

# Outline Covering Java SE 9

---

SEPTEMBER, 2018

LOLH

---

## Short Contents

### The Java Language

1	Java SE 9 Introduction . . . . .	3
2	Classes . . . . .	4
3	Methods and Classes . . . . .	10
4	Packages . . . . .	15
5	Interfaces . . . . .	19
6	Generics . . . . .	23
7	Enumerations . . . . .	37

### The Java Standard Library

8	String Handling . . . . .	43
9	<code>java.lang</code> . . . . .	44
10	<code>java.util</code> — Part 1: The Collections Framework . . . . .	47
11	<code>java.util</code> — Part 2: Utility Classes . . . . .	48
12	<code>java.io</code> — Input/Output . . . . .	49
13	NIO . . . . .	50
14	Networking . . . . .	51
15	Event Handling . . . . .	52
16	AWT: Working with Windows, Graphics, and Text . . . . .	53
17	Using AWT Controls, Layout Managers, and Menus . . . . .	54
18	Images . . . . .	55
19	The Concurrency Utilities . . . . .	56
20	The Stream API . . . . .	57
21	Regular Expressions and Other Packages . . . . .	58
22	Introducing Swing . . . . .	59
A	The Makefile . . . . .	60
B	Code Chunk Summaries . . . . .	62
	List of Tables . . . . .	67
	List of General Forms . . . . .	68
	Bibliography . . . . .	69
	Index . . . . .	70
	Function Index . . . . .	75

# Table of Contents

## The Java Language

<b>1</b>	<b>Java SE 9 Introduction .....</b>	<b>3</b>
<b>2</b>	<b>Classes .....</b>	<b>4</b>
2.1	Class Fundamentals .....	4
2.1.1	General Form of a Class .....	4
2.2	Declaring Objects .....	5
2.3	Methods .....	5
2.4	Constructors .....	6
2.5	The <code>this</code> Keyword .....	6
2.5.1	Instance Variable Hiding .....	6
2.6	A Stack Class .....	6
2.6.1	Stack Instance Variables .....	7
2.6.2	Stack Constructor Subsection .....	7
2.6.3	Stack Instance Methods Subsection .....	7
2.6.3.1	Stack Push and Pop Subsubsection .....	8
2.6.4	Stack TestStack Subsection .....	8
<b>3</b>	<b>Methods and Classes .....</b>	<b>10</b>
3.1	Overloading Methods .....	10
3.1.1	Overloading Constructors .....	10
3.2	Objects as Parameters .....	11
3.3	Argument Passing .....	11
3.4	Returning Objects .....	11
3.5	Recursion .....	11
3.6	Access Control .....	12
3.6.1	An Improved <code>Stack</code> Class .....	13
3.7	<code>static</code> Keyword .....	14
3.8	<code>final</code> Keyword .....	14
3.9	Arrays Revisited .....	14
3.10	Nested and Inner Classes .....	14
3.11	The <code>String</code> Class .....	14
3.12	Using Command-Line Arguments .....	14
3.13	Varargs: Variable-Length Arguments .....	14
<b>4</b>	<b>Packages .....</b>	<b>15</b>
4.1	Introduction to Packages .....	15
4.2	Defining Packages .....	15
4.3	Finding Packages and <code>CLASSPATH</code> .....	16
4.4	Packages and Member Access .....	16
4.5	Importing Packages .....	17

<b>5</b>	<b>Interfaces</b>	<b>19</b>
5.1	Defining Interfaces	19
5.2	Implementing Interfaces	20
5.3	Accessing Implementations Through Interface References	20
5.4	Partial Implementations	20
5.5	Nested Interfaces	20
5.6	Applying Interfaces	21
5.7	Variables in Interfaces	21
5.8	Interfaces Can Be Extended	21
5.9	Default Interface Methods	21
5.10	Use Static Methods in an Interface	22
5.11	Private Interface Methods	22
<b>6</b>	<b>Generics</b>	<b>23</b>
6.1	Motivation for Generics	23
6.2	What Are Generics	24
6.3	A Simple Generics Example	24
6.3.1	Class Gen<T>	24
6.3.2	Class GenDemo	26
6.3.2.1	Implementation of Class GenDemo with Type Integer	27
6.3.2.2	Implementation of Class GenDemo with Type String	28
6.4	Notes About Generics	29
6.4.1	Generics Work Only with Reference Types	29
6.4.2	Generic Types Differ Based on their Type Arguments	29
6.4.3	Generics and Subtyping	29
6.4.4	How Generics Improve Type Safety	29
6.5	A Generic Class with Two Type Parameters	29
6.5.1	Example of Code with Two Type Parameters	30
6.5.1.1	Class TwoGen	30
6.5.1.2	Class SimpGen	31
6.6	The General Form of a Generic Class	32
6.7	Bounded Types	32
6.8	Using Wildcard Arguments	32
6.8.1	Wildcard Motivation	32
6.8.2	Wildcard Syntax	33
6.8.3	Bounded Wildcards	34
6.9	Creating a Generic Method	34
6.9.1	Example of Generic Method	34
6.9.1.1	Method isIn()	35
6.9.1.2	GenMethDemo Main	35
6.10	Generic Constructors	36
<b>7</b>	<b>Enumerations</b>	<b>37</b>
7.1	Enumeration Basics	37
7.2	Enum Methods <code>values()</code> and <code>valueOf()</code>	38
7.3	Java Enumerations are Class Types	38
7.4	Enumerations Inherit <code>Enum</code>	39

