Generic Programming

Topics in This Chapter

Chapter 6

You often need to implement classes and methods that work with multiple types. For example, an `ArrayList<T>` stores elements of an arbitrary class `T`. We say that the `ArrayList` class is *generic*, and `T` is a *type parameter*. The basic idea is very simple and incredibly useful. The first two sections of this chapter cover the simple part.

In any programming language with generic types, the details get tricky when you restrict or vary type parameters. For example, suppose you want to sort elements. Then you must specify that `T` provides an ordering. Furthermore, if the type parameter varies, what does that mean for the generic type? For example, what should be the relationship between `ArrayList<String>` to a method that expects an `ArrayList<Object>`? Sections 6.3, "Type Bounds" (page 210) and 6.4, "Type Variance and Wildcards" (page 211) show you how Java deals with these issues.

In Java, generic programming is more complex than it perhaps should be, because generics were added when Java had been around for a while, and they were designed to be backward-compatible. As a consequence, there are a number of unfortunate restrictions, some of which affect every Java programmer. Others are only of interest to implementors of generic classes. See Sections 6.5, "Generics in the Java Virtual Machine" (page 216) and 6.6, "Restrictions on Generics" (page 220) for the details. The final section covers generics and reflection, and you can safely skip it if you are not using reflection in your own programs.

The key points of this chapter are:

1. A generic class is a class with one or more type parameters.
2. A generic method is a method with type parameters.
3. You can require a type parameter to be a subtype of one or more types.
4. Generic types are invariant: When `S` is a subtype of `T`, there is no relationship between `G<S>` and `G<T>`.
5. By using wildcards `G<? extends T>` or `G<? super T>`, you can specify that a method can accept an instantiation of a generic type with a subclass or superclass argument.
6. Type parameters are erased when generic classes and methods are compiled.
7. Erasure puts many restrictions on generic types. In particular, you can't instantiate generic classes or arrays, cast to a generic type, or throw an object of a generic type.
8. The `Class<T>` class is generic, which is useful because methods such as `cast` are declared to produce a value of type `T`.
9. Even though generic classes and methods are erased in the virtual machine, you can find out at runtime how they were declared.

6.1 Generic Classes

A *generic class* is a class with one or more *type parameters*. As a simple example, consider this class for storing key/value pairs:

```
public class Entry<K, V> {
    private K key;
    private V value;

    public Entry(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() { return key; }
    public V getValue() { return value; }
}
```

As you can see, the type parameters `K` and `V` are specified inside angle brackets after the name of the class. In the definitions of class members, they are used as types for instance variables, method parameters, and return values.

You *instantiate* the generic class by substituting types for the type variables. For example, `Entry<String, Integer>` is an ordinary class with methods `String getKey()` and `Integer getValue()`.

CAUTION: Type parameters cannot be instantiated with primitive types. For example, `Entry<String, int>` is not valid in Java.

When you *construct* an object of a generic class, you can omit the type parameters from the constructor. For example,

```
Entry<String, Integer> entry = new Entry<>("Fred", 42);
    // Same as new Entry<String, Integer>("Fred", 42)
```

Note that you still provide an empty pair of angle brackets before the construction arguments. Some people call this empty bracket pair a *diamond*. When you use the diamond syntax, the type parameters for the constructor are inferred.

6.2 Generic Methods

Just like a generic class is a class with type parameters, a *generic method* is a method with type parameters. A generic method can be a method of a regular class or a generic class. Here is an example of a generic method in a class that is not generic:

```
public class Arrays {
    public static <T> void swap(T[] array, int i, int j) {
        T temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
}
```

This `swap` method can be used to swap elements in an arbitrary array, as long as the array element type is not a primitive type.

```
String[] friends = ...;
Arrays.swap(friends, 0, 1);
```

When you declare a generic method, the type parameter is placed after the modifiers (such as public and static) and before the return type:

```
public static <T> void swap(T[] array, int i, int j)
```

When calling a generic method, you do not need to specify the type parameter. It is inferred from the method parameter and return types. For example, in the call Arrays.swap(friends, 0, 1), the type of friends is String[], and the compiler can infer that T should be String.

You can, if you like, supply the type explicitly, before the method name, like this:

```
Arrays.<String>swap(friends, 0, 1);
```

One reason why you might want to do this is to get better error messages when something goes wrong—see Exercise 5.

Before plunging into the morass of technical details in the sections that follow, it is worth contemplating the examples of the Entry class and the swap method and to admire how useful and natural generic types are. With the Entry class, the key and value types can be arbitrary. With the swap method, the array type can be arbitrary. That is plainly expressed with type variables.

6.3 Type Bounds

Sometimes, the type parameters of a generic class or method need to fulfill certain requirements. You can specify a *type bound* to require that the type extends certain classes or implements certain interfaces.

Suppose, for example, you have an ArrayList of objects of a class that implements the AutoCloseable interface, and you want to close them all:

```
public static <T extends AutoCloseable> void closeAll(ArrayList<T> elems)
        throws Exception {
    for (T elem : elems) elem.close();
}
```

The type bound extends AutoCloseable ensures that the element type is a subtype of AutoCloseable. Therefore, the call elem.close() is valid. You can pass an ArrayList<PrintStream> to this method, but not an ArrayList<String>. Note that the extends keyword in a type bound actually means "subtype"—the Java designers just used the existing extends keyword instead of coming up with another keyword or symbol.

