Java Generics

Reuse of code and type-safety

There are situations when methods and classes do not depend on the data types on which they operate. For example, the *Collections.sort()* method can process Collections of strings, integers, or even custom classes without any modification. **It does not matter what the collection stores: the algorithm is always the same**.

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 come with complications.

Integer-based resizable array

Let's start from an enhanced array for integers.

```java
public class EnhancedResizableArray {
    public static final int DEFAULT_CAPACITY = 4;
    Integer[] v;

    public EnhancedResizableArray() {
        this.v = new Integer[DEFAULT_CAPACITY];
    }

    public int get(int index) {
        return v[index];
    }

    public void set(int index, int value) {
        if (index >= v.length) {
            Integer[] tmp = new Integer[index * 2];
            System.arraycopy(v, 0, tmp, 0, v.length);
            v = tmp;
        }
        v[index] = value;
    }

    public boolean contains(int value) {
        for (int i : v) {
            if (i == value) {
                return true;
            }
        }
        return false;
    }

    public void fill(int value) {
        Arrays.fill(v, value);
    }

    public int[] toArray() {
        return Arrays.copyOf(v, v.length);
    }

    public int length() {
        return v.length;
    }
}
```

Writing a slightly different version of the EnhancedResizableArray class for each possible type it could manage would represent a massive violation of the **DRY (Don't Repeat Yourself)** principle.

Object-based resizable array

To mitigate this issue we could use *Object[]* instead of *Integer[]*!

```java
public class EnhancedResizableArrayObject {
    public static final int DEFAULT_CAPACITY = 4;
    Object[] v;

    public EnhancedResizableArrayObject() {
        this.v = new Object[DEFAULT_CAPACITY];
    }

    public Object get(int index) {
        return v[index];
    }

    public void set(int index, Object value) {
        if (index >= v.length) {
            Object[] tmp = new Object[index * 2];
            System.arraycopy(v, 0, tmp, 0, v.length);
            v = tmp;
        }
        v[index] = value;
    }

    public boolean contains(Object value) {
        for (Object i : v) {
            if (i.equals(value)) {
                return true;
            }
        }
        return false;
    }

    public void fill(Object value) {
        Arrays.fill(v, value);
    }

    public Object[] toArray() {
        return Arrays.copyOf(v, v.length);
    }

    public int length() {
        return v.length;
    }
}
```

Now we are talking! We can store every kind of Object within the same container!

```
public static void main(String[] args) {
    EnhancedResizableArrayObject resizableArrayObject = new Enhance
    resizableArrayObject.set(0, "Hello");
    resizableArrayObject.set(1, "World");
    // The compiler complains here!
    String s = resizableArrayObject.get(0);
}
```

However, the compiler complains about the last line. It doesn't know what data type is returned. The compiler requires an explicit casting:

```
String s = (String) resizableArrayObject.get(0);
```

There is no contract that could guarantee that the return type of the list is a *String*. The array could hold any object. It can only guarantee that it is an *Object* and therefore requires an explicit cast.

This cast can be annoying, it clutters our code, and can cause type-related runtime errors if a programmer makes a mistake with the explicit casting (e.g., forget to use *instanceof*).

```
public static void main(String[] args) {
    EnhancedResizableArrayObject resizableArrayObject = new Enhance
    resizableArrayObject.set(0, "Hello");
    resizableArrayObject.set(1, new Point(2,3));

    // Run-time error! java.lang.ClassCastException
    String s = (String)resizableArrayObject.get(1);
}
```

Generic resizable array

It would be much easier if programmers could express their intention to use specific types and the compiler ensured the correctness of such types. This is the core idea behind generics.

```
List<Integer> list = new ArrayList<>();
```

By adding the diamond operator < > containing the type, we specify the type held inside the list allowing the compiler to enforce type-safety. In small programs, this might seem like a trivial addition. But in larger programs, this can add significant robustness and makes the program easier to read.

```java
public class EnhancedResizableArrayGeneric<T> {
    public static final int DEFAULT_CAPACITY = 4;
    T[] v;

    public EnhancedResizableArrayGeneric() {
        this.v = (T[])new Object[DEFAULT_CAPACITY];
    }

    public T get(int index) {
        return v[index];
    }

    public void set(int index, T value) {
        if (index >= v.length) {
            T[] tmp = (T[])new Object[index * 2];
            System.arraycopy(v, 0, tmp, 0, v.length);
            v = tmp;
        }
        v[index] = value;
    }

    public boolean contains(T value) {
        for (T i : v) {
            if (i.equals(value)) {
                return true;
            }
        }
        return false;
    }

    public void fill(T value) {
        Arrays.fill(v, value);
    }

    public T[] toArray() {
        return Arrays.copyOf(v, v.length);
    }

    public int length() {
        return v.length;
    }
}
```

```
public static void main(String[] args) {
    EnhancedResizableArrayGeneric<String> resizableArrayGeneric = ne
    resizableArrayGeneric.set(0, "Hello");
    resizableArrayGeneric.set(1, new Point(2,3));
    // Compile-time error!
    String s = (String)resizableArrayGeneric.get(1);
}
```

We traded a potential run-time error for a compile-time error! Nice!