## The Java Standard Library

<b>8</b>	<b>String Handling .....</b>	<b>43</b>
<b>9</b>	<b>java.lang.....</b>	<b>44</b>
9.1	Primitive Type Wrappers.....	45
9.1.1	Number .....	45
9.1.2	Double and Float .....	45
9.1.3	isInfinite() and isNaN() .....	45
9.1.4	Byte, Short, Integer, Long .....	46
9.1.5	Converting Numbers to and from String.....	46
<b>10</b>	<b>java.util — Part 1: The Collections Framework.....</b>	<b>47</b>
<b>11</b>	<b>java.util — Part 2: Utility Classes.....</b>	<b>48</b>
<b>12</b>	<b>java.io — Input/Output.....</b>	<b>49</b>
<b>13</b>	<b>NIO.....</b>	<b>50</b>
<b>14</b>	<b>Networking.....</b>	<b>51</b>
<b>15</b>	<b>Event Handling .....</b>	<b>52</b>
<b>16</b>	<b>AWT: Working with Windows, Graphics, and Text.....</b>	<b>53</b>
<b>17</b>	<b>Using AWT Controls, Layout Managers, and Menus.....</b>	<b>54</b>
<b>18</b>	<b>Images .....</b>	<b>55</b>
<b>19</b>	<b>The Concurrency Utilities .....</b>	<b>56</b>
<b>20</b>	<b>The Stream API.....</b>	<b>57</b>
<b>21</b>	<b>Regular Expressions and Other Packages...</b>	<b>58</b>
<b>22</b>	<b>Introducing Swing.....</b>	<b>59</b>

<b>Appendix A</b>	<b>The Makefile</b>	<b>60</b>
A.1	Makefile Constants	60
A.2	Makefile Default Targets	60
A.3	Makefile Tangle Weave Targets	60
A.4	Makefile Clean Targets	61
<b>Appendix B</b>	<b>Code Chunk Summaries</b>	<b>62</b>
B.1	Source File Definitions	62
B.2	Code Chunk Definitions	62
B.3	Code Chunk References	64
<b>List of Tables</b>		<b>67</b>
<b>List of General Forms</b>		<b>68</b>
<b>Bibliography</b>		<b>69</b>
<b>Index</b>		<b>70</b>
<b>Function Index</b>		<b>75</b>

# 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 it. 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 parentheses 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 reference 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
<i>&lt;Stack Constructor&gt;</i>	See “ <a href="#">Stack Constructor Subsection</a> ”, page 7.
<i>&lt;Stack Instance Methods&gt;</i>	See “ <a href="#">Stack Instance Methods Subsection</a> ”, page 7.
<i>&lt;Stack Instance Variables&gt;</i>	See “ <a href="#">Stack Instance Variables</a> ”, 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 href="#">A Stack Class</a> ”, page 6.
{StackImproved.java}	See “ <a href="#">An Improved Stack Class</a> ”, 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 href="#">A Stack Class</a> ”, page 6.
{StackImproved.java}	See “ <a href="#">An Improved Stack Class</a> ”, 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 guide 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, their return type alone is insufficient 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. Providing 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 variables 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) {
    ...
}
```

### 3.6.1 An Improved Stack Class

#### StackImproved.java

Compare this code with that of [Section 2.6 “A Stack Class”, page 6](#).<sup>1</sup>

```
{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 “ <a href="#">Stack Constructor Subsection</a> ”, page 7.
<Stack Instance Methods>	See “ <a href="#">Stack Instance Methods Subsection</a> ”, page 7.
<Stack Private Instance Variables>	See “ <a href="#">An Improved Stack Class</a> ”, 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];
private int tos;
```

This chunk is called by {StackImproved.java}; see its first definition at “[An Improved Stack Class](#)”, page 13.

<sup>1</sup> Notice how all of the prior code except what is changed can easily be reused using TexiWebJr's modular system.

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

### 3.8 final Keyword

### 3.9 Arrays Revisited

### 3.10 Nested and Inner Classes

### 3.11 The String Class

### 3.12 Using Command-Line Arguments

### 3.13 Varargs: Variable-Length Arguments

## 4 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.

### 4.1 Introduction to Packages

Java provides a mechanism for partitioning the class namespace into manageable 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.

### 4.2 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 4.1: 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 hierarchy of packages. To do so, separate each package name from the one above it by use of a period. The general form of a multileveled package statement is:

```
package pkg1[.pkg2[.pkg3]]
```

GeneralForm 4.2: 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.

### 4.3 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 current 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
```

### 4.4 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 containers 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 variety 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 4.1: Package Access Table — Shows all combinations of the access control modifiers

## 4.5 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 4.3: 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;
import 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.



