Outline Covering Java SE 9

SEPTEMBER, 2018

LOLH

Short Contents

Table of Contents

The Java Language

The Java Language

1 Java SE 9 Introduction

2 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.

2.1 Introduction to Packages

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

2.2 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 2.1: Package Statement — General Form

pkg is the name of the package. For example:

```
package mypackage;
```

File System Directories

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

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

Hierarchy of Packages

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

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

GeneralForm 2.2: 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.

2.3 Finding Packages and CLASSPATH

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

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

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

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

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

Example Finding a Package

Consider the following package specification:

```
package mypack;
```

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

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

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

then the class path to `mypack` is

```
/MyPrograms/Java
```

2.4 Packages and Member Access

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

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

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

The three access modifiers

- private
- public
- protected

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

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

Table 2.1: Package Access Table — Shows all combinations of the access control modifiers

2.5 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 2.3: Import Statement — General Form

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

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

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

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

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

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

3 Interfaces

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

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

3.1 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 3.1: Interface Definition — Simplified General Form

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

Variable Declarations inside Interfaces

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

3.2 Implementing Interfaces

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

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

GeneralForm 3.2: 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.

3.3 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.

3.4 Partial Implementations

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

3.5 Nested Interfaces

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

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

3.6 Applying Interfaces

See detailed example . . .

3.7 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.

3.8 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.

3.9 Default Interface Methods

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

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

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

Interfaces Do No Maintain State and Cannot Be Created

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

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

3.10 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 3.3: 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.

3.11 Private Interface Methods

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

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

4 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.

4.1 Motivation for Generics

Code Fragment Without Generics

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

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

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

Code Fragment with Generics

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

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

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

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

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

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

4.2 What Are Generics

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

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

4.3 A Simple Generics Example

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

{`SimpleGenerics.java`} ≡

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

The following table lists called chunk definition points.

Chunk name	**First definition point**
<*Class Gen*>	See "Class Gen¡T¿", page 14.
<*Class GenDemo*>	See "Class GenDemo", page 15.

4.3.1 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 13.

The following table lists called chunk definition points.

Chunk name	First definition point
<*Constructor taking parameter of Type T*>	See "Class Gen¡T¿", page 14.
<*Instance Variable ob of Type T*>	See "Class Gen¡T¿", page 14.
<*Method returning object of type T*>	See "Class Gen¡T¿", page 15.
<*Method showing type of T*>	See "Class Gen¡T¿", page 15.

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 14.

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 14.

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

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

<*Method returning object of type T*> ≡

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

This chunk is called by <*Class Gen*>; see its first definition at "Class Gen¡T¿", page 14.

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

<*Method showing type of T*> ≡

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

This chunk is called by <*Class Gen*>; see its first definition at "Class Gen¡T¿", page 14.

4.3.2 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 13.

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 16.
<*Create a Gen object for Strings*>	See "Implementation of Class GenDemo with Type String", page 17.

4.3.2.1 Implementation of Class GenDemo with Type Integer

<*Create a Gen object for Integers*> ≡

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

This chunk is called by <*Class GenDemo*>; see its first definition at "Class GenDemo", page 15.

The following table lists called chunk definition points.

Chunk name	First definition point
<*Get Value*>	See "Implementation of Class GenDemo with Type Integer", page 17.
<*Integer Type Parameter*>	See "Implementation of Class GenDemo with Type Integer", page 16.
<*Reference to Integer Instance*>	See "Implementation of Class GenDemo with Type Integer", page 17.
<*Show Type*>	See "Implementation of Class GenDemo with Type Integer", page 17.

Integer Type Declaration

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

<*Integer Type Parameter*> ≡

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

This chunk is called by <*Create a Gen object for Integers*>; see its first definition at "Implementation of Class GenDemo with Type Integer", page 16.

Reference Assignment

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

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

<*Reference to Integer Instance*> ≡

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

This chunk is called by <*Create a Gen object for Integers*>; see its first definition at "Implementation of Class GenDemo with Type Integer", page 16.

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 16.

