Outline Covering Java SE 9

SEPTEMBER, 2018

LOLH

Short Contents

Table of Contents

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 wihtin 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 16 “Event Handling”, page 68. 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 `String`s

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 10 "`java.lang`", page 60.

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 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.

7.1 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 Integers: '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.

7.2 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.

7.3 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 41.
<*Class GenDemo*>	See "Class GenDemo", page 42.

7.3.1 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 40.

The following table lists called chunk definition points.

Chunk name	First definition point
<*Constructor taking parameter of Type T*>	See “Class Gen¡T¿”, page 41.
<*Instance Variable ob of Type T*>	See “Class Gen¡T¿”, page 41.
<*Method returning object of type T*>	See “Class Gen¡T¿”, page 42.
<*Method showing type of T*>	See “Class Gen¡T¿”, page 42.

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 41.

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 41.

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 41.

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 41.

7.3.2 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 40.

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 43.
<*Create a Gen object for Strings*>	See "Implementation of Class GenDemo with Type String", page 44.

7.3.2.1 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 42.

The following table lists called chunk definition points.

Chunk name	First definition point
<*Get Value*>	See "Implementation of Class GenDemo with Type Integer", page 44.
<*Integer Type Parameter*>	See "Implementation of Class GenDemo with Type Integer", page 43.
<*Reference to Integer Instance*>	See "Implementation of Class GenDemo with Type Integer", page 44.
<*Show Type*>	See "Implementation of Class GenDemo with Type Integer", page 44.

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 43.

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 43.

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 43.

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 43.

7.3.2.2 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 42.

7.4 Notes About Generics

7.4.1 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.

7.4.2 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.

7.4.3 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>`.

7.4.4 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.

7.5 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.

7.5.1 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 47.
<*Class TwoGen*>	See “Class TwoGen”, page 46.

7.5.1.1 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 46.

The following table lists called chunk definition points.

Chunk name	**First definition point**
<*Class Declaration*>	See “Class TwoGen”, page 46.
<*Constructor of Two Parameters*>	See “Class TwoGen”, page 46.
<*Instance Methods Show and Get*>	See “Class TwoGen”, page 47.
<*Two Instance Variables Declarations*>	See “Class TwoGen”, page 46.

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 46.

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 46.

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 46.

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 46.

7.5.1.2 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 46.

7.6 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 7.1: General Form for Declaring and Creating a Reference to a Generic Class

7.7 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`.

7.8 Using Wildcard Arguments

7.8.1 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.

7.8.2 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.

7.8.3 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 7.2: 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 7.3: General Form of Lower Bounded Wildcard Syntax

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

7.9 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 7.4: General Form for Declaring a Generic Method

7.9.1 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 51.
<*Static Method isIn*>	See "Method isIn()", page 51.

7.9.1.1 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 51.

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)
```

7.9.1.2 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 51.

7.10 Generic Constructors

It is possible for constructors to be generic, even if their class is not (see "Class Gen¡T¿", page 41). The syntax is the same (type parameters come first).

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

8 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

8.1 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`'.

8.2 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.

8.3 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.

8.4 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

9 String Handling

10 `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

10.1 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*.

10.1.1 Number

10.1.2 Double and Float

10.1.3 isInfinite() and isNaN()

10.1.4 Byte, Short, Integer, Long

10.1.5 Converting Numbers to and from String

11 `java.util` — Part 1: The Collections Framework

12 `java.util` — Part 2: Utility Classes

13 `java.io` — Input/Output

14 NIO

15 Networking

16 Event Handling

17 AWT: Working with Windows, Graphics, and Text

18 Using AWT Controls, Layout Managers, and Menus

19 Images

20 The Concurrency Utilities

21 The Stream API

22 Regular Expressions and Other Packages

23 Introducinvg Swing

Appendix A The Makefile

{`Makefile`} ≡

<*Makefile CONSTANTS*>
<*Makefile DEFAULTS*>
<*Makefile TANGLE WEAVE*>
<*Makefile CLEAN*>

The following table lists called chunk definition points.

Chunk name	First definition point
<*Makefile CLEAN*>	See "Makefile Clean Targets", page 77.
<*Makefile CONSTANTS*>	See "Makefile Constants", page 76.
<*Makefile DEFAULTS*>	See "Makefile Default Targets", page 76.
<*Makefile TANGLE WEAVE*>	See "Makefile Tangle Weave Targets", page 76.

A.1 Makefile Constants

<*Makefile CONSTANTS*> ≡

