CHAPTER 5

Generics

SINCE Java 5, generics have been a part of the language. 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, which are not limited to collections, come at a price. This chapter tells you how to maximize the benefits and minimize the complications.

Item 26: Don't use raw types

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, the `List` interface has a single type parameter, `E`, representing its element type. The full name of the interface is `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 type parameters [JLS, 4.8]. For example, 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. They exist primarily for compatibility with pre-generics code.

Before generics were added to Java, this would have been an exemplary collection declaration. As of Java 9, it is still legal, but far from exemplary:

```
// Raw collection type - don't do this!

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

If you use this declaration today and then accidentally put a coin into your stamp collection, the erroneous insertion compiles and runs without error (though the compiler does emit a vague warning):

```
// Erroneous insertion of coin into stamp collection
stamps.add(new Coin( ... )); // Emits "unchecked call" warning
```

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

```
// Raw iterator type - don't do this!
for (Iterator i = stamps.iterator(); i.hasNext(); )
    Stamp stamp = (Stamp) i.next(); // Throws ClassCastException
        stamp.cancel();
```

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 until runtime, long after it has happened, and in code that may be distant from the code containing the error. Once you see the `ClassCastException`, you have to search through the codebase 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, the type declaration contains the information, not 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* it to be true, assuming your entire codebase compiles without emitting (or suppressing; see Item 27) any warnings. When `stamps` is declared with a parameterized type declaration, the erroneous insertion generates a compile-time error message that tells you *exactly* what is wrong:

```
Test.java:9: error: incompatible types: Coin cannot be converted
to Stamp
    c.add(new Coin());
              ^
```

The compiler inserts invisible casts for you when retrieving elements from collections and guarantees that they won't fail (assuming, again, that all of your code did not generate or suppress any compiler warnings). 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 putting a `BigInteger` into a collection that is supposed to contain only `BigDecimal` instances.

As noted earlier, it is legal to use raw types (generic types without their type parameters), but you should never do it. **If you use raw types, you lose all the safety and expressiveness benefits of generics.** Given that you shouldn't use them, why did the language designers permit raw types in the first place? For compatibility. Java was about to enter its second decade when generics were added, and there was an enormous amount of code in existence that did not use generics. It was deemed critical that all of this code remain legal and interoperate with newer code that does use generics. It had to be legal to pass instances of parameterized types to methods that were designed for use with raw types, and vice versa. This requirement, known as *migration compatibility*, drove the decisions to support raw types and to implement generics using *erasure* (Item 28).

While you shouldn't use raw types such as `List`, 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 the generic type system, 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 28). As a consequence, **you lose type safety if you use a raw type such as `List`, but not if you use a parameterized type such as `List<Object>`.**

To make this concrete, consider the following program:

```
// Fails at runtime - unsafeAdd method uses a raw type (List)!
public static void main(String[] args) {
    List<String> strings = new ArrayList<>();
    unsafeAdd(strings, Integer.valueOf(42));
    String s = strings.get(0); // Has 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] unchecked call to add(E) as a
member of the 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)`, which is an `Integer`, 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 but emits the error message:

```
Test.java:5: error: incompatible types: List<String> cannot be
converted to List<Object>
    unsafeAdd(strings, Integer.valueOf(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. The safe alternative is to use *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` declaration looks with unbounded wildcard types:

```
// Uses unbounded wildcard type - typesafe and flexible
static int numElementsInCommon(Set<?> s1, Set<?> s2) { ... }
```

What is the difference between the unbounded wildcard type `Set<?>` and the raw type `Set`? Does the question mark 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 119); **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: error: incompatible types: String cannot be
converted to CAP#1
    c.add("verboten");
          ^
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
```

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, whatever its element type may be. 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 30) or *bounded wildcard types* (Item 31).

There are a few minor exceptions to the rule that you should not use raw types. **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.

