Revised and Updated for Java SE 6

Core Java™

Volume I · Fundamentals

EIGHTH EDITION

Cay S. Horstmann ▪ Gary Cornell

Table of Contents

Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

Chapter 12

Generic Programming

Generics constitute the most significant change in the Java programming language since the 1.0 release. The addition of generics to Java SE 5.0 was the result of one of the first Java Specification Requests, JSR 14, that was formulated in 1999. The expert group spent about five years on specifications and test implementations.

Generics are desirable because they let you write code that is safer and easier to read than code that is littered with `Object` variables and casts. Generics are particularly useful for collection classes, such as the ubiquitous `ArrayList`.

Generics are—at least on the surface—similar to templates in C++. In C++, as in Java, templates were first added to the language to support strongly typed collections. However, over the years, other uses were discovered. After reading this chapter, perhaps you will find novel uses for Java generics in your programs.

Why Generic Programming?

Generic programming means to write code that can be reused for objects of many different types. For example, you don't want to program separate classes to collect `String` and `File` objects. And you don't have to—the single class `ArrayList` collects objects of any class. This is one example of generic programming.

Before Java SE 5.0, generic programming in Java was always achieved with *inheritance*. The `ArrayList` class simply maintained an array of `Object` references:

```
public class ArrayList // before Java SE 5.0
{
   public Object get(int i) { . . . }
   public void add(Object o) { . . . }
   . . .
   private Object[] elementData;
}
```

This approach has two problems. A cast is necessary whenever you retrieve a value:

```
ArrayList files = new ArrayList();
. . .
String filename = (String) names.get(0);
```

Moreover, there is no error checking. You can add values of any class:

```
files.add(new File(". . ."));
```

This call compiles and runs without error. Elsewhere, casting the result of `get` to a `String` will cause an error.

Generics offer a better solution: *type parameters*. The `ArrayList` class now has a type parameter that indicates the element type:

```
ArrayList<String> files = new ArrayList<String>();
```

This makes your code easier to read. You can tell right away that this particular array list contains `String` objects.

The compiler can make good use of this information too. No cast is required for calling `get`. The compiler knows that the return type is `String`, not `Object`:

```
String filename = files.get(0);
```


Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

The compiler also knows that the add method of an ArrayList<String> has a parameter of type String. That is a lot safer than having an Object parameter. Now the compiler can check that you don't insert objects of the wrong type. For example, the statement

```
files.add(new File(". . ."); // can only add String objects to an ArrayList<String>
```

will not compile. A compiler error is much better than a class cast exception at runtime.

This is the appeal of type parameters: they make your programs easier to read and safer.

Who Wants to Be a Generic Programmer?

It is easy to use a generic class such as ArrayList. Most Java programmers will simply use types such as ArrayList<String> as if they had been built into the language, just like String[] arrays. (Of course, array lists are better than arrays because they can expand automatically.)

However, it is not so easy to implement a generic class. The programmers who use your code will want to plug in all sorts of classes for your type parameters. They expect everything to work without onerous restrictions and confusing error messages. Your job as a generic programmer, therefore, is to anticipate all the potential future uses of your class.

How hard can this get? Here is a typical issue that the designers of the standard class library had to grapple with. The ArrayList class has a method addAll to add all elements of another collection. A programmer may want to add all elements from an ArrayList<Manager> to an ArrayList<Employee>. But, of course, doing it the other way around should not be legal. How do you allow one call and disallow the other? The Java language designers invented an ingenious new concept, the *wildcard type*, to solve this problem. Wildcard types are rather abstract, but they allow a library builder to make methods as flexible as possible.

Generic programming falls into three skill levels. At a basic level, you just use generic classes—typically, collections such as ArrayList—without thinking how and why they work. Most application programmers will want to stay at that level until something goes wrong. You may encounter a confusing error message when mixing different generic classes, or when interfacing with legacy code that knows nothing about type parameters. At that point, you need to learn enough about Java generics to solve problems systematically rather than through random tinkering. Finally, of course, you may want to implement your own generic classes and methods.

Application programmers probably won't write lots of generic code. The folks at Sun have already done the heavy lifting and supplied type parameters for all the collection classes. As a rule of thumb, only code that traditionally involved lots of casts from very general types (such as Object or the Comparable interface) will benefit from using type parameters.

In this chapter, we tell you everything you need to know to implement your own generic code. However, we expect most readers to use this knowledge primarily for help with troubleshooting, and to satisfy their curiosity about the inner workings of the parameterized collection classes.

Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

Definition of a Simple Generic Class

A *generic class* is a class with one or more type variables. In this chapter, we use a simple `Pair` class as an example. This class allows us to focus on generics without being distracted by data storage details. Here is the code for the generic `Pair` class:

```
public class Pair<T>
{
   public Pair() { first = null; second = null; }
   public Pair(T first, T second) { this.first = first;  this.second = second; }

   public T getFirst() { return first; }
   public T getSecond() { return second; }

   public void setFirst(T newValue) { first = newValue; }
   public void setSecond(T newValue) { second = newValue; }

   private T first;
   private T second;
}
```

The `Pair` class introduces a type variable `T`, enclosed in angle brackets < >, after the class name. A generic class can have more than one type variable. For example, we could have defined the `Pair` class with separate types for the first and second field:

```
public class Pair<T, U> { . . . }
```

The type variables are used throughout the class definition to specify method return types and the types of fields and local variables. For example:

```
private T first; // uses type variable
```

NOTE: It is common practice to use uppercase letters for type variables, and to keep them short. The Java library uses the variable `E` for the element type of a collection, `K` and `V` for key and value types of a table, and `T` (and the neighboring letters `U` and `S`, if necessary) for "any type at all".

You *instantiate* the generic type by substituting types for the type variables, such as

```
Pair<String>
```

You can think of the result as an ordinary class with constructors

```
Pair<String>()
Pair<String>(String, String)
```

and methods

```
String getFirst()
String getSecond()
void setFirst(String)
void setSecond(String)
```

In other words, the generic class acts as a factory for ordinary classes.

The program in Listing 12–1 puts the `Pair` class to work. The static `minmax` method traverses an array and simultaneously computes the minimum and maximum value. It uses a `Pair` object to return both results. Recall that the `compareTo` method compares two

Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

strings, returning 0 if the strings are identical, a negative integer if the first string comes before the second in dictionary order, and a positive integer otherwise.

C++ NOTE: Superficially, generic classes in Java are similar to template classes in C++. The only obvious difference is that Java has no special `template` keyword. However, as you will see throughout this chapter, there are substantial differences between these two mechanisms.

Listing 12–1 PairTest1.java

```
1. /**
2.  * @version 1.00 2004-05-10
3.  * @author Cay Horstmann
4.  */
5. public class PairTest1
6. {
7.    public static void main(String[] args)
8.    {
9.       String[] words = { "Mary", "had", "a", "little", "lamb" };
10.      Pair<String> mm = ArrayAlg.minmax(words);
11.      System.out.println("min = " + mm.getFirst());
12.      System.out.println("max = " + mm.getSecond());
13.   }
14. }
15.
16. class ArrayAlg
17. {
18.   /**
19.    * Gets the minimum and maximum of an array of strings.
20.    * @param a an array of strings
21.    * @return a pair with the min and max value, or null if a is null or empty
22.    */
23.   public static Pair<String> minmax(String[] a)
24.   {
25.      if (a == null || a.length == 0) return null;
26.      String min = a[0];
27.      String max = a[0];
28.      for (int i = 1; i < a.length; i++)
29.      {
30.         if (min.compareTo(a[i]) > 0) min = a[i];
31.         if (max.compareTo(a[i]) < 0) max = a[i];
32.      }
33.      return new Pair<String>(min, max);
34.   }
35. }
```


Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

Generic Methods

In the preceding section, you have seen how to define a generic class. You can also define a single method with type parameters.

```
class ArrayAlg
{
   public static <T> T getMiddle(T[] a)
   {
      return a[a.length / 2];
   }
}
```

This method is defined inside an ordinary class, not inside a generic class. However, it is a generic method, as you can see from the angle brackets and the type variable. Note that the type variables are inserted after the modifiers (`public static`, in our case) and before the return type.

You can define generic methods both inside ordinary classes and inside generic classes.

