Object-Oriented Programming Theory Notes

Tommaso Facchin

January 2026

Indice

1 Java

1.1 Encapsulation and abstraction

1.1.1 Class, methods and fields

A class in Java is a simple way to group related data and behavior under one name. A *class* describes how something should look and behave, and from that description you create *objects*, which are the actual things used at runtime. Each object has its own data, so two objects of the same class can hold different values even though they share the same structure.
Inside a class you usually find two main ingredients: *fields* and *methods*. Fields are just variables that live inside the class and represent its state, for example a person's name or age. Methods are operations that the class can perform, like printing information, updating a value, or computing a result based on the current fields. When you call a method on an object, that method can read and update the fields of that specific object, which is what makes each instance behave in a slightly different way.
Not all methods and fields play the same role. Some of them belong to a specific object, while others are shared by the whole class. For example, you might have one field that counts how many objects have been created in total; this kind of information is naturally shared and does not really belong to a single instance. Java also lets you control how visible your fields and methods are from the outside, but at the beginning it is enough to understand that the class is the place where you collect everything that logically belongs together. Over time, you refine this structure by deciding which fields to store, which methods to expose, and how objects of your class should be used in the rest of the program.

1.1.2 Static and final

In Java, the keyword `static` means "belonging to the class rather than to a single object". A *static* field is shared by all instances of the same class, so there is only one copy of that value in memory. This is useful for things like counters, configuration flags, or constants that are the same for every object. Static methods work in a similar way: they are called on the class itself and do not need an existing object to run, so they usually work only with their parameters, local variables, and other static members.
The keyword `final` is used to mark something as "not changeable" after it has been set. A *final* variable can be assigned only once, so after you give it an initial value, that value cannot be replaced with another one. A final field in a class is often used for constants or for data that should not change during the lifetime of an object. You can also use `final` on method parameters and local variables to make it clear that they will not be reassigned, which can make the code easier to reason about.
When applied to methods and classes, `final` has a slightly different meaning. A final method cannot be overridden in subclasses, which can be useful if you want to fix a certain behavior and prevent accidental changes in derived classes. A final class cannot be extended at all, so nobody can create a subclass of it. Together, `static` and `final` give you basic tools to control sharing of data, immutability, and how flexible your class hierarchy is, even in the early stages of learning the language.

1.1.3 Aliasing

In Java, aliasing happens when two or more variables refer to the *same* object in memory, not just to two separate objects with equal content. If you assign one reference variable to another, both variables become aliases of the same underlying instance. Changing the state of the object through one variable is immediately visible when you access it through the other, because there is only one object being shared.
This is very different from copying primitive values like `int` or `double`, where each variable holds its own independent value. With references, the variable does not store the object itself, but a sort of pointer to it, so assignments behave more like "share this object" than "duplicate this object". Aliasing is not necessarily bad, but it can lead to surprising bugs if you forget that multiple parts of your code are working on the same instance and accidentally modify it in one place while expecting it to stay unchanged in another.

1.1.4 Information hiding

Information hiding is the idea of keeping the internal details of a class invisible from the outside, and exposing only what other parts of the program really need to know. The class decides which data and helper methods stay private and which operations are safe to make public. From the point of view of the client code, you only interact with a clean interface, without depending on how things are actually stored or computed internally.
In Java this is usually done with access modifiers like `private`, `protected` and `public`. Fields are often declared `private`, while the class provides public methods to read or change them in a controlled way. This separation lets you change the internal representation later (for example, swapping one data structure for another) without forcing all the code that uses the class to be rewritten, as long as the public interface stays consistent.

Information hiding also helps with correctness and robustness. By limiting direct access to the internal state, you can enforce invariants inside the class, validate inputs before updating fields, and avoid situations where external code leaves your objects in an invalid or inconsistent state. In practice, this leads to code that is easier to maintain, test, and reason about, because each class has a clear boundary between its internal details and its visible behavior.

1.1.5 Modules

A module is a way to group together a set of related packages, classes, and resources under a single named unit. In Java, a module describes what it contains and which parts of it are visible to other modules, usually through a special descriptor file. This makes the overall structure of a large application more explicit: you can see which modules depend on which others, and you can avoid accidental use of internal code that was not meant to be reused.
Conceptually, modules sit at a higher level than classes and packages. A package organizes classes inside a project, while a module organizes packages and defines their boundaries. By declaring which packages are exported and which other modules are required, you can build clearer, more maintainable systems where dependencies are controlled instead of growing in an ad-hoc way. Even if you do not use the full Java module system at the beginning, thinking in terms of "modules" as coherent blocks of functionality is helpful when you design and split your codebase.

1.1.6 Documentation and Javadoc

Writing good documentation for a class means explaining what it does, how it should be used, and any important details that are not obvious from the code alone. In Java, the usual way to do this is to write structured comments directly above classes, methods, and fields, so that the explanation stays close to the code it refers to. Clear documentation is especially helpful for future you or for anyone else who has to use your code without reading every implementation detail.
Javadoc is the standard tool for turning these comments into readable documentation pages. You write special comments starting with `/** ... */` and use tags like `@param`, `@return`, and `@throws` to describe method parameters, return values, and possible exceptions. When you run the Javadoc tool, it scans the source files, extracts these comments, and generates HTML pages with a consistent layout, including class summaries, method lists, and links between related types.
Using Javadoc comments forces you to think about the public interface of your classes: what each method is supposed to do, what its inputs and outputs mean, and what guarantees it offers. This makes the code easier to understand and use, even in larger projects. Over time, a well-documented codebase feels more like a library you can browse and reuse, rather than a collection of files that you have to re-learn from scratch every time.

1.1.7 Java Virtual Machine (JVM) and Java bytecode

The Java Virtual Machine (JVM) is the program that executes compiled Java code. When you write Java source files and compile them, the compiler does not produce native machine code directly; instead, it generates an intermediate format called bytecode. This bytecode is stored in `.class` files and is what the JVM actually reads and runs. Because the JVM is available on many platforms, the same bytecode can, in principle, be executed on different operating systems without recompiling.
Java bytecode is a low-level, stack-based instruction set designed to be simple and portable. Each instruction tells the JVM to perform a small operation, such as loading a value, calling a method, or performing an arithmetic calculation. At runtime, the JVM interprets this bytecode or compiles frequently used parts into native code using a Just-In-Time (JIT) compiler to improve performance. From the developer's point of view, this whole process is mostly invisible: you write Java source code, compile it once, and rely on the JVM to handle the details of memory management, bytecode execution, and interaction with the underlying operating system.

