Generics in Java

Table of contents

Outline . . . 1
Why generics . . . 2
Generic methods, and the two exercises . . . 5
Bounded generics . . . 7
Bounds through wildcards . . . 7
Type erasure . . . 8
Practical example: the bookshop . . . 8
Homework . . . 8
Further reading . . . 9
Applied class . . . 9
Practical . . . 9

See csse2002[1] for course logistics.

Outline

1. Generics
2. Bounded Generics
3. Bounds through Wildcards
4. Type Erasure
5. Practical Example

Definitions, the full listings and the bookshop worked example are in java-generics[2]. What follows is the order of the argument and the exercises.

[1] courses/csse2002/csse2002.pdf
[2] courses/csse2002/java-generics.pdf

Why generics

There are two reasons for generics, and they have nothing to do with each other. Keep them apart:

1. **Type safety** — without a declared element type, a collection accepts anything and hands everything back as `Object`, so the caller casts, and a wrong cast is a run-time crash.
2. **Reuse** — without a type parameter, a class that works on one type has to be copied once per type.

Everything after that is the same parameter `T` being progressively *constrained*: unbounded (`<T>`) → bounded at the declaration site (`<T extends Number>`) → bounded at the use site (`? extends T`, `? super T`) → and finally erased entirely, so none of it survives compilation.

Why 01 — the cast you should not have to write

Prior to Java 1.5 collection classes could hold any type of object: `ArrayList` declared `public boolean add(Object o)`. Every element comes back as `Object`, so each `get` needs a cast, and the compiler cannot check that the cast is right.

Generics **provide type safety**. Type safety allows us to catch and prevent type-related errors early and thus prevent unintended bugs. A shopping cart is a second example: a raw `List cart` accepts `new Product("Laptop")` and then, by accident, `"Not a product"`. The generic version turns that second `add` into a compile-time error and drops the `(Product)` cast from the loop.

See java-casting[3] for what the cast itself is doing, and java-exceptions[4] for `ClassCastException`.

> **i** Fun fact: a list of `Object`
>
> `List<Object> list = new ArrayList<>();` accepts both `"Hello"` and `123`, and both come back as `Object`. What that shows is written up in java-generics[5].

Why 02 — the class you should not have to copy

The second motivation is duplication. A class that prints an `Integer` has to be written again for `Double`, `Float` and `String`. The four classes differ in one word, and one generic class replaces them.

[3] courses/csse2002/java-casting.pdf
[4] courses/csse2002/java-exceptions.pdf
[5] courses/csse2002/java-generics.pdf

