CHAPTER 5

Generics

IN release 1.5, *generics* were added to Java. Before generics, you had to cast every object you read from a collection. If someone accidentally inserted an object of the wrong type, casts could fail at runtime. With generics, you tell the compiler what types of objects are permitted in each collection. The compiler inserts casts for you automatically and tells you at compile time if you try to insert an object of the wrong type. This results in programs that are both safer and clearer, but these benefits come with complications. This chapter tells you how to maximize the benefits and minimize the complications. For a more detailed treatment of this material, see Langer's tutorial [Langer08] or Naftalin and Wadler's book [Naftalin07].

Item 23: Don't use raw types in new code

First, a few terms. A class or interface whose declaration has one or more *type parameters* is a *generic* class or interface [JLS, 8.1.2, 9.1.2]. For example, as of release 1.5, the `List` interface has a single type parameter, `E`, representing the element type of the list. Technically the name of the interface is now `List<E>` (read "list of E"), but people often call it `List` for short. Generic classes and interfaces are collectively known as *generic types*.

Each generic type defines a set of *parameterized types*, which consist of the class or interface name followed by an angle-bracketed list of *actual type parameters* corresponding to the generic type's formal type parameters [JLS, 4.4, 4.5]. For example, `List<String>` (read "list of string") is a parameterized type representing a list whose elements are of type `String`. (`String` is the actual type parameter corresponding to the formal type parameter `E`.)

Finally, each generic type defines a *raw type*, which is the name of the generic type used without any accompanying actual type parameters [JLS, 4.8]. For exam-

ple, the raw type corresponding to `List<E>` is `List`. Raw types behave as if all of the generic type information were erased from the type declaration. For all practical purposes, the raw type `List` behaves the same way as the interface type `List` did before generics were added to the platform.

Before release 1.5, this would have been an exemplary collection declaration:

```
// Now a raw collection type - don't do this!

/**
 * My stamp collection. Contains only Stamp instances.
 */
private final Collection stamps = ... ;
```

If you accidentally put a coin into your stamp collection, the erroneous insertion compiles and runs without error:

```
// Erroneous insertion of coin into stamp collection
stamps.add(new Coin( ... ));
```

You don't get an error until you retrieve the coin from the stamp collection:

```
// Now a raw iterator type - don't do this!
for (Iterator i = stamps.iterator(); i.hasNext(); ) {
    Stamp s = (Stamp) i.next(); // Throws ClassCastException
    ... // Do something with the stamp
}
```

As mentioned throughout this book, it pays to discover errors as soon as possible after they are made, ideally at compile time. In this case, you don't discover the error till runtime, long after it has happened, and in code that is far removed from the code containing the error. Once you see the `ClassCastException`, you have to search through the code base looking for the method invocation that put the coin into the stamp collection. The compiler can't help you, because it can't understand the comment that says, "`Contains only Stamp instances`."

With generics, you replace the comment with an improved type declaration for the collection that tells the compiler the information that was previously hidden in the comment:

```
// Parameterized collection type - typesafe
private final Collection<Stamp> stamps = ... ;
```

From this declaration the compiler knows that `stamps` should contain only `Stamp` instances and *guarantees* this to be the case, assuming your entire code base is compiled with a compiler from release 1.5 or later and all the code compiles without emitting (or suppressing; see Item 24) any warnings. When `stamps` is declared with a parameterized type, the erroneous insertion generates a compile-time error message that tells you exactly what is wrong:

```
Test.java:9: add(Stamp) in Collection<Stamp> cannot be applied
to (Coin)
    stamps.add(new Coin());
               ^
```

As an added benefit, you no longer have to cast manually when removing elements from collections. The compiler inserts invisible casts for you and guarantees that they won't fail (assuming, again, that all of your code was compiled with a generics-aware compiler and did not produce or suppress any warnings). This is true whether you use a for-each loop (Item 46):

```
// for-each loop over a parameterized collection - typesafe
for (Stamp s : stamps) { // No cast
    ... // Do something with the stamp
}
```

or a traditional `for` loop:

```
// for loop with parameterized iterator declaration - typesafe
for (Iterator<Stamp> i = stamps.iterator(); i.hasNext(); ) {
    Stamp s = i.next(); // No cast necessary
    ... // Do something with the stamp
}
```

While the prospect of accidentally inserting a coin into a stamp collection may appear far-fetched, the problem is real. For example, it is easy to imagine someone putting a `java.util.Date` instance into a collection that is supposed to contain only `java.sql.Date` instances.

As noted above, it is still legal to use collection types and other generic types without supplying type parameters, but you should not do it. **If you use raw types, you lose all the safety and expressiveness benefits of generics.** Given that you shouldn't use raw types, why did the language designers allow them? To provide compatibility. The Java platform was about to enter its second decade when generics were introduced, and there was an enormous amount of Java code in

existence that did not use generics. It was deemed critical that all of this code remain legal and interoperable with new code that does use generics. It had to be legal to pass instances of parameterized types to methods that were designed for use with ordinary types, and vice versa. This requirement, known as *migration compatibility*, drove the decision to support raw types.

While you shouldn't use raw types such as `List` in new code, it is fine to use types that are parameterized to allow insertion of arbitrary objects, such as `List<Object>`. Just what is the difference between the raw type `List` and the parameterized type `List<Object>`? Loosely speaking, the former has opted out of generic type checking, while the latter has explicitly told the compiler that it is capable of holding objects of any type. While you can pass a `List<String>` to a parameter of type `List`, you can't pass it to a parameter of type `List<Object>`. There are subtyping rules for generics, and `List<String>` is a subtype of the raw type `List`, but not of the parameterized type `List<Object>` (Item 25). As a consequence, **you lose type safety if you use a raw type like `List`, but not if you use a parameterized type like `List<Object>`.**

To make this concrete, consider the following program:

```
// Uses raw type (List) - fails at runtime!
public static void main(String[] args) {
    List<String> strings = new ArrayList<String>();
    unsafeAdd(strings, new Integer(42));
    String s = strings.get(0); // Compiler-generated cast
}

private static void unsafeAdd(List list, Object o) {
    list.add(o);
}
```

This program compiles, but because it uses the raw type `List`, you get a warning:

```
Test.java:10: warning: unchecked call to add(E) in raw type List
    list.add(o);
             ^
```

And indeed, if you run the program, you get a `ClassCastException` when the program tries to cast the result of the invocation `strings.get(0)` to a `String`. This is a compiler-generated cast, so it's normally guaranteed to succeed, but in this case we ignored a compiler warning and paid the price.