1.2 Polymorphism

1.2.1 Extends, overriding and overloading

When one class `extends` another, it means the new class is built on top of an existing one. The subclass automatically inherits the fields and methods of the superclass, and can add new behavior or specialize the existing one. This is a way to reuse code and express "is-a" relationships, for example a `Student` that extends `Person`. The subclass can be used in many places where the superclass is expected, which makes it easier to write generic code that works with whole families of related types.
Overriding happens when a subclass provides its own implementation of a method that already exists in the superclass, with the same name, parameter types, and return type (or a compatible one). At runtime, when you call that method on an object, Java chooses the version based on the actual class of the object, not just the static type of the variable. This is a key part of polymorphism: different subclasses can override the same method to behave in different ways, while client code calls it through a common interface.
Overloading, in contrast, is about having multiple methods with the same name but different parameter lists in the same class (or in a class and its superclass). The compiler picks which overloaded method to call at compile time, based on the number and types of the arguments. Overloading is mainly a convenience feature: it lets you use a single, meaningful method name for several related operations, instead of inventing many slightly different names, while overriding is about changing or refining behavior in a class hierarchy.

1.2.2 Abstract and final

The keywords `abstract` and `final` are closely connected to how polymorphism works, because they tell you whether a class or method is meant to be extended or not. An `abstract` class is a class that cannot be instantiated directly: you cannot create objects of that type, you can only subclass it. Typically it contains some common fields and concrete methods, plus one or more `abstract` methods, which are declared but not implemented. These abstract methods act like "slots" that subclasses must fill in, forcing each concrete subclass to provide its own version of the behavior.
When you call an abstract method through a reference of the abstract type, the actual method that runs is the implementation defined in the concrete subclass. This is exactly the kind of dynamic polymorphism you want: client code can work with the abstract type and still get different behavior depending on which subclass instance it receives. Abstract classes are often used to capture a general concept (for example, a generic `Shape`) and let specific subclasses (`Circle`, `Rectangle`, ...) override the abstract methods in their own way.
The `final` keyword goes in the opposite direction: it restricts further extension or overriding. A `final` method cannot be overridden in subclasses, which means its implementation is fixed once and for all in the class where it is declared. A `final` class cannot be subclassed at all, so you cannot use inheritance to create polymorphic variants of it. From the point of view of polymorphism, `abstract` opens the door to specialization and dynamic dispatch through a common interface, while `final` closes that door and freezes part of the hierarchy when you want to prevent further changes or keep certain behaviors stable.

1.2.3 Inheritance, subtyping, and the substitution principle

Inheritance in Java is the mechanism that lets one class reuse and extend the code of another. A subclass automatically gets the fields and methods of its superclass, and can add new members or refine existing behavior. Conceptually, this models an "is-a" relationship: if `Student` extends `Person`, then every `Student` is also a `Person`, and can be used wherever a `Person` is expected.
Subtyping is the type-theoretic view behind this idea. A type `S` is a subtype of `T` when every object of `S` can safely be treated as an object of `T`. In Java, class inheritance and interface implementation introduce subtyping relationships: a subclass is a subtype of its superclass, and a class that implements an interface is a subtype of that interface. This is what allows variables, parameters, and return types to be declared using a general type while actually holding more specific objects at runtime.
The substitution principle (often called the Liskov Substitution Principle) says that instances of a subtype should be replaceable for instances of their supertype without breaking the correctness of the program. In practice, this means that code written against a base type should still behave sensibly when you pass in any of its subclasses. The more your subclasses respect the expectations and contracts established by their supertypes, the easier it is to exploit polymorphism: you can plug in new implementations without changing the client code that relies on the common supertype.

1.2.4 Static and dynamic types

Static and dynamic types describe two different ways of looking at the type of a variable or expression in Java. The *static type* is the type that the compiler sees in the source code: it comes from declarations like `Person p` or `List<String> names`. This type does not change during execution and is used at compile time to check that method

calls, assignments, and casts are valid. When you read code, the static type is what you see written next to the variable name.
The *dynamic type* (sometimes called the runtime type) is the actual class of the object that the variable refers to while the program is running. For example, if you declare `Person p = new Student();`, the static type of `p` is `Person`, but its dynamic type at runtime is `Student`. The dynamic type can vary over the lifetime of the variable as you assign different objects to it, as long as they are all compatible with the static type.
Polymorphism in Java is largely about this gap between static and dynamic types. You can declare variables, parameters, and return types using a common supertype or an interface, and then pass in objects of many different subclasses. At compile time, method calls are checked using the static type, but at runtime the JVM dispatches to the implementation determined by the dynamic type. Understanding this distinction helps explain why code like calling an overridden method on a base-type reference still executes the subclass version: the call is written against the static type, but the actual behavior depends on the dynamic type of the object.

1.2.5 Single and multiple inheritance

Single and multiple inheritance describe how many direct supertypes a class can extend. In Java, classes use *single inheritance*: each class can have at most one direct superclass, declared with the `extends` keyword. This keeps the class hierarchy simpler to understand and avoids certain ambiguities, like which implementation to inherit when two parent classes define the same method or field.
Multiple inheritance would allow a class to extend more than one superclass at the same time. This can be powerful, because a class could reuse code from several different parents, but it also introduces tricky questions about method resolution and the layout of state. Java avoids these problems for classes by not supporting multiple inheritance of implementation. Instead, it allows a form of multiple inheritance through *interfaces*: a class can implement many interfaces, gaining multiple types and sets of method contracts, while still inheriting concrete code from only a single superclass. This design lets you use polymorphism in a flexible way without taking on all the complexity of full multiple inheritance of classes.

1.2.6 Abstract classes and Interfaces

Abstract classes and interfaces are two different ways to describe common behavior in Java and to support polymorphism.
An abstract class is a class that you cannot instantiate directly. It can contain fields, concrete methods with full implementations, and abstract methods that have only a signature and no body. The idea is to capture shared state and behavior for a group of related subclasses, while leaving some operations unspecified so that each concrete subclass has to provide its own version. You typically use an abstract class when there is a clear "base implementation" that most subclasses can reuse, plus a few methods that are meant to be specialized.
An interface, on the other hand, focuses purely on the external behavior that a type should provide. Traditionally, interfaces contained only abstract method declarations, but modern Java also allows default methods with a body and static methods. Unlike abstract classes, interfaces cannot hold ordinary instance state in fields (beyond constants), and they do not represent a base implementation so much as a contract: any class that implements the interface promises to provide those methods. A key difference is that a class can implement many interfaces but can extend only one superclass, so interfaces are heavily used to model capabilities and to enable flexible polymorphism across unrelated class hierarchies.