A 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<?> s = (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. 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 later in this chapter) are summarized in the following table:

Term	**Example**	**Item**
Parameterized type	`List<String>`	Item 26
Actual type parameter	`String`	Item 26
Generic type	`List<E>`	Items 26, 29
Formal type parameter	`E`	Item 26
Unbounded wildcard type	`List<?>`	Item 26
Raw type	`List`	Item 26
Bounded type parameter	`<E extends Number>`	Item 29
Recursive type bound	`<T extends Comparable<T>>`	Item 30
Bounded wildcard type	`List<? extends Number>`	Item 31
Generic method	`static <E> List<E> asList(E[] a)`	Item 30
Type token	`String.class`	Item 33

Item 27: Eliminate unchecked warnings

When you program with generics, you will see many compiler warnings: unchecked cast warnings, unchecked method invocation warnings, unchecked parameterized vararg type 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 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
        Set<Lark> exaltation = new HashSet();
                               ^
  required: Set<Lark>
  found:    HashSet
```

You can then make the indicated correction, causing the warning to disappear. Note that you don't actually have to specify the type parameter, merely to indicate that it's present with the *diamond operator* (<>), introduced in Java 7. The compiler will then *infer* the correct actual type parameter (in this case, `Lark`):

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

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 will behave as you intended.

If you can't eliminate a warning, but 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 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 on any declaration, 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
       return (T[]) Arrays.copyOf(elements, size, a.getClass());
                                ^
  required: T[]
  found:    Object[]
```

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;
}
```

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

Every time you use a `@SuppressWarnings("unchecked")` annotation, add a comment saying why it is 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 a `@SuppressWarnings("unchecked")` annotation in the narrowest possible scope. Record the rationale for your decision to suppress the warning in a comment.

Item 28: 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 the array type `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 type at runtime. As noted earlier, if you try to put 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 allowed generic types to interoperate freely with legacy code that didn't use generics (Item 26), ensuring a smooth transition to generics in Java 5.

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. Therefore, 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 = List.of(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) must generate 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. Because of erasure, the only parameterized types that are reifiable are unbounded wildcard types such as `List<?>` and `Map<?,?>` (Item 26). It is legal, though rarely 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 collection to return an array of its element type (but see Item 33 for a partial solution). It also means that you get confusing warnings when using varargs methods (Item 53) 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. The `SafeVarargs` annotation can be used to address this issue (Item 32).

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

For example, suppose you want to write a Chooser class with a constructor that takes a collection, and a single method that returns an element of the collection chosen at random. Depending on what collection you pass to the constructor, you could use a chooser as a game die, a magic 8-ball, or a data source for a Monte Carlo simulation. Here's a simplistic implementation without generics:

```
// Chooser - a class badly in need of generics!
public class Chooser {
    private final Object[] choiceArray;

    public Chooser(Collection choices) {
        choiceArray = choices.toArray();
    }

    public Object choose() {
        Random rnd = ThreadLocalRandom.current();
        return choiceArray[rnd.nextInt(choiceArray.length)];
    }
}
```

To use this class, you have to cast the choose method's return value from Object to the desired type every time you use invoke the method, and the cast will fail at runtime if you get the type wrong. Taking the advice of Item 29 to heart, we attempt to modify Chooser to make it generic. Changes are shown in boldface:

```
// A first cut at making Chooser generic - won't compile
public class Chooser<T> {
    private final T[] choiceArray;

    public Chooser(Collection<T> choices) {
        choiceArray = choices.toArray();
    }

    // choose method unchanged
}
```

If you try to compile this class, you'll get this error message:

```
Chooser.java:9: error: incompatible types: Object[] cannot be
converted to T[]
        choiceArray = choices.toArray();
                                     ^
  where T is a type-variable:
    T extends Object declared in class Chooser
```

No big deal, you say, I'll cast the `Object` array to a `T` array:

```
            choiceArray = (T[]) choices.toArray();
```

This gets rid of the error, but instead you get a warning:

```
    Chooser.java:9: warning: [unchecked] unchecked cast
            choiceArray = (T[]) choices.toArray();
                                               ^
      required: T[], found: Object[]
      where T is a type-variable:
    T extends Object declared in class Chooser
```

The compiler is telling you that it can't vouch for the safety of the cast at runtime because the program won't know what type `T` represents—remember, element type information is erased from generics at runtime. Will the program work? Yes, but the compiler can't prove it. You could prove it to yourself, put the proof in a comment and suppress the warning with an annotation, but you're better off eliminating the cause of warning (Item 27).

To eliminate the unchecked cast warning, use a list instead of an array. Here is a version of the `Chooser` class that compiles without error or warning:

```
    // List-based Chooser - typesafe
    public class Chooser<T> {
        private final List<T> choiceList;

        public Chooser(Collection<T> choices) {
            choiceList = new ArrayList<>(choices);
        }

        public T choose() {
            Random rnd = ThreadLocalRandom.current();
            return choiceList.get(rnd.nextInt(choiceList.size()));
        }
    }
```

This version is a tad more verbose, and perhaps a tad slower, but it's worth it for the peace of mind that 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. As a rule, 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 29: Favor generic types

It is generally not too difficult to parameterize your 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 (toy) stack implementation from Item 7:

```
// 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 should have been parameterized to begin with, but since it wasn't, we can *generify* it after the fact. In other words, we can parameterize it without harming clients of the original non-parameterized version. As it stands, the client has 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 declaration. In this case there is one type parameter, representing the element type of the stack, and the conventional name for this type parameter is `E` (Item 68).

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 28, 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 reasonable 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 27). 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 to `E`, but you will get a warning:

```
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 27, 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;
}
```

Both techniques for eliminating the generic array creation have their adherents. The first is more readable: the array is declared to be of type `E[]`, clearly indicating that it contains only `E` instances. It is also more concise: in a typical generic class, you read from the array at many points in the code; the first technique requires only a single cast (where the array is created), while the second requires a separate cast each time an array element is read. Thus, the first technique is preferable and more commonly used in practice. It does, however, cause *heap pollution* (Item 32): the runtime type of the array does not match its compile-time type (unless `E` happens to be `Object`). This makes some programmers sufficiently queasy that they opt for the second technique, though the heap pollution is harmless in this situation.

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<>();
    for (String arg : args)
        stack.push(arg);
    while (!stack.isEmpty())
        System.out.println(stack.pop().toUpperCase());
}
```

The foregoing example may appear to contradict Item 28, 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 `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 61).

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` 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. If you have any existing types that should be generic but aren't, generify them. This will make life easier for new users of these types without breaking existing clients (Item 26).

Item 30: Favor generic methods

Just as classes can be generic, so can methods. Static utility methods that operate on parameterized types are usually generic. All of the “algorithm” methods in `Collections` (such as `binarySearch` and `sort`) are generic.

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

```
// Uses raw types - unacceptable! (Item 26)
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 its declaration to declare a *type parameter* representing the element type for the three sets (the two arguments and the return value) and use this type parameter throughout the method. **The type parameter list, which declares the type parameters, goes between a 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 and generic types (Items 29, 68):

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

At least for simple generic methods, that’s all there is to it. This method compiles without generating any warnings and provides type safety as well as ease of use. Here's a simple program to exercise the method. This 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 = Set.of("Tom", "Dick", "Harry");
    Set<String> stooges = Set.of("Larry", "Moe", "Curly");
    Set<String> aflCio = union(guys, stooges);
    System.out.println(aflCio);
}
```

When you run the program, it prints `[Moe, Tom, Harry, Larry, Curly, Dick]`. (The order of the elements in the output 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 exactly the same. You can make the method more flexible by using *bounded wildcard types* (Item 31).

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 28), 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, called the *generic singleton factory*, is used for function objects (Item 42) such as `Collections.reverseOrder`, and occasionally for collections such as `Collections.emptySet`.

Suppose that you want to write an identity function dispenser. The libraries provide `Function.identity`, so there's no reason to write your own (Item 59), but it is instructive. It would be wasteful to create a new identity function object time one is requested, because it's stateless. If Java's generics were reified, you would need one identity function per type, but since they're erased a generic singleton will suffice. Here's how it looks:

```
// Generic singleton factory pattern
private static UnaryOperator<Object> IDENTITY_FN = (t) -> t;

@SuppressWarnings("unchecked")
public static <T> UnaryOperator<T> identityFunction() {
    return (UnaryOperator<T>) IDENTITY_FN;
}
```

The cast of `IDENTITY_FN` to `(UnaryFunction<T>)` generates an unchecked cast warning, as `UnaryOperator<Object>` is not a `UnaryOperator<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 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 `UnaryOperator<String>` and a `UnaryOperator<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" };
    UnaryOperator<String> sameString = identityFunction();
    for (String s : strings)
        System.out.println(sameString.apply(s));

    Number[] numbers = { 1, 2.0, 3L };
    UnaryOperator<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*. A common use of recursive type bounds is in connection with the `Comparable` interface, which defines a type's natural ordering (Item 14). This interface is shown here:

```
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.