If you replace the raw type `List` with the parameterized type `List<Object>` in the `unsafeAdd` declaration and try to recompile the program, you'll find that it no longer compiles. Here is the error message:

```
Test.java:5: unsafeAdd(List<Object>,Object) cannot be applied
to (List<String>,Integer)
    unsafeAdd(strings, new Integer(42));
    ^
```

You might be tempted to use a raw type for a collection whose element type is unknown and doesn't matter. For example, suppose you want to write a method that takes two sets and returns the number of elements they have in common. Here's how you might write such a method if you were new to generics:

```
// Use of raw type for unknown element type - don't do this!
static int numElementsInCommon(Set s1, Set s2) {
    int result = 0;
    for (Object o1 : s1)
        if (s2.contains(o1))
            result++;
    return result;
}
```

This method works but it uses raw types, which are dangerous. Since release 1.5, Java has provided a safe alternative known as *unbounded wildcard types*. If you want to use a generic type but you don't know or care what the actual type parameter is, you can use a question mark instead. For example, the unbounded wildcard type for the generic type `Set<E>` is `Set<?>` (read "set of some type"). It is the most general parameterized `Set` type, capable of holding *any* set. Here is how the `numElementsInCommon` method looks with unbounded wildcard types:

```
// Unbounded wildcard type - typesafe and flexible
static int numElementsInCommon(Set<?> s1, Set<?> s2) {
    int result = 0;
    for (Object o1 : s1)
        if (s2.contains(o1))
            result++;
    return result;
}
```

What is the difference between the unbounded wildcard type `Set<?>` and the raw type `Set`? Do the question marks really buy you anything? Not to belabor the point, but the wildcard type is safe and the raw type isn't. You can put any element

into a collection with a raw type, easily corrupting the collection's type invariant (as demonstrated by the `unsafeAdd` method on page 112); **you can't put any element (other than `null`) into a `Collection<?>`.** Attempting to do so will generate a compile-time error message like this:

```
WildCard.java:13: cannot find symbol
symbol  : method add(String)
location: interface Collection<capture#825 of ?>
        c.add("verboten");
          ^
```

Admittedly this error message leaves something to be desired, but the compiler has done its job, preventing you from corrupting the collection's type invariant. Not only can't you put any element (other than `null`) into a `Collection<?>`, but you can't assume anything about the type of the objects that you get out. If these restrictions are unacceptable, you can use *generic methods* (Item 27) or *bounded wildcard types* (Item 28).

There are two minor exceptions to the rule that you should not use raw types in new code, both of which stem from the fact that generic type information is erased at runtime (Item 25). **You must use raw types in class literals.** The specification does not permit the use of parameterized types (though it does permit array types and primitive types) [JLS, 15.8.2]. In other words, `List.class`, `String[].class`, and `int.class` are all legal, but `List<String>.class` and `List<?>.class` are not.

The second exception to the rule concerns the `instanceof` operator. Because generic type information is erased at runtime, it is illegal to use the `instanceof` operator on parameterized types other than unbounded wildcard types. The use of unbounded wildcard types in place of raw types does not affect the behavior of the `instanceof` operator in any way. In this case, the angle brackets and question marks are just noise. **This is the preferred way to use the `instanceof` operator with generic types:**

```
// Legitimate use of raw type - instanceof operator
if (o instanceof Set) {       // Raw type
    Set<?> m = (Set<?>) o;    // Wildcard type
    ...
}
```

Note that once you've determined that `o` is a `Set`, you must cast it to the wildcard type `Set<?>`, not the raw type `Set`. This is a checked cast, so it will not cause a compiler warning.

In summary, using raw types can lead to exceptions at runtime, so don't use them in new code. They are provided only for compatibility and interoperability with legacy code that predates the introduction of generics. As a quick review, `Set<Object>` is a parameterized type representing a set that can contain objects of any type, `Set<?>` is a wildcard type representing a set that can contain only objects of some unknown type, and `Set` is a raw type, which opts out of the generic type system. The first two are safe and the last is not.

For quick reference, the terms introduced in this item (and a few introduced elsewhere in this chapter) are summarized in the following table:

Term	Example	Item
Parameterized type	`List<String>`	Item 23
Actual type parameter	`String`	Item 23
Generic type	`List<E>`	Items 23, 26
Formal type parameter	`E`	Item 23
Unbounded wildcard type	`List<?>`	Item 23
Raw type	`List`	Item 23
Bounded type parameter	`<E extends Number>`	Item 26
Recursive type bound	`<T extends Comparable<T>>`	Item 27
Bounded wildcard type	`List<? extends Number>`	Item 28
Generic method	`static <E> List<E> asList(E[] a)`	Item 27
Type token	`String.class`	Item 29

Item 24: Eliminate unchecked warnings

When you program with generics, you will see many compiler warnings: unchecked cast warnings, unchecked method invocation warnings, unchecked generic array creation warnings, and unchecked conversion warnings. The more experience you acquire with generics, the fewer warnings you'll get, but don't expect newly written code that uses generics to compile cleanly.

Many unchecked warnings are easy to eliminate. For example, suppose you accidentally write this declaration:

```
Set<Lark> exaltation = new HashSet();
```

The compiler will gently remind you what you did wrong:

```
Venery.java:4: warning: [unchecked] unchecked conversion
found   : HashSet, required: Set<Lark>
    Set<Lark> exaltation = new HashSet();
                           ^
```

You can then make the indicated correction, causing the warning to disappear:

```
Set<Lark> exaltation = new HashSet<Lark>();
```

Some warnings will be *much* more difficult to eliminate. This chapter is filled with examples of such warnings. When you get warnings that require some thought, persevere! **Eliminate every unchecked warning that you can.** If you eliminate all warnings, you are assured that your code is typesafe, which is a very good thing. It means that you won't get a `ClassCastException` at runtime, and it increases your confidence that your program is behaving as you intended.

If you can't eliminate a warning, and you can prove that the code that provoked the warning is typesafe, then (and only then) suppress the warning with an `@SuppressWarnings("unchecked")` annotation. If you suppress warnings without first proving that the code is typesafe, you are only giving yourself a false sense of security. The code may compile without emitting any warnings, but it can still throw a `ClassCastException` at runtime. If, however, you ignore unchecked warnings that you know to be safe (instead of suppressing them), you won't notice when a new warning crops up that represents a real problem. The new warning will get lost amidst all the false alarms that you didn't silence.

The SuppressWarnings annotation can be used at any granularity, from an individual local variable declaration to an entire class. **Always use the SuppressWarnings annotation on the smallest scope possible.** Typically this will be a variable declaration or a very short method or constructor. Never use SuppressWarnings on an entire class. Doing so could mask critical warnings.

If you find yourself using the SuppressWarnings annotation on a method or constructor that's more than one line long, you may be able to move it onto a local variable declaration. You may have to declare a new local variable, but it's worth it. For example, consider this toArray method, which comes from ArrayList:

```
public <T> T[] toArray(T[] a) {
    if (a.length < size)
        return (T[]) Arrays.copyOf(elements, size, a.getClass());
    System.arraycopy(elements, 0, a, 0, size);
    if (a.length > size)
        a[size] = null;
    return a;
}
```

If you compile ArrayList, the method generates this warning:

```
ArrayList.java:305: warning: [unchecked] unchecked cast
found   : Object[], required: T[]
    return (T[]) Arrays.copyOf(elements, size, a.getClass());
                               ^
```

It is illegal to put a SuppressWarnings annotation on the return statement, because it isn't a declaration [JLS, 9.7]. You might be tempted to put the annotation on the entire method, but don't. Instead, declare a local variable to hold the return value and annotate its declaration, like so:

```
// Adding local variable to reduce scope of @SuppressWarnings
public <T> T[] toArray(T[] a) {
    if (a.length < size) {
        // This cast is correct because the array we're creating
        // is of the same type as the one passed in, which is T[].
        @SuppressWarnings("unchecked") T[] result =
            (T[]) Arrays.copyOf(elements, size, a.getClass());
        return result;
    }
    System.arraycopy(elements, 0, a, 0, size);
    if (a.length > size)
        a[size] = null;
    return a;
}
```

This method compiles cleanly and minimizes the scope in which unchecked warnings are suppressed.

Every time you use an `@SuppressWarnings("unchecked")` annotation, add a comment saying why it's safe to do so. This will help others understand the code, and more importantly, it will decrease the odds that someone will modify the code so as to make the computation unsafe. If you find it hard to write such a comment, keep thinking. You may end up figuring out that the unchecked operation isn't safe after all.

In summary, unchecked warnings are important. Don't ignore them. Every unchecked warning represents the potential for a `ClassCastException` at runtime. Do your best to eliminate these warnings. If you can't eliminate an unchecked warning and you can prove that the code that provoked it is typesafe, suppress the warning with an `@SuppressWarnings("unchecked")` annotation in the narrowest possible scope. Record the rationale for your decision to suppress the warning in a comment.