1.2.7 Static and dynamic dispatch

Static and dynamic dispatch describe how Java decides which method body to execute when you make a method call. With *static dispatch*, the decision is made at compile time, based only on the static types of the expressions involved. This is what happens with method overloading: if you have several methods with the same name but different parameter types, the compiler picks which one to call by looking at the declared types of the arguments in the source code. Once compiled, that choice is fixed; the actual objects you pass at runtime do not change which overloaded version was selected.
Dynamic dispatch, instead, happens at runtime and is based on the dynamic type of the object. This is what you get with overriding: a subclass can override a method from its superclass, and when you call that method through a reference of the base type, the JVM chooses the implementation according to the real class of the object. In other words, the method name and parameter types are resolved statically, but the exact body that runs is chosen dynamically. This mechanism is at the core of subtype polymorphism in Java, because it allows code written against a general type to transparently use specialized behavior provided by different subclasses.

1.2.8 Generics

Generic types in Java let you write classes and methods that work with values of different types while still keeping type safety. Instead of fixing a class to a specific type, you introduce one or more type parameters, usually written between angle brackets, such as `<T>` or `<K, V>`. When you later use that generic class, you plug in concrete types for those parameters, for example `List<String>` or `Map<Integer, String>`, and the compiler checks that you only put the right kinds of objects into them.

The main benefit of generics is that they move type checking from runtime to compile time. Without generics, you would often use containers of `Object` and perform casts when reading elements, which is both verbose and error-prone. With generics, you declare exactly what a collection is supposed to hold, and the compiler prevents you from inserting incompatible types or forgetting a cast. This leads to clearer APIs and fewer class cast errors at runtime.

Generic types are used heavily in the standard library, especially in collections. Interfaces like `List<T>`, `Set<T>` and `Map<K, V>` are all generic, and many algorithms are written in terms of these parameterized types. When combined with polymorphism, generics let you express very general code: you can write a method that operates on a `List<T>` without caring about the concrete element type, and still get strong static type checking for each specific use.

1.3 Java in action

1.3.1 class hierarchy and Object class

The Java class hierarchy is organized as a tree rooted at a single base type called `Object`. Every class you define without explicitly extending another class automatically extends `Object`, either directly or indirectly. This means that all Java objects share a common ancestor, which provides a minimal set of operations that every instance in the system can support.
The `Object` class defines methods such as `toString()`, `equals(Object)`, `hashCode()`, `getClass()`, and `wait()/notify()/noti` among others. Concrete classes can inherit these default implementations or override some of them to provide behavior that better fits their own semantics, for example returning a more meaningful string representation or defining what it means for two instances to be considered equal. Because every class ultimately extends `Object`, these methods are available on any reference type, which is why you can safely call `toString()` or `equals()` on variables of many different types.
Thinking in terms of the class hierarchy helps when you design your own types. You can choose to extend a non-`Object` superclass to reuse code and behavior, but you always know that at the top of the chain there is `Object`, providing a common interface that all classes share. This shared root type plays an important role in collections, reflection, and generic code, since it allows APIs to accept or return values of the most general reference type when they truly need to work with "any object".

1.3.2 Primitive types, autoboxing and String

In Java there is an important distinction between primitive, or "native", types and reference types. Primitive types such as `int`, `double`, `boolean`, and `char` directly store simple values and are not objects. They are usually more lightweight and efficient to work with, and many low-level operations and arithmetic expressions are defined in terms of these primitives.
Sometimes, though, you need to treat primitive values as objects, for example when working with collections like `List` or `Map` that can only store references. For this purpose, Java provides wrapper classes such as `Integer`, `Double`, and `Boolean`. Autoboxing is the automatic conversion from a primitive type to its corresponding wrapper type, while unboxing is the reverse conversion. This means you can write code that looks like it is passing around primitives, but the compiler will transparently insert the necessary conversions when an object is required or when a primitive value is needed back.
Strings sit in a special place in the type system. A `String` in Java is a reference type, implemented as an object, but it behaves in many ways like a fundamental building block of the language. String literals like `"hello"` create `String` instances, and the class provides methods for common operations such as concatenation, substring extraction, and searching. Unlike many other objects, strings are immutable: once created, their content cannot be changed, and operations that seem to modify a string actually return a new instance. This immutability makes them safer to share between different parts of a program and works well with the way Java handles string literals and caching.

1.3.3 Exceptions

Exceptions in Java are objects that represent unusual or error conditions that occur while a program is running. Instead of crashing immediately when something goes wrong, Java lets you create, throw, and catch exceptions to handle these situations in a controlled way. This separates the normal flow of the program from the code that deals with problems such as invalid input, missing files, or failed network connections.
When an error is detected, code can `throw` an exception object. The runtime then starts looking for a matching `catch` block in the current method; if none is found, the exception is propagated up the call stack to the caller, and so on. If it reaches the top of the stack without being caught, the program usually terminates with a stack trace. By surrounding code that might fail with a `try` block and one or more `catch` blocks, you can intercept specific exception types and decide what to do: log the error, recover with a default value, retry, or rethrow a more meaningful exception.
Java distinguishes between checked and unchecked exceptions. Checked exceptions (subclasses of `Exception` but not `RuntimeException`) must be either caught or declared in the `throws` clause of a method, making potential failure points visible in the method's signature. Unchecked exceptions (subclasses of `RuntimeException`) can be thrown without being declared and are typically used for programming errors like null dereferences or index out of bounds. Using exceptions well means choosing meaningful exception types, not abusing them for normal control flow, and ensuring that resources such as files or streams are cleaned up properly, often with `try`-with-resources or `finally` blocks.

1.3.4 Annotations

Annotations in Java are a way to attach structured metadata to code elements such as classes, methods, fields, parameters, and local variables. They look like small markers starting with `@`, for example `@Override` or `@Deprecated`,

and do not change what the code does by themselves. Instead, they provide extra information that tools, libraries, or the Java compiler can read and act upon, either at compile time or at runtime.
Some annotations are handled directly by the compiler. A common example is `@Override`, which tells the compiler that a method is intended to override a method from a superclass or interface; if the signature does not match, you get a compile-time error instead of a silent bug. Others, like `@Deprecated`, signal that a class or method should no longer be used and can trigger warnings in the IDE.
Frameworks and libraries also define their own annotations to control behavior without requiring a lot of configuration code. For instance, in libraries for dependency injection or web development, annotations can be used to mark classes as components, map methods to HTTP endpoints, or describe how objects should be serialized. At runtime, these tools often use reflection to inspect the annotations present on the code and then adjust their behavior accordingly, which allows you to keep configuration close to the code and make it more declarative.

