtain a character-based stream that is attached to the console, wrap `System.in` in a `BufferedReader` object. `BufferedReader` supports a buffered input stream. A commonly-used constructor is:

BufferedReader(Reader *inputReader*)

Here, *inputReader* is the stream that is linked to the instance of `BufferedReader` that is being created. `Reader` is the abstract class. One of its concrete subclasses is `InputStreamReader`, which converts bytes to characters. To obtain a `InputStreamReader` object that is linked to `System.in`, use the following constructor:

InputStreamReader(InputStream *inputStream*)

Because `System.in` refers to an object of type `InputStream`, it can be used for *inputStream*. Putting it all together, the following line of code creates a `BufferedReader` that is connected to the keyboard:

```
BufferedReader br = new BufferedReader(
  new InputStreamReader(System.in));
```

After this statement executes, `br` is a character-based stream that is linked to the console through `System.in`.

7.2.1 Reading Characters

To read a character from a `BufferedReader`, use `read()`. The version of `read()` that we will be using is

int read() throws IOException

Each time that `read()` is called, it reads a character from the input stream and returns it as an integer value.[1] It returns -1 when an attempt is made to read at the end of the stream. It can throw an `IOException`.

Program Demonstrating Reading Characters from Console

The following program demonstrates `read()` by reading characters from the console until the user types a "q". Any I/O exceptions that might be generated are simply thrown out of `main()`. In more sophisticated applications, you can handle the exceptions explicitly.

{BRRead.java} ≡

```
<Import java.io>
class BRRead {
  public static void main(String[] args[]) throws IOException {
    <BRRead BufferedReader Constructor>
    <BRRead Enter Characters>
  }
}
```

The following table lists called chunk definition points.

Chunk name	First definition point
<*BRRead BufferedReader Constructor*>	See "BRRead BufferedReader Constructor Section", page 45.
<*BRRead Enter Characters*>	See "BRRead Enter Characters Section", page 45.
<*Import java.io*>	See "Import `java.io`", page 44.

7.2.1.1 Import `java.io`

<*Import java.io*> ≡

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

This chunk is called by the following chunks:

Chunk name	First definition point
{BRRead.java}	See "Reading Characters", page 44.
{BRReadLines.java}	See "Reading Strings", page 45.
{BufferedInputStreamDemo.java}	See "Buffered Input Example", page 112.
{BufferedReaderDemo.java}	See "Buffered Reader Demo", page 120.
{CopyFile.java}	See "Demonstration Writing to a File", page 55.
{CopyFileMultTryWR.java}	See "Demonstration of Multiple Resources", page 60.
{DirListOnly}	See "Example Program Using FilenameFilter Interface", page 105.
{FileReaderDemo.java}	See "`FileReader`", page 119.
{OnlyExt.java}	See "Example Program Using FilenameFilter Interface", page 104.
{PrinterWriterDemo.java}	See "Demonstration Using a `PrintWriter` for Console Output", page 47.
{ShowFile.java}	See "Demonstration Reading From a File", page 49.
{ShowFileAlt.java}	See "`close()` Within `finally` Block", page 52.
{ShowFileSingleTry}	See "Demonstration Reading From a File with a Single `try` Block", page 53.

[1] Note that `System.in` is line buffered by default; this means that no input is actually passed to the program until the user presses `enter`. This does not make `file` particularly valuable for interactive console input.

{ShowFileTryWR.java} See "Demonstration of Automatically Closing a File", page 58.

7.2.1.2 BRRead BufferedReader Constructor Section

<*BRRead BufferedReader Constructor*> ≡

```
BufferedReader br = new BufferedReader(
  new InputStreamReader(System.in));
```

This chunk is called by {BRRead.java}; see its first definition at "Reading Characters", page 44.

7.2.1.3 BRRead Enter Characters Section

<*BRRead Enter Characters*> ≡

```
char c;
do {
  c = (char) br.read();
  System.out.println(c);
} while (c != 'q');
```

This chunk is called by {BRRead.java}; see its first definition at "Reading Characters", page 44.

7.2.2 Reading Strings

To read a string frmo the keyboard, use the version of `readLine()` that is a member of the `BufferedReader` class. Its general form is:

String readLine() throws IOException

It returns a `String` object.

Program Demonstrating Reading a String from Console

The following program demonstrates `BufferedReader` and the `readLine()` method; the program reas and displays lines of text until the word "stop" is entered.

{BRReadLines.java} ≡

```
<Import java.io>

class BRReadLines {
  public static void main(String[] args) throws IOException {
    <BRReadLines BufferedReader Constructor>
    <BRReadLines Enter Lines>
  }
}
```

The following table lists called chunk definition points.

Chunk name	First definition point
<*BRReadLines BufferedReader Constructor*>	See "BRReadLines BufferedReader Constructor", page 46.
<*BRReadLines Enter Lines*>	See "BRReadLines Enter Lines", page 46.
<*Import java.io*>	See "Import `java.io`", page 44.

7.2.2.1 BRReadLines BufferedReader Constructor

<*BRReadLines BufferedReader Constructor*> ≡

```
// create a BufferedReader using System.in
BufferedReader br = new BufferedReader(
  new InputStreamReader(System.in))
```

This chunk is called by {`BRReadLines.java`}; see its first definition at "Reading Strings", page 45.

7.2.2.2 BRReadLines Enter Lines

<*BRReadLines Enter Lines*> ≡

```
String str;

System.out.println("Enter lines of text.");
System.out.println("Enter 'stop' to quit.");

do {
  str = br.readLine();
  System.out.println(str);

} while (!str.equals("stop"));
```

This chunk is called by {`BRReadLines.java`}; see its first definition at "Reading Strings", page 45.

7.3 Writing Console Output

The methods

- `print()`
- `println()`

are defined by the class `PrintStream` (which is the type of object referenced by `System.out`). Remember, `System.out` is a byte stream, but is acceptable for simple program output. A character-based alternative is described in the next section.

Because `PrintStream` is an output stream (type byte) derived from `OutputStream` (an abstract byte stream class), it also implements the low-level method `write()`. Thus, `write()` can be used to write to the console. The simplest form of `write()` defined by `PrintStream` is here:

void write(int *byteval*)

This method writes the byte specified by *byteval*. Although *byteval* is declared as in integer, only the low-order eight bits are written.

Here is a short example that uses `write()` to output the character "A" followed by a newline to the screen:

```
//Demonstrate System.out.write()
class WriteDemo {
  public static void main (String[] args) {
    int b;
```

```
    b = 'A';
    System.out.write(b);
    System.out.write('\n');
  }
}
```

7.4 The PrintWriter Class

Using `System.out` to write to the console is probably best for debugging purposes or for sample programs. For real-world programs, the recommended method of writing to the console when using Java is through a `PrintWriter` stream. `PrintWriter` is one of the character-based classes. User a character-based class for console output makes internationalizing your program easier.

7.4.1 `PrintWriter` Constructors

`PrintWriter` defines several constructors. Here is one:

PrintWriter(OutputStream *outStream*, boolean *flusingOn*)

outputStream is an object of type `OutputStream`, and *flushingOn* controls whether Java flushes the output stream every time a `println` method is called. If *flushingOn* is `true`, flushing automatically takes place. If `false`, flushing is not automatic.

`PrintWriter` supports `print()` and `println()` methods. You can use these methods in the same way you used them with `System.out`. If an argument is not a simple type, the `PrintWriter` methods call the object's `toString()` method and then display the result.

Writing to the Console with a `PrintWriter`

To write to the console by using a `PrintWriter`, specify `System.out` for the output stream and automatic flushing.

```
PrintWriter pw = new PrintWriter(System.out, true);
```

7.4.2 Demonstration Using a `PrintWriter` for Console Output

{`PrinterWriterDemo.java`} ≡

```
<Import java.io>

public class PrintWriterDemo {
  public static void main (String[] args) {
    <PrintWriterDemo PrintWriter Constructor>
    <PrintWriterDemo Printing To Console>
  }
}
```

The following table lists called chunk definition points.

Chunk name	**First definition point**
<*Import java.io*>	See "Import `java.io`", page 44.
<*PrintWriterDemo PrintWriter Constructor*>	See "PrintWriterDemo PrintWriter Constructor", page 48.
<*PrintWriterDemo Printing To Console*>	See "PrintWriterDemo Printing To Console", page 48.

7.4.2.1 PrintWriterDemo PrintWriter Constructor

<*PrintWriterDemo PrintWriter Constructor*> ≡

```
PrintWriter pw = new PrintWriter (System.out, true);
```

This chunk is called by {`PrinterWriterDemo.java`}; see its first definition at "Demonstration Using a `PrintWriter` for Console Output", page 47.

7.4.2.2 PrintWriterDemo Printing To Console

<*PrintWriterDemo Printing To Console*> ≡

```
pw.println("This is a string");
int i = -7;
pw.println(i);
double d = 4.5e-7;
pw.println(d);
```

This chunk is called by {`PrinterWriterDemo.java`}; see its first definition at "Demonstration Using a `PrintWriter` for Console Output", page 47.

7.4.3 `PrintWriter` Concluding Comments

There is nothing wrong with using `System.out` to write simple text output to the console. Using a `PrintWriter` makes your real-world applications easier to internationalize. There is no other adavantage gained by using a `PrintWriter` in the simple programs, however.

7.5 Reading and Writing Files

Java provides a number of classes and methods that allow you to read and write files. The purpose of this section is to introduce the basic techniques that read from and write to a file. Although byte streams are used, these techniques can be adapted to the character-based streams.

.....

7.5.1 `FileInputStream` and `FileOutputStream`

Two of the most often-used stream classes are:

- `FileInputStream`
- `FileOutputStream`

which create byte streams linked to files.

Open a File

To open a file, create an object of one of these clases, specifying the name of the file as an argument to the constructor. We will use the following constructors:

FileInputStream(String *fileName*) throws FileNotFoundException
FileOutputStream(String *fileName*) throws FileNotFoundException

Note that when an output file is opened, any preexisting file by the same name is destroyed.

Close a File

When you are done with a file, you must close it. This is done by calling the `close()` method, which is implemented by both `FileInputStream` and `FileOutputStream`.

void close() throws IOException

Closing a file releases the system resources allocated to the file. Failure to close a file can result in "memory leaks" because of unused resources remaining allocated.

`AutoClosable Interface`

Beginning with JDK 7, the `close()` method is specified by the `AutoCloseable` interface in `java.lang`. `AutoCloseable` is inherited by the `Closable` interface in `java.io`. Both interfaces are implemented by the stream classes, including `FileInputStream` and `FileOutputStream`.

Try With Resources

There are two basic approaches you can use to close a file. The first is the traditional approach, in which `close()` is called explicitly when the file is no longer needed. This is the approach used by all versions of Java prior to JDK 7.

The second is to use the `try-with-resources` statement added by JDK 7, which automatically closes a file when it is no longer needed. In this approach, no explicit call to `close()` is executed.

Reading From A File

To read from a file, you can use a version of `read()` that is defined within `FileInputStream`.

int read() throws IOException

Each time that it is called, it reads a single byte from the file and returns the byte as an integer value. `read()` returns -1 when an attempt is made to read at the end of the stream.

Writing to a File

To write to a file, you can use the `write()` method defined by `FileOutputStream`. Its simplest form is:

void write(int *byteval* throws IOException

This method writes the byte specified by *byteval* to the file. Although *byteval* is declared as an integer, only the low-order eight bits are written to th file.

7.5.2 Demonstration Reading From a File

The following program uses `read()` to input and display the contents of a file that contains ASCII test. The name of the file is specified as a command-line argument.

{`ShowFile.java`} ≡

```
<ShowFile Initial Comments>
<Import java.io>
class ShowFile {
  public static void main (String[] args) {
    <ShowFile Instance Variable Declarations>
    <ShowFile Open a File>
```

```
    <ShowFile Read a File>
    <ShowFile Close a File>
  }
}
```

The following table lists called chunk definition points.

Chunk name	First definition point
<*Import java.io*>	See "Import `java.io`", page 44.
<*ShowFile Close a File*>	See "ShowFile Close a File", page 51.
<*ShowFile Initial Comments*>	See "ShowFile Initial Comments", page 50.
<*ShowFile Instance Variable Declarations*>	See "ShowFile Instance Variable Declarations", page 50.
<*ShowFile Open a File*>	See "ShowFile Open a File", page 51.
<*ShowFile Read a File*>	See "ShowFile Read a File", page 51.

7.5.2.1 ShowFile Initial Comments

<*ShowFile Initial Comments*> ≡

```
/* Display a text file.
   To use this program, specify the name
   of the file that you want to see.
   For example, to see a file called TEST.TXT,
   use the following command line:

   java ShowFile TEST.TXT
 */
```

This chunk is called by the following chunks:

Chunk name	First definition point
{`ShowFile.java`}	See "Demonstration Reading From a File", page 49.
{`ShowFileAlt.java`}	See "`close()` Within `finally` Block", page 52.
{`ShowFileSingleTry`}	See "Demonstration Reading From a File with a Single `try` Block", page 53.

7.5.2.2 ShowFile Instance Variable Declarations

<*ShowFile Instance Variable Declarations*> ≡

```
int i;
FileInputstream fin;
```

This chunk is called by the following chunks:

Chunk name	First definition point
{`ShowFile.java`}	See "Demonstration Reading From a File", page 49.
{`ShowFileAlt.java`}	See "`close()` Within `finally` Block", page 52.
{`ShowFileSingleTry`}	See "Demonstration Reading From a File with a Single `try` Block", page 53.

7.5.2.3 ShowFile Open a File

Notice all of the `try/catch` blocks that handle the I/O errors that might occur. Each I/O operation is monitored for exceptions, and if an exception occurs, it is handled.

<*ShowFile Open a File*> ≡

```
// First, confirm that a filename has been specified.
if (args.length != 1) {
  System.out.printli("Usage: ShowFile filename");
  return;
}

// Attempt to open the file
try {
  fin = new FileInputStream(args[0]);
} catch (FileNotFoundException e) {
  System.out.println("Cannot Open File");
  return;
}
```

This chunk is called by the following chunks:

Chunk name	First definition point
{`ShowFile.java`}	See "Demonstration Reading From a File", page 49.
{`ShowFileAlt.java`}	See "`close()` Within `finally` Block", page 52.
{`ShowFileSingleTry`}	See "Demonstration Reading From a File with a Single `try` Block", page 53.

7.5.2.4 ShowFile Read a File

<*ShowFile Read a File*> ≡

```
// At this point, the file is open and can be read.
// The following reads characters until EOF is encountered.
try {
  do {
    i = fin.read();
    if (i != -1) System.out.print ((char) i);
  } while (i != -1);
} catch (IOException e) {
  System.out.println("Error Reading File");
}
```

This chunk is called by {`ShowFile.java`}; see its first definition at "Demonstration Reading From a File", page 49.

7.5.2.5 ShowFile Close a File

<*ShowFile Close a File*> ≡

```
// Close the file
try {
```

```
    fin.close();
  } catch (IOException e) {
    System.out.println("Error Closing File");
  }
```

This chunk is called by the following chunks:

Chunk name	First definition point
{ShowFile.java}	See "Demonstration Reading From a File", page 49.
{ShowFileAlt.java}	See "`close()` Within `finally` Block", page 52.

7.5.2.6 `close()` Within `finally` Block

Although the preceding example closes the file stream after the file is read, there is a variation that is often useful. The variation is to call `close()` within a `finally` block. In this approach, all of the methods that access the file are contained within a `try` block, and the `finally` block is used to close the file. This way, no matter how the `try` block terminates, the file is closed.

One advantage tothis approach in general is that if the code that accesses a file terminates because of some non-I/O related exception, the file is still closed by the `finally` block.

Here is how the `try` block that reads the file can be recode:

{ShowFileAlt.java} ≡

```
  <ShowFile Initial Comments>
  <Import java.io>
  class ShowFileAlt {
    public static void main (String[] args) {
      <ShowFile Instance Variable Declarations>
      <ShowFile Open a File>
      <ShowFileAlt Read a File>
      <ShowFile Close a File>
    }
  }
```

The following table lists called chunk definition points.

Chunk name	First definition point
<*Import java.io*>	See "Import `java.io`", page 44.
<*ShowFile Close a File*>	See "ShowFile Close a File", page 51.
<*ShowFile Initial Comments*>	See "ShowFile Initial Comments", page 50.
<*ShowFile Instance Variable Declarations*>	See "ShowFile Instance Variable Declarations", page 50.
<*ShowFile Open a File*>	See "ShowFile Open a File", page 51.
<*ShowFileAlt Read a File*>	See "`close()` Within `finally` Block", page 52.

ShowFileAlt Read a File

<*ShowFileAlt Read a File*> ≡

```
  try {
    do {
      i = fin.read();
```

```
    if (i != -1) System.out.print ((char) i);
  } while (i != -1);
} catch (IOException e) {
  System.out.println("Error Reading File.");
} finally {
  // Close the file on the way out of the block.
  try {
    fin.close();
  } catch (IOException e) {
    System.out.println ("Error Closing File.");
  }
}
```

This chunk is called by {`ShowFileAlt.java`}; see its first definition at "`close()` Within `finally` Block", page 52.

7.5.3 Demonstration Reading From a File with a Single `try` Block

Sometimes it's easier to wrap the portions of a program that open a file and access the file within a single `try` block (rather than separating the two) and then use a `finally` block to close the file.

Here is another way to write the `ShowFile` program:

{`ShowFileSingleTry`} ≡

```
<ShowFile Initial Comments>
<ShowFileSingleTry Additional Initial Comment>
<Import java.io>
class ShowFileSingleTry {
  public static void main (String[] args) {
    <ShowFile Instance Variable Declarations>
    <ShowFile Open a File>
    <ShowFileSingleTry Read a File>
  }
}
```

The following table lists called chunk definition points.

Chunk name	**First definition point**
<*Import java.io*>	See "Import `java.io`", page 44.
<*ShowFile Initial Comments*>	See "ShowFile Initial Comments", page 50.
<*ShowFile Instance Variable Declarations*>	See "ShowFile Instance Variable Declarations", page 50.
<*ShowFile Open a File*>	See "ShowFile Open a File", page 51.
<*ShowFileSingleTry Additional Initial Comment*>	See "ShowFile SingleTry Additional Initial Comment", page 53.
<*ShowFileSingleTry Read a File*>	See "ShowFileSingleTry Read a File", page 54.

7.5.3.1 ShowFile SingleTry Additional Initial Comment

<*ShowFileSingleTry Additional Initial Comment*> ≡

```
/* This variation wraps the code that opens and
```

```
    accesses the file within a single try block.
    The file is closed by the finally block.
 */
```

This chunk is called by {ShowFileSingleTry}; see its first definition at "Demonstration Reading From a File with a Single try Block", page 53.

7.5.3.2 ShowFileSingleTry Read a File

In this approach, `fin` is initialized to `null`. In the `finally` block, the file is closed only if `fin` is not `null`. This works because `fin` will be non-`null` only if the file is successfully opened. Thus, `close()` is not called if an exception occurrs while opening the file.

<*ShowFileSingleTry Read a File*> ≡

```
// The following code opens a file, reads characters until EOF
// is encountered, and then closes the file via a finally block.

try {
  fin = new FileInputStream(args[0]);

  do {
    i = fin.read();
    if (i != -1) System.out.print((char) i);
  } while (i != -1);

} catch (FileNotFoundException e) {
  System.out.println("File Not Found.");

} catch (IOException e) {
  System.out.println("An I/O Error Occurred");

} finally {
  // Close file in all cases
  try {
    if (fin != null) fin.close();
  } catch (IOException e) {
    System.out.println("Error Clsoing File");
  }
}
```

This chunk is called by {ShowFileSingleTry}; see its first definition at "Demonstration Reading From a File with a Single try Block", page 53.

More Compact Catch Code

It is possible to make the `try/catch` sequence a bit more compact. Because `FileNotFoundException` is a subclass of `IOException`, it need not be caught separately. Here is the sequence recoded to eliminate catching `FileNotFoundException`. In this case, the standard exception message, which describes the error, is displayed.

```
} catch (IOException e) {
  System.out.println("I/O Error: " + e);
} finally {
  try {
    if (fin != null) fin.close();
  } catch (IOException e) {
    System.out.println("Error Closing File");
  }
}
```

In this approach, any error, including an error opening the file, is simply handled by the single `catch` statement. This approach may not be appropriate in cases in which you want to deal separately with a failure to open a file, such as might be caused by a mistyped filename. In such a situation, you might want to prompt for the correct name before entering a `try` block that accesses the file.

The next example uses `write()` to copy a file.

7.5.4 Demonstration Writing to a File

This example uses `write()` to copy a file.

Notice that all potential I/O errors are handled in the programs by the use of exceptions. This differs from some computer languages that use error codes to report file errors. They enable Java to easily differentiate the end-of-file condition from file errors when input is being performed.

{`CopyFile.java`} ≡

```
<CopyFile Initial Comments>
<Import java.io>
class CopyFile {
  public static void main (String[] args) {
    <CopyFile Instance Variable Declarations>
    <CopyFile Check For 2 Files>
    <CopyFile Copy a File>
  }
}
```

The following table lists called chunk definition points.

Chunk name	**First definition point**
<*CopyFile Check For 2 Files*>	See "CopyFile Check for 2 Files", page 56.
<*CopyFile Copy a File*>	See "CopyFile Copy a File", page 56.
<*CopyFile Initial Comments*>	See "CopyFile Initial Comments", page 55.
<*CopyFile Instance Variable Declarations*>	See "CopyFile Instance Variable Declarations", page 56.
<*Import java.io*>	See "Import `java.io`", page 44.