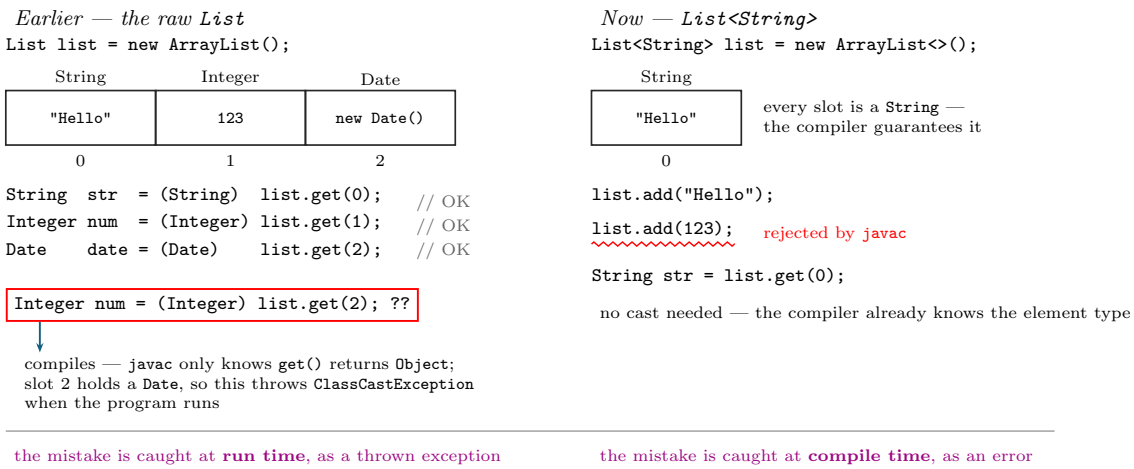

Figure 1: Type safety with and without generics. Left: a raw `List` holding a String, an Integer and a Date, with each cell's type above it and its index below. The three casts that match the element types all succeed. The boxed line, `Integer num = (Integer) list.get(2);`, compiles, but slot 2 holds a `Date`, so it throws `ClassCastException` when the program runs. Right: the same list parameterised by `String`, where the bad line `list.add(123)` is refused before the program ever runs and `get` needs no cast. The red squiggle marks a compile-time error. The magenta lines give the verdict: the raw list's mistake is caught at run time, the generic list's at compile time.

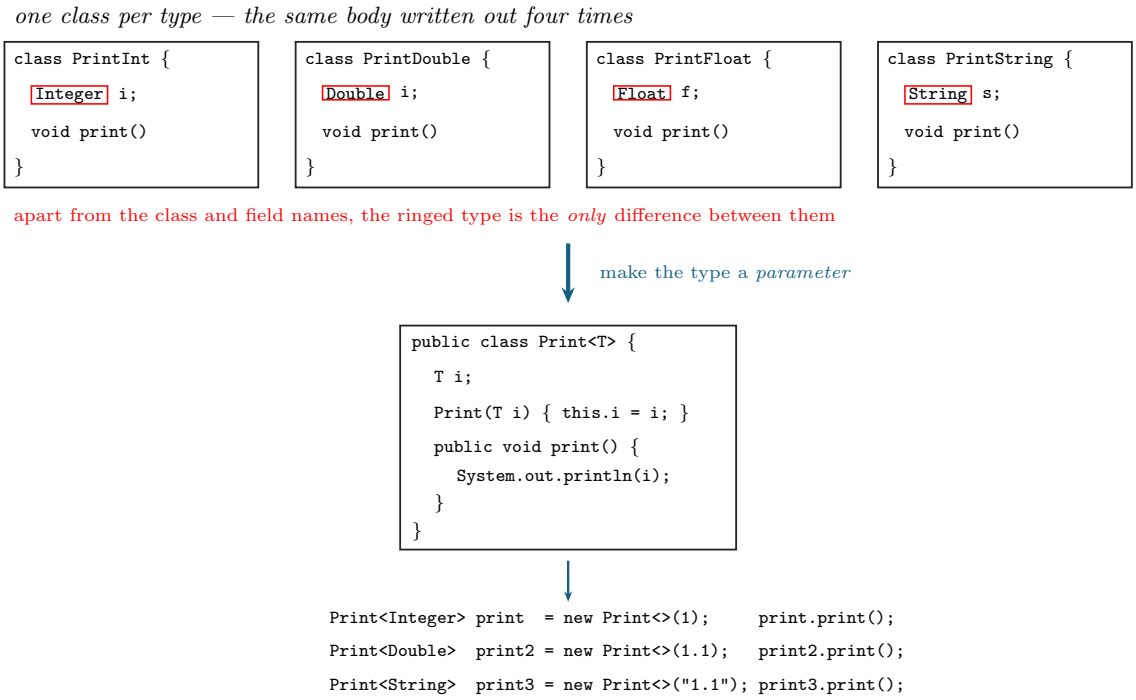

Figure 2: Reuse: four classes whose bodies are identical apart from one word, and the single generic class `Print<T>` that replaces them. Apart from the class and field names, the type outlined in red (`Integer`/`Double`/`Float`/`String`) is the only difference between the four. Each of the four is a `public class` and also carries a constructor and a one-line `print()` body (`System.out.println(...)`); all of that is elided here so the four fit in one row. At the bottom, three instantiations of the generic class with three different element types.

Generics enable **reference types** — `Integer`, `String`, … and user-defined types — to be *parameters* when defining classes, interfaces and methods. The word *reference* matters: `Print<int>` is not legal, only `Print<Integer>`.