Exercise 14 has a more interesting variant of this method.

NOTE: In this example, we need a type bound because the parameter is of type `ArrayList`. If the method accepted an array, you wouldn't need a generic method. You could simply use a regular method

```
public static void closeAll(AutoCloseable[] elems) throws Exception
```

This works because an array type such as `PrintStream[]` is a subtype of `AutoCloseable[]`. However, as you will see in the following section, an `ArrayList<PrintStream>` is *not* a subtype of `ArrayList<AutoCloseable>`. Using a bounded type parameter solves this problem.

A type parameter can have multiple bounds, such as

```
T extends Runnable & AutoCloseable
```

This syntax is similar to that for catching multiple exceptions, the only difference being that the types are combined with an "and" operator, whereas multiple exceptions are combined with an "or" operator.

You can have as many interface bounds 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.

6.4 Type Variance and Wildcards

Suppose you need to implement a method that processes an array of objects that are subclasses of the class `Employee`. You simply declare the parameter to have type `Employee[]`:

```
public static void process(Employee[] staff) { ... }
```

If `Manager` is a subclass of `Employee`, you can pass a `Manager[]` array to the method since `Manager[]` is a subtype of `Employee[]`. This behavior is called *covariance*. Arrays vary in the same way as the element types.

Now, suppose you want to process an array list instead. However, there is a problem: The type `ArrayList<Manager>` is *not* a subtype of `ArrayList<Employee>`.

There is a reason for this restriction. If it were legal to assign an `ArrayList<Manager>` to a variable of type `ArrayList<Employee>`, you could corrupt the array list by storing nonmanagerial employees:

```
ArrayList<Manager> bosses = new ArrayList<>();
ArrayList<Employee> empls = bosses; // Not legal, but suppose it is . . .
empls.add(new Employee(...)); // A nonmanager in bosses!
```

Since conversion from `ArrayList<Manager>` to `ArrayList<Employee>` is disallowed, this error cannot occur.

NOTE: Can you generate the same error with arrays, where the conversion from `Manager[]` to `Employee[]` is permitted? Sure you can, as you saw in Chapter 4. Java arrays are covariant, which is convenient but unsound. When you store a mere `Employee` in a `Manager[]` array, an `ArrayStoreException` is thrown. In contrast, all generic types in Java are *invariant*.

In Java, you use *wildcards* to specify how method parameter and return types should be allowed to vary. This mechanism is sometimes called *use-site variance*. You will see the details in the following sections.

6.4.1 Subtype Wildcards

In many situations it is perfectly safe to convert between different array lists. Suppose a method never writes to the array list, so it cannot corrupt its argument. Use a wildcard to express this fact:

```
public static void printNames(ArrayList<? extends Employee> staff) {
    for (int i = 0; i < staff.size(); i++) {
        Employee e = staff.get(i);
        System.out.println(e.getName());
    }
}
```

The wildcard type `? extends Employee` indicates some unknown subtype of `Employee`. You can call this method with an `ArrayList<Employee>` or an array list of a subtype, such as `ArrayList<Manager>`.

The `get` method of the class `ArrayList<? extends Employee>` has return type `? extends Employee`. The statement

```
Employee e = staff.get(i);
```

is perfectly legal. Whatever type `?` denotes, it is a subtype of `Employee`, and the result of `staff.get(i)` can be assigned to the `Employee` variable `e`. (I didn't use an enhanced `for` loop in this example to show exactly how the elements are fetched from the array list.)

What happens if you try to store an element into an `ArrayList<? extends Employee>`? That would not work. Consider a call

```
staff.add(x);
```

The `add` method has parameter type `? extends Employee`, and there is *no object* that you can pass to this method. If you pass, say, a `Manager` object, the compiler will refuse. After all, `?` could refer to *any* subclass, perhaps `Janitor`, and you can't add a `Manager` to an `ArrayList<Janitor>`.

NOTE: You can, of course, pass `null`, but that's not an object.

In summary, you can convert from `? extends Employee` to `Employee`, but you can never convert anything to `? extends Employee`. This explains why you can read from an `ArrayList<? extends Employee>` but cannot write to it.

6.4.2 Supertype Wildcards

The wildcard type `? extends Employee` denotes an arbitrary subtype of `Employee`. The converse is the wildcard type `? super Employee` which denotes a supertype of `Employee`. These wildcards are often useful as parameters in functional objects. Here is a typical example. The `Predicate` interface has a method for testing whether an object of type `T` has a particular property:

```
public interface Predicate<T> {
    boolean test(T arg);
    ...
}
```

This method prints the names of all employees with a given property:

```
public static void printAll(Employee[] staff, Predicate<Employee> filter) {
    for (Employee e : staff)
        if (filter.test(e))
            System.out.println(e.getName());
}
```

You can call this method with an object of type `Predicate<Employee>`. Since that is a functional interface, you can also pass a lambda expression:

```
printAll(employees, e -> e.getSalary() > 100000);
```

Now suppose you want to use a `Predicate<Object>` instead, for example

```
Predicate<Object> evenLength = e -> e.toString().length() % 2 == 0;
printAll(employees, evenLength);
```

This should not be a problem. After all, every `Employee` is an `Object` with a `toString` method. However, like all generic types, the `Predicate` interface is invariant, and there is no relationship between `Predicate<Employee>` and `Predicate<Object>`.