7.5.4.1 CopyFile Initial Comments

<*CopyFile Initial Comments*> ≡

```
/* Copy a file
   To use this program, specify the name
```

```
    of the source file and the destination file.
    For example, to copy a file called FIRST.TXT
    to a file called SECOND.TXT, use the following
    command line.

    java CopyFile FIRST.TXT SECOND.TXT
  */
```

This chunk is called by {CopyFile.java}; see its first definition at "Demonstration Writing to a File", page 55.

7.5.4.2 CopyFile Instance Variable Declarations

<*CopyFile Instance Variable Declarations*> ≡

```
int i;
FileInputStream fin = null;
FileOutputStream fout = null;
```

This chunk is called by {CopyFile.java}; see its first definition at "Demonstration Writing to a File", page 55.

7.5.4.3 CopyFile Check for 2 Files

<*CopyFile Check For 2 Files*> ≡

```
// First, confirm that both files have been specified
if (args.length != 2) {
  System.out.println("Usage: CopyFile from to");
  return;
}
```

This chunk is called by {CopyFile.java}; see its first definition at "Demonstration Writing to a File", page 55.

7.5.4.4 CopyFile Copy a File

Notice that there are two separate `try` blocks used when closing the files. This ensures that both files are closed, even if the call to `fin.close()` throws an exception.

<*CopyFile Copy a File*> ≡

```
// Copy a file
try {
  // Attempt to open the files
  fin = new FileInputStream(args[0]);
  fout = new FileOutputStream(args[1]);

  do {
    i = fin.read();
    if (i != -1) fout.write(i);
  } while (i != -1);
```

```
} catch (IOException e) {
  System.out.println("I/O Error: " + e);

} finally {
  try {
    if (fin != null) fin.close();
  } catch (IOException e2) {
    System.out.println("Error Closing Input File");
  }

  try {
    if (fout != null) fout.close();
  } catch (IOException e2) {
    System.out.println("Error Closing Output File");
  }
}
```

This chunk is called by {`CopyFile.java`}; see its first definition at "Demonstration Writing to a File", page 55.

7.6 Automatically Closing Files

JDK 7 added a feature that offers another way to manage resources,such as file streams, by automating the closing process. This feature, sometimes referred to as *automatic resource management*, or ARM for short, is based on an expanded version of the `try` statement. This form of `try` is called the `try-with-resources` statement. The principal advantage of automatic resource management is that it prevents situations in which a file (or other resource) is inadvertently not released after it is no longer needed.

Here is its general form:

```
try (resource-specification) {
  // use the resource
}
```

GeneralForm 7.1: General Form Automatic Resource Management

Typically, *resource-specification* is a statement that declares and initializes, such as a file stream. It consists of a variable declaration in which the variable is initialized with a reference to the object begin managed. When the `try` block ends, the resource is automatically released. In the case of a file, this means that the file is automatically closed. This form of `try` can also include `catch` and `finally` clauses. A resource declared in the `try` statement is implicitly `final`. This means that you can't assign to the resource after it has been created. The scope of the resource is limited to the `try-with-resources` statement.

Beginning with JDK 9, it is also possible for the resource specification of the `try` to consist of a variable that has been declared and initialized earlier in the program. However, that variable must be effectively final, which means that it has not been assigned a new value after being given its initial value.

`AutoCloseable` Interface

The `try-with-resources` statement can be used only with those resources that implement the `AutoCloseable` interface defined by `java.lang`. This interface defines the `close()` method. `AutoCloseable` is inherited by the `Closeable` interface in `java.io`. Both interfaces are implmented by the stream classes. Thus, `try-with-resources` can be used when working with streams, including file streams.

Multiple Resources

You can manage more than one resource within a single `try` statement. Simply separate each resource specification with a semicolon.

7.6.1 Demonstration of Automatically Closing a File

Here is a reworked verion of the `ShowFile` program using `try-with-resources`.

{`ShowFileTryWR.java`} ≡

```
<ShowFileTryWR Initial Comments>
<Import java.io>
class ShowFileTryWR {
  public static void main (String[] args) {
    <ShowFileTryWR Instance Variable Declaration>
    <ShowFileTryWR Check CL Args> <Number 1> <ShowFileTryWR Check CL Args End>
    <ShowFileTryWR Open a File TryWR>
  }
}
```

The following table lists called chunk definition points.

Chunk name	**First definition point**
<*Import java.io*>	See "Import `java.io`", page 44.
<*Number 1*>	See "Check CL Args", page 59.
<*ShowFileTryWR Check CL Args*>	See "Check CL Args", page 59.
<*ShowFileTryWR Check CL Args End*>	See "Check CL Args", page 59.
<*ShowFileTryWR Initial Comments*>	See "Initial Comments", page 58.
<*ShowFileTryWR Instance Variable Declaration*>	See "Instance Variable Declaration", page 59.
<*ShowFileTryWR Open a File TryWR*>	See "Open a File TryWR", page 59.

7.6.1.1 Initial Comments

<*ShowFileTryWR Initial Comments*> ≡

```
/* This version of teh ShowFile program uses a try-with-resources
   statement to automatically close a file after it is no longer needed.
 */
```

This chunk is called by {`ShowFileTryWR.java`}; see its first definition at "Demonstration of Automatically Closing a File", page 58.

7.6.1.2 Instance Variable Declaration

<*ShowFileTryWR Instance Variable Declaration*> ≡

```
int i;
```

This chunk is called by the following chunks:

Chunk name	First definition point
{CopyFileMultTryWR.java}	See "Demonstration of Multiple Resources", page 60.
{ShowFileTryWR.java}	See "Demonstration of Automatically Closing a File", page 58.

7.6.1.3 Check CL Args

<*ShowFileTryWR Check CL Args*> ≡

```
// First, confirm that a filename has been specified.
if (args.length !=
```

This chunk is called by the following chunks:

Chunk name	First definition point
{CopyFileMultTryWR.java}	See "Demonstration of Multiple Resources", page 60.
{ShowFileTryWR.java}	See "Demonstration of Automatically Closing a File", page 58.

<*ShowFileTryWR Check CL Args End*> ≡

```
) {
  System.out.println("Usage: ShowFile filename");
  return;
}
```

This chunk is called by the following chunks:

Chunk name	First definition point
{CopyFileMultTryWR.java}	See "Demonstration of Multiple Resources", page 60.
{ShowFileTryWR.java}	See "Demonstration of Automatically Closing a File", page 58.

<*Number 1*> ≡

```
1
```

This chunk is called by {ShowFileTryWR.java}; see its first definition at "Demonstration of Automatically Closing a File", page 58.

7.6.1.4 Open a File TryWR

Pay special attention to how the file is opened within the `try` statement. The resource-specification portion of the `try` declares a `FileInputStream` called `fin`, which is then assigned a reference to the file opened by its constructor. Therefore, here, the variable `fin` is local to the `try` block, being created when the `try` is entered. When the `try` is left, the stream associated with `fin` is automatically closed by an implicit call to `close()`. Since you don't call `close()` explicitly, you can't forget to close the file. This is a key advantage of using `try-with-resources`.

<*ShowFileTryWR Open a File TryWR*> ≡

```
/* The following code uses a try-with-resources statement to open
```

```
    a file and then automatically close it when the try block is left. */

try (FileInputStream fin = new FileInputStream(args[0])) {

  do {
    i = fin.read();
    if (i != -1) System.out.print((char) i);
  } while (i != -1);

} catch (FileNotFoundException e) {
  System.out.println("File Not Found.");

} catch (IOException e) {
  System.out.println("An I/O Error Occurred.");
}
```

This chunk is called by {`ShowFileTryWR.java`}; see its first definition at "Demonstration of Automatically Closing a File", page 58.

7.6.2 Demonstration of Multiple Resources

The following program shows an example of handling multiple resources in a single `try` statement. It reworks the `CopyFile` program shown earlier so that it uses a single `try-with-resources` statement to manage both `fin` and `fout`.

{`CopyFileMultTryWR.java`} ≡

```
<CopyFileMultTryWR Initial Comments>
<Import java.io>
class CopyFileMultTryWR {
  public static void main (String[] args) throws IOException {
    <ShowFileTryWR Instance Variable Declaration>
    <ShowFileTryWR Check CL Args> <Number 2> <ShowFileTryWR Check CL Args End>
    <CopyFileMultTryWR Manage Two Files>
  }
}
```

The following table lists called chunk definition points.

Chunk name	**First definition point**
<*CopyFileMultTryWR Initial Comments*>	See "CopyFileMultTryWR Initial Comments", page 61.
<*CopyFileMultTryWR Manage Two Files*>	See "CopyFileMultTryWR Manage Two Files", page 61.
<*Import java.io*>	See "Import `java.io`", page 44.
<*Number 2*>	See "Demonstration of Multiple Resources", page 61.
<*ShowFileTryWR Check CL Args*>	See "Check CL Args", page 59.
<*ShowFileTryWR Check CL Args End*>	See "Check CL Args", page 59.
<*ShowFileTryWR Instance Variable Declaration*>	See "Instance Variable Declaration", page 59.

<*Number 2*> ≡

```
2
```

This chunk is called by {`CopyFileMultTryWR.java`}; see its first definition at "Demonstration of Multiple Resources", page 60.

7.6.2.1 CopyFileMultTryWR Initial Comments

<*CopyFileMultTryWR Initial Comments*> ≡

```
/* A version of CopyFile that uses try-with-resources.
   It demonstrates two resources (in this case files) being
   managed by a single try statement
 */
```

This chunk is called by {`CopyFileMultTryWR.java`}; see its first definition at "Demonstration of Multiple Resources", page 60.

7.6.2.2 CopyFileMultTryWR Manage Two Files

Note how the input and output files are opened within the `try` block. After this block ends, both `fin` and `fout` will have been closed. This code is much shorter. The ability to streamline source code is a side-benefit of automatic resouce management.

<*CopyFileMultTryWR Manage Two Files*> ≡

```
// Open and manage two files via the try statement.
try (FileInputStream fin = new FileInputStream(args[0]);
     FileOutputStream fout = new FileOutputStream(args[1])) {

  do {
    i = fin.read();
    if (i != -1) fout.write(i);
  } while (i != -1);

} catch (IOException e) {
  System.out.println("I/O Error: " + e);
}
```

This chunk is called by {`CopyFileMultTryWR.java`}; see its first definition at "Demonstration of Multiple Resources", page 60.

8 Miscellaneous Java Keywords

This chapter looks at several more Java keywords:

- `volatile`
- `instanceof`
- `native`
- `strictfp`
- `assert`

8.1 The `transient` and `volative` Modifiers

8.2 Using `instanceof`

8.3 `strictfp`

8.4 Native Methods

8.5 Using `assert`

8.6 Static Import

8.7 Invoking Overloaded Constructors Through `this()`

8.8 Compact API Profiles

9 Generics

Generics, introduced in J2SE 5.0, allows a type or method to operate on objects of various types while providing compile-time type safety. It adds compile-time type safety to the Collections Framework and eliminates the need of casting. In other words, generics allow you to abstract over types.

Through the use of generics, it is possible to create classes, interfaces, and methods that will work in a type-safe manner with various kinds of data. Many algorithms are logically the same no matter what type of data they are being applied to. For example, the mechanism that supports a stack is the same whether that stack is storing items of type `Integer`, `String`, `Object`, or `Thread`. With generics, you can define an algorithm once, independently of any specific type of data, and then apply that algorithm to a wide variety of data types without any additional effort.

Perhaps the one feature of Java that has been most significantly affected by generics is the *Collections Framework*. A *collection* is a group of objects. The Collections Framework defines several classes, such as lists and maps, that manage collections. The collection classes have always been able to work with any type of object. The benefit that generics adds is that the collection classes can now be used with complete type safety.

This chapter describes the syntax, theory, and use of generics. It also shows how generics provide type safety for some previously difficult cases.

9.1 Motivation for Generics

Code Fragment Without Generics

Here is a typical code fragment abstracting over types by using `Object` and type casting.

```
List myIntList = new LinkedList(); // 1
myIntList.add(new Integer(0)); // 2
Integer x = (Integer) myIntList.iterator().next(); // 3
```

The cast on line 3 is annoying, although essential. The compiler can guarantee only that an `Object` will be returned by the iterator. This therefore adds both clutter and the possibility of a run-time error.

Code Fragment with Generics

Generics allow a programmer to mark their intent to restrict a list to a particular data type. Here is a version of the same code that uses generics.

```
List<Integer> myIntList = new LinkedList<Integer>(); // 1'
myIntList.add(new Integer(0)); // 2'
Integer x = myIntList.iterator().next(); // 3'
```

In line 1, the type declaration for the variable `myIntList` specifies that it is to hold a `List` of `Integer`s: '`List<Integer>`'. `List` is a *generic interface* that takes a *type parameter* (`Integer`). The type parameter is also specified when creating the `List` object ('`new LinkedList<Integer>()`'). Also, the cast on line 3 is gone.

So has this just moved the clutter around, from a type cast to a type parameter? No, because this has given the compiler the ability to check the type correctness of the program

at compile-time. When we say that `myIntList` is declared with type `List<Integer>`, this tells us something about the variable `myIntList`, which holds true wherever and whenever it is used, and the compiler will guarantee it. In contrast, the cast tells us something the programmer thinks is true at a single point in the code.

The net effect, especially in large programs, is improved readability and robustness.

9.2 What Are Generics

The term *generics* means *parameterized types*. Parameterized types are important because they enable you to create classes, interfaces, and methods in whicht the type of data upon which they operate is specified as a parameter. Using generics, it is possible to create a single class, for example, that automatically works with different types of data. A class, interface, or method that operates on a parameterized type is called *generic*, as in *generic class* or *generic method*.

Java has always given the ability to create generalized classes, interfaces, and methods by operating through references of type `Object`. Generics added the type safety that was lacking. They also streamlined the process, because it is no longer necessary to explicitly employ casts to translate between `Object` and the type of data that is being operate upon. Wtih generics, all casts are automatic and implicit.

9.3 A Simple Generics Example

The following program defines two classes. The first is the generic class `Gen`, and the second is `GenDemo`, which uses `Gen`.

{`SimpleGenerics.java`} ≡

<*Class Gen*>
<*Class GenDemo*>

The following table lists called chunk definition points.

Chunk name	**First definition point**
<*Class Gen*>	See "Class Gen¡T¿", page 65.
<*Class GenDemo*>	See "Class GenDemo", page 66.

9.3.1 Class Gen<T>

This is a simple generic class. The class `Gen` is declared with a parameter of '`<T>`':

```
class Gen<T> {
```

'`T`' is the name of a *type parameter*. This name is used as a placeholder for the actual type that will be passed to `Gen` when an object is created. Thus, '`T`' is used within `Gen` whenever the type parameter is needed.

Notice that '`T`' is contained within '`< >`'. This syntax can be generalized. Whenever a type parameter is being declared, it is specified within angle brackets.

Because `Gen` uses a type parameter, `Gen` is a *generic class*, which is also called a *parameterized type*.

Outline of Class Gen<T>

Class `Gen` contains four parts:

- an instance variable declaration
- a constructor
- a method returning the instance variable
- a method describing the type of the instance variable

<*Class Gen*> ≡

```
class Gen<T> {
  <Instance Variable ob of Type T>
  <Constructor taking parameter of Type T>
  <Method returning object of type T>
  <Method showing type of T>
}
```

This chunk is called by {`SimpleGenerics.java`}; see its first definition at "A Simple Generics Example", page 64.

The following table lists called chunk definition points.

Chunk name	First definition point
<*Constructor taking parameter of Type T*>	See "Class Gen¡T¿", page 65.
<*Instance Variable ob of Type T*>	See "Class Gen¡T¿", page 65.
<*Method returning object of type T*>	See "Class Gen¡T¿", page 66.
<*Method showing type of T*>	See "Class Gen¡T¿", page 66.

Implementation of Class Gen<T>

'`T`' is used to declare an object called `ob`. '`T`' is a placeholder for the actual type that will be specified when a `Gen` object is created. Thus, `ob` will be an object of the type passed to '`T`'.

<*Instance Variable ob of Type T*> ≡

```
T ob;   // declare an object of type T
```

This chunk is called by <*Class Gen*>; see its first definition at "Class Gen¡T¿", page 65.

The Constructor

Here is the constructor for `Gen`. Notice that its parameter, `o`, is of type '`T`'. This means that the actual type of `o` is determined by the type passed to '`T`' when a `Gen` object is created. Because both the parameter `o` and the member variable `ob` are of type '`T`', they will both be the same actual type when a `Gen` object is created.

<*Constructor taking parameter of Type T*> ≡

```
// Pass the constructor a reference to
// an object of type T
Gen (T o) {
  ob = o;
}
```

This chunk is called by <*Class Gen*>; see its first definition at "Class Gen¡T¿", page 65.

Instance Methods `getob()` and `showType()`

The type parameter '`T`' can also be used to specify the return type of a method, as here in `getob()`. Because `ob` is also of type '`T`', its type is compatible wih the return type specified by `getob()`.

<*Method returning object of type T*> ≡

```
// Return ob
T getob() {
  return ob;
}
```

This chunk is called by <*Class Gen*>; see its first definition at "Class Gen¡T¿", page 65.

The method `showType()` displays the type of '`T`' by calling `getName()` on the `Class` object return by the call to `getClass()` on `ob`. The `getClass()` method is defined by `Object` and is thus a member of *all* class types. It returns a `Class` object that corresponds to the type of the class of the object on which it is called. `Class` defines the `getName()` method, which returns a string representation of the class name.

<*Method showing type of T*> ≡

```
// Show type of T
void showType() {
  System.out.println("Type of T is " + ob.getClass().getName();
}
```

This chunk is called by <*Class Gen*>; see its first definition at "Class Gen¡T¿", page 65.

9.3.2 Class GenDemo

The `GenDemo` class demonstrates the generic `Gen` class.

But first, take note: The Java compiler does not actually create different versions of `Gen`, or of any other generic class. The compiler removes all generic type information, substituting the necessary casts, to make your code **behave as if** a specific version of `Gen` were created. There is really only one version of `Gen` that actually exists.

The process of removing generic type information is called *type erasure*.

`GenDemo` first creates a version of `Gen` for integers and calls the methods defined in `Gen` on it. It then does the same for a String object.

<*Class GenDemo*> ≡

```
// Demonstrate the generic class
class GenDemo {
  public static void main(String args[]) {
    <Create a Gen object for Integers>
    <Create a Gen object for Strings>
  }
}
```

This chunk is called by {`SimpleGenerics.java`}; see its first definition at "A Simple Generics Example", page 64.

The following table lists called chunk definition points.

Chunk name	First definition point
<*Create a Gen object for Integers*>	See "Implementation of Class GenDemo with Type Integer", page 67.
<*Create a Gen object for Strings*>	See "Implementation of Class GenDemo with Type String", page 68.

9.3.2.1 Implementation of Class GenDemo with Type Integer

<*Create a Gen object for Integers*> ≡

```
<Integer Type Parameter>
<Reference to Integer Instance>
<Show Type>
<Get Value>
```

This chunk is called by <*Class GenDemo*>; see its first definition at "Class GenDemo", page 66.

The following table lists called chunk definition points.

Chunk name	First definition point
<*Get Value*>	See "Implementation of Class GenDemo with Type Integer", page 68.
<*Integer Type Parameter*>	See "Implementation of Class GenDemo with Type Integer", page 67.
<*Reference to Integer Instance*>	See "Implementation of Class GenDemo with Type Integer", page 68.
<*Show Type*>	See "Implementation of Class GenDemo with Type Integer", page 68.

Integer Type Declaration

A reference to an Integer is declared in `iOb`. Here, the type '`Integer`' is specified within the angle brackets after `Gen`. '`Integer`' is a *type argument* that is passed to `Gen`'s type parameter, '`T`'. This effectively creates a version of `Gen` in which all references to '`T`' are translated into references to '`Integer`'. Thus, `ob` is of type '`Integer`', and the return type of `getob()` is of type '`Integer`'.

<*Integer Type Parameter*> ≡

```
Gen<Integer> iOb;
```

This chunk is called by <*Create a Gen object for Integers*>; see its first definition at "Implementation of Class GenDemo with Type Integer", page 67.

Reference Assignment

The next line assigns to `iOb` a reference to an instance of an '`Integer`' version of the `Gen` class. When the `Gen` constructor is called, the type argument '`Integer`' is also specified. This is because the type of the object (in this case `iOb` to which the reference is being assigned is of type `Gen<Integer>`. Thus, the reference returned by `new` must also be of type `Gen<Integer>`. If it isn't, a compile-time error will result. This type checking is one of the main benefits of generics because it ensures type safety.

Notice the use of autoboxing to encapsulate the value 88 within an Integer object.

<*Reference to Integer Instance*> ≡

```
iOb = new Gen<Integer>(88);
```

This chunk is called by <*Create a Gen object for Integers*>; see its first definition at "Implementation of Class GenDemo with Type Integer", page 67.

The automatic autoboxing could have been written explicitly, like so:

```
iOb = new Gen<Integer>(Integer.valueOf(88));
```

but there would be no value to doing it that way.

Showing the Reference's Type

The program then uses `Gen`'s instance method to show the type of `ob`, which is an '`Integer`' in this case.

<*Show Type*> ≡

```
iOb.showType();
```

This chunk is called by <*Create a Gen object for Integers*>; see its first definition at "Implementation of Class GenDemo with Type Integer", page 67.

