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