Type parameters

A *generic type* is a class (or interface) that is parameterized over types. To declare a generic class, we need to declare a class with the type parameter section delimited by angle brackets < > following the class name.

Remember that only a reference type can be used as a concrete type for generics. This means that instead of primitive types, we use wrapper classes such as `Integer`, `Double`, `Boolean`, and so on.

Single type parameter

In the following example, the class `GenericType` has a single type parameter named `T`. We assume that the type `T` is "some type" and write the class body regardless of the concrete type.

```java
class GenericType<T> {

    /**
     * A field of "some type"
     */
    private T t;

    /**
     * Takes a value of "some type" and assigns it to the field
     */
    public GenericType(T t) {
        this.t = t;
    }

    /**
     * Returns a value of "some type"
     */
    public T get() {
        return t;
    }

    /**
     * Takes a value of "some type" and assigns it to a field
     */
    public void set(T t) {
        this.t = t;
    }
}
```

After being declared, a type parameter can be used inside the class body as an ordinary type. For instance, the above example uses the type parameter `T` as:

- a type for a *field*
- a *constructor* argument type
- an *instance method* argument and return type

Both *get()* and *set()* methods do not depend on the concrete type of `T`. They can take/return a string or a number in the same way.

```java
public static void main(String[] args) {
    GenericType<Integer> obj1 = new GenericType<>(10);
    Integer i = obj1.get();

    GenericType<String> obj2 = new GenericType<>("abc");
    String s = obj2.get();
}
```

Multiple type parameters

A class can have any number of type parameters. For example, the following class has two. Most generic classes have just one or two type parameters.

```java
public class Pair<K, V> {
    K key;
    V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() {
        return key;
    }

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

    public V getValue() {
        return value;
    }

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

The naming convention for type parameters

There is a naming convention that restricts type parameter names to **single uppercase letters**. Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class name.

The most commonly used type parameter names are:

- T -- Type
- S, U, V etc. -- 2nd, 3rd, 4th types
- E -- Element (used extensively in collections)
- K -- Key
- V -- Value
- N -- Number

Some examples from the Java API:

```
interface List<E> {
    boolean add(E e);
    E remove(int index);
    // ...
}

interface Map<K,V> {
    Set<K> keySet();
    Collection<V> values();
    // ...
}

interface Function<T,R> {
    R apply(T t);
}

class Collections {
    static <T extends Comparable<? super T>> void sort(List<T> list
    // ...
}
```

Creating objects of generic types

To create an object of a generic class, we need to specify the type argument following the type name:

```
List<Person> persons = new ArrayList<>();
```

If a class has multiple type parameters, we need to specify all of them:

```
Map<Integer, String> map = new HashMap<>();
```

Type Inference

Type inference is a Java compiler's ability to look at each method invocation and corresponding declaration to determine the type argument (or arguments) that make the invocation applicable. The inference algorithm determines the types of the arguments and, if available, the type that the result is being assigned, or returned. Finally, the inference algorithm tries to find the **most specific** type that works with all the arguments.

To illustrate this last point, in the following example, inference determines that the second argument being passed to the `pick` method is of type `Serializable`:

```java
public static <T> T pick(T a1, T a2) {
    return a2;
}

public static void main(String[] args) {
    Serializable s = pick("d", new ArrayList<String>());
}
```

Generic Methods

Generic static methods

Static methods cannot use type parameters of their class! Type parameters of the class these methods belong to can only be used in instance methods. If you want to use type parameters in a static method, declare this method's own type parameters.

The declaration of the generic type `T` surrounded by angle brackets allows us to use this type in the method. We remind you that it can belong to a generic or a non-generic class because it does not matter for generic methods.

```java
public static <T> T doSomething(T t) {
    return t;
}
```

The type parameter `T` can be used to declare the return type and the type of the method's arguments. A generic method can take or return values of non-generic types as well. For instance, the following method takes a generic array and returns its size as an `int`.

```java
public static <E> int length(E[] array) {
    return array.length;
}
```

We can pass different arrays to the method and find their length:

```java
public static void main(String[] args) {
    Integer[] intArray = { 1, 2, 3, 4 };
    String[] stringArray = { "a", "b", "c", "d" };
    int len1 = length(intArray); // pass an array of Integers
    int len2 = length(stringArray); // pass an array of Strings
    // ...
}
```

Just like in generic classes, the type parameter section can contain multiple type parameters separated by commas.

For instance, the following method declares two type parameters.