Item 25: Prefer lists to arrays

Arrays differ from generic types in two important ways. First, arrays are *covariant*. This scary-sounding word means simply that if `Sub` is a subtype of `Super`, then the array type `Sub[]` is a subtype of `Super[]`. Generics, by contrast, are *invariant*: for any two distinct types `Type1` and `Type2`, `List<Type1>` is neither a subtype nor a supertype of `List<Type2>` [JLS, 4.10; Naftalin07, 2.5]. You might think this means that generics are deficient, but arguably it is arrays that are deficient.

This code fragment is legal:

```
// Fails at runtime!
Object[] objectArray = new Long[1];
objectArray[0] = "I don't fit in"; // Throws ArrayStoreException
```

but this one is not:

```
// Won't compile!
List<Object> ol = new ArrayList<Long>(); // Incompatible types
ol.add("I don't fit in");
```

Either way you can't put a `String` into a `Long` container, but with an array you find out that you've made a mistake at runtime; with a list, you find out at compile time. Of course you'd rather find out at compile time.

The second major difference between arrays and generics is that arrays are *reified* [JLS, 4.7]. This means that arrays know and enforce their element types at runtime. As noted above, if you try to store a `String` into an array of `Long`, you'll get an `ArrayStoreException`. Generics, by contrast, are implemented by *erasure* [JLS, 4.6]. This means that they enforce their type constraints only at compile time and discard (or *erase*) their element type information at runtime. Erasure is what allows generic types to interoperate freely with legacy code that does not use generics (Item 23).

Because of these fundamental differences, arrays and generics do not mix well. For example, it is illegal to create an array of a generic type, a parameterized type, or a type parameter. None of these array creation expressions are legal: `new List<E>[]`, `new List<String>[]`, `new E[]`. All will result in *generic array creation* errors at compile time.

Why is it illegal to create a generic array? Because it isn't typesafe. If it were legal, casts generated by the compiler in an otherwise correct program could fail at runtime with a `ClassCastException`. This would violate the fundamental guarantee provided by the generic type system.

To make this more concrete, consider the following code fragment:

```
// Why generic array creation is illegal - won't compile!
List<String>[] stringLists = new List<String>[1];  // (1)
List<Integer> intList = Arrays.asList(42);         // (2)
Object[] objects = stringLists;                    // (3)
objects[0] = intList;                              // (4)
String s = stringLists[0].get(0);                  // (5)
```

Let's pretend that line 1, which creates a generic array, is legal. Line 2 creates and initializes a `List<Integer>` containing a single element. Line 3 stores the `List<String>` array into an `Object` array variable, which is legal because arrays are covariant. Line 4 stores the `List<Integer>` into the sole element of the `Object` array, which succeeds because generics are implemented by erasure: the runtime type of a `List<Integer>` instance is simply `List`, and the runtime type of a `List<String>[]` instance is `List[]`, so this assignment doesn't generate an `ArrayStoreException`. Now we're in trouble. We've stored a `List<Integer>` instance into an array that is declared to hold only `List<String>` instances. In line 5, we retrieve the sole element from the sole list in this array. The compiler automatically casts the retrieved element to `String`, but it's an `Integer`, so we get a `ClassCastException` at runtime. In order to prevent this from happening, line 1 (which creates a generic array) generates a compile-time error.

Types such as `E`, `List<E>`, and `List<String>` are technically known as *non-reifiable* types [JLS, 4.7]. Intuitively speaking, a non-reifiable type is one whose runtime representation contains less information than its compile-time representation. The only parameterized types that are reifiable are unbounded wildcard types such as `List<?>` and `Map<?,?>` (Item 23). It is legal, though infrequently useful, to create arrays of unbounded wildcard types.

The prohibition on generic array creation can be annoying. It means, for example, that it's not generally possible for a generic type to return an array of its element type (but see Item 29 for a partial solution). It also means that you can get confusing warnings when using varargs methods (Item 42) in combination with generic types. This is because every time you invoke a varargs method, an array is created to hold the varargs parameters. If the element type of this array is not reifiable, you get a warning. There is little you can do about these warnings other than to suppress them (Item 24), and to avoid mixing generics and varargs in your APIs.

When you get a generic array creation error, the best solution is often to use the collection type `List<E>` in preference to the array type `E[]`. You might sacrifice some performance or conciseness, but in exchange you get better type safety and interoperability.

For example, suppose you have a synchronized list (of the sort returned by `Collections.synchronizedList`) and a function that takes two values of the type held by the list and returns a third. Now suppose you want to write a method to "reduce" the list by applying the function across it. If the list contains integers and the function adds two integer values, the `reduce` method returns the sum of all the values in the list. If the function multiplies two integer values, the method returns the product of the values in the list. If the list contains strings and the function concatenates two strings, the method returns a string consisting of all the strings in the list in sequence. In addition to a list and a function, the `reduce` method takes an initial value for the reduction, which is returned if the list is empty. (The initial value is typically the identity element for the function, which is `0` for addition, `1` for multiplication, and `""` for string concatenation.) Here's how the code might have looked without generics:

```
// Reduction without generics, and with concurrency flaw!
static Object reduce(List list, Function f, Object initVal) {
    synchronized(list) {
        Object result = initVal;
        for (Object o : list)
            result = f.apply(result, o);
        return result;
    }
}

interface Function {
    Object apply(Object arg1, Object arg2);
}
```

Now, suppose you've read Item 67, which tells you not to call "alien methods" from a synchronized region. So, you modify the `reduce` method to copy the contents of the list while holding the lock, which allows you to perform the reduction on the copy. Prior to release 1.5, the natural way to do this would have been using `List`'s `toArray` method (which locks the list internally):

```
// Reduction without generics or concurrency flaw
static Object reduce(List list, Function f, Object initVal) {
    Object[] snapshot = list.toArray(); // Locks list internally
    Object result = initVal;
    for (Object o : snapshot)
        result = f.apply(result, o);
    return result;
}
```

If you try to do this with generics you'll get into trouble of the sort that we discussed above. Here's a generic version of the `Function` interface:

```
interface Function<T> {
    T apply(T arg1, T arg2);
}
```

And here's a naive attempt to apply generics to the revised version of the `reduce` method. This is a *generic method* (Item 27). Don't worry if you don't understand the declaration. For the purposes of this item, you should concentrate on the method body:

```
// Naive generic version of reduction - won't compile!
static <E> E reduce(List<E> list, Function<E> f, E initVal) {
    E[] snapshot = list.toArray();  // Locks list
    E result = initVal;
    for (E e : snapshot)
        result = f.apply(result, e);
    return result;
}
```

If you try to compile this method, you'll get the following error:

```
Reduce.java:12: incompatible types
found   : Object[], required: E[]
        E[] snapshot = list.toArray();  // Locks list
                                    ^
```

No big deal, you say, I'll cast the `Object` array to an `E` array:

```
E[] snapshot = (E[]) list.toArray();
```

That gets rid of the error, but now you get a warning:

```
Reduce.java:12: warning: [unchecked] unchecked cast
found   : Object[], required: E[]
        E[] snapshot = (E[]) list.toArray();  // Locks list
                                            ^
```

The compiler is telling you that it can't check the safety of the cast at runtime because it doesn't know what `E` is at runtime—remember, element type information is erased from generics at runtime. Will this program work? Yes, it turns out that it will, but it isn't safe. With minor modifications, you could get it to throw a

`ClassCastException` on a line that doesn't contain an explicit cast. The compile-time type of `snapshot` is `E[]` which could be `String[]`, `Integer[]`, or any other kind of array. The runtime type is `Object[]`, and that's dangerous. Casts to arrays of non-reifiable types should be used only under special circumstances (Item 26).