## 5 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 interface, 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 allows 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.

### 5.1 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 5.1: 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 must 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**.

## 5.2 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 classname [extends superclass] [implements interface [,interface. . .] ] {  
    class-body  
}
```

GeneralForm 5.2: 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.

## 5.3 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.

## 5.4 Partial Implementations

If a class includes an interface but does not implement the methods 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.

## 5.5 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 nested 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.

## 5.6 Applying Interfaces

See detailed example . . .

## 5.7 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.

## 5.8 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.

## 5.9 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, containing 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 will be 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 Not 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.

## 5.10 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 5.3: 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.

## 5.11 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 interface 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.

## 6 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.

### 6.1 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.

## 6.2 What Are Generics

The term *generics* means *parameterized types*. Parameterized types are important because they enable you to create classes, interfaces, and methods in which 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 operated upon. With generics, all casts are automatic and implicit.

## 6.3 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 chunk definition points.

Chunk name	First definition point
<Class Gen>	See “Class <code>Gen&lt;T&gt;</code> ”, page 25.
<Class GenDemo>	See “Class <code>GenDemo</code> ”, page 26.

### 6.3.1 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 24.

The following table lists called chunk definition points.

Chunk name	First definition point
<code>&lt;Constructor taking parameter of Type T&gt;</code>	See “Class <code>Gen&lt;T&gt;</code> ”, page 25.
<code>&lt;Instance Variable ob of Type T&gt;</code>	See “Class <code>Gen&lt;T&gt;</code> ”, page 25.
<code>&lt;Method returning object of type T&gt;</code>	See “Class <code>Gen&lt;T&gt;</code> ”, page 26.
<code>&lt;Method showing type of T&gt;</code>	See “Class <code>Gen&lt;T&gt;</code> ”, page 26.

## 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 25.

## 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 25.

## 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 with 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<sub>i</sub>*”, page 25.

The method `showType()` displays the type of ‘T’ by calling `getName()` on the `Class` object returned 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<sub>i</sub>*”, page 25.

### 6.3.2 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 24.



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 27.
<Create a Gen object for Strings>	See “Implementation of Class GenDemo with Type String”, page 28.

### 6.3.2.1 Implementation of Class GenDemo with Type Integer

<Create a Gen object for Integers>  $\equiv$

```

    <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 26.

The following table lists called chunk definition points.

Chunk name	First definition point
<Get Value>	See “Implementation of Class GenDemo with Type Integer”, page 28.
<Integer Type Parameter>	See “Implementation of Class GenDemo with Type Integer”, page 27.
<Reference to Integer Instance>	See “Implementation of Class GenDemo with Type Integer”, page 28.
<Show Type>	See “Implementation of Class GenDemo with Type Integer”, page 28.

### Integer Type Declaration

A reference to an Integer is declared in `i0b`. 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>  $\equiv$

```

    Gen<Integer> i0b;

```

This chunk is called by <Create a Gen object for Integers>; see its first definition at “Implementation of Class GenDemo with Type Integer”, page 27.

### Reference Assignment

The next line assigns to `i0b` 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 `i0b` 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 27](#).

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 27](#).

## Showing the Reference’s Value

The program now obtains the value of `ob` by assigning `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 27](#).

### 6.3.2.2 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 26](#).

## 6.4 Notes About Generics

### 6.4.1 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.

### 6.4.2 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.

### 6.4.3 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>`.

### 6.4.4 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.

## 6.5 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.

### 6.5.1 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 31.
<Class TwoGen>	See “Class TwoGen”, page 30.

#### 6.5.1.1 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 30.

The following table lists called chunk definition points.

Chunk name	First definition point
<Class Declaration>	See “Class TwoGen”, page 30.
<Constructor of Two Parameters>	See “Class TwoGen”, page 30.
<Instance Methods Show and Get>	See “Class TwoGen”, page 31.
<Two Instance Variables Declarations>	See “Class TwoGen”, page 30.

### 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 30.

### 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 30.

### Constructor

<Constructor of Two Parameters> ≡

```
TwoGen(T o1, V o2) {
```

```
ob1 = o1;
```

```

    ob2 = o2;
}

```

This chunk is called by `<Class TwoGen>`; see its first definition at [“Class TwoGen”, page 30](#).

## 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 30](#).

### 6.5.1.2 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 = tgObj.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 30](#).

## 6.6 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 6.1: General Form for Declaring and Creating a Reference to a Generic Class

## 6.7 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, you 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 connect them.

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

Any type argument passed to ‘*T*’ must be a subclass of *MyClass* and implement *MyInterface*.

