# Why Use Generics?

In a nutshell, generics enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods. Much like the more familiar formal parameters used in method declarations, type parameters provide a way for you to re-use the same code with different inputs. The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types.

Code that uses generics has many benefits over non-generic code:

* Stronger type checks at compile time.  
  A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.
* Elimination of casts.  
  The following code snippet without generics requires casting:

List list = new ArrayList();

list.add("hello");

String s = **(String)** list.get(0);

* When re-written to use generics, the code does not require casting:

List<String> list = new ArrayList<String>();

list.add("hello");

String s = list.get(0); // no cast

* Enabling programmers to implement generic algorithms.  
  By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

# Generic Types

A generic type is a generic class or interface that is parameterized over types. The following Box class will be modified to demonstrate the concept.

## A Simple Box Class

Begin by examining a non-generic Box class that operates on objects of any type. It needs only to provide two methods: set, which adds an object to the box, and get, which retrieves it:

public class Box {

private Object object;

public void set(Object object) { this.object = object; }

public Object get() { return object; }

}

Since its methods accept or return an Object, you are free to pass in whatever you want, provided that it is not one of the primitive types. There is no way to verify, at compile time, how the class is used. One part of the code may place an Integer in the box and expect to get Integers out of it, while another part of the code may mistakenly pass in a String, resulting in a runtime error.

## A Generic Version of the Box Class

A generic class is defined with the following format:

class name<T1, T2, ..., Tn> { /\* ... \*/ }

The type parameter section, delimited by angle brackets (<>), follows the class name. It specifies the type parameters (also called type variables) T1, T2, ..., and Tn.

To update the Box class to use generics, you create a generic type declaration by changing the code "public class Box" to "public class Box<T>". This introduces the type variable, T, that can be used anywhere inside the class.

With this change, the Box class becomes:

/\*\*

\* Generic version of the Box class.

\* @param <T> the type of the value being boxed

\*/

public class Box<T> {

// T stands for "Type"

private T t;

public void set(T t) { this.t = t; }

public T get() { return t; }

}

As you can see, all occurrences of Object are replaced by T. A type variable can be any **non-primitive** type you specify: any class type, any interface type, any array type, or even another type variable.

This same technique can be applied to create generic interfaces.

## Type Parameter Naming Conventions

By convention, type parameter names are single, uppercase letters. This stands in sharp contrast to the variable [naming](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/variables.html" \l "naming" \t "https://docs.oracle.com/javase/tutorial/java/generics/_top) conventions that you already know about, and with good reason: Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name.

The most commonly used type parameter names are:

* E - Element (used extensively by the Java Collections Framework)
* K - Key
* N - Number
* T - Type
* V - Value
* S,U,V etc. - 2nd, 3rd, 4th types

You'll see these names used throughout the Java SE API and the rest of this lesson.

## Invoking and Instantiating a Generic Type

To reference the generic Box class from within your code, you must perform a generic type invocation, which replaces T with some concrete value, such as Integer:

Box<Integer> integerBox;

You can think of a generic type invocation as being similar to an ordinary method invocation, but instead of passing an argument to a method, you are passing a type argument — Integer in this case — to the Box class itself.

**Type Parameter and Type Argument Terminology:** Many developers use the terms "type parameter" and "type argument" interchangeably, but these terms are not the same. When coding, one provides type arguments in order to create a parameterized type. Therefore, the T in Foo<T> is a type parameter and the String in Foo<String> f is a type argument. This lesson observes this definition when using these terms.

Like any other variable declaration, this code does not actually create a new Box object. It simply declares that integerBox will hold a reference to a "Box of Integer", which is how Box<Integer> is read.

An invocation of a generic type is generally known as a parameterized type.

To instantiate this class, use the new keyword, as usual, but place <Integer> between the class name and the parenthesis:

Box<Integer> integerBox = new Box<Integer>();

## The Diamond

In Java SE 7 and later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments (<>) as long as the compiler can determine, or infer, the type arguments from the context. This pair of angle brackets, <>, is informally called the diamond. For example, you can create an instance of Box<Integer> with the following statement:

Box<Integer> integerBox = new Box<>();

For more information on diamond notation and type inference, see [Type Inference](https://docs.oracle.com/javase/tutorial/java/generics/genTypeInference.html" \t "https://docs.oracle.com/javase/tutorial/java/generics/_top).

## Multiple Type Parameters

As mentioned previously, a generic class can have multiple type parameters. For example, the generic OrderedPair class, which implements the generic Pair interface:

public interface Pair<K, V> {

public K getKey();

public V getValue();

}

public class OrderedPair<K, V> implements Pair<K, V> {

private K key;

private V value;

public OrderedPair(K key, V value) {

this.key = key;

this.value = value;

}

public K getKey() { return key; }

public V getValue() { return value; }

}

The following statements create two instantiations of the OrderedPair class:

Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);

Pair<String, String> p2 = new OrderedPair<String, String>("hello", "world");

The code, new OrderedPair<String, Integer>, instantiates K as a String and V as an Integer. Therefore, the parameter types of OrderedPair's constructor are String and Integer, respectively. Due to [autoboxing](https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html" \t "https://docs.oracle.com/javase/tutorial/java/generics/_top), it is valid to pass a String and an int to the class.