```
FILENAME := JavaSE9
```

This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 76.

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 76.

A.3 Makefile Tangle Weave Targets

<*Makefile TANGLE WEAVE*> ≡

```
.PHONY: tangle weave jrtangle jrweave
tangle: jrtangle
weave: jrweave

jrtangle: $(FILENAME).twjr
    jrtangle $(FILENAME).twjr

jrweave: $(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 76.

A.4 Makefile Clean Targets

<*Makefile CLEAN*> ≡

```
.PHONY: clean
clean:
    rm -f *~
    rm -f $(FILENAME).???
```

This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 76.

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.

{`FindAreas.java` }
: This chunk is defined in "Applying", page 21.

{`GenMethDemo.java`}
: This chunk is defined in "Example of Generic Method", page 51.

{`Makefile`}
: This chunk is defined in "The Makefile", page 76.

{`SimpleGenerics.java`}
: This chunk is defined in "A Simple Generics Example", page 40.

{`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 46.

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.

<*Call Overridden Methods One By One* >
: This chunk is defined in "FindAreas Main Class Section", page 25.

<*Stack Instance Methods*>
: This chunk is defined in "Stack Instance Methods Subsection", page 7.

<*Stack Instance Variables*>
: This chunk is defined in "Stack Instance Variables", page 7.

<*Stack Pop*>
: This chunk is defined in "Stack Push and Pop Subsubsection", page 8.

<*Stack Private Instance Variables*>
: This chunk is defined in "An Improved `Stack` Class", page 13.

<*Stack Push*>
: This chunk is defined in "Stack Push and Pop Subsubsection", page 8.

<*Static Method isIn*>
: This chunk is defined in "Method isIn()", page 51.

<*TestStack Main Method*>
: This chunk is defined in "Stack TestStack Subsection", page 8.

<*Triangle Area Method Declaration* >
: This chunk is defined in "FindAreas SubClass Triangle Section", page 24.

<*Triangle Constructor* >
: This chunk is defined in "FindAreas SubClass Triangle Section", page 24.

<*Two Instance Variables Declarations*>
: This chunk is defined in "Class TwoGen", page 46.

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.

<*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.

<*Class Declaration*>
: This chunk is called by <*Class TwoGen*>; see its first definition at "Class TwoGen", page 46.

<*Class Gen*>
: This chunk is called by {`SimpleGenerics.java`}; see its first definition at "A Simple Generics Example", page 40.

<*Class GenDemo*>
: This chunk is called by {`SimpleGenerics.java`}; see its first definition at "A Simple Generics Example", page 40.

<*Class SimpGen*>
: This chunk is called by {`TwoTypeParameters.java`}; see its first definition at "Example of Code with Two Type Parameters", page 46.

<*Class TwoGen*>
: This chunk is called by {`TwoTypeParameters.java`}; see its first definition at "Example of Code with Two Type Parameters", page 46.

<*Constructor of Two Parameters*>
: This chunk is called by <*Class TwoGen*>; see its first definition at "Class TwoGen", page 46.

<*Constructor taking parameter of Type T*>
: This chunk is called by <*Class Gen*>; see its first definition at "Class Gen¡T¿", page 41.

<*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 42.

<*Create a Gen object for Strings*>
: This chunk is called by <*Class GenDemo*>; see its first definition at "Class GenDemo", page 42.

<*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.

<*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 51.

<*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 51.

<*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 46.

List of Tables

List of General Forms

Bibliography

Index

<

=

{

A

B

C

D

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Function Index