1.3.5 Reflection

Reflection in Java is the ability of a program to inspect and interact with its own classes and objects while it is running, even if their exact types were not known when the code was compiled. At the center of this mechanism is the `Class` type: every loaded class has a corresponding `Class` object, and from that object you can discover a lot of structural information. You can obtain a `Class` instance in different ways, for example with `MyClass.class`, with `obj.getClass()`, or with `Class.forName("mypackage.MyClass")` if you only have the name as a string. Once you have it, you can start exploring the members of that type.
The reflection API lets you list and inspect constructors, methods, and fields. Calling `clazz.getDeclaredMethods()` returns an array of `Method` objects representing all methods declared in the class, including private ones, while `clazz.getMethods()` returns only the public methods, including those inherited from superclasses. Each `Method` object exposes information such as the method name, return type, parameter types, modifiers, and annotations. Similarly, you can call `getDeclaredFields()` to inspect the fields of a class, or `getDeclaredConstructors()` to inspect its constructors. This is useful when you want to analyze a type at runtime without having written any specific code for it in advance.
Reflection can also be used to create objects and invoke methods dynamically. After you obtain a `Constructor` object, you can call `newInstance(...)` on it to instantiate a new object, passing the required arguments as an array. With a `Method` object, you can call `invoke(target, args...)` to execute that method on a given instance (or with a `null` target if the method is static). For fields, you can use `get(Object target)` and `set(Object target, Object value)` to read and write their values. Sometimes you also need to call `setAccessible(true)` (or the newer `trySetAccessible`) to bypass Java's normal access checks when working with non-public members, which is powerful but should be done with care because it breaks encapsulation.
Because reflection shifts many checks to runtime, it comes with trade-offs. Code that uses reflection is typically slower than direct method calls, more verbose, and more fragile: errors such as a misspelled field name or an incompatible argument type show up only when that part of the code runs. It also allows you to bypass access control, which can be necessary for certain frameworks but makes it easier to violate encapsulation and invariants if you are not careful. For day-to-day programming, it is usually better to rely on normal method calls and interfaces, and reserve reflection for cases where dynamic behavior is really needed, such as building frameworks, serializers, dependency injection containers, or testing tools that must work with arbitrary classes.

1.3.6 Library management and Gradle

Library management in Java is mostly about reusing external code (dependencies) without copying `.jar` files around by hand. In small projects you can still drop a JAR on the classpath manually, but this becomes painful as soon as you depend on more than a couple of libraries, or when different libraries require specific versions of the same dependency. A build tool solves this by letting you declare, in a single configuration file, which libraries you need and letting the tool download them, cache them, and put them on the classpath for compilation, testing, and execution.
Gradle is a popular build system that takes care of these tasks and more. A Gradle project typically has a file like `build.gradle` (Groovy) or `build.gradle.kts` (Kotlin) where you describe your project: which plugins you use (for example the Java plugin), which Java version you target, and which dependencies your code needs. Dependencies are listed with coordinates of the form `"group:artifact:version"`, for example:

```
dependencies {
    implementation("org.springframework.boot:spring-boot-starter-.2.0")
    testImplementation("org.junit.jupiter:junit-jupiter:5.10.0")
}
```

When you run commands like `gradle build` or `gradle test`, Gradle reads this file, downloads the required JARs from repositories such as Maven Central (only the first time), and compiles your source code with the correct classpath. It also wires together typical lifecycle tasks: compiling main sources, compiling tests, running tests, packaging a JAR,

and so on. For everyday work this means you can clone a project, install a JDK, and immediately build and run everything without manually setting up IDE project files or hunting for matching library versions.
Using Gradle also helps keep your project structure and dependencies explicit. Source files usually live in standard directories like `src/main/java` and `src/test/java`, so tools and IDEs know where to find them. If you need to add or update a library, you change a single line in the build file instead of touching multiple IDE configurations. Over time, this makes it easier to maintain and share Java projects: teammates and CI servers can all use the same build description, and you avoid the "works on my machine" problems that often appear when dependencies are managed manually.

1.3.7 MVC pattern (Model-View-Controller)

The Model–View–Controller (MVC) pattern is a way to structure an application by separating three main responsibilities: data and business logic (Model), presentation (View), and user interaction logic (Controller). The idea is to avoid mixing everything in a single class, so that changes in one area (for example the user interface) do not force you to rewrite the core of the application. This separation makes the code easier to understand, test, and evolve over time.
The *Model* represents the data and the rules that operate on that data. It knows nothing about how the information is shown to the user: it just exposes operations to query and update state, and may signal when something has changed.
The *View* is responsible for presenting the model to the user, for example drawing a window, rendering HTML, or printing text to the console. It should depend on the model to read information, but not contain complex business logic itself.
The *Controller* sits between the user and the model/view. It receives user input (button clicks, form submissions, commands), decides what that input means, and calls the appropriate methods on the model to update the state. It can also trigger changes in the view, such as switching screens or refreshing a part of the interface. In many Java applications and frameworks, especially on the web or in GUI toolkits, you will see classes explicitly named controllers, models, and views, or at least classes that play those roles. Thinking in MVC terms helps keep responsibilities clear: the model focuses on correctness of the domain logic, the view on showing things nicely, and the controller on connecting user actions to model operations.

1.3.8 Spring

Spring is a framework that helps you build Java applications by taking care of a lot of the plumbing code that would otherwise clutter your project. Instead of manually creating and wiring together all your objects, configuring them, and dealing with cross-cutting concerns like logging or transactions in every class, you describe how things should be connected and let Spring handle the details.
At the core of Spring is the IoC (Inversion of Control) container, which manages objects called *beans*. You typically define which classes should become beans, how they depend on each other, and any configuration they need, using annotations or configuration classes. When the application starts, the container creates these beans, injects their dependencies (Dependency Injection), and keeps them around so that the rest of your code can just ask for what it needs without calling `new` all over the place.
On top of this core, Spring offers many modules for common tasks: accessing databases, handling transactions, building web applications, exposing REST APIs, securing endpoints, and more. In practice this means you can focus on writing "plain" Java classes that express your domain logic, while annotations and configuration tell Spring how to expose them as controllers, services, or repositories. Even if you only see a small part of Spring at the beginning, the main idea stays the same: let the framework manage object creation and infrastructure, so your code can stay cleaner and more focused on the actual problem you are solving.