So what should you do? Use a list instead of an array. Here is a version of the `reduce` method that compiles without error or warning:

```
// List-based generic reduction
static <E> E reduce(List<E> list, Function<E> f, E initVal) {
    List<E> snapshot;
    synchronized(list) {
        snapshot = new ArrayList<E>(list);
    }
    E result = initVal;
    for (E e : snapshot)
        result = f.apply(result, e);
    return result;
}
```

This version is a tad more verbose than the array version, but it's worth it for the peace of mind that comes from knowing you won't get a `ClassCastException` at runtime.

In summary, arrays and generics have very different type rules. Arrays are covariant and reified; generics are invariant and erased. As a consequence, arrays provide runtime type safety but not compile-time type safety and vice versa for generics. Generally speaking, arrays and generics don't mix well. If you find yourself mixing them and getting compile-time errors or warnings, your first impulse should be to replace the arrays with lists.

Item 26: Favor generic types

It is generally not too difficult to parameterize your collection declarations and make use of the generic types and methods provided by the JDK. Writing your own generic types is a bit more difficult, but it's worth the effort to learn how.

Consider the simple stack implementation from Item 6:

```
// Object-based collection - a prime candidate for generics
public class Stack {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack() {
        elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        Object result = elements[--size];
        elements[size] = null; // Eliminate obsolete reference
        return result;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    private void ensureCapacity() {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}
```

This class is a prime candidate for *generification*, in other words, for being compatibly enhanced to take advantage of generic types. As it stands, you have to cast objects that are popped off the stack, and those casts might fail at runtime. The first step in generifying a class is to add one or more type parameters to its decla-

ration. In this case there is one type parameter, representing the element type of the stack, and the conventional name for this parameter is `E` (Item 44).

The next step is to replace all the uses of the type `Object` with the appropriate type parameter, and then try to compile the resulting program:

```
// Initial attempt to generify Stack = won't compile!
public class Stack<E> {
    private E[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack() {
        elements = new E[DEFAULT_INITIAL_CAPACITY];
    }

    public void push(E e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public E pop() {
        if (size==0)
            throw new EmptyStackException();
        E result = elements[--size];
        elements[size] = null; // Eliminate obsolete reference
        return result;
    }
    ... // no changes in isEmpty or ensureCapacity
}
```

You'll generally get at least one error or warning, and this class is no exception. Luckily, this class generates only one error:

```
Stack.java:8: generic array creation
        elements = new E[DEFAULT_INITIAL_CAPACITY];
                   ^
```

As explained in Item 25, you can't create an array of a non-reifiable type, such as `E`. This problem arises every time you write a generic type that is backed by an array. There are two ways to solve it. The first solution directly circumvents the prohibition on generic array creation: create an array of `Object` and cast it to the

generic array type. Now in place of an error, the compiler will emit a warning. This usage is legal, but it's not (in general) typesafe:

```
Stack.java:8: warning: [unchecked] unchecked cast
found   : Object[], required: E[]
        elements = (E[]) new Object[DEFAULT_INITIAL_CAPACITY];
                   ^
```

The compiler may not be able to prove that your program is typesafe, but you can. You must convince yourself that the unchecked cast will not compromise the type safety of the program. The array in question (`elements`) is stored in a private field and never returned to the client or passed to any other method. The only elements stored in the array are those passed to the `push` method, which are of type `E`, so the unchecked cast can do no harm.

Once you've proved that an unchecked cast is safe, suppress the warning in as narrow a scope as possible (Item 24). In this case, the constructor contains only the unchecked array creation, so it's appropriate to suppress the warning in the entire constructor. With the addition of an annotation to do this, `Stack` compiles cleanly and you can use it without explicit casts or fear of a `ClassCastException`:

```
// The elements array will contain only E instances from push(E).
// This is sufficient to ensure type safety, but the runtime
// type of the array won't be E[]; it will always be Object[]!
@SuppressWarnings("unchecked")
public Stack() {
    elements = (E[]) new Object[DEFAULT_INITIAL_CAPACITY];
}
```

The second way to eliminate the generic array creation error in `Stack` is to change the type of the field `elements` from `E[]` to `Object[]`. If you do this, you'll get a different error:

```
Stack.java:19: incompatible types
found   : Object, required: E
        E result = elements[--size];
                   ^
```

You can change this error into a warning by casting the element retrieved from the array from `Object` to `E`:

```
Stack.java:19: warning: [unchecked] unchecked cast
found   : Object, required: E
        E result = (E) elements[--size];
                       ^
```

Because E is a non-reifiable type, there's no way the compiler can check the cast at runtime. Again, you can easily prove to yourself that the unchecked cast is safe, so it's appropriate to suppress the warning. In line with the advice of Item 24, we suppress the warning only on the assignment that contains the unchecked cast, not on the entire pop method:

```
// Appropriate suppression of unchecked warning
public E pop() {
    if (size==0)
        throw new EmptyStackException();

    // push requires elements to be of type E, so cast is correct
    @SuppressWarnings("unchecked") E result =
        (E) elements[--size];

    elements[size] = null; // Eliminate obsolete reference
    return result;
}
```

Which of the two techniques you choose for dealing with the generic array creation error is largely a matter of taste. All other things being equal, it is riskier to suppress an unchecked cast to an array type than to a scalar type, which would suggest the second solution. But in a more realistic generic class than Stack, you would probably be reading from the array at many points in the code, so choosing the second solution would require many casts to E rather than a single cast to E[], which is why the first solution is used more commonly [Naftalin07, 6.7].

The following program demonstrates the use of our generic Stack class. The program prints its command line arguments in reverse order and converted to uppercase. No explicit cast is necessary to invoke String's toUpperCase method on the elements popped from the stack, and the automatically generated cast is guaranteed to succeed:

```
// Little program to exercise our generic Stack
public static void main(String[] args) {
    Stack<String> stack = new Stack<String>();
    for (String arg : args)
        stack.push(arg);
    while (!stack.isEmpty())
        System.out.println(stack.pop().toUpperCase());
}
```

The foregoing example may appear to contradict Item 25, which encourages the use of lists in preference to arrays. It is not always possible or desirable to use lists inside your generic types. Java doesn't support lists natively, so some generic types, such as `ArrayList`, *must* be implemented atop arrays. Other generic types, such as `HashMap`, are implemented atop arrays for performance.

The great majority of generic types are like our `Stack` example in that their type parameters have no restrictions: you can create a `Stack<Object>`, `Stack<int[]>`, `Stack<List<String>>`, or a `Stack` of any other object reference type. Note that you can't create a `Stack` of a primitive type: trying to create a `Stack<int>` or `Stack<double>` will result in a compile-time error. This is a fundamental limitation of Java's generic type system. You can work around this restriction by using boxed primitive types (Item 49).

There are some generic types that restrict the permissible values of their type parameters. For example, consider `java.util.concurrent.DelayQueue`, whose declaration looks like this:

```
class DelayQueue<E extends Delayed> implements BlockingQueue<E>;
```

The type parameter list (`<E extends Delayed>`) requires that the actual type parameter `E` must be a subtype of `java.util.concurrent.Delayed`. This allows the `DelayQueue` implementation and its clients to take advantage of `Delayed` methods on the elements of a `DelayQueue`, without the need for explicit casting or the risk of a `ClassCastException`. The type parameter `E` is known as a *bounded type parameter*. Note that the subtype relation is defined so that every type is a subtype of itself [JLS, 4.10], so it is legal to create a `DelayQueue<Delayed>`.