The remedy is to allow any `Predicate<? super Employee>`:

```
public static void printAll(Employee[] staff, Predicate<? super Employee> filter) {
    for (Employee e : staff)
        if (filter.test(e))
            System.out.println(e.getName());
}
```

Have a close look at the call `filter.test(e)`. Since the parameter of `test` has a type that is some supertype of `Employee`, it is safe to pass an `Employee` object.

This situation is typical. Functions are naturally *contravariant* in their parameter types. For example, when a function is expected that can process employees, it is OK to give one that is willing to process arbitrary objects.

In general, when you specify a generic functional interface as a method parameter, you should use a `super` wildcard.

NOTE: Some programmers like the "PECS" mnemonic for wildcards: producer `extends`, consumer `super`. An `ArrayList` from which you read values is a producer, so you use an `extends` wildcard. A `Predicate` to which you give values for testing is a consumer, and you use `super`.

6.4.3 Wildcards with Type Variables

Consider a generalization of the method of the preceding section that prints arbitrary elements fulfilling a condition:

```
public static <T> void printAll(T[] elements, Predicate<T> filter) {
    for (T e : elements)
        if (filter.test(e))
            System.out.println(e.toString());
}
```

This is a generic method that works for arrays of any type. The type parameter is the type of the array that is being passed. However, it suffers from the limitation that you saw in the preceding section. The type parameter of `Predicate` must exactly match the type parameter of the method.

The solution is the same that you already saw—but this time, the bound of the wildcard is a type variable:

```
public static <T> void printAll(T[] elements, Predicate<? super T> filter)
```

This method takes a filter for elements of type `T` or any supertype of `T`.

Here is another example. The `Collection<E>` interface, which you will see in detail in the following chapter, describes a collection of elements of type `E`. It has a method

```
public boolean addAll(Collection<? extends E> c)
```

You can add all elements from another collection whose element type is also `E` or some subtype. With this method, you can add a collection of managers to a collection of employees, but not the other way around.

To see how complex type declarations can get, consider the definition of the `Collections.sort` method:

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

The `List` interface, covered in detail in the next chapter, describes a sequence of elements, such as a linked list or `ArrayList`. The `sort` method is willing to sort any `List<T>`, provided `T` is a subtype of `Comparable`. But the `Comparable` interface is again generic:

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

Its type parameter specifies the argument type of the `compareTo` method. So, it would seem that `Collections.sort` could be declared as

```
public static <T extends Comparable<T>> void sort(List<T> list)
```

But that is too restrictive. Suppose that the `Employee` class implements `Comparable<Employee>`, comparing employees by salary. And suppose that the `Manager` class extends `Employee`. Note that it implements `Comparable<Employee>`, and *not* `Comparable<Manager>`. Therefore, `Manager` is *not* a subtype of `Comparable<Manager>`, but it is a subtype of `Comparable<? super Manager>`.

NOTE: In some programming languages (such as C# and Scala), you can declare type parameters to be covariant or contravariant. For example, by declaring the type parameter of `Comparable` to be contravariant, one doesn't have to use a wildcard for each `Comparable` parameter. This "declaration-site variance" is convenient, but it is less powerful than the "use-site variance" of Java wildcards.

6.4.4 Unbounded Wildcards

It is possible to have unbounded wildcards for situations where you only do very generic operations. For example, here is a method to check whether an `ArrayList` has any `null` elements:

```
public static boolean hasNulls(ArrayList<?> elements) {
    for (Object e : elements) {
        if (e == null) return true;
    }
    return false;
}
```

Since the type parameter of the `ArrayList` doesn't matter, it makes sense to use an `ArrayList<?>`. One could equally well have made `hasNulls` into a generic method:

```
public static <T> boolean hasNulls(ArrayList<T> elements)
```

But the wildcard is easy to understand, so that's the preferred approach.

6.4.5 Wildcard Capture

Let's try to define a `swap` method using wildcards:

```
public static void swap(ArrayList<?> elements, int i, int j) {
    ? temp = elements.get(i); // Won't work
    elements.set(i, elements.get(j));
    elements.set(j, temp);
}
```

That won't work. You can use `?` as a type argument, but not as a type.

However, there is a workaround. Add a helper method, like this:

```
public static void swap(ArrayList<?> elements, int i, int j) {
    swapHelper(elements, i, j);
}

private static <T> void swapHelper(ArrayList<T> elements, int i, int j) {
    T temp = elements.get(i);
    elements.set(i, elements.get(j));
    elements.set(j, temp);
}
```

The call to `swapHelper` is valid because of a special rule called *wildcard capture*. The compiler doesn't know what `?` is, but it stands for some type, so it is OK to call a generic method. The type parameter `T` of `swapHelper` "captures" the wildcard type. Since `swapHelper` is a generic method, not a method with wildcards in parameters, it can make use of the type variable `T` to declare variables.

What have we gained? The user of the API sees an easy-to-understand `ArrayList<?>` instead of a generic method.