## 6.8 Using Wildcard Arguments

### 6.8.1 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.

## 6.8.2 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.

### 6.8.3 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 6.2: 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 6.3: General Form of Lower Bounded Wildcard Syntax

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

## 6.9 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 6.4: General Form for Declaring a Generic Method

#### 6.9.1 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 35.
<Static Method isIn>	See “Method isIn()”, page 35.

### 6.9.1.1 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 35.

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 arguments are specified:

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

### 6.9.1.2 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 35](#).

## 6.10 Generic Constructors

It is possible for constructors to be generic, even if their class is not (see “[Class GenT<sub>i</sub>](#)”, [page 25](#)). The syntax is the same (type parameters come first).

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

## 7 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

### 7.1 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 enumeration 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'`.

## 7.2 Enum Methods `values()` and `valueOf()`

All enumerations inherit two methods:

```
public static enum-type[] values ()
```

 [Method on Enum]

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

```
public static enum-type valueOf (String str)
```

 [Method on Enum]

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.

## 7.3 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.

## 7.4 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**





## 8 String Handling

## 9 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`

## 9.1 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*.

### 9.1.1 Number

### 9.1.2 Double and Float

### 9.1.3 `isInfinite()` and `isNaN()`

**9.1.4 Byte, Short, Integer, Long****9.1.5 Converting Numbers to and from String**

## 10 java.util — Part 1: The Collections Framework

## 11 `java.util` — Part 2: Utility Classes

## 12 java.io — Input/Output

## **13 NIO**



## 14 Networking

## **15 Event Handling**

## 16 AWT: Working with Windows, Graphics, and Text

## **17 Using AWT Controls, Layout Managers, and Menus**

## 18 Images

## 19 The Concurrency Utilities

## 20 The Stream API

## 21 Regular Expressions and Other Packages



## 22 Introducinv Swing

## Appendix A The Makefile

`{Makefile}`  $\equiv$

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

The following table lists called chunk definition points.

Chunk name	First definition point
<code>&lt;Makefile CLEAN&gt;</code>	See “ <a href="#">Makefile Clean Targets</a> ”, page 61.
<code>&lt;Makefile CONSTANTS&gt;</code>	See “ <a href="#">Makefile Constants</a> ”, page 60.
<code>&lt;Makefile DEFAULTS&gt;</code>	See “ <a href="#">Makefile Default Targets</a> ”, page 60.
<code>&lt;Makefile TANGLE WEAVE&gt;</code>	See “ <a href="#">Makefile Tangle Weave Targets</a> ”, page 60.

### A.1 Makefile Constants

`<Makefile CONSTANTS>`  $\equiv$

```
FILENAME := JavaSE9
```

This chunk is called by `{Makefile}`; see its first definition at “[The Makefile](#)”, page 60.

### A.2 Makefile Default Targets

`<Makefile DEFAULTS>`  $\equiv$

```
.PHONY: all
all: tangle weave
```

This chunk is called by `{Makefile}`; see its first definition at “[The Makefile](#)”, page 60.