In summary, generic types are safer and easier to use than types that require casts in client code. When you design new types, make sure that they can be used without such casts. This will often mean making the types generic. Generify your existing types as time permits. This will make life easier for new users of these types without breaking existing clients (Item 23).

Item 27: Favor generic methods

Just as classes can benefit from generification, so can methods. Static utility methods are particularly good candidates for generification. All of the "algorithm" methods in `Collections` (such as `binarySearch` and `sort`) have been generified.

Writing generic methods is similar to writing generic types. Consider this method, which returns the union of two sets:

```
// Uses raw types - unacceptable! (Item 23)
public static Set union(Set s1, Set s2) {
    Set result = new HashSet(s1);
    result.addAll(s2);
    return result;
}
```

This method compiles, but with two warnings:

```
Union.java:5: warning: [unchecked] unchecked call to
HashSet(Collection<? extends E>) as a member of raw type HashSet
        Set result = new HashSet(s1);
                     ^
Union.java:6: warning: [unchecked] unchecked call to
addAll(Collection<? extends E>) as a member of raw type Set
        result.addAll(s2);
                      ^
```

To fix these warnings and make the method typesafe, modify the method declaration to declare a type parameter representing the element type for the three sets (two arguments and the return value) and use the type parameter in the method. **The type parameter list, which declares the type parameter, goes between the method's modifiers and its return type.** In this example, the type parameter list is `<E>` and the return type is `Set<E>`. The naming conventions for type parameters are the same for generic methods as for generic types (Items 26, 44):

```
// Generic method
public static <E> Set<E> union(Set<E> s1, Set<E> s2) {
    Set<E> result = new HashSet<E>(s1);
    result.addAll(s2);
    return result;
}
```

At least for simple generic methods, that's all there is to it. Now the method compiles without generating any warnings and provides type safety as well as ease

of use. Here's a simple program to exercise our method. The program contains no casts and compiles without errors or warnings:

```
// Simple program to exercise generic method
public static void main(String[] args) {
    Set<String> guys = new HashSet<String>(
        Arrays.asList("Tom", "Dick", "Harry"));
    Set<String> stooges = new HashSet<String>(
        Arrays.asList("Larry", "Moe", "Curly"));
    Set<String> aflCio = union(guys, stooges);
    System.out.println(aflCio);
}
```

When you run the program, it prints `[Moe, Harry, Tom, Curly, Larry, Dick]`. The order of the elements is implementation-dependent.

A limitation of the `union` method is that the types of all three sets (both input parameters and the return value) have to be the same. You can make the method more flexible by using *bounded wildcard types* (Item 28).

One noteworthy feature of generic methods is that you needn't specify the value of the type parameter explicitly as you must when invoking generic constructors. The compiler figures out the value of the type parameters by examining the types of the method arguments. In the case of the program above, the compiler sees that both arguments to `union` are of type `Set<String>`, so it knows that the type parameter `E` must be `String`. This process is called *type inference*.

As discussed in Item 1, you can exploit the type inference provided by generic method invocation to ease the process of creating parameterized type instances. To refresh your memory, the need to pass the values of type parameters explicitly when invoking generic constructors can be annoying. The type parameters appear redundantly on the left- and right-hand sides of variable declarations:

```
// Parameterized type instance creation with constructor
Map<String, List<String>> anagrams =
    new HashMap<String, List<String>>();
```

To eliminate this redundancy, write a *generic static factory method* corresponding to each constructor that you want to use. For example, here is a generic static factory method corresponding to the parameterless `HashMap` constructor:

```
// Generic static factory method
public static <K,V> HashMap<K,V> newHashMap() {
    return new HashMap<K,V>();
}
```

With this generic static factory method, you can replace the repetitious declaration above with this concise one:

```
// Parameterized type instance creation with static factory
Map<String, List<String>> anagrams = newHashMap();
```

It would be nice if the language did the same kind of type inference when invoking constructors on generic types as it does when invoking generic methods. Someday it might, but as of release 1.6, it does not.

A related pattern is the *generic singleton factory.* On occasion, you will need to create an object that is immutable but applicable to many different types. Because generics are implemented by erasure (Item 25), you can use a single object for all required type parameterizations, but you need to write a static factory method to repeatedly dole out the object for each requested type parameterization. This pattern is most frequently used for function objects (Item 21) such as `Collections.reverseOrder`, but it is also used for collections such as `Collections.emptySet`.

Suppose you have an interface that describes a function that accepts and returns a value of some type `T`:

```
public interface UnaryFunction<T> {
    T apply(T arg);
}
```

Now suppose that you want to provide an identity function. It would be wasteful to create a new one each time it's required, as it's stateless. If generics were reified, you would need one identity function per type, but since they're erased you need only a generic singleton. Here's how it looks:

```
// Generic singleton factory pattern
private static UnaryFunction<Object> IDENTITY_FUNCTION =
    new UnaryFunction<Object>() {
        public Object apply(Object arg) { return arg; }
    };

// IDENTITY_FUNCTION is stateless and its type parameter is
// unbounded so it's safe to share one instance across all types.
@SuppressWarnings("unchecked")
public static <T> UnaryFunction<T> identityFunction() {
    return (UnaryFunction<T>) IDENTITY_FUNCTION;
}
```

The cast of `IDENTITY_FUNCTION` to `(UnaryFunction<T>)` generates an unchecked cast warning, as `UnaryFunction<Object>` is not a `UnaryFunction<T>` for every `T`. But the identity function is special: it returns its argument unmodified, so we know that it is typesafe to use it as a `UnaryFunction<T>` whatever the value of `T`. Therefore, we can confidently suppress the unchecked cast warning that is generated by this cast. Once we've done this, the code compiles without error or warning.

Here is a sample program that uses our generic singleton as a `UnaryFunction<String>` and a `UnaryFunction<Number>`. As usual, it contains no casts and compiles without errors or warnings:

```
// Sample program to exercise generic singleton
public static void main(String[] args) {
    String[] strings = { "jute", "hemp", "nylon" };
    UnaryFunction<String> sameString = identityFunction();
    for (String s : strings)
        System.out.println(sameString.apply(s));

    Number[] numbers = { 1, 2.0, 3L };
    UnaryFunction<Number> sameNumber = identityFunction();
    for (Number n : numbers)
        System.out.println(sameNumber.apply(n));
}
```

It is permissible, though relatively rare, for a type parameter to be bounded by some expression involving that type parameter itself. This is what's known as a *recursive type bound*. The most common use of recursive type bounds is in connection with the `Comparable` interface, which defines a type's natural ordering:

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

The type parameter `T` defines the type to which elements of the type implementing `Comparable<T>` can be compared. In practice, nearly all types can be compared only to elements of their own type. So, for example, `String` implements `Comparable<String>`, `Integer` implements `Comparable<Integer>`, and so on.

There are many methods that take a list of elements that implement `Comparable`, in order to sort the list, search within it, calculate its minimum or maximum, and the like. To do any of these things, it is required that every element in the list

be comparable to every other element in the list, in other words, that the elements of the list be *mutually comparable*. Here is how to express that constraint:

```
// Using a recursive type bound to express mutual comparability
public static <T extends Comparable<T>> T max(List<T> list) {...}
```

The type bound `<T extends Comparable<T>>` may be read as "for every type `T` that can be compared to itself," which corresponds more or less exactly to the notion of mutual comparability.