When you call a generic method, you can place the actual types, enclosed in angle brackets, before the method name:

```
String[] names = { "John", "Q.", "Public" };
String middle = ArrayAlg.<String>getMiddle(names);
```

In this case (and indeed in most cases), you can omit the `<String>` type parameter from the method call. The compiler has enough information to infer the method that you want. It matches the type of `names` (that is, `String[]`) against the generic type `T[]` and deduces that `T` must be `String`. That is, you can simply call

```
String middle = ArrayAlg.getMiddle(names);
```

In almost all cases, type inference for generic methods works smoothly. Occasionally, the compiler gets it wrong,and you'll need to decipher an error report. Consider this example:

```
double middle = ArrayAlg.getMiddle(3.14, 1729, 0);
```

The error message is: "found: `java.lang.Number&java.lang.Comparable<? extends java.lang.Number&java.lang.Comparable<?>>`, required: `double`". You will learn later in this chapter how to decipher the "found" type declaration. In a nutshell, the compiler autoboxed the parameters into a `Double` and two `Integer` objects, and then it tried to find a common supertype of these classes. It actually found two: `Number` and the `Comparable` interface, which is itself a generic type. In this case, the remedy is to write all parameters as `double` values.

TIP: Peter von der Ahé recommends this trick if you want to see which type the compiler infers for a generic method call: Purposefully introduce an error and study the resulting error message. For example, consider the call `ArrayAlg.getMiddle("Hello", 0, null)`. Assign the result to a `JButton`, which can't possibly be right. You will get an error report "found: `java.lang.Object&java.io.Serializable&java.lang.Comparable<? extends java.lang.Object&java.io.Serializable&java.lang.Comparable<?>>`."

In plain English, you can assign the result to `Object`, `Serializable`, or `Comparable`.

Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

C++ NOTE: In C++, you place the type parameters after the method name. That can lead to nasty parsing ambiguities. For example, `g(f<a,b>(c))` can mean "call `g` with the result of `f<a,b>(c)`", or "call `g` with the two `boolean` values `f<a` and `b>(c)`".

Bounds for Type Variables

Sometimes, a class or a method needs to place restrictions on type variables. Here is a typical example. We want to compute the smallest element of an array:

```
class ArrayAlg
{
   public static <T> T min(T[] a) // almost correct
   {
      if (a == null || a.length == 0) return null;
      T smallest = a[0];
      for (int i = 1; i < a.length; i++)
         if (smallest.compareTo(a[i]) > 0) smallest = a[i];
      return smallest;
   }
}
```

But there is a problem. Look inside the code of the `min` method. The variable `smallest` has type `T`, which means that it could be an object of an arbitrary class. How do we know that the class to which `T` belongs has a `compareTo` method?

The solution is to restrict `T` to a class that implements the `Comparable` interface—a standard interface with a single method, `compareTo`. You achieve this by giving a *bound* for the type variable `T`:

```
public static <T extends Comparable> T min(T[] a) . . .
```

Actually, the `Comparable` interface is itself a generic type. For now, we will ignore that complexity and the warnings that the compiler generates. "Wildcard Types" on page 632 discusses how to properly use type parameters with the `Comparable` interface.

Now, the generic `min` method can only be called with arrays of classes that implement the `Comparable` interface, such as `String`, `Date`, and so on. Calling `min` with a `Rectangle` array is a compile-time error because the `Rectangle` class does not implement `Comparable`.

C++ NOTE: In C++, you cannot restrict the types of template parameters. If a programmer instantiates a template with an inappropriate type, an (often obscure) error message is reported inside the template code.

You may wonder why you use the `extends` keyword rather than the `implements` keyword in this situation—after all, `Comparable` is an interface. The notation

```
<T extends BoundingType>
```

expresses that `T` should be a *subtype* of the bounding type. Both `T` and the bounding type can be either a class or an interface. The `extends` keyword was chosen because it is a reasonable approximation of the subtype concept, and the Java designers did not want to add a new keyword (such as `sub`) to the language.

Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

A type variable or wildcard can have multiple bounds. For example:

```
T extends Comparable & Serializable
```

The bounding types are separated by ampersands (&) because commas are used to separate type variables.

As with Java inheritance, you can have as many interface supertypes as you like, but at most one of the bounds can be a class. If you have a class as a bound, it must be the first one in the bounds list.

In the next sample program (Listing 12–2), we rewrite the `minmax` method to be generic. The method computes the minimum and maximum of a generic array, returning a `Pair<T>`.

Listing 12–2 PairTest2.java

```
1. import java.util.*;
2.
3. /**
4.  * @version 1.00 2004-05-10
5.  * @author Cay Horstmann
6.  */
7. public class PairTest2
8. {
9.    public static void main(String[] args)
10.   {
11.      GregorianCalendar[] birthdays =
12.         {
13.            new GregorianCalendar(1906, Calendar.DECEMBER, 9), // G. Hopper
14.            new GregorianCalendar(1815, Calendar.DECEMBER, 10), // A. Lovelace
15.            new GregorianCalendar(1903, Calendar.DECEMBER, 3), // J. von Neumann
16.            new GregorianCalendar(1910, Calendar.JUNE, 22), // K. Zuse
17.         };
18.      Pair<GregorianCalendar> mm = ArrayAlg.minmax(birthdays);
19.      System.out.println("min = " + mm.getFirst().getTime());
20.      System.out.println("max = " + mm.getSecond().getTime());
21.   }
22. }
23.
24. class ArrayAlg
25. {
26.    /**
27.       Gets the minimum and maximum of an array of objects of type T.
28.       @param a an array of objects of type T
29.       @return a pair with the min and max value, or null if a is
30.       null or empty
31.    */
32.    public static <T extends Comparable> Pair<T> minmax(T[] a)
33.    {
34.       if (a == null || a.length == 0) return null;
```


Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

Listing 12–2 PairTest2.java (continued)

```
35.       T min = a[0];
36.       T max = a[0];
37.       for (int i = 1; i < a.length; i++)
38.       {
39.          if (min.compareTo(a[i]) > 0) min = a[i];
40.          if (max.compareTo(a[i]) < 0) max = a[i];
41.       }
42.       return new Pair<T>(min, max);
43.    }
44. }
```

Generic Code and the Virtual Machine

The virtual machine does not have objects of generic types—all objects belong to ordinary classes. An earlier version of the generics implementation was even able to compile a program that uses generics into class files that executed on 1.0 virtual machines! This backward compatibility was only abandoned fairly late in the development for Java generics. If you use the Sun compiler to compile code that uses Java generics, the resulting class files will *not* execute on pre-5.0 virtual machines.

NOTE: If you want to have the benefits of generics while retaining bytecode compatibility with older virtual machines, check out `http://sourceforge.net/projects/retroweaver`. The Retroweaver program rewrites class files so that they are compatible with older virtual machines.

Whenever you define a generic type, a corresponding *raw* type is automatically provided. The name of the raw type is simply the name of the generic type, with the type parameters removed. The type variables are *erased* and replaced by their bounding types (or `Object` for variables without bounds.)

For example, the raw type for `Pair<T>` looks like this:

```
public class Pair
{
   public Pair(Object first, Object second)
   {
      this.first = first;
      this.second = second;
   }

   public Object getFirst() { return first; }
   public Object getSecond() { return second; }

   public void setFirst(Object newValue) { first = newValue; }
   public void setSecond(Object newValue) { second = newValue; }

   private Object first;
   private Object second;
}
```


Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

Because T is an unbounded type variable, it is simply replaced by Object.

The result is an ordinary class, just as you might have implemented it before generics were added to the Java programming language.

Your programs may contain different kinds of Pair, such as Pair<String> or Pair<GregorianCalendar>, but erasure turns them all into raw Pair types.

C++ NOTE: In this regard, Java generics are very different from C++ templates. C++ produces different types for each template instantiation, a phenomenon called "template code bloat." Java does not suffer from this problem.

The raw type replaces type variables with the first bound, or Object if no bounds are given. For example, the type variable in the class Pair<T> has no explicit bounds, hence the raw type replaces T with Object. Suppose we declare a slightly different type:

```
public class Interval<T extends Comparable & Serializable> implements Serializable
{
   public Interval(T first, T second)
   {
      if (first.compareTo(second) <= 0) { lower = first; upper = second; }
      else { lower = second; upper = first; }
   }
   . . .
   private T lower;
   private T upper;
}
```

The raw type Interval looks like this:

```
public class Interval implements Serializable
{
   public Interval(Comparable first, Comparable second) { . . . }
   . . .
   private Comparable lower;
   private Comparable upper;
}
```

NOTE: You may wonder what happens if you switch the bounds: class Interval<Serializable & Comparable>. In that case, the raw type replaces T with Serializable, and the compiler inserts casts to Comparable when necessary. For efficiency, you should therefore put tagging interfaces (that is, interfaces without methods) at the end of the bounds list.