Showing the Reference's Value

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

<*Get Value*> ≡

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

This chunk is called by <*Create a Gen object for Integers*>; see its first definition at "Implementation of Class GenDemo with Type Integer", page 16.

4.3.2.2 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 15.

4.4 Notes About Generics

4.4.1 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.

4.4.2 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.

4.4.3 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>`.

4.4.4 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.

4.5 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.

4.5.1 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 20.
<*Class TwoGen*>	See "Class TwoGen", page 19.

4.5.1.1 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 19.

The following table lists called chunk definition points.

Chunk name	First definition point
<*Class Declaration*>	See "Class TwoGen", page 19.
<*Constructor of Two Parameters*>	See "Class TwoGen", page 19.
<*Instance Methods Show and Get*>	See "Class TwoGen", page 20.
<*Two Instance Variables Declarations*>	See "Class TwoGen", page 19.

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 19.

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 19.

Constructor

<*Constructor of Two Parameters*> ≡

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

```
    ob2 = o2;
  }
```

This chunk is called by <*Class TwoGen*>; see its first definition at "Class TwoGen", page 19.

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 19.

4.5.1.2 Class SimpGen

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

<*Class SimpGen*> ≡

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

    // Show the types
    tgObj.showTypes();

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

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

This chunk is called by {`TwoTypeParameters.java`}; see its first definition at "Example of Code with Two Type Parameters", page 19.

4.6 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 4.1: General Form for Declaring and Creating a Reference to a Generic Class

4.7 Bounded Types

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

<*T* extends *superclass*>

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

Interface Type as a Bound

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

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

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

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

4.8 Using Wildcard Arguments

4.8.1 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.

4.8.2 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.

4.8.3 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 4.2: 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 4.3: General Form of Lower Bounded Wildcard Syntax

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

4.9 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 4.4: General Form for Declaring a Generic Method

4.9.1 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 24.
<*Static Method isIn*>	See "Method isIn()", page 24.

4.9.1.1 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 24.

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

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

Explicitly Including Type Arguments

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

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

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

4.9.1.2 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 24.

4.10 Generic Constructors

It is possible for constructors to be generic, even if their class is not (see "Class Gen¡T¿", page 14). The syntax is the same (type parameters come first).

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

5 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

5.1 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 must be part of the expression. Enumeration constants can be compared using the '`==`' relational operator. Furthermore, an enumneration value can be used to control a `swith` statement. The `enum` prefix is not required for switch.

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

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

The Java Standard Library

6 String Handling

7 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

7.1 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*.

7.1.1 Number

7.1.2 Double and Float

7.1.3 isInfinite() and isNaN()

7.1.4 Byte, Short, Integer, Long

7.1.5 Converting Numbers to and from String

8 java.util — Part 1: The Collections Framework

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

10 `java.io` — Input/Output

11 NIO

12 Networking

13 Event Handling

14 AWT: Working with Windows, Graphics, and Text

15 Using AWT Controls, Layout Managers, and Menus

16 Images

17 The Concurrency Utilities

18 The Stream API

19 Regular Expressions and Other Packages

20 Introducinvg Swing

Appendix A The Makefile

{`Makefile`} ≡

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

The following table lists called chunk definition points.

Chunk name	First definition point
<*Makefile CLEAN*>	See "Makefile Clean Targets", page 47.
<*Makefile CONSTANTS*>	See "Makefile Constants", page 46.
<*Makefile DEFAULTS*>	See "Makefile Default Targets", page 46.
<*Makefile TANGLE WEAVE*>	See "Makefile Tangle Weave Targets", page 46.

A.1 Makefile Constants

<*Makefile CONSTANTS*> ≡

```
FILENAME := JavaSE9
```

This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 46.

A.2 Makefile Default Targets

<*Makefile DEFAULTS*> ≡

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

This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 46.