Here is a method to go with the declaration above. It calculates the maximum value of a list according to its elements' natural order, and it compiles without errors or warnings:

```
// Returns the maximum value in a list - uses recursive type bound
public static <T extends Comparable<T>> T max(List<T> list) {
    Iterator<T> i = list.iterator();
    T result = i.next();
    while (i.hasNext()) {
        T t = i.next();
        if (t.compareTo(result) > 0)
            result = t;
    }
    return result;
}
```

Recursive type bounds can get much more complex than this, but luckily it doesn't happen too often. If you understand this idiom, and its wildcard variant (Item 28), you'll be able to deal with many of the recursive type bounds that you see in practice.

In summary, generic methods, like generic types, are safer and easier to use than methods that require their clients to cast input parameters and return values. Like types, you should make sure that your new methods can be used without casts, which will often mean making them generic. And like types, you should generify your existing methods to make life easier for new users without breaking existing clients (Item 23).

Item 28: Use bounded wildcards to increase API flexibility

As noted in Item 25, parameterized types are *invariant*. In other words, for any two distinct types `Type1` and `Type2`, `List<Type1>` is neither a subtype nor a supertype of `List<Type2>`. While it is counterintuitive that `List<String>` is not a subtype of `List<Object>`, it really does make sense. You can put any object into a `List<Object>`, but you can put only strings into a `List<String>`.

Sometimes you need more flexibility than invariant typing can provide. Consider the stack from Item 26. To refresh your memory, here is its public API:

```
public class Stack<E> {
    public Stack();
    public void push(E e);
    public E pop();
    public boolean isEmpty();
}
```

Suppose we want to add a method that takes a sequence of elements and pushes them all onto the stack. Here's a first attempt:

```
// pushAll method without wildcard type - deficient!
public void pushAll(Iterable<E> src) {
    for (E e : src)
        push(e);
}
```

This method compiles cleanly, but it isn't entirely satisfactory. If the element type of the `Iterable` `src` exactly matches that of the stack, it works fine. But suppose you have a `Stack<Number>` and you invoke `push(intVal)`, where `intVal` is of type `Integer`. This works, because `Integer` is a subtype of `Number`. So logically, it seems that this should work, too:

```
Stack<Number> numberStack = new Stack<Number>();
Iterable<Integer> integers = ... ;
numberStack.pushAll(integers);
```

If you try it, however, you'll get this error message because, as noted above, parameterized types are invariant:

```
StackTest.java:7: pushAll(Iterable<Number>) in Stack<Number>
cannot be applied to (Iterable<Integer>)
        numberStack.pushAll(integers);
                    ^
```

Luckily, there's a way out. The language provides a special kind of parameterized type call a *bounded wildcard type* to deal with situations like this. The type of the input parameter to `pushAll` should not be "`Iterable` of `E`" but "`Iterable` of some subtype of `E`," and there is a wildcard type that means precisely that: `Iterable<? extends E>`. (The use of the keyword `extends` is slightly misleading: recall from Item 26 that *subtype* is defined so that every type is a subtype of itself, even though it does not extend itself.) Let's modify `pushAll` to use this type:

```
// Wildcard type for parameter that serves as an E producer
public void pushAll(Iterable<? extends E> src) {
    for (E e : src)
        push(e);
}
```

With this change, not only does `Stack` compile cleanly, but so does the client code that wouldn't compile with the original `pushAll` declaration. Because `Stack` and its client compile cleanly, you know that everything is typesafe.

Now suppose you want to write a `popAll` method to go with `pushAll`. The `popAll` method pops each element off the stack and adds the elements to the given collection. Here's how a first attempt at writing the `popAll` method might look:

```
// popAll method without wildcard type - deficient!
public void popAll(Collection<E> dst) {
    while (!isEmpty())
        dst.add(pop());
}
```

Again, this compiles cleanly and works fine if the element type of the destination collection exactly matches that of the stack. But again, it doesn't seem entirely satisfactory. Suppose you have a `Stack<Number>` and variable of type `Object`. If you pop an element from the stack and store it in the variable, it compiles and runs without error. So shouldn't you be able to do this, too?

```
Stack<Number> numberStack = new Stack<Number>();
Collection<Object> objects = ... ;
numberStack.popAll(objects);
```

If you try to compile this client code against the version of `popAll` above, you'll get an error very similar to the one that we got with our first version of `pushAll`: `Collection<Object>` is not a subtype of `Collection<Number>`. Once again, wildcard types provide a way out. The type of the input parameter to `popAll`

should not be "collection of `E`" but "collection of some supertype of `E`" (where supertype is defined such that `E` is a supertype of itself [JLS, 4.10]). Again, there is a wildcard type that means precisely that: `Collection<? super E>`. Let's modify `popAll` to use it:

```
// Wildcard type for parameter that serves as an E consumer
public void popAll(Collection<? super E> dst) {
    while (!isEmpty())
        dst.add(pop());
}
```

With this change, both `Stack` and the client code compile cleanly.

The lesson is clear. **For maximum flexibility, use wildcard types on input parameters that represent producers or consumers.** If an input parameter is both a producer and a consumer, then wildcard types will do you no good: you need an exact type match, which is what you get without any wildcards.

Here is a mnemonic to help you remember which wildcard type to use:

PECS stands for producer-`extends`, consumer-`super`.

In other words, if a parameterized type represents a `T` producer, use `<? extends T>`; if it represents a `T` consumer, use `<? super T>`. In our `Stack` example, `pushAll`'s `src` parameter produces `E` instances for use by the `Stack`, so the appropriate type for `src` is `Iterable<? extends E>`; `popAll`'s `dst` parameter consumes `E` instances from the `Stack`, so the appropriate type for `dst` is `Collection<? super E>`. The PECS mnemonic captures the fundamental principle that guides the use of wildcard types. Naftalin and Wadler call it the *Get and Put Principle* [Naftalin07, 2.4].

With this mnemonic in mind, let's take a look at some method declarations from previous items. The `reduce` method in Item 25 has this declaration:

```
static <E> E reduce(List<E> list, Function<E> f, E initVal)
```

Although lists can both consume and produce values, the `reduce` method uses its `list` parameter only as an `E` **producer**, so its declaration should use a wildcard type that **`extends`** `E`. The parameter `f` represents a function that both consumes and produces `E` instances, so a wildcard type would be inappropriate for it. Here's the resulting method declaration:

```
// Wildcard type for parameter that serves as an E producer
static <E> E reduce(List<? extends E> list, Function<E> f,
                    E initVal)
```

And would this change make any difference in practice? As it turns out, it would. Suppose you have a `List<Integer>`, and you want to reduce it with a `Function<Number>`. This would not compile with the original declaration, but it does once you add the bounded wildcard type.

Now let's look at the `union` method from Item 27. Here is the declaration:

```
public static <E> Set<E> union(Set<E> s1, Set<E> s2)
```

Both parameters, `s1` and `s2`, are `E` producers, so the PECS mnemonic tells us that the declaration should be:

```
public static <E> Set<E> union(Set<? extends E> s1,
                               Set<? extends E> s2)
```

Note that the return type is still `Set<E>`. **Do not use wildcard types as return types.** Rather than providing additional flexibility for your users, it would force them to use wildcard types in client code.