Translating Generic Expressions

When you program a call to a generic method, the compiler inserts casts when the return type has been erased. For example, consider the sequence of statements

```
Pair<Employee> buddies = . . .;
Employee buddy = buddies.getFirst();
```

The erasure of getFirst has return type Object. The compiler automatically inserts the cast to Employee. That is, the compiler translates the method call into two virtual machine instructions:

Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

- A call to the raw method `Pair.getFirst`
- A cast of the returned `Object` to the `Employee` type

Casts are also inserted when you access a generic field. Suppose the `first` and `second` fields of the `Pair` class were public. (Not a good programming style, perhaps, but it is legal Java.) Then the expression

```
Employee buddy = buddies.first;
```

also has a cast inserted in the resulting byte codes.

Translating Generic Methods

Type erasure also happens for generic methods. Programmers usually think of a generic method such as

```
public static <T extends Comparable> T min(T[] a)
```

as a whole family of methods, but after erasure, only a single method is left:

```
public static Comparable min(Comparable[] a)
```

Note that the type parameter `T` has been erased, leaving only its bounding type `Comparable`.

Erasure of method brings up a couple of complexities. Consider this example:

```
class DateInterval extends Pair<Date>
{
   public void setSecond(Date second)
   {
      if (second.compareTo(getFirst()) >= 0)
         super.setSecond(second);
   }
   . . .
}
```

A date interval is a pair of `Date` objects, and we'll want to override the methods to ensure that the second value is never smaller than the first. This class is erased to

```
class DateInterval extends Pair // after erasure
{
   public void setSecond(Date second) { . . . }
   . . .
}
```

Perhaps surprisingly, there is another `setSecond` method, inherited from `Pair`, namely,

```
public void setSecond(Object second)
```

This is clearly a different method because it has a parameter of a different type—`Object` instead of `Date`. But it *shouldn't* be different. Consider this sequence of statements:

```
DateInterval interval = new DateInterval(. . .);
Pair<Date> pair = interval; // OK--assignment to superclass
pair.setSecond(aDate);
```

Our expectation is that the call to `setSecond` is polymorphic and that the appropriate method is called. Because `pair` refers to a `DateInterval` object, that should be `DateInterval.setSecond`. The problem is that the type erasure interferes with polymorphism. To fix this problem, the compiler generates a *bridge method* in the `DateInterval` class:

Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

```
public void setSecond(Object second) { setSecond((Date) second); }
```

To see why this works, let us carefully follow the execution of the statement

```
pair.setSecond(aDate)
```

The variable `pair` has declared type `Pair<Date>`, and that type only has a single method called `setSecond`, namely `setSecond(Object)`. The virtual machine calls that method on the object to which `pair` refers. That object is of type `DateInterval`. Therefore, the method `DateInterval.setSecond(Object)` is called. That method is the synthesized bridge method. It calls `DateInterval.setSecond(Date)`, which is what we want.

Bridge methods can get even stranger. Suppose the `DateInterval` method also overrides the `getSecond` method:

```
class DateInterval extends Pair<Date>
{
   public Date getSecond() { return (Date) super.getSecond().clone(); }
   . . .
}
```

In the erased type, there are two `getSecond` methods:

```
Date getSecond() // defined in DateInterval
Object getSecond() // defined in Pair
```

You could not write Java code like that—it would be illegal to have two methods with the same parameter types—here, no parameters. However, in the virtual machine, the parameter types *and the return type* specify a method. Therefore, the compiler can produce bytecodes for two methods that differ only in their return type, and the virtual machine will handle this situation correctly.

NOTE: Bridge methods are not limited to generic types. We already noted in Chapter 5 that, starting with Java SE 5.0, it is legal for a method to specify a more restrictive return type when overriding another method. For example:

```
public class Employee implements Cloneable
{
   public Employee clone() throws CloneNotSupportedException { ... }
}
```

The `Object.clone` and `Employee.clone` methods are said to have *covariant return types*.

Actually, the `Employee` class has *two* clone methods:

```
Employee clone() // defined above
Object clone() // synthesized bridge method, overrides Object.clone
```

The synthesized bridge method calls the newly defined method.

In summary, you need to remember these facts about translation of Java generics:

- There are no generics in the virtual machines, only ordinary classes and methods.
- All type parameters are replaced by their bounds.
- Bridge methods are synthesized to preserve polymorphism.
- Casts are inserted as necessary to preserve type safety.

Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

Calling Legacy Code

Lots of Java code was written before Java SE 5.0. If generic classes could not interoperate with that code, they would probably not be widely used. Fortunately, it is straightforward to use generic classes together with their raw equivalents in legacy APIs.

Let us look at a concrete example. To set the labels of a `JSlider`, you use the method

```
void setLabelTable(Dictionary table)
```

In Chapter 9, we used the following code to populate the label table:

```
Dictionary<Integer, Component> labelTable = new Hashtable<Integer, Component>();
labelTable.put(0, new JLabel(new ImageIcon("nine.gif")));
labelTable.put(20, new JLabel(new ImageIcon("ten.gif")));
. . .
slider.setLabelTable(labelTable); // WARNING
```

In Java SE 5.0, the `Dictionary` and `Hashtable` classes were turned into a generic class. Therefore, we are able to form `Dictionary<Integer, Component>` instead of using a raw `Dictionary`. However, when you pass the `Dictionary<Integer, Component>` object to `setLabelTable`, the compiler issues a warning.

```
Dictionary<Integer, Component> labelTable = . . .;
slider.setLabelTable(labelTable); // WARNING
```

After all, the compiler has no assurance about what the `setLabelTable` might do to the `Dictionary` object. That method might replace all the keys with strings. That breaks the guarantee that the keys have type `Integer`, and future operations may cause bad cast exceptions.

There isn't much you can do with this warning, except ponder it and ask what the `JSlider` is likely going to do with this `Dictionary` object. In our case, it is pretty clear that the `JSlider` only reads the information, so we can ignore the warning.

Now consider the opposite case, in which you get an object of a raw type from a legacy class. You can assign it to a parameterized type variable, but of course you will get a warning. For example:

```
Dictionary<Integer, Components> labelTable = slider.getLabelTable(); // WARNING
```

That's ok—review the warning and make sure that the label table really contains `Integer` and `Component` objects. Of course, there never is an absolute guarantee. A malicious coder might have installed a different `Dictionary` in the slider. But again, the situation is no worse than it was before Java SE 5.0. In the worst case, your program will throw an exception.

After you are done pondering the warning, you can use an *annotation* to make it disappear. The annotation must be placed before the method whose code generates the warning, like this:

```
@SuppressWarnings("unchecked")
public void configureSlider() { . . . }
```

Unfortunately, this annotation turns off checking for all code inside the method. It is a good idea to isolate potentially unsafe code into separate methods so that they can be reviewed more easily.

Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

NOTE: The `Hashtable` class is a concrete subclass of the abstract `Dictionary` class. Both `Dictionary` and `Hashtable` have been declared as "obsolete" ever since they were superseded by the `Map` interface and the `HashMap` class of Java SE 1.2. Apparently though, they are still alive and kicking. After all, the `JSlider` class was only added in Java SE 1.3. Didn't its programmers know about the `Map` class by then? Does this make you hopeful that they are going to adopt generics in the near future? Well, that's the way it goes with legacy code.

Restrictions and Limitations

In the following sections, we discuss a number of restrictions that you need to consider when working with Java generics. Most of these restrictions are a consequence of type erasure.

Type Parameters Cannot Be Instantiated with Primitive Types

You cannot substitute a primitive type for a type parameter. Thus, there is no `Pair<double>`, only `Pair<Double>`. The reason is, of course, type erasure. After erasure, the `Pair` class has fields of type `Object`, and you can't use them to store `double` values.

This is an annoyance, to be sure, but it is consistent with the separate status of primitive types in the Java language. It is not a fatal flaw—there are only eight primitive types, and you can always handle them with separate classes and methods when wrapper types are not an acceptable substitute.

Runtime Type Inquiry Only Works with Raw Types

Objects in the virtual machine always have a specific nongeneric type. Therefore, all type inquiries yield only the raw type. For example,

```
if (a instanceof Pair<String>) // same as a instanceof Pair
```

really only tests whether `a` is a `Pair` of any type. The same is true for the test

```
if (a instanceof Pair<T>) // T is ignored
```

or the cast

```
Pair<String> p = (Pair<String>) a; // WARNING--can only test that a is a Pair
```

To remind you of the risk, you will get a compiler warning whenever you use `instanceof` or cast expressions that involve generic types.