1.4 Advanced Java elements

1.4.1 Events, callback and anonymous class

Event-driven programming is a style where the flow of the program is controlled by events rather than by a fixed, linear sequence of instructions. An event can be a user action (a button click, a key press), a message arriving from the network, a timer firing, or some change in the system. Instead of constantly checking if something has happened, you register pieces of code that should run when a specific event occurs, and the underlying framework or runtime calls them at the appropriate time.
A callback is a method or function that you pass as an argument (or otherwise register) so that some other piece of code can call it later. In Java, callbacks are often modeled with interfaces: you implement an interface with a specific method (for example, `actionPerformed` in GUI code or `run` in a task), and then you hand an instance of that implementation to a framework. When the event happens, the framework invokes your method, effectively "calling you back". This decouples the event source from the concrete behavior: the source just knows it has to call an interface method, while your code decides what to do when that happens.
Anonymous classes provide a compact way to define small callback implementations directly where they are used, without creating a separate named class. Instead of writing a full class like `MyButtonListener` in its own file, you can create an unnamed class that implements the required interface on the spot, overriding just the method you need. This is very common in event-driven Java code, especially before lambda expressions were introduced: you see patterns like "new interface() public void method(...) ... " embedded inline. The combination of events, callbacks, and anonymous classes lets you express "when X happens, run this piece of code" in a flexible and localized way, which is exactly what you want in interactive or reactive applications.

1.4.2 Exceptions, error reporting, resource tracking

Exceptions in Java are a structured way to say "something went wrong here" without mixing error codes and checks into every line of normal logic. Instead of returning special values to signal failure, a method can throw an exception, and some other part of the program decides how to handle it. This separates the code that describes the normal behavior from the code that deals with problems, which usually makes both easier to read.
At a high level, you can think of three pieces: where exceptions are thrown, where they are caught, and what information they carry. Code that detects an error throws an exception object with a message (and sometimes extra details), and then stops its normal work. Somewhere higher in the call chain, a `try`/`catch` block catches that exception and decides what to do: maybe show an error to the user, log it for debugging, or convert it into a simpler form. The goal is not to avoid all failures, but to make sure that when failures happen they are noticed, reported clearly, and do not silently corrupt the program's state.
Resource tracking fits into this picture because many errors happen while dealing with external resources such as files, sockets, or database connections. You want these resources to be released reliably even when something goes wrong. That is why patterns like `try`-with-resources exist: they guarantee that cleanup happens automatically when you leave a block, whether you exit normally or because an exception was thrown. From a high-level perspective, good use of exceptions, error reporting, and resource tracking means your program can recognize unexpected situations, give useful feedback, and still leave the system in a clean, consistent state.

1.4.3 Parametric polymorphism

Parametric polymorphism is the idea of writing code that works uniformly for many different types, without having to duplicate the logic for each concrete type. In Java this shows up as generics: instead of fixing a class or method to use a specific type, you introduce one or more type parameters, like `T` or `K, V`, and then you can instantiate that generic code with different actual types later. The same implementation is reused, but the compiler treats each instantiation with a concrete type as if it were specialized, checking at compile time that you use it consistently.
A classic example is a generic container such as `List<T>`. The operations "add an element", "remove an element", or "iterate over the elements" do not really depend on whether `T` is `String`, `Integer`, or some custom class. With parametric polymorphism, you write those operations once in terms of the abstract type parameter `T`, and then the same list implementation can be used as `List<String>`, `List<Person>`, and so on. The important point is that this polymorphism is driven by type parameters rather than by inheritance: the behavior of the code does not change depending on the runtime type of `T`, but the type system still enforces that you only store and retrieve values of the right kind.
This form of polymorphism is different from subtype polymorphism, where you call a method on a base-type reference and get different behavior depending on the dynamic type of the object. With parametric polymorphism, the methods themselves are usually the same for all type arguments; what changes is the type information the compiler tracks. The benefit is that you can express very general algorithms and data structures once, reuse them for many types, and still get strong static guarantees about how they are used, reducing the need for casts and lowering the risk of type-related runtime errors.

1.4.4 Functional programming: lambda, function objects and closures

Functional programming in Java is about treating behaviour as values: instead of just passing data around, you also pass small pieces of code that can be stored in variables, given to methods, and returned as results. This style is useful when you want to describe "what to do" without tightly coupling it to "when and where it runs". Java supports this mainly through lambda expressions, functional interfaces, and the notion of closures.
A lambda expression is a compact way to write an anonymous function. Syntactically it looks like `(x) -> x * 2` or `(a, b) -> a + b`: you specify parameters on the left and an expression or block of code on the right. In Java, lambdas do not exist in isolation; they are always used in a context where the compiler expects a functional interface, that is, an interface with exactly one abstract method (for example `Runnable`, `Comparator<T>`, or `Function<T, R>`). When you assign a lambda to a variable of a functional interface type or pass it as an argument, the lambda becomes an instance of that interface. This is what "function object" usually means in Java: an object whose main purpose is to represent a piece of behaviour via its single abstract method.
Closures are lambda expressions or anonymous classes that can capture values from their surrounding scope. If a lambda uses a local variable defined outside of it, that variable's value is "closed over" and becomes part of the function's environment. In Java, the captured variables must be effectively final, meaning you do not reassign them after their initial definition, but you can still use their value inside the lambda. This lets you write small functions that carry some configuration or context with them, for example filtering a list based on a threshold defined outside the lambda. By combining lambdas, function objects, and closures, Java lets you write code that is more declarative and compositional, especially when working with streams, event handling, and higher-order utilities that accept behaviour as parameters.

1.4.5 Design pattern: factory, singleton, command, listener-observer and consumer-producer

Design patterns are recurring solutions to common design problems in object-oriented code. They are not language features but conventions that help structure code in a way that other developers can quickly recognize and reuse.

- **Factory** Object creation is moved into a dedicated method or class instead of calling constructors everywhere. The client asks the factory for an object and the factory decides which concrete implementation to instantiate and how to configure it.
- **Singleton** Ensures there is exactly one instance of a class in the whole application and provides a global access point to it. Typically implemented with a private constructor and a static method like `getInstance()`, but it should be used carefully because it introduces shared global state.
- **Command** Represents an action as an object that encapsulates all information needed to perform it later. Useful for undo/redo, task queues, and asynchronous execution, because commands can be stored, logged, and executed at a different time or in a different context.
- **Listener–observer** An object (the subject) maintains a list of listeners/observers and notifies them when an event happens or its state changes. The subject does not know the concrete behavior of listeners, it just calls their callback method, which keeps event producers and consumers loosely coupled.
- **Consumer–producer** One or more producers generate data or tasks, and one or more consumers process them, usually via a shared (often blocking) queue. This pattern decouples production speed from consumption speed and is common in concurrent programs where work is handed off between threads.

