Why Use Generics?

In a nutshell, generics enable types (classes and interfaces) to be parameters when defining classes, interfaces, and methods. Much like formal parameters in method declarations, type parameters provide a way 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:

1. **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.
2. **Elimination of Casts**
   The following code snippet without generics requires casting:

   ```
   Integer salary = 100000;
   salary = "another string"; //compiler error

   List list = new ArrayList(); //Raw List type
   list.add("hello");
   String s = (String) list.get(0);
   ```

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

```
List<E>  // E is type parameter

public class SalaryCalculator{
  //method definition
        Double calculateAnnualSalary(Employee employee); //method parameter
}
public static void main (String args[]){
  SalaryCalculator salaryCalculator = new SalaryCalculator();
        //method invocation
  salaryCalculator.calculateAnnualSalary(employee);//method argument
}


List<String> list = new ArrayList<String>();//generic List type
List<String> list = new ArrayList<>(); //Java 7 type inference happening
StringList stringList = new StringList();//type safety if generics were not introduce
list.add("hello");
//list.add(123423);//compile-time error
String s = list.get(0);   // no cast

Conceptual mapping...
                Method parameter  <=> Type Parameter
    method invocation <=> Generic Type invocation
    method argument   <=> Type Argument
```

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

Generic Types

A **generic type** is a generic class or interface that is parameterized over types. This allows classes, interfaces, and methods to operate on different data types without being rewritten for each type. The following Box class will be modified to demonstrate the concept.

A Simple Box Class

```
 term - Non-Generic Box class      vs     Generic Box Class
```

Let's begin by examining a **non-generic** Box class that operates on objects of any type. It provides 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 can pass in any object, provided it's not a primitive type. However, there's 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 `Integer` out, while another part 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 and specifies the type parameters (also called type variables) `T1, T2, ..., Tn`.
To update the Box class to use generics, create a generic type declaration by changing the code `public class Box` to `public class Box<T>`. This introduces the type variable `T`, which can be used anywhere inside the class.

TIP: For writing your first few generic classes, First you write your Type Specific Class

```
public class BoxSpecific {
 private Integer object;

 public void set(Integer object) { this.object = object; }
 public Integer get() { return object; }
}
```

Then introduce Type Parameter after the class name with angular braces and then replace the specific type with Type Parameter everywhere like the following code snippet

```java
 public class BoxSpecificToGeneric<T> {
  private T object;

  public void set(T object) { this.object = object; }
  public T get() { return object; }
}
```

With this change, the Box class becomes:

```java
public class BoxSpecificToGeneric<T> {
 private T object;

 public void set(T object) { this.object = object; }
 public T get() { return object; }
}
```

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, interface type, 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 contrast to variable naming conventions, and with good reason: without this convention, it would be difficult to distinguish between a type variable and an ordinary class or interface name.

Common type parameter names include:

- **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

These names are used throughout the Java SE API and in this lesson.

Invoking and Instantiating a Generic Type

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

```
Box<Integer> integerBox;
```

You can think of a generic type invocation as similar to an ordinary method invocation, but instead of passing an method argument, you are passing a type argument (e.g., `Integer`) to the Box class itself.

Type Parameter and Type Argument Terminology

Many developers use "type parameter" and "type argument" interchangeably, but these terms are not the same. When coding, you provide type arguments 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 distinction.

To instantiate this class, use the `new` keyword, placing `<Integer>` between the class name and parentheses:

```
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 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<>();
```

Multiple Type Parameters

A generic class can have multiple type parameters. For example, the generic `OrderedPair` class, which implements the generic `Pair` interface:

```java
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:

```java
Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);//pre-Java7
Pair<String, Integer> p1 = new OrderedPair<>("Even", 8);//Type inference with Dimaond ope
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, it is valid to pass a `String` and an `int` to the class.

As mentioned in **The 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:

```java
OrderedPair<String, Integer> p1 = new OrderedPair<>("Even", 8);
OrderedPair<String, String>  p2 = new OrderedPair<>("hello", "world");
```

Parameterized Types

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

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

This demonstrates how generics in Java provide a flexible and powerful way to write reusable, type-safe code.

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 considered a raw type.***

NOTE: about (non-negotiable) terminologies.

Generic Type (class or interface)

```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}
```

When you instantiate with passing a type argument (lets say String, Integer) for the type parameter K,V
you are doing generic type invocation.
When you instantiate withOUT passing a type argument for the type parameter K,V
you are generating a RAW Type instance for a Generic Type Class.

If you Defined Pair without Type Parameters in the first place.

```
public interface Pair {
    public Integer getKey();
    public Integer getValue();
}
```

Its called a non-Generic Pair interface

```
public class Pair {
    public Integer getKey(){
        return 1;
    }
    public Integer getValue(){
      return 2;
    }
}
```

Its called a non-Generic Pair class.

