Java Generics

Table of contents

Introduced in 2026-04-30-generics-in-java[1] (Lecture, Week 9).

Why generics?

Prior to Java 1.5, collection classes could hold *any* type of object — `ArrayList.add(Object o)` accepted anything, allowing types to be mixed, but required an explicit cast when retrieving elements:

```
List list = new ArrayList();
list.add("Hello");
list.add(123);           // allowed
String str = (String) list.get(0);   // OK, but no compile-time guarantee
Integer num = (Integer) list.get(1);
```

There was no guarantee only one type would end up in the collection, since `add` accepted any `Object` — a `String` could silently get mixed into a list that was only ever meant to hold `Cars`. **Generics** let a class/interface/method declare *which* type(s) it works with, giving:

[1] courses/csse2002/2026-04-30-generics-in-java.pdf

```
List<Car> carList = new ArrayList<Car>(); // or new ArrayList<>() - the diamond
 ↪  operator
carList.add("Not a product"); // compile-time error, not a runtime surprise
Car c = carList.get(0);       // no cast needed - the compiler already knows it's a
 ↪  Car
```

This is called **type safety** — catching and preventing type-related errors *early* (at compile time), rather than allowing unintended bugs (a stray `ClassCastException`) to surface at runtime.

Generic classes

A generic class parameterises a type across its whole body:

```
public class Print<T> {
    T i;
    Print(T i) { this.i = i; }
    public void print() { System.out.println(i); }
}

Print<Integer> p1 = new Print<>(1);
Print<Double> p2 = new Print<>(1.1);
Print<String> p3 = new Print<>("1.1");
```

By convention, type parameters are single letters: `T` (Type), `E` (Element — used throughout the Collections Framework), `K` (Key), `V` (Value), `N` (Number), and `S`/`U`/`V`… for a 2nd/3rd/4th type parameter.

Generic methods

A single method (rather than a whole class) can be made generic, independently of whether its enclosing class is generic:

```
public <T> void print(T data) {
    System.out.println("Lets shout " + data + "!!!");
}
// print(124);              // T inferred as Integer
// print("I love CSSE2002"); // T inferred as String
```

The compiler determines `T` from the argument's type automatically — this is **type inference**. Multiple type parameters can be declared together: `public <T, S, R> R genericMethod(T value1, S value2) { ... }`.

Bounded generics

A type parameter can normally accept *any* reference type. **Bounded generics** restrict it to a specific type (or one of its subtypes) using `extends`:

```
public <T extends Number> void print(T data) {
    System.out.println("Lets shout " + data + "!!!");
}
// print(123);    // OK, Integer is a Number
// print(123.2);  // OK, Double is a Number
// print("123");  // compile-time error, String is not a Number
```

`<T extends X>` means `T` can only accept data that are subtypes of `X` (despite the keyword, this works the same way whether `X` is a class or an interface).

Bounds through wildcards

Wildcards (`?`) bound what types can be substituted for a generic type parameter *at the use site* (e.g. in a method parameter), rather than when a generic class/method is declared.

Unbounded wildcard: `List<?>`

`List<?>` means "a list of some unknown type". Since Java doesn't know whether it's actually `List<String>`, `List<Integer>`, or something else, it **prevents writes**:

```
public static void addSomething(List<?> list) {
    list.add("Hello"); // compile-time error - could break whatever the real element
 ↪  type is
}
```

Use `<T>` when the method needs to *work with* the type; use `<?>` when the method only needs to *read* values (e.g. a `printList` that only calls `.get()`/iterates never needs to know the concrete type).