6.5 Generics in the Java Virtual Machine

When generic types and methods were added to Java, the Java designers wanted the generic forms of classes to be compatible with their preexisting versions. For example, it should be possible to pass an `ArrayList<String>` to a method from pre-generic days that accepted the `ArrayList` class, which collects elements of type `Object`. The language designers decided on an implementation that "erases" the types in the virtual machine. This was very popular at the time since it enabled Java users to gradually migrate to using generics. As you can imagine, there are drawbacks to this scheme, and, as so often happens with compromises made in the interest of compatibility, the drawbacks remain long after the migration has successfully completed.

In this section, you will see what goes on in the virtual machine, and the next section examines the consequences.

6.5.1 Type Erasure

When you define a generic type, it is compiled into a *raw* type. For example, the `Entry<K, V>` class of Section 6.1, "Generic Classes" (page 208) turns into

```
public class Entry {
    private Object key;
    private Object value;

    public Entry(Object key, Object value) {
        this.key = key;
        this.value = value;
    }

    public Object getKey() { return key; }
    public Object getValue() { return value; }
}
```

Every `K` and `V` is replaced by `Object`.

If a type variable has bounds, it is replaced with the first bound. Suppose we declare the `Entry` class as

```
public class Entry<K extends Comparable<? super K> & Serializable,
                  V extends Serializable>
```

Then it is erased to a class

```
public class Entry {
    private Comparable key;
    private Serializable value;
    ...
}
```

6.5.2 Cast Insertion

Erasure sounds somehow dangerous, but it is actually perfectly safe. Suppose for example, you used an `Entry<String, Integer>` object. When you construct the object, you must provide a key that is a `String` and a value that is an `Integer` or is converted to one. Otherwise, your program does not even compile. You are therefore guaranteed that the `getKey` method returns a `String`.

However, suppose your program compiled with "unchecked" warnings, perhaps because you used casts or mixed generic and raw `Entry` types. Then it is possible for an `Entry<String, Integer>` to have a key of a different type.

Therefore, it is also necessary to have safety checks at runtime. The compiler inserts a cast whenever one reads from an expression with erased type. Consider, for example,

```
Entry<String, Integer> entry = ...;
String key = entry.getKey();
```

Since the erased `getKey` method returns an `Object`, the compiler generates code equivalent to

```
String key = (String) entry.getKey();
```

6.5.3 Bridge Methods

In the preceding sections, you have seen the basics of what erasure does. It is simple and safe. Well, almost simple. When erasing method parameter and return types, it is sometimes necessary for the compiler to synthesize *bridge methods*. This is an implementation detail, and you don't need to know about it unless you want to know why such a method shows up in a stack trace, or you want an explanation for one of the more obscure limitations on Java generics (see Section 6.6.6, "Methods May Not Clash after Erasure," page 224).

Consider this example:

```
public class WordList extends ArrayList<String> {
    public boolean add(String e) {
        return isBadWord(e) ? false : super.add(e);
    }
    ...
}
```

Now consider this code fragment:

```
WordList words = ...;
ArrayList<String> strings = words; // OK—conversion to superclass
strings.add("C++");
```

The last method call invokes the (erased) `add(Object)` method of the `ArrayList` class.

One would reasonably expect dynamic method lookup to work in this case so that the `add` method of `WordList`, not the `add` method of `ArrayList`, is called when `add` is invoked on a `WordList` object.

To make this work, the compiler synthesizes a bridge method in the `WordList` class:

```
public boolean add(Object e) {
    return add((String) e);
}
```

In the call `strings.add("C++")`, the `add(Object)` method is called, and it calls the `add(String)` method of the `WordList` class.

Bridge methods can also be called when the return type varies. Consider this method:

```
public class WordList extends ArrayList<String> {
    public String get(int i) {
        return super.get(i).toLowerCase();
    }
    ...
}
```

In the `WordList` class, there are two `get` methods:

```
String get(int) // Defined in WordList
Object get(int) // Overrides the method defined in ArrayList
```

The second method is synthesized by the compiler, and it calls the first. This is again done to make dynamic method lookup work.

These methods have the same parameter types but different return types. In the Java language, you cannot implement such a pair of methods. But in the virtual machine, a method is specified by its name, the parameter types, *and* the return type, which allows the compiler to generate this method pair.

NOTE: Bridge methods are not only used for generic types. They are also used to implement covariant return types. For example, in Chapter 4, you saw how you should declare a `clone` method with the appropriate return type:

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

In this case, the `Employee` class has two `clone` methods:

```
Employee clone() // Defined above
Object clone() // Synthesized bridge method
```

The bridge method, again generated to make dynamic method lookup work, calls the first method.

6.6 Restrictions on Generics

There are several restrictions when using generic types and methods in Java—some merely surprising and others genuinely inconvenient. Most of them are consequences of type erasure. The following sections show you those that you will most likely encounter in practice.

6.6.1 No Primitive Type Arguments

A type parameter can never be a primitive type. For example, you cannot form an `ArrayList<int>`. As you have seen, in the virtual machine there is only one type, the raw `ArrayList` that stores elements of type `Object`. An `int` is not an object.

When generics were first introduced, this was not considered a big deal. After all, one can form an `ArrayList<Integer>` and rely on autoboxing. Now that generics are more commonly used, however, the pain is increasing. There is a profusion of functional interfaces such as `IntFunction`, `LongFunction`, `DoubleFunction`, `ToIntFunction`, `ToLongFunction`, `ToDoubleFunction`—and that only takes care of unary functions and three of the eight primitive types.