Raw types often appear in legacy code because many API classes (such as the Collections classes) were not generic prior to JDK 5.0. When using raw types, you essentially revert to pre-generics behavior — a `Box` gives you `Object` references. For backward compatibility, assigning a parameterized type to its raw type is allowed:

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;  // OK
```

However, if you assign a raw type to a parameterized type, you receive a warning:

```
Box rawBox = new Box();           // rawBox is a raw type of Box<T>
Box<Integer> intBox = rawBox;     // warning: unchecked conversion
```

You also receive 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 indicates that raw types bypass generic type checks, deferring the catch of unsafe code to runtime. Therefore, it is advisable to avoid using raw types.

For more information on how the Java compiler handles raw types, see the **Type Erasure** section.

Unchecked Error Messages

When mixing legacy code with generic code, you may encounter warning messages like 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 necessary type checks to ensure type safety. The "unchecked" warning is disabled by default, though the compiler provides 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 can also be used to suppress unchecked warnings. If you are unfamiliar with the `@SuppressWarnings` syntax, see the section on **Annotations**.

Generic Methods

Generic methods are methods that introduce their own type parameters. While similar to declaring a generic type, the type parameter's scope is confined to the method where it's declared. Both 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 also appear before the method's return type.

Consider the `Util` class that includes a generic method, `compare`, which compares two `Pair` objects:

```java
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; }
}
```

To invoke this method, the complete syntax would be:

```java
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.<Integer, String>compare(p1, p2);
```

In this example, the type has been explicitly provided, as shown in bold. However, you can often omit the type, and the compiler will infer the necessary type:

```java
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 on *Type Inference*.

Bounded Type Parameters

Bounded type parameters allow you to restrict the types that can be used as type arguments in a parameterized type. For instance, if a method operates on numbers, you might want it to accept only instances of `Number` or its subclasses. This is where bounded type parameters come into play.

To declare a bounded type parameter, you list the type parameter's name, followed by the `extends` keyword, and then the upper bound, which, in this example, is `Number`. Here, `extends` is used in a general sense to mean either "extends" (for classes) or "implements" (for interfaces).

Consider the following example:

```java
public class Box<T> {

    private T t;

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

    public T get() {
        return t;
    }

    public <U extends Number> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.set(new Integer(10));
        integerBox.inspect("some text"); // error: this is still String!
    }
}
```

By modifying the generic method to include a bounded type parameter, the compilation will fail if the wrong type is used, as in the invocation of `inspect` with a `String`:

```
Box.java:21: <U>inspect(U) in Box<java.lang.Integer> cannot
  be applied to (java.lang.String)
                        integerBox.inspect("10");
                                  ^
1 error
```

In addition to limiting the types that can be used to instantiate a generic type, bounded type parameters allow you to invoke methods defined in the bounds. For example:

```
public class NaturalNumber<T extends Integer> {

    private T n;

    public NaturalNumber(T n)  { this.n = n; }

    public boolean isEven() {
        return n.intValue() % 2 == 0;
    }

    // ...
}
```

The `isEven` method can invoke the `intValue` method defined in the `Integer` class through `n`.

Multiple Bounds

A type parameter can also have multiple bounds:

```
<T extends B1 & B2 & B3>
```

A type variable with multiple bounds is a subtype of all the types listed in the bound. If one of the bounds is a class, it must be specified first. For example:

```
Class A { /* ... */ }
interface B { /* ... */ }
interface C { /* ... */ }

class D <T extends A & B & C> { /* ... */ }
```

If the class bound is not specified first, you'll get a compile-time error:

```
class D <T extends B & A & C> { /* ... */ }  // compile-time error
```

Generic Methods and Bounded Type Parameters

Bounded type parameters are essential for implementing generic algorithms. Consider the following method, which counts the number of elements in an array `T[]` that are greater than a specified element `elem`:

```
public static <T> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e > elem)  // compiler error
            ++count;
    return count;
}
```

While the method's implementation is straightforward, it doesn't compile because the greater-than operator (>) only applies to primitive types like `short` , `int` , `double` , `long` , `float` , `byte` , and `char` . You cannot use the `>` operator to compare objects. To resolve this issue, you can use a type parameter bounded by the `Comparable<T>` interface:

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

The corrected method would be:

```
public static <T extends Comparable<T>> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e.compareTo(elem) > 0)
            ++count;
    return count;
}
```

By bounding the type parameter `T` with `Comparable<T>` , the method can now use the `compareTo` method to compare objects, ensuring that the method compiles and functions correctly.