Showing the Reference's Value

The program now obtains the value of `ob` by assiging `ob` to an '`int`' variable. The return type of `getob()` is '`Integer`', which unboxes into '`int`' when assigned to an '`int`' variable (`v`). There is no need to cast the return type of `getob()` to '`Integer`'.

<*Get Value*> ≡

```
int v = iOb.getob();
System.out.println("value: " + v);
System.out.println();
```

This chunk is called by <*Create a Gen object for Integers*>; see its first definition at "Implementation of Class GenDemo with Type Integer", page 67.

9.3.2.2 Implementation of Class GenDemo with Type String

<*Create a Gen object for Strings*> ≡

```
// Create a Gen object for Strings.
Gen<String> strOb = new Gen<String>("Generics Test");

// Show the type of data used by strOb
strOb.showType();

// Get the value of strOb.  Again, notice
// that no cast is needed.
String str = strOb.getob();
System.out.println("value: " + str);
```

This chunk is called by <*Class GenDemo*>; see its first definition at "Class GenDemo", page 66.

9.4 Notes About Generics

9.4.1 Generics Work Only with Reference Types

When declaring an instance of a generic type, the type argument passed to the type parameter must be a reference type. It cannot be a primitive type, such as '`int`' or '`char`'.

You can use the type wrappers to encapsulate a primitive type. Java's autoboxing and auto-unboxing mechanism makes the use of the type wrapper transparent.

9.4.2 Generic Types Differ Based on their Type Arguments

A reference of one specific version of a generic type is not type-compatible with another version of the same generic type. In other words, the following line of code is an error and will not compile:

```
iOb = strOb; // Gen<Integer> != Gen<String>
```

These are references to different types because their type arguments differ.

9.4.3 Generics and Subtyping

Is the following legal?

```
List<String> ls = new ArrayList<String>(); // 1
List<Object> lo = ls; // 2
```

Line 1 is legal. What about line 2? This boils down to the question: "is a List of String a List of Object." Most people instinctively answer, "Sure!"

Now look at these lines:

```
lo.add(new Object()); // 3
String s = ls.get(0); // 4: Attempts to assign an Object to a String!
```

Here we've aliased `ls` and `lo`. Accessing `ls`, a list of `String`, through the alias `lo`, we can insert arbitrary objects into it. As a result `ls` does not hold just `Strings` anymore, and when we try and get something out of it, we get a rude surprise.

The Java compiler will prevent this from happening of course. Line 2 will cause a compile time error.

The take-away is that, if `Foo` is a subtype (subclass or subinterface) of `Bar`, and `G` is some generic type declaration, it is not the case that `G<Foo>` is a subtype of `G<Bar>`.

9.4.4 How Generics Improve Type Safety

Generics automatically ensure the type safety of all operations involving a generic class, such as `Gen`. They eliminate the need for the coder to enter cases and to type-check code by hand.

9.5 A Generic Class with Two Type Parameters

You can declare more than one type parameter in a generic type. To specify two or more type parameters, use a comma-separated list. When an object is created, the same number of type arguments must be passed as there are type parameters. The type arguments can be the same or different.

9.5.1 Example of Code with Two Type Parameters

{`TwoTypeParameters.java`} ≡

<*Class TwoGen*>
<*Class SimpGen*>

The following table lists called chunk definition points.

Chunk name	First definition point
<*Class SimpGen*>	See "Class SimpGen", page 71.
<*Class TwoGen*>	See "Class TwoGen", page 70.

9.5.1.1 Class TwoGen

<*Class TwoGen*> ≡

<*Class Declaration*>
  <*Two Instance Variables Declarations*>
  <*Constructor of Two Parameters*>
  <*Instance Methods Show and Get*>

This chunk is called by {`TwoTypeParameters.java`}; see its first definition at "Example of Code with Two Type Parameters", page 70.

The following table lists called chunk definition points.

Chunk name	First definition point
<*Class Declaration*>	See "Class TwoGen", page 70.
<*Constructor of Two Parameters*>	See "Class TwoGen", page 70.
<*Instance Methods Show and Get*>	See "Class TwoGen", page 71.
<*Two Instance Variables Declarations*>	See "Class TwoGen", page 70.

Class Declaration

Notice how `TwoGen` is declared. It specifies two type parameters: '`T`' and '`V`', separated by a comma. Because it has two type parameters, two type arguments must be passed to `TwoGen` when an object is created.

<*Class Declaration*> ≡

```
class TwoGen<T, V> {
```

This chunk is called by <*Class TwoGen*>; see its first definition at "Class TwoGen", page 70.

Instance Variables Declarations

<*Two Instance Variables Declarations*> ≡

```
T ob1;
V ob2;
```

This chunk is called by <*Class TwoGen*>; see its first definition at "Class TwoGen", page 70.

Constructor

<*Constructor of Two Parameters*> ≡

```
TwoGen(T o1, V 02) {
  ob1 = o1;
```

```
    ob2 = o2;
  }
```

This chunk is called by <*Class TwoGen*>; see its first definition at "Class TwoGen", page 70.

Instance Methods Show and Get

<*Instance Methods Show and Get*> ≡

```
  void showTypes() {
    System.out.println("Type of T is " + ob1.getClass().getName());
    System.out.println("Type of V is " + ob2.getClass().getName());
  }

  T getob1() {
    return ob1;
  }

  V getob2() {
    return ob2;
  }
```

This chunk is called by <*Class TwoGen*>; see its first definition at "Class TwoGen", page 70.

9.5.1.2 Class SimpGen

Two type arguments must be supplied to the constructor. In this case, the two type parameters are '`Integer`' and '`String`'.

<*Class SimpGen*> ≡

```
class SimpGen {
  public static void main(String args[]) {
    TwoGen<Integer, String> tgObj =
      new TwoGen<Integer, String>(88, "Generics");

    // Show the types
    tgObj.showTypes();

    // Obtain and show values
    int v = tgObj.getob1();
    System.out.println("value: " + v);

    String str = thObj.getob2();
    System.out.println("value: " + str);
  }
}
```

This chunk is called by {`TwoTypeParameters.java`}; see its first definition at "Example of Code with Two Type Parameters", page 70.

9.6 The General Form of a Generic Class

The generics syntax shown above can be generalized. Here is the syntax for declaring a generic class:

class *class-name*<*type-param-list*> { . . .

Here is the full syntax for declaring a reference to a generic class and instance creation:

class-name<*type-arg-list*> *var-name* =
new *class-name*<*type-arg-list*>(*cons-arg-list*);

GeneralForm 9.1: General Form for Declaring and Creating a Reference to a Generic Class

9.7 Bounded Types

Sometimes it can be useful to limit the types that can be passed to a type parameter. Java provides *bounded types*. When specifying a type parameter, you can create an upper bound that declares the superclass from which all type arguments must be derived. This is accomplished through the use of an `extends` clause when specifying the type parameter:

<*T* extends *superclass*>

This specifies that *T* can only be replaced by *superclass* or subclasses of *superclass*. Thus, *superclass* defines an inclusive, upper limit.

Interface Type as a Bound

In addition to using a class type as a bound, you can also use an interface type. In fact, youi can specify multiple interfaces as bounds. Furthermore, a bound can include both a class type and one or more interfaces. In this case, the class type must be specified first. When a bound includes an interface type, only type arguments that implement that interface are legal.

When specifying a bound that has a class and an interface, or multiple interfaces, use the `&` operator to connnect them.

```
class Gen<T extends MyClass & MyInterface> { ...
```

Any type argument passed to '`T`' must be a subclass of `MyClass` and implement `MyInterface`.

9.8 Using Wildcard Arguments

9.8.1 Wildcard Motivation

Consider the problem of writing a routine that prints out all the elements in a collection. Here's how you might write it in an older version of the language (i.e., a pre-5.0 release):

```
void printCollection(Collection c) {
    Iterator i = c.iterator();
    for (k = 0; k < c.size(); k++) {
        System.out.println(i.next());
    }
}
```

And here is a naive attempt at writing it using generics (and the new *for loop* syntax):

```
    for (Object e : c) {
        System.out.println(e);
    }
}
```

The problem is that this new version is much less useful than the old one. Whereas the old code could be called with any kind of collection as a parameter, the new code only takes `Collection<Object>`, which, as we've just demonstrated, is *not* a supertype of all kinds of collections!

So what is the supertype of all kinds of collections? It's written *`Collection<?>`* (pronounced *collection of unknown*), that is, a collection whose element type matches anything. It's called a *wildcard type*. We can write:

```
void printCollection(Collection<?> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```

and now, we can call it with any type of collection. Notice that inside `printCollection()`, we can still read elements from `c` and give them type `Object`. This is always safe, since whatever the actual type of the collection, it does contain objects. It isn't safe to add arbitrary objects to it however:

```
Collection<?> c = new ArrayList<String>();
c.add(new Object()); // Compile time error
```

Since we don't know what the element type of `c` stands for, we cannot add objects to it. The `add()` method takes arguments of type `E`, the element type of the collection. When the actual type parameter is `?`, it stands for some unknown type. Any parameter we pass to `add` would have to be a subtype of this unknown type. Since we don't know what type that is, we cannot pass anything in. The sole exception is `null`, which is a member of every type.

On the other hand, given a `List<?>`, we can call `get()` and make use of the result. The result type is an unknown type, but we always know that it is an object. It is therefore safe to assign the result of `get()` to a variable of type `Object` or pass it as a parameter where the type `Object` is expected.

9.8.2 Wildcard Syntax

Sometimes type safety can get in the way of perfectly acceptable constructs. In such cases, there is a *wildcard* argument that can be used. The wildcard argument is specified by the `?`, and it represents an unknown type. It would be used in place of a type parameter, for example:

```
boolean sameAvg(Stats<?> ob) {
  if(average() == ob.average())
    return true;

  return false;
}
```

Here, '`Stats<?>`' matches any `Stats` object (`Integer`, `Double`), allowing any two `Stats` objects to have their averages compared. The wildcard does not affect what type of `Stats` object can be created. That is governed by the `extends` clause in the `Stats` declaration. The wildcard simply matches any *valid* `Stats` object.

9.8.3 Bounded Wildcards

Wildcard arguments can be bounded in much the same way that a type parameter can be bounded (the *bounded wildcard argument*. A bounded wildcard is especially important when you are creating a generic type that will operate on a class hierarchy.

A bounded wildcard specifies either an upper bound or a lower bound for the type argument. This enables you to restrict the types of objects upon which a method will operate.

Upper Bounded Wildcard

The most common bounded wildcard is the upper bound, which is created using an `extends` clause.In general, to establish an upper bound for a wildcard, use the following type of wildcard expression:

<? extends *superclass*>

GeneralForm 9.2: General Form of Upper Bounded Wildcard Syntax

where *superclass* is the name of the class that serves as the upper bound. This is an inclusive clause.

Lower Bounded Wildcard

You can also specify a lower bound for a wildcard by adding a `super` clause to a wildcard declaration. Here is its general form:

<? super *subclass*>

GeneralForm 9.3: General Form of Lower Bounded Wildcard Syntax

Only classes that are superclasses of *subclass* are acceptable arguments

9.9 Creating a Generic Method

It is possible to declare a generic method that uses one or more type parameters of its own. It is also possible to create a generic method that is enclosed within a non-generic class.

Generalized Form

< *type-param-list* > *ret-type meth-name* (*param-list*) { . . .

GeneralForm 9.4: General Form for Declaring a Generic Method

9.9.1 Example of Generic Method

The following program declares a non-generic class called `GenMethDemo` and a static **generic method** within that class called `isIn()`. The `isIn()` method determines if an object is a member of an array. It can be used with any type of object and array as long as the array contains objects that are compatible with the type of the object being sought.

{GenMethDemo.java} ≡

```
class GenMethDemo {
  <Static Method isIn>
  <GenMethDemo Main>
}
```

The following table lists called chunk definition points.

Chunk name	First definition point
<*GenMethDemo Main*>	See "GenMethDemo Main", page 75.
<*Static Method isIn*>	See "Method isIn()", page 75.

9.9.1.1 Method isIn()

The **type parameters** are declared *before* the return type of the method.

<*Static Method isIn*> ≡

```
static <T extends Comparable<T>, V extends T> boolean isIn(T x, V[] y) {

  for (int i = 0; i < y.length; i++)
    if (x.equals(y[i]) return true;

  return false;
}
```

This chunk is called by {GenMethDemo.java}; see its first definition at "Example of Generic Method", page 75.

The type T is **upper-bounded** by the `Comparable` interface, which must be of the same type as T. Likewise, the second type, V, is also **upper-bounded** by T. Thus, V must be either the same type as T or a subclass of T. This relationship enforces that `isIn()` can be called only with arguments that are compatible with each other.

While `isIn()` is static in this case, generic methods can be either static or non-static; there is no restriction in this regard.

Explicitly Including Type Arguments

There is generally no need to specify type arguments when calling this method from within the `main` routine. This is because the type arguments are automatically discerned, and the types of T and V are adjusted accordingly.

Although type inference will be sufficient for most generic method calls, you can explicitly specify the type argument if needed. For example, here is how the first call to `isIn()` looks when the type argumetns are specified:

```
GenMethDemo.<Integer, Integer>isIn(2, nums)
```

9.9.1.2 GenMethDemo Main

<*GenMethDemo Main*> ≡

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

  // call isIn() with Integer type
```

```
    Integer nums[] = { 1, 2, 3, 4, 5 };

    if ( isIn(2, nums) )
      System.out.println("2 is in nums");

    if ( @isIn(7, nums))
      System.out.println("7 is not in nums");

    System.out.println();

    // call isIn() with String type
    String strs[] = { "one", "two", "three", "four", "five" };

    if ( isIn("two", strs))
      System.out.println("two is in strs");

    if ( !isIn("seven", strs))
      System.out.println("seven is not in strs");

    // call isIn() with mixed types
    // WILL NOT COMPILE! TYPES MUST BE COMPATIBLE
    // if ( isIn("two", nums))
    //    System.out.println("two is in nums");
  }
```

This chunk is called by {`GenMethDemo.java`}; see its first definition at "Example of Generic Method", page 75.

9.10 Generic Constructors

It is possible for constructors to be generic, even if their class is not (see "Class Gen¡T¿", page 65). The syntax is the same (type parameters come first).

< *type-param-list*> *constructor-name* (*param-list*) { . . .

10 Enumerations

Enumerations were added by JDK 5. In earlier versions of Java, enumerations were implemented using `final` variables.

An *enumeration* is a list of named constants that define a new data type and its legal values. In other words, an enumeration defines a class type. An *enumeration object* can only hold values that were declared in the list. Other values are not allowed. An enumeration allows the programmer to define a set of values that a data type can legally have.

By making enumerations classes, the capabilities of the enumeration are greatly expanded. An enumeration can have:

- constructors
- methods
- instance variables

10.1 Enumeration Basics

An enumeration is created using the `enum` keyword.

```
enum Apple {
     Jonathon, GoldenDel, RedDel, Winesap, Cortland
}
```

enumeration constants

The enum constants '`Jonathon`', '`GoldenDel`', etc. are called *enumeration constants*. The enumeration constants are declared as '`public static final`' members of the enum. Their type is the type of the enumeration in which they are declared. These constants are called *self-typed*, in which "self" refers to the enclosing enumeration.

enumeration objects

You can create a variable of an enumeration type. You do not instantiate an `enum` using `new`. Rather, you declare an `enum` variable like you do for primitive types: '`Apple ap`'. Now, the variable `ap` can only hold values of type '`Apple`'.

```
Apple ap;
ap = Apple.RedDel;
```

The `enum` type (i.e., `Apple`) must be part of the expression.

Comparing for Equality; Switch

Two enumeration constants can be compared for equality using the `==` relational operator. Furthermore, an enumneration value can be used to control a `switch` statement. The `enum` prefix (type) is not required for `switch`.

```
switch(ap) {
  case Jonathon: ...
  case Winesap: ...
}
```

Printing Enum Types

When an enumeration object is printed, its name is output (without the `enum` type): '`System.out.println(ap)`' would produce '`RedDel`'.

10.2 Enum Methods `values()` and `valueOf()`

All enumerations inherit two methods:

`public static` *enum-type*`[]` [Method on `Enum`]
`values ()`

The `values()` method returns an array that contains a list of the enumeration constants.

`public static` *enum-type* [Method on `Enum`]
`valueOf` (*String* ***str***)

The `valueOf()` method returns the enumeration constant whose value corresponds to the string passed in *str*.

Examples using `values()` and `valueOf()` Methods

'`Apple allapples[] = Apple.values();`' is an example of using the `values()` method to populate an array with enumeration constants.

```
for(Apple a : Apple.values()) {
  System.out.println(a);
}
```

is an example of iterating directly on the `values()` method.

```
Apple ap;
ap = Apple.valueOf("Winesap");
System.out.println("ap contains " + ap);
```

is an example of using the `valueOf()` method to obtain the enumeration constant corresponding to the value of a string.

10.3 Java Enumerations are Class Types

A Java enumeration is a class type. That is, `enum` defines a class, which has much the same capabilities as other classes. An enumeration can be given constructors, instance variables, and methods. It can even implement interfaces. Each enumeration constant is an object of its enumeration type. When an enumeration is given a constructor, the constructor is called when each enumeration constant is created. Also, each enumeration constant has its own copy of any instance variables defined by the enumeration.

```
enum Apple {
  Jonathon(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);
  private int price;
  Apple(int p) { price = p; }
  int getPrice() { return price; }
}

class EnumDemo {
```

```
  public static void main (String[] args) {
    Apple ap;
  }
}
```

In this example, the enumeration '`Apple`' is given an instance variable `price`, a constructor, and an instance method '`getPrice()`'. When the variable '`ap`' is declared in '`main()`', the constructor for '`Apple`' is called once for each constant that is specified. The arguments to the constructor are placed in parentheses after the name of each constant. Thereafter, each enumeration constant has its own copy of '`price`', which can be obtained by calling the instance method '`getPrice()`'. In addition, there can be multiple overloaded constructors just as for any other class.

Restrictions on Enums

- An enumeration cannot inherit another class.
- An `enum` cannot be a superclass (`enum` cannot be extended).

The key is to remember that each enumeration constant is an object of the class in which it is defined.

10.4 Enumerations Inherit `Enum`

All enumerations automatically inherit from one superclass: `java.lang.Enum`. This class defines several methods that are available for use by all enumerations.

`ordinal()` and `compareTo()`

`final int` [Method on `Enum`]
`ordinal ()`

> The `ordinal()` method returns a value that indicates an enumeration constant's position in the list of constants, called its *ordinal value*. In other words, calling `ordinal()` returns the ordinal value of the invoking constant (zero indexed).

`final int` [Method on `Enum`]
`compareTo (`*enum-type* `e)`

> The ordinal values of two constants can be compared using the `compareTo()` method. Both the invoking constant and *e* must be of the same enumeration *enum-type*. This method returns a negative value, a zero, or a positive value depending on whether the invoking constant's ordinal value is less than, equal to, or greater than the passed-in enumeration constant's ordinal value.

`equals()` and `==`

`boolean` [Method on `Enum`]
`equals (`*enum-type* `e)`
`boolean` [Method on `Enum`]
`== (`*enum-type* `e)`

> Compare for equality an invoking enum constant with a referenced enum constant.

An invoking enum constant can compare for equality itself with any other object by using `equals()` or, equivalently, `==`, which overrides the `equals()` method defined in `Object`. `equals()` will return true only if both objects refer to the same constant within the same enumeration. (In other words, `equals` does not just compare ordinal values in general.)

The Java Standard Library

11 String Handling

12 Exploring `java.lang`

Classes and interfaces defined by `java.lang`, which is automatically imported into all programs. Contains classes and interfaces that are fundamental to all of Java programming. Beginning with JDK 9, all of `java.lang` is part of the `java.base` module.

`java.lang` includes the following classes

- Boolean
- Byte
- Character
  - Character.Subset
  - Character.UnicodeBlock
- Class
- ClassLoader
- ClassValue
- Compiler
- Double
- Enum
- Float
- InheritableThreadLocal
- Integer
- Long
- Math
- Module
  - ModuleLayer
  - ModuleLayer.Controller
- Number
- Object
- Package
- Process
  - ProcessBuilder
  - ProcessBuilder.Redirect
- Runtime
  - RuntimePermission
  - Runtime.Version
- SecurityManager
- Short
- StackFramePermission
- StackTraceElement
- StackWalker

- StrictMath
- String
  - StringBuffer
  - StringBuilder
- System
  - System.LoggerFinder
- Thread
  - ThreadGroup
  - ThreadLocal
- Throwable
- Void

`java.lang` includes the following interfaces

- Appendable
- AutoClosable
- CharSequence
- Clonable
- Comparable
- Iterable
- ProcessHandle
  - ProcessHandle.Info
- Readable
- Runnable
- StackWalker.StackFrame
- System.Logger
- Thread.UncaughtExceptionHandler

12.1 Primitive Type Wrappers

Java uses primitive types for '`int`', '`char`', etc. for performance reasons. These primitives are not part of the object hierarchy; they are passed by-value, not by reference. Sometimes you may need to create an object representation for a primitive type. To store a primitive in a class, you need to wrap the primitive type in a class.

Java provides classes that correspond to each of the primitive types. These classes encapsulate or *wrap* the primitive types within a class. They are commonly referred to as *type wrappers*.

12.1.1 Number

12.1.2 Double and Float

12.1.3 isInfinite() and isNaN()

12.1.4 Byte, Short, Integer, Long

12.1.5 Converting Numbers to and from String

12.2 The `Iterable` Interface

`Iterable` must be implemented by any class whose objects will be used by the for-each version of the `for` loop. In other words, for an object to be used within a for-each style `for` loop, its class must implement `Iterable`. `Iterable` is a generic interface that has this declaration:

```
interface Iterable<T>
```

Here, `T` is the type of the object being iterated. It defines one abstract method, `iterator()`, which is declared as:

```
Iterator<T> iterator()
```

It returns an iterator to the elements contained in the invoking object.