As mentioned in [The Diamond](https://docs.oracle.com/javase/tutorial/java/generics/types.html" \l "diamond), because a Java compiler can infer the K and V types from the declaration OrderedPair<String, Integer>, these statements can be shortened using diamond notation:

OrderedPair<String, Integer> p1 = new OrderedPair**<>**("Even", 8);

OrderedPair<String, String> p2 = new OrderedPair**<>**("hello", "world");

To create a generic interface, follow the same conventions as for creating a generic class.

## Parameterized Types

You can also substitute a type parameter (that is, K or V) with a parameterized type (that is, List<String>). For example, using the OrderedPair<K, V> example:

OrderedPair<String, **Box<Integer>**> p = new OrderedPair<>("primes", new Box<Integer>(...));

# Raw Types

A raw type is the name of a generic class or interface without any type arguments. For example, given the generic Box class:

public class Box<T> {

public void set(T t) { /\* ... \*/ }

// ...

}

To create a parameterized type of Box<T>, you supply an actual type argument for the formal type parameter T:

Box<Integer> intBox = new Box<>();

If the actual type argument is omitted, you create a raw type of Box<T>:

Box rawBox = new Box();

Therefore, Box is the raw type of the generic type Box<T>. However, a non-generic class or interface type is not a raw type.

Raw types show up in legacy code because lots of API classes (such as the Collections classes) were not generic prior to JDK 5.0. When using raw types, you essentially get pre-generics behavior — a Box gives you Objects. For backward compatibility, assigning a parameterized type to its raw type is allowed:

Box<String> stringBox = new Box<>();

Box rawBox = stringBox; // OK

But if you assign a raw type to a parameterized type, you get a warning:

Box rawBox = new Box(); // rawBox is a raw type of Box<T>

Box<Integer> intBox = rawBox; // warning: unchecked conversion

You also get a warning if you use a raw type to invoke generic methods defined in the corresponding generic type:

Box<String> stringBox = new Box<>();

Box rawBox = stringBox;

rawBox.set(8); // warning: unchecked invocation to set(T)

The warning shows that raw types bypass generic type checks, deferring the catch of unsafe code to runtime. Therefore, you should avoid using raw types.

The [Type Erasure](https://docs.oracle.com/javase/tutorial/java/generics/erasure.html" \t "https://docs.oracle.com/javase/tutorial/java/generics/_top) section has more information on how the Java compiler uses raw types.

## Unchecked Error Messages

As mentioned previously, when mixing legacy code with generic code, you may encounter warning messages similar to the following:

Note: Example.java uses unchecked or unsafe operations.

Note: Recompile with -Xlint:unchecked for details.

This can happen when using an older API that operates on raw types, as shown in the following example:

public class WarningDemo {

public static void main(String[] args){

Box<Integer> bi;

bi = createBox();

}

static Box createBox(){

return new Box();

}

}

The term "unchecked" means that the compiler does not have enough type information to perform all type checks necessary to ensure type safety. The "unchecked" warning is disabled, by default, though the compiler gives a hint. To see all "unchecked" warnings, recompile with -Xlint:unchecked.

Recompiling the previous example with -Xlint:unchecked reveals the following additional information:

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

found : Box

required: Box<java.lang.Integer>

bi = createBox();

^

1 warning

To completely disable unchecked warnings, use the -Xlint:-unchecked flag. The @SuppressWarnings("unchecked") annotation suppresses unchecked warnings. If you are unfamiliar with the @SuppressWarnings syntax, see [Annotations](https://docs.oracle.com/javase/tutorial/java/annotations/index.html" \t "https://docs.oracle.com/javase/tutorial/java/generics/_top).

# Generic Methods

Generic methods are methods that introduce their own type parameters. This is similar to declaring a generic type, but the type parameter's scope is limited to the method where it is declared. Static and non-static generic methods are allowed, as well as generic class constructors.

The syntax for a generic method includes a list of type parameters, inside angle brackets, which appears before the method's return type. For static generic methods, the type parameter section must appear before the method's return type.

The Util class includes a generic method, compare, which compares two Pair objects:

public class Util {

**public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2)** {

return p1.getKey().equals(p2.getKey()) &&

p1.getValue().equals(p2.getValue());

}

}

public class Pair<K, V> {

private K key;

private V value;

public Pair(K key, V value) {

this.key = key;

this.value = value;

}

public void setKey(K key) { this.key = key; }

public void setValue(V value) { this.value = value; }

public K getKey() { return key; }

public V getValue() { return value; }

}

The complete syntax for invoking this method would be:

Pair<Integer, String> p1 = new Pair<>(1, "apple");

Pair<Integer, String> p2 = new Pair<>(2, "pear");

boolean same = Util.**<Integer, String>**compare(p1, p2);

The type has been explicitly provided, as shown in bold. Generally, this can be left out and the compiler will infer the type that is needed:

Pair<Integer, String> p1 = new Pair<>(1, "apple");

Pair<Integer, String> p2 = new Pair<>(2, "pear");

boolean same = Util.compare(p1, p2);

This feature, known as type inference, allows you to invoke a generic method as an ordinary method, without specifying a type between angle brackets. This topic is further discussed in the following section, [Type Inference](https://docs.oracle.com/javase/tutorial/java/generics/genTypeInference.html" \t "https://docs.oracle.com/javase/tutorial/java/generics/_top).