6.6.2 At Runtime, All Types Are Raw

In the virtual machine, there are only raw types. For example, you cannot inquire at runtime whether an `ArrayList` contains `String` objects. A condition such as

```
if (a instanceof ArrayList<String>)
```

is a compile-time error since no such check could ever be executed.

A cast to an instantiation of a generic type is equally ineffective, but it is legal.

```
Object result = ...;
ArrayList<String> list = (ArrayList<String>) result;
    // Warning—this only checks whether result is a raw ArrayList
```

Such a cast is allowed because there is sometimes no way to avoid it. If `result` is the outcome of a very general process (such as calling a method through reflection, see Chapter 4) and its exact type is not known to the compiler, the programmer must use a cast. A cast to `ArrayList` or `ArrayList<?>` would not suffice.

To make the warning go away, annotate the variable like this:

```
@SuppressWarnings("unchecked") ArrayList<String> list
    = (ArrayList<String>) result;
```

CAUTION: Abusing the `@SuppressWarnings` annotation can lead to *heap pollution*—objects that should belong to a particular generic type instantiation but actually belong to a different one. For example, you can assign an `ArrayList<Employee>` to an `ArrayList<String>` reference. The consequence is a `ClassCastException` when an element of the wrong type is retrieved.

TIP: The trouble with heap pollution is that the reported runtime error is far from the source of the problem—the insertion of a wrong element. If you need to debug such a problem, you can use a "checked view." Where you constructed, say, an `ArrayList<String>`, instead use

```
List<String> strings
    = Collections.checkedList(new ArrayList<>(), String.class);
```

The view monitors all insertions into the list and throws an exception when an object of the wrong type is added.

The `getClass` method always returns a raw type. For example, if `list` is an `ArrayList<String>`, then `list.getClass()` returns `ArrayList.class`. In fact, there is no `ArrayList<String>.class`—such a class literal is a syntax error.

Also, you cannot have type variables in class literals. There is no `T.class`, `T[].class`, or `ArrayList<T>.class`.

6.6.3 You Cannot Instantiate Type Variables

You cannot use type variables in expressions such as `new T(...)` or `new T[...]`. These forms are outlawed because they would not do what the programmer intends when `T` is erased.

If you want to create a generic instance or array, you have to work harder. Suppose you want to provide a `repeat` method so that `Arrays.repeat(n, obj)` makes an array containing `n` copies of `obj`. Of course, you'd like the element type of the array to be the same as the type of `obj`. This attempt does not work:

```
public static <T> T[] repeat(int n, T obj) {
    T[] result = new T[n]; // Error—cannot construct an array new T[...]
    for (int i = 0; i < n; i++) result[i] = obj;
    return result;
}
```

To solve this problem, ask the caller to provide the array constructor as a method reference:

```
String[] greetings = Arrays.repeat(10, "Hi", String[]::new);
```

Here is the implementation of the method:

```
public static <T> T[] repeat(int n, T obj, IntFunction<T[]> constr) {
    T[] result = constr.apply(n);
    for (int i = 0; i < n; i++) result[i] = obj;
    return result;
}
```

Alternatively, you can ask the user to supply a class object, and use reflection.

```
public static <T> T[] repeat(int n, T obj, Class<T> cl) {
    @SuppressWarnings("unchecked") T[] result
        = (T[]) java.lang.reflect.Array.newInstance(cl, n);
    for (int i = 0; i < n; i++) result[i] = obj;
    return result;
}
```

This method is called as follows:

```
String[] greetings = Arrays.repeat(10, "Hi", String.class);
```

Another option is to ask the caller to allocate the array. Usually, the caller is allowed to supply an array of any length, even zero. If the supplied array is too short, the method makes a new one, using reflection.

```
public static <T> T[] repeat(int n, T obj, T[] array) {
    T[] result;
    if (array.length >= n)
        result = array;
    else {
        @SuppressWarnings("unchecked") T[] newArray
            = (T[]) java.lang.reflect.Array.newInstance(
                array.getClass().getComponentType(), n);
        result = newArray;
    }
    for (int i = 0; i < n; i++) result[i] = obj;
    return result;
}
```

TIP: You *can* instantiate an `ArrayList` with a type variable. For example, the following is entirely legal:

```
public static <T> ArrayList<T> repeat(int n, T obj) {
    ArrayList<T> result = new ArrayList<>(); // OK
    for (int i = 0; i < n; i++) result.add(obj);
    return result;
}
```

This is much simpler than the workarounds you just saw, and I recommend it whenever you don't have a compelling reason for producing an array.

NOTE: If a generic class needs a generic array that is a private part of the implementation, you can get away with just constructing an `Object[]` array. This is what the `ArrayList` class does:

```
public class ArrayList<E> {
    private Object[] elementData;

    public E get(int index) {
        return (E) elementData[index];
    }
    ...
}
```

6.6.4 You Cannot Construct Arrays of Parameterized Types

Suppose you want to create an array of `Entry` objects:

```
Entry<String, Integer>[] entries = new Entry<String, Integer>[100];
    // Error—cannot construct an array with generic component type
```

This is a syntax error. The construction is outlawed because, after erasure, the array constructor would create a raw `Entry` array. It would then be possible to add `Entry` objects of any type (such as `Entry<Employee, Manager>`) without an `ArrayStoreException`.