`Iterable` Default Methods

`forEach()`

Beginning with JDK 8, `Iterable` also defines two default methods. The first is called `forEach()`:

```
default void forEach(Consumer<? super T> action)
```

For each element being iterated, `forEach()` executes the code specified by *action*. (`Consumer` is a functional interface added by JDK 8 and defined in `java.util.function`.)

`spliterator()`

The second default method is `spliterator()`:

```
default Spliterator<T> spliterator()
```

It returns a `Spliterator` to the sequence being iterated.

13 java.util — Part 1: The Collections Framework

The package `java.util` contains a large assortment of classes and interfaces that support a braod range of functionality. For example, `java.util` has classes that

- generate pseudorandom numnbers,
- manage date and time,
- observe events,
- manipulate sets of bits,
- tokenize strings, and
- handle formatted data

Collections Framework

The `java.util` package also contains one of Java's most powerful subsystems: the Collections Framework. This is a sophisticated hierarchy of interfaces and classes that provide state-of-the-art technology for managing groups of objects. Beginning with JDK 9, `java.util` is part of the `java.base` module.

13.1 Collections Overview

The Java Collections Framework standardizes the way in whcih groups of objects are handled by programs. Collections were not part of the original Java release, but were added by J2SE 1.2. Prior to the Collections Framework, Java provided ad hoc classes such as `Dictionary`, `Vector`, `Stack`, and `Properties` to stored and manipulate groups of objects. These classes lacked a central unifying theme. The way that you used `Vector` was different from the way that you used `Properties`. This early, ad hoc approach was not designed to be easily extended or adapted. Collections were an answer to these (and other) problems.

Collections Goals

The Collections Framework was designed to meet several goals. *First*, the framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hash tables) are highly efficient. You seldom, if ever, need to code one of these "data engines" manually. *Second*, the framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability. *Third*, extending and/or adapting a collection had to be easy. Toward this end, the entire Collections Framework is built upon a set of standard interfaces. Several standard implementations (such as `LinkedList`, `HashSet`, and `TreeSet` of these interfaces are provided that you may use as-is. You may also implement your own collection. Various special-purpose implementations are created for your convenience, and some partial implementations are provided that make creating your own collection class easier. *Finally*, mechanisms were added that allow the integration of standard arrays into the Collections Framework.

1. high-performance
2. interoperatibility
3. standard interfaces and implementations; easily extensibible
4. integration of arrays

Algorithms

Algorithms are another important part of the collection mechanism. Algorithms operate on collections and are defined as static methods within the `Collections` class. Thus, they are available for all collections. Each collection class need not implement its own versions. The algorithms provide a standard means of manipulating collections.

`Iterator` Interface

Antoher item closely associated with the Collections Framework is the `Iterator` interface. An *iterator* offers a general-purpose standardized way of accessing the elements within a collection, one at a time. Thus, an iterator provides a means of *enumerating the contents of a collection*. Because each collection provides an iterator, the elements of any collection class can be accessed through the methods defined by `Iterator`. Thus, with only small changes, the code that cycles through a set can also be used to cycle through a list.

`Splitarator`

JDK 8 added another type of iterator called a *spliterator*. These are iterators that provide support for parallel iteration. The interfaces that support spliterators are `Spliterator` and several nested interfaces that support primitive types. JDK 8 also added iterator interfaces designed for use with primitive types such as `PrimitiveIterator` and `PrimitiveIterator.OfDouble`.

Map Interfaces and Classes

In addition to collections, the framework defines several map interfaces and classes. *Maps* store key/value pairs. Although maps are part of the Collections Framework, they are not "collections" in the strict use of the term. You can, however, obtain a *collection-view* of a map. Such a view contains the elements from the map stored in a collection. Thus, you can process the contents of a map as a collection.

13.2 The Collection Interfaces

The Collections Framework defines several core interfaces. Beginning with the collection interfaces is necessary because they determine the fundamental nature of the collection classes. The concrete classes simply provide different implementations of the standard interfaces. The interfaces that underpin collections are summarized:

`Collection` Enables you to work with groups of objects; it is at the top of the collections hierarchy.

`Deque` Extends `Queue` to handle a double-ended queue.

`List` Extends `Collection` to handle sequences (lists of objects).

`NavigableSet` Extends `SortedSet` to handle retrieval of elements based on closest-match searchesl.

`Queue` Extends `Collection` to handle special types of lists in which elements are removed only from the head.

`Set` Extends `Collection` to handle sets, which must contain unique elements.

`SortedSet` Extends `Set` to handle sorted sets

Table 13.1: Summary of `Collection` Interfaces

In addition to the collection interfaces, collections also use the following interfaces:

- `Comparator` (defines how two objects are compared)
- `RandomAccess` (indicates that a collection supports efficient random access to its elements)
- `Iterator` (the iterators enumerate the objects within a collection)
- `ListIterator`
- `Spliterator`

Optional Interface Methods — Modifiable Collections

To provide the greatest flexibility in their use, the collection interfaces allow some methods to be optional. The optional methods enable you to modify the contents of a collection. Collections that support these methods are called *modifiable*. Collections that do not allow their contents to be changed are called *unmodifiable*. If an attempt is made to use one of these methods on an unmodifiable collection, an `UnsupportedOperationException` is thrown. All of the built-in collections are modifiable.

13.2.1 The `Collection` Interface

The `Collection` interface is the foundation upon which the Collections Framework is build because it must be implemented by any class that defines a collection. `Collection` is a generic interface that has this declaration:

```
interface Collection<E>
```

Here, `E` specifies the type of objects that the collection will hold. `Collections` extends the `Iterable` interface. This means that all collections can be cycled through by use of their for-each style `for` loop. (Only classes that implement `Iterable` can be cycled through by the `for`. See Section 12.2 "The `Iterable` Interface", page 86.)

Collection Core Methods

`Collection` declares the core methods that all collecitons will have. These methods are summarized in Table 13.2. Because all collections implement `Collection`, familiarity with its methods is necessary for a clear understanding of the framework. Several of these methods can throw an `UnsupportedOperationException`. This occurs if a collection cannot be modified. A `ClassCastException` is generated when one object is incompatible with another, such as when an attempt is made to add an incompatible object to a collection. A `NullPointerException` is thrown if an attempt is made to store a `null` object and `null` elements are not allowed in the collection. An `IllegalArgumentException` is thrown if an invalid argument is used. An `IllegalStateException` is thrown if an attempt is made to add an element to a fixed-length collection that is full.

boolean add(E *obj*)	Adds *obj* to the invoking collection. Returns `true` if *obj* was added to the collection. Returns `false` if *obj* is already a member of the collection and the collection does not allow duplicates
boolean addAll(Collection<? extends E> *c*)	Adds all the elements of *c* to the invoking collection. Returns `true` if the collection changed (i.e., the elements were added). Otherwise, returns `false`.

FIXME: continue adding methods

Table 13.2: The Methods Declared by `Collection`

Adding and Removing Objects to and from Collections

Objects are added to a collection by calling `add()` Notice that `add()` takes an argument of type `E`, whcih means that objects added to a collection must be compatible with the type of data expected by the collection. You can add the entire contents of one collection to another by called `addAll()`.

You can remove an object by using `remove()`. To remove a group of objects, call `removeAll()`. You can remove all elements except those of a specified group by calling `retainAll()`. To remove an element only if it satisfies some condition,you cna use `removeIf()`. To empty a collection, call `clear()`.

Determine Whether a Collection Contains an Object

You can determine whether a collection contains a specific object by calling `contains()`. To determine whether one collection contains all the members of another, call `containsAll()`. You can determine when a collection is empty by calling `isEmpty()`. The number of elements currently held in a collection can be determined by calling `size()`.

`toArray()` Methods

```
Object[] toArray()
<T> T[] toArray(T array[])
```

The `toArray()` methods return an array that contains the elements stored in the collection. The first returns an array of `Object`. The second returns an array of elements that have the same type as the array specified as a parameter. Normally, the second form is more convenient because it returns the desired array type. Often, processing the contents

of a collection by using array-like syntax is advantageous. By providing a pathway between collections and arrays, you have the best of both.

Comparing Collections

Two collections can be compared for equality by called `equals()`. The precise meaning of "equality" may differ from collection to collection.

Collection Iterators

Another important method is `iterator()`, which returns an iterator to a collection. The `spliterator()` methods returns a spliterator to the collection. Iterators are frequently used when working with collections. Finally, the `stream()` and `parallelStream()` methods return a `Stream` that uses the collection as a source of elements.

13.2.2 The `List` Interface

13.2.3 The `Set` Interface

13.2.4 The `SortedSet` Interface

13.2.5 The `NavigableSet` Interface

13.2.6 The `Queue` Interface

13.2.7 The `Dequeue` Interface

13.3 The Collection Classes

Some collection classes provide full implementations that can be used as-is. Others are abstract, providing skeletal implementations that are used as starting points for creating concrete collections. As a general rule, the collection classes are not synchronized, but it is possible to obtain synchronized version.

Core Collection Classes

AbstractCollection	Implements most of the `Collections` interface
AbstractList	Extends `AbstractCollection` and implments most of the `List` interface
AbstractQueue	Extends `AbstractCollection` and implements parts of the `Queue` interface

FIXME: Complete entering Classes

Table 13.3: Collection Core Classes

13.4 Accessing a Collection via an Interator

Often you will want to cycle through the elements in a collection. One way to do this is to employ an *iterator*, which is an object that implements either the `Iterator` of the `ListIterator` interface. `Iterator` enables you to cycle through a collection, obtaining or

removing elements. `ListIterator` extends `Iterator` to allow bidirectional traversal of a list, and the modification of elements. `Iterator` and `ListIterator` are generic interfaces which are declared as shown:

```
interface Iterator<E>
interface ListIterator<E>
```

Here, `E` specifies the type of objects being iterated. The `Iterator` interface declares the methods shown in Table 13.4, while the `ListIterator` methods are shown in Table 13.5. In both cases, operations which modify the underlying collections are optional.

default void forEachRemaining(Consumer<? super E> *action*)	The action specified by *action* is executed on each unprocessed element in the collection. (Added by JDK 8.)
boolean hasNext()	Returns `true` if there are more elements. Otherwise, returns `false`.
E next()	Returns the next element. Throws `NoSuchElementException` if there is not a next element.
default void remove()	Removes the current element. Throws `IllegalStateException` if an attempt is made to call `remove()` that is not preceded by a call to `next()`. The default version throws an `UnsupporterdOperationException`.

Table 13.4: The Methods Provided by `Iterator`

void add(E *obj*)	Inserts *obj* into the list in front of the element that will be returned by the next call to `next()`.
default void forEachRemaining(Consumer<? super E> *action*)	The action specified by *action* is executed on each unprocessed element in the collection. (Added by JDK 8.)
boolean hasNext()	Returns `true` if there is a next element. Otherwise returns `false`.
boolean hasPrevious()	Returns `true` if there is a previous element. Otherwise returns `false`.
E next()	Returns the next element. A `NoSuchElementException` is thrown if there is not a next element.
int nextIndex()	Returns the index of the next element. If there is not a next element, returns the size of the list.
E previous()	Returns the previous element. A `NoSuchElementException` is thrown if there is not a previous element.
int previousIndex()	Returns the index of the previous element. If there is not a previous element, returns -1.
void remove()	Removes the current element from the list. An `IllegalStateException` is thrown if `remove()` is called before `next()` or `previous()` is invoked.
void set(E *obj*)	Assigns *obj* to the current element. This is the element last returned by a call to either `next()` or `previous()`.

Table 13.5: The Methods Declared by `ListIterator`

13.4.1 Using an Iterator

You must obtain an iterator to access a collection through iteration. Each of the collection classes provides an `iterator()` method that returns an iterator to the start of the collection. By using this iterator object, you can access each element in the collection, one element at a time.

Process To Cycle Through the Contents of a Collection

1. Obtain an iterator to the start of the collection by calling the collection's `iterator()` method;
2. Set up a loop that makes a call to `hasNext()`. Have the loop iterate as long as `hasNext()` returns `true`;
3. Within the loop, obtain each element by calling `next()`.

For collections that implement `List`, you can obtain an iterator by calling `listIterator()`. A list iterator gives you the ability to access a collection in either the forward or backward direction and lets you modify an element. Otherwise, `ListIterator` is used just like `Iterator`.

13.4.2 The For-Each Alternative to Iterators

If you won't be modifying the contents of a collection or obtaining elements in reverse order, then the for-each version of the `for` loop is often a more convenient alternative to cycling through a collection than is using an iterator. The `for` can cycle through any collection of objects that implement the `Iterable` interface. Because all of the collection classes implement this inteface, they can all be operated upon by `for`.

13.5 Spliterators

JDK 8 added a new type of iterator called a *spliterator* that is defined by the `Spliterator` interface. A spliterator cycles through a sequence of elements and in this regard it is similar to the iterators. However, the techniques required to use it differ. Furthermore, it offers substantially more functionality than does either `Iterator` or `ListIterator`. Perhaps the most important aspect of `Spliterator` is its ability to provide support for parallel iteration of portions of the sequence. Thus, `Spliterator` supports parallel programming.

However, you can use `Spliterator` even if you won't be using parallel execution. One reason you might want to do is because it offers a streamlined approach that combines the `hasNext()` and `next()` operations in one method.

`Spliterator` is a generic interface that is declared thus:

```
interface Spliterator<T>
```

`Spliterator` declares the methods shown in Table 13.6.

int characteristics()	Returns the characteristics of the invoking spliterator, encoded into an integer.
long estimateSize()	Estimates the number of elements left to iterate and returns the result. Returns `Long.MAX_VALUE` if the count cannot be obtained for any reason.
default void forEachRemaining(Consumer<? super T> *action*)	Applies *action* to each unprocessed element in the data source.
default Comparator<? super T> getComparator()	Returns the comparator used by the invoking spliterator or `null` if natural ordering is used. If the sequence is unordered, `IllegalStateException` is thrown.
default long getExactSizeIfKnown()	If the invoking spliterator is sized, returns the number of elements left to iterate. Returns -1 otherwise.
default boolean hasCharacteristics(int *val*)	Returns `true` if the invoking spliterator has the characteristics passed in *val*. Returns `false` otherwise.
boolean tryAdvance(Consumer<? super T> *action*)	Executes *action* on the next element in the iteation. Returns `true` if there is a next element. Returns `false` if no elements remain.
Spliterator<T> trySplit()	If possible, split the invoking spliterator, returning a reference to a new spliterator for the partition. Otherwise, returns `null`. Thus, if successful, the original spliterator iterates over one portion of the sequence and the returned spliterator iterates over the other portion.

Table 13.6: The Methods Declared by Spliterator

Characteristics

Each `Spliterator` has a set of attributes, called *characteristics*, associated with it. These are defined by static `int` fields in `Spliterator`, such as

- SORTED
- DISTINCT
- SIZED
- IMMUTABLE

to name a few. You can obtain the characteristics by calling `characteristics()`. You can determine if a characteristic is present by calling `hasCharacteristics()`.

Spliterator Subinterfaces

There are several nested subinterfaces of `Spliterator` designed for use with the primitive types `double`, `int`, and `long`. These are called `Spliterator.OfDouble`, `Spliterator.OfInt`, and `Spliterator.OfLong`. There is also a generalized version called `Spliterator.OfPrimitive`, which offers additional flexibility and servers as a superinterface.

13.6 Storing User-Defined Classes in Collections

13.7 RandomAccess Interface

13.8 Working with Maps

13.9 Comparators

13.10 The Collection Algorithms

13.11 Arrays Class

13.12 Legacy Classes and Interfaces

14 java.util — Part 2: Utility Classes

15 Input/Output — `java.io`

This chapter explores `java.io`, which provides support for I/O operations. Data is retrieved from an *input* source. The results of a program are sent to an *output* destination. In Java, these sources or destinations are defined very broadly. Although physically different, these devices are all handled by the same abstraction: the *stream*. An I/O stream is a logical entity that either produces or consumes information. An I/O stream is linked to a physical device by the Java I/O system. All I/O streams behave in the same manner, even if the phyiscal devices they are linked to differ.

Beginning with version 1.4, a second I/O system was added to Java, called NIO (which meant New I/O). NIO is packaged in `java.nio` and its subpackages. The NIO is described in Chapter 16 "NIO", page 124.

15.1 I/O Classes and Interfaces

15.1.1 I/O Classes Defined by `java.io`

- BufferedInputStream / BufferedOutputStream
- BufferedReader / Buffered Writer
- ByteArrayInputStream / ByteArrayOutputStream
- CharArrayReader / CharArrayWriter
- Console
- DataInputStream / DataOutputStream
- File
- FileDescriptor
- FileInputStream / FileOutputStream
- FilePermission
- FileReader / FileWriter
- FilterInputStream / FilterOutputStream
- FilterReader / FilterWriter
- InputStream / OutputStream
- InputStreamReader / OutputStreamWriter
- LineNumberReader
- ObjectInputFilter.Config
- ObjectInputStream / ObjectOutputStream
- ObjectInputStream.GetField / ObjectOutputStream.PutField
- ObjectStreamClass
- ObjectStreamField
- PipedInputStream / PipedOutputStream
- PipedReader / PipedWriter
- PrintStream / PrintWriter
- PushbackInputStream / PusbbackReader

- RandomAccessFile
- Reader / Writer
- SequenceInputStream
- SerializablePermission
- StreamTokenizer
- StringReader / StringWriter

15.1.2 I/O Interfaces Defined by `java.io`

- Closeable
- DataInput / DataOutput
- Externalize
- FileFilter
- FilenameFilter
- Flushable
- ObjectInput / ObjectOutput
- ObjectInputFilter
- ObjectInputFilter.FilterInfo
- ObjectInputValidation
- ObjectStreamConstants
- Serializable

15.2 File

The `File` class does not operate on streams. It deals directly with files and the file system. The `File` class does not specify how information is retrieved from or stored in files; rather, it describes the properties of a file itself. A `File` object is used to obtain or manipulate the information associated with a disk file, such as the permissions, time, date, and directory path, and to navigate subdirectory hierarchies.[1]

Files and Directories in Java

Files are a primary source and destination for data within programs. Files are a central resource for storing persistent and shared information. A directory in Java is treated simply as a `File` with one additional property — a list of filenames that can be examined by the `list()` method.

Constructors Used to Create `File` Objects

- File(String *directoryPath*)
- File(String *directoryPath*, String *filename*)
- File(File *dirObj*, String *filename*)
- File(URI *uriObj*)

[1] The `Path` interface and `Files` class, part of the NIO system, offer a powerful alternative to `File`. See Chapter 16 "NIO", page 124.

dirObj is a `File` object that specifies a directory, while *uriObj* is a `URI` object that describes a file.

Examples Creating Files and Directories

The following example creates three files. The first `File` object is constructed with a directory path as the only argument. The second includes two arguments — the path and the filename. The third includes the file path assigned to `f1` and a filename; `f3` refers to the same file as `f2`.

```
File f1 = new File("/");
File f2 = new File("/", "autoexec.bat");
File f3 = new File(f1, "autoexec.bat");
```

15.2.1 `File` Methods

`File` defines many methods that obtain the standard properties of a `File` object.

`getName()`
: returns the name of the file

`getParent()`
: returns the name of the parent directory

`getPath()`
`getAbsolutePath()`
: returns the path

`exists()` returns `true` if the file exists, `false` if it does not

`canWrite()`
`canRead()`
: returns whether the file is writeable/readable

`isDirectory()`
: returns whether the file is a directory

`isFile()` returns whether the file is a regular file (`true`) or a non-file (`false`) such as directory, device drivers, named pipes, etc.

`isAbsolute()`
: returns whether the file is an absolute path (`true`) or a relative path (`false`)

`lastModified()`
: returns the modification date and time

`length()` returns the file's size

Table 15.1: `File` Property Methods

15.2.2 `File` Utility Methods

`renameTo()`
: boolean renameTo(File *newName*); returns `true` upon success or `false` if the file cannot be renamed

`delete()`
: boolean delete(); deletes a file or directory (if the directory is empty); returns `true` if it successfully deletes or `false` if the file or directory cannot be removed;

`deleteOnExit()`
: removes the file associated with the invoking object when the Java Virtual Machines terminates

`getFreeSpace()`
: returns the number of free bytes of storage (as a `long`) available on the partition associated with the invoking object

`getTotalSpace()`
: returns the stoarage capacity (as a `long`) of the partition associated with the invoking object

`getUsableSpace()`
: returns the number of usable free bytes of storage (as a `long`) available on the partition associated with the invoking object

`isHidden()`
: returns `true` if the invoking file is hidden, or `false` otherwise

`setLastModifiedTime()`
: sets the time stamp on the invoking file to that specified by the arguement (`long` *millisec*), which is the number of milliseconds from January 1, 1970, UTC

`setReadOnly()`
: sets the invoking to read-only; returns `true` on success

`readable()`
`writeable()`
`executable()`
`compareTo()`
: because `File` implements the `Comparable` interface

`toPath()`
: returns a `Path` object that represents the file encapsulated by the invoking `File` object; (in other words, `toPath()` converts a `File` into a `Path`);[2]

`mkdir()`
`mkdirs()`
: the first creates a directory, returning `true` on success and `false` on failure; use the second to create a directory for which no path exists; it creates both a directory and all the parents of the directory;

Table 15.2: File Utility Methods

[2] `toPath()` forms a bridge between the older `File` class and the newer `Path` interface; See Chapter 16 "NIO", page 124.