```
public static <T, U> void method(T t, U u) {
    // do something
}
```

We frequently use generic static methods for different operations with arrays and collections: sorting an array, searching for a value in a collection, reversing an array, and so on.

Generic instance methods

Just like static methods, instance methods can have their own type parameters. There is no difference in their declaration compared to static methods, excluding the absence of the `static` keyword.

```
class SimpleClass {

    public <T> T getParameterizedObject(T t) {
        return t;
    }
}
```

The class above does not provide a type parameter, so we have to specify the type parameter in the method declaration to make the method `getParameterizedObject` generic. Note that in this example we cannot use `T` as the type for a field in the class, because it belongs to the method rather than the class itself.

Now we can create an instance of the class and invoke the method. It will return a value that has the same type as the value that was passed in.

```
public static void main(String[] args) {
    SimpleClass instance = new SimpleClass();
    Integer value = instance.getParameterizedObject(601); // this w
    String value = instance.getParameterizedObject("Hello"); // thi
}
```

Although generic methods can belong to non-generic classes, it is more common for a generic method to belong to a class that is also generic.

```java
class SimpleClass<T> {

    public <U> T getParameterizedObject(T t, U u) {
        return t;
    }
}
```

The method receives arguments of both the class's type (`T`) and the method's own type (U). Since `T` was already declared in the class header, the method only has to declare the generic type `U`. The method returns a variable of type `T`.

As another example, consider the method *toArray()* defined within the java.util.List interface.

```java
interface List<E> {
    boolean add(E e);
    E remove(int index);
    <T> T[] toArray(T[] a);
    // ...
}
```

Type bounds

Constraining Type Parameters (classes)

Let's now consider an example that will reveal another aspect of generics. Imagine that we have a generic `Stack<E>` class that can contain objects of any class.

```java
public class Stack<E> {
    private List<E> items = new ArrayList<>();

    public void push(E e) {
        items.addLast(e);
    }

    public E pop() {
        return items.removeLast();
    }

    public boolean isEmpty() {
        return items.isEmpty();
    }
}
```

We can put any type of object inside `Stack<E>`. However, there are some situations in which we want to restrict these objects. We can say, for example, that the stack has to be able to contain only `CharSequence` objects (i.e., a common ancestor of both String and StringBuilder).

```
public class Stack<E extends CharSequence> {
    // ...
}
```

Now creating three `Stack` objects will lead to different results:

```
public static void main(String[] args) {
    Stack<String> strings = new Stack<>(); // ok
    Stack<StringBuilder> builders = new Stack<>(); // ok
    Stack<Point> points = new Stack<>(); // compile-time error!
}
```

It is worth noting that, a type variable may have a single type bound:

```
<T extends A>
```

or multiple type bounds:

```
<T extends A & B & C & ...>
```

The first type bound ("A") can be a class or an interface. The rest of the type bounds ("B" onwards) must be interfaces.

Constraining Type Parameters (methods)

Consider a generic method that finds the average of the values in a List. How can you compute averages when you know nothing about the element type? You need to have a mechanism for measuring the elements.

```
public interface Measurable {
    double getMeasure();
}
```

We can constrain the type of the elements, requiring that the type implement the Measurable type. In Java, this is achieved by adding the clause extends Measurable after the type parameter. We can express this concept with two different syntaxes:

```
public static <T extends Measurable> double average(List<T> objects
    // ...
}
```

or

```
public static double average(List<? extends Measurable> objects) {
    // ...
}
```

We are saying that T is a subtype of the Measurable type (i.e., T has at least the methods of the Measurable type). Here is the complete method:

```
public static <T extends Measurable> double average(List<T> objects
    if (objects.size() == 0) {
        return 0.0;
    }
    double sum = 0;
    for (T obj : objects) {
        sum = sum + obj.getMeasure();
    }
    return sum / objects.size();
}
```

Using the alternative syntax, we do not have access to the *T* type. Instead, we can use its upper bound: *Measurable*.

```
public static double average(List<? extends Measurable> objects) {
    if (objects.size() == 0) {
        return 0.0;
    }
    double sum = 0;
    for (Measurable obj : objects) {
        sum = sum + obj.getMeasure();
    }
    return sum / objects.size();
}
```

Which syntax?

Use < ? > whenever constraints between either parameters or return values are absent:

```java
// from java.util.Collections
public static void reverse(List<?> list) {}
public static void shuffle(List<?> list, Random rnd) {}
public static void swap(List<?> list, int i, int j) {}
```

Use < T > whenever constraints between either parameters or return values are present:

```java
// from java.util.Collections
public static <T> void sort(List<T> list, Comparator<? super T> c)
public static <T> void fill(List<? super T> list, T obj) {}
public static <T> void copy(List<? super T> dest, List<? extends T>
```

Given the absence of constraints, the rightmost approach for the above case would be:

```java
public static double average(List<? extends Measurable> objects) {
    // ...
}
```

Bounded wildcards

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:

```java
public class Stack<E> {
    // ...
    public void pushAll(Iterable<E> src) {
        for (E e : src) {
            push(e);
        }
    }
    // ...
}
```

This method compiles cleanly, but it isn't entirely satisfactory. It does not work with subtypes!