### A.3 Makefile Tangle Weave Targets

`<Makefile TANGLE WEAVE>`  $\equiv$

```
.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 60.

## A.4 Makefile Clean Targets

*<Makefile CLEAN>*  $\equiv$

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

This chunk is called by {**Makefile**}; see its first definition at [“The Makefile”, page 60](#).

## 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

`{GenMethDemo.java}`

This chunk is defined in “[Example of Generic Method](#)”, page 35.

`{Makefile}`

This chunk is defined in “[The Makefile](#)”, page 60.

`{SimpleGenerics.java}`

This chunk is defined in “[A Simple Generics Example](#)”, page 24.

`{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 30.

### B.2 Code Chunk Definitions

*<Class Declaration>*

This chunk is defined in “[Class TwoGen](#)”, page 30.

*<Class Gen>*

This chunk is defined in “[Class Gen\*T<sub>i</sub>\*](#)”, page 25.

*<Class GenDemo>*

This chunk is defined in “[Class GenDemo](#)”, page 26.

*<Class SimpGen>*

This chunk is defined in “[Class SimpGen](#)”, page 31.

*<Class TwoGen>*

This chunk is defined in “[Class TwoGen](#)”, page 30.

*<Constructor of Two Parameters>*

This chunk is defined in “[Class TwoGen](#)”, page 30.

*<Constructor taking parameter of Type T>*

This chunk is defined in “[Class Gen\*T<sub>i</sub>\*](#)”, page 25.

*<Create a Gen object for Integers>*

This chunk is defined in “[Implementation of Class GenDemo with Type Integer](#)”, page 27.

## &lt;Create a Gen object for Strings&gt;

This chunk is defined in “Implementation of Class GenDemo with Type String”, page 28.

## &lt;GenMethDemo Main&gt;

This chunk is defined in “GenMethDemo Main”, page 35.

## &lt;Get Value&gt;

This chunk is defined in “Implementation of Class GenDemo with Type Integer”, page 28.

## &lt;Instance Methods Show and Get&gt;

This chunk is defined in “Class TwoGen”, page 31.

## &lt;Instance Variable ob of Type T&gt;

This chunk is defined in “Class Gen<sub>i</sub>T<sub>i</sub>”, page 25.

## &lt;Integer Type Parameter&gt;

This chunk is defined in “Implementation of Class GenDemo with Type Integer”, page 27.

## &lt;Makefile CLEAN&gt;

This chunk is defined in “Makefile Clean Targets”, page 61.

## &lt;Makefile CONSTANTS&gt;

This chunk is defined in “Makefile Constants”, page 60.

## &lt;Makefile DEFAULTS&gt;

This chunk is defined in “Makefile Default Targets”, page 60.

## &lt;Makefile TANGLE WEAVE&gt;

This chunk is defined in “Makefile Tangle Weave Targets”, page 60.

## &lt;Method returning object of type T&gt;

This chunk is defined in “Class Gen<sub>i</sub>T<sub>i</sub>”, page 26.

## &lt;Method showing type of T&gt;

This chunk is defined in “Class Gen<sub>i</sub>T<sub>i</sub>”, page 26.

## &lt;Reference to Integer Instance&gt;

This chunk is defined in “Implementation of Class GenDemo with Type Integer”, page 28.

## &lt;Show Type&gt;

This chunk is defined in “Implementation of Class GenDemo with Type Integer”, page 28.

## &lt;Stack Constructor&gt;

This chunk is defined in “Stack Constructor Subsection”, page 7.

## &lt;Stack Instance Methods&gt;

This chunk is defined in “Stack Instance Methods Subsection”, page 7.

## &lt;Stack Instance Variables&gt;

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 35.

*<TestStack Main Method>*

This chunk is defined in “Stack TestStack Subsection”, page 8.

*<Two Instance Variables Declarations>*

This chunk is defined in “Class TwoGen”, page 30.

## B.3 Code Chunk References

*<Class Declaration>*

This chunk is called by *<Class TwoGen>*; see its first definition at “Class TwoGen”, page 30.

*<Class Gen>*

This chunk is called by {SimpleGenerics.java}; see its first definition at “A Simple Generics Example”, page 24.

*<Class GenDemo>*

This chunk is called by {SimpleGenerics.java}; see its first definition at “A Simple Generics Example”, page 24.

*<Class SimpGen>*

This chunk is called by {TwoTypeParameters.java}; see its first definition at “Example of Code with Two Type Parameters”, page 30.

*<Class TwoGen>*

This chunk is called by {TwoTypeParameters.java}; see its first definition at “Example of Code with Two Type Parameters”, page 30.

*<Constructor of Two Parameters>*

This chunk is called by *<Class TwoGen>*; see its first definition at “Class TwoGen”, page 30.

*<Constructor taking parameter of Type T>*

This chunk is called by *<Class Gen>*; see its first definition at “Class Gen;T<sub>i</sub>”, page 25.

*<Create a Gen object for Integers>*

This chunk is called by *<Class GenDemo>*; see its first definition at “Class GenDemo”, page 26.

*<Create a Gen object for Strings>*

This chunk is called by *<Class GenDemo>*; see its first definition at “Class GenDemo”, page 26.

## &lt;GenMethDemo Main&gt;

This chunk is called by {GenMethDemo.java}; see its first definition at “Example of Generic Method”, page 35.

## &lt;Get Value&gt;

This chunk is called by <Create a Gen object for Integers>; see its first definition at “Implementation of Class GenDemo with Type Integer”, page 27.

## &lt;Instance Methods Show and Get&gt;

This chunk is called by <Class TwoGen>; see its first definition at “Class TwoGen”, page 30.

## &lt;Instance Variable ob of Type T&gt;

This chunk is called by <Class Gen>; see its first definition at “Class Gen;T<sub>i</sub>”, page 25.