15.2.3 Directories

A *directory* is a `File` that contains a list of other files and directories. When you create a `File` object that is a directory, the `isDirectory()` method will return true. In this case, you can call `list()` on that object to extract the list of other files and directories inside. It has two forms. Here is the more general form:

`String[]` [Method on `File`]
`list ()`

`list()` is used to extract the list of other files and directories inside the calling File object

GeneralForm 15.1: Obtaining a list of files in a directory

15.2.4 Using `list()` to Examine Directory Contents

This program illustrates how to use `list()` to examine the contents of a directory.

{`DirList.java`} ≡

```
<Import java.io.File>
class DirList {
  public static void main(String[] args) {
    <DirList Instance Variable Declarations>
    <DirList Examine Directory Contents>
  }
}
```

The following table lists called chunk definition points.

Chunk name	**First definition point**
<*DirList Examine Directory Contents*>	See "DirList Examine Directory Contents", page 103.
<*DirList Instance Variable Declarations*>	See "DirList Instance Variable Declarations", page 102.
<*Import java.io.File*>	See "Import `java.io.File`", page 102.

15.2.4.1 Import `java.io.File`

<*Import java.io.File*> ≡

```
import java.io.File;
```

This chunk is called by {`DirList.java`}; see its first definition at "Using `list()` to Examine Directory Contents", page 102.

15.2.4.2 DirList Instance Variable Declarations

<*DirList Instance Variable Declarations*> ≡

```
<DirList Obtain Directory From Command-Line Args>
String dirname = args[0];
File f1 = new File(dirname);
```

This chunk is called by {`DirList.java`}; see its first definition at "Using `list()` to Examine Directory Contents", page 102.

The called chunk <*DirList Obtain Directory From Command-Line Args*> is first defined at "DirList Obtain Directory From Command-Line Args", page 103.

15.2.4.3 DirList Examine Directory Contents

<*DirList Examine Directory Contents*> ≡

```
if (f1.isDirectory()) {
  System.out.println("Directory of " + dirname);

  String[] s = f1.list();

  <DirList Examine Directory Contents For-Loop>

} else {
  System.out.println(dirname + " is not a directory");
}
```

This chunk is called by {`DirList.java`}; see its first definition at "Using `list()` to Examine Directory Contents", page 102.

The called chunk <*DirList Examine Directory Contents For-Loop*> is first defined at "Examine Directory Contents For-Loop", page 103.

15.2.4.4 Examine Directory Contents For-Loop

<*DirList Examine Directory Contents For-Loop*> ≡

```
for (int i = 0; i < s.length; i++) {
  File f = new File(dirname + "/" + s[i]);

  if (f.isDirectory()) {
    System.out.println(s[i] + " is a directory");
  } else {
    System.out.println(s[i] + " is a file");
  }
}
```

This chunk is called by <*DirList Examine Directory Contents*>; see its first definition at "DirList Examine Directory Contents", page 103.

15.2.4.5 DirList Obtain Directory From Command-Line Args

<*DirList Obtain Directory From Command-Line Args*> ≡

```
if (args.length != 1) {
  System.out.println("Usage: java DirList <directory>");
  return;
}
```

This chunk is called by the following chunks:

Chunk name	First definition point
<*DirList Instance Variable Declarations*>	See "DirList Instance Variable Declarations", page 102.
{`DirListOnly`}	See "Example Program Using FilenameFilter Interface", page 105.

15.2.5 Using `FilenameFilter`

You can limit the number of files returned by the `list()` method to include only those files that match a certain filename pattern, or *filter*. To do this, use a second form of list:

`String[]` [Method on `File`]
`list` (*FilenameFilter* ***FFObj***)
: Returns an array of String filenames found in the directory named by the calling File object

GeneralForm 15.2: Obtaining a filtered list of files in a directory

Here, *FFObj* is an object of a class that implements the `FilenameFilter` interface. This interface defines a single method, `accept()`, whcih is called once for each file in a list. It's general form is given here:

`boolean` [Method on `FilenameFilter`]
`accept` (*File* ***directory***, *String* ***filename***)
: The `accept()` method returns `true` for files in the directory specified by the *directory* that should be included in the list (that is, those that match the *filename* argument) and returns `false` for those files that should be excluded.

GeneralForm 15.3: `accept()` Form to be used with `list()`

15.2.6 Example Program Using FilenameFilter Interface

The `OnlyExt` class implements `FilenameFilter` by defining an `accept()` method, which will be used by a variation of the preceding program listing the contents of a directory to filter the directory listing.

OnlyExt Class

{`OnlyExt.java`} ≡

```
<Import java.io>
public class OnlyExt implements FilenameFilter {
  <OnlyExt Instance Variable Declarations>
  <OnlyExt Constructor>
  <OnlyExt Accept Method Implementation>
}
```

The following table lists called chunk definition points.

Chunk name	First definition point
<*Import java.io*>	See "Import `java.io`", page 44.

<*OnlyExt Accept Method Implementation*>	See "OnlyEct Accept Method Implementation", page 106.
<*OnlyExt Constructor*>	See "OnlyExt Constructor", page 106.
<*OnlyExt Instance Variable Declarations*>	See "OnlyExt Instance Variable Declarations", page 106.

DirListOnly Class

This class now `list()`s the directory contents by including the `FilenameFilter` object from `OnlyExt`.

{`DirListOnly`} ≡

```
<Import java.io>
class DirListOnly {
  public static void main(String[] args) {
    <DirList Obtain Directory From Command-Line Args>
    <DirListOnly FilenameFilter Object>
    <DirListOnly FilenameFilter Object List>
    <DirListOnly Print List>
  }
}
```

The following table lists called chunk definition points.

Chunk name	**First definition point**
<*DirList Obtain Directory From Command-Line Args*>	See "DirList Obtain Directory From Command-Line Args", page 103.
<*DirListOnly FilenameFilter Object*>	See "DirListOnly FilenameFilter Object", page 105.
<*DirListOnly FilenameFilter Object List*>	See "DirListOnly FilenameFilter Object List", page 105.
<*DirListOnly Print List*>	See "DirListOnly Print List", page 106.
<*Import java.io*>	See "Import `java.io`", page 44.

15.2.6.1 DirListOnly FilenameFilter Object

<*DirListOnly FilenameFilter Object*> ≡

```
FilenameFilter only = new OnlyExt("html");
```

This chunk is called by {`DirListOnly`}; see its first definition at "Example Program Using FilenameFilter Interface", page 105.

15.2.6.2 DirListOnly FilenameFilter Object List

Recallthe corresopnding line in the `DirList` program: See Section 15.2.4.3 "DirList Examine Directory Contents", page 103.

<*DirListOnly FilenameFilter Object List*> ≡

```
String[] s = f1.list(only);
```

This chunk is called by {`DirListOnly`}; see its first definition at "Example Program Using FilenameFilter Interface", page 105.

15.2.6.3 DirListOnly Print List

<*DirListOnly Print List*> ≡

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

This chunk is called by {`DirListOnly`}; see its first definition at "Example Program Using FilenameFilter Interface", page 105.

15.2.6.4 OnlyExt Instance Variable Declarations

<*OnlyExt Instance Variable Declarations*> ≡

```
String ext;
```

This chunk is called by {`OnlyExt.java`}; see its first definition at "Example Program Using FilenameFilter Interface", page 104.

15.2.6.5 OnlyExt Constructor

<*OnlyExt Constructor*> ≡

```
public OnlyExt(String ext) {
  this.ext = "." + ext;
}
```

This chunk is called by {`OnlyExt.java`}; see its first definition at "Example Program Using FilenameFilter Interface", page 104.

15.2.6.6 OnlyEct Accept Method Implementation

<*OnlyExt Accept Method Implementation*> ≡

```
public boolean accept(File dir, String name) {
  return name.endsWith(ext);
}
```

This chunk is called by {`OnlyExt.java`}; see its first definition at "Example Program Using FilenameFilter Interface", page 104.

15.2.7 `listFiles()` Alternative

There is a variation to the `list()` method, called `listFiles()`, that might be useful.

`File[]` [Method on `File`]
`listFiles ()`
`File[]` [Method on `File`]
`listFiles (`*FilenameFilter* `FFObjb`)
`File[]` [Method on `File`]
`listFiles (`*FileFilter* `FObjb`)

These methods return the file list as an array of `File` objects instead of Strings. The first method returns all files, and the second returns those files that satisfy the specified `FilenameFilter`. The third version of `listFiles()` returns those files with path names that satisfy the specified `FileFilter`. `FileFilter` defines only a single method, `accept()`, whcih is called once for each file in a list. Its general form is shown below (see GeneralForm 15.5).

GeneralForm 15.4: `File.listFiles()` Form

FileFilter.accept()

`boolean` [Method on `FileFilter`]
`accept (`*File* `path`)

Called once for each file in a list; it returns `true` for files that should be included in the list (that is, those that match the *path* argument) and `false` for those that should be excluded.

GeneralForm 15.5: FileFilter.accept() Method

15.2.8 Creating Directories

- boolean mkdir()
- boolen mkdirs()

15.3 The `AutoCloseable`, `Closeable`, and `Flushable` Interfaces

15.4 I/O Exceptions

`IOException` and `FileNotFoundException`

Two exceptions play an important role in I/O handling. The first is `IOException`. If an I/O error occurs, an `IOException` is thrown. In many cases, if a file cannot be opened, a `FileNotFoundException` is thrown. `FileNotFoundException` is a subclass of `IOException`, so both can be caught with a single `catch` that catches `IOException`. You might find it useful to `catch` each exception separately.

`SecurityException`

`SecurityException`

Another exception class that could occur during I/O is `SecurityException`. In situations in which a security manager is present, several of the file classes will throw a

`SecurityException` if a security violation occurs when attempting to open a file. By default, application run via `java` do not use a security manager. Other applications could generate a `SecurityExcpetion`, which will need to be handled.

15.5 Two Ways to Close a Stream

In general, a stream must be closed when it is no longer needed. Failure to do so can lead to memory leaks and resource starvation.

Traditional Method with `close()` in `finally`

Beginning with JDK 7, there are two basic ways in which you can close a stream. The first is to explicitly call `close()` on the stream. This is the traditional approach that has been used since the original release of Java. With this approach, `close()` is typically called within a `finally` block.

```
try {
  open and access file ...
} catch (IOException) {
   handle IOException ...
} finally {
   close the file ...
}
```

GeneralForm 15.6: Simplified skeleton for traditional approach to close a stream

`try-with-resources` Statement

The second approach to closing a stream is to automate the process by using the `try-with-resources` statement added by JDK 7. The `try-with-resources` is an enhanced form of `try` with the following form:

```
try (resource-specification) {
  use the resource ...
}
```

GeneralForm 15.7: Closing a file using `try-with-resources` Statement

Typically, *resource-specification* is a statement or statements that declare and initialize a resource, such as a file or other stream-related resource. It consists of a variable declaration in which the variable is initialized with a reference to the object being managed. When the `try` block ends, the resource is automatically released. In the case of a file, the file is automatically closed (there is no need to call `close()`.

`try-with-resources` Under JDK 9

Beginning with JDK 9, it is also possible for the resource specification of the `try` to consist of a variable that has been declared and initialized earlier in the program. However, that variable must be effectively `final`, which means that it has not been assigned a new value after being given its initial value.

Here are three key points about `try-with-resources` statement:

- Resources must be objects of classes the implement `AutoCloseable` interface

- A resource declared in `try` is *implicitly* `final`, while a resource declared outside the `try` must be *effectively* `final`.
- More than one resource can be handled by separating each declaration with a semicolon.

Principal Advantages to using `try-with-resources`

A principal advantage of `try-with-resources` is that the resource is closed automatically when the `try` block ends. Thus, it is not possible to forget to close a stream. Another advantage is that the `try-with-resources` approach typically results in shorter, clear, easier-to-maintain source code.

15.6 The Stream Classes

Java's stream-based I/O is built upon four abstract classes:

- Byte Streams
  - `InputStream`
  - `OutputStream`
- Character Streams
  - `Reader`
  - `Writer`

The top-level classes define the basic functionality common to all stream classes. In general, you should use the character stream classes when working with characters or strings and use the byte stream classes when working with bytes or other binary objects.

15.7 The Byte Streams

The byte stream classes provide an environment for handling byte-oriented I/O. A byte stream can be used with any type of object, including binary data. The byte streams are topped by `InputStream` and `OutputStream`.

15.7.1 `InputStream`

`InputStream` is an abstract class that defines Java's model of streaming byte input. It implements the `AutoCloseable` and `Closeable` interfaces. Most of the methods in this class will throw an `IOException` when an I/O error occurs.[3]

[3] The exceptions are `mark()` and `markSupported()`.

15.7.1.1 `InputStream` Methods

int available()
Returns the number of bytes of input currently available for reading

void close()
Closes the input source. Further read attempts will generate an `IOException`.

void mark(int *numBytes***)**
Places a mark at the current point in the input stream that will remain valid until *numBytes* bytes are read.

boolean markSupported()
Returns `true` if `mark()` / `reset()` are supported by the invoking stream.

int read() Returns an integer representation of the next available byte of input. `-1` is returned when an attempt is made to read at the end of the stream.

int read(byte *buffere[]***)**
Attempts to read up to *buffer.length* bytes into *buffer* and returns the actual number of bytes that were successfully read. `-1` is returned when an attempt is made to read at the end of the stream.

int read(byte *buffer[]***, int** *offset***, int** *numBytes***)**
Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer[offset]*, returning the numnber of bytes successfully read. `-1` is returned when an attempt is made to read at the end of the stream.

byte[] readAllBytes()
Beginning at the current position, reads to the end of the stream, returning a byte array that holds the input. (Added by JDK 9.)

int readNBytes(byte *buffer[]***, int** *offset***, int** *numBytes***)**
Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer[offset]*, returning the number of types successfully read. (Added by JDK 9.)

void reset()
Resets the input pointer to the previously set mark.

long skip(long *numBytes***)**
Ignores (that is, skips) *numBytes* bytes of input, returning the number of bytes actually ignored.

long transferTo(OutputStream *strm***)**
Copies the bytes in teh invoking stream into *strm*, returning the number of bytes copies. (Added by JDK 9.)

Table 15.3: The Methods Defined by `InputStream`

15.7.2 `OutputStream`

`OutputStream` is an abstract class that defines streaming byte output. It implements `AutoCloseable`, `Closeable`, and `Flushable` interfaces. Most of the methods defined by this class return `void` and throw an `IOException` in the case of I/O errors.

15.7.2.1 `OutputStream` Methods

void close()
: Closes the output stream. Further write attempts will generate an `IOException`.

void flush()
: Finalizes the output state so that any buffers are cleared (it flushes the output buffers).

void write(int *b***)**
: Writes a single byte to an output stream. Note that the parameter is an `int`, which allows you to call `write()` with an expression without having to cast it back to `byte`.

void write(byte *buffer[]***)**
: Writes a complete array of bytes to an output stream.

void write(byte *buffer[]***, int** *offset***, int** *numBytes***)**
: Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer[offset]*.

Table 15.4: The Methods Defined by `OutputStream`

15.7.3 `FileInputStream`

The `FileInputStream` class creates an `InputStream` that you can use to read bytes from a file. Two commonly used constructors are:

FileInputStream(String *filePath*)
FileInputStream(File *fileObj*)

Either can throw a `FileNotFoundException`.

15.7.4 `FileOutputStream`

15.7.5 `ByteArrayInputStream`

15.7.6 `ByteArrayOutputStream`

15.7.7 Filtered Byte Streams

Filtered streams are simply wrappers around underlying input or output streams that transparently provide some extended level of functionality. These streams are typcially accessed by methods that are expecting a generic stream, which is a superclass of the filtered streams. Typical extensions are buffering, character translation, and raw data translation. The filtered byte streams are:

- `FilterInputStream`
- `FilterOutputStream`

Their constructors are:

FilterOutputStream (OutputStream *os*)
FilterInputStream (InputStream *is*)

15.7.8 Buffered Byte Streams

For the byte-oriented streams, a *buffered stream* extends a filtered stream class by attaching a memory buffer to the I/O stream. This buffer allows Java to do I/O operations on more than a byte at a time, thereby improving performance. Because the buffer is available, skipping, marking, and resetting of the stream become possible. The buffered byte stream classes are:

- `BufferedInputStream`
- `BufferedOutputStream`
- `PushbackInputStream`

15.7.8.1 `BufferedInputStream`

Java's `BufferedInputStream` class allows you to "wrap" any `InputStream` into a buffered stream to improve performance. `BufferedInputStream` has two constructors:

BufferedInputStream (InputStream *inputStream*)
BufferedInputStream (InputStream *inputStream*, int *bufSize*)

The first form creates a buffered stream using a default buffer size. In the second, the size of the buffer is passed in *bufSize*. Use of sizes that are multiples of a memory page, a disk block, and son, can have a significant positive impact on performance. A good guess for a size is around 8,192 bytes, and attaching even a rather small buffer to an I/O stream is always a good idea. That way, the low-level system can read blocks of data from the disk or network and store the results in your buffer. Thus, even if you are reading the data a byte at a time out of the `InputStream`, you will be manipulating fast memory most of the time.

Buffering an input stream provides the foundation required to support moving backward in the stream of the available buffer. Beyond the `read()` and `skip()` methods implemented in any `InputStream`, `BufferedInputStream` also supports the `mark()` and `reset()` methods. This support is reflected by the `BufferedInputStream.markSupported()` returning `true`.

15.7.8.2 Buffered Input Example

The following example contrives a situation where we can use `mark()` to remember where we are in an input stream and later use `reset()` to get back there. This example is parsing a stream for the HTML entity reference for the "copyright" symbol. Such a reference begins with an ampersand (`&`) and ends with a semicolon (`;`) without any intervening whitespace. The sample input has two ampersands to show the case where the `reset()` happens and where it does not.

{`BufferedInputStreamDemo.java`} ≡

```
<Import java.io>
class BufferedInputStreamDemo {
  public static void main(String[] args) {

    <BufferedInputStreamDemo Instance Variables>
```

```
        <BufferedInputStreamDemo TryWithResources BufferedInputStream>
        <Catch IOException>

    }
}
```

The following table lists called chunk definition points.

Chunk name	First definition point
<*BufferedInputStreamDemo Instance Variables*>	See "BufferedInputStreamDemo Instance Variables", page 113.
<*BufferedInputStreamDemo TryWithResources BufferedInputStream*>	See "BufferedInputStreamDemo TryWithResources BufferedInputStream", page 113.
<*Catch IOException*>	See "Catch IOException", page 119.
<*Import java.io*>	See "Import `java.io`", page 44.

15.7.8.3 BufferedInputStreamDemo Instance Variables

<*BufferedInputStreamDemo Instance Variables*> ≡

```
<BufferedInputStreamDemo String>
<BufferedInputStreamDemo Buffer>
<BufferedInputStreamDemo ByteArrayInputStream>
<BufferedInputStreamDemo Utility Variables>
```

This chunk is called by {`BufferedInputStreamDemo.java`}; see its first definition at "Buffered Input Example", page 112.

The following table lists called chunk definition points.

Chunk name	First definition point
<*BufferedInputStreamDemo Buffer*>	See "BufferedInputStreamDemo Buffer", page 115.
<*BufferedInputStreamDemo ByteArrayInputStream*>	See "BufferedInputStreamDemo ByteArrayInputStream", page 115.
<*BufferedInputStreamDemo String*>	See "BufferedInputStreamDemo String Into Buffer", page 115.
<*BufferedInputStreamDemo Utility Variables*>	See "BufferedInputStreamDemo Utility Variables", page 115.

15.7.8.4 BufferedInputStreamDemo TryWithResources BufferedInputStream

<*BufferedInputStreamDemo TryWithResources BufferedInputStream*> ≡

```
try (BufferedInputStream f = new BufferedInputStream(in)) {

  <BufferedInputStreamDemo While Loop>

}
```

This chunk is called by {`BufferedInputStreamDemo.java`}; see its first definition at "Buffered Input Example", page 112.

The called chunk <*BufferedInputStreamDemo While Loop*> is first defined at "BufferedInputStreamDemo While Loop", page 114.

15.7.8.5 BufferedInputStreamDemo While Loop

<*BufferedInputStreamDemo While Loop*> ≡

```
while ( (c = f.read()) != -1 ) {

    <BufferedInputStreamDemo Switch on Character>

}
```

This chunk is called by <*BufferedInputStreamDemo TryWithResources BufferedInputStream*>; see its first definition at "BufferedInputStreamDemo TryWithResources BufferedInputStream", page 113.

The called chunk <*BufferedInputStreamDemo Switch on Character*> is first defined at "BufferedInputStreamDemo Switch on Character", page 114.

15.7.8.6 BufferedInputStreamDemo Switch on Character

<*BufferedInputStreamDemo Switch on Character*> ≡

```
switch (c) {
  case '&':
    if (!marked) {
      f.mark(32);
      marked = true;
    } else marked = false;
    break;

  case ';':
    if (marked) {
      marked = false;
      System.out.print("(c)");
    } else System.out.print( (char) c);
    break;

  case ' ':
    if (marked) {
      marked = false;
      f.reset();
      System.out.print("&");
    } else System.out.print( (char) c);
    break;

  default:
    if (!marked) System.out.print( (char) c);
    break;
}
```

This chunk is called by the following chunks:

Chunk name	First definition point
<*BufferedInputStreamDemo While Loop*>	See "BufferedInputStreamDemo While Loop", page 114.
<*BufferedReaderDemo TryWithResources BufferedReader*>	See "BufferedReaderDemo TryWithResources BufferedReader", page 122.