Many methods take a collection of elements implementing `Comparable` to sort it, search within it, calculate its minimum or maximum, and the like. To do these things, it is required that every element in the collection be comparable to every other element in it, 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 <E extends Comparable<E>> E max(Collection<E> c);
```

The type bound <E extends Comparable<E>> may be read as “any type E that can be compared to itself,” which corresponds more or less precisely to the notion of mutual comparability.

Here is a method to go with the previous declaration. It calculates the maximum value in a collection according to its elements’ natural order, and it compiles without errors or warnings:

```
// Returns max value in a collection - uses recursive type bound
public static <E extends Comparable<E>> E max(Collection<E> c) {
    if (c.isEmpty())
        throw new IllegalArgumentException("Empty collection");

    E result = null;
    for (E e : c)
        if (result == null || e.compareTo(result) > 0)
            result = Objects.requireNonNull(e);

    return result;
}
```

Note that this method throws IllegalArgumentException if the list is empty. A better alternative would be to return an Optional<E> (Item 55).

Recursive type bounds can get much more complex, but luckily they rarely do. If you understand this idiom, its wildcard variant (Item 31), and the *simulated self-type* idiom (Item 2), you’ll be able to deal with most of the recursive type bounds you encounter in practice.

In summary, generic methods, like generic types, are safer and easier to use than methods requiring their clients to put explicit casts on input parameters and return values. Like types, you should make sure that your methods can be used without casts, which often means making them generic. And like types, you should generify existing methods whose use requires casts. This makes life easier for new users without breaking existing clients (Item 26).

Item 31: Use bounded wildcards to increase API flexibility

As noted in Item 28, 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>`. Although 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>`. Since a `List<String>` can't do everything a `List<Object>` can, it isn't a subtype (by the Liskov substitution principal, Item 10).

Sometimes you need more flexibility than invariant typing can provide. Consider the `Stack` class from Item 29. 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<>();
Iterable<Integer> integers = ... ;
numberStack.pushAll(integers);
```

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

```
StackTest.java:7: error: incompatible types: Iterable<Integer>
cannot be converted to Iterable<Number>
        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 29 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 a 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 isn't 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` shown earlier, 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 and constructor declarations from previous items in this chapter. The `Chooser` constructor in Item 28 has this declaration:

```
public Chooser(Collection<T> choices)
```

This constructor uses the collection `choices` only to **produce** values of type `T` (and stores them for later use), so its declaration should use a wildcard type that **`extends`** `T`. Here’s the resulting constructor declaration:

```
// Wildcard type for parameter that serves as an T producer
public Chooser(Collection<? extends T> choices)
```

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

Now let's look at the union method from Item 30. 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 as follows:

```
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 bounded wildcard types as return types.** Rather than providing additional flexibility for your users, it would force them to use wildcard types in client code. With the revised declaration, this code will compile cleanly:

```
    Set<Integer> integers = Set.of(1, 3, 5);
    Set<Double>  doubles  = Set.of(2.0, 4.0, 6.0);
    Set<Number>  numbers  = union(integers, doubles);
```

Properly used, wildcard types are nearly invisible to the 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 its API.**

Prior to Java 8, the type inference rules were not clever enough to handle the previous code fragment, which requires the compiler to use the contextually specified return type (or *target type*) to infer the type of E. The target type of the union invocation shown earlier is Set<Number>. If you try to compile the fragment in an earlier version of Java (with an appropriate replacement for the Set.of factory), you'll get a long, convoluted error message like this:

```
Union.java:14: error: incompatible types
        Set<Number> numbers = union(integers, doubles);
                                   ^
  required: Set<Number>
  found:    Set<INT#1>
  where INT#1,INT#2 are intersection types:
    INT#1 extends Number,Comparable<? extends INT#2>
    INT#2 extends Number,Comparable<?>
```

