

A Bestselling Hands-on Java Tutorial

3rd Edition
Covers J2SE 5.0
Includes CD-ROM



Learning Java™



O'REILLY®

Patrick Niemeyer & Jonathan Knudsen

Generics

It's been almost 10 years since the introduction of the Java programming language (and the first edition of this book). In that time, the Java language has matured and come into its own. But only with Java 5.0, the sixth major release of Java, did the core language itself change in a significant way. Yes, there were subtle changes and drop-ins over the years. Inner classes, added very early on, were important. But no language improvements prior to this point have affected all Java code or all Java developers in the way that Java 5.0 will.

Generics are about abstraction. Generics let you create classes and methods that work in the same way on different types of objects. The term “generic” comes from the idea that we'd like to be able to write general algorithms that can be broadly reused for many types of objects rather than having to adapt our code to fit each circumstance. This concept is not new; it is the impetus behind object-oriented programming itself. Java generics do not so much add new capabilities to the language as they make reusable Java code easier to write and easier to read.

Generics take reuse to the next level by making the *type* of the objects we work with an explicit parameter of the generic code. For this reason, generics are also referred to as *parameterized types*. In the case of a generic class, the developer specifies a type as a parameter (an argument) whenever she uses the generic type. The class is parameterized by the supplied type, to which the code adapts itself.

In other languages, generics are sometimes referred to as *templates*, which is more of an implementation term. Templates are like intermediate classes, waiting for their type parameters so that they can be used. Java takes a different path, which has both benefits and drawbacks that we'll describe in detail in this chapter.

There is much to say about Java generics and some of the fine points may be a bit obscure at first. But don't get discouraged. The vast majority of what you'll do with generics is easy and intuitive. The rest will come with a little patience and tinkering.

We begin our discussion with the most compelling case for generics: container classes and collections. Next, we take a step back and look at the good, bad, and

ugly of how Java generics work before getting into the details of writing generic classes. We then introduce generic methods, which intelligently infer their parameter types based upon how they are invoked. We conclude by looking at a couple of real-world generic classes in the Java API.

Containers: Building a Better Mousetrap

In an object-oriented programming language like Java, polymorphism means that objects are always to some degree interchangeable. Any child of a type of object can serve in place of its parent type and, ultimately, every object is a child of `java.lang.Object`, the object-oriented “Eve,” so to speak. It is natural, therefore, for the most general types of *containers* in Java to work with the type `Object` so that they can hold most anything. By containers, we mean classes that hold instances of other classes in some way. The Java Collections Framework is the best example of containers. A `List`, for example, holds an ordered collection of elements of type `Object`. A `Map` holds an association of key-value pairs, with the keys and values also being of the most general type, `Object`. With a little help from wrappers for primitive types, this arrangement has served us well. But (not to get too Zen on you), in a sense, a “collection of any type” is also a “collection of no type,” and working with `Objects` pushes a great deal of responsibility onto the user of the container.

It’s kind of like a costume party for objects where everybody is wearing the same mask and disappears into the crowd of the collection. Once objects are dressed as the `Object` type the compiler can no longer see the real types and loses track of them. It’s up to the user to pierce the anonymity of the objects later using a type cast. And like attempting to yank off a party-goer’s fake beard, you’d better have the cast correct or you’ll get an unwelcome surprise.

```
Date date = new Date();
List list = new ArrayList();
list.add( date );
...
Date firstElement = (Date)list.get(0); // Is the cast correct? Maybe.
```

The `List` interface has an `add()` method that accepts any type of `Object`. Here, we assigned an instance of `ArrayList`, which is simply an implementation of the `List` interface, and added a `Date` object. Is the cast in this example correct? It depends on what happens in the elided “...” period of time.

Can Containers Be Fixed?

It’s natural to ask if there is a way to make this situation better. What if we know that we are only going to put `Dates` into our list? Can’t we just make our own list that only accepts `Date` objects, get rid of the cast, and let the compiler help us again? The answer, surprisingly perhaps, is no. At least, not in a very satisfying way.

Our first instinct may be to try to “override” the methods of `ArrayList` in a subclass. But of course, rewriting the `add()` method in a subclass would not actually override anything; it would add a new *overloaded* method.

```
public void add( Object o ) { ... }  
public void add( Date d ) { ... } // overloaded method
```

The resulting object still accepts any kind of object—it just invokes different methods to get there.

Moving along, we might take on a bigger task. For example, we might write our own `DateList` class that does not extend `ArrayList` but delegates the guts of its methods to the `ArrayList` implementation. With a fair amount of tedious work, that would get us an object that does everything a `List` does but works with `Dates`. However, we’ve now shot ourselves in the foot because our container is no longer an implementation of `List` and we can’t use it interoperably with all of the utilities that deal with collections, such as `Collections.sort()`, or add it to another collection with the `Collection.addAll()` method.

To generalize, the problem is that instead of refining the behavior of our objects, what we really want to do is to change their contract with the user. We want to adapt their API to a more specific type and polymorphism doesn’t allow that. It would seem that we are stuck with `Objects` for our collections. But this is where generics come in.

Enter Generics

Generics in Java are an enhancement to the syntax of classes that allow us to specialize the class for a given type or set of types. A generic class requires one or more *type parameters* wherever we refer to the class type and uses them to customize itself.

If you look at the source or Javadoc for the `List` class, for example, you’ll see it defined something like this:

```
public class List< E > {  
    ...  
    public void add( E element ) { ... }  
    public E get( int i ) { ... }  
}
```

The identifier `E` between the angle brackets (`<>`) is a *type variable*. It indicates that the class `List` is generic and requires a Java type as an argument to make it complete. The name `E` is arbitrary, but there are conventions that we’ll see as we go on. In this case, the type variable `E` represents the type of elements we want to store in the list. The `List` class refers to the type variable within its body and methods as if it were a real type, to be substituted later. The type variable may be used to declare instance variables, arguments to methods, and the return type of methods. In this case, `E` is

used as the type for the elements we'll be adding via the `add()` method and the return type of the `get()` method. Let's see how to use it.

The same angle bracket syntax supplies the type parameter when we want to use the `List` type:

```
List<String> listOfStrings;
```

In this snippet, we declared a variable called `listOfStrings` using the generic type `List` with a type parameter of `String`. `String` refers to the `String` class, but we could have specialized `List` with any Java class type. For example:

```
List<Date> dates;  
List<java.math.BigDecimal> decimals;  
List<Foo> foos;
```

Completing the type by supplying its type parameter is called *instantiating the type*. It is also sometimes called *invoking the type*, by analogy with invoking a method and supplying its arguments. Whereas with a regular Java type, we simply refer to the type by name, a generic type must be instantiated with parameters wherever it is used.* Specifically this means we must instantiate the type everywhere types can appear: as the declared type of a variable (as shown in this code snippet), as the type of a method argument, as the return type of a method, or in an object allocation expression using the `new` keyword.

Returning to our `listOfStrings`, what we have now is effectively a `List` in which the type `String` has been substituted for the type variable `E` in the class body:

```
public class List< String > {  
    ...  
    public void add( String element ) { ... }  
    public String get( int i ) { ... }  
}
```

We have specialized the `List` class to work with elements of type `String` and *only* elements of type `String`. This method signature is no longer capable of accepting an arbitrary `Object` type.

`List` is just an interface. To use the variable, we'll need to create an instance of some actual implementation of `List`. As we did in our introduction, we'll use `ArrayList`. As before, `ArrayList` is a class that implements the `List` interface, but, in this case, both `List` and `ArrayList` are generic classes. As such, they require type parameters to instantiate them where they are used. Of course, we'll create our `ArrayList` to hold `String` elements to match our `List of Strings`:

```
List<String> listOfStrings = new ArrayList<String>();
```

* That is, unless you want to use a generic type in a nongeneric way. We'll talk about "raw" types later in this chapter.

There is no new syntax in this example. As always, the new keyword takes a Java type and parentheses with possible arguments for the class’s constructor. In this case, the type is `ArrayList<String>`—the generic `ArrayList` type instantiated with the `String` type.

We can now use our specialized `List` with strings. The compiler prevents us from even trying to put anything other than a `String` object (or a subtype of `String` if there were any) into the list and allows us to fetch them with the `get()` method without requiring any cast:

```
List<String> listOfStrings = new ArrayList<String>();
listOfStrings.add("eureka! ");
String s = listOfStrings.get(0); // "eureka! "

listOfStrings.add( new Date() ); // Compile-time Error!
```

Let’s take another example from the `Collections` API. The `Map` interface provides a dictionary-like mapping that associates key objects with value objects. Keys and values do not have to be of the same type. The generic `Map` interface requires two type parameters: one for the key type and one for the value type. The Javadoc looks like this:

```
public class Map< K, V > {
    ...
    public V put( K key, V value ) { ... } // returns any old value
    public V get( K key ) { ... }
}
```

We can make a `Map` that stores `Employee` objects by `Integer` “employee id” numbers like this:

```
Map< Integer, Employee > employees = new HashMap< Integer, Employee >();
Integer bobsId = ...;
Employee bob = ...;

employees.put( bobsId, bob );
Employee employee = employees.get( bobsId );
```

Here, we used `HashMap`, which is a generic class that implements the `Map` interface, and instantiated both types with the type parameters `Integer` and `Employee`. The `Map` now works only with keys of type `Integer` and holds values of type `Employee`.