Properly used, wildcard types are nearly invisible to users of a class. They cause methods to accept the parameters they should accept and reject those they should reject. **If the user of a class has to think about wildcard types, there is probably something wrong with the class's API.**

Unfortunately, the type inference rules are quite complex. They take up sixteen pages in the language specification [JLS, 15.12.2.7–8], and they don't always do what you want them to. Looking at the revised declaration for `union`, you might think that you could do this:

```
Set<Integer> integers = ... ;
Set<Double> doubles = ... ;
Set<Number> numbers = union(integers, doubles);
```

If you try it you'll get this error message:

```
Union.java:14: incompatible types
found : Set<Number & Comparable<? extends Number &
                                Comparable<?>>>
required: Set<Number>
        Set<Number> numbers = union(integers, doubles);
                              ^
```

Luckily there is a way to deal with this sort of error. If the compiler doesn't infer the type that you wish it had, you can tell it what type to use with an *explicit*

type parameter. This is not something that you have to do very often, which is a good thing, as explicit type parameters aren't very pretty. With the addition of this explicit type parameter, the program compiles cleanly:

```
Set<Number> numbers = Union.<Number>union(integers, doubles);
```

Next let's turn our attention to the `max` method from Item 27. Here is the original declaration:

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

Here is a revised declaration that uses wildcard types:

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

To get the revised declaration from the original one, we apply the PECS transformation twice. The straightforward application is to the parameter `list`. It produces `T` instances, so we change the type from `List<T>` to `List<? extends T>`. The tricky application is to the type parameter `T`. This is the first time we've seen a wildcard applied to a type parameter. `T` was originally specified to extend `Comparable<T>`, but a comparable of `T` consumes `T` instances (and produces integers indicating order relations). Therefore the parameterized type `Comparable<T>` is replaced by the bounded wildcard type `Comparable<? super T>`. Comparables are always consumers, so you should **always use `Comparable<? super T>` in preference to `Comparable<T>`.** The same is true of comparators, so you should **always use `Comparator<? super T>` in preference to `Comparator<T>`.**

The revised `max` declaration is probably the most complex method declaration in the entire book. Does the added complexity really buy you anything? Yes, it does. Here is a simple example of a list that would be excluded by the original declaration but is permitted by the revised one:

```
List<ScheduledFuture<?>> scheduledFutures = ... ;
```

The reason that you can't apply the original method declaration to this list is that `java.util.concurrent.ScheduledFuture` does not implement `Comparable<ScheduledFuture>`. Instead, it is a subinterface of `Delayed`, which extends `Comparable<Delayed>`. In other words, a `ScheduledFuture` instance isn't merely comparable to other `ScheduledFuture` instances; it's comparable to any `Delayed` instance, and that's enough to cause the original declaration to reject it.

There is one slight problem with the revised declaration for `max`: it prevents the method from compiling. Here is the method with the revised declaration:

```
// Won't compile - wildcards can require change in method body!
public static <T extends Comparable<? super T>> T max(
        List<? extends T> list) {
    Iterator<T> i = list.iterator();
    T result = i.next();
    while (i.hasNext()) {
        T t = i.next();
        if (t.compareTo(result) > 0)
            result = t;
    }
    return result;
}
```

Here's what happens when you try to compile it:

```
Max.java:7: incompatible types
found   : Iterator<capture#591 of ? extends T>
required: Iterator<T>
        Iterator<T> i = list.iterator();
                                    ^
```

What does this error message mean, and how do we fix the problem? It means that `list` is not a `List<T>`, so its `iterator` method doesn't return `Iterator<T>`. It returns an iterator of some subtype of `T`, so we replace the iterator declaration with this one, which uses a bounded wildcard type:

```
Iterator<? extends T> i = list.iterator();
```

That is the only change that we have to make to the body of the method. The elements returned by the iterator's `next` method are of some subtype of `T`, so they can be safely stored in a variable of type `T`.

There is one more wildcard-related topic that bears discussing. There is a duality between type parameters and wildcards, and many methods can be declared using one or the other. For example, here are two possible declarations for a static method to swap two indexed items in a list. The first uses an unbounded type parameter (Item 27) and the second an unbounded wildcard:

```
// Two possible declarations for the swap method
public static <E> void swap(List<E> list, int i, int j);
public static void swap(List<?> list, int i, int j);
```

Which of these two declarations is preferable, and why? In a public API, the second is better because it's simpler. You pass in a list—any list—and the method swaps the indexed elements. There is no type parameter to worry about. As a rule, **if a type parameter appears only once in a method declaration, replace it with a wildcard.** If it's an unbounded type parameter, replace it with an unbounded wildcard; if it's a bounded type parameter, replace it with a bounded wildcard.

There's one problem with the second declaration for `swap`, which uses a wildcard in preference to a type parameter: the straightforward implementation won't compile:

```
public static void swap(List<?> list, int i, int j) {
    list.set(i, list.set(j, list.get(i)));
}
```

Trying to compile it produces this less-than-helpful error message:

```
Swap.java:5: set(int,capture#282 of ?) in List<capture#282 of ?>
cannot be applied to (int,Object)
        list.set(i, list.set(j, list.get(i)));
                            ^
```

It doesn't seem right that we can't put an element back into the list that we just took it out of. The problem is that the type of `list` is `List<?>`, and you can't put any value except `null` into a `List<?>`. Fortunately, there is a way to implement this method without resorting to an unsafe cast or a raw type. The idea is to write a private helper method to *capture* the wildcard type. The helper method must be a generic method in order to capture the type. Here's how it looks:

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

// Private helper method for wildcard capture
private static <E> void swapHelper(List<E> list, int i, int j) {
    list.set(i, list.set(j, list.get(i)));
}
```

The `swapHelper` method knows that `list` is a `List<E>`. Therefore, it knows that any value it gets out of this list is of type `E`, and that it's safe to put any value of type `E` into the list. This slightly convoluted implementation of `swap` compiles cleanly. It allows us to export the nice wildcard-based declaration of `swap`, while taking advantage of the more complex generic method internally. Clients of the

`swap` method don't have to confront the more complex `swapHelper` declaration, but they do benefit from it

In summary, using wildcard types in your APIs, while tricky, makes the APIs far more flexible. If you write a library that will be widely used, the proper use of wildcard types should be considered mandatory. Remember the basic rule: producer-`extends`, consumer-`super` (PECS). And remember that all comparables and comparators are consumers.

Item 29: Consider typesafe heterogeneous containers

The most common use of generics is for collections, such as `Set` and `Map`, and single-element containers, such as `ThreadLocal` and `AtomicReference`. In all of these uses, it is the container that is parameterized. This limits you to a fixed number of type parameters per container. Normally that is exactly what you want. A `Set` has a single type parameter, representing its element type; a `Map` has two, representing its key and value types; and so forth.

Sometimes, however, you need more flexibility. For example, a database row can have arbitrarily many columns, and it would be nice to be able to access all of them in a typesafe manner. Luckily, there is an easy way to achieve this effect. The idea is to parameterize the *key* instead of the *container*. Then present the parameterized key to the container to insert or retrieve a value. The generic type system is used to guarantee that the type of the value agrees with its key.

As a simple example of this approach, consider a `Favorites` class that allows its clients to store and retrieve a "favorite" instance of arbitrarily many other classes. The `Class` object will play the part of the parameterized key. The reason this works is that class `Class` was generified in release 1.5. The type of a class literal is no longer simply `Class`, but `Class<T>`. For example, `String.class` is of type `Class<String>`, and `Integer.class` is of type `Class<Integer>`. When a class literal is passed among methods to communicate both compile-time and runtime type information, it is called a *type token* [Bracha04].

The API for the `Favorites` class is simple. It looks just like a simple map, except that the key is parameterized instead of the map. The client presents a `Class` object when setting and getting favorites. Here is the API:

```
// Typesafe heterogeneous container pattern - API
public class Favorites {
    public <T> void putFavorite(Class<T> type, T instance);
    public <T> T getFavorite(Class<T> type);
}
```

Here is a sample program that exercises the `Favorites` class, storing, retrieving, and printing a favorite `String`, `Integer`, and `Class` instance:

```
// Typesafe heterogeneous container pattern - client
public static void main(String[] args) {
    Favorites f = new Favorites();
    f.putFavorite(String.class, "Java");
    f.putFavorite(Integer.class, 0xcafebabe);
    f.putFavorite(Class.class, Favorites.class);
```

