Java Generics

Table of contents

Why generics? (1) Type safety 1
Why generics? (2) One class instead of one class per type 3
Generic classes 3
Generic methods 3
Bounded generics 5
Bounds through wildcards 6
Type erasure 9
Practical example: a bookshop 10

Introduced in 2026-04-30-generics-in-java[1] (Lecture, Week 9). Practised again in week12-lab-generics[2].

Generics let **reference types** (`Integer`, `String`, … and user-defined types) be *parameters* when defining classes, interfaces and methods. There are two separate motivations for that, and they are worth keeping apart.

Why generics? (1) Type safety

Prior to Java 1.5, collection classes could hold *any* type of object — `ArrayList.add` was declared

```
public boolean add(Object o) {
    // code to implement add method.
}
```

so anything could go in, and an explicit cast was needed on the way out (see java-casting[3]):

[1] courses/csse2002/2026-04-30-generics-in-java.pdf
[2] courses/csse2002/week12-lab-generics.pdf
[3] courses/csse2002/java-casting.pdf

```
List list = new ArrayList();
list.add("Hello");
list.add(123);           // allowed
list.add(new Date());    // allowed

String str = (String) list.get(0);    // OK
Integer num = (Integer) list.get(1);  // OK
Date date = (Date) list.get(2);       // OK
```

The trap: `Integer num = (Integer) list.get(2);` compiles, because the compiler only knows `get` returns `Object`. It fails at *run time* with a `ClassCastException`, because slot 2 holds a `Date`. Nothing about the list stops a `String` being mixed into a collection that was only ever meant to hold `Car`s, because `add` accepts any `Object`.

Declaring the element type fixes both problems at once:

```
List<String> list = new ArrayList<>();  // <> is the diamond operator
list.add("Hello");
list.add(123);                // compile-time error
String str = list.get(0);     // no cast needed
```

```
List<Car> carList = new ArrayList<Car>();
Car c = carList.get(0);       // the compiler already knows it is a Car
```

This is **type safety**: type-related errors are caught and prevented *early*, at compile time, instead of surfacing as unintended bugs at run time.

> **i** What about `List<Object>`?
>
> ```
> List<Object> list = new ArrayList<>();
> list.add("Hello");
> list.add(123);
>
> Object object1 = list.get(0);
> Object object2 = list.get(1);
> ```
>
> It compiles, and it accepts both. But parameterising by `Object` buys nothing, because everything is an `Object`. Elements come back typed `Object`, so you are casting again, and `add` accepts anything again. Generics only pay off when the parameter is a *specific* type.

Why generics? (2) One class instead of one class per type

The second motivation is duplication, not safety. A class that prints an `Integer`:

```
public class PrintInt {
    Integer i;
    PrintInt(Integer i) { this.i = i; }
    public void print() { System.out.println(i); }
}
```

Wanting the same thing for `Double`, `Float` and `String` means writing `PrintDouble`, `PrintFloat` and `PrintString` — four classes with identical bodies and one word changed. Generics collapse them into one class that still preserves type safety.

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> print  = new Print<>(1);
Print<Double>  print2 = new Print<>(1.1);
Print<String>  print3 = new Print<>("1.1");
```

By convention, type parameters are single letters: `T` (Type), `E` (Element — used extensively by the Java Collections Framework, see java-collections-framework[4]), `K` (Key), `V` (Value), `N` (Number), and `S`/`U`/`V`… for a 2nd/3rd/4th type.

Generic methods

A single *method* can be made generic, independently of whether its enclosing class is. The type parameter is declared between the modifiers and the return type:

[4] courses/csse2002/java-collections-framework.pdf

```java
public <T> void genericsMethod(T data) {
    // do something with data
}

public <T> T genericsMethod(T data) {
    // do something with data here
    return data;                // returning any T-type value
}
```

Multiple type parameters can be declared together, and the return type can be a third one:

```java
public <T, S, R> R genericMethod(T value1, S value2) {
    return null;
}

String result = genericMethod(42, 3.14);
```

Type inference

A `Printer` whose `print` method is generic, in place of a fixed-type one:

```java
public class Printer {
    public <T> void print(T data) {
        System.out.println("Lets shout " + data + "!!!");
    }
}

Printer printer = new Printer();
printer.print(124);                 // T inferred as Integer
printer.print("I love CSSE2002");    // T inferred as String
```

> The Java compiler can automatically determine the type of `T` based on the type of the argument you pass to the method. This is known as **type inference**.

What if `Printer` were a generic *class* instead?