15.7.8.7 BufferedInputStreamDemo String Into Buffer

<*BufferedInputStreamDemo String*> ≡

```
String s = "This is a &copy; copyright symbol " + "but this is &copy not.\n"
```

This chunk is called by the following chunks:

Chunk name	First definition point
<*BufferedInputStreamDemo Instance Variables*>	See "BufferedInputStreamDemo Instance Variables", page 113.
<*BufferedReaderDemo Instance Variables*>	See "BufferedReaderDemo Instance Variables", page 121.

15.7.8.8 BufferedInputStreamDemo Buffer

<*BufferedInputStreamDemo Buffer*> ≡

```
byte buf[] = s.getBytes();
```

This chunk is called by <*BufferedInputStreamDemo Instance Variables*>; see its first definition at "BufferedInputStreamDemo Instance Variables", page 113.

15.7.8.9 BufferedInputStreamDemo ByteArrayInputStream

<*BufferedInputStreamDemo ByteArrayInputStream*> ≡

```
ByteArrayInputStream in = new ByteArrayInputStream(buf);
```

This chunk is called by <*BufferedInputStreamDemo Instance Variables*>; see its first definition at "BufferedInputStreamDemo Instance Variables", page 113.

15.7.8.10 BufferedInputStreamDemo Utility Variables

<*BufferedInputStreamDemo Utility Variables*> ≡

```
int c;
boolean marked = false;
```

This chunk is called by <*BufferedInputStreamDemo Instance Variables*>; see its first definition at "BufferedInputStreamDemo Instance Variables", page 113.

15.7.8.11 `BufferedOutputStream`

15.7.8.12 `PushbackInputStream`

15.7.9 `SequenceInputStream`

15.7.10 `PrintStream`

15.7.11 `DataOutputStream` and `DataInputStream`

15.7.12 `RandomAccessFile`

15.8 The Character Streams

While th byte stream classes provide sufficient functionality to handle any type of I/O operation, they cannot work directly with Unicode characters. Since one of the purposes of Java is to support the "write once, run anywhere" philosophy, it was necessary to include direction I/O support for characters. In this section, several of the character I/O classes are discussed.

At the top of the character stream hierarchies are the

- `Reader`
- `Writer`

abstract classes.

15.8.1 `Reader`

`Reader` is an abstact class that defines Java's model of streaming character input. It implements the `AutoCloseable`, `Closeable`, and `Readable` interfaces. All of the methods in this class (except `markSupported()`) will throw an `IOException` on error conditions.

Reader Methods

abstract void close()
: Closes the input source. Further read attempts will general an `IOException`.

void mark(int *numChars***)**
: Places a mark at the current point in the input stream that will remain valid until *numChars* characters are read.

boolean markSupported()
: Returns `true` if `mark()`/`reset()` are supported on this stream.

int read(char *buffer[]***)**
: Attempts to read up to *buffer.length* characters into *buffer* and returns the actual number of characters that were successfully read. `-1` is returned when an attempt is made to read at the end of the stream.

int read(CharBuffer *buffer***)**
: Attempts to read characters into *buffer* and returns the actual number of characters that were successfully read. `-1` is returned when an attempt is made to read at the end of the stream.

abstract int read(char *buffer[]***, int** *offset***, int** *numChars***)**
: Attempts to read up to *numChars* characters into *buffer* starting at *buffer[offset]*, returning the number of characters successfully read. `-1` is returned whn an attempt is made to read at the end of the stream.

boolean read()
: Returns `true` if the next input request will not wait. Otherwise, it returns `false`.

void reset()
: Resets the input pointer to the previously set mark.

long skip(long *numChars***)**
: Skips over *numChars* characters of input, returning the number of characters actually skipped.

Table 15.5: The Methods Defined by `Reader`

15.8.2 `Writer`

`Writer` is an abstract class that defines streaming character output. It implements the `AutoCloseable`, `Closeable`, `Flushable`, and `Appendable` interfaces. All of the methods in this class throw an `IOException` in the case of errors.

`Writer` Methods

Writer append(char *ch***)**
Appends *ch* tot he end of the invoking output stream. Returns a reference to the invoking stream.

Writer append(CharSequence *chars***)**
Appends *chars* to the end of the invoking output stream. Returns a reference to the invoking stream.

Writer append(CharSequence *chars***, int** *begin***, int** *end***)**
Appends the subrange of *chars* specified by *begin* and *end* - 1 to the end of the invoking output stream. Returns a refernce to the invoking stream.

abstract void close()
Closes the output stream. Further write attempts will generate an `IOException`.

abstract void flush()
Finalizes the output state so that any buffers are cleared (it flushes the output buffers).

void write(int *ch***)**
Writes a single character to the invoking output stream. The parameter is an `int`, whcih allows you to call `write()` with an expression without have to case it back to `char`. However, only the low-order 16 bits are written.

void write(char *buffer[]***)**
Writes a complete array of characters to the invoking output stream.

abstract void(char *buffer[]***, int** *offset***, int** *numChars***)**
Writes a subrange of *numChars* characters from the array *buffer*, beginning at *buffer[offset]* to the invoking output stream.

void write(String *str***)**
Writes *str* to the invoking output stream.

void write(String *str***, int** *offset***, int** *numChars***)**
Writes a subrange of *numChars* characters from the string *str*, beginning at the specified *offset*

Table 15.6: The Methods Defined by Writer

15.8.3 `FileReader`

The `FileReader` class creates a `Reader` that you can use to read the contents of a file. Two commonly used constructors are:

FileReader (String *filePath*)
FileReader (File *fileObj*)

Either can throw a `FileNotFoundException`.

Example Reading Lines From a File

This example shows how to read lines from a file and display them on the standard output device. It reads its own source file, which must be in the current directory.

{`FileReaderDemo.java`} ≡

```
<Import java.io>
class FileReaderDemo {
  public static void main (String[] args) {
    <FileReaderDemo TryWithResources FileReader>
    <Catch IOException>
  }
}
```

The following table lists called chunk definition points.

Chunk name	First definition point
<*Catch IOException*>	See "Catch IOException", page 119.
<*FileReaderDemo TryWithResources FileReader*>	See "FileReaderDemo TryWtihResources FileReader", page 119.
<*Import java.io*>	See "Import `java.io`", page 44.

15.8.3.1 FileReaderDemo TryWtihResources FileReader

<*FileReaderDemo TryWithResources FileReader*> ≡

```
try (FileReader fr = new FileReader("FileReaderDemo.java")) {
  int c;

  while ((c = fr.read()) != -1)
    System.out.print((char) c);
}
```

This chunk is called by {`FileReaderDemo.java`}; see its first definition at "`FileReader`", page 119.

15.8.3.2 Catch IOException

<*Catch IOException*> ≡

```
catch (IOException e) {
  System.out.println("I/O Error: " + e);
}
```

This chunk is called by the following chunks:

Chunk name	First definition point
{`BufferedInputStreamDemo.java`}	See "Buffered Input Example", page 112.
{`BufferedReaderDemo.java`}	See "Buffered Reader Demo", page 120.
{`FileReaderDemo.java`}	See "`FileReader`", page 119.

15.8.4 `FileWriter`

`FileWriter` creates a `Writer` that you can use to write to a file. Four commonly used constructors are:

FileWriter (String *filePath*)

FileWriter (String *filePath*, boolean *append*)
FileWriter (File *fileObj*)
FileWriter (File *fileObj*, boolean *append*)

They can all throw an `IOException`. If *append* is `true`, then output is appended to the end of the file. Creation of a `FileWriter` is not dependent on the file already existing. `FileWriter` will create the file before opening it for output when you create the object. In the case where you attept to open a read-only file, an `IOException` will be thrown.

15.8.5 `CharArrayReader`

15.8.6 `CharArrayWriter`

15.8.7 `BufferedReader`

`BufferedReader` improves performance by buffering input. It has two constructors:

BufferedReader (Reader *inputStream*)
BufferedReader (Reader *inputStream*, int *bufSize*)

The first form creates a buffered character stream using a default buffer size. In the second, the size of the buffer is passed in *bufSize*. Closing a `BufferedReader` also causes the underlying stream specified by *inputStream* to be closed.

Buffering an input character stream provides the foundation required to support moving backward in the stream within the available buffer. To support this, `BufferedReader` implements the `mark()` and `reset()` methods, and `BufferedReader.markSupported()` returns `true`.

JDK 8 added a new method to `BufferedReader` called `lines()`. It returns a `Stream` reference to the sequence of lines read by the reader. (`Stream` is part of the stream API discussed in Chapter 23 "The Stream API", page 131.

15.8.8 Buffered Reader Demo

This example reworks the `BufferedInputStream` example (see Section 15.7.8.2 "Buffered Input Example", page 112) so that it uses a `BufferedReader` character stream rather than a buffered byte stream. It uses the `mark()` and `reset()` methods to parse a stream for the HTML-entity reference for the copyright symbol. Such a reference begins with an ampersand (`&`) and ends with a semicolon (`;`) without any intervening whitespace. The sample has two ampersands to show the case where the `reset()` happens and where it does not.

{`BufferedReaderDemo.java`} ≡

```
<Import java.io>
class BufferedReaderDemo {
  public static void main (String[] args) throws IOException {

    <BufferedReaderDemo Instance Variables>

    <BufferedReaderDemo TryWithResources BufferedReader>
```

```
        <Catch IOException>

    }
}
```

The following table lists called chunk definition points.

Chunk name	First definition point
<*BufferedReaderDemo Instance Variables*>	See "BufferedReaderDemo Instance Variables", page 121.
<*BufferedReaderDemo TryWithResources BufferedReader*>	See "BufferedReaderDemo TryWithResources BufferedReader", page 122.
<*Catch IOException*>	See "Catch IOException", page 119.
<*Import java.io*>	See "Import `java.io`", page 44.

15.8.8.1 BufferedReaderDemo Instance Variables

The String is copied directly from the prior implementation.

<*BufferedReaderDemo Instance Variables*> ≡

```
    <BufferedInputStreamDemo String>
    <BufferedReaderDemo Buffer>
```

This chunk is called by {`BufferedReaderDemo.java`}; see its first definition at "Buffered Reader Demo", page 120.

The following table lists called chunk definition points.

Chunk name	First definition point
<*BufferedInputStreamDemo String*>	See "BufferedInputStreamDemo String Into Buffer", page 115.
<*BufferedReaderDemo Buffer*>	See "BufferedReaderDemo Buffer", page 121.

15.8.8.2 BufferedReaderDemo Buffer

Notice that this implementation uses a `char` buffer, with a `getChars()` method to transfer the string characters, while the `BufferedInputStream` implementation uses a `byte` buffer, and a `getBytes()` method to transfer the string bytes. See Section 15.7.8.8 "BufferedInputStreamDemo Buffer", page 115.

<*BufferedReaderDemo Buffer*> ≡

```
    char buf[] = new char[s.length()];
    s.getChars(0, s.length(), buf, 0);
```

This chunk is called by <*BufferedReaderDemo Instance Variables*>; see its first definition at "BufferedReaderDemo Instance Variables", page 121.

15.8.8.3 BufferedReaderDemo TryWithResources BufferedReader

Notice that the Switch code is identical to the byte stream example.

<*BufferedReaderDemo TryWithResources BufferedReader*> ≡

```
try ( BufferedReader f = new BufferedReader(in) ) {
  while ( (c = f.read()) != -1 ) {
    <BufferedInputStreamDemo Switch on Character>
  }
}
```

This chunk is called by {`BufferedReaderDemo.java`}; see its first definition at "Buffered Reader Demo", page 120.

The called chunk <*BufferedInputStreamDemo Switch on Character*> is first defined at "BufferedInputStreamDemo Switch on Character", page 114.

15.8.9 `BufferedWriter`

A `BufferedWriter` is a `Writer` that buffers output. Using a `BufferedWriter` can improve performance by reducing the number of times data is actually phyiscally written to the output device. A `BufferedWriter` has these constructors:

BufferedWriter (Writer *outputStream*)
BufferedWriter (Writer *outputStream*, int *bufSize*)

15.8.10 `PushbackReader`

15.8.11 `PrintWriter`

15.9 The Console Class

15.10 Serialization

Serialization is the process of writing the state of an object to a byte stream. This is useful when you want to save the state of your program to a persistent storage area, such as a file. At a later time, you may restore these objects by using the process of deserialization.

Assume that an object to be serialized has references to other objects, which, in turn, have references to still more objects. This set of objects and the relationships among them form a directed graph. There may also be circular references within this object graph. Objects may also contain references to themselves. The object serialization and deserialization facilities have been designed to work correctly in these scenarios. If you attempt to serialize an object at the top of an object graph, all of the other referenced objects are recurseively located and serialized. Similarly, during the process of deserialization, all of these objects and their references are correctly restored.

15.10.1 `Serializable`

Only an object that implements the `Serializable` interface can be saved and restored by the serialization facilities. The `Serializable` interface defines no members. It is simply used to indicate that a class may be serialized. If a class is serializable, all of its subclasses are also serializable.

Variables that are declared as `transient` are not saved by the serialization facilities. Also `static` variables are not saved.

15.10.2 `Externalizable`

The Java facilities for serialization and deserialization have been designed so that much of the work to save and restore the state of an object occurs automatically. However, there are cases in which the programmer may need to have control over these processes. The `Externalizable` interface is designed for these sitautions.

The `Externalizable` interface defines these two methods:

void readExternal (ObjectInput *inStream*) throws IOException, ClassNotFoundException
void writeExternal (ObjectOutput *outStream* throws IOException

15.10.3 `ObjectOutput`

15.10.4 `ObjectOutputStream`

15.10.5 `ObjectInput`

15.10.6 `ObjectInputStream`

15.10.7 A Serializable Example

15.11 Stream Benefits

16 NIO

17 Networking

18 Event Handling

19 AWT: Working with Windows, Graphics, and Text

20 Using AWT Controls, Layout Managers, and Menus

21 Images

22 The Concurrency Utilities

23 The Stream API

24 Regular Expressions

25 Reflection

Reflection is the ability of software to analyze or modify itself at runtime rather than at compile time. This is provided by the `java.lang.reflect` package and elements in `Class`. Beginning with JDK 9, `java.lang.reflect` is part of the `java.base` module. Reflection allows you to analyze a software component and describe its capabilities dynamcically at run-time rather than at compile time. For example, by using reflection, you can determine what methods, constructors, and fields a class supports.

Drawbacks of Reflection

Because reflection is dynamic, the java virtual machine is unable to make certain optimizations and could therefor run slower. Also, since reflection allows code to perform operations that would be illegal in non-reflective code, such as accessing private fields and methods, the use of reflection can result in unexpected behavior. Reflection breaks abstractions.

25.1 `java.lang.reflect` Package

This package provides classes and interfaces for obtaining reflective information about classes and objects. Reflection allows programmatic access to information about the fields, methods and constructors of loaded classes, and the use of reflected fields, methods, and constructors to operate on their underlying counterparts, within encapsulation and security restrictions.

The package `java.lang.reflect` includes several interfaces. Of special interest is `Member`, which defines methods that allow you to get information abouta field, constructor, or method of a class. There are also ten classes in this package.

Classes in this package, along with java.lang.Class accommodate applications such as debuggers, interpreters, object inspectors, class browsers, and services such as Object Serialization and JavaBeans that need access to either the public members of a target object (based on its runtime class) or the members declared by a given class.

Note that none of the classes in `java.lang.reflect` offer any public constructors. All reflection interactions must go through `java.lang.Class`. Note also that `java.lang.Class` does not offer any public constructors. Instances of this class are created by the Java Virtual Machine.

25.1.1 Classes Defined in `java.lang.reflect`

AccessibleObject	Allows you to bypass the default access control checks
Array	Allows you to dynamically create and manipulate arrays
Constructor	Provides information about a constructor
Executable	An abstract superclass extended by `Member` and `Constructor`
Field	Provides information about a field
Method	Provides information about a method
Modifier	Provides information about class and member access modifiers
Parameter	Provides information about parameters
Proxy	Supports dynamic proxy classes
ReflectPermission	Allows reflection of private or protected members of a class

Table 25.1: Classes Defined in `java.lang.reflect`

25.1.2 Interfaces Defined in `java.lang.reflect`

AnnotatedArrayType
: represents the potentially annotated use of an array type, whose component type may itself represent the annotated use of a type.

AnnotatedElement
: represents an annotated element of the program currently running in this VM.

AnnotatedParameterizedType
: represents the potentially annotated use of a parameterized type, whose type arguments may themselves represent annotated uses of types.

AnnotatedType
: represents the potentially annotated use of a type in the program currently running in this VM.

AnnotatedTypeVariable
: represents the potentially annotated use of a type variable, whose declaration may have bounds which themselves represent annotated uses of types.

AnnotatedWildcardType
: represents the potentially annotated use of a wildcard type argument, whose upper or lower bounds may themselves represent annotated uses of types.

GenericArrayType
: represents an array type whose component type is either a parameterized type or a type variable.

GenericDeclaration
: A common interface for all entities that declare type variables.

InvocationHandler
: the interface implemented by the invocation handler of a proxy instance.

Member an interface that reflects identifying information about a single member (a field or a method) or a constructor.

ParameterizedType
: ParameterizedType represents a parameterized type such as Collection<String>.

Type Type is the common superinterface for all types in the Java programming language.

TypeVariable<D extends GenericDeclaration>
: the common superinterface for type variables of kinds.

WildcardType
: WildcardType represents a wildcard type expression, such as ?, ? extends Number, or ? super Integer.

Table 25.2: Interfaces Defined in `java.lang.reflect`

25.1.3 Reflection Demonstration

The following application illustrates a simple use of the Java reflection capabilities. It prints the constructors, fields, and methods of the class `java.awt.Dimension`. The program begins by using the `forName()` mthod of `Class` to get a class object for `java.awt.Dimension`. Once this is obtained, `getConstructors()`, `getFields()`, and `getMethods()` are used to analyze this class object. They return arrays of `Constructor`, `Field`, and `Method` objects that provide the information about the object. The `Constructor`, `Field`, and `Method` classes define several methods that can be used to obtain information about an object. Each supports the `toString()` method. Therefore, using these objects as arguments tot he `println()` method is straightforward.

ReflectionDemo1.java

{`ReflectionDemo1.java`} ≡

```
<Import java.lang.reflect>

public class ReflectionDemo1 {
  public static void main(String[] args) {

    try {

      <ReflectionDemo1 Class forName Call>

      System.out.println("Constructors:");
      <ReflectionDemo1 getConstructors Call>

      System.out.println("Fields:");
      <ReflectionDemo1 getFields Call>

      System.out.println("Methods:");
      <ReflectionDemo1 getMethods Call>

    }
    <Catch Exception>

  }
}
```

The following table lists called chunk definition points.

Chunk name	**First definition point**
<*Catch Exception*>	See "Catch Exception", page 137.
<*Import java.lang.reflect*>	See "Import java.lang.reflect", page 138.
<*ReflectionDemo1 Class forName Call*>	See "ReflectionDemo1 Class forName Call", page 137.
<*ReflectionDemo1 getConstructors Call*>	See "ReflectionDemo1 getConstructors Call", page 137.
<*ReflectionDemo1 getFields Call*>	See "ReflectionDem1 getFields Call", page 137.
<*ReflectionDemo1 getMethods Call*>	See "ReflectionDemo1 getMethods Call", page 137.

25.1.3.1 ReflectionDemo1 Class forName Call

<*ReflectionDemo1 Class forName Call*> ≡

```
Class<?> c = Class.forName("java.awt.Dimension");
```

This chunk is called by {`ReflectionDemo1.java`}; see its first definition at "Reflection Demonstration", page 136.

25.1.3.2 ReflectionDemo1 getConstructors Call

<*ReflectionDemo1 getConstructors Call*> ≡

```
Constructor<?> constructors[] = c.getConstructors();

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

This chunk is called by {`ReflectionDemo1.java`}; see its first definition at "Reflection Demonstration", page 136.

25.1.3.3 ReflectionDem1 getFields Call

<*ReflectionDemo1 getFields Call*> ≡

```
Filed fields[] = c.getFields();

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

This chunk is called by {`ReflectionDemo1.java`}; see its first definition at "Reflection Demonstration", page 136.

25.1.3.4 ReflectionDemo1 getMethods Call

<*ReflectionDemo1 getMethods Call*> ≡

```
Method methods[] = c.getMethods();

for (int i = 0; i < methods.length; i++) {
  System.out.print.n(" " + methods[i]);
}
```

This chunk is called by {`ReflectionDemo1.java`}; see its first definition at "Reflection Demonstration", page 136.

25.1.3.5 Catch Exception

<*Catch Exception*> ≡

```
catch (Exception e) {
  System.out.println("Exception: " + e);
```

```
}
```

This chunk is called by {`ReflectionDemo1.java`}; see its first definition at "Reflection Demonstration", page 136.

25.1.3.6 Import java.lang.reflect

<*Import java.lang.reflect*> ≡

```
import java.lang.reflect.*;
```

This chunk is called by {`ReflectionDemo1.java`}; see its first definition at "Reflection Demonstration", page 136.

25.2 Classes and Reflection

For every type of object, the Java virtual machine instantiates an immutable instance of `java.lang.Class` which provides methods to examine the runtime properties of the object including its members and type information.

`Class` also provides the ability to create new classes and objects. Most importantly, it is the entry point for all of the Reflective APIs.

