# Chapter 3 - Polymorphism

Polymorphism

Chapter 3 – Polymorphism

Chapter Goals