1.4.6 Threads

Threads in Java let a program perform multiple sequences of actions "at the same time" within a single process. Conceptually, each thread has its own call stack and executes its own flow of instructions, but all threads within the same JVM share the same heap, so they can access the same objects. This is useful for tasks like handling multiple client requests, performing background work while keeping a user interface responsive, or overlapping I/O and computation. At the same time, shared access to memory introduces the risk of race conditions and other subtle bugs if threads are not coordinated properly.
There are two classic ways to create a thread in Java: extending the `Thread` class or implementing the `Runnable` interface. Extending `Thread` means writing a subclass and overriding its `run()` method, then creating an instance and calling `start()`, which launches a new thread that executes `run()`. Implementing `Runnable` separates the task from the thread itself: you put the work inside `run()` of a `Runnable` implementation (often written as a lambda or anonymous class) and pass it to a `Thread` object. The `start()` method then creates a new thread and runs the `Runnable`. In modern code, the `Runnable` approach is usually preferred, especially when combined with higher-level concurrency utilities.
Once you have multiple threads, you need ways to coordinate them. One basic tool is `join()`, which lets one thread wait for another to finish before continuing. Another is interruption: by calling `interrupt()` on a thread and checking

the interrupted status inside long-running tasks, you can request that a thread stop what it is doing in a cooperative way. For many real-world programs, it is common to avoid managing raw threads directly and instead rely on thread pools and executors, which reuse a fixed number of threads to run many short tasks, reducing overhead and giving you more control over the lifecycle and number of threads.

Because threads share memory, you must be careful when they read and write the same data. If two threads modify a shared object without coordination, the order of operations can interleave in unpredictable ways, leading to inconsistent state or hard-to-reproduce bugs called race conditions. Java provides synchronization mechanisms to deal with this, such as the `synchronized` keyword on methods or blocks, which ensures that only one thread at a time can execute the protected region for a given lock object. There are also higher-level constructs in the `java.util.concurrent` package, like locks, atomic variables, and concurrent collections, which help you design safer patterns for sharing data.

A common pattern is to limit direct sharing of mutable state and instead communicate between threads using queues or other thread-safe structures. For example, in a producer–consumer setup, producers add tasks or messages to a blocking queue, and consumers take them out and process them. The queue handles the necessary synchronization, so threads do not need to coordinate manually on every item. More generally, writing good multithreaded code is often about designing clear ownership and communication patterns: decide which thread is responsible for which data, use well-defined handoff points, and avoid subtle implicit sharing when possible.

2 C++

2.1 C++ fundamentals

2.1.1 Pass-by-value vs references

In C++, passing arguments by value means that the function receives its own copy of the data. Changes made inside the function affect only that copy and are not visible to the caller. This is simple to reason about but can be inefficient for large objects, because each call requires a full copy.
Passing by reference lets the function work directly on the original object. Using a non-const reference allows the function to modify the argument, while a const reference lets it read the value without copying it and without the ability to change it. In practice, small types like ints are often passed by value, while larger or non-trivial types are usually passed by (const) reference to avoid unnecessary copies.

2.1.2 Const-correctness

Const-correctness is the discipline of marking data and functions as const whenever they are not supposed to modify state. A const variable cannot be changed after initialization, and a const reference promises that the referenced object will not be modified through that reference. This makes code easier to understand and enables the compiler to catch accidental mutations.
Member functions can also be marked const to say they do not alter the observable state of the object. This allows const objects (or const references to objects) to call those methods, and it clearly separates "read-only" operations from those that can mutate state. Writing const-correct code usually pays off quickly, because violations show up as compile-time errors instead of subtle runtime bugs.

2.1.3 Value-oriented style and stack objects

A value-oriented style in C++ treats objects primarily as values rather than as bare pointers or shared resources. In practice, this means preferring to create objects directly on the stack, pass them by value when they are small or cheap to copy, and rely on RAII (Resource Acquisition Is Initialization) for automatic cleanup. Stack allocation is fast and has clear lifetime semantics: objects are destroyed automatically when they go out of scope.
Even when an object manages a resource (like a file handle or dynamic memory), the value itself can still live on the stack and encapsulate the resource. This keeps ownership and lifetime explicit and avoids many manual new/delete pairs. Higher-level constructs like smart pointers fit into this style by behaving like value types that manage dynamic resources safely.

2.1.4 Member initialization syntax

C++ classes use a special syntax, the member initializer list, to initialize fields in a constructor. Instead of assigning values inside the constructor body, you write initializations after the constructor signature, before the opening brace. For example: `MyClass::MyClass(int n) :  value(n), flag(true) {}`.
Using the initializer list is not just syntactic sugar. Some members must be initialized there, such as references and const data members, because they cannot be assigned to later. It is also more efficient for complex types, since it constructs them directly in the desired state instead of default-constructing and then reassigning.

2.1.5 Default constructor

A default constructor is a constructor that can be called without any arguments. It either has no parameters or all its parameters have default values. The default constructor is used when you create objects with empty parentheses or no parentheses at all, and it is often required by various language features and libraries (for example, when containers need to create elements).
If you do not define any constructors, the compiler generates a default constructor for you. However, as soon as you declare certain other constructors, the implicit one may no longer be provided. Being explicit about whether a class should or should not be default-constructible helps clarify intent and avoids surprising compilation errors.

2.1.6 Copy constructor

The copy constructor defines how an object is initialized from another object of the same type. Its typical signature takes a const reference to the same class, for example `MyClass(const MyClass& other)`. The compiler uses it in situations like passing objects by value, returning them by value, or explicitly writing `MyClass b = a;`.
If you do not define a copy constructor, the compiler will generate one that performs a member-wise copy. For simple value-like classes this is often fine, but for classes managing resources, a naive copy can lead to double frees or shared

ownership issues. In those cases, you either implement a custom copy constructor that defines the correct semantics or delete copying entirely.

2.1.7 Conversion constructors and `explicit`

A constructor that can be called with a single argument acts as a conversion constructor: it allows the compiler to implicitly convert from that argument type to the class type. For example, a constructor `MyClass(int n)` enables code like `MyClass m = 42;` without an explicit cast. This can be convenient, but it can also introduce unexpected conversions and ambiguous overloads.
The `explicit` keyword is used to disable these implicit conversions. Marking a constructor as `explicit` means it can still be used, but only in places where the conversion is written out clearly, such as `MyClass m(42);` or `MyClass m{42};`. Using `explicit` on single-argument constructors is a common guideline to keep conversions under control and avoid subtle bugs.

