**Don’t use raw types**

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.

**// 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):

[Click here to view code image](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ch5_images.xhtml#pch5ex2a)

**// 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:

[Click here to view code image](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ch5_images.xhtml#pch5ex3a)

**// Raw iterator type - don't do this!**

for (**Iterator** i = stamps.iterator(); i.hasNext(); ) {

    Stamp stamp = **(Stamp)** i.next(); **// Throws ClassCastException**

        stamp.cancel();

}

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.

With generics, the type declaration contains the information, not the comment:

[Click here to view code image](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ch5_images.xhtml#pch5ex4a)

**// 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](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ch5.xhtml#lev27)) 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:

[Click here to view code image](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ch5_images.xhtml#pch5ex5a)

Test.java:9: error: incompatible types: Coin cannot be converted

to Stamp

    stamps.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).

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:

[Click here to view code image](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ch5_images.xhtml#pch5ex9a)

**// 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:

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:

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.

|  |  |  |
| --- | --- | --- |
| **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 |

### ****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:

[Click here to view code image](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ch5_images.xhtml#pch5ex13a)

Set<Lark> exaltation = new HashSet();

The compiler will gently remind you what you did wrong:

[Click here to view code image](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ch5_images.xhtml#pch5ex14a)

Venery.java:4: warning: [unchecked] unchecked conversion

        Set<Lark> exaltation = new HashSet();

                               ^

  required: Set<Lark>

  found:    HashSet

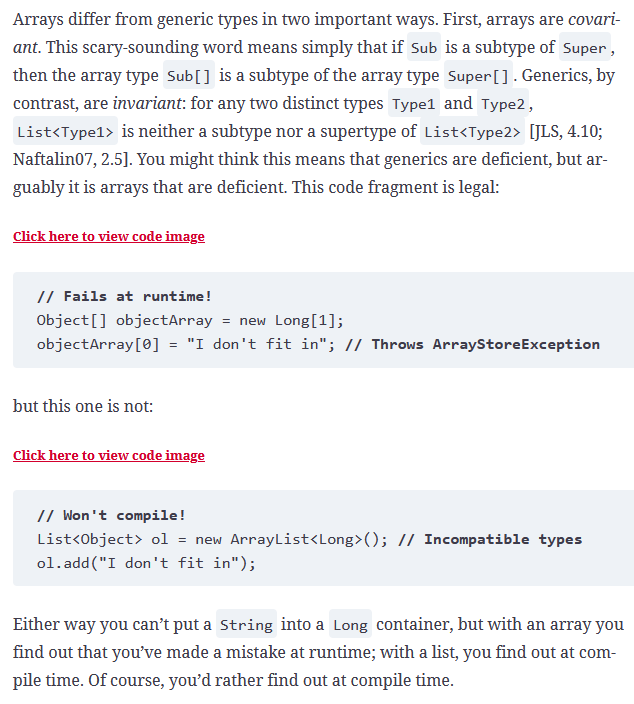
You can then make the indicated correction, causing the warning to disappear. Note that you don’t actually have to specify the type parameter, merely to indicate that it’s present with the *diamond operator* (<>), introduced in Java 7. The compiler will then *infer* the correct actual type parameter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

[Click here to view code image](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ch5_images.xhtml#pch5ex15a)

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.

### ****Prefer lists to arrays****



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](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ch5.xhtml#lev26)), ensuring a smooth transition to generics in Java 5.

### ****Favor generic types****

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](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ch5.xhtml#lev26)).

A screenshot of a computer program

Description automatically generated with low confidence

### ****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:

A screenshot of a computer program

Description automatically generated with low confidence

A screenshot of a computer code

Description automatically generated with low confidence

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](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ch3.xhtml#lev14)). This interface is shown here:

A screenshot of a computer program

Description automatically generated with low confidence

### 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](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ch5.xhtml#lev26)).

### Use Bounded wildcards to increase API flexibility.

As noted in [Item 28](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ch5.xhtml#lev28), 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](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ch3.xhtml#lev10)).

Sometimes you need more flexibility than invariant typing can provide. Consider the Stack class from [Item 29](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ch5.xhtml#lev29). To refresh your memory, here is its public API:

[Click here to view code image](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ch5_images.xhtml#pch5ex47a)

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:

[Click here to view code image](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ch5_images.xhtml#pch5ex48a)

**// 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:

[Click here to view code image](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ch5_images.xhtml#pch5ex49a)

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:

[Click here to view code image](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ch5_images.xhtml#pch5ex50a)

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 called 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](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ch5.xhtml#lev29) 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:

[Click here to view code image](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ch5_images.xhtml#pch5ex51a)

**// Wildcard type for a parameter that serves as an E producer**

public void pushAll(**Iterable<? extends E>** src) {

    for (E e : src)

        push(e);

}

**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 wild-card types. Naftalin and Wadler call it the Get and Put Principle [Naftalin07, 2.4].

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.

### [Combine generics and varargs judiciously](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ch5.xhtml#:-:text=Combine%20generics%20and%20varargs%20judiciously)

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.

### ****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.

**// 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:

[Click here to view code image](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ch5_images.xhtml#pch5ex80a)

**// 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.

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.