Naming the type parameter

The convention is a single letter. These show up often in the Java libraries:

Letter	Stands for
`T`	Type
`E`	Element — used extensively by the Java Collections Framework (java-collections-framework[6])
`K`	Key
`V`	Value
`N`	Number
`S`, `U`, `V`…	2nd, 3rd, 4th types

Generic methods, and the two exercises

A *method* can carry its own type parameter, declared between the modifiers and the return type, whether or not its class is generic:

```
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
}
```

Several at once are allowed, and the return type may be a third one:

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

String result = genericMethod(42, 3.14);   // T = Integer, S = Double, R = String
```

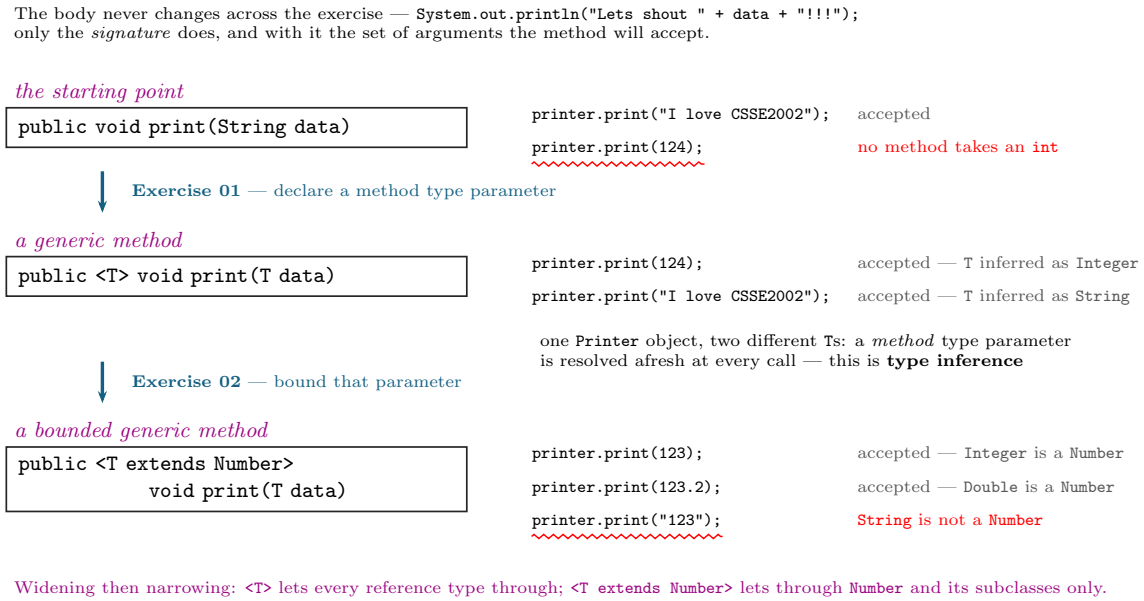

Figure 3: Exercises 01 and 02 as one progression. The body is the same at every stage; only the signature changes, and with it the set of arguments the method accepts. A red squiggle marks a call the compiler rejects. At the starting point `printer.print(124)` is rejected, and removing that restriction is what Exercise 01 exists to do. Exercise 02 then bounds the parameter, so the String call on the bottom row is rejected. The teal arrows are the two exercises, each a change of signature.

The two exercises take **one** method — `Printer.print` — through three signatures. Exercise 02 is an edit to Exercise 01's answer, so revise them as one sequence.

At the middle stage the Java compiler automatically determines the type of `T` from the type of the argument passed to the method. This is known as **type inference**.

Bounded generics

In general, the type parameter can accept any reference data type. Bounded generics are used if we want to use generics for some specific types.

The form is `<T extends X>`: `T` can only accept data that are subtypes of `X`. It is the java-inheritance[7] relation being used as a constraint. The example classes are `Person`, `Student` and `Professor`.

Despite the keyword, `extends` in a bound also covers *implementing an interface*. There is no `implements` form of a bound. This is needed to read library signatures.

Bounds through wildcards

The bound above sits at the *declaration* site. Wildcards move the same idea to the *use* site — a parameter list. There are three forms:

Form	Name	Use
`List<?>`	unbounded wildcard	read only; writing is refused outright
`? extends T`	upper bound wildcard	mainly to read data from a structure
`? super T`	lower bound wildcard	to perform write operations

Choosing between `<T>` and `<?>`:

- 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.

[6] courses/csse2002/java-collections-framework.pdf
[7] courses/csse2002/java-inheritance.pdf

The reason upper bounds are needed at all is **invariance** (*Effective Java*, Item 31): parameterized types are invariant, so `List<Type1>` is neither a subtype nor a supertype of `List<Type2>`. So `sum(List<Number>)` rejects a `List<Double>` even though `Number n = 5.0;` is fine. Widening the parameter to `sum(List<? extends Number>)` fixes it.

java-generics[8] carries the code for all three forms, the `addSomething` / `sum` / `addNumbers` examples, and a figure of the `Number` tree showing which lists each wildcard admits.

Type erasure

All of the above is **compile time only**. Java implements generics by erasing every type parameter during compilation — replacing it with its upper bound, or with `Object` if unbounded — so generic type information is not available at run time. The compiler then inserts the casts the erased code needs. The `(String)` cast that generics took out of the source is still there in the compiled code; you just did not type it.