Note that the *type* `Entry<String, Integer>[]` is perfectly legal. You can declare a variable of that type. If you really want to initialize it, you can, like this:

```
@SuppressWarnings("unchecked") Entry<String, Integer>[] entries
    = (Entry<String, Integer>[]) new Entry<?, ?>[100];
```

But it is simpler to use an array list:

```
ArrayList<Entry<String, Integer>> entries = new ArrayList<>(100);
```

Recall that a varargs parameter is an array in disguise. If such a parameter is generic, you can bypass the restriction against generic array creation. Consider this method:

```
public static <T> ArrayList<T> asList(T... elements) {
    ArrayList<T> result = new ArrayList<>();
    for (T e : elements) result.add(e);
    return result;
}
```

Now consider this call:

```
Entry<String, Integer> entry1 = ...;
Entry<String, Integer> entry2 = ...;
ArrayList<Entry<String, Integer>> entries = Lists.asList(entry1, entry2);
```

The inferred type for `T` is the generic type `Entry<String, Integer>`, and therefore `elements` is an array of type `Entry<String, Integer>`. That is just the kind of array creation that you cannot do yourself!

In this case, the compiler reports a warning, not an error. If your method only reads elements from the parameter array, it should use the `@SafeVarargs` annotation to suppress the warning:

```
@SafeVarargs public static <T> ArrayList<T> asList(T... elements)
```

This annotation can be applied to methods that are `static`, `final`, or `private`, or to constructors. Any other methods might be overridden and are not eligible for the annotation.

6.6.5 Class Type Variables Are Not Valid in Static Contexts

Consider a generic class with type variables, such as `Entry<K, V>`. You cannot use the type variables `K` and `V` with static variables or methods. For example, the following does not work:

```
public class Entry<K, V> {
    private static V defaultValue;
        // Error—V in static context
    public static void setDefault(V value) { defaultValue = value; }
        // Error—V in static context
    ...
}
```

After all, type erasure means there is only one such variable or method in the erased `Entry` class, and not one for each `K` and `V`.

6.6.6 Methods May Not Clash after Erasure

You may not declare methods that would cause clashes after erasure. For example, the following would be an error:

```
public interface Ordered<T> extends Comparable<T> {
    public default boolean equals(T value) {
        // Error—erasure clashes with Object.equals
        return compareTo(value) == 0;
    }
    ...
}
```

The `equals(T value)` method erases to `equals(Object value)`, which clashes with the same method from `Object`.

Sometimes the cause for a clash is more subtle. Here is a nasty situation:

```
public class Employee implements Comparable<Employee> {
    ...
    public int compareTo(Employee other) {
        return name.compareTo(other.name);
    }
}

public class Manager extends Employee implements Comparable<Manager> {
    // Error—cannot have two instantiations of Comparable as supertypes
    ...
    public int compareTo(Manager other) {
        return Double.compare(salary, other.salary);
    }
}
```

The class `Manager` extends `Employee` and therefore picks up the supertype `Comparable<Employee>`. Naturally, managers want to compare each other by salary, not by name. And why not? There is no erasure. Just two methods

```
public int compareTo(Employee other)
public int compareTo(Manager other)
```

The problem is that *the bridge methods clash*. Recall from Section 6.5.3, "Bridge Methods" (page 218) that both of these methods yield a bridge method

```
public int compareTo(Object other)
```

6.6.7 Exceptions and Generics

You cannot throw or catch objects of a generic class. In fact, you cannot even form a generic subclass of `Throwable`:

```
public class Problem<T> extends Exception
    // Error—a generic class can't be a subtype of Throwable
```

You cannot use a type variable in a `catch` clause:

```
public static <T extends Throwable> void doWork(Runnable r, Class<T> cl) {
    try {
        r.run();
    } catch (T ex) { // Error—can't catch type variable
        Logger.getGlobal().log(..., ..., ex);
    }
}
```

However, you *can* have a type variable in the `throws` declaration:

```
public static <V, T extends Throwable> V doWork(Callable<V> c, T ex) throws T {
    try {
        return c.call();
    } catch (Throwable realEx) {
        ex.initCause(realEx);
        throw ex;
    }
}
```

CAUTION: You can use generics to remove the distinction between checked and unchecked exceptions. The key ingredient is this pair of methods:

```
public class Exceptions {
    @SuppressWarnings("unchecked")
    private static <T extends Throwable>
            void throwAs(Throwable e) throws T {
        throw (T) e; // The cast is erased to (Throwable) e
    }
    public static <V> V doWork(Callable<V> c) {
        try {
            return c.call();
        } catch (Throwable ex) {
            Exceptions.<RuntimeException>throwAs(ex);
            return null;
        }
    }
}
```

Now consider this method:

```
public static String readAll(Path path) {
    return doWork(() -> new String(Files.readAllBytes(path)));
}
```

Even though `Files.readAllBytes` throws a checked exception when the path is not found, that exception is neither declared nor caught in the `readAll` method!

6.7 Reflection and Generics

In the following sections, you will see what you can do with the generic classes in the reflection package and how you can find out the small amount of generic type information in the virtual machine that survives the erasure process.