2.1.8 Templated copy-conversion constructors

Template constructors that take parameters of arbitrary types make it possible to define generalized copy-conversion behavior between related types. A typical pattern is something like `template<typename U> MyClass(const MyClass<U>& other)`, which allows you to construct a `MyClass<T>` from a `MyClass<U>` when some compatibility condition holds. This is often used with class templates such as containers or wrappers.
These constructors let you support conversions like "vector of convertible types" or "smart pointer to derived to smart pointer to base" in a controlled way. At the same time, they must be designed carefully to avoid unintended matches and to respect the usual rules of copyability and type safety. Constraints and `enable_if`-style techniques (or concepts in modern C++) can help limit which template instantiations are allowed.

2.1.9 Const overloads on `this`

Methods in C++ can be overloaded based on whether they are const or non-const. A const member function promises not to modify the logical state of the object and can be called on const instances, while a non-const version can perform mutations. The compiler distinguishes them by attaching const to the hidden `this` pointer, effectively treating `T*` and `const T*` as different receiver types.
This pattern is common when you need both a read-only and a mutable view on the same underlying data. For example, you can provide a pair of methods returning a reference, where the const overload returns a const reference and the non-const one returns a mutable reference. Clients with const objects still get safe access, while non-const clients can modify the data when appropriate.

2.1.10 Assignment operator

The assignment operator (`operator=`) defines how an existing object is updated to take on the value of another object of the same type. Unlike the copy constructor, which initializes a new object, the assignment operator works on an already constructed instance. Its typical signature returns a reference to the current object so that chained assignments like `a = b = c;` behave as expected.
For simple classes, the compiler-generated assignment operator performs a member-wise assignment and is usually sufficient. For classes that manage resources, you often need to implement a custom assignment operator to handle self-assignment safely, release old resources, and correctly copy or transfer ownership from the right-hand side. In modern C++, this often comes together with move assignment to provide efficient behavior in all cases.

2.1.11 Overloading operators and methods

Operator overloading in C++ lets you define how built-in operators such as `+`, `-`, `==`, or `«` behave for your own types. Under the hood, this is just syntactic sugar for specially named functions, but it can make code much more readable when operators match the intuition of the domain (for example, adding vectors or comparing complex numbers). The key is to keep overloaded operators consistent with their usual meaning and avoid surprising side effects.
Method overloading works similarly: you can define several functions with the same name but different parameter lists. The compiler chooses which overload to call based on the types and number of arguments. This lets you express variations of the same operation in a compact way, but it also means you should design overload sets carefully to avoid ambiguity and ensure that the "best" overload is always the one you intend.

2.1.12 Destructors

A destructor is a special member function that runs automatically when an object's lifetime ends. Its job is to clean up any resources owned by the object, such as memory, file handles, or locks. In C++, destructors are central to RAII:

by tying resource management to object lifetime, you ensure that resources are released even when control leaves a scope due to exceptions or early returns.
If you do not define a destructor, the compiler generates one that simply calls the destructors of the member subobjects. For many classes this is enough, especially when you rely on other RAII types (like smart pointers) to manage resources. You only need a custom destructor when the class directly owns resources that require explicit cleanup.

2.1.13 Lvalues and rvalues

Lvalues and rvalues describe different "value categories" in C++ and are important for understanding references and move semantics. Roughly speaking, an lvalue refers to an object with a persistent identity and a stable location in memory, something you can take the address of and assign to. Variables and dereferenced pointers are typical examples of lvalues.
Rvalues represent temporary or disposable values, such as the result of most expressions *by value*. They are usually not bound to a named object that lives beyond the full expression. Modern C++ introduces rvalue references (with `&&`) to let you detect and efficiently "steal" resources from such temporaries. This is the foundation of move constructors and move assignment operators, which allow heavy objects to be transferred cheaply instead of copied.

2.2 Generic programming (GP)

2.2.1 Templated classes: parametric types

Templated classes in C++ let you define a single class blueprint that works with many different types. Instead of hard-coding a specific type, you introduce one or more type parameters, such as `template<typename T>` or `template<class T, class U>`. When you use the class, you plug in concrete types, for example `Vector<int>` or `Vector<std::string>`, and the compiler generates a specialized version for those types. This is the essence of **parametric** typing: the same class logic is reused for different type arguments.
This style avoids code duplication and keeps APIs more general. A single container template, for instance, can store any element type without needing separate implementations for integers, strings, or custom structs. From the user's point of view, templated classes behave like normal classes, but with extra flexibility and type safety provided at compile time.

2.2.2 Templated functions: parametric polymorphism

Templated functions generalize the idea of templates to standalone functions and methods. Instead of writing one overload per type, you declare a function template like `template<typename T> T max(const T& a, const T& b)`, and the compiler instantiates it for each type you call it with. This is often called *parametric polymorphism*: the function works for any type that supports the operations it uses, without caring about the concrete type.
Function templates are heavily used in the standard library, for algorithms such as `std::sort` or `std::accumulate`. They let you write algorithms once and apply them to many different containers and value types. In everyday code, this means you can express "do X for any type T that behaves like this" in a concise and type-safe way.

2.2.3 Static dispatch and overload resolution

In the context of generic programming, "static dispatch" refers to decisions that the compiler makes at compile time about which function or method to call. When you have overloaded functions or templates, the compiler uses the static types of the arguments and the overload resolution rules to pick the best match. Once compiled, that choice is baked into the generated code; there is no runtime lookup involved.
This is different from virtual functions, where the exact method implementation is chosen at runtime based on the dynamic type of the object. In generic code, static dispatch is often preferred because it can be fully optimized by the compiler, inlined, and stripped of unused branches. The trade-off is that error messages can become more complex, since everything happens during template instantiation and overload resolution.

2.2.4 Generative templates and delayed type checking

C++ templates form a generative system: they do not describe a single piece of code, but a pattern from which the compiler generates many concrete instantiations. Type checking for the body of a template is effectively delayed until the compiler instantiates it with specific template arguments. As a result, some errors only appear when a particular instantiation is compiled, not when the template itself is first parsed.
This delayed type checking is powerful but can produce long, cryptic error messages when something goes wrong. On the positive side, it allows templates to be extremely flexible: only the parts of the code that are actually used with a given type need to be valid for that type. This is one of the reasons why generic programming in C++ can express rich compile-time constraints and patterns, even though it sometimes feels more like a meta-language embedded in C++.