Upper bounded wildcard: `? extends T`

Represents `T` or any subclass of `T`. Used mainly to **read** data — you can't safely *add* to it, because the method doesn't know the exact subtype:

```java
List<? extends Number> nums = new ArrayList<Integer>();
Number n = nums.get(0); // OK: read as Number
nums.add(10);           // compile-time error - could be a List<Double>, etc.
```

This matters because **generics in Java are invariant**: even though `Double` is a subclass of `Number`, `List<Double>` is *not* a subclass of `List<Number>` (Effective Java, Item 31: parameterized types are invariant — `List<Type1>` is neither a subtype nor a supertype of `List<Type2>`). So a method that should accept a `List<Integer>` *or* a `List<Double>` and only needs to read `Number`s from it must be written as:

```java
public static double sum(List<? extends Number> list) {
    double sum = 0;
    for (Number n : list) {
        sum += n.doubleValue();
    }
    return sum;
}
```

Lower bounded wildcard: `? super T`

Represents `T` or any of its superclasses. Used when you want to **write** to a generic collection while keeping some flexibility in what type of collection is accepted:

```java
public static void addNumbers(List<? super Integer> list) {
    list.add(1);
    list.add(2);
}
// addNumbers(new ArrayList<Number>());  // OK, Number is a superclass of Integer
// addNumbers(new ArrayList<Object>());  // OK, Object is a superclass of Integer
// addNumbers(new ArrayList<Double>());  // compile-time error, Double isn't a
 ↪  superclass of Integer
```

Wildcards are usually bounded (`? extends T`, `? super T`) rather than left fully unbounded, to reduce flexibility just enough to improve type safety.

Type erasure

Generics only provide type checking at **compile time** — Java implements them via **type erasure**, which removes all type parameter information during compilation. Each type parameter is replaced with:

- its upper bound (e.g. `Number`, if declared `<T extends Number>`), or

- `Object`, if unbounded.

As a result, generic type information is not available at runtime — a compiled `List<String>` and a compiled `List<Integer>` are the same erased `List` class. This is why the compiler needs to insert a hidden cast for you:

```
// Source (compile time)
List<String> list = new ArrayList<>();
list.add("Hello");
String s = list.get(0);

// After erasure (what actually runs)
List list = new ArrayList();
list.add("Hello");
String s = (String) list.get(0); // compiler inserts this cast
```

For a bounded type parameter, the compiler substitutes the bound itself:

```
// Source
class SomeClass<T extends Number> {
    private T t;
    public void add(T t) { this.t = t; }
    public T get() { return t; }
}
// After erasure
class SomeClass {
    private Number t;
    public void add(Number t) { this.t = t; }
    public Number get() { return t; }
}
```

Practical example: a bookshop

A bookshop sells books across genres (`Fiction`, `Action`, `Fantasy`, …), stored on shelves; the shop needs to store and retrieve books of these different genres.

Without generics

A single `BookShelf` storing the abstract `Book` type loses genre information on retrieval — every `getItem()` call needs an explicit (unsafe) cast back to the specific genre:

```
public class BookShelf {
    private List<Book> inventory = new ArrayList<>();
    public void addItem(Book item) { inventory.add(item); }
    public Book getItem(int index) { return inventory.get(index); }
}
// Fiction f = (Fiction) shelf.getItem(0); // risk of ClassCastException, not type
 ↪  safe
```

Writing separate `FictionShelf`/`ActionShelf` classes fixes the type safety but duplicates the whole class for every genre.

With generics

A single generic `BookShelf<T extends Book>` gives type safety *and* reuse in one class:

```
public class BookShelf<T extends Book> {
    private List<T> inventory = new ArrayList<>();
    public void addItem(T item) { inventory.add(item); }
    public T getItem(int index) { return inventory.get(index); }
    public void displayBooks() {
        for (T book : inventory) {
            System.out.println(book + "(" + book.getGenre() + ")");
        }
    }
}

BookShelf<Fiction> fictionShelf = new BookShelf<>();
BookShelf<Action> actionShelf = new BookShelf<>();
fictionShelf.addItem(new Fiction("The Hobbit", "J.R.R. Tolkien"));
actionShelf.addItem(new Action("Die Hard", "Roderick Thorp"));
fictionShelf.displayBooks(); // getItem() on fictionShelf now returns Fiction
 ↪  directly, no cast
```