25.2.1 `java.lang.Class`

Instances of the class `Class` represent classes and interfaces in a running Java application. An `enum` is a kind of class and an `annotation` is a kind of interface. Every array also belongs to a class that is reflected as a `Class` object that is shared by all arrays with the same element type and number of dimensions. The primitive Java types (`boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, and `double`), and the keyword `void` are also represented as `Class` objects.

`Class` has no public constructor. Instead `Class` objects are constructed automatically by the Java Virtual Machine as classes are loaded and by calls to the `defineClass` method in the class loader.

The following example uses a Class object to print the class name of an object:

```
void printClassName(Object obj) {
        System.out.println("The class of " + obj +
                           " is " + obj.getClass().getName());
    }
```

It is also possible to get the `Class` object for a named type (or for `void`) using a class literal.

```
System.out.println("The name of class Foo is: "+Foo.class.getName());
```

25.2.1.1 `Class` getConstructor

`Constructor<T>` [Constructor on `Class`]
`getConstructor` (*parameterType*)

Returns a `Constructor` object that reflects the specified public constructor of the class represented by this `Class` object.

`Class` getConstructors

`Constructor<?>[]` [Constructor on `Class`]
`getConstructors ()`

Returns an array containing `Constructor` objects reflecting all the public constructors of the class represented by this `Class` object.

25.2.2 Retrieving Class Objects

The entry point for all reflection operations is `java.lang.Class`. None of the classes in `java.lang.reflect` have public constructors.[1] To get to these classes, it is necessary to invoke appropriate methods on `Class`. There are several ways to get a `Class` depending on whether the code has access to an object, the name of the class, a type, or an existing `Class`.

25.2.2.1 `Object.getClass()`

If an instance of an object is available, then the simplest way to get its `Class` is to invoke `Object.getClass()`.

String.getClass()

```
Class c = "foo".getClass();
```

returns the `Class` for `String`;

System.console.getClass()

```
Class c = System.console().getClass();
```

returns the `Class` corresponding to `java.io.Console`.[2]

Enum.getClass()

```
enum E { A, B }
Class c  = A.getClass();
```

`A` is an instance of the `enum` `E`; thus, `getClass()` returns the `Class` corresponding to the enumeration type `E`.

Array.getClass()

Since arrays are `Objects`, it is possible to invoke `getClass()` on an instance of an array. The returned `Class` corresponds to an array with component type `byte`.

```
import java.util.HashSet;
import java.util.Set;

Set<String> s = new HashSet<String>();
Class c = s.getClass();
```

`java.util.Set` is an interface to an object of type `java.util.HashSet`. The value returned by `getClass()` is the class corresponding to `java.util.HashSet`.

---

[1] with the exception of `java.lang.reflect.ReflectPermission`.

[2] There is a unique console associated with the virtual machine which is returned by the `static` method `System.console()`.

25.2.2.2 The `.class` Syntax

If the type is available, but there is no instance, then it is possible to obtain a `Class` by appending `.class` to the name of the type. This is also the easiest way to obtain the `Class` for a primitive type.

Primitive.class

```
boolean b;
Class c = b.getClass();  // compile-time error
Class c = boolean.class; // correct
```

The statement '`boolean.getClass()`' would produce a compile-time error because a `boolean` is a primitive type and cannot be dereferenced. The `.class` syntax returns the `Class` corresponding to the type `boolean`.

Type.class

```
Class c = java.io.PrintStream.class;
```

The variable `c` will be the `Class` corresponding to the type `java.io.PrintStream`.

Multi-dimensional Array

```
Class c = int[][].class;
```

The `.class` syntax may be used to retrieve a `Class` corresonding to a multi-dimensional array of a given type.

25.2.2.3 `Class.forName()` and `Class.getName()` Methods

If the fully-qualified name of a class is available, it is possible to get the corresponding `Class` using the `static` method `Class.forName()`. This cannot be used for primitive types.

The syntax for names of array classes is described by `Class.getName()`. This syntax is applicable to references and primitive types.

`forName()`

```
Class c = Class.forName("com.duke.MyLocalServiceProvider");
```

This statement will create a class from the given fully-qualified name.

`getName()`

FIXME: should these be `getName()` instead of `forName()`?

```
Class cDoubleArray = Class.forName("[D");
Class cStringArray = Class.forName("[[Ljava.lang.String;");
```

The variable `cDoubleArray` will contain the `Class` corresponding to an array of primitive type `double` (i.e., the same as `double[].class`. The `cStringArray` variable will contain the `Class` corresponding to a two-dimensional array of `String` (i.e., identical to String[][].class).

25.2.2.4 TYPE Field for Primitive Type Wrappers

The `.class` syntax is a more convenient and the preferred way to obtain the `Class` for a primitive type; however, there is another way to acquire the `Class`. Each of the primitive types and `void` has a wrapper class in `java.lang` that is used for boxing of primitive types

to reference types. Each wrapper class contains a field name `TYPE` which is equal to the `Class` for the primitive type being wrapped.

Wrapper TYPE for Primitive

```
Class c = Double.TYPE;
```

There is a class `java.lang.Double` which is used to wrap the primitive type `double` whenever an `Object` is required. The value of `Double.TYPE` is identical to that of `double.class`.

Wrapper TYPE for void

```
Class c = Void.TYPE;
```

`Void.TYPE` is identical to `void.class`.

25.2.2.5 Methods that Return Classes

There are several `Reflection` APIs which return classes, but these may only be accessed if a `Class` has already been obtained, either directly or indirectly.

`Class.getSuperClass()`

Returns the super class for the given class.

```
Class c = javax.swing.JButton.class.getSuperclass();
```

`Class.getClasses()`

Returns all the public classes, interfaces, and enums that are members of the class including inherited members.

```
Class<?>[] c = Character.class.getClasses();
```

`Class.getDeclaredClasses()`

Returns all of the classes interfaces, and enums that are explicitly declared in this class.

```
Class<?>[] c = Character.class.getDeclaredClasses();
```

`Class.getDeclaringClass()`

java.lang.reflect.Field.getDeclaringClass()
java.lang.reflect.Method.getDeclaringClass()
java.lang.reflect.Constructor.getDeclaringClass()

Returns the `Class` in which these members were declared. `Anonymous Class Declarations` will not have a declaring class but will have an enclosing class.

```
import java.lang.reflect.Field;

Field f = System.class.getField("out");
Class c = f.getDeclaringClass();
```

The field out is declared in `System`.

```
public class MyClass {
    static Object o = new Object() {
        public void m() {}
    };
```

```
        static Class<c> = o.getClass().getEnclosingClass();
    }
```

The declaring class of the anonymous class defined by `o` is `null`

`Class.getEnclosingClass()`

Returns the immediately enclosing class of the class.

```
Class c = Thread.State.class().getEnclosingClass();
```

The enclosing class of the enum `Thread.State` is `Thread`.

```
public class MyClass {
    static Object o = new Object() {
        public void m() {}
    };
    static Class<c> = o.getClass().getEnclosingClass();
}
```

The anonymous class defined by `o` is enclosed by `MyClass`.

25.2.3 Examing Class Modifiers and Types

A class may be declared with one or more modifiers which affect its runtime behavior:

- Access modifiers: public, protected, and private
- Modifier requiring override: abstract
- Modifier restricting to one instance: static
- Modifier prohibiting value modification: final
- Modifier forcing strict floating point behavior: strictfp
- Annotations

Not all modifiers are allowed on all classes, for example an interface cannot be `final` and an `enum` cannot be `abstract`. `java.lang.reflect.Modifier` contains declarations for all possible modifiers. It also contains methods which may be used to decode the set of modifiers returned by `Class.getModifiers()`.

25.2.4 Discovering Class Members

There are two categories of methods provided in `Class` for accessing fields, methods, and constructors:

1. methods which enumerate these members; and
2. methods which search for particular members.

Also there are distinct methods for

- accessing members declared directly on the class; versus
- methods which search the superinterfaces and superclasses for inherited members.

The following tables provide a summary of all the member-locating methods and their characteristics.

Class Methods for Locating Fields

`Class` API	List of members?	Inherited members?	Private members?
getDeclarareField()	no	no	yes
getField()	no	yes	no
getDeclaredFields()	yes	no	yes
getFields()	yes	yes	no

Class Methods for Locating Methods

`Class` API	List of members?	Inherited members?	Private members?
getDeclarareMethod()	no	no	yes
getMethod()	no	yes	no
getDeclaredMethods()	yes	no	yes
getMethods()	yes	yes	no

Class Methods for Locating Constructors

`Class` API	List of members?	Inherited members?	Private members?
getDeclarareConstructor()	no	N/A[3]	yes
getConstructor()	no	N/A	no
getDeclaredConstructors()	yes	N/A	yes
getConstructors()	yes	N/A	no

Table 25.3: Class Methods for Locating Fields, Methods, and Constructors

25.3 Members and Reflection

Reflection defines an interface `java.lang.reflect.Member` which is implemented by

- `java.lang.reflect.Field`,
- `java.lang.reflect.Method`, and
- `java.lang.reflect.Constructor`.

For each member, the lesson will describe the associated APIs to retrieve declaration and type information, any operations unique to the member (for example, setting the value of a field or invoking a method), and commonly encountered errors. Each concept will be illustrated with code samples and related output which approximate some expected reflection uses.

Note about Members and the Specification

According to The Java Language Specification, Java SE 7 Edition, the *members of a class* are the inherited components of the class body including fields, methods, nested classes, in-

[3] Constructors are not inherited

terfaces, and enumerated types. Since constructors are not inherited, they are not members. This differs from the implementing classes of `java.lang.reflect.Member`.

25.3.1 `reflect.Fields`

Fields have a type and a value. The `java.lang.reflect.Field` class provides methods for accessing type information and setting and getting values of a field on a given object.

25.3.2 `reflect.Method`

Methods have return values, parameters, and may throw exceptions. The `java.lang.reflect.Method` class provides methods for obtaining the type information for the parameters and return value. It may also be used to invoke methods on a given object.

25.3.3 `reflect.Constructors`

The Reflection APIs for constructors are defined in `java.lang.reflect.Constructor` and are similar to those for methods, with two major exceptions: first, constructors have no return values; second, the invocation of a constructor creates a new instance of an object for a given class.

Similar to methods, reflection provides APIs to

- discover and retrieve the constructors of a class and
- obtain declaration information such as the modifiers, parameters, annotations, and thrown exceptions.
- New instances of classes may also be created using a specified constructor.

The key classes used when working with constructors are `Class` and `java.lang.reflect.Constructor`. Common operations involving constructors are covered in the following sections.

25.3.3.1 Finding Constructors

This section illustrates how to retrieve constructors with specific parameters.

A constructor declaration includes the name, modifiers, parameters, and list of throwable exceptions. The `java.lang.reflect.Constructor` class provides a way to obtain this information.

25.3.3.2 Retrieving and Parsing Constructor Modifiers

This section shows how to obtain the modifiers of a constructor declaration and other information about the constructor.

Because of the role of constructors in the language, fewer modifiers are meaningful than for methods:

- Access modifiers:
  - `public`,
  - `protected`, and
  - `private`
- Annotations

25.3.3.3 Creating New Class Instances

This section shows how to instantiate an instance of an object by invoking its constructor.

There are two reflective methods for creating instances of classes:

1. `java.lang.reflect.Constructor.newInstance()` and
2. `Class.newInstance()`.

The former is preferred and is thus used in these examples because:

- `Class.newInstance()` can only invoke the zero-argument constructor, while `Constructor.newInstance()` may invoke any constructor, regardless of the number of parameters.
- `Class.newInstance()` throws any exception thrown by the constructor, regardless of whether it is checked or unchecked. `Constructor.newInstance()` always wraps the thrown exception with an `InvocationTargetException`.
- `Class.newInstance()` requires that the constructor be visible; `Constructor.newInstance()` may invoke private constructors under certain circumstances.

25.4 Arrays and Enumerate Types and Reflection

From the Java virtual machine's perspective, arrays and enumerated types (or `enums`) are just classes. Many of the methods in `Class` may be used on them. Reflection provides a few specific APIs for arrays and enums. This lesson uses a series of code samples to describe how to distinguish each of these objects from other classes and operate on them. Various errors are also be examined.

25.4.1 Arrays and Reflection

Arrays have a component type and a length (which is not part of the type). Arrays may be maniuplated either in their entirety or component by component. Reflection provides the `java.lang.reflect.Array` class for the latter purpose.

25.4.1.1 Identifying Array Types

Array types may be identified by invoking `Class.isArray()`. To obtain a `Class` use one of the methods described in Section 25.2.2 "Retrieving Class Objects", page 139, section.

25.4.1.2 Creating New Arrays

Just as in non-reflective code, reflection supports the ability to dynamically create arrays of arbitrary type and dimensions via `java.lang.reflect.Array.newInstance()`. Consider ArrayCreator, a basic interpreter capable of dynamically creating arrays. The syntax that will be parsed is as follows:

```
fully_qualified_class_name variable_name[] =
{ val1, val2, val3, ... }
```

25.4.1.3 Getting and Setting Arrays and Their Components

Just as in non-reflective code, an array field may be set or retrieved in its entirety or component by component. To set the entire array at once, use `java.lang.reflect.Field.set(Object obj, Object value)`. To retrieve the entire

array, use `Field.get(Object)`. Individual components can be set or retrieved using methods in `java.lang.reflect.Array`.

`Array` provides methods of the form `setFoo()` and `getFoo()` for setting and getting components of any primitive type. For example, the component of an int array may be set with '`Array.setInt(Object array, int index, int value)`' and may be retrieved with '`Array.getInt(Object array, int index)`'.

Setting a Field of Type Array

Accessing Elements of a Multidimensional Array

25.4.2 Enumerated Types and Reflection

Enums are treated very much like ordinary classes in reflection code. `Class.isEnum()` tells whether a `Class` represents an enum. `Class.getEnumConstants()` retrieves the enum constants defined in an enum. `java.lang.reflect.Field.isEnumConstant()` indicates whether a field is an enumerated type.

25.4.2.1 Enumerated Types in Reflection

An *enum* is a language construct that is used to define type-safe enumerations which can be used when a fixed set of named values is desired. All `enums` implicitly extend `java.lang.Enum`. `Enums` may contain one or more `enum` constants, which define unique instances of the `enum` type. An `enum` declaration defines an `enum` type which is very similar to a class in that it may have members such as fields, methods, and constructors (with some restrictions).

Since `enums` are classes, reflection has no need to define an explicit java.lang.reflect.Enum class. The only Reflection APIs that are specific to `enums` are `Class.isEnum()`, `Class.getEnumConstants()`, and `java.lang.reflect.Field.isEnumConstant()`. Most reflective operations involving `enums` are the same as any other class or member. For example, `enum` constants are implemented as `public static final` fields on the `enum`. The following sections show how to use `Class` and `java.lang.reflect.Field` with `enums`.

Examining Enums

Illustrates how to retrieve an enum's constants and any other fields, constructors, and methods

Reflection provides three enum-specific APIs:

`Class.isEnum()`
: Indicates whether this class represents an enum type

`Class.getEnumConstants()`
: Retrieves the list of enum constants defined by the enum in the order they're declared

`java.lang.reflect.Field.isEnumConstant()`
: Indicates whether this field represents an element of an enumerated type

Sometimes it is necessary to dynamically retrieve the list of enum constants; in non-reflective code this is accomplished by invoking the implicitly declared static method

`values()` on the enum. If an instance of an `enum` type is not available the only way to get a list of the possible values is to invoke `Class.getEnumConstants()` since it is impossible to instantiate an `enum` type.

Getting and Setting Fields with Enum Types

Shows how to set and get fields with an enum constant value

Fields which store `enums` are set and retrieved as any other reference type, using `Field.set()` and `Field.get()`. For more information on accessing fields, see the Fields section. See Section 25.3.1 "`reflect.Fields`", page 144,

26 Introducinvg Swing

Appendix A The Makefile

{`Makefile`} ≡

<*Makefile CONSTANTS*>
<*Makefile DEFAULTS*>
<*Makefile TANGLE WEAVE*>
<*Makefile PDF*>
<*Makefile HTML*>
<*Makefile CLEAN Targets*>
<*Makefile MAKEFILE Target*>

The following table lists called chunk definition points.

Chunk name	First definition point
<*Makefile CLEAN Targets*>	See "Makefile Clean Targets", page 151.
<*Makefile CONSTANTS*>	See "Makefile Constants", page 149.
<*Makefile DEFAULTS*>	See "Makefile Default Targets", page 149.
<*Makefile HTML*>	See "Makefile HTML", page 151.
<*Makefile MAKEFILE Target*>	See "Make the Makefile", page 149.
<*Makefile PDF*>	See "Makefile PDF", page 150.
<*Makefile TANGLE WEAVE*>	See "Makefile Tangle Weave Targets", page 150.

A.1 Makefile Constants

FIXME: Relativize ROOT

<*Makefile CONSTANTS*> ≡

```
ROOT := /usr/local/dev/programming/Java/JavaSE9
FILENAME := JavaSE9
AUX := {aux,cps,fns,log,toc}
```

This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 149.

A.2 Makefile Default Targets

<*Makefile DEFAULTS*> ≡

```
.PHONY: all
all: tangle weave
```

This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 149.

A.3 Make the Makefile

<*Makefile MAKEFILE Target*> ≡

```
.PHONY : makefile
makefile : jrtangle worldclean
```

This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 149.

A.4 Makefile Tangle Weave Targets

<*Makefile TANGLE WEAVE*> ≡

```
.PHONY: tangle weave jrtangle jrweave texi

tangle: jrtangle
weave: jrweave

jrtangle: $(FILENAME).twjr
    jrtangle $(FILENAME).twjr

jrweave: texi

texi: $(FILENAME).texi

$(FILENAME).texi: $(FILENAME).twjr
    jrweave $(FILENAME).twjr > $(FILENAME).texi
```

This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 149.

A.5 Makefile PDF

<*Makefile PDF*> ≡

<*Makefile MAKEPDF*>
<*Makefile OPENPDF*>

This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 149.
The following table lists called chunk definition points.

Chunk name	**First definition point**
<*Makefile MAKEPDF*>	See "Makefile MAKEPDF", page 150.
<*Makefile OPENPDF*>	See "Makefile OPENPDF", page 150.

A.5.1 Makefile MAKEPDF

<*Makefile MAKEPDF*> ≡

```
.PHONY : makepdf
makepdf : ${FILENAME}.pdf

${FILENAME}.pdf : ${FILENAME}.texi
    pdftexi2dvi ${FILENAME}.texi
```

This chunk is called by <*Makefile PDF*>; see its first definition at "Makefile PDF", page 150.

A.5.1.1 Makefile OPENPDF

<*Makefile OPENPDF*> ≡

```
.PHONY : pdf
pdf : makepdf
    open ${FILENAME}.pdf
```

This chunk is called by <*Makefile PDF*>; see its first definition at "Makefile PDF", page 150.

A.6 Makefile HTML

<*Makefile HTML*> ≡

```
.PHONY: html
html : ${FILENAME}.texi
    makeinfo --html ${FILENAME}.texi
```

This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 149.

A.7 Makefile Clean Targets

<*Makefile CLEAN Targets*> ≡

<*Makefile CLEAN*>
<*Makefile DISTCLEAN*>
<*Makefile WORLDCLEAN*>

This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 149.

The following table lists called chunk definition points.

Chunk name	**First definition point**
<*Makefile CLEAN*>	See "Makefile Clean", page 151.
<*Makefile DISTCLEAN*>	See "Makefile DistClean", page 151.
<*Makefile WORLDCLEAN*>	See "Makefile WorldClean", page 152.

A.7.1 Makefile Clean

<*Makefile CLEAN*> ≡

```
.PHONY: clean
clean:
    rm -f *~
    rm -f $(FILENAME).??
```

This chunk is called by <*Makefile CLEAN Targets*>; see its first definition at "Makefile Clean Targets", page 151.

A.7.2 Makefile DistClean

<*Makefile DISTCLEAN*> ≡

```
.PHONY : distclean
distclean : clean
    rm -fv ${FILENAME}.${AUX}
```

This chunk is called by <*Makefile CLEAN Targets*>; see its first definition at "Makefile Clean Targets", page 151.

A.7.3 Makefile WorldClean

<*Makefile WORLDCLEAN*> ≡

```
.PHONY: worldclean
worldclean :
    for file in ${ROOT}/*; do \
      [ \
        $${file} != "${ROOT}/${FILENAME}.twjr" -a \
        $$file != "${ROOT}/Makefile" -a \
        $$file != ".git" \
      ] \
      && rm -rfv "$${file}" || :; \
    done
```

This chunk is called by <*Makefile CLEAN Targets*>; see its first definition at "Makefile Clean Targets", page 151.

Appendix B Code Chunk Summaries

This appendix presents alphabetical lists of all the file definitions, the code chunk definitions, and the code chunk references.

B.1 Source File Definitions

{AbstractAreas.java }
: This chunk is defined in "Improved Figure Class", page 27.

{BRRead.java}
: This chunk is defined in "Reading Characters", page 44.

{BRReadLines.java}
: This chunk is defined in "Reading Strings", page 45.

{BufferedInputStreamDemo.java}
: This chunk is defined in "Buffered Input Example", page 112.

{BufferedReaderDemo.java}
: This chunk is defined in "Buffered Reader Demo", page 120.

{CopyFile.java}
: This chunk is defined in "Demonstration Writing to a File", page 55.

{CopyFileMultTryWR.java}
: This chunk is defined in "Demonstration of Multiple Resources", page 60.

{DirList.java}
: This chunk is defined in "Using `list()` to Examine Directory Contents", page 102.