2.2.5 Templating containers and iterators

Generic programming shines when you combine templates with containers and iterators. Containers such as `std::vector<T>` or `std::list<T>` are templates over the element type, so they can hold any T that meets their requirements. Iterators generalize how you move through a sequence; instead of coding directly against a specific container, algorithms work with iterators that satisfy certain concepts (input iterator, random access iterator, and so on).
By templating both containers and algorithms over iterators, the standard library achieves a high degree of reuse. A single sorting algorithm can work with any random access iterator, regardless of the underlying container. This decouples data structures from algorithms and is one of the core ideas behind the STL-style generic programming approach.

2.2.6 Type traits and the template system

Type traits are small, template-based utilities that let you ask questions about types at compile time. For example, you can check whether a type is integral, whether it is trivially copyable, or whether it has a certain member function.

In the standard library, these utilities live in headers like `<type_traits>` and expose results through nested typedefs or constexpr boolean values.
Combined with templates, type traits allow you to write conditional code at compile time. You can enable or disable certain overloads depending on properties of the template arguments, or select between different implementations for performance or correctness reasons. Modern C++ adds features like `std::enable_if` and concepts to make this pattern more expressive and readable, but the underlying idea remains: use compile-time information about types to guide how templates are instantiated.

2.2.7 Member types

Many generic classes define nested types, often called member types, that describe related type information. For example, a container might expose `value_type`, `iterator`, and `const_iterator` as nested typedefs or using-declarations. Generic code can then refer to these member types using syntax like `Container::value_type` to discover what kind of elements the container holds or what iterator type to use.
This pattern helps generic code stay flexible and decoupled from concrete implementations. Instead of assuming that every container uses a particular element type or iterator, templates can work in terms of member types defined by the container itself. In more advanced settings, traits classes and aliases build on this idea to standardize common member-type names across many templates.

2.3 Object-oriented programming (OOP) in C++

In C++, object-oriented programming builds on classes, inheritance, and virtual methods to model families of related types and share behavior. The language gives you both low-level control (manual memory management with `new` and `delete`) and higher-level tools (smart pointers, generic containers, and concepts) to structure programs around objects while keeping lifetimes and ownership explicit.

2.3.1 Inheritance, subsumption, and dynamic dispatch

Inheritance lets one class reuse and extend another. With `public` inheritance you model a clean "is-a" relationship: a derived class object can be used anywhere a base class is expected, which is the core idea behind subsumption for pointers and references. A `Derived*` can be implicitly converted to `Base*`, and the same holds for references, so code can store heterogeneous objects behind base-class handles and still treat them uniformly.
Virtual methods and the `override` keyword are what turn this static hierarchy into runtime polymorphism. Marking a method as `virtual` in the base tells the compiler to dispatch calls through a vtable; derived classes provide specialized behavior by overriding that method. Using `override` in the derived class makes the intent explicit and lets the compiler catch mistakes where the signature does not actually match the base version.

2.3.2 Heap allocation and smart pointers

C++ lets you create objects either with automatic storage (on the stack) or dynamically on the heap. Using `new` constructs an object on the heap and returns a raw pointer; later you are responsible for calling `delete` to destroy the object and free the memory. This gives a lot of flexibility for objects whose lifetimes outlive the current scope, but it also introduces classic problems like leaks, double deletes, and dangling pointers if you are not disciplined.
Modern code tends to avoid bare `new`/`delete` in user code and wraps heap objects in smart pointers. Types like `std::unique_ptr` express exclusive ownership, while `std::shared_ptr` implements reference-counted shared ownership. These wrappers still give you pointer-like syntax, but they automatically release resources when the last owner goes out of scope, fitting nicely with RAII and making object lifetimes easier to reason about.

2.3.3 STL containers, iterators, and traits

The Standard Template Library (STL) provides a generic backbone for storing and processing objects in an OOP design. Containers such as `std::vector`, `std::list`, and `std::map` are templates that can hold any type, including class hierarchies accessed through pointers or smart pointers. Algorithms operate over iterator pairs instead of concrete containers, so the same function (like `std::sort`) can work with many different underlying data structures as long as their iterators meet the right requirements.
Iterator and type traits help generic code adapt to different kinds of iterators and value types. Utilities like `std::iterator_trait` expose nested types (value type, difference type, iterator category), and more modern type traits and concepts let you express constraints directly on template parameters. Together, these tools connect the object-oriented world of classes and virtual methods with the generic world of templates and compile-time requirements, making it possible to write flexible libraries that still feel type-safe and efficient.

2.4 Modern C++ extensions

2.4.1 Modules (C++20)

Modules are a modern alternative to the traditional header-based inclusion model. Instead of relying on preprocessor `#include` chains and textual substitution, a module lets you compile an interface once and then import it where needed. This reduces compile times, avoids many macro-related issues, and gives a clearer separation between what a component exports and what it keeps internal.
From the programmer's point of view, modules make dependencies more explicit and less fragile. You expose a well-defined interface, and importing code sees only what the module chooses to make visible. Over time, this encourages better boundaries between components and helps large codebases scale without drowning in header includes.

2.4.2 `auto` and type inference (C++11)

The `auto` keyword lets the compiler deduce the type of a variable from its initializer. This is especially useful when the type is long or noisy (for example, iterator types or complex template instantiations), or when it would be easy to make a mistake copying the exact type by hand. You still get static typing, but you do not have to spell out the full type name every time.
Used thoughtfully, `auto` can improve readability by focusing attention on what a variable represents rather than on its exact, possibly verbose type. At the same time, overusing it everywhere can make code harder to understand, so many developers reserve it for obvious cases where the initialization clearly communicates the underlying type.

2.4.3 Lambda expressions (C++11)

Lambda expressions are a compact way to define anonymous function objects directly at the point of use. They capture variables from the surrounding scope and can be passed to algorithms, stored in containers, or used as callbacks without the ceremony of writing a separate named struct. A lambda has a parameter list, an optional capture list, and a body, and the compiler synthesizes a suitable function-object type behind the scenes.
Lambdas fit naturally with generic programming and the STL. Instead of defining small functor types just to customize an algorithm, you can write the logic inline, right next to the call. This often makes code more local and easier to follow, especially in event-driven or algorithm-heavy parts of a program.

2.4.4 `decltype` (C++11/14)

The `decltype` construct lets you query the type of an expression at compile time. You write `decltype(expr)` to obtain the exact type (including references and const qualifiers) that the expression would have. This is particularly useful in templates, where you want to define a type based on the result of some operation without manually recreating complicated type expressions.
In more advanced code, `decltype` combines with `auto` and trailing return types to express functions whose return type depends on their template parameters. It also helps when writing generic wrapper types or forwarding utilities, because you can preserve the precise type of an expression instead of approximating it.