```
public class Printer<T> {
    public void print(T data) {
        System.out.println("Lets shout " + data + "!!!");
    }

    public static void main(String[] args) {
        Printer<String> printer = new Printer<>();
        printer.print("data");
        printer.print(123);
    }
}
```

A *class* type parameter is fixed once, when the object is created. `new Printer<>()` assigned to `Printer<String>` pins `T = String` for the life of that object, so `printer.print(123)` is a compile-time error. A *method* type parameter is inferred afresh at every call, which is why the generic-method version above accepted an `Integer` and a `String` through the same object. Choose the class parameter when one object should work with one type throughout; choose the method parameter when each call is independent.

Bounded generics

A type parameter can normally accept *any* reference type. **Bounded generics** restrict it to a specific type or its subtypes, using `extends`. The form is `<T extends X>`, and it means `T` can only accept data that are subtypes of `X`. Restricting the `Printer` to numbers:

```
public class Printer {
    public <T extends Number> void print(T data) {
        System.out.println("Lets shout " + data + "!!!");
    }
}

printer.print(123);      // OK, Integer is a Number
printer.print(123.2);    // OK, Double is a Number
printer.print("123");    // compile-time error - String is not a Number
```

Despite the keyword, `extends` here also covers implementing an interface.

Bounds through wildcards

Generic-type bounding restricts what may be substituted for the type parameter. Wildcards (`?`) do this *at the use site* — in a method's parameter list — rather than where a generic class or method is declared. Three forms:

Form	Name
`List<?>`	unbounded wildcard
`? extends T`	upper bound wildcard
`? super T`	lower bound wildcard

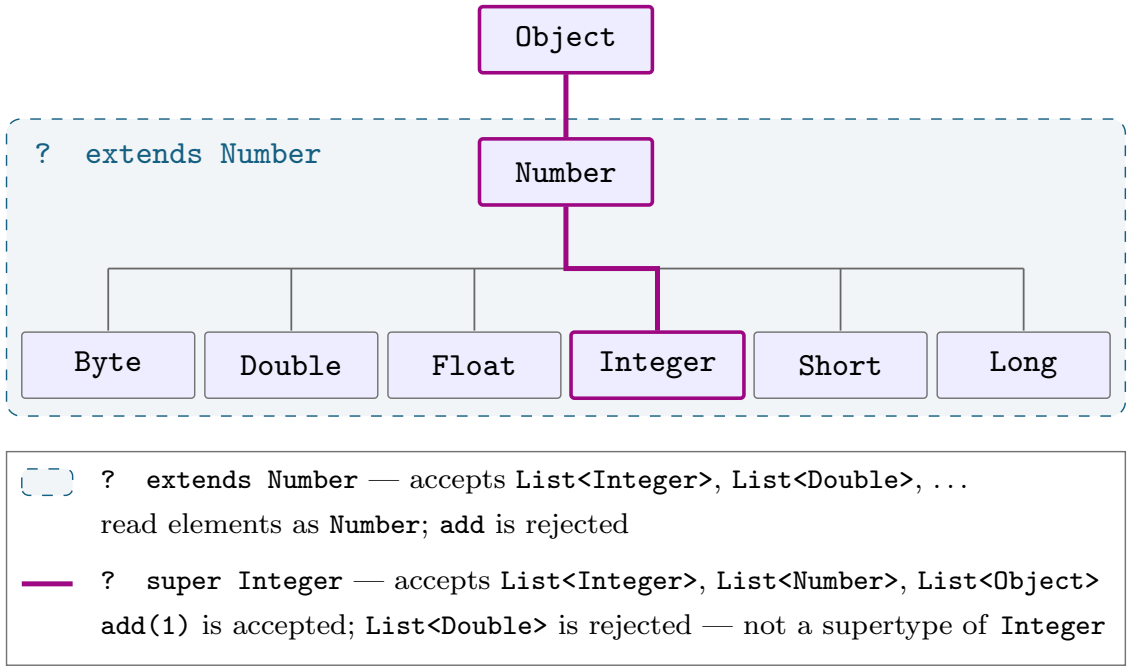


Figure 1: The two bounded wildcards drawn on the `Number` hierarchy. `? extends Number` admits `Number` and everything below it — you may read elements as `Number`, but `add` is refused; `? super Integer` admits `Integer` and everything above it — you may `add` an `Integer`, and `List<Double>` is refused. `Object` is on top because the `addNumbers` example passes a `List<Object>`.

Unbounded wildcard: `List<?>`

`List<?>` means "a list of some unknown type". Java does not know whether it is actually a `List<String>`, a `List<Integer>` or a `List<Product>` — **therefore it prevents the caller**

from writing to the list:

```
public class Main {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>();
        addSomething(numbers);
    }

    public static void addSomething(List<?> list) {
        list.add("Hello");   // error
    }
}
```

The compiler has to refuse: if Java allowed `"Hello"` to be added, the `List<Integer>` would now contain a `String`, breaking type safety.

Choosing between `<T>` and `<?>` for a read-only method:

```
public <T> void printList(List<T> toPrint) {
    for (int i = 0; i < toPrint.size(); i++) {
        System.out.println(toPrint.get(i));
    }
}
```

```
public void printList(List<?> toPrint) {
    for (int i = 0; i < toPrint.size(); i++) {
        System.out.println(toPrint.get(i));
    }
}
```

> Use `<T>` when you need to **work with the type**. Use `<?>` when you only need to **read values**.
>
> Wildcards are usually bounded (`<? extends T>`, `<? super T>`) to reduce flexibility and improve safety.

Upper bounded wildcard: `? extends T`

Represents `T` or any **subclass of** `T`. Used mainly to **read data** from a structure — you cannot safely *add* to it, because the exact subtype is unknown:

```
List<? extends Number> nums = new ArrayList<Integer>();

Number n = nums.get(0);  // OK: read as Number
nums.add(10);            // compile-time error
```

The reason this form is needed at all is **invariance**. Write a sum over a `List<Number>`:

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

List<Double> numbers = new ArrayList<>();
numbers.add(1.1);
sum(numbers);   // does not compile
```

In standard inheritance `Number n = 5.0;` is fine, because `Double` is a subclass of `Number`. But **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>`.) Widening the parameter with an upper bound is what makes the method accept both:

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

…and the price is that the method still *cannot safely write*, because it does not know whether the actual list is a `List<Double>`, a `List<Integer>`, or something else.

Lower bounded wildcard: `? super T`

Represents `T` or any of its **superclasses**. Used when you want to **perform write operations** on a generic collection but keep flexibility in which collections are accepted:

```
public class Main {
    public static void addNumbers(List<? super Integer> list) {
        list.add(1);   // Allowed: Integer is within bounds
        list.add(2);   // Allowed: Integer is within bounds
    }

    public static void main(String[] args) {
        List<Number> numbers = new ArrayList<>();
```

```
        addNumbers(numbers);   // fine: Number is a superclass of Integer

        List<Object> objects = new ArrayList<>();
        addNumbers(objects);   // fine: Object is also a superclass of Integer

        List<Double> doubles = new ArrayList<>();
        addNumbers(doubles);   // compilation error
    }
}
```

Type erasure

Generics provide type checking at **compile time** only. Java implements them by **type erasure**, which removes all type parameters during compilation, replacing each with

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

As a result, generic type information is **not available at run time**.

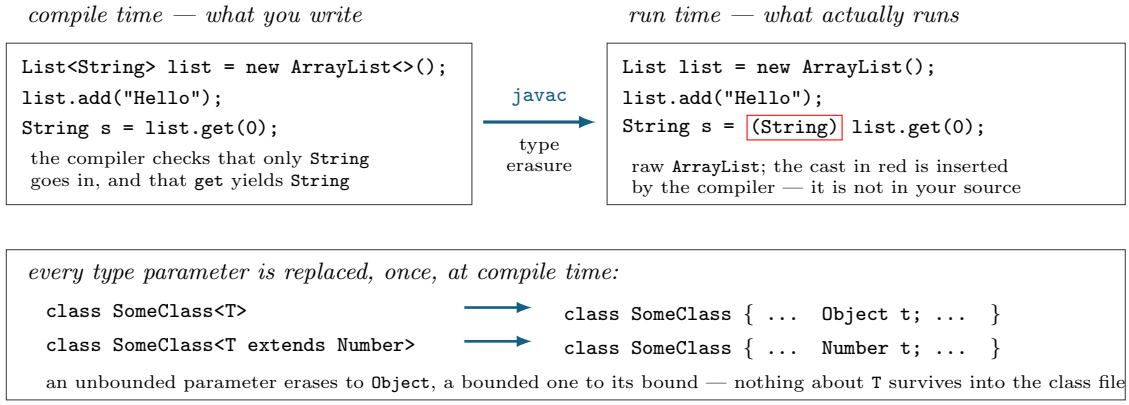


Figure 2: Type erasure as a pipeline: the compiler checks the source against the type parameters, erases each parameter to its bound (`Object` if unbounded), and inserts the casts the erased code needs — so the cast you were told you did not have to write still happens, just not by you. The top row is the compile-time source and what runs after `javac`, with the inserted cast in the red rectangle. The bottom row is what an unbounded and a bounded parameter each erase to.