Generics, Inheritance, and Subtypes

As you may already know, it is possible to assign an object of one type to an object of another type, provided the types are compatible. For example, you can assign an `Integer` to an `Object` since `Object` is a supertype of `Integer`:

```
Object someObject = new Object();
Integer someInteger = new Integer(10);
someObject = someInteger;   // OK
```

In object-oriented terminology, this is known as an "is-a" relationship. Since an `Integer` is a kind of `Object`, the assignment is allowed. Similarly, `Integer` is also a kind of `Number`, so the following code is valid as well:

```
public void someMethod(Number n) { /* ... */ }

someMethod(new Integer(10));   // OK
someMethod(new Double(10.1));  // OK
```

The same concept applies to generics. You can create a generic type invocation with `Number` as its type argument, and any subsequent invocation of `add` will be allowed if the argument is compatible with `Number`:

```
Box<Number> box = new Box<Number>();
box.add(new Integer(10));   // OK
box.add(new Double(10.1));  // OK
```

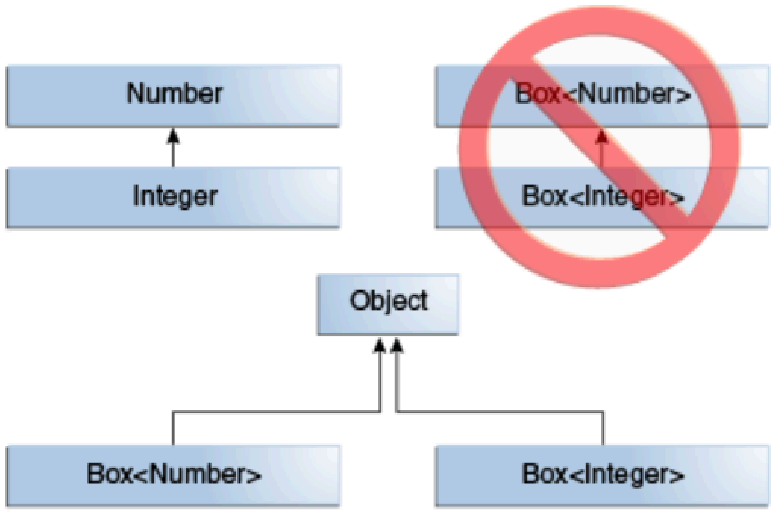
However, consider the following method:

```
public void boxTest(Box<Number> n) { /* ... */ }
```

What type of argument does this method accept? By looking at its signature, you can see that it accepts a single argument of type `Box<Number>`. But does that mean you can pass in a `Box<Integer>` or `Box<Double>`, as you might expect? The answer is "no." This is because `Box<Integer>` and `Box<Double>` are not subtypes of `Box<Number>`.

This is a common misunderstanding when programming with generics, but it's a crucial concept to understand. Although `Integer` and `Double` are subtypes of `Number`, `Box<Integer>` and

`Box<Double>` are not subtypes of `Box<Number>`. This distinction highlights an important aspect of generics in Java.



`Box<Integer>` is not a subtype of `Box<Number>` even though `Integer` is a subtype of `Number`.

Note

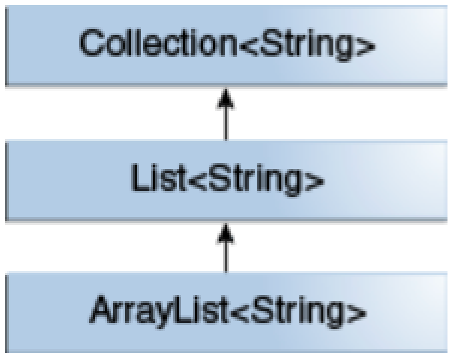
Given two concrete types `A` and `B` (for example, `Number` and `Integer`), `MyClass<A>` has no relationship to `MyClass<B>`, regardless of whether or not `A` and `B` are related. The common parent of `MyClass<A>` and `MyClass<B>` is `Object`.

For information on how to create a subtype-like relationship between two generic classes when the type parameters are related, see *Wildcards and Subtyping*.

Generic Classes and Subtyping

You can create a subtype of a generic class or interface by extending or implementing it. The relationship between the type parameters of one class or interface and those of another is determined by the `extends` and `implements` clauses.

For example, in the Collections framework, `ArrayList<E>` implements `List<E>`, and `List<E>` extends `Collection<E>`. As a result, `ArrayList<String>` is a subtype of `List<String>`, which is itself a subtype of `Collection<String>`. As long as you do not change the type argument, the subtyping relationship is maintained between the types.



A sample `Collections` hierarchy

Custom List Interface with Generic Payload

Imagine you want to define your own list interface, `PayloadList`, that associates an optional value of generic type `P` with each element. The declaration might look like this:

```
interface PayloadList<E, P> extends List<E> {
    void setPayload(int index, P val);
    // ...
}
```

The following parameterizations of `PayloadList` would be subtypes of `List<String>`:

- `PayloadList<String, String>`
- `PayloadList<String, Integer>`
- `PayloadList<String, Exception>`