6.7.1 The Class<T> Class

The `Class` class has a type parameter, namely the class that the `Class` object describes. Huh? Let's do this slowly.

Consider the `String` class. In the virtual machine, there is a `Class` object for this class, which you can obtain as `"Fred".getClass()` or, more directly, as the class literal `String.class`. You can use that object to find out what methods the class has, or to construct an instance.

The type parameter helps write typesafe code. Consider for example the `getConstructor` method of `Class<T>`. It is declared to return a `Constructor<T>`. And the `newInstance` method of `Constructor<T>` is declared to returns an object of type `T`. That's why `String.class` has type `Class<String>`: Its `getConstructor` method yields a `Constructor<String>`, whose `newInstance` method returns a `String`.

That information can save you a cast. Consider this method:

```
public static <T> ArrayList<T> repeat(int n, Class<T> cl)
        throws ReflectiveOperationException {
    ArrayList<T> result = new ArrayList<>();
    for (int i = 0; i < n; i++)
        result.add(cl.getConstructor().newInstance());
    return result;
}
```

The method compiles since `cl.getConstructor().newInstance()` returns a result of type `T`.

Suppose you call this method as `repeat(10, Employee.class)`. Then `T` is inferred to be the type `Employee` since `Employee.class` has type `Class<Employee>`. Therefore, the return type is `ArrayList<Employee>`.

In addition to the `getConstructor` method, there are several other methods of the `Class` class that use the type parameter. They are:

```
Class<? super T> getSuperclass()
<U> Class<? extends U> asSubclass(Class<U> clazz)
T cast(Object obj)
Constructor<T> getDeclaredConstructor(Class<?>... parameterTypes)
T[] getEnumConstants()
```

As you have seen in Chapter 4, there are many situations where you know nothing about the class that a `Class` object describes. Then, you can simply use the wildcard type `Class<?>`.

6.7.2 Generic Type Information in the Virtual Machine

Erasure only affects instantiated type parameters. Complete information about the *declaration* of generic classes and methods is available at runtime.

For example, suppose a call `obj.getClass()` yields `ArrayList.class`. You cannot tell whether `obj` was constructed as an `ArrayList<String>` or `ArrayList<Employee>`. But you can tell that the class `ArrayList` is a generic class with a type parameter `E` that has no bounds.

Similarly, consider the method

```
static  <T extends Comparable<? super T>> void sort(List<T> list)
```

of the `Collections` class. As you saw in Chapter 4, you can get the corresponding `Method` object as

```
Method m = Collections.class.getMethod("sort", List.class);
```

From this `Method` object, you can recover the entire method signature.

The interface `Type` in the `java.lang.reflect` package represents generic type declarations. The interface has the following subtypes:

1. The `Class` class, describing concrete types
2. The `TypeVariable` interface, describing type variables (such as `T extends Comparable<? super T>`)
3. The `WildcardType` interface, describing wildcards (such as `? super T`)
4. The `ParameterizedType` interface, describing generic class or interface types (such as `Comparable<? super T>`)
5. The `GenericArrayType` interface, describing generic arrays (such as `T[]`)

Note that the last four subtypes are interfaces—the virtual machine instantiates suitable classes that implement these interfaces.

Both classes and methods can have type variables. Technically speaking, constructors are not methods, and they are represented by a separate class in the reflection library. They too can be generic. To find out whether a `Class`, `Method`, or `Constructor` object comes from a generic declaration, call the `getTypeParameters` method. You get an array of `TypeVariable` instances, one for each type variable in the declaration, or an array of length 0 if the declaration was not generic.

The `TypeVariable<D>` interface is generic. The type parameter is `Class<T>`, `Method`, or `Constructor<T>`, depending on where the type variable was declared. For example, here is how you get the type variable of the `ArrayList` class:

```
TypeVariable<Class<ArrayList>>[] vars = ArrayList.class.getTypeParameters();
String name = vars[0].getName(); // "E"
```

And here is the type variable of the `Collections.sort` method:

```
Method m = Collections.class.getMethod("sort", List.class);
TypeVariable<Method>[] vars = m.getTypeParameters();
String name = vars[0].getName(); // "T"
```

The latter variable has a bound, which you can process like this:

```
Type[] bounds = vars[0].getBounds();
if (bounds[0] instanceof ParameterizedType) { // Comparable<? super T>
    ParameterizedType p = (ParameterizedType) bounds[0];
    Type[] typeArguments = p.getActualTypeArguments();
    if (typeArguments[0] instanceof WildcardType) { // ? super T
        WildcardType t = (WildCardType) typeArguments[0];
        Type[] upper = t.getUpperBounds(); // ? extends ... & ...
        Type[] lower = t.getLowerBounds(); // ? super ... & ...
        if (lower.length > 0) {
            String description = lower[0].getTypeName(); // "T"
            ...
        }
    }
}
```

This gives you a flavor of how you can analyze generic declarations. I won't dwell on the details since this is not something that commonly comes up in practice. The key point is that the declarations of generic classes and methods are not erased and you have access to them through reflection.