## &lt;Integer Type Parameter&gt;

This chunk is called by <Create a Gen object for Integers>; see its first definition at “Implementation of Class GenDemo with Type Integer”, page 27.

## &lt;Makefile CLEAN&gt;

This chunk is called by {Makefile}; see its first definition at “The Makefile”, page 60.

## &lt;Makefile CONSTANTS&gt;

This chunk is called by {Makefile}; see its first definition at “The Makefile”, page 60.

## &lt;Makefile DEFAULTS&gt;

This chunk is called by {Makefile}; see its first definition at “The Makefile”, page 60.

## &lt;Makefile TANGLE WEAVE&gt;

This chunk is called by {Makefile}; see its first definition at “The Makefile”, page 60.

## &lt;Method returning object of type T&gt;

This chunk is called by <Class Gen>; see its first definition at “Class Gen;T<sub>i</sub>”, page 25.

## &lt;Method showing type of T&gt;

This chunk is called by <Class Gen>; see its first definition at “Class Gen;T<sub>i</sub>”, page 25.

## &lt;Reference to Integer Instance&gt;

This chunk is called by <Create a Gen object for Integers>; see its first definition at “Implementation of Class GenDemo with Type Integer”, page 27.

## &lt;Show Type&gt;

This chunk is called by <Create a Gen object for Integers>; see its first definition at “Implementation of Class GenDemo with Type Integer”, page 27.

## &lt;Stack Constructor&gt;

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 35.

<TestStack Main Method>

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

<Two Instance Variables Declarations>

This chunk is called by <Class TwoGen>; see its first definition at “Class TwoGen”, page 30.



List of Tables

Table 4.1: Package Access Table ..... 17

## List of General Forms

GeneralForm 2.1: Class Declaration — General Form .....	4
GeneralForm 2.2: Method Declaration — General Form .....	5
GeneralForm 4.1: Package Statement — General Form .....	15
GeneralForm 4.2: Package Statement — Multilevel Form .....	15
GeneralForm 4.3: Import Statement — General Form .....	17
GeneralForm 5.1: Interface Definition — Simplified General Form .....	19
GeneralForm 5.2: Class Implementing Interface — General Form .....	20
GeneralForm 5.3: Interface Static Method, Calling .....	22
GeneralForm 6.1: General Form Generic Class .....	32
GeneralForm 6.2: Upper Bounded Wildcard .....	34
GeneralForm 6.3: Lower Bounded Wildcard .....	34
GeneralForm 6.4: Generic Method Declaration .....	34

## Bibliography

# Index

<	
<Class Declaration>, definition .....	30
<Class Declaration>, use .....	30
<Class Gen>, definition .....	25
<Class Gen>, use .....	24
<Class GenDemo>, definition .....	26
<Class GenDemo>, use .....	24
<Class SimpGen>, definition .....	31
<Class SimpGen>, use .....	30
<Class TwoGen>, definition .....	30
<Class TwoGen>, use .....	30
<Constructor of Two Parameters>, definition ..	30
<Constructor of Two Parameters>, use .....	30
<Constructor taking parameter of Type T>, definition .....	25
<Constructor taking parameter of Type T>, use .....	25
<Create a Gen object for Integers>, definition ..	27
<Create a Gen object for Integers>, use .....	26
<Create a Gen object for Strings>, definition ..	28
<Create a Gen object for Strings>, use .....	26
<GenMethDemo Main>, definition .....	35
<GenMethDemo Main>, use .....	35
<Get Value>, definition .....	28
<Get Value>, use .....	27
<Instance Methods Show and Get>, definition ..	31
<Instance Methods Show and Get>, use .....	30
<Instance Variable ob of Type T>, definition ..	25
<Instance Variable ob of Type T>, use .....	25
<Integer Type Parameter>, definition .....	27
<Integer Type Parameter>, use .....	27
<Makefile CLEAN>, definition .....	61
<Makefile CLEAN>, use .....	60
<Makefile CONSTANTS>, definition .....	60
<Makefile CONSTANTS>, use .....	60
<Makefile DEFAULTS>, definition .....	60
<Makefile DEFAULTS>, use .....	60
<Makefile TANGLE WEAVE>, definition .....	60
<Makefile TANGLE WEAVE>, use .....	60
<Method returning object of type T>, definition .....	26
<Method returning object of type T>, use .....	25
<Method showing type of T>, definition .....	26
<Method showing type of T>, use .....	25
<Reference to Integer Instance>, definition .....	28
<Reference to Integer Instance>, use .....	27
<Show Type>, definition .....	28
<Show Type>, use .....	27
<Stack Constructor>, definition .....	7
<Stack Constructor>, use .....	6, 13
<Stack Instance Methods>, definition .....	7
<Stack Instance Methods>, use .....	6, 13
<Stack Instance Variables>, definition .....	7
<Stack Instance Variables>, use .....	6
<Stack Pop>, definition .....	8
<Stack Pop>, use .....	7
<Stack Private Instance Variables>, definition ..	13
<Stack Private Instance Variables>, use .....	13
<Stack Push>, definition .....	8
<Stack Push>, use .....	7
<Static Method isIn>, definition .....	35
<Static Method isIn>, use .....	35
<TestStack Main Method>, definition .....	8
<TestStack Main Method>, use .....	7
<Two Instance Variables Declarations>, definition .....	30
<Two Instance Variables Declarations>, use ....	30
=	
== .....	37
{	
{GenMethDemo.java}, definition .....	35
{Makefile}, definition .....	60
{SimpleGenerics.java}, definition .....	24
{Stack.java}, definition .....	6
{StackImproved.java}, definition .....	13
{TestStack.java}, definition .....	7
{TwoTypeParameters.java}, definition .....	30
A	
abstract class .....	19
abstract methods, interface .....	19, 21