```
        String favoriteString = f.getFavorite(String.class);
        int favoriteInteger = f.getFavorite(Integer.class);
        Class<?> favoriteClass = f.getFavorite(Class.class);
        System.out.printf("%s %x %s%n", favoriteString,
            favoriteInteger, favoriteClass.getName());
    }
```

As you might expect, this program prints `Java cafebabe Favorites`.

A `Favorites` instance is *typesafe*: it will never return an `Integer` when you ask it for a `String`. It is also *heterogeneous*: unlike an ordinary map, all the keys are of different types. Therefore, we call `Favorites` a *typesafe heterogeneous container*.

The implementation of `Favorites` is surprisingly tiny. Here it is, in its entirety:

```
// Typesafe heterogeneous container pattern - implementation
public class Favorites {
    private Map<Class<?>, Object> favorites =
        new HashMap<Class<?>, Object>();

    public <T> void putFavorite(Class<T> type, T instance) {
        if (type == null)
            throw new NullPointerException("Type is null");
        favorites.put(type, instance);
    }

    public <T> T getFavorite(Class<T> type) {
        return type.cast(favorites.get(type));
    }
}
```

There are a few subtle things going on here. Each `Favorites` instance is backed by a private `Map<Class<?>, Object>` called `favorites`. You might think that you couldn't put anything into this `Map` because of the unbounded wildcard type, but the truth is quite the opposite. The thing to notice is that the wildcard type is nested: it's not the type of the `Map` that's a wildcard type but the type of its key. This means that every key can have a *different* parameterized type: one can be `Class<String>`, the next `Class<Integer>`, and so on. That's where the heterogeneity comes from.

The next thing to notice is that the value type of the `favorites` `Map` is simply `Object`. In other words, the `Map` does not guarantee the type relationship between keys and values, which is that every value is of the type represented by its key. In fact, Java's type system is not powerful enough to express this. But we know that it's true, and we take advantage of it when it comes time to retrieve a favorite.

The `putFavorite` implementation is trivial: it simply puts into `favorites` a mapping from the given `Class` object to the given favorite instance. As noted, this discards the "type linkage" between the key and the value; it loses the knowledge that the value is an instance of the key. But that's OK, because the `getFavorites` method can and does reestablish this linkage.

The implementation of the `getFavorite` method is trickier than that of `putFavorite`. First it gets from the `favorites` map the value corresponding to the given `Class` object. This is the correct object reference to return, but it has the wrong compile-time type. Its type is simply `Object` (the value type of the `favorites` map) and we need to return a `T`. So, the `getFavorite` implementation *dynamically casts* the object reference to the type represented by the `Class` object, using `Class`'s `cast` method.

The `cast` method is the dynamic analog of Java's cast operator. It simply checks that its argument is an instance of the type represented by the `Class` object. If so, it returns the argument; otherwise it throws a `ClassCastException`. We know that the cast invocation in `getFavorite` will never throw `ClassCastException`, assuming the client code compiled cleanly. That is to say, we know that the values in the `favorites` map always match the types of the keys.

So what does the `cast` method do for us, given that it simply returns its argument? The signature of the `cast` method takes full advantage of the fact that class `Class` has been generified. Its return type is the type parameter of the `Class` object:

```
public class Class<T> {
    T cast(Object obj);
}
```

This is precisely what's needed by the `getFavorite` method. It is what allows us to make `Favorites` typesafe without resorting to an unchecked cast to `T`.

There are two limitations to the `Favorites` class that are worth noting. First, a malicious client could easily corrupt the type safety of a `Favorites` instance, simply by using a `Class` object in its raw form. But the resulting client code would generate an unchecked warning when it was compiled. This is no different from the normal collection implementations such as `HashSet` and `HashMap`. You can easily put a `String` into a `HashSet<Integer>` by using the raw type `HashSet` (Item 23). That said, you can have runtime type safety if you're willing to pay for it. The way to ensure that `Favorites` never violates its type invariant is to have the

putFavorite method check that instance is indeed an instance of the type represented by type. And we already know how to do this. Just use a dynamic cast:

```
// Achieving runtime type safety with a dynamic cast
public <T> void putFavorite(Class<T> type, T instance) {
    favorites.put(type, type.cast(instance));
}
```

There are collection wrappers in java.util.Collections that play the same trick. They are called checkedSet, checkedList, checkedMap, and so forth. Their static factories take a Class object (or two) in addition to a collection (or map). The static factories are generic methods, ensuring that the compile-time types of the Class object and the collection match. The wrappers add reification to the collections they wrap. For example, the wrapper throws a ClassCastException at runtime if someone tries to put Coin into your Collection<Stamp>. These wrappers are useful for tracking down who adds an incorrectly typed element to a collection in an application that mixes generic and legacy code.

The second limitation of the Favorites class is that it cannot be used on a non-reifiable type (Item 25). In other words, you can store your favorite String or String[], but not your favorite List<String>. If you try to store your favorite List<String>, your program won't compile. The reason is that you can't get a Class object for List<String>: List<String>.class is a syntax error, and it's a good thing, too. List<String> and List<Integer> share a single Class object, which is List.class. It would wreak havoc with the internals of a Favorites object if the "type literals" List<String>.class and List<Integer>.class were legal and returned the same object reference.

There is no entirely satisfactory workaround for the second limitation. There is a technique called *super type tokens* that goes a long way toward addressing the limitation, but this technique has limitations of its own [Gafter07].

The type tokens used by Favorites are unbounded: getFavorite and putFavorite accept any Class object. Sometimes you may need to limit the types that can be passed to a method. This can be achieved with a *bounded type token*, which is simply a type token that places a bound on what type can be represented, using a bounded type parameter (Item 27) or a bounded wildcard (Item 28).

The annotations API (Item 35) makes extensive use of bounded type tokens. For example, here is the method to read an annotation at runtime. This method

comes from the AnnotatedElement interface, which is implemented by the reflective types that represent classes, methods, fields, and other program elements:

```
public <T extends Annotation>
    T getAnnotation(Class<T> annotationType);
```

The argument annotationType is a bounded type token representing an annotation type. The method returns the element's annotation of that type, if it has one, or null, if it doesn't. In essence, an annotated element is a typesafe heterogeneous container whose keys are annotation types.

Suppose you have an object of type Class<?> and you want to pass it to a method that requires a bounded type token, such as getAnnotation. You could cast the object to Class<? extends Annotation>, but this cast is unchecked, so it would generate a compile-time warning (Item 24). Luckily, class Class provides an instance method that performs this sort of cast safely (and dynamically). The method is called asSubclass, and it casts the Class object on which it's called to represent a subclass of the class represented by its argument. If the cast succeeds, the method returns its argument; if it fails, it throws a ClassCastException.

Here's how you use the asSubclass method to read an annotation whose type is unknown at compile time. This method compiles without error or warning:

```
// Use of asSubclass to safely cast to a bounded type token
static Annotation getAnnotation(AnnotatedElement element,
                                String annotationTypeName) {
    Class<?> annotationType = null; // Unbounded type token
    try {
        annotationType = Class.forName(annotationTypeName);
    } catch (Exception ex) {
        throw new IllegalArgumentException(ex);
    }
    return element.getAnnotation(
        annotationType.asSubclass(Annotation.class));
}
```

In summary, the normal use of generics, exemplified by the collections APIs, restricts you to a fixed number of type parameters per container. You can get around this restriction by placing the type parameter on the key rather than the container. You can use Class objects as keys for such typesafe heterogeneous containers. A Class object used in this fashion is called a type token. You can also use a custom key type. For example, you could have a DatabaseRow type representing a database row (the container), and a generic type Column<T> as its key.