In the same spirit, the `getClass` method always returns the raw type. For example:

```
Pair<String> stringPair = . . .;
Pair<Employee> employeePair = . . .;
if (stringPair.getClass() == employeePair.getClass()) // they are equal
```

The comparison yields `true` because both calls to `getClass` return `Pair.class`.

You Cannot Throw or Catch Instances of a Generic Class

You can neither throw nor catch objects of a generic class. In fact, it is not even legal for a generic class to extend `Throwable`. For example, the following definition will not compile:

```
public class Problem<T> extends Exception { /* . . . */ } // ERROR--can't extend Throwable
```

You cannot use a type variable in a `catch` clause. For example, the following method will not compile:

Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

```
public static <T extends Throwable> void doWork(Class<T> t)
{
   try
   {
      do work
   }
   catch (T e) // ERROR--can't catch type variable
   {
      Logger.global.info(...)
   }
}
```

However, it is ok to use type variables in exception specifications. The following method is legal:

```
public static <T extends Throwable> void doWork(T t) throws T // OK
{
   try
   {
      do work
   }
   catch (Throwable realCause)
   {
      t.initCause(realCause);
      throw t;
   }
}
```

Arrays of Parameterized Types Are Not Legal

You cannot declare arrays of parameterized types, such as

```
Pair<String>[] table = new Pair<String>[10]; // ERROR
```

What's wrong with that? After erasure, the type of `table` is `Pair[]`. You can convert it to `Object[]`:

```
Object[] objarray = table;
```

An array remembers its component type and throws an `ArrayStoreException` if you try to store an element of the wrong type:

```
objarray[0] = "Hello"; // ERROR--component type is Pair
```

But erasure renders this mechanism ineffective for generic types. The assignment

```
objarray[0] = new Pair<Employee>();
```

would pass the array store check but still result in a type error. For this reason, arrays of parameterized types are outlawed.

TIP: If you need to collect parameterized type objects, simply use an `ArrayList`: `ArrayList<Pair<String>>` is safe and effective.

Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

You Cannot Instantiate Type Variables

You cannot use type variables in expression such as `new T(...)`, `new T[...]`, or `T.class`. For example, the following `Pair<T>` constructor is illegal:

```
public Pair() { first = new T(); second = new T(); } // ERROR
```

Type erasure would change `T` to `Object`, and surely you don't want to call `new Object()`.

As a workaround, you can construct generic objects through reflection, by calling the `Class.newInstance` method.

Unfortunately, the details are a bit complex. You cannot call

```
first = T.class.newInstance(); // ERROR
```

The expression `T.class` is not legal. Instead, you must design the API so that you are handed a `Class` object, like this:

```
public static <T> Pair<T> makePair(Class<T> cl)
{
   try { return new Pair<T>(cl.newInstance(), cl.newInstance()) }
   catch (Exception ex) { return null; }
}
```

This method could be called as follows:

```
Pair<String> p = Pair.makePair(String.class);
```

Note that the `Class` class is itself generic. For example, `String.class` is an instance (indeed, the sole instance) of `Class<String>`. Therefore, the `makePair` method can infer the type of the pair that it is making.

You cannot construct a generic array:

```
public static <T extends Comparable> T[] minmax(T[] a) { T[] mm = new T[2]; . . . } // ERROR
```

Type erasure would cause this method to always construct an array `Object[2]`.

If the array is only used as a private instance field of a class, you can declare the array as `Object[]` and use casts when retrieving elements. For example, the `ArrayList` class could be implemented as follows:

```
public class ArrayList<E>
{
   private Object[] elements;
   @SuppressWarnings("unchecked") public E get(int n) { return (E) elements[n]; }
   public void set(int n, E e) { elements[n] = e; } // no cast needed
   . . .
}
```

The actual implementation is not quite as clean:

```
public class ArrayList<E>
{
   private E[] elements;
   public ArrayList() { elements = (E[]) new Object[10]; }
   . . .
}
```

Here, the cast `E[]` is an outright lie, but type erasure makes it undetectable.

Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

This technique does not work for our `minmax` method since we are returning a `T[]` array, and a runtime error results if we lie about its type. Suppose we implement

```
public static <T extends Comparable> T[] minmax(T[] a)
{
   Object[] mm = new Object[2];
   . . .;
   return (T[]) mm; // compiles with warning
}
```

The call

```
String[] ss = minmax("Tom", "Dick", "Harry");
```

compiles without any warning. A `ClassCastException` occurs when the `Object[]` reference is assigned to the `String[]` variable.

In this situation, you can use reflection and call `Array.newInstance`:

```
public static <T extends Comparable> T[] minmax(T[] a)
{
   T[] mm = (T[]) Array.newInstance(a.getClass().getComponentType(), 2);
   . . .
}
```

The `toArray` method of the `ArrayList` class is not so lucky. It needs to produce a `T[]` array, but it doesn't have the component type. Therefore, there are two variants:

```
Object[] toArray()
T[] toArray(T[] result)
```

The second method receives an array parameter. If the array is large enough, it is used. Otherwise, a new array of sufficient size is created, using the component type of `result`.

Type Variables Are Not Valid in Static Contexts of Generic Classes

You cannot reference type variables in static fields or methods. For example, the following clever idea won't work:

```
public class Singleton<T>
{
   public static T getSingleInstance() // ERROR
   {
      if (singleInstance == null) construct new instance of T
      return singleInstance;
   }
   private static T singleInstance; // ERROR
}
```

If this could be done, then a program could declare a `Singleton<Random>` to share a random number generator and a `Singleton<JFileChooser>` to share a file chooser dialog. But it can't work. After type erasure there is only one `Singleton` class, and only one `singleInstance` field. For that reason, static fields and methods with type variables are simply outlawed.

Beware of Clashes After Erasure

It is illegal to create conditions that cause clashes when generic types are erased. Here is an example. Suppose we add an `equals` method to the `Pair` class, like this:

Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

```
public class Pair<T>
{
   public boolean equals(T value) { return first.equals(value) && second.equals(value); }
   . . .
}
```

Consider a `Pair<String>`. Conceptually, it has two `equals` methods:

```
boolean equals(String) // defined in Pair<T>
boolean equals(Object) // inherited from Object
```

But the intuition leads us astray. The erasure of the method

```
boolean equals(T)
```

is

```
boolean equals(Object)
```

which clashes with the `Object.equals` method.

The remedy is, of course, to rename the offending method.

The generics specification cites another rule: "To support translation by erasure, we impose the restriction that a class or type variable may not at the same time be a subtype of two interface types which are different parameterizations of the same interface." For example, the following is illegal:

```
class Calendar implements Comparable<Calendar> { . . . }
class GregorianCalendar extends Calendar implements Comparable<GregorianCalendar>
   { . . . } // ERROR
```

`GregorianCalendar` would then implement both `Comparable<Calendar>` and `Comparable<GregorianCalendar>`, which are different parameterizations of the same interface.

It is not obvious what this restriction has to do with type erasure. After all, the nongeneric version

```
class Calendar implements Comparable { . . . }
class GregorianCalendar extends Calendar implements Comparable { . . . }
```

is legal. The reason is far more subtle. There would be a conflict with the synthesized bridge methods. A class that implements `Comparable<X>` gets a bridge method

```
public int compareTo(Object other) { return compareTo((X) other); }
```

You cannot have two such methods for different types `X`.

Inheritance Rules for Generic Types

When you work with generic classes, you need to learn a few rules about inheritance and subtypes. Let's start with a situation that many programmers find unintuitive. Consider a class and a subclass, such as `Employee` and `Manager`. Is `Pair<Manager>` a subclass of `Pair<Employee>`? Perhaps surprisingly, the answer is "no." For example, the following code will not compile:

```
Manager[] topHonchos = . . .;
Pair<Employee> result = ArrayAlg.minmax(topHonchos); // ERROR
```

The `minmax` method returns a `Pair<Manager>`, not a `Pair<Employee>`, and it is illegal to assign one to the other.

In general, there is *no* relationship between `Pair<S>` and `Pair<T>`, no matter how `S` and `T` are related (see Figure 12–1).

Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

This seems like a cruel restriction, but it is necessary for type safety. Suppose we were allowed to convert a `Pair<Manager>` to a `Pair<Employee>`. Consider this code:

```
Pair<Manager> managerBuddies = new Pair<Manager>(ceo, cfo);
Pair<Employee> employeeBuddies = managerBuddies; // illegal, but suppose it wasn't
employeeBuddies.setFirst(lowlyEmployee);
```

Clearly, the last statement is legal. But `employeeBuddies` and `managerBuddies` refer to the *same object*. We now managed to pair up the CFO with a lowly employee, which should not be possible for a `Pair<Manager>`.