abstract over types .....	23
access control table .....	17
access control, packages .....	16
access control, single class .....	12
access modifiers .....	12
access, member .....	16
accessibility .....	15
API, Stream .....	57
argument passing .....	11
arguments, command-line .....	14
arguments, varargs .....	14
Arrays .....	14
auto-boxing, generics .....	29
auto-unboxing, generics .....	29
autoboxing in generic reference .....	27
AWT .....	53
AWT Controls .....	54
AWT Layout Managers, Menus .....	54

**B**

bounded types .....	32
bounded wildcards .....	34
bounded wildcards, lower bound .....	34
bounded wildcards, upper bound .....	34

**C**

casts, eliminated in generics .....	29
casts, generics, automatic, implicit .....	24
<b>Class</b> .....	26
Class fundamentals .....	4
class name, from <code>getName()</code> .....	26
class namespace, compartmentalize .....	15
<b>Class</b> object, from <code>getClass()</code> .....	26
class <b>String</b> .....	14
class, general form .....	4
class, new data type .....	4
classed in <code>java.lang</code> .....	44
Classes .....	4
classes, nested and inner .....	14
<b>CLASSPATH</b> <code>-classpath</code> .....	16
Collections Framework .....	23
collections, generics .....	23
collisions, prevention .....	15
command-line arguments .....	14
compartmentalized .....	15
compile time .....	19
compile-time type check .....	23
Concurrency Utilities .....	56
Constants .....	60
constructor .....	5
Constructors .....	6
constructors, overloading .....	10
containers, packages as .....	15
creating generic method .....	34

**D**

data type, enumeration .....	37
default access level .....	12
default method, interface, motivation .....	21
default methods, interface .....	21
default package .....	15
difference between class and interface .....	21
dispatch through an interface .....	20
dot operator .....	4
dynamic allocation, run time .....	5
dynamic dispatch, interface method look-ups .....	20
dynamic method resolution .....	19

**E**

encapsulation, access control .....	12
enum <code>valueOf()</code> .....	38
enum <code>values()</code> .....	38
enum variable, declare .....	37
enumeration capabilities .....	37
enumeration comparison .....	37
enumeration constants .....	37, 38
enumeration constructor .....	38
enumeration instance variables .....	38
enumeration methods .....	38
enumeration object .....	37
enumeration restrictions .....	39
enumeration variable .....	37
Enumeration, basics .....	37
Enumerations .....	37
enumerations as class types .....	38
enumerations inherit <b>Enum</b> .....	39
enums, printing .....	38
equality, enum types .....	37
erasure .....	26
Event Handling .....	52
example generic method .....	34
example, generics .....	24
exposure of code .....	15
extending interfaces .....	21
<b>extends</b> clause .....	32
<b>extends</b> , with interfaces .....	21

**F**

<b>final</b> Keyword .....	14
<b>final</b> , traditional enums .....	37
finding packages .....	16
fully qualified name .....	17

**G**

generic class .....	24
generic class, general form .....	32
generic class, method .....	24
generic class, two type parameters .....	29
generic code, demonstrating an implementation .....	26
generic constructors .....	36
generic interface .....	23
generic method, creating .....	34
generic method, example .....	34
generic method, static .....	35
generic methods, including type arguments .....	35
generic reference assignment to <b>Integer</b> .....	27
generic reference to <b>Integer</b> .....	27
generic reference, creating .....	27
generic type argument, reference type .....	29
generic type checking .....	27
generic types differ, type arguments .....	29
Generics (chapter) .....	23
generics eliminate casts .....	29

generics ensure type safety .....	29
generics example .....	24
generics improve type safety .....	29
generics, bounded types .....	32
generics, casts .....	24
generics, compile-time error, mismatched types .....	27
generics, generic constructors .....	36
generics, interface as bound .....	32
generics, introduction .....	23
generics, motivation .....	23
generics, motivation, readability and robustness .....	23
generics, only reference types .....	29
generics, subtyping .....	29
generics, two type arguments .....	30
generics, two type parameters, declaration .....	30
generics, type safety benefit .....	27
generics, what they are .....	24
generics, wildcard arguments .....	32
<code>getClass()</code> , defined in <code>Object</code> .....	26
<code>getName()</code> , defined in <code>Class</code> .....	26
Graphics .....	53

## H

hiding, instance variables .....	6
hierarchical structure, packages .....	15
hierarchy of packages .....	15

## I

Images .....	55
<code>implements</code> clause .....	20
import is optional .....	17
import packages .....	15
import statement, general form and example ...	17
imported packages must be public .....	17
importing packages .....	17
index interface, default methods .....	21
inheriting interfaces .....	21
inner classes .....	14
instance variables .....	4
instance, class .....	4
intefaces, applying .....	21
interface as bound, generics .....	32
interface default access, no modified .....	19
interface definition, simplified general form ....	19
interface method defintion, declared <code>public</code> ....	20
interface methods, abstract methods .....	19
interface methods, private .....	22