{DirListOnly}
: This chunk is defined in "Example Program Using FilenameFilter Interface", page 105.

{FileReaderDemo.java}
: This chunk is defined in "`FileReader`", page 119.

{FindAreas.java }
: This chunk is defined in "Applying", page 21.

{GenMethDemo.java}
: This chunk is defined in "Example of Generic Method", page 75.

{Makefile}
: This chunk is defined in "The Makefile", page 149.

{OnlyExt.java}
: This chunk is defined in "Example Program Using FilenameFilter Interface", page 104.

{PrinterWriterDemo.java}
: This chunk is defined in "Demonstration Using a `PrintWriter` for Console Output", page 47.

{ReflectionDemo1.java}
This chunk is defined in "Reflection Demonstration", page 136.

{ShowFile.java}
This chunk is defined in "Demonstration Reading From a File", page 49.

{ShowFileAlt.java}
This chunk is defined in "`close()` Within `finally` Block", page 52.

{ShowFileSingleTry}
This chunk is defined in "Demonstration Reading From a File with a Single `try` Block", page 53.

{ShowFileTryWR.java}
This chunk is defined in "Demonstration of Automatically Closing a File", page 58.

{SimpleGenerics.java}
This chunk is defined in "A Simple Generics Example", page 64.

{Stack.java}
This chunk is defined in "A Stack Class", page 6.

{StackImproved.java}
This chunk is defined in "An Improved `Stack` Class", page 13.

{TestStack.java}
This chunk is defined in "A Stack Class", page 7.

{TwoTypeParameters.java}
This chunk is defined in "Example of Code with Two Type Parameters", page 70.

B.2 Code Chunk Definitions

<*AbstractAreas Abstract Area Method Declaration* >
This chunk is defined in "AbstractAreas Abstract Class Figure Section", page 27.

<*AbstractAreas Abstract Class Figure* >
This chunk is defined in "AbstractAreas Abstract Class Figure Section", page 27.

<*AbstractAreas Main Class* >
This chunk is defined in "Abstract Main Class", page 27.

<*AbstractAreas Main Method Declaration* >
This chunk is defined in "Abstract Main Class", page 28.

<*BRRead BufferedReader Constructor*>
This chunk is defined in "BRRead BufferedReader Constructor Section", page 45.

<*BRRead Enter Characters*>
This chunk is defined in "BRRead Enter Characters Section", page 45.

B.3 Code Chunk References

<*AbstractAreas Abstract Area Method Declaration* >
This chunk is called by <*AbstractAreas Abstract Class Figure* >; see its first definition at "AbstractAreas Abstract Class Figure Section", page 27.

<*AbstractAreas Abstract Class Figure* >
This chunk is called by {`AbstractAreas.java` }; see its first definition at "Improved Figure Class", page 27.

<*AbstractAreas Main Class* >
This chunk is called by {`AbstractAreas.java` }; see its first definition at "Improved Figure Class", page 27.

<*AbstractAreas Main Method Declaration* >
This chunk is called by <*AbstractAreas Main Class* >; see its first definition at "Abstract Main Class", page 27.

<*BRRead BufferedReader Constructor*>
This chunk is called by {`BRRead.java`}; see its first definition at "Reading Characters", page 44.

<*BRRead Enter Characters*>
This chunk is called by {`BRRead.java`}; see its first definition at "Reading Characters", page 44.

<*BRReadLines BufferedReader Constructor*>
This chunk is called by {`BRReadLines.java`}; see its first definition at "Reading Strings", page 45.

<*BRReadLines Enter Lines*>
This chunk is called by {`BRReadLines.java`}; see its first definition at "Reading Strings", page 45.

<*BufferedInputStreamDemo Buffer*>
This chunk is called by <*BufferedInputStreamDemo Instance Variables*>; see its first definition at "BufferedInputStreamDemo Instance Variables", page 113.

<*BufferedInputStreamDemo ByteArrayInputStream*>
This chunk is called by <*BufferedInputStreamDemo Instance Variables*>; see its first definition at "BufferedInputStreamDemo Instance Variables", page 113.

<*BufferedInputStreamDemo Instance Variables*>
This chunk is called by {`BufferedInputStreamDemo.java`}; see its first definition at "Buffered Input Example", page 112.

<*BufferedInputStreamDemo String*>
This chunk is called by the following chunks:

Chunk name	**First definition point**
<*BufferedInputStreamDemo Instance Variables*>	See "BufferedInputStreamDemo Instance Variables", page 113.
<*BufferedReaderDemo Instance Variables*>	See "BufferedReaderDemo Instance Variables", page 121.

<*BufferedInputStreamDemo Switch on Character*>
This chunk is called by the following chunks:

Chunk name	First definition point
<*BufferedInputStreamDemo While Loop*>	See "BufferedInputStreamDemo While Loop", page 114.
<*BufferedReaderDemo TryWithResources BufferedReader*>	See "BufferedReaderDemo TryWithResources BufferedReader", page 122.

<*BufferedInputStreamDemo TryWithResources BufferedInputStream*>
This chunk is called by {`BufferedInputStreamDemo.java`}; see its first definition at "Buffered Input Example", page 112.

<*BufferedInputStreamDemo Utility Variables*>
This chunk is called by <*BufferedInputStreamDemo Instance Variables*>; see its first definition at "BufferedInputStreamDemo Instance Variables", page 113.

<*BufferedInputStreamDemo While Loop*>
This chunk is called by <*BufferedInputStreamDemo TryWithResources BufferedInputStream*>; see its first definition at "BufferedInputStreamDemo TryWithResources BufferedInputStream", page 113.

<*BufferedReaderDemo Buffer*>
This chunk is called by <*BufferedReaderDemo Instance Variables*>; see its first definition at "BufferedReaderDemo Instance Variables", page 121.

<*BufferedReaderDemo Instance Variables*>
This chunk is called by {`BufferedReaderDemo.java`}; see its first definition at "Buffered Reader Demo", page 120.

<*BufferedReaderDemo TryWithResources BufferedReader*>
This chunk is called by {`BufferedReaderDemo.java`}; see its first definition at "Buffered Reader Demo", page 120.

<*Call Overridden Methods One By One* >
This chunk is called by <*FindAreas Main Method Declaration* >; see its first definition at "FindAreas Main Class Section", page 24.

<*Call Overridden Methods One By One Except Figure* >
This chunk is called by <*AbstractAreas Main Method Declaration* >; see its first definition at "Abstract Main Class", page 28.

<*Catch Exception*>
This chunk is called by {`ReflectionDemo1.java`}; see its first definition at "Reflection Demonstration", page 136.

<*Catch IOException*>
This chunk is called by the following chunks:

Chunk name	First definition point
{`BufferedInputStreamDemo.java`}	See "Buffered Input Example", page 112.

{BufferedReaderDemo.java} See "Buffered Reader Demo", page 120.
{FileReaderDemo.java} See "FileReader", page 119.

<*Class Declaration*>
This chunk is called by <*Class TwoGen*>; see its first definition at "Class TwoGen", page 70.

<*Class Gen*>
This chunk is called by {SimpleGenerics.java}; see its first definition at "A Simple Generics Example", page 64.

<*Class GenDemo*>
This chunk is called by {SimpleGenerics.java}; see its first definition at "A Simple Generics Example", page 64.

<*Class SimpGen*>
This chunk is called by {TwoTypeParameters.java}; see its first definition at "Example of Code with Two Type Parameters", page 70.

<*Class TwoGen*>
This chunk is called by {TwoTypeParameters.java}; see its first definition at "Example of Code with Two Type Parameters", page 70.

<*Constructor of Two Parameters*>
This chunk is called by <*Class TwoGen*>; see its first definition at "Class TwoGen", page 70.

<*Constructor taking parameter of Type T*>
This chunk is called by <*Class Gen*>; see its first definition at "Class Gen¡T¿", page 65.

<*CopyFile Check For 2 Files*>
This chunk is called by {CopyFile.java}; see its first definition at "Demonstration Writing to a File", page 55.

<*CopyFile Copy a File*>
This chunk is called by {CopyFile.java}; see its first definition at "Demonstration Writing to a File", page 55.

<*CopyFile Initial Comments*>
This chunk is called by {CopyFile.java}; see its first definition at "Demonstration Writing to a File", page 55.

<*CopyFile Instance Variable Declarations*>
This chunk is called by {CopyFile.java}; see its first definition at "Demonstration Writing to a File", page 55.

<*CopyFileMultTryWR Initial Comments*>
This chunk is called by {CopyFileMultTryWR.java}; see its first definition at "Demonstration of Multiple Resources", page 60.

<*CopyFileMultTryWR Manage Two Files*>
This chunk is called by {CopyFileMultTryWR.java}; see its first definition at "Demonstration of Multiple Resources", page 60.

<*Create Basic Figure Objects* >
This chunk is called by <*FindAreas Main Method Declaration* >; see its first definition at "FindAreas Main Class Section", page 24.

<*Create Basic Figure Objects Except Figure* >
This chunk is called by <*AbstractAreas Main Method Declaration* >; see its first definition at "Abstract Main Class", page 28.

<*Create Basic Figure Reference Variable* >
This chunk is called by the following chunks:

Chunk name	First definition point
<*AbstractAreas Main Method Declaration* >	See "Abstract Main Class", page 28.
<*FindAreas Main Method Declaration* >	See "FindAreas Main Class Section", page 24.

<*Create a Gen object for Integers*>
This chunk is called by <*Class GenDemo*>; see its first definition at "Class GenDemo", page 66.

<*Create a Gen object for Strings*>
This chunk is called by <*Class GenDemo*>; see its first definition at "Class GenDemo", page 66.

<*DirList Examine Directory Contents*>
This chunk is called by {`DirList.java`}; see its first definition at "Using `list()` to Examine Directory Contents", page 102.

<*DirList Examine Directory Contents For-Loop*>
This chunk is called by <*DirList Examine Directory Contents*>; see its first definition at "DirList Examine Directory Contents", page 103.

<*DirList Instance Variable Declarations*>
This chunk is called by {`DirList.java`}; see its first definition at "Using `list()` to Examine Directory Contents", page 102.

<*DirList Obtain Directory From Command-Line Args*>
This chunk is called by the following chunks:

Chunk name	First definition point
<*DirList Instance Variable Declarations*>	See "DirList Instance Variable Declarations", page 102.
{`DirListOnly`}	See "Example Program Using FilenameFilter Interface", page 105.

<*DirListOnly FilenameFilter Object*>
This chunk is called by {`DirListOnly`}; see its first definition at "Example Program Using FilenameFilter Interface", page 105.

<*DirListOnly FilenameFilter Object List*>
This chunk is called by {`DirListOnly`}; see its first definition at "Example Program Using FilenameFilter Interface", page 105.

<*DirListOnly Print List*>
This chunk is called by {`DirListOnly`}; see its first definition at "Example Program Using FilenameFilter Interface", page 105.

<*Figure Area Method Declaration* >
This chunk is called by <*FindAreas SuperClass Figure* >; see its first definition at "FindAreas Superclass Figure Section", page 21.

<*Figure Constructor* >
This chunk is called by the following chunks:

Chunk name	First definition point
<*AbstractAreas Abstract Class Figure* >	See "AbstractAreas Abstract Class Figure Section", page 27.
<*FindAreas SuperClass Figure* >	See "FindAreas Superclass Figure Section", page 21.

<*Figure Instance Variable Declarations* >
This chunk is called by the following chunks:

Chunk name	First definition point
<*AbstractAreas Abstract Class Figure* >	See "AbstractAreas Abstract Class Figure Section", page 27.
<*FindAreas SuperClass Figure* >	See "FindAreas Superclass Figure Section", page 21.

<*FileReaderDemo TryWithResources FileReader*>
This chunk is called by {`FileReaderDemo.java`}; see its first definition at "`FileReader`", page 119.

<*FindAreas Main Class* >
This chunk is called by {`FindAreas.java` }; see its first definition at "Applying", page 21.

<*FindAreas Main Method Declaration* >
This chunk is called by <*FindAreas Main Class* >; see its first definition at "FindAreas Main Class Section", page 24.

<*FindAreas SubClass Rectangle* >
This chunk is called by the following chunks:

Chunk name	First definition point
{`AbstractAreas.java` }	See "Improved Figure Class", page 27.
{`FindAreas.java` }	See "Applying", page 21.

<*FindAreas SubClass Triangle* >
This chunk is called by the following chunks:

Chunk name	First definition point
{`AbstractAreas.java` }	See "Improved Figure Class", page 27.
{`FindAreas.java` }	See "Applying", page 21.

<*FindAreas SuperClass Figure* >
This chunk is called by {`FindAreas.java` }; see its first definition at "Applying", page 21.

<*GenMethDemo Main*>
This chunk is called by {`GenMethDemo.java`}; see its first definition at "Example of Generic Method", page 75.

<*Get Value*>
This chunk is called by <*Create a Gen object for Integers*>; see its first definition at "Implementation of Class GenDemo with Type Integer", page 67.

<*Import java.io*>
This chunk is called by the following chunks:

Chunk name	First definition point
{`BRRead.java`}	See "Reading Characters", page 44.
{`BRReadLines.java`}	See "Reading Strings", page 45.
{`BufferedInputStreamDemo.java`}	See "Buffered Input Example", page 112.
{`BufferedReaderDemo.java`}	See "Buffered Reader Demo", page 120.
{`CopyFile.java`}	See "Demonstration Writing to a File", page 55.
{`CopyFileMultTryWR.java`}	See "Demonstration of Multiple Resources", page 60.
{`DirListOnly`}	See "Example Program Using FilenameFilter Interface", page 105.
{`FileReaderDemo.java`}	See "`FileReader`", page 119.
{`OnlyExt.java`}	See "Example Program Using FilenameFilter Interface", page 104.
{`PrinterWriterDemo.java`}	See "Demonstration Using a `PrintWriter` for Console Output", page 47.
{`ShowFile.java`}	See "Demonstration Reading From a File", page 49.
{`ShowFileAlt.java`}	See "`close()` Within `finally` Block", page 52.
{`ShowFileSingleTry`}	See "Demonstration Reading From a File with a Single `try` Block", page 53.
{`ShowFileTryWR.java`}	See "Demonstration of Automatically Closing a File", page 58.

<*Import java.io.File*>
This chunk is called by {`DirList.java`}; see its first definition at "Using `list()` to Examine Directory Contents", page 102.

<*Import java.lang.reflect*>
This chunk is called by {`ReflectionDemo1.java`}; see its first definition at "Reflection Demonstration", page 136.

<*Instance Methods Show and Get*>
This chunk is called by <*Class TwoGen*>; see its first definition at "Class TwoGen", page 70.

<*Instance Variable ob of Type T*>
This chunk is called by <*Class Gen*>; see its first definition at "Class Gen¡T¿", page 65.

<*Integer Type Parameter*>
This chunk is called by <*Create a Gen object for Integers*>; see its first definition at "Implementation of Class GenDemo with Type Integer", page 67.

<*Makefile CLEAN*>
This chunk is called by <*Makefile CLEAN Targets*>; see its first definition at "Makefile Clean Targets", page 151.

<*Makefile CLEAN Targets*>
This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 149.

<*Makefile CONSTANTS*>
This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 149.

<*Makefile DEFAULTS*>
This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 149.

<*Makefile DISTCLEAN*>
This chunk is called by <*Makefile CLEAN Targets*>; see its first definition at "Makefile Clean Targets", page 151.

<*Makefile HTML*>
This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 149.

<*Makefile MAKEFILE Target*>
This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 149.

<*Makefile MAKEPDF*>
This chunk is called by <*Makefile PDF*>; see its first definition at "Makefile PDF", page 150.

<*Makefile OPENPDF*>
This chunk is called by <*Makefile PDF*>; see its first definition at "Makefile PDF", page 150.

<*Makefile PDF*>
This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 149.

<*Makefile TANGLE WEAVE*>
This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 149.

<*Makefile WORLDCLEAN*>
This chunk is called by <*Makefile CLEAN Targets*>; see its first definition at "Makefile Clean Targets", page 151.

<*Method returning object of type T*>
This chunk is called by <*Class Gen*>; see its first definition at "Class Gen¡T¿", page 65.

<*ReflectionDemo1 getMethods Call*>
This chunk is called by {`ReflectionDemo1.java`}; see its first definition at "Reflection Demonstration", page 136.

<*Show Type*>
This chunk is called by <*Create a Gen object for Integers*>; see its first definition at "Implementation of Class GenDemo with Type Integer", page 67.

<*ShowFile Close a File*>
This chunk is called by the following chunks:

Chunk name	First definition point
{`ShowFile.java`}	See "Demonstration Reading From a File", page 49.
{`ShowFileAlt.java`}	See "`close()` Within `finally` Block", page 52.

<*ShowFile Initial Comments*>
This chunk is called by the following chunks:

Chunk name	First definition point
{`ShowFile.java`}	See "Demonstration Reading From a File", page 49.
{`ShowFileAlt.java`}	See "`close()` Within `finally` Block", page 52.
{`ShowFileSingleTry`}	See "Demonstration Reading From a File with a Single `try` Block", page 53.

<*ShowFile Instance Variable Declarations*>
This chunk is called by the following chunks:

Chunk name	First definition point
{`ShowFile.java`}	See "Demonstration Reading From a File", page 49.
{`ShowFileAlt.java`}	See "`close()` Within `finally` Block", page 52.
{`ShowFileSingleTry`}	See "Demonstration Reading From a File with a Single `try` Block", page 53.

<*ShowFile Open a File*>
This chunk is called by the following chunks:

Chunk name	First definition point
{`ShowFile.java`}	See "Demonstration Reading From a File", page 49.
{`ShowFileAlt.java`}	See "`close()` Within `finally` Block", page 52.
{`ShowFileSingleTry`}	See "Demonstration Reading From a File with a Single `try` Block", page 53.

<*ShowFile Read a File*>
This chunk is called by {`ShowFile.java`}; see its first definition at "Demonstration Reading From a File", page 49.

<*ShowFileAlt Read a File*>
This chunk is called by {`ShowFileAlt.java`}; see its first definition at "`close()` Within `finally` Block", page 52.

<*ShowFileSingleTry Additional Initial Comment*>
This chunk is called by {`ShowFileSingleTry`}; see its first definition at "Demonstration Reading From a File with a Single `try` Block", page 53.

<*ShowFileSingleTry Read a File*>
This chunk is called by {`ShowFileSingleTry`}; see its first definition at "Demonstration Reading From a File with a Single `try` Block", page 53.

<*ShowFileTryWR Check CL Args*>
This chunk is called by the following chunks:

Chunk name	**First definition point**
{`CopyFileMultTryWR.java`}	See "Demonstration of Multiple Resources", page 60.
{`ShowFileTryWR.java`}	See "Demonstration of Automatically Closing a File", page 58.

<*ShowFileTryWR Check CL Args End*>
This chunk is called by the following chunks:

Chunk name	**First definition point**
{`CopyFileMultTryWR.java`}	See "Demonstration of Multiple Resources", page 60.
{`ShowFileTryWR.java`}	See "Demonstration of Automatically Closing a File", page 58.

<*ShowFileTryWR Initial Comments*>
This chunk is called by {`ShowFileTryWR.java`}; see its first definition at "Demonstration of Automatically Closing a File", page 58.

<*ShowFileTryWR Instance Variable Declaration*>
This chunk is called by the following chunks:

Chunk name	**First definition point**
{`CopyFileMultTryWR.java`}	See "Demonstration of Multiple Resources", page 60.
{`ShowFileTryWR.java`}	See "Demonstration of Automatically Closing a File", page 58.

<*ShowFileTryWR Open a File TryWR*>
This chunk is called by {`ShowFileTryWR.java`}; see its first definition at "Demonstration of Automatically Closing a File", page 58.

<*Stack Constructor*>
This chunk is called by the following chunks:

Chunk name	**First definition point**
{`Stack.java`}	See "A Stack Class", page 6.
{`StackImproved.java`}	See "An Improved `Stack` Class", page 13.

<*Stack Instance Methods*>
This chunk is called by the following chunks:

Chunk name	**First definition point**
{`Stack.java`}	See "A Stack Class", page 6.
{`StackImproved.java`}	See "An Improved `Stack` Class", page 13.

<*Stack Instance Variables*>
This chunk is called by {`Stack.java`}; see its first definition at "A Stack Class", page 6.

<*Stack Pop*>
This chunk is called by <*Stack Instance Methods*>; see its first definition at "Stack Instance Methods Subsection", page 7.

<*Stack Private Instance Variables*>
This chunk is called by {`StackImproved.java`}; see its first definition at "An Improved `Stack` Class", page 13.

<*Stack Push*>
This chunk is called by <*Stack Instance Methods*>; see its first definition at "Stack Instance Methods Subsection", page 7.

<*Static Method isIn*>
This chunk is called by {`GenMethDemo.java`}; see its first definition at "Example of Generic Method", page 75.

<*TestStack Main Method*>
This chunk is called by {`TestStack.java`}; see its first definition at "A Stack Class", page 7.

<*Triangle Area Method Declaration* >
This chunk is called by <*FindAreas SubClass Triangle* >; see its first definition at "FindAreas SubClass Triangle Section", page 23.

<*Triangle Constructor* >
This chunk is called by <*FindAreas SubClass Triangle* >; see its first definition at "FindAreas SubClass Triangle Section", page 23.

<*Two Instance Variables Declarations*>
This chunk is called by <*Class TwoGen*>; see its first definition at "Class TwoGen", page 70.

List of Tables

List of General Forms

Bibliography

Index

=

{

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Function Index