Generics

Josh Wilcox (jw14g24)

May 6, 2025

Table of Contents

1 What are Generics

2 Generic Lambdas

3 How Generics Work

4 Writing simple Generic Data Structures
- Generics with Nested Classes

5 The Wildcards

What are Generics

- Generics allow you to write code that works with different types while maintaining type safety
- They enable you to create classes, interfaces, and methods that can work with any data type
- Examples include:
  - Collections (ArrayList¡T¿, HashMap¡K,V¿)
  - Optional¡T¿
  - Custom generic classes
- Introduced in Java 5 to provide compile-time type safety

Generic Lambdas

- All functional interfaces are **generic**
- Common generic functional interfaces:
  - `Predicate<T>` - takes T, returns boolean
  - `Function<T,R>` - takes T, returns R
  - `BiFunction<T,U,R>` - takes T and U, returns R
- Type inference works automatically with lambda expressions
- Makes functional programming type-safe in Java

How Generics Work

How Generics are Possible

- There are two ways of allowing the use of Generics in Java
  - **Code Specialisation -** Create new class for every different type used in the same generic object
  - **Code Sharing -** Use one general class and determine types at runtime

Code Specialisation

- New class is generated for every instantiation of a new generic type or method
- At compile time:
  - Form list of all types the data structure uses in the code
  - Create a new class of that data structure and compile separately
- **Benefit**
  - Does not impact runtime performance
  - Easy to optimise compilation
- **Problem**
  - You need to know all the possible types at runtime
  - The executable is bigger

Code Sharing

- Compiler generates code for only one representation of a generic type
- It erases the generic type and replaces it with `Object` added
- At compile time:
  - All types are stripped from a generic and compiled as a raw type
  - Type checks and **casts** are automatically
    - Must be performed at runtime
- **Benefit**
  - No need to create loads of extra class files that might not be needed
- **Problem**
  - Extra type checking and casting takes time

Writing simple Generic Data Structures

Wrappers

```java
public class Wrapper<E> {
    protected E element; // can store a single thing of type E
    public Wrapper(){}
    public Wrapper(E a){element =a;}
    public E get(){return element;} // Pass or return an object of type E
    public void set(E a){element =a;}
    public String toString(){
        return element.toString();
    }
}

public static void main(String[] args){

    Wrapper<String> str= new Wrapper<String>("Wrap Me"); // Enforce type String
    Wrapper<Car> car= new Wrapper<>();
    car.set(new Car());
    Wrapper<Integer> num= new Wrapper<>(33); // Enforce type Integer
    Wrapper raw=new Wrapper("Some string"); // Raw types are still allowed!

    str.set("Arsenal");
    num.set(new Integer(11));
    raw.set(99);
    }
```

Pair

```java
public class Pair<K,V> {
    private K key;
    private V value;
    public Pair(K a1, V a2) {
    key = a1;
    value = a2;
    }
    public K getKey() { return key; }
    public V getValue() { return value; }
    public void setKey(K arg) { key = arg; }
    public void setValue(V arg) {value = arg; }
}
```

Generics with Nested Classes

Important Detail

- Inner classes within a class that uses generic uses the same generic type as the outer class

```java
public class ArrayList<E>{
    // All the stuff
    public class ArrayIterator implements Iterator<E>{
        // Iterator functionality
        @Override
        public boolean hasNext() {
        return false;
        }
        @Override
        public E next() {
        return null;
        }
    }
}
```

- You could not make an `ArrayList<String>` then create a nested class of the same as `ArrayIterator<Integer>`

Static Nested classes are different!

- Static nested classes **DO NOT** refer to the generic type of the enclosing class
- They are literally just classes within a class here

The Wildcards

What are Wildcards?

- Wildcards (?) represent unknown types in generics
- Different from type parameters (T) in several ways:
  - Cannot be used to declare variables or create new instances
  - Can only be used as type arguments in method parameters
  - Can use upper/lower bounds with extends/super

Examples:

```java
// Type parameter example
public <T> void processElements(List<T> list) { ... }

// Wildcard example
public void processElements(List<?> list) { ... }

// Bounded wildcard examples
public void processNumbers(List<? extends Number> list) { ... }
public void addIntegers(List<? super Integer> list) { ... }
```

- Use wildcards when you only need to read from or write to a collection
- Use type parameters when you need to refer to the type multiple times