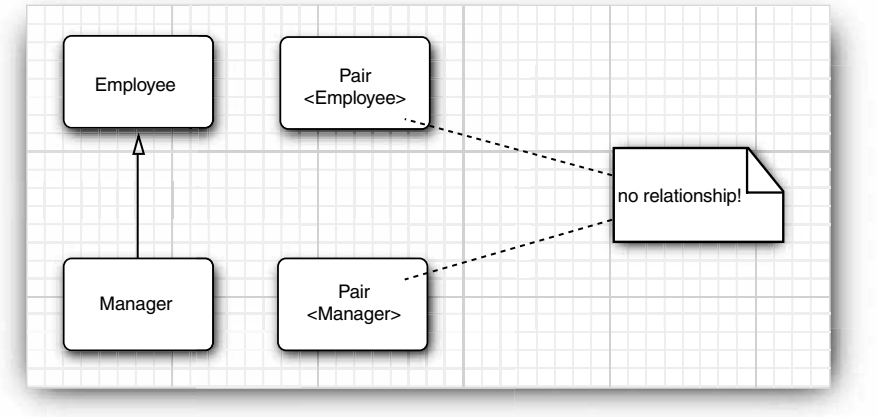

Figure 12–1 No inheritance relationship between pair classes

NOTE: You just saw an important difference between generic types and Java arrays. You can assign a `Manager[]` array to a variable of type `Employee[]`:

```
Manager[] managerBuddies = { ceo, cfo };
Employee[] employeeBuddies = managerBuddies; // OK
```

However, arrays come with special protection. If you try to store a lowly employee into `employeeBuddies[0]`, the virtual machine throws an `ArrayStoreException`.

You can always convert a parameterized type to a raw type. For example, `Pair<Employee>` is a subtype of the raw type `Pair`. This conversion is necessary for interfacing with legacy code.

Can you convert to the raw type and then cause a type error? Unfortunately, you can. Consider this example:

```
Pair<Manager> managerBuddies = new Pair<Manager>(ceo, cfo);
Pair rawBuddies = managerBuddies; // OK
rawBuddies.setFirst(new File(". . ."); // only a compile-time warning
```

This sounds scary. However, keep in mind that you are no worse off than you were with older versions of Java. The security of the virtual machine is not at stake. When the

Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

foreign object is retrieved with `getFirst` and assigned to a `Manager` variable, a `ClassCastException` is thrown, just as in the good old days. You merely lose the added safety that generic programming normally provides.

Finally, generic classes can extend or implement other generic classes. In this regard, they are no different from ordinary classes. For example, the class `ArrayList<T>` implements the interface `List<T>`. That means, an `ArrayList<Manager>` can be converted to a `List<Manager>`. However, as you just saw, an `ArrayList<Manager>` is *not* an `ArrayList<Employee>` or `List<Employee>`. Figure 12–2 shows these relationships.

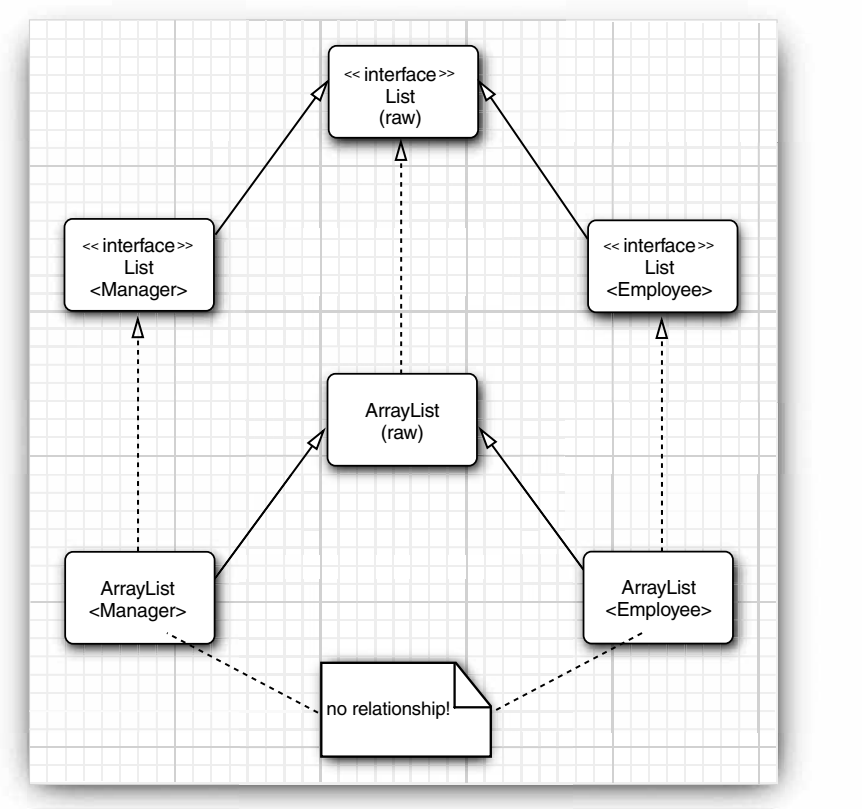

Figure 12–2 Subtype relationships among generic list types

Wildcard Types

It was known for some time among researchers of type systems that a rigid system of generic types is quite unpleasant to use. The Java designers invented an ingenious (but nevertheless safe) "escape hatch": the *wildcard type*. For example, the wildcard type

```
Pair<? extends Employee>
```


Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

denotes any generic Pair type whose type parameter is a subclass of Employee, such as Pair<Manager>, but not Pair<String>.

Let's say you want to write a method that prints out pairs of employees, like this:

```
public static void printBuddies(Pair<Employee> p)
{
   Employee first = p.getFirst();
   Employee second = p.getSecond();
   System.out.println(first.getName() + " and " + second.getName() + " are buddies.");
}
```

As you saw in the preceding section, you cannot pass a Pair<Manager> to that method, which is rather limiting. But the solution is simple—use a wildcard type:

```
public static void printBuddies(Pair<? extends Employee> p)
```

The type Pair<Manager> is a subtype of Pair<? extends Employee> (see Figure 12–3).

Can we use wildcards to corrupt a Pair<Manager> through a Pair<? extends Employee> reference?

```
Pair<Manager> managerBuddies = new Pair<Manager>(ceo, cfo);
Pair<? extends Employee> wildcardBuddies = managerBuddies; // OK
wildcardBuddies.setFirst(lowlyEmployee); // compile-time error
```

No corruption is possible. The call to setFirst is a type error. To see why, let us have a closer look at the type Pair<? extends Employee>. Its methods look like this:

```
? extends Employee getFirst()
void setFirst(? extends Employee)
```

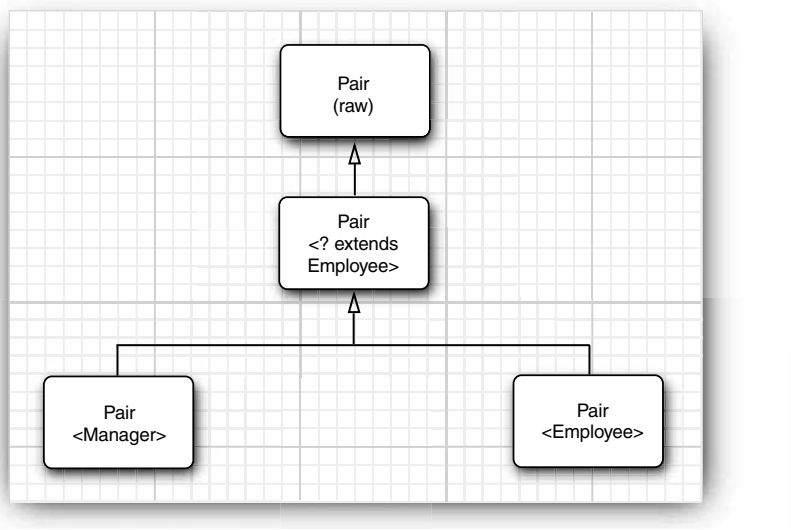

Figure 12–3 Subtype relationships with wildcards

Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

This makes it impossible to call the `setFirst` method. The compiler only knows that it needs some subtype of `Employee`, but it doesn't know which type. It refuses to pass any specific type—after all, `?` might not match it.

We don't have this problem with `getFirst`: It is perfectly legal to assign the return value of `getFirst` to an `Employee` reference.