Luckily there is a way to deal with this sort of error. If the compiler doesn't infer the correct type, you can always tell it what type to use with an *explicit type* *argument* [JLS, 15.12]. Even prior to the introduction of target typing in Java 8, this isn't something that you had to do often, which is good because explicit type arguments aren't very pretty. With the addition of an explicit type argument, as shown here, the code fragment compiles cleanly in versions prior to Java 8:

```
// Explicit type parameter - required prior to Java 8
Set<Number> numbers = Union.<Number>union(integers, doubles);
```

Next let's turn our attention to the max method in Item 30. 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, we applied the PECS heuristic 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. Originally, T was 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 generally **use Comparable<? super T> in preference to Comparable<T>.** The same is true of comparators; therefore, you should generally **use Comparator<? super T> in preference to Comparator<T>.**

The revised max declaration is probably the most complex method declaration in this book. Does the added complexity really buy you anything? Again, 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 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 is comparable to any Delayed instance, and that's enough to cause the original declaration to reject it. More generally, the wildcard is required to support types that do not implement Comparable (or Comparator) directly but extend a type that does.

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 30) 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. 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: error: incompatible types: Object cannot be
converted to CAP#1
        list.set(i, list.set(j, list.get(i)));
                                        ^
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
```

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, 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. It is worth noting that the helper method has precisely the signature that we dismissed as too complex for the public method.

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). Also remember that all comparables and comparators are consumers.

Item 32: Combine generics and varargs judiciously

Varargs methods (Item 53) and generics were both added to the platform in Java 5, so you might expect them to interact gracefully; sadly, they do not. The purpose of varargs is to allow clients to pass a variable number of arguments to a method, but it is a *leaky abstraction*: when you invoke a varargs method, an array is created to hold the varargs parameters; that array, which should be an implementation detail, is visible. As a consequence, you get confusing compiler warnings when varargs parameters have generic or parameterized types.

Recall from Item 28 that a non-reifiable type is one whose runtime representation has less information than its compile-time representation, and that nearly all generic and parameterized types are non-reifiable. If a method declares its varargs parameter to be of a non-reifiable type, the compiler generates a warning on the declaration. If the method is invoked on varargs parameters whose inferred type is non-reifiable, the compiler generates a warning on the invocation too. The warnings look something like this:

```
warning: [unchecked] Possible heap pollution from
    parameterized vararg type List<String>
```

Heap pollution occurs when a variable of a parameterized type refers to an object that is not of that type [JLS, 4.12.2]. It can cause the compiler's automatically generated casts to fail, violating the fundamental guarantee of the generic type system.

For example, consider this method, which is a thinly disguised variant of the code fragment on page 127:

```
// Mixing generics and varargs can violate type safety!
static void dangerous(List<String>... stringLists) {
    List<Integer> intList = List.of(42);
    Object[] objects = stringLists;
    objects[0] = intList;             // Heap pollution
    String s = stringLists[0].get(0); // ClassCastException
}
```

This method has no visible casts yet throws a `ClassCastException` when invoked with one or more arguments. Its last line has an invisible cast that is generated by the compiler. This cast fails, demonstrating that type safety has been compromised, and **it is unsafe to store a value in a generic varargs array parameter.**

This example raises an interesting question: Why is it even legal to declare a method with a generic varargs parameter, when it is illegal to create a generic array explicitly? In other words, why does the method shown previously generate only a warning, while the code fragment on page 127 generates an error? The answer is that methods with varargs parameters of generic or parameterized types can be very useful in practice, so the language designers opted to live with this inconsistency. In fact, the Java libraries export several such methods, including `Arrays.asList(T... a)`, `Collections.addAll(Collection<? super T> c, T... elements)`, and `EnumSet.of(E first, E... rest)`. Unlike the `dangerous` method shown earlier, these library methods are typesafe.

Prior to Java 7, there was nothing the author of a method with a generic varargs parameter could do about the warnings at the call sites. This made these APIs unpleasant to use. Users had to put up with the warnings or, preferably, to eliminate them with `@SuppressWarnings("unchecked")` annotations at every call site (Item 27). This was tedious, harmed readability, and hid warnings that flagged real issues.

In Java 7, the `SafeVarargs` annotation was added to the platform, to allow the author of a method with a generic varargs parameter to suppress client warnings automatically. In essence, **the `SafeVarargs` annotation constitutes a promise by the author of a method that it is typesafe.** In exchange for this promise, the compiler agrees not to warn the users of the method that calls may be unsafe.

It is critical that you do not annotate a method with `@SafeVarargs` unless it actually *is* safe. So what does it take to ensure this? Recall that a generic array is created when the method is invoked, to hold the varargs parameters. If the method doesn't store anything into the array (which would overwrite the parameters) and doesn't allow a reference to the array to escape (which would enable untrusted code to access the array), then it's safe. In other words, if the varargs parameter array is used only to transmit a variable number of arguments from the caller to the method—which is, after all, the purpose of varargs—then the method is safe.

It is worth noting that you can violate type safety without ever storing anything in the varargs parameter array. Consider the following generic varargs method, which returns an array containing its parameters. At first glance, it may look like a handy little utility:

```
// UNSAFE - Exposes a reference to its generic parameter array!
static <T> T[] toArray(T... args) {
    return args;
}
```

This method simply returns its varargs parameter array. The method may not look dangerous, but it is! The type of this array is determined by the compile-time types of the arguments passed in to the method, and the compiler may not have enough information to make an accurate determination. Because this method returns its varargs parameter array, it can propagate heap pollution up the call stack.

To make this concrete, consider the following generic method, which takes three arguments of type T and returns an array containing two of the arguments, chosen at random:

```
static <T> T[] pickTwo(T a, T b, T c) {
    switch(ThreadLocalRandom.current().nextInt(3)) {
      case 0: return toArray(a, b);
      case 1: return toArray(a, c);
      case 2: return toArray(b, c);
    }
    throw new AssertionError(); // Can't get here
}
```

This method is not, in and of itself, dangerous and would not generate a warning except that it invokes the toArray method, which has a generic varargs parameter.

When compiling this method, the compiler generates code to create a varargs parameter array in which to pass two T instances to toArray. This code allocates an array of type Object[], which is the most specific type that is guaranteed to hold these instances, no matter what types of objects are passed to pickTwo at the call site. The toArray method simply returns this array to pickTwo, which in turn returns it to its caller, so pickTwo will always return an array of type Object[].

Now consider this main method, which exercises pickTwo:

```
public static void main(String[] args) {
    String[] attributes = pickTwo("Good", "Fast", "Cheap");
}
```

There is nothing at all wrong with this method, so it compiles without generating any warnings. But when you run it, it throws a ClassCastException, though it contains no visible casts. What you don't see is that the compiler has generated a hidden cast to String[] on the value returned by pickTwo so that it can be stored in attributes. The cast fails, because Object[] is not a subtype of String[]. This failure is quite disconcerting because it is two levels removed from the method that actually causes the heap pollution (toArray), and the varargs parameter array is not modified after the actual parameters are stored in it.

This example is meant to drive home the point that **it is unsafe to give another method access to a generic varargs parameter array,** with two exceptions: it is safe to pass the array to another varargs method that is correctly annotated with @SafeVarargs, and it is safe to pass the array to a non-varargs method that merely computes some function of the contents of the array.

Here is a typical example of a safe use of a generic varargs parameter. This method takes an arbitrary number of lists as arguments and returns a single list containing the elements of all of the input lists in sequence. Because the method is annotated with `@SafeVarargs`, it doesn't generate any warnings, on the declaration or at its call sites:

```
// Safe method with a generic varargs parameter
@SafeVarargs
static <T> List<T> flatten(List<? extends T>... lists) {
    List<T> result = new ArrayList<>();
    for (List<? extends T> list : lists)
        result.addAll(list);
    return result;
}
```

The rule for deciding when to use the `SafeVarargs` annotation is simple: **Use `@SafeVarargs` on every method with a varargs parameter of a generic or parameterized type,** so its users won't be burdened by needless and confusing compiler warnings. This implies that you should *never* write unsafe varargs methods like `dangerous` or `toArray`. Every time the compiler warns you of possible heap pollution from a generic varargs parameter in a method you control, check that the method is safe. As a reminder, a generic varargs methods is safe if:

1. it doesn't store anything in the varargs parameter array, and
2. it doesn't make the array (or a clone) visible to untrusted code.

If either of these prohibitions is violated, fix it.

Note that the `SafeVarargs` annotation is legal only on methods that can't be overridden, because it is impossible to guarantee that every possible overriding method will be safe. In Java 8, the annotation was legal only on static methods and final instance methods; in Java 9, it became legal on private instance methods as well.

An alternative to using the `SafeVarargs` annotation is to take the advice of Item 28 and replace the varargs parameter (which is an array in disguise) with a `List` parameter. Here's how this approach looks when applied to our `flatten` method. Note that only the parameter declaration has changed:

```
// List as a typesafe alternative to a generic varargs parameter
static <T> List<T> flatten(List<List<? extends T>> lists) {
    List<T> result = new ArrayList<>();
    for (List<? extends T> list : lists)
        result.addAll(list);
    return result;
}
```

This method can then be used in conjunction with the static factory method `List.of` to allow for a variable number of arguments. Note that this approach relies on the fact that the `List.of` declaration is annotated with `@SafeVarargs`:

```
audience = flatten(List.of(friends, romans, countrymen));
```

The advantage of this approach is that the compiler can *prove* that the method is typesafe. You don't have to vouch for its safety with a `SafeVarargs` annotation, and you don't have worry that you might have erred in determining that it was safe. The main disadvantage is that the client code is a bit more verbose and may be a bit slower.

This trick can also be used in situations where it is impossible to write a safe varargs method, as is the case with the `toArray` method on page 147. Its `List` analogue *is* the `List.of` method, so we don't even have to write it; the Java libraries authors have done the work for us. The `pickTwo` method then becomes this:

```
static <T> List<T> pickTwo(T a, T b, T c) {
    switch(rnd.nextInt(3)) {
      case 0: return List.of(a, b);
      case 1: return List.of(a, c);
      case 2: return List.of(b, c);
    }
    throw new AssertionError();
}
```

and the main method becomes this:

```
public static void main(String[] args) {
    List<String> attributes = pickTwo("Good", "Fast", "Cheap");
}
```

The resulting code is typesafe because it uses only generics, and not arrays.

In summary, varargs and generics do not interact well because the varargs facility is a leaky abstraction built atop arrays, and arrays have different type rules from generics. Though generic varargs parameters are not typesafe, they are legal. If you choose to write a method with a generic (or parameterized) varargs parameter, first ensure that the method is typesafe, and then annotate it with `@SafeVarargs` so it is not unpleasant to use.

Item 33: Consider typesafe heterogeneous containers

Common uses of generics include collections, such as `Set<E>` and `Map<K,V>`, and single-element containers, such as `ThreadLocal<T>` and `AtomicReference<T>`. 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 types. The `Class` object for the type will play the part of the parameterized key. The reason this works is that class `Class` is generic. The type of a class literal is not 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 would expect, this program prints `Java cafebabe Favorites`. Note, incidentally, that Java's `printf` method differs from C's in that you should use `%n` where you'd use `\n` in C. The `%n` generates the applicable platform-specific line separator, which is `\n` on many but not all platforms.

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<>();

    public <T> void putFavorite(Class<T> type, T instance) {
        favorites.put(Objects.requireNonNull(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 the time comes 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 getFavorite 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: it is 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 analogue 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 won't 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 their 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 is generic. 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, 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 a 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 26). 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 actually 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 a Coin into your Collection<Stamp>. These wrappers are useful for tracking down client code that adds an incorrectly typed element to a collection, in an application that mixes generic and raw types.

The second limitation of the Favorites class is that it cannot be used on a non-reifiable type (Item 28). 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>. The class literal 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 this limitation.

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 30) or a bounded wildcard (Item 31).

The annotations API (Item 39) 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 27). 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 is 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.

This page intentionally left blank