```
    public static void main(String[] args) {
        Stack<Number> numberStack = new Stack<>();

        // Ok
        numberStack.pushAll(new ArrayList<Number>(List.of(1,2,3)));

        // Compile-time error! List<Integer> is not a subtype of Li
        numberStack.pushAll(new ArrayList<Integer>(List.of(1,2,3)))
    }
```

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>*.

```
public class Stack<E> {
    // ...
    public void pushAll(Iterable<? extends E> src) {
        for (E e : src) {
            push(e);
        }
    }
    // ...
}
```

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:

```
public class Stack<E> {
    // ...
    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 does not seem entirely satisfactory when using subtypes.

```java
public static void main(String[] args) {
    Stack<Integer> integerStack = new Stack<>();
    integerStack.pushAll(new ArrayList<>(List.of(1,2,3)));

    // Ok
    integerStack.popAll(new ArrayList<Integer>());

    // Compile-time error! List<Number> is not a supertype of List<I
    integerStack.popAll(new ArrayList<Number>());
}
```

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). Again, there is a wildcard type that means precisely that: *Collection<? super E>*.

```java
public class Stack<E> {
    // ...
    public void popAll(Collection<? super E> dst) {
        while (!isEmpty()) {
            dst.add(pop());
        }
    }
    // ...
}
```

Josh Bloch's Rule

PECS is a mnemonic to help you remember which wildcard type to use. It stands for **producer-extends, consumer-super**.

In other words, if a parameterized type:

- represents a T producer, use *<? extends T>*
- represents a T consumer, use *<? super T>*

Here is the complete stack code:

```
public class Stack<E> {
    private List<E> items = new ArrayList<>();

    public void push(E e) {
        items.addLast(e);
    }

    public void pushAll(Iterable<? extends E> src) {
        for (E e : src) {
            push(e);
        }
    }

    public E pop() {
        return items.removeLast();
    }

    public void popAll(Collection<? super E> dst) {
        while (!isEmpty()) {
            dst.add(pop());
        }
    }

    public boolean isEmpty() {
        return items.isEmpty();
    }
}
```

Type Erasure

As noted above, it is still legal to use collection types and other generic types without supplying type parameters, but **you should not do it**. If you use raw types, you lose all the safety and expressiveness benefits of generics.

Given that you shouldn't use raw types, why did the language designers allow them? To provide compatibility. The Java platform was about to enter its second decade when generics were introduced, and there was an enormous amount of Java code in existence that did not use generics. It was deemed critical that all of this code remain legal and interoperable with new code that does use generics. It had to be legal to pass instances of parameterized types to methods that were designed for use with ordinary types, and vice versa. This requirement, known as *migration compatibility*, drove the decision to support raw types.

Type parameters are not *compiled* (i.e., passed to the runtime environment) but, instead, are *erased*. That is, they are replaced with ordinary Java types. **Each type parameter is replaced with its bound, or with Object if it is not bounded.**

```java
public class Pair<K, V> {
    K key;
    V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() {
        return key;
    }

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

    public V getValue() {
        return value;
    }

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

After type erasure, it becomes:

```
public class Pair {
    Object key;
    Object value;

    public Pair(Object key, Object value) {
        this.key = key;
        this.value = value;
    }

    public Object getKey() {
        return key;
    }

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

    public Object getValue() {
        return value;
    }

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

The same process is applied to generic methods. In the example below, returning the smallest item of a generic list, you can see how bounded types (i.e., Measurable is the upper bound of generic type E) are erased:

```
public static <E extends Measurable> E min(List<E> list) {
    if (list.isEmpty()) {
        throw new IllegalArgumentException();
    }

    E smallest = list.get(0);
    for (E item : list) {
        if (item.getMeasure() < smallest.getMeasure()) {
            smallest = item;
        }
    }
    return smallest;
}
```

After type erasure, it becomes:

```java
public static Measurable min(List<Measurable> list) {
    if (list.isEmpty()) {
        throw new IllegalArgumentException();
    }

    Measurable smallest = list.get(0);
    for (Measurable item : list) {
        if (item.getMeasure() < smallest.getMeasure()) {
            smallest = item;
        }
    }
    return smallest;
}
```

Resources

- https://www.baeldung.com/java-generics
- https://www.baeldung.com/java-generics-vs-extends-object
- https://www.baeldung.com/java-type-erasure