The reason we used `Integer` here to hold our number is that the type parameters to a generic class must be class types. We can’t parameterize a generic class with a primitive type, such as `int` or `boolean`. Fortunately, autoboxing of primitives in Java 5.0 (see Chapter 5) makes it appear almost as if we can, by allowing us to use primitive types as if they were wrapper types:

```
employees.put( 42, bob );
Employee bob = employees.get( 42 );
```

Here, autoboxing converted the integer 42 to an `Integer` wrapper for us twice.

In Chapter 11, we'll see that all of the Java collection classes and interfaces have been made generic in Java 5.0. Furthermore, dozens of other APIs now use generics to let you adapt them to specific types. We'll talk about them as they occur throughout the book.

Talking About Types

Before we move on to more important things, we should say a few words about the way we describe a particular parameterization of a generic class. Since the most common and compelling case for generics is for container-like objects, it's common to think in terms of a generic type “holding” a parameter type. In our example, we called our `List<String>` a “list of strings” because, sure enough, that's what it was. Similarly, we might have called our employee map a “Map of employee ids to Employee objects.” However, these descriptions focus a little more on what the classes *do* than on the type itself. Take instead a single object container called `Trap<E>` that could be instantiated on an object of type `Mouse` or of type `Bear`, that is, `Trap<Mouse>` or `Trap<Bear>`. Our instinct is to call the new type a “mouse trap” or “bear trap.” Similarly, we could have thought of our list of strings as a new type: “string list” or our employee map as a new “integer employee object map” type. There is no harm in using whatever verbiage you prefer, but these latter descriptions focus more on the notion of the generic as a *type* and may help a little bit later when we have to discuss how generic types are related in the type system. There we'll see that the container terminology turns out to be a little counterintuitive.

In the following section, we'll carry on our discussion of generic types in Java from a different perspective. We've seen a little of what they can do; now we need to talk about how they do it.

“There Is No Spoon”

In the movie *The Matrix*,* the hero Neo is offered a choice. Take the blue pill and remain in the world of fantasy or take the red pill and see things as they really are. In dealing with generics in Java, we are faced with a similar ontological dilemma. We can go only so far in any discussion of generics before we are forced to confront the reality of how they are implemented. Our fantasy world is one created by the compiler to

* For those of you who might like some context for the title of this section, here is where it comes from:

Boy: Do not try and bend the spoon. That's impossible. Instead only try to realize the truth.

Neo: What truth?

Boy: There is no spoon.

Neo: There is no spoon?

Boy: Then you'll see that it is not the spoon that bends, it is only yourself.

—Wachowski, Andy and Larry. *The Matrix*. 1 videocassette (136 minutes). Warner Brothers, 1999.

make our lives writing code easier to accept. Our reality (though not quite the dystopian nightmare in the movie) is a harsher place, filled with unseen dangers and questions. Why don't casts and tests work properly with generics? Why can't I implement what appear to be two different generic interfaces in one class? Why is it that I can declare an array of generic types, even though there is no way in Java to create such an array?!? We'll answer these questions and more in this chapter, and you won't even have to wait for the sequel. Let's get started.

The design goals for Java generics were formidable: add a radical new syntax to the language that introduces parameterized types, safely, with no impact on performance and, oh, by the way, make it backward-compatible with all existing Java code and don't change the compiled classes in any serious way. It's actually fairly amazing that these conditions could be satisfied at all and no surprise that it took a while. But as always, the compromises lead to some headaches.

To accomplish this feat, Java employs a technique called *erasure*, which relates to the idea that since most everything we do with generics applies statically, at compile time, generic information does not really have to be carried over into the compiled classes. The generic nature of the classes, enforced by the compiler, can be "erased" in the compiled classes, allowing us to maintain compatibility with nongeneric code. While Java does retain information about the generic features of classes in the compiled form, this information is used mainly by the compiler. The Java runtime does not actually know anything about generics at all.

Erasure

Let's take a look at a compiled generic class: our friend, `List`. We can do this easily with the `javap` command:

```
% javap java.util.List

public interface java.util.List extends java.util.Collection{
    ...
    public abstract boolean add(java.lang.Object);
    public abstract java.lang.Object get(int);
}
```

The result looks exactly like it did prior to Java generics, as you can confirm with any older version of the JDK. Notably, the type of elements used with the `add()` and `get()` methods is `Object`. Now, you might think that this is just a ruse and that when the actual type is instantiated, Java will create a new version of the class internally. But that's not the case. This is the one and only `List` class and it is the actual runtime type used by all parameterizations of `List`, for example, `List<Date>` and `List<String>`, as we can confirm:

```
List<Date> dateList = new ArrayList<Date>();
System.out.println( dateList instanceof List ); // true!
```

But our generic `dateList` clearly does not implement the `List` methods just discussed:

```
dateList.add( new Object() ); // Compile-time Error!
```

This illustrates the somewhat schizophrenic nature of Java generics. The compiler believes in them, but the runtime says they are an illusion. What if we try something a little more sane looking and simply check that our `dateList` is a `List<Date>`:

```
System.out.println( dateList instanceof List<Date> ); // Compile-time Error!  
// Illegal, generic type for instanceof
```

This time the compiler simply puts its foot down and says, “no.” You can’t test for a generic type in an `instanceof` operation. Since there are no actual differentiable classes for different parameterizations of `List` at runtime, there is no way for the `instanceof` operator to tell the difference between one incarnation of `List` and another. All of the generic safety checking was done at compile time and now we’re just dealing with a single actual `List` type.

What has really happened is that the compiler has erased all of the angle bracket syntax and replaced the type variables in our `List` class with a type that can work at runtime with any allowed type: in this case, `Object`. We would seem to be back where we started, except that the compiler still has the knowledge to enforce our usage of the generics in the code at compile time and it can, therefore, handle the cast for us. If you decompile a class using a `List<Date>` (the `javap` command with the `-c` option shows you the bytecode, if you dare), you will see that the compiled code actually contains the cast to `Date`, even though we didn’t write it ourselves.

We can now answer one of the questions we posed at the beginning of the section (“Why can’t I implement what appear to be two different generic interfaces in one class?”). We can’t have a class that implements two different generic `List` instantiations because they are really the same type at runtime and there is no way to tell them apart:

```
public abstract class DualList implements List<String>, List<Date> { }  
// Error: java.util.List cannot be inherited with different arguments:  
//   <java.lang.String> and <java.util.Date>
```

Raw Types

Although the compiler treats different parameterizations of a generic type as truly different types (with different APIs) at compile time, we have seen that only one real type exists at runtime. For example, the class of `List<Date>` and `List<String>` share the plain old Java class `List`. `List` is called the *raw type* of the generic class. Every generic has a raw type. It is the degenerate, “plain” Java form in which all of the generic type information has been removed and the type variables replaced by some general Java type like `Object`.

Java 5.0 has been carefully arranged such that the raw type of all of the generic classes works out to be exactly the same as the earlier, nongeneric types. So the raw type of a `List` in Java 5.0 is the same as the old, nongeneric `List` type that has been around since JDK 1.2. Since the vast majority of current Java code in existence today

does not (and may never) use generics, this type equivalency and compatibility is very important.