Exercises

1. Implement a class `Stack<E>` that manages an array list of elements of type `E`. Provide methods `push`, `pop`, and `isEmpty`.
2. Reimplement the `Stack<E>` class, using an array to hold the elements. If necessary, grow the array in the `push` method. Provide two solutions, one with an `E[]` array and one with an `Object[]` array. Both solutions should compile without warnings. Which do you prefer, and why?
3. Implement a class `Table<K, V>` that manages an array list of `Entry<K, V>` elements. Supply methods to get the value associated with a key, to put a value for a key, and to remove a key.
4. In the previous exercise, make `Entry` into a nested class. Should that class be generic?
5. Consider this variant of the `swap` method where the array can be supplied with varargs:

```
public static <T> T[] swap(int i, int j, T... values) {
    T temp = values[i];
    values[i] = values[j];
    values[j] = temp;
    return values;
}
```

Now have a look at the call

```
Double[] result = Arrays.swap(0, 1, 1.5, 2, 3);
```

What error message do you get? Now call

```
Double[] result = Arrays.<Double>swap(0, 1, 1.5, 2, 3);
```

Has the error message improved? What do you do to fix the problem?

6. Implement a generic method that appends all elements from one array list to another. Use a wildcard for one of the type arguments. Provide two equivalent solutions, one with a `? extends E` wildcard and one with `? super E`.
7. Implement a class `Pair<E>` that stores a pair of elements of type `E`. Provide accessors to get the first and second element.
8. Modify the class of the preceding exercise by adding methods `max` and `min`, getting the larger or smaller of the two elements. Supply an appropriate type bound for `E`.
9. In a utility class `Arrays`, supply a method

   ```
   public static <E> Pair<E> firstLast(ArrayList<___> a)
   ```

   that returns a pair consisting of the first and last element of `a`. Supply an appropriate type argument.
10. Provide generic methods `min` and `max` in an `Arrays` utility class that yield the smallest and largest element in an array.
11. Continue the preceding exercise and provide a method `minMax` that yields a `Pair` with the minimum and maximum.
12. Implement the following method that stores the smallest and largest element in `elements` in the `result` list:

    ```
    public static <T> void minmax(List<T> elements,
        Comparator<? super T> comp, List<? super T> result)
    ```

    Note the wildcard in the last parameter—any supertype of `T` will do to hold the result.

13. Given the method from the preceding exercise, consider this method:

    ```
    public static <T> void maxmin(List<T> elements,
            Comparator<? super T> comp, List<? super T> result) {
        minmax(elements, comp, result);
        Lists.swapHelper(result, 0, 1);
    }
    ```

    Why would this method not compile without wildcard capture? Hint: Try to supply an explicit type `Lists.<___>swapHelper(result, 0, 1)`.

14. Implement an improved version of the `closeAll` method in Section 6.3, "Type Bounds" (page 210). Close all elements even if some of them throw an exception. In that case, throw an exception afterwards. If two or more calls throw an exception, chain them together.

15. Implement a method `map` that receives an array list and a `Function<T, R>` object (see Chapter 3), and that returns an array list consisting of the results of applying the function to the given elements.

16. What is the erasure of the following methods in the `Collection` class?

    ```
    public static <T extends Comparable<? super T>>
        void sort(List<T> list)
    public static <T extends Object & Comparable<? super T>>
        T max(Collection<? extends T> coll)
    ```

17. Define a class `Employee` that implements `Comparable<Employee>`. Using the `javap` utility, demonstrate that a bridge method has been synthesized. What does it do?

18. Consider the method

    ```
    public static <T> T[] repeat(int n, T obj, IntFunction<T[]> constr)
    ```

    in Section 6.6.3, "You Cannot Instantiate Type Variables" (page 221). The call `Arrays.repeat(10, 42, int[]::new)` will fail. Why? How can you fix that? What do you need to do for the other primitive types?

19. Consider the method

    ```
    public static <T> ArrayList<T> repeat(int n, T obj)
    ```

    in Section 6.6.3, "You Cannot Instantiate Type Variables" (page 221). This method had no trouble constructing an `ArrayList<T>` which contains an array of `T` values. Can you produce a `T[]` array from that array list without using a `Class` value or a constructor reference? If not, why not?

20. Implement the method

    ```
    @SafeVarargs public static final <T> T[] repeat(int n,  T... objs)
    ```

    Return an array with `n` copies of the given objects. Note that no `Class` value or constructor reference is required since you can reflectively increase `objs`.
21. Using the `@SafeVarargs` annotation, write a method that can construct arrays of generic types. For example,

    ```
    List<String>[] result = Arrays.<List<String>>construct(10);
        // Sets result to a List<String>[] of size 10
    ```
22. Improve the method `public static <V, T extends Throwable> V doWork(Callable<V> c, T ex) throws T` of Section 6.6.7, "Exceptions and Generics" (page 225) so that one doesn't have to pass an exception object, which may never get used. Instead, accept a constructor reference for the exception class.
23. In the cautionary note at the end of Section 6.6.7, "Exceptions and Generics" (page 225), the `throwAs` helper method is used to "cast" `ex` into a `RuntimeException` and rethrow it. Why can't you use a regular cast, i.e. `throw (RuntimeException) ex`?
24. Which methods can you call on a variable of type `Class<?>` without using casts?
25. Write a method `public static String genericDeclaration(Method m)` that returns the declaration of the method `m` listing the type parameters with their bounds and the types of the method parameters, including their type arguments if they are generic types.