This is the key idea behind bounded wildcards. We now have a way of distinguishing between the safe accessor methods and the unsafe mutator methods.

Supertype Bounds for Wildcards

Wildcard bounds are similar to type variable bounds, but they have an added capability— you can specify a *supertype bound*, like this:

```
? super Manager
```

This wildcard is restricted to all supertypes of `Manager`. (It was a stroke of good luck that the existing `super` keyword describes the relationship so accurately.)

Why would you want to do this? A wildcard with a supertype bound gives you the opposite behavior of the wildcards described in the section "Wildcard Types" on page 632. You can supply parameters to methods, but you can't use the return values. For example, `Pair<? super Manager>` has methods

```
void setFirst(? super Manager)
? super Manager getFirst()
```

The compiler doesn't know the exact type of the `setFirst` method but can call it with any object of type `Manager`, `Employee`, or `Object`, but not a subtype such as `Executive`. However, if you call `getFirst`, there is no guarantee about the type of the returned object. You can only assign it to an `Object`.

Here is a typical example. We have an array of managers and want to put the manager with the lowest and highest bonus into a `Pair` object. What kind of `Pair`? A `Pair<Employee>` should be fair game or, for that matter, a `Pair<Object>` (see Figure 12–4). The following method will accept any appropriate `Pair`:

```
public static void minmaxBonus(Manager[] a, Pair<? super Manager> result)
{
   if (a == null || a.length == 0) return;
   Manager min = a[0];
   Manager max = a[0];
   for (int i = 1; i < a.length; i++)
   {
      if (min.getBonus() > a[i].getBonus()) min = a[i];
      if (max.getBonus() < a[i].getBonus()) max = a[i];
   }
   result.setFirst(min);
   result.setSecond(max);
}
```

Intuitively speaking, wildcards with supertype bounds let you write to a generic object, wildcards with subtype bounds let you read from a generic objects.

Here is another use for supertype bounds. The `Comparable` interface is itself a generic type. It is declared as follows:

Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

```
public interface Comparable<T>
{
    public int compareTo(T other);
}
```

Here, the type variable indicates the type of the `other` parameter. For example, the `String` class implements `Comparable<String>`, and its `compareTo` method is declared as

```
public int compareTo(String other)
```

This is nice—the explicit parameter has the correct type. Before Java SE 5.0, `other` was an `Object`, and a cast was necessary in the implementation of the method.

Because `Comparable` is a generic type, perhaps we should have done a better job with the `min` method of the `ArrayAlg` class? We could have declared it as

```
public static <T extends Comparable<T>> T min(T[] a)
```

This looks more thorough than just using `T extends Comparable`, and it would work fine for many classes. For example, if you compute the minimum of a `String` array, then `T` is the type `String`, and `String` is a subtype of `Comparable<String>`. But we run into a problem when processing an array of `GregorianCalendar` objects. As it happens, `GregorianCalendar` is a subclass of `Calendar`, and `Calendar` implements `Comparable<Calendar>`. Thus, `GregorianCalendar` implements `Comparable<Calendar>` but not `Comparable<GregorianCalendar>`.

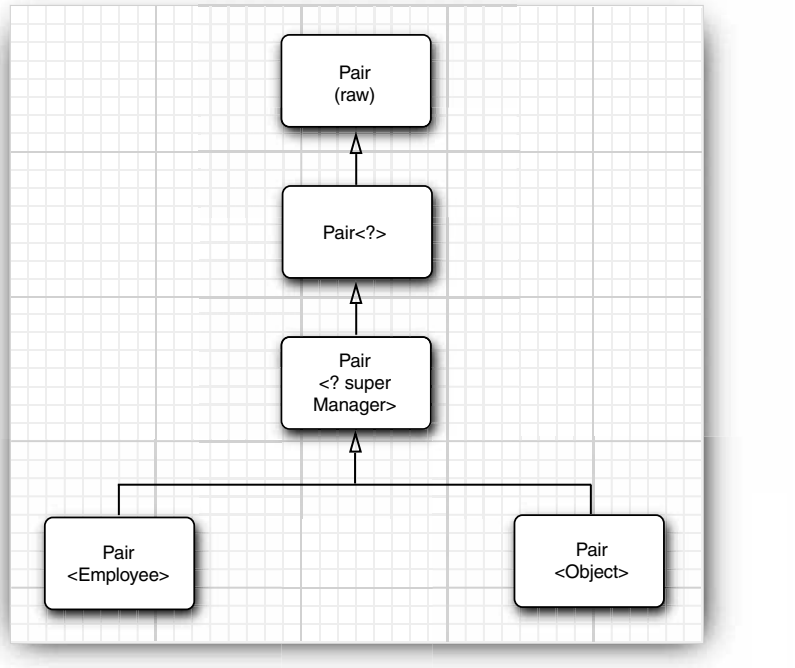

Figure 12–4 A wildcard with a supertype bound

Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

In a situation such as this one, supertypes come to the rescue:

```
public static <T extends Comparable<? super T>> T min(T[] a) . . .
```

Now the `compareTo` method has the form

```
int compareTo(? super T)
```

Maybe it is declared to take an object of type `T`, or—for example, when `T` is `GregorianCalendar`—a supertype of `T`. At any rate, it is safe to pass an object of type `T` to the `compareTo` method.

To the uninitiated, a declaration such as `<T extends Comparable<? super T>>` is bound to look intimidating. This is unfortunate, because the intent of this declaration is to help application programmers by removing unnecessary restrictions on the call parameters. Application programmers with no interest in generics will probably learn quickly to gloss over these declarations and just take for granted that library programmers will do the right thing. If you are a library programmer, you'll need to get used to wildcards, or your users will curse you and throw random casts at their code until it compiles.

Unbounded Wildcards

You can even use wildcards with no bounds at all, for example, `Pair<?>`. At first glance, this looks identical to the raw `Pair` type. Actually, the types are very different. The type `Pair<?>` has methods such as

```
? getFirst()
void setFirst(?)
```

The return value of `getFirst` can only be assigned to an `Object`. The `setFirst` method can never be called, *not even with an* `Object`. That's the essential difference between `Pair<?>` and `Pair`: you can call the `setObject` method of the raw `Pair` class with *any* `Object`.

Why would you ever want such a wimpy type? It is useful for very simple operations. For example, the following method tests whether a pair contains a given object. It never needs the actual type.

```
public static boolean hasNulls(Pair<?> p)
{
   return p.getFirst() == null || p.getSecond() == null;
}
```

You could have avoided the wildcard type by turning `contains` into a generic method:

```
public static <T> boolean hasNulls(Pair<T> p)
```

However, the version with the wildcard type seems easier to read.

Wildcard Capture

Let us write a method that swaps the elements of a pair:

```
public static void swap(Pair<?> p)
```

A wildcard is not a type variable, so we can't write code that uses ? as a type. In other words, the following would be illegal:

```
? t = p.getFirst(); // ERROR
p.setFirst(p.getSecond());
p.setSecond(t);
```

That's a problem because we need to temporarily hold the first element when we do the swapping. Fortunately, there is an interesting solution to this problem. We can write a helper method, `swapHelper`, like this:

Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

```
public static <T> void swapHelper(Pair<T> p)
{
   T t = p.getFirst();
   p.setFirst(p.getSecond());
   p.setSecond(t);
}
```

Note that `swapHelper` is a generic method, whereas `swap` is not—it has a fixed parameter of type `Pair<?>`.

Now we can call `swapHelper` from `swap`:

```
public static void swap(Pair<?> p) { swapHelper(p); }
```

In this case, the parameter `T` of the `swapHelper` method *captures the wildcard*. It isn't known what type the wildcard denotes, but it is a definite type, and the definition of `<T>swapHelper` makes perfect sense when `T` denotes that type.

Of course, in this case, we were not compelled to use a wildcard. We could have directly implemented `<T> void swap(Pair<T> p)` as a generic method without wildcards. However, consider this example in which a wildcard type occurs naturally in the middle of a computation:

```
public static void maxminBonus(Manager[] a, Pair<? super Manager> result)
{
   minmaxBonus(a, result);
   PairAlg.swapHelper(result); // OK--swapHelper captures wildcard type
}
```

Here, the wildcard capture mechanism cannot be avoided.

Wildcard capture is only legal in very limited circumstances. The compiler must be able to guarantee that the wildcard represents a single, definite type. For example, the `T` in `ArrayList<Pair<T>>` can never capture the wildcard in `ArrayList<Pair<?>>`. The array list might hold two `Pair<?>`, each of which has a different type for `?`.