The two erasure examples and the compile-time/run-time pair are drawn out in java-generics[9].

Practical example: the bookshop

A worked design problem: a bookshop selling `Fiction`, `Action` and `Fantasy` books held in bookshelves. There are three designs: one `BookShelf` of `Book` (reusable, needs casts, and one line throws), a separate shelf class per genre (type safe, duplicated), and finally `BookShelf<T extends Book>` (both). The listings and a figure comparing the three designs are in java-generics[10].

The bound is needed in that last version: `displayBooks()` calls `book.getGenre()`, which is only legal because the compiler knows every `T` is at least a `Book`. An unbounded `BookShelf<T>` would not compile.

Homework

- **What if we use a generic class instead?** Rewrite the `Printer` exercise with the type parameter on the *class* rather than the method, then work out why `printer.print(123)` is now an error on a `Printer<String>`. The reasoning is written up in java-generics[11].

[8] courses/csse2002/java-generics.pdf
[9] courses/csse2002/java-generics.pdf
[10] courses/csse2002/java-generics.pdf
[11] courses/csse2002/java-generics.pdf

Further reading

- https://docs.oracle.com/javase/tutorial/java/generics/index.html
- https://www.baeldung.com/java-generics
- https://youtu.be/K1iu1kXkVoA

The course's CodeCamp playlist has supplementary material, and the course Git repo has the code samples.

Generics come back in the Week 12 practical, week12-lab-generics[12].

Applied class

See week9-tutorial-substitution-principle-and-pre-post-conditions[13] — more practice with the Liskov Substitution Principle (from java-solid-principles[14], Week 7) and pre/postconditions (from java-specification[15], Week 3). It runs in three parts: ranking six pairs of predicates by **strength**; deciding, for each of those pairs, whether putting them in as *preconditions* or as *postconditions* keeps substitutability; and then writing Javadoc for a small `q2` method — `@param`/`@return` first, then `@requires`/`@ensures`, then a frame condition saying the method changes nothing in the array.

The worked `X`/`Y`/`Z` example is the one to remember: `Y` widens the precondition and narrows the postcondition, so it is legal; `Z` narrows the postcondition correctly but *also* narrows the precondition (`> 5` instead of `>= 5`, forbidding 5), and that is the violation.

Practical

See week9-lab-refactoring[16] — refactoring in the small and in the large, extending java-refactoring[17] (Week 6). The "in the small" half is a Connect 4 class taken through style fixes, better names, `while`→`for`, magic numbers to constants, decomposition into helper methods, comments, and finally removing the duplicated win-checking code into `checkWin` / `checkHorizontalWin` / `checkVerticalWin`. The "in the large" half breaks a monolithic `Library` class into `Book`, `BookShelf`, `Borrower` and `BorrowingSystem` — see java-cohesion-and-coupling[18].

That practical solution lands on `Book` and `BookShelf` — the same two classes the bookshop generics example is built from, arrived at from cohesion rather than from type safety.

[12] courses/csse2002/week12-lab-generics.pdf
[13] courses/csse2002/week9-tutorial-substitution-principle-and-pre-post-conditions.pdf
[14] courses/csse2002/java-solid-principles.pdf
[15] courses/csse2002/java-specification.pdf
[16] courses/csse2002/week9-lab-refactoring.pdf
[17] courses/csse2002/java-refactoring.pdf
[18] courses/csse2002/java-cohesion-and-coupling.pdf