It is still possible to use raw types in Java 5.0 just as before. The only difference is that the Java 5.0 compiler generates a new type of warning wherever they are used in an “unsafe” way. For example:

```
// nongeneric Java code using the raw type, same as always
List list = new ArrayList(); // assignment ok
list.add("foo"); // unchecked warning on usage of raw type
```

This snippet uses the raw `List` type just as any good old-fashioned Java code prior to Java 5.0 would have. The difference is that now the Java compiler issues an *unchecked warning* about the code if we attempt to insert an object into the list.

```
% javac MyClass.java
Note: MyClass.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

The compiler instructs us to use the `-Xlint:unchecked` option to get more specific information about the locations of unsafe operations:

```
% javac -Xlint:unchecked MyClass.java
warning: [unchecked] unchecked call to add(E) as a member of the raw type java.util.
List: list.add("foo");
```

Note that creating and assigning the raw `ArrayList` do not generate a warning. It is only when we try to use an “unsafe” method (one that refers to a type variable) that we get the warning. This means that it’s still okay to use older style nongeneric Java APIs that work with raw types. We only get warnings when we do something unsafe in our own code.

One more thing about erasure before we move on. In the previous examples, the type variables were replaced by the `Object` type, which could represent any type applicable to the type variable `E`. Later we’ll see that this is not always the case. We can place limitations or *bounds* on the parameter types, and, when we do, the compiler can be more restrictive about the erasure of the type. We’ll explain in more detail later after we discussed bounds, but, for example:

```
class Bounded< E extends Date > {
    public void addElement( E element ) { ... }
}
```

This parameter type declaration says that the element type `E` may be any subtype of the `Date` type. In this case, the erasure of the `addElement()` method can be more restrictive than `Object`, and the compiler uses `Date`:

```
public void addElement( Date element ) { ... }
```

`Date` is called the *upper bound* of this type, meaning that it is the top of the object hierarchy here and the type can be instantiated only on type `Date` or on “lower” (more derived) types.

Now that we have a handle on what generic types really are, we can go into a little more detail about how they behave.

Parameterized Type Relationships

We know now that parameterized types share a common, raw type. This is why our parameterized `List<Date>` is just a `List` at runtime. In fact, we can assign any instantiation of `List` to the raw type if we want:

```
List list = new ArrayList<Date>();
```

We can even go the other way and assign a raw type to a specific instantiation of the generic type:

```
List<Date> dates = new ArrayList(); // unchecked warning
```

This statement generates an unchecked warning on the assignment, but thereafter the compiler trusts you that the list contained only `Dates` prior to the assignment. It is also permissible, but pointless, to perform a cast in this statement. We'll talk about casting to generic types a bit later.

Whatever the runtime types, the compiler is running the show and does not let us assign things that are clearly incompatible:

```
List<Date> dates = new ArrayList<String>(); // Compile-time Error!
```

Of course, the `ArrayList<String>` does not implement the methods of `List<Date>` conjured by the compiler, so these types are incompatible.

But what about more interesting type relationships? The `List` interface, for example, is a subtype of the more general `Collection` interface. Is a particular instantiation of the generic `List` also assignable to some instantiation of the generic `Collection`? Does it depend on the type parameters and their relationships? Clearly, a `List<Date>` is not a `Collection<String>`. But is a `List<Date>` a `Collection<Date>`? What about a `List<Date>` being a `Collection<Object>`?

We'll just blurt out the answer first, then walk through it and explain. The rule is that for the simple types of generic instantiations we've discussed so far in this chapter *inheritance applies only to the "base" generic type and not to the parameter types*. Furthermore, assignability applies only when the two generic types are instantiated on *exactly the same parameter type*. In other words, there is still one-dimensional inheritance, following the base generic class type, but with the catch that the parameter types must be identical.

For example, recalling that a `List` is a type of `Collection`, we can assign instantiations of `List` to instantiations of `Collection` when the type parameter is exactly the same:

```
Collection<Date> cd;  
List<Date> ld = new ArrayList<Date>();  
cd = ld; // OK!
```

This code snippet says that a `List<Date>` is a `Collection<Date>`—pretty intuitive. But trying the same logic on a variation in the parameter types fails:

```
List<Object> lo;
List<Date> ld = new ArrayList<Date>();
lo = ld; // Compile-time Error! Incompatible types.
```

Although our intuition tells us that the `Dates` in that `List` could all live happily as `Objects` in a `List`, the assignment is an error. We’ll explain precisely why in the next section, but for now just note that the type parameters are not exactly the same and, as we’ve said, there is no inheritance relationship among the parameter types in generics. This is a case where thinking of the instantiation in terms of types and not in terms of what they do helps. These are not really a “list of dates” and a “list of objects” but more like a `DateList` and an `ObjectList`, the relationship of which is not immediately obvious.

Try to pick out what’s okay and what’s not okay in the following example:

```
Collection<Number> cn;
List<Integer> li = new ArrayList<Integer>();
cn = li; // Compile-time Error! Incompatible types.
```

It is possible for an instantiation of `List` to be an instantiation of `Collection`, but only if the parameter types are exactly the same. Inheritance doesn’t follow the parameter types and this example fails.

One more thing: earlier we mentioned that this rule applies to the simple types of instantiations we’ve discussed so far in this chapter. What other types are there? Well, the kinds of instantiations we’ve seen so far where we plug in an actual Java type as a parameter are called *concrete type instantiations*. Later we’ll talk about *wild-card instantiations*, which are akin to mathematical set operations on types. We’ll see that it’s possible to make more exotic instantiations of generics where the type relationships are actually two-dimensional, depending both on the base type and the parameterization. But don’t worry, this doesn’t come up very often and is not as scary as it sounds.

Why Isn’t a `List<Date>` a `List<Object>`?

It’s a reasonable question. Even with our brains thinking of arbitrary `DateList` and `ObjectList` types, we can still ask why they couldn’t be assignable. Why shouldn’t we be able to assign our `List<Date>` to a `List<Object>` and work with the `Date` elements as `Object` types?

The reason gets back to the heart of the rationale for generics that we discussed in the introduction: changing APIs. In the simplest case, supposing an `ObjectList` type extends a `DateList` type, the `DateList` would have all of the methods of `ObjectList` and we could still insert `Objects` into it. Now, you might object that generics let us change the APIs, so that doesn’t apply anymore. That’s true, but there is a bigger

problem. If we could assign our `Datelist` to an `ObjectList` variable, we would have to be able to use `Object` methods to insert elements of types other than `Date` into it. We could *alias* the `Datelist` as an `ObjectList` and try to trick it into accepting some other type:

```
Datelist datelist = new Datelist();
ObjectList objectList = datelist;
objectList.add( new Foo() ); // should be runtime error!
```

We'd expect to get a runtime error when the actual `Datelist` implementation was presented with the wrong type of object. And therein lies the problem. Java generics have no runtime representation. Even if this functionality were useful, there is no way with the current scheme for Java to know what to do at runtime. Another way to look at it is that this feature is simply dangerous because it allows for an error at runtime that couldn't be caught at compile time. In general, we'd like to catch type errors at compile time. By disallowing these assignments, Java can guarantee that your code is typesafe if it compiles with no unchecked warnings.

Actually, that last sentence is not entirely true, but it doesn't have to do with generics; it has to do with arrays. If this all sounds familiar to you, it's because we mentioned it already in relation to Java arrays. Array types have an inheritance relationship that allows this kind of aliasing to occur:

```
Date [] dates = new Date[10];
Object [] objects = dates;
objects[0] = "not a date"; // Runtime ArrayStoreException!
```

However, arrays have runtime representations as different classes and they check themselves at runtime, throwing an `ArrayStoreException` in just this case. So, in theory, Java code is not guaranteed typesafe by the compiler if you use arrays in this way.

Casts

We've now talked about relationships between generic types and even between generic types and raw types. But we haven't brought up the concept of a cast yet. No cast was necessary when we interchanged generics with their raw types. Instead, we just crossed a line that triggers unchecked warnings from the compiler:

```
List list = new ArrayList<Date>();
List<Date> dl = list; // unchecked warning
```

Normally, we use a cast in Java to work with two types that could be assignable. For example, we could attempt to cast an `Object` to a `Date` because it is plausible that the `Object` is a `Date` value. The cast then performs the check at runtime to see if we are correct. Casting between unrelated types is a compile-time error. For example, we can't even try to cast an `Integer` to a `String`. Those types have no inheritance relationship. What about casts between compatible generic types?

```
Collection<Date> cd = new ArrayList<Date>();
List<Date> ld = (List<Date>)cd; // Ok!
```

This code snippet shows a valid cast from a more general `Collection<Date>` to a `List<Date>`. The cast is plausible here because a `Collection<Date>` is assignable from and could actually be a `List<Date>`. Similarly, the following cast catches our mistake where we have aliased a `TreeSet<Date>` as a `Collection<Date>` and tried to cast it to a `List<Date>`:

```
Collection<Date> cd = new TreeSet<Date>();
List<Date> ld = (List<Date>)cd; // Runtime ClassCastException!
ld.add( new Date() );
```

There is one case where casts are not effective with generics, however, and that is when we are trying to differentiate the types based on their parameter types:

```
Object o = new ArrayList<String>();
List<Date> ldfo = (List<Date>)o; // unchecked warning, ineffective
Date d = ldfo.get(0); // unsafe at runtime, implicit cast may fail
```

Here, we aliased an `ArrayList<String>` as a plain `Object`. Next, we cast it to a `List<Date>`. Unfortunately, Java does not know the difference between a `List<String>` and a `List<Date>` at runtime, so the cast is fruitless. The compiler warns us of this by generating an unchecked warning at the location of the cast; we should be aware that we might find out later when we try to use the cast object that it is incorrect. Casts are ineffective at runtime because of erasure and the lack of type information.

Writing Generic Classes

Now that we have (at least some of) the “end user” view of generics, let’s try writing a few classes ourselves. In this section, we’ll talk about how type variables are used in the definition of generic classes, where they may appear, and some of their limitations. We’ll also talk about subclassing generic types.

The Type Variable

We’ve already seen the basics of how type variables are used in the declaration of a generic class. One or more type variables are declared in the angle bracket (`<>`) type declaration and used throughout the body and instance methods of the class. For example:

```
class Mouse { }
class Bear { }

class Trap< T >
{
    T trapped;

    public void snare( T trapped ) { this.trapped = trapped; }
    public T release() { return trapped; }
}
```

```
// usage
Trap<Mouse> mouseTrap = new Trap<Mouse>();
mouseTrap.snare( new Mouse() );
Mouse mouse = mouseTrap.release();
```

Here, we created a generic `Trap` class that can hold any type of object. We used the type variable `T` to declare an instance variable of the parameter type as well as in the argument type and return type of the two methods.

The scope of the type variable is the instance portion of the class, including methods and any instance initializer blocks. The static portion of the class is not affected by the generic parameterization, and type variables are not visible in static methods or static initializers. As you might guess, just as all instantiations of the generic type have only one actual class (the raw type), they have only one, shared, static context as well. You cannot even invoke a static method through a parameterized type. You must use the raw type or an instance of the object.

The type variable can also be used in the type instantiation of other generic types used by the class. For example, if we wanted our `Trap` to hold more than one animal, we could create a `List` for them like so:

```
List<T> trappedList = new ArrayList<T>();
```

Just to cover all the bases, we should mention that instantiations of generic types on the type variable act just like any other type and can serve in all the places that other instantiations of a type can. For example, a method in our class can take a `List<T>` as an argument:

```
public void trapAll( List<T> list ) { ... }
```

The effective type of the `trapAll()` method in a `Trap<Mouse>` is then simply:

```
trapAll( List<Mouse> list ) { ... }
```

We should note that this is *not* what we mean by the term “generic method.” This is just a regular Java method that happens to take a generic type as an argument. We’ll talk about real generic methods, which can infer their types from arguments and assignment contexts later in this chapter. A type variable can also be used to parameterize a generic parent class, as we’ll see in the next section.

Subclassing Generics

Generic types can be subclassed just like any other class, by either generic or nongeneric child classes. A nongeneric subclass must extend a particular instantiation of the parent type, filling in the required parameters to make it concrete:

```
class Datelist extends ArrayList<Date> { }

Datelist datelist = new Datelist();
datelist.add( new Date() );
List<Date> ld = datelist;
```

Here, we have created a nongeneric subclass, `DateList`, of the concrete generic instantiation `ArrayList<Date>`. The `DateList` is a type of `ArrayList<Date>` and inherits the particular instantiation of all of the methods, just as it would from any other parent. We can even assign it back to the parent type if we wish, as shown in this example.

A generic subtype of a generic class may extend either a concrete instantiation of the class, as in the previous example, or it may share a type variable that it “passes up” to the parent upon instantiation:

```
class AdjustableTrap< T > extends Trap< T > {
    public void setSize( int i ) { ... }
}
```

Here, the type variable `T` used to instantiate the `AdjustableTrap` class is passed along to instantiate the base class, `Trap`. When the user instantiates the `AdjustableTrap` on a particular parameter type, the parent class is instantiated on that type as well.

Exceptions and Generics

Types appear in the body of classes in another place—the throws clauses of methods. Type variables may be used to define the type of exceptions thrown by methods, but to do so we need to introduce the concept of bounds. We cover bounds more in the next section, but in this case, the usage is really simple. We just need to ensure that the type variable we want to use as our exception type is actually a type of `Throwable`. We can do that by adding an `extends` clause to the declaration of our type variable, like this:

```
< T extends Throwable >
```

Here is an example class, parameterized on a type that must be a kind of `Throwable`. Its `test()` method accepts an instance of that kind of object and throws it as a checked exception:

```
ExceptionTester< T extends Throwable > {
    public void test( T exception ) throws T { throw exception; }
}

try {
    new E<ClassNotFoundException>().test(
        new ClassNotFoundException() );
} catch ( ClassNotFoundException e ) { ... }
```

The addition of the bound imposes the restriction that the parameter type used to instantiate the class, `T`, must be a type of `Throwable`. And we referenced the type `T` in the `throws` clause. So, an `ExceptionTester<ClassNotFoundException>` can throw a `ClassNotFoundException` from its `test()` method. Note that this is a checked exception and that has not been lost on the compiler. The compiler enforces the checked exception type that it just applied.

No generic Throwables

We saw that a type variable can be used to specify the type of `Throwable` in the `throws` clause of a method. Perhaps ironically, however, we cannot use generics to create new types of exceptions. No generic subtypes of `Throwable` are allowed. If you think about this for a moment, you'll see that to be useful, generic `Throwables` would require `try/catch` blocks that can differentiate instantiations of `Throwable`. And since (once again) there is no runtime representation of generics, this isn't possible with erasure.

Parameter Type Limitations

We have seen the parameter types (type variables) of a generic class used to declare instance variables, method arguments, and return types as well as “passed along” to parameterize a generic superclass. One thing that we haven't talked about is the question of how or whether we can use the type variable of a generic class to construct instances of the parameter type or work with objects of the type in other concrete ways. We deliberately avoided this issue in our previous “exception tester” example by simply passing our exception object in as an argument. Could we have done away with this argument? The answer, unfortunately, is that due to the limitations of erasure, there really is no parameter-type information to work with at runtime. In this section, we'll look at this problem and a workaround.

Since the type variable `T` has faithfully served as our parameter type everywhere else, you might imagine that we could use it to construct an instance of `T` using the `new` keyword. But we can't:

```
T element = new T(); // Error! Invalid syntax.
```

Remember that all type information is erased in the compiled class. The raw type does not have any way of knowing the type of object you want to construct at runtime. Nor is there any way to get at the `Class` of the parameter type through the type variable, for the same reason. So reflection won't help us here either. This means that, in general, generics are limited to working with parameter types in relatively hands-off ways (by reference only). This is one reason that generics are more useful for containers than for some other applications. This problem comes up often though and there is a solution, although it's not quite as elegant as we'd like.

Using `Class<T>`

The only real way to get the type information that we need at runtime is to have the user explicitly pass in a `Class` reference, generally as one of the arguments to a method. Then we can explicitly refer to the class using reflection and create instances or do whatever else is necessary. This may sound like a really bad solution, without much type safety and placing a big burden on the developer to do the right thing. Fortunately, we can use a trick of generics to enforce this contract with the user and

make it safe. The basic idea again is to have one of our methods accept the `Class` of the parameter type so that we can use it at runtime. Following our “exception tester” example:

```
public void test( Class type ) throws T { ... }
```

This isn’t much better than it was before. Specifically, it doesn’t guarantee that the `Class` type passed to the method will match the parameterized type of the class (used in the `throws` clause here).

Fortunately, Java 5.0 introduced some changes in the `Class` class. `Class` is, itself, now a generic type. Specifically, all instances of the `Class` class created by the Java VM are instantiated with their own type as a parameter. The class of the `String` type, for example, is now `Class<String>`, not just some arbitrary instance of the raw `Class` type that happens to know about strings.

This has two side effects. First, we can specify a particular instantiation of `Class` using the parameter type in our class. And second, since the `Class` class is now generic, all of the reflective and instance creation methods can be typed properly and no longer require casts, so we can write our `test()` method like this:

```
public void test( Class<T> type ) throws T {  
    throw type.newInstance();  
}
```

The only `Class` instance that can be passed to our `test()` method now is `Class<T>`, the `Class` for the parameter type, `T`, on which we instantiated `ExceptionTester`. So, although the user still has the burden of passing in this seemingly extraneous `Class` argument, at least the compiler will ensure that we do it and do it correctly:

```
ExceptionTester<ArithmeticException> et =  
    new ExceptionTester<ArithmeticException>();  
  
et.test( ArithmeticException.class ); // no other .class will work
```

In this code snippet, attempting to pass any other `Class` argument to the `test()` method generates a compile-time error.

Bounds

In the process of discussing generics, we’ve already had to mention bounds a few times. A bound is a constraint on the type of a type parameter. Bounds use the `extends` keyword and some new syntax to limit the parameter types that may be applied to a generic type. In the case of a generic class, the bounds simply limit the type that may be supplied to instantiate it.

A type variable may extend a class or interface type, meaning that its instantiation must be of that type or a subtype:

```
class EmployeeList< T extends Employee > { ... }
```

Here, we made a generic `EmployeeList` type that can be instantiated only with `Employee` types. We could further require that the `Employee` type implement one or more interfaces using the special `&` syntax:

```
class EmployeeList< T extends Employee & Ranked & Printable > { ... }
```

The order of the `&` interface bounds is not significant, but only one class type can be specified and if there is one, it must come first. When a type has no specific bounds, the bound `extends Object` is implicit.

By applying bounds to our type, we not only limit the instantiations of the generic class but we make the type arguments more useful. Now that we know our type must extend some type or implement some set of interfaces, we can use variables and arguments declared with `T` by those other type names. Here is a somewhat contrived extension of our previous example:

```
class EmployeeList< T extends Employee & Ranked & Printable >
{
    Ranked ranking;
    List<Printable> printList = new ArrayList<Printable>();

    public void addEmployee( T employee ) {
        this.ranking = employee; // T as Ranked
        printList.add( employee ); // T as Printable
    }
}
```

Type variables can also refer to other type variables within the type declaration:

```
class Foo <A, B extends A> { ... }
```

We'll see a particularly vicious example of this later when we talk about the definition of the `Enum` class. We'll also see a more convenient technique for declaring how individual elements of a generic class relate to the parameter type when we cover wildcards in the next section.

Erasure and Bounds (Working with Legacy Code)

We mentioned earlier in our discussion of erasure that the resulting type used in place of the type parameter in the raw type for the generic class is the bound of the type variable. Specifically, we have seen many generics with no explicit bounds that defaulted to a bound of type `Object`. We also showed a quick example of a type that imposed a bound of `extends Date` and said that the type of its methods would be `Date` instead of `Object`. We can now be a little more specific.

The type after erasure used for the parameter type of a generic class is the *leftmost bound*. That is, the first bound specified after `extends` becomes the type used in the erasure. This implies that if the type extends a class type, it is always the erased type because it must always come first. But if the type extends only interface types, the choice is up to us. This fine point is important for backward compatibility with nongeneric

code. Often when creating generic versions of nongeneric APIs, we have the opportunity to “tighten up” the specification a bit. Being aware of the leftmost bound gives us a way to explicitly control the type of the erased class. For example, suppose we create a generic `List` class that we only want instantiated on `Listable` objects, but we’d prefer not to change the API of our old `List` class, which accepted `Object` type elements. Our initial attempt:

```
class List< E extends Listable > { ... }
```

produces a raw type that accepts only `Listable`. However, we can insert a somewhat gratuitous additional type, `Object`, as the leftmost bound in order to get back our old API, without changing the new generic bounds:

```
class List< E extends Object & Listable > { ... }
```

Inserting `Object` doesn’t change the actual bounds of the generic class but does change the erased signature.

Wildcards

We mentioned earlier that the kinds of generic type instantiations discussed so far in this chapter have all been concrete type instantiations. We described this as meaning that all of the parameter arguments are real Java types. For example, `List<String>` and `List<Date>` are instantiations of the generic `List` class with the concrete types `String` and `Date`. Now we’re going to look at another kind of generic type instantiation: *wildcard instantiation*.

As we’ll see in this section, wildcards are Java’s way of introducing polymorphism into the type parameter portion of the generic equation. A wildcard instantiation uses a question mark (?) in place of an actual type parameter at instantiation time and denotes that the type can be assigned any of a range of possible instantiations of the generic type. The ? wildcard by itself is called the *unbounded wildcard* and denotes that any type instantiation is acceptable (assignable to the type).

```
List<?> anyInstantiationOfList = new ArrayList<Date>();  
anyInstantiationOfList = new ArrayList<String>(); // another instantiation
```

In this snippet, we declared a variable `anyInstantiationOfList` whose type is the unbounded wildcard instantiation of the generic `List` type. (What a mouthful.) This means that the type we instantiated can be assigned any particular concrete instantiation of the `List` type, whether `Dates`, `Strings`, or `Foos`. Here, we assigned it a `List<Date>` first and, subsequently, a `List<String>`.

A Supertype of All Instantiations

The unbounded wildcard instantiation is a kind of supertype of all of these concrete instantiations. In contrast to the generic type relationships that we saw earlier, which followed only raw, “base” generic types, wildcards let us implement polymorphism

on the parameter types. The unbounded wildcard is to generic type parameters what the `Object` type is to regular Java types: a supertype of everything.

```
// A List<Object> is not a List<Date>!
List<Object> objectList = new ArrayList<Date>() // Error!
```

```
// A List<?> can be a List<Date>
List<?> anyList = new ArrayList<Date>(); // Yes!
```

We are reminded in this example that `List<Object>` is not a `List<Date>`; polymorphism doesn't flow that way with generic instantiations of concrete types. But `List<?>`, the unbounded wildcard instantiation, can be assigned any instantiation of `List`. As we go on, we'll see that wildcards add a new dimension to the assignability of generic types.

Bounded Wildcards

A *bounded wildcard* is a wildcard that uses the `extends` keyword just as a type variable would, to limit the range of assignable types. For example:

```
List<? extends Date> dateInstantiations = new ArrayList<Date>();
dateInstantiations = new ArrayList<MyDate>(); // another instantiation
```

Our `dateInstantiations` variable is limited to holding instantiations of `List` on parameter types of `Date` and its subclasses. So, we can assign it a `List<Date>` or a `List<MyDate>`. In the same way that the unbounded wildcard serves as a superclass for all instantiations of a generic type, bounded wildcards create more limited super-types covering a narrower range of instantiations. In this case, our wildcard instantiation, `List<? extends Date>`, is the supertype of all instantiations of `List` on `Date` types. As with type parameter bounds, the bound `Date` is called the upper bound of the type.

Wildcard bounds may extend interfaces as well as use the `&` syntax to add interface requirements to the bound:

```
Trap< ? extends Catchable & Releaseable > trap;
```

In this case, the instantiation serves as a supertype of the set of instantiations on types implementing both the `Catchable` and `Releaseable` interfaces.

Thinking Outside the Container

Let's be clear about what the wildcard means in the context of a container type such as `List`. The unbounded wildcard instantiation may be assigned any type instantiation, but it does ultimately refer to *some particular type instantiation*. A wildcard instantiation serves as the type of a variable, and that variable eventually holds some actual concrete instantiation of the generic type:

```
List<?> someInstantiationOfList;
someInstantiationOfList = new ArrayList<Date>();
someInstantiationOfList = new ArrayList<String>();
```

In this example, our `List<?>` variable is either a `List<String>` or a `List<Date>`. It is *not* some new kind of `List` that can hold either `String` or `Date` elements.

In the same way, a wildcard with bounds ultimately holds one of the concrete instantiations assignable to its bounds. Imagine for a moment that we have a private class `Foo` with only one subclass `Bar` and no others. The expression `Collection<? extends Foo>` in this case means the set of two possibilities: either `Collection<Foo>` or `Collection<Bar>`. That is, either a `Collection` of elements with a common supertype of `Foo` or a collection of elements with a common supertype of `Bar`. Again, the wildcard instantiation matches either of those generic type instantiations. It does *not* create a new type of collection that can contain either `Foo`s or `Bar`s. (That is actually the job of `Collection<Foo>`, which can contain both `Foo` and `Bar` elements.)

For this reason, wildcard type instantiations are valid types for referencing an object, but they cannot be used as the type to create an instance of an object. In general, you cannot use a wildcard type with the `new` keyword to allocate an object instance because the wildcard denotes one or a possible set of objects. It doesn't make sense.

Lower Bounds

We saw the `extends` construct used to specify an upper bound for both type variables and wildcard instantiations. It implies a type that is “at the top” of the object hierarchy for the bound. Wildcard instantiations actually allow another type of bound called a *lower bound* as well. A lower bound is specified with the keyword `super` and, as you might guess, requires that instantiations be of a certain type or any of its supertypes, up to `Object`. For example:

```
List< ? super MyDate > listOfAssignableFromMyDate;  
listOfAssignableFromMyDate = new ArrayList<MyDate>();  
listOfAssignableFromMyDate = new ArrayList<Date>();  
listOfAssignableFromMyDate = new ArrayList<Object>();
```

This wildcard instantiation creates a type that can hold any instantiation of `List` on the type `MyDate` or any of its supertypes. In our example world, that means the wildcard type can be assigned one of only three types: `List<MyDate>`, `List<Date>`, or `List<Object>`. Here, we have cut off the object inheritance hierarchy after three generations. No further subclasses of `MyDate` can be used.

As we hinted in the example, it may help to read `? super MyDate` as “Assignable from `MyDate`.” Lower bounds are useful for cases where we want to be sure that a particular container instantiation can hold a particular element type, without limiting it to just the specific type of the element. We'll show a good example of this when we talk about generic methods later. For now, just try to digest this as complementary to upper bounds.

One last thing about lower bounds: only the wildcard instantiation syntax can use the `super` keyword to refer to lower bounds. Bounds of type variables in generic

class declarations cannot have lower bounds. Erasure replaces all references to the type variables with their upper bounds, so runtime types have no way to enforce the contract.

Reading, Writing, and Arithmetic

We've glossed over an important issue so far in our discussion of wildcard types: namely, how can we use them? What kinds of types does the compiler enforce for variables and arguments which referred to the type variables in the generic class? For example, if we have a `List<?>` list of any instantiation type, what are the rules about putting objects into it and getting them back out? What is their type?

We have to take the two cases separately. Drawing on the analogy of a container, we'll call getting a return value from a method on an object as a specific type *reading the object as a type*. Conversely, we'll call passing arguments of a specific type to methods of the object *writing the object as a type*. So, for example, a `List<Date>` can be read and written as the `Date` type and a `Trap<Mouse>` has methods that can be read and written as the `Mouse` type.

To be more precise, though, we should say that `List<Date>` can be read as the `Date` type but can be written as any subtype of `Date`. After all, we could add a `MyDate` to a `List<Date>`. Let's look now at the wildcard instantiation `List< ? extends Date >`. We know it holds an instantiation of the `List` type on some type of `Date`. What more can we say about the elements of such a `List`, which could hold *any* instantiation of the `Date` type? Well, the elements will always be subtypes of `Date`. This means that, at a minimum, we should be able to read the object through our wildcard type as type `Date`:

```
List< ? extends Date > someDateList = new ArrayList<MyDate>();  
...  
Date date = someDateList.get( 0 ); // read as Date
```

The compiler lets us assign the value directly to a `Date` because it knows that whatever the instantiation of the `List`, the elements must be a subtype of `Date`. (Of course, we could have read the object as type `Object` or any supertype of `Date` if we'd wanted as well.)

But what about going the other way and writing? If `someDateList` could be an instantiation of `List` on any subclass of `Date`, how can we know what type of objects to write to it? (How can we safely call its `add()` method?) The answer is that we can't. Since we don't know the correct type, the compiler won't let us write anything to the `List` through our wildcard instantiation of the type:

```
List< ? extends Date > someDateList = new ArrayList<MyDate>();  
someDateList.add( new Date() ); // Compile-time Error!  
someDateList.add( new MyDate() ); // Compile-time Error!
```

Another way to put this is that since our wildcard instantiation has an upper bound of `Date`, we can read the type as `Date`. We'll reiterate that in the form of a rule in a moment.

Recall that an unbounded wildcard is really just a wildcard with a bound of type `Object` `<? extends Object>`. Obviously, even an unbounded wildcard instantiation holds objects that can be assigned to `Object`, so it's okay to read an unbounded wildcard as the `Object` type:

```
List<?> someList = new ArrayList<String>();
...
Object object = someList.get( 0 ); // read as Object
```

But, of course, we cannot know the actual type of the elements, so we cannot write to the list through our unbounded wildcard type.

What about lower bounds? Well, the situation is neatly reversed with respect to reading and writing. Since we know that the elements of any instantiation matching our lower bounded wildcard must be a supertype of the lower bound, we can write to the object as the lower bound type through our wildcard:

```
List< ? super MyDate > listAssignableMyDate = new ArrayList<Date>();
listAssignableMyDate.add( new MyDate() );
listAssignableMyDate.add( new Date() ); // Compile-time Error!
```

But not knowing what supertype of `MyDate` the elements are, we cannot read the list as any specific type. Of course, the `List` must still hold some type of `Object`, so we can always read the lower bounded list as type `Object` through the wildcard. The type `Object` is the default upper bound:

```
Object obj = listAssignableMyDate.get( 0 ); // read as Object
```

Whew. Well, having gone through that explanation, we can now sum it up concisely in an easy-to-remember rule:

Wildcard instantiations of generic types can be read as their upper bound and written as their lower bound.

To elaborate: all wildcard instantiations have an upper bound of `Object` even if none other is specified, so all wildcard instantiations can at least be read as type `Object`. But not all wildcards have a lower bound. Only those using the `super` construct have a lower bound and so only those wildcard instantiations can be written as a type more specific than `Object`.

<?>, <Object>, and the Raw Type

We've covered a lot of ground and the semantics can be a bit hard to follow. Let's exercise our knowledge by reviewing a few cases that may or may not have similarities.

Natural questions to ask are, What good is the unbounded wildcard anyway? Why not just use the raw type? How do unbounded wildcard instantiation and raw types

compare? The first difference is that the compiler will issue unchecked warnings when we use methods of the raw type. But that's superficial. Why is the compiler warning us? It's because it cannot stop us from abusing our raw type by foisting the wrong type of objects on it. Using an unbounded wildcard is like putting on boxing gloves and saying that we want to play by the rules. Doing so comes at a cost. The compiler guarantees that we are safe by allowing us only the operations that it knows are safe, namely, reading as type `Object` (the upper bound of everything). The compiler does not let us write to an unbounded wildcard at all. So why use the unbounded wildcard? To play by the rules of generics and guarantee that we don't do anything unsafe.

Next, we can knock down any notion that an unbounded wildcard instantiation is similar to an instantiation on the type `Object`. Remember that a `List<?>` holds *some* instantiation of `List`. It could be a `List<Date>` for all we know. But a `List<Object>` is actually a list that holds concrete `Object` types. The `List<Object>` can be read and written as `Object`. The `List<?>` can only be read (not written) and only read as `Object` in a degenerate sense. The elements of `List<?>` are actually all of some unknown type. The elements of the unknown type list all have a common supertype that could be `Object` but could be some other common type more restrictive than `Object`. The knowledge of what "could be" in the `List<?>` doesn't do much for us in practice but means something completely different from `List<Object>`.

Finally, let's round out the comparisons by asking how `List<Object>` and the raw type compare. Now we're onto something. In fact, the raw type after erasure is effectively `List<Object>` as you'll recall. But in this case, we're telling the compiler that that is okay. Here, we are asking for a type with elements that can hold any type safely and the compiler obliges. The answer to the question of how `List<Object>` and the raw type `List` compare is that `List<Object>` is the "generic safe" version of the raw type of yesterday.

Wildcard Type Relationships

Before we leave our wild discussion of wildcard types, let's return one more time to the notion of wildcard type instantiations as types in the Java type system. Earlier in this chapter, we described how regular concrete instantiations of generic types are related by virtue of their "base" generic type inheritance, only with the proviso that their type parameters are exactly the same. Later, we tried to instill the idea that wildcard instantiations add an inheritance relationship to the type parameters, the other half of the generic instantiation. Now, we'll bring the two together. Things can get arcane pretty quickly, but the simple cases are easy to swallow.

The question is, If we have two different wildcard instantiations of a type or related types, how, if at all, are they related? For example, since an unbounded wildcard instantiation can hold any instantiation, can it be assigned a value with a more restrictive bound?

```
List< ? extends Date > dateLists = ...;
List< ? > anyLists;
anyLists = dateLists; // Ok!
```

The answer is yes. For purposes of assignability, wildcard instantiations can be considered as types with possible supertype or subtype relationships determined by their bounds. Let's spell out the unbounded wildcard instantiation as it really is, an instantiation with an upper bound of `Object`:

```
List< ? extends Date > dateLists = ...;
List< ? extends Object > objectLists;
objectLists = dateLists; // Ok!
```

The rule is that if the “base” generic, raw type is assignable and the bounds of the wildcard instantiation are also assignable, the overall types are assignable. Let's look at another example:

```
List< ? extends Integer > intLists = ...;
Collection< ? extends Number > numCollections;
numCollections = intLists; // Ok!
```

What this effectively says is that some `List` of `Integer` types can be treated as some `Collection` of `Number` types through the wildcard instantiation. If you think about it, you'll see that there is no conflict here. A `List` is certainly a `Collection`. And all we're doing is widening the type by which we can read the elements from `Integer` to `Number`. In neither case could we have written to the collection via the wildcard instantiation anyway.

What all this ultimately means is that with the introduction of wildcard instantiations, the type relationships of Java generic classes becomes two-dimensional. There is the raw type relationship to consider and then the wildcard parameter relationship. In fact, if you consider that generic classes may have more than one type parameter, the relationships can get even more complicated (N-dimensional). Fortunately, none of this comes up very often in the real world.

Generic Methods

Thus far in this chapter, we've talked about generic types and the implementation of generic classes. Now, we're going to look at a different kind of generic animal: *generic methods*. Generic methods essentially do for individual methods what type parameters do for generic classes. But as we'll see, generic methods are smarter and can figure out their parameter types from their usage context, without having to be explicitly parameterized. (In reality, of course, it is the compiler that does this.) Generic methods can appear in any class (not just generic classes) and are very useful for a wide variety of applications.

First, let's quickly review the way that we've seen regular methods interact with generic types. We've seen that generic classes can contain methods that use type variables in their arguments and return types in order to adapt themselves to the parameterization

of the class. We've also mentioned that generic types themselves can be used in most of the places that any other type can be used. So methods of generic or nongeneric classes can use generic types as argument and return types as well. Here are examples of those usages:

```
// Not generic methods

class GenericClass< T > {
    // method using generic class parameter type
    public void T cache( T entry ) { ... }
}
class RegularClass {
    // method using concrete generic type
    public List<Date> sortDates( List<Date> dates ) { ... }
    // method using wildcard generic type
    public List<?> reverse( List<?> dates ) { ... }
}
```

The `cache()` method in `GenericClass` accepts an argument of the parameter type `T` and also returns a value of type `T`. The `sortDates()` method, which appears in the nongeneric example class, works with a concrete generic type, and the `reverse()` method works with a wildcard instantiation of a generic type. These are examples of methods that work with generics, but they are not true generic methods.

Generic Methods Introduced

Like generic classes, generic methods have a parameter type declaration using the `<>` syntax. This syntax appears before the return type of the method:

```
// generic method
<T> T cache( T entry ) { ... }
```

This `cache()` method looks very much like our earlier example, except that it has its own parameter type declaration that defines the type variable `T`. This method is a generic method and can appear in either a generic or nongeneric class. The scope of `T` is limited to the method `cache()` and hides any definition of `T` in any enclosing generic class. As with generic classes, the type `T` can have bounds:

```
<T extends Entry & Cacheable > T cache( T entry ) { ... }
```

Unlike a generic class, it does not have to be instantiated with a specific parameter type for `T` before it is used. Instead, it infers the parameter type `T` from the type of its argument, `entry`. For example:

```
BlogEntry newBlogEntry = ...;
NewspaperEntry newNewspaperEntry = ...;

BlogEntry oldEntry = cache( newBlogEntry );
NewspaperEntry old = cache( newNewspaperEntry );
```

Here, our generic method `cache()` inferred the type `BlogEntry` (which we'll presume for the sake of the example is a type of `Entry` and `Cacheable`). `BlogEntry` became the

type `T` of the return type and may have been used elsewhere internally by the method. In the next case, the `cache()` method was used on a different type of `Entry` and was able to return the new type in exactly the same way. That's what's powerful about generic methods: the ability to infer a parameter type from their usage context. We'll go into detail about that next.

Another difference with generic class components is that generic methods may be static:

```
class MathUtils {
    public static <T extends Number> T max( T x, T y ) { ... }
}
```

Constructors for classes are essentially methods, too, and follow the same rules as generic methods, minus the return type.

Type Inference from Arguments

In the previous section, we saw a method infer its type from an argument:

```
<T> T cache( T entry ) { ... }
```

But what if there is more than one argument? We saw just that situation in our last snippet, the static generic method `max(x, y)`. All looks well when we give it two identical types:

```
Integer max = MathUtils.max( new Integer(1), new Integer( 2 ) );
```

But what does it make of the arguments in this invocation?

```
MathUtils.max( new Integer(1), new Float( 2 ) );
```

In this case, the Java compiler does something really smart. It climbs up the argument type parent classes, looking for the *nearest common supertype*. Java also identifies the *nearest common interfaces* implemented by both of the types. It identifies that both the `Integer` and the `Float` types are subtypes of the `Number` type. It also recognizes that each of these implements (a certain generic instantiation of) the `Comparable` interface. Java then effectively makes this combination of types the parameter type of `T` for this method invocation. The resulting type is, to use the syntax of bounds, `Number & Comparable`. What this means to us is that the result type `T` is assignable to anything matching that particular combination of types.

```
Number max = MathUtils.max( new Integer(1), new Float( 2 ) );
Comparable max = MathUtils.max( new Integer(1), new Float( 2 ) );
```

In English, this statement says that we can work with our `Integer` and our `Float` at the same time only if we think of them as `Numbers` or `Comparables`, which makes sense. The return type has become a new type, which is effectively a `Number` that also implements the `Comparable` interface.

This same inference logic works with any number of arguments. But to be useful, the arguments really have to share some important common supertype or interface. If they don't have anything in common, the result will be their de facto common ancestor, the `Object` type. For example, the nearest common supertype of a `String` and a `List` is `Object` along with the `Serializable` interface. There's not much a method could do with a type lacking real bounds anyway.

Type Inference from Assignment Context

We've seen a generic method infer its parameter type from its argument types. But what if the type variable isn't used in any of the arguments or the method has no arguments? Suppose the method only has a parametric return type:

```
<T> T foo() { ... }
```

You might guess that this is an error because the compiler would appear to have no way of determining what type we want. But it's not! The Java compiler is smart enough to look at the context in which the method is called. Specifically, if the result of the method is assigned to a variable, the compiler tries to make the type of that variable the parameter type. Here's an example. We'll make a factory for our `Trap` objects:

```
<T> Trap<T> makeTrap() { return new Trap<T>(); }

// usage
Trap<Mouse> mouseTrap = makeTrap();
Trap<Bear> bearTrap = makeTrap();
```

The compiler has, as if by magic, determined what kind of instantiation of `Trap` we want based on the assignment context.

Before you get too excited about the possibilities, there's not much you can do with a plain type parameter in the body of that method. For example, we can't create instances of any particular concrete type `T`, so this limits the usefulness of factories. About all we can do is the sort of thing shown here, where we create instances of generics parameterized correctly for the context.

Furthermore, the inference only works on assignment to a variable. Java does not try to guess the parameter type based on the context if the method call is used in other ways, such as to produce an argument to a method or as the value of a return statement from a method. In those cases, the inferred type defaults to type `Object`. (See the section "Explicit Type Invocation" for a solution.)

Explicit Type Invocation

Although it should not be needed often, a syntax does exist for invoking a generic method with specific parameter types. The syntax is a bit awkward and involves a

class or instance object prefix, followed by the familiar angle bracket type list, placed before the actual method invocation. Here are some examples:

```
Integer i = MathUtilities.<Integer>max( 42, 42 );
String s = fooObject.<String>foo( "foo" );
String s = this.<String>foo( "foo" );
```

The prefix must be a class or object instance containing the method. One situation where you'd need to use explicit type invocation is if you are calling a generic method that infers its type from the assignment context, but you are not assigning the value to a variable directly. For example, if you wanted to pass the result of our `makeTrap()` method as a parameter to another method, it would otherwise default to `Object`.

Wildcard Capture

Generic methods can do one more trick for us involving taming wildcard instantiations of generic types. The term *wildcard capture* refers to the fact that generic methods can work with arguments whose type is a wildcard instantiation of a type, just as if the type were known:

```
<T> Set<T> listToSet( List<T> list ) {
    Set<T> set = new HashSet<T>();
    set.addAll( list );
    return set;
}

// usage
List<?> list = new ArrayList<Date>();
Set<?> set = listToSet( list );
```

The result of these examples is that we converted an unknown instantiation of `List` to an unknown instantiation of `Set`. The type variable `T` represents the actual type of the argument, `list`, for purposes of the method body. The wildcard instantiation must match any bounds of the method parameter type. But since we can work with the type variable only through its bounds types, the compiler is free to refer to it by this new name, `T`, as if it were a known type. That may not seem very interesting, but it is useful because it allows methods that accept wildcard instantiations of types to delegate to other generic methods to do their work.

Another way to look at this is that generic methods are a more powerful alternative to methods using wildcard instantiations of types. We'll do a little comparison next.

Wildcard Types Versus Generic Methods

You'll recall that trying to work with an object through a wildcard instantiation of its generic type limits us to "reading" the object. We cannot "write" types to the object because its parameter type is unknown. In contrast, because generic methods can

infer or capture an actual type for their arguments, they allow us to do a lot more with broad ranges of types than we could with wildcard instantiations alone.

For example, suppose we wanted to write a utility method that swaps the first two elements of a list. Using wildcards, we'd like to write something like this:

```
// Bad implementation
List<?> swap( List<?> list ) {
    Object tmp = list.get(0);
    list.set( 0, list.get(1) ); // error, can't write
    list.set( 1, tmp ); // error, can't write
    return list;
}
```

But we are not allowed to call the `set()` method of our list because we don't know what type it actually holds. We are really stuck and there isn't much we can do. But the corresponding generic method gives us a real type to hang our hat on:

```
<T> List<T> swap( List<T> list ) {
    T tmp = list.get( 0 );
    list.set( 0, list.get(1) );
    list.set( 1, tmp );
    return list;
}
```

Here, we are able to declare a variable of the correct (inferred) type and write using the `set()` methods appropriately. It would seem that generic methods are the only way to go here. But there is a third path. Wildcard capture, as described in the previous section, allows us to delegate our wildcard version of the method to our actual generic method and use it as if the type were inferred, even though it's open-ended:

```
List<?> swap( List<?> list ) {
    return swap2( list ); // delegate to generic form
}
```

Here, we renamed the generic version as `swap2()` and delegated.

Arrays of Parameterized Types

There is one place where we haven't yet considered how generic types affect the Java language: array types. After everything we've seen, it would seem natural to expect that arrays of generic types would come along for the ride. But as we'll see, Java has a schizophrenic relationship with arrays of parameterized types.

The first thing we need to do is recall how arrays work for regular Java types. An array is a kind of built-in collection of some base type of element. Furthermore, array types (including all multidimensional variations of the array) are true types in the Java language and are represented at runtime by unique class types. This is where the trouble begins. Although arrays in Java act a lot like generic collections (they change their APIs to adopt a particular type for "reading" and "writing"), they do not behave like Java generics with respect to their type relationships. As we saw in Chapter 6,

arrays exist in the Java class hierarchy stemming from `Object` and extending down parallel branches with the plain Java objects.

Arrays are *covariant subtypes* of other types of arrays, which means that, unlike concrete generic types, although they change their method signatures, they are still related to their parents. This means that `String []` in Java is a subtype of `Object []`. This brings up the aliasing problem that we mentioned earlier. An array of `String`s can be aliased as an array of `Object`s and we can attempt to put things into it illegally that won't be noticed until runtime:

```
String [] strings = new String[5];
Object [] objects = strings;
objects[0] = new Date(); // Runtime ArrayStoreException!
```

To prevent disaster, Java must check every array assignment for the correct type at runtime. But recall that generic types do not have real representations at runtime; there is only the raw type. So Java would have no way to know the difference between a `Trap<Mouse>` and a `Trap<Bear>` element in an array once the array was aliased as, say, an `Object []`. For this reason, Java does not allow you to create arrays of generic types—at least not concrete ones. (More on that later in this chapter.)

Using Array Types

Now, since we just said that Java won't let you make any of these arrays, you'd expect that would be pretty much the end of the story. But no! Even though we don't have real array implementations that perform the needed runtime behavior, Java allows us to declare the array type anyway. The catch is that you have to break type safety to use them by using an array of the raw type as their implementation:

```
Trap<Mouse> [] tma = new Trap[10]; // unchecked warning
Trap<Mouse> tm = new Trap<Mouse>();
tma[0] = tm;
Trap<Mouse> again = tma[0];
```

Here, we declared an array of a generic type, `Trap<Mouse>`. Assigning any value (other than null) to the variable `tma` results in an unchecked warning from the compiler at the point of the assignment.

What we are effectively telling the compiler here is to trust us to make sure that the array contains only the correct generic types and asking it to allow us to use it thereafter as if it were checked. We do not get warnings at each usage as we would with a raw type, only at the point where we assign the array. The catch is that the compiler can't prevent us from abusing the array. The unchecked warning at the point where we assign the array is just a representative warning that reminds us that it's possible to abuse the array later.

What Good Are Arrays of Generic Types?

Why does Java let us even declare arrays of generic types? One important usage is that it allows generic types to be used in variable-length argument methods. For example:

```
void useLists( List<String> ... lists ) {  
    List<String> ls0 = lists[0];  
}
```

Another answer is that it's an escape hatch to preserve our ability to use arrays when necessary. You might want to do this for at least two reasons. First, arrays are faster than collections in many cases. The Java runtime is very good at optimizing array access and sometimes it just might be worth it to you to eat the compiler warning to get the benefits. Second, there is the issue of interfacing generic code to legacy code in which only the Javadoc and your faith in the developer are your guarantees as to the contents. By assigning raw arrays to generic instantiations, we can at least ensure that in simple usage we don't abuse the types in the new code.

Wildcards in Array Types

In general, wildcard instantiations of generics can be used as the base type for arrays in the same way that concrete instantiations can. Let's look at an example:

```
ArrayList<?>[] arrayOfArrayLists = ...;
```

This type declaration is an array of unbounded wildcard instantiations of `ArrayList`. Each element of the array can hold an instance of the wildcard type, meaning in this case, that each element of the array could hold a different instantiation of `ArrayList`. For example:

```
arrayOfArrayLists[0] = new ArrayList<Date>();  
arrayOfArrayLists[1] = new ArrayList<String>();
```

There is also a secret surprise we are going to spring on you relating to wildcard types in arrays. Although we said that Java won't let us create arrays of generic types, there is an exception to the rule. Java does allow us to create arrays of unbounded wildcard instantiations. Here are two examples:

```
ArrayList<?>[] arrayOfArrayLists = new ArrayList<?>[10];  
arrayOfArrayLists[0] = new ArrayList<Date>();
```

```
Trap<?> [] arrayOfTraps = new Trap<?>[10];  
arrayOfTraps[0] = new Trap<Mouse>();
```

Here, we not only declared two arrays of wildcard instantiations, but we allocated the arrays as well! The trick is that the arrays must be of the unbounded wildcard type. Why does this work? Because each element in the unbounded wildcard instantiation of the array can hold any instantiation, no runtime check of the generic portion of the

type is necessary at runtime. Any instantiation of `ArrayList` is assignable to the element of type `ArrayList<?>`, so only the check of the raw type is required.

The term *reifiable type* is used to refer to any type that is unchanged by erasure. This includes plain Java concrete types, primitives, and unbounded wildcard instantiations. Reifiable types are kind of like the real people in *The Matrix*: they still exist when unplugged from the simulation.

Case Study: The Enum Class

If you take a look at the definition of the `java.lang.Enum` class in Java 5.0, you'll see a rather bizarre-looking generic type declaration:

```
Enum< E extends Enum<E> > { ... }
```

In trying to parse this, you may be hampered by two thoughts, which we'll try to dispel right away. First, upon quick inspection this may appear to be recursive. The type variable `E` seems to be defined as something that's not yet finished being defined. But it's not really. We often have mathematical equations of the form $x = \text{function}(x)$ and they are not recursive. What they really call out for is a special value of x that satisfies the condition. Next, although it's pretty clear that `E` is a subtype of some formulation of the generic `Enum` type, you may jump to the conclusion that `E` itself must be a generic type. Remember that concrete types can extend generics just as well as generics can.

With these thoughts in mind, let's hunt for some arrangement that satisfies these bounds. Let's focus only on the bound for a moment:

```
E extends Enum<E>
```

`E` is a subclass of some parameterization of `Enum` and, in particular, the parameterization of `Enum` is on the subclass type itself. To say this again, what it does is to require that any invocations of the `Enum` type are by subclasses of some parameterization of the `Enum` type. And specifically, the parameterizations of the `Enum` type supply their own type as the type parameter to their parent, `Enum`. What kind of class satisfies this condition?

```
class Foo extends Enum<Foo> { }
```

This `Foo` class does. The declaration of `Foo`, in fact, reads just as the bound does. `Foo` is a plain concrete type that extends `Enum` parameterized by its own type.*

What does this accomplish exactly? The first implication of this arrangement is that `Enum` can be instantiated only by subclasses of itself. Next, we have the condition that the `Enum` must be instantiated with the child type as its parameter type. This means

* In real life, Java doesn't let us extend the `Enum` type; that's reserved for the `enum` keyword and the compiler. But the structure is as shown.

that any methods of the parent Enum class that refer to the type variable E will now refer to the *child* type. This peculiar bound has guaranteed that child types customize their parent with their own type. In fact, this is exactly what the Enum class in Java needs to make enums work. The `compareTo()` method of a Java enum refers to the type variable and is intended to be applicable only to other instances of the specific child enum type:

```
public int compareTo( E e ) { ... }
```

For example, a Dog enum type should be able to compare only types of Dog and comparing a Dog with a Cat should produce a compile-time error. The bound accomplishes just that by adapting the `compareTo()` method to the Dog type:

```
class Dog extends Enum<Dog> { ... }
```

Normally, a nonfinal base class, having no way to know what children it may have in the future, could only refer to its own type as a general supertype for all of the children when it wants to work with others of its own kind. Methods of a nongeneric Enum class could only supply methods that work on any Enum. But through the magic of generics, we can effectively change the API of the class based on how it is invoked with parameters. In this case, we arranged that all subclasses must supply themselves as the parameter for the base class, tailoring its methods to themselves and pushing the base type down a generation.

Case Study: The `sort()` Method

Poking around in the `java.util.Collections` class we find all kinds of static utility methods for working with collections. Among them is this goody—the static generic method `sort()`:

```
<T extends Comparable<? super T>> void sort( List<T> list ) { ... }
```

Another nut for us to crack. Let's focus on the last part of the bound:

```
Comparable<? super T>
```

This is a wildcard instantiation of the `Comparable` interface, so we can read the `extends` as `implements`, if it helps. `Comparable` holds a `compareTo()` method for some parameter type. A `Comparable<String>` means that the `compareTo()` method takes type `String`. Therefore, `Comparable<? super T>` is the set of instantiations of `Comparable` on `T` and all of its superclasses. A `Comparable<T>` suffices and, at the other end, so does a `Comparable<Object>`. What this means in English is that the elements must be comparable to their own type or some supertype of their own type. This is sufficient to ensure that the elements can all be compared to one another, but not as restrictive as saying that they must all implement the `compareTo()` method themselves. Some of the elements may inherit the `Comparable` interface from a parent class that knows how to compare only to a supertype of `T` and that is exactly what is allowed here.

Conclusion

Java generics are a very powerful and useful addition to the language. Although some of the details we delved into later in this chapter may seem daunting, the common usage is very simple and compelling: generics make collections better. As you begin to write more code using generics you will find that your code becomes more readable and more understandable. Generics make explicit what previously had to be inferred from usage. They complete the promise of type safety in the Java language and make Java a better language, despite their idiosyncrasies.