The test program in Listing 12–3 gathers up the various methods that we discussed in the preceding sections, so that you can see them in context.

Listing 12–3 PairTest3.java

```
1. import java.util.*;
2.
3. /**
4.    * @version 1.00 2004-05-10
5.    * @author Cay Horstmann
6.    */
7. public class PairTest3
8. {
9.    public static void main(String[] args)
10.    {
11.       Manager ceo = new Manager("Gus Greedy", 800000, 2003, 12, 15);
12.       Manager cfo = new Manager("Sid Sneaky", 600000, 2003, 12, 15);
13.       Pair<Manager> buddies = new Pair<Manager>(ceo, cfo);
14.       printBuddies(buddies);
```


Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

Listing 12–3 PairTest3.java (continued)

```
15.
16.       ceo.setBonus(1000000);
17.       cfo.setBonus(500000);
18.       Manager[] managers = { ceo, cfo };
19.
20.       Pair<Employee> result = new Pair<Employee>();
21.       minmaxBonus(managers, result);
22.       System.out.println("first: " + result.getFirst().getName()
23.          + ", second: " + result.getSecond().getName());
24.       maxminBonus(managers, result);
25.       System.out.println("first: " + result.getFirst().getName()
26.          + ", second: " + result.getSecond().getName());
27.    }
28.
29.    public static void printBuddies(Pair<? extends Employee> p)
30.    {
31.       Employee first = p.getFirst();
32.       Employee second = p.getSecond();
33.       System.out.println(first.getName() + " and " + second.getName() + " are buddies.");
34.    }
35.
36.    public static void minmaxBonus(Manager[] a, Pair<? super Manager> result)
37.    {
38.       if (a == null || a.length == 0) return;
39.       Manager min = a[0];
40.       Manager max = a[0];
41.       for (int i = 1; i < a.length; i++)
42.       {
43.          if (min.getBonus() > a[i].getBonus()) min = a[i];
44.          if (max.getBonus() < a[i].getBonus()) max = a[i];
45.       }
46.       result.setFirst(min);
47.       result.setSecond(max);
48.    }
49.
50.    public static void maxminBonus(Manager[] a, Pair<? super Manager> result)
51.    {
52.       minmaxBonus(a, result);
53.       PairAlg.swapHelper(result); // OK--swapHelper captures wildcard type
54.    }
55. }
56.
57. class PairAlg
58. {
59.    public static boolean hasNulls(Pair<?> p)
60.    {
61.       return p.getFirst() == null || p.getSecond() == null;
62.    }
63.
```


Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

Listing 12–3 PairTest3.java (continued)

```
64.    public static void swap(Pair<?> p) { swapHelper(p); }
65.
66.    public static <T> void swapHelper(Pair<T> p)
67.    {
68.       T t = p.getFirst();
69.       p.setFirst(p.getSecond());
70.       p.setSecond(t);
71.    }
72. }
73.
74. class Employee
75. {
76.    public Employee(String n, double s, int year, int month, int day)
77.    {
78.       name = n;
79.       salary = s;
80.       GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
81.       hireDay = calendar.getTime();
82.    }
83.
84.    public String getName()
85.    {
86.       return name;
87.    }
88.
89.    public double getSalary()
90.    {
91.       return salary;
92.    }
93.
94.    public Date getHireDay()
95.    {
96.       return hireDay;
97.    }
98.
99.    public void raiseSalary(double byPercent)
100.   {
101.      double raise = salary * byPercent / 100;
102.      salary += raise;
103.   }
104.
105.   private String name;
106.   private double salary;
107.   private Date hireDay;
108. }
109.
110. class Manager extends Employee
111. {
```


Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

Listing 12–3 PairTest3.java (continued)

```
112.    /**
113.       @param n the employee's name
114.       @param s the salary
115.       @param year the hire year
116.       @param month the hire month
117.       @param day the hire day
118.    */
119.    public Manager(String n, double s, int year, int month, int day)
120.    {
121.       super(n, s, year, month, day);
122.       bonus = 0;
123.    }
124.
125.    public double getSalary()
126.    {
127.       double baseSalary = super.getSalary();
128.       return baseSalary + bonus;
129.    }
130.
131.    public void setBonus(double b)
132.    {
133.       bonus = b;
134.    }
135.
136.    public double getBonus()
137.    {
138.       return bonus;
139.    }
140.
141.    private double bonus;
142. }
```

Reflection and Generics

The `Class` class is now generic. For example, `String.class` is actually an object (in fact, the sole object) of the class `Class<String>`.

The type parameter is useful because it allows the methods of `Class<T>` to be more specific about their return types. The following methods of `Class<T>` take advantage of the type parameter:

```
T newInstance()
T cast(Object obj)
T[] getEnumConstants()
Class<? super T> getSuperclass()
Constructor<T> getConstructor(Class... parameterTypes)
Constructor<T> getDeclaredConstructor(Class... parameterTypes)
```

The `newInstance` method returns an instance of the class, obtained from the default constructor. Its return type can now be declared to be `T`, the same type as the class that is being described by `Class<T>`. That saves a cast.

Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

The cast method returns the given object, now declared as type T if its type is indeed a subtype of T. Otherwise, it throws a BadCastException.

The getEnumConstants method returns null if this class is not an enum class or an array of the enumeration values, which are known to be of type T.

Finally, the getConstructor and getDeclaredConstructor methods return a Constructor<T> object. The Constructor class has also been made generic so that its newInstance method has the correct return type.

API **java.lang.Class<T> 1.0**

- T newInstance() **5.0**
  returns a new instance constructed with the default constructor.
- T cast(Object obj) **5.0**
  returns obj if it is null or can be converted to the type T, or throws a BadCastException otherwise.
- T[] getEnumConstants() **5.0**
  returns an array of all values if T is an enumerated type, null otherwise.
- Class<? super T> getSuperclass() **5.0**
  returns the superclass of this class, or null if T is not a class or the class Object.
- Constructor<T> getConstructor(Class... parameterTypes) **5.0**
- Constructor<T> getDeclaredConstructor(Class... parameterTypes) **5.0**
  gets the public constructor, or the constructor with the given parameter types.

API **java.lang.reflect.Constructor<T> 1.1**

- T newInstance(Object... parameters) **5.0**
  returns a new instance constructed with the given parameters.

Using Class<T> *Parameters for Type Matching*

It is sometimes useful to match the type variable of a Class<T> parameter in a generic method. Here is the canonical example:

```
public static <T> Pair<T> makePair(Class<T> c) throws InstantiationException,
      IllegalAccessException
{
   return new Pair<T>(c.newInstance(), c.newInstance());
}
```

If you call

```
makePair(Employee.class)
```

then Employee.class is an object of type Class<Employee>. The type parameter T of the makePair method matches Employee, and the compiler can infer that the method returns a Pair<Employee>.

Generic Type Information in the Virtual Machine

One of the notable features of Java generics is the erasure of generic types in the virtual machine. Perhaps surprisingly, the erased classes still retain some faint memory of their generic origin. For example, the raw Pair class knows that it originated from the generic

Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

class Pair<T>, even though an object of type Pair can't tell whether it was constructed as a Pair<String> or Pair<Employee>.

Similarly, consider a method

```
public static Comparable min(Comparable[] a)
```

that is the erasure of a generic method

```
public static <T extends Comparable<? super T>> T min(T[] a)
```

You can use the reflection API enhancements of Java SE 5.0 to determine that

- The generic method has a type parameter called T;
- The type parameter has a subtype bound that is itself a generic type;
- The bounding type has a wildcard parameter;
- The wildcard parameter has a supertype bound; and
- The generic method has a generic array parameter.

In other words, you get to reconstruct everything about generic classes and methods that their implementors declared. However, you won't know how the type parameters were resolved for specific objects or method calls.

NOTE: The type information that is contained in class files to enable reflection of generics is incompatible with older virtual machines.

In order to express generic type declarations, Java SE 5.0 introduced a new interface Type in the java.lang.reflect package. The interface has the following subtypes:

- The Class class, describing concrete types
- The TypeVariable interface, describing type variables (such as T extends Comparable<? super T>)
- The WildcardType interface, describing wildcards (such as ? super T)
- The ParameterizedType interface, describing generic class or interface types (such as Comparable<? super T>)
- The GenericArrayType interface, describing generic arrays (such as T[])

Figure 12–5 shows the inheritance hierarchy. Note that the last four subtypes are interfaces—the virtual machine instantiates suitable classes that implement these interfaces.