interface public access .....	19
interface references, accessing implementations .....	20
interface variable declarations .....	20
interface, implement .....	19
interface, partial implementation .....	20
interface, static method .....	22

interface, traditional form .....	21
Interfaces (chapter) .....	19
interfaces in <code>java.lang</code> .....	45
interfaces, defining .....	19
interfaces, extending .....	21
interfaces, final variables in .....	21
interfaces, implementing .....	20
interfaces, inheriting .....	21
interfaces, introduction .....	19
interfaces, key aspect, no state .....	21
interfaces, key feature, reference look-ups .....	20
interfaces, nested .....	20
interfaces, shared constants .....	21
introduction to Java SE 9 .....	3
Introduction to Packages (section) .....	15
iteration, iterative .....	11

## J

J2SE 5.0 .....	23
Java SE 9 introduction .....	3
<code>java.io</code> .....	49
<code>java.lang</code> .....	17, 44
<code>java.util</code> Collections Framework .....	47
<code>java.util</code> Utility Classes .....	48
JDK 5 .....	37
JDK 8, default method in interface .....	21
JDK 8, static interface method .....	22
JDK 9, package part of module .....	16
JDK 9, private interface method .....	22

## K

keyword <code>final</code> .....	14
keyword <code>interface</code> .....	19
keyword <code>static</code> .....	14
keyword, <code>enum</code> .....	37

## L

lower bounded wildcard .....	34
------------------------------	----

## M

'main()' method, class .....	4
Makefile Weave .....	60
Makefile Clean targets .....	61
Makefile defaults .....	60
Makefile Tangle .....	60
Makefile, The (appendix) .....	60
member access .....	16
member interfaces .....	20
members .....	4
method signatures compatible .....	19
method, static, interface .....	22
Methods .....	5
methods .....	4
Methods and Classes .....	10
methods, enumeration .....	38
methods, overloading .....	10
module path .....	16
modules, packages .....	16

## N

name, method .....	5
naming mechanism .....	15
nested classes .....	14
nested interfaces .....	20
Networking .....	51
<b>new</b> operator .....	5
NIO .....	50

## O

<b>Object</b> .....	26
object references, interfaces .....	20
<b>Object</b> type .....	24
object, class .....	4
objects as parameters .....	11
objects, declaring .....	5
objects, dynamical allocation .....	11
objects, references to .....	11
objects, returning from methods .....	11
<i>one interface, multiple methods</i> .....	10
overload, overloaded .....	10
overloading constructors .....	10
overloading methods .....	10
overloading, automatic type conversion .....	10

## P

<b>package</b> command .....	15
package namespace .....	15
package renaming .....	16
package statement .....	15
package statement, example .....	15
package statement, general form .....	15
package statement, multilevel form .....	15
Packages (chapter) .....	15
packages hierarchy .....	15
packages stored in file system .....	15
packages, access control .....	16
Packages, Defining (section) .....	15
packages, finding, example .....	16
packages, how stored .....	15
packages, import .....	15
packages, importing .....	17
packages, purposes, prevent collisions .....	15
parameter list, method .....	5
parameter, generic class .....	24
parameterized type .....	24
parameterized types .....	24
parameters, as objects .....	11
partitioning mechanism .....	15
polymorphism, one interface	
multiple methods .....	19
polymorphism, overloading of methods .....	10
preexisting code, default method, interface .....	21
Primitive Wrappers .....	45
private access modifier .....	12
protected access modifier .....	12
public access modifier .....	12

## R

recursion, recursive .....	11
Regular Expressions .....	58
run time, dynamic allocation .....	5
run-time .....	19
run-time system, finding packages .....	16

## S

self-typed constants .....	37
Stack Class .....	6
<b>Stack</b> class, improved .....	13
stack exhaustion, recursion .....	11
stack overrun, recursion .....	11
standard Java classes, imported implicitly .....	17
static environment .....	19
static generic method .....	35
<b>static</b> Keyword .....	14
static method, interface .....	22
Stream API .....	57
<b>String</b> Class .....	14
String Handling .....	43
Strings .....	43

Swing.....	59
<code>switch</code> statement, enum types .....	37

## T

template, class .....	4
Text .....	53
<code>this</code> Keyword .....	6
type abstraction, generics .....	23
type argument, passed to type parameter .....	27
type correctness .....	23
type erasure .....	26
type parameter .....	23
type parameter, generic class .....	24
type safety, generics .....	24
type wrappers .....	45
type wrappers, generics .....	29
type, method .....	5

## U

upper bound.....	32
upper bound wildcard argument.....	34
upper bounded wildcard .....	34

## V

varargs .....	14
variable, enum type .....	37
variable-length arguments .....	14
visibility mechanism .....	15

## W

wildcard arguments, generics .....	32
wildcard syntax .....	33
wildcards, bounded .....	34
wildcards, motivation .....	32
Windows .....	53
Wrappers, Primitives .....	45



# Function Index

<b>=</b>		<b>O</b>	
<code>==</code> on Enum.....	39	<code>ordinal</code> on Enum.....	39
<b>C</b>		<b>V</b>	
<code>compareTo</code> on Enum.....	39	<code>valueOf</code> on Enum.....	38
<b>E</b>		<code>values</code> on Enum.....	38
<code>equals</code> on Enum.....	39		