Spelled out for a bounded parameter — the compiler replaces `T` with the extended class:

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

// After type erasure, the compiler generates
class SomeClass {
    private Number t;
    public void add(Number t) { this.t = t; }
    public Number get() { return t; }
}
```

and for an unbounded one, `T` becomes `Object` (`private Object t;`, `public void add(Object t)`, `public Object get()`).

Practical example: a bookshop

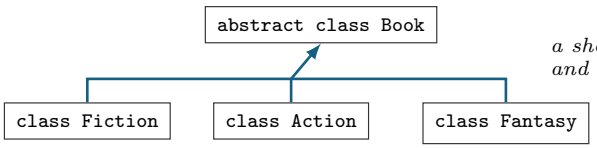
> Imagine you are developing a system for a bookshop that sells books across various genres: `Fiction`, `Action`, and `Fantasy`. Books are in bookshelves. Your job is to design the system in a way that allows the shop to store books of these different genres and retrieve them when needed.

The genres are subclasses of an abstract `Book`:

```java
public abstract class Book {
    private String title;
    private String author;

    public Book(String title, String author) {
        this.title = title;
        this.author = author;
    }
    public String getTitle()  { return title; }
    public String getAuthor() { return author; }
    public abstract String getGenre();
}

public class Fiction extends Book {
    public Fiction(String title, String author) { super(title, author); }
    @Override
    public String getGenre() { return "Fiction"; }
}
// Action is identical, returning "Action"
```



a shelf has to hold `Book`*s of one genre and give them back as that genre*

```
class BookShelf {
  List<Book> inventory;
  void addItem(Book item)
  Book getItem(int index)
}
```

One shelf, element type `Book`. Reusable, but `getItem` only ever promises a `Book`, so the caller has to cast: `Fiction f = (Fiction) shelf.getItem(0);`

reusable: yes type safe: no
a wrong cast is a `ClassCastException` at run time

```
class FictionShelf {
  List<Fiction> inventory;
  void addItem(Fiction b)
  Fiction getItem(int i)
}
```

A separate shelf class per genre. No cast needed — but `ActionShelf` and `FantasyShelf` are the same class written out again, once per genre.

reusable: no type safe: yes

```
class BookShelf<T extends Book> {
  List<T> inventory;
  void addItem(T item)
  T getItem(int index)
}
```

One generic shelf. `T` is fixed when the shelf is created, so `getItem` gives that genre back directly; the bound `extends Book` is what keeps `getGenre()` legal inside the class:

```
BookShelf<Fiction> shelf = new BookShelf<>();
```

reusable: yes type safe: yes

Figure 3: The three shelf designs, and why only the third is both reusable and type safe. One shelf of `Book` is reusable but needs a cast; a separate shelf class per genre gives better type safety but creates duplicated code; `class BookShelf<T extends Book>` gets both. The stacked outlines behind `FictionShelf` stand for the same class written out again for the other genres.

Without generics

One `BookShelf` storing the abstract `Book` type loses the genre on retrieval:

```
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); }

    public void displayBooks() {
        for (Book book : inventory) {
            System.out.println(book.getTitle() + " (" + book.getGenre() + ")");
        }
    }
}
```

The failure, worked through. The commented-out line is the one that breaks:

```
BookShelf shelf = new BookShelf();
shelf.addItem(new Fiction("The Hobbit", "J.R.R. Tolkien"));
shelf.addItem(new Action("Die Hard", "Roderick Thorp"));
shelf.addItem(new Fiction("Harry Potter", "J.K. Rowling"));

Book b = shelf.getItem(0);                      // returns Book

Fiction f  = (Fiction) shelf.getItem(0);        // fine
// Fiction f2 = (Fiction) shelf.getItem(1);     // breaks: index 1 is an
//                                              // Action -> ClassCastException
Action f2  = (Action) shelf.getItem(1);         // the caller has to know
```

`getItem()` returns only `Book`, so retrieving a genre needs a cast, and a wrong cast is a `ClassCastException` — **not type safe**.

Writing separate `FictionShelf`/`ActionShelf` classes instead does give better type safety, but every shelf is the same class copied out again:

```
public class FictionShelf {
    private List<Fiction> inventory = new ArrayList<>();
    public void addItem(Fiction book) { inventory.add(book); }
    public Fiction getItem(int index) { return inventory.get(index); }
    public void displayBooks() { for (Book book : inventory) { /* ... */ } }
}
// class ActionShelf { ...the same again with Action... }
```

With generics

A single generic `BookShelf<T extends Book>` gives type safety *and* reuse from one class. The bound matters twice over: it is what lets `displayBooks` call `book.getGenre()` (the compiler knows every `T` is at least a `Book`), and what stops a `BookShelf<String>` ever being created.

```
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"));
fictionShelf.addItem(new Fiction("Harry Potter", "J.K. Rowling"));
actionShelf.addItem(new Action("Die Hard", "Roderick Thorp"));

fictionShelf.displayBooks();   // getItem() here returns Fiction, no cast
actionShelf.displayBooks();
```