Listing 12–4 uses the generic reflection API to print out what it discovers about a given class. If you run it with the Pair class, you get this report:

```
class Pair<T> extends java.lang.Object
public T getFirst()
public T getSecond()
public void setFirst(T)
public void setSecond(T)
```

If you run it with ArrayAlg in the PairTest2 directory, the report displays the following method:

```
public static <T extends java.lang.Comparable> Pair<T> minmax(T[])
```

The API notes at the end of this section describe the methods used in the example program.

Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

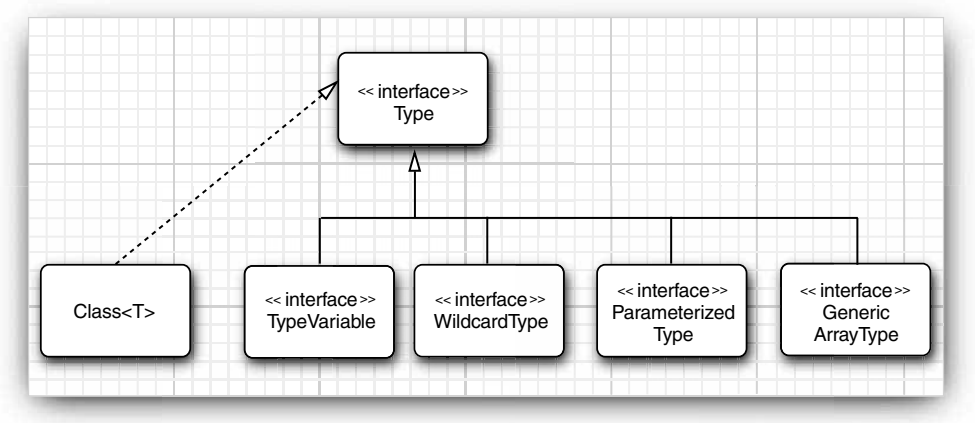

Figure 12–5 The `Type` class and its descendants

Listing 12–4 GenericReflectionTest.java

```
1. import java.lang.reflect.*;
2. import java.util.*;
3.
4. /**
5.  * @version 1.10 2004-05-15
6.  * @author Cay Horstmann
7.  */
8. public class GenericReflectionTest
9. {
10.    public static void main(String[] args)
11.    {
12.       // read class name from command-line args or user input
13.       String name;
14.       if (args.length > 0) name = args[0];
15.       else
16.       {
17.          Scanner in = new Scanner(System.in);
18.          System.out.println("Enter class name (e.g. java.util.Collections): ");
19.          name = in.next();
20.       }
21.
22.       try
23.       {
24.          // print generic info for class and public methods
25.          Class cl = Class.forName(name);
26.          printClass(cl);
27.          for (Method m : cl.getDeclaredMethods())
28.             printMethod(m);
```


Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

Listing 12–4 GenericReflectionTest.java (continued)

```
29.       }
30.       catch (ClassNotFoundException e)
31.       {
32.          e.printStackTrace();
33.       }
34.    }
35.
36.    public static void printClass(Class cl)
37.    {
38.       System.out.print(cl);
39.       printTypes(cl.getTypeParameters(), "<", ", ", ">", true);
40.       Type sc = cl.getGenericSuperclass();
41.       if (sc != null)
42.       {
43.          System.out.print(" extends ");
44.          printType(sc, false);
45.       }
46.       printTypes(cl.getGenericInterfaces(), " implements ", ", ", "", false);
47.       System.out.println();
48.    }
49.
50.    public static void printMethod(Method m)
51.    {
52.       String name = m.getName();
53.       System.out.print(Modifier.toString(m.getModifiers()));
54.       System.out.print(" ");
55.       printTypes(m.getTypeParameters(), "<", ", ", "> ", true);
56.
57.       printType(m.getGenericReturnType(), false);
58.       System.out.print(" ");
59.       System.out.print(name);
60.       System.out.print("(");
61.       printTypes(m.getGenericParameterTypes(), "", ", ", "", false);
62.       System.out.println(")");
63.    }
64.
65.    public static void printTypes(Type[] types, String pre, String sep, String suf,
66.          boolean isDefinition)
67.    {
68.       if (pre.equals(" extends ") && Arrays.equals(types, new Type[] { Object.class })) return;
69.       if (types.length > 0) System.out.print(pre);
70.       for (int i = 0; i < types.length; i++)
71.       {
72.          if (i > 0) System.out.print(sep);
73.          printType(types[i], isDefinition);
74.       }
75.       if (types.length > 0) System.out.print(suf);
76.    }
77.
```


Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

Listing 12–4 GenericReflectionTest.java (continued)

```
78.    public static void printType(Type type, boolean isDefinition)
79.    {
80.       if (type instanceof Class)
81.       {
82.          Class t = (Class) type;
83.          System.out.print(t.getName());
84.       }
85.       else if (type instanceof TypeVariable)
86.       {
87.          TypeVariable t = (TypeVariable) type;
88.          System.out.print(t.getName());
89.          if (isDefinition)
90.             printTypes(t.getBounds(), " extends ", " & ", "", false);
91.       }
92.       else if (type instanceof WildcardType)
93.       {
94.          WildcardType t = (WildcardType) type;
95.          System.out.print("?");
96.          printTypes(t.getUpperBounds(), " extends ", " & ", "", false);
97.          printTypes(t.getLowerBounds(), " super ", " & ", "", false);
98.       }
99.       else if (type instanceof ParameterizedType)
100.      {
101.         ParameterizedType t = (ParameterizedType) type;
102.         Type owner = t.getOwnerType();
103.         if (owner != null)
104.         {
105.            printType(owner, false);
106.            System.out.print(".");
107.         }
108.         printType(t.getRawType(), false);
109.         printTypes(t.getActualTypeArguments(), "<", ", ", ">", false);
110.      }
111.      else if (type instanceof GenericArrayType)
112.      {
113.         GenericArrayType t = (GenericArrayType) type;
114.         System.out.print("");
115.         printType(t.getGenericComponentType(), isDefinition);
116.         System.out.print("[]");
117.      }
118.
119.   }
120. }
```


Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

API `java.lang.Class<T>` 1.0

- `TypeVariable[] getTypeParameters()` 5.0
  gets the generic type variables if this type was declared as a generic type, or an array of length 0 otherwise.
- `Type getGenericSuperclass()` 5.0
  gets the generic type of the superclass that was declared for this type, or `null` if this type is `Object` or not a class type.
- `Type[] getGenericInterfaces()` 5.0
  gets the generic types of the interfaces that were declared for this type, in declaration order, or an array of length 0 if this type doesn't implement interfaces.

API `java.lang.reflect.Method` 1.1

- `TypeVariable[] getTypeParameters()` 5.0
  gets the generic type variables if this method was declared as a generic method, or an array of length 0 otherwise.
- `Type getGenericReturnType()` 5.0
  gets the generic return type with which this method was declared.
- `Type[] getGenericParameterTypes()` 5.0
  gets the generic parameter types with which this method was declared. If the method has no parameters, an array of length 0 is returned.

API `java.lang.reflect.TypeVariable` 5.0

- `String getName()`
  gets the name of this type variable.
- `Type[] getBounds()`
  gets the subclass bounds of this type variable, or an array of length 0 if the variable is unbounded.

API `java.lang.reflect.WildcardType` 5.0

- `Type[] getLowerBounds()`
  gets the subclass (`extends`) bounds of this type variable, or an array of length 0 has no subclass bounds
- `Type[] getUpperBounds()`
  gets the superclass (`super`) bounds of this type variable, or an array of length 0 has no superclass bounds.

API `java.lang.reflect.ParameterizedType` 5.0

- `Type getRawType()`
  gets the raw type of this parameterized type.

Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

- `Type[] getActualTypeArguments()`
  gets the type parameters with which this parameterized type was declared.
- `Type getOwnerType()`
  gets the outer class type if this is an inner type, or `null` if this is a top-level type.

API **java.lang.reflect.GenericArrayType 5.0**

- `Type getGenericComponentType()`
  gets the generic component type with which this array type was declared.

You now know how to use generic classes and how to program your own generic classes and methods if the need arises. Just as importantly, you know how to decipher the generic type declarations that you may encounter in the API documentation and in error messages. For an exhaustive discussion of everything there is to know about Java generics, turn to Angelika Langer's excellent list of frequently (and not so frequently) asked questions at `http://angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html`.

In the next chapter, you will see how the Java collections framework puts generics to work.

Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14

Core Java™ Volume I—Fundamentals, Eighth Edition By Gary Cornell, Cay S. Horstmann
ISBN: 9780132354769 Publisher: Prentice Hall
Print Publication Date: 2007/09/14