A.3 Makefile Tangle Weave Targets

<*Makefile TANGLE WEAVE*> ≡

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

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

jrweave: $(FILENAME).texi

$(FILENAME).texi: $(FILENAME).twjr
    jrweave $(FILENAME).twjr > $(FILENAME).texi
```

This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 46.

A.4 Makefile Clean Targets

<*Makefile CLEAN*> ≡

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

This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 46.

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 24.

{`Makefile`}
This chunk is defined in "The Makefile", page 46.

{`SimpleGenerics.java`}
This chunk is defined in "A Simple Generics Example", page 13.

{`TwoTypeParameters.java`}
This chunk is defined in "Example of Code with Two Type Parameters", page 19.

B.2 Code Chunk Definitions

<*Class Declaration*>
This chunk is defined in "Class TwoGen", page 19.

<*Class Gen*>
This chunk is defined in "Class Gen¡T¿", page 14.

<*Class GenDemo*>
This chunk is defined in "Class GenDemo", page 15.

<*Class SimpGen*>
This chunk is defined in "Class SimpGen", page 20.

<*Class TwoGen*>
This chunk is defined in "Class TwoGen", page 19.

<*Constructor of Two Parameters*>
This chunk is defined in "Class TwoGen", page 19.

<*Constructor taking parameter of Type T*>
This chunk is defined in "Class Gen¡T¿", page 14.

<*Create a Gen object for Integers*>
This chunk is defined in "Implementation of Class GenDemo with Type Integer", page 16.

<*Create a Gen object for Strings*>
This chunk is defined in "Implementation of Class GenDemo with Type String", page 17.

<*GenMethDemo Main*>
This chunk is defined in "GenMethDemo Main", page 24.

<*Get Value*>
This chunk is defined in "Implementation of Class GenDemo with Type Integer", page 17.

<*Instance Methods Show and Get*>
This chunk is defined in "Class TwoGen", page 20.

<*Instance Variable ob of Type T*>
This chunk is defined in "Class Gen¡T¿", page 14.

<*Integer Type Parameter*>
This chunk is defined in "Implementation of Class GenDemo with Type Integer", page 16.

<*Makefile CLEAN*>
This chunk is defined in "Makefile Clean Targets", page 47.

<*Makefile CONSTANTS*>
This chunk is defined in "Makefile Constants", page 46.

<*Makefile DEFAULTS*>
This chunk is defined in "Makefile Default Targets", page 46.

<*Makefile TANGLE WEAVE*>
This chunk is defined in "Makefile Tangle Weave Targets", page 46.

<*Method returning object of type T*>
This chunk is defined in "Class Gen¡T¿", page 15.

<*Method showing type of T*>
This chunk is defined in "Class Gen¡T¿", page 15.

<*Reference to Integer Instance*>
This chunk is defined in "Implementation of Class GenDemo with Type Integer", page 17.

<*Show Type*>
This chunk is defined in "Implementation of Class GenDemo with Type Integer", page 17.

<*Static Method isIn*>
This chunk is defined in "Method isIn()", page 24.

<*Two Instance Variables Declarations*>
This chunk is defined in "Class TwoGen", page 19.

B.3 Code Chunk References

<*Class Declaration*>
This chunk is called by <*Class TwoGen*>; see its first definition at "Class TwoGen", page 19.

<*Class Gen*>
This chunk is called by {`SimpleGenerics.java`}; see its first definition at "A Simple Generics Example", page 13.

<*Makefile DEFAULTS*>
This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 46.

<*Makefile TANGLE WEAVE*>
This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 46.

<*Method returning object of type T*>
This chunk is called by <*Class Gen*>; see its first definition at "Class Gen¡T¿", page 14.

<*Method showing type of T*>
This chunk is called by <*Class Gen*>; see its first definition at "Class Gen¡T¿", page 14.

<*Reference to Integer Instance*>
This chunk is called by <*Create a Gen object for Integers*>; see its first definition at "Implementation of Class GenDemo with Type Integer", page 16.

<*Show Type*>
This chunk is called by <*Create a Gen object for Integers*>; see its first definition at "Implementation of Class GenDemo with Type Integer", page 16.

<*Static Method isIn*>
This chunk is called by {`GenMethDemo.java`}; see its first definition at "Example of Generic Method", page 24.

<*Two Instance Variables Declarations*>
This chunk is called by <*Class TwoGen*>; see its first definition at "Class TwoGen", page 19.

List of Tables

List of General Forms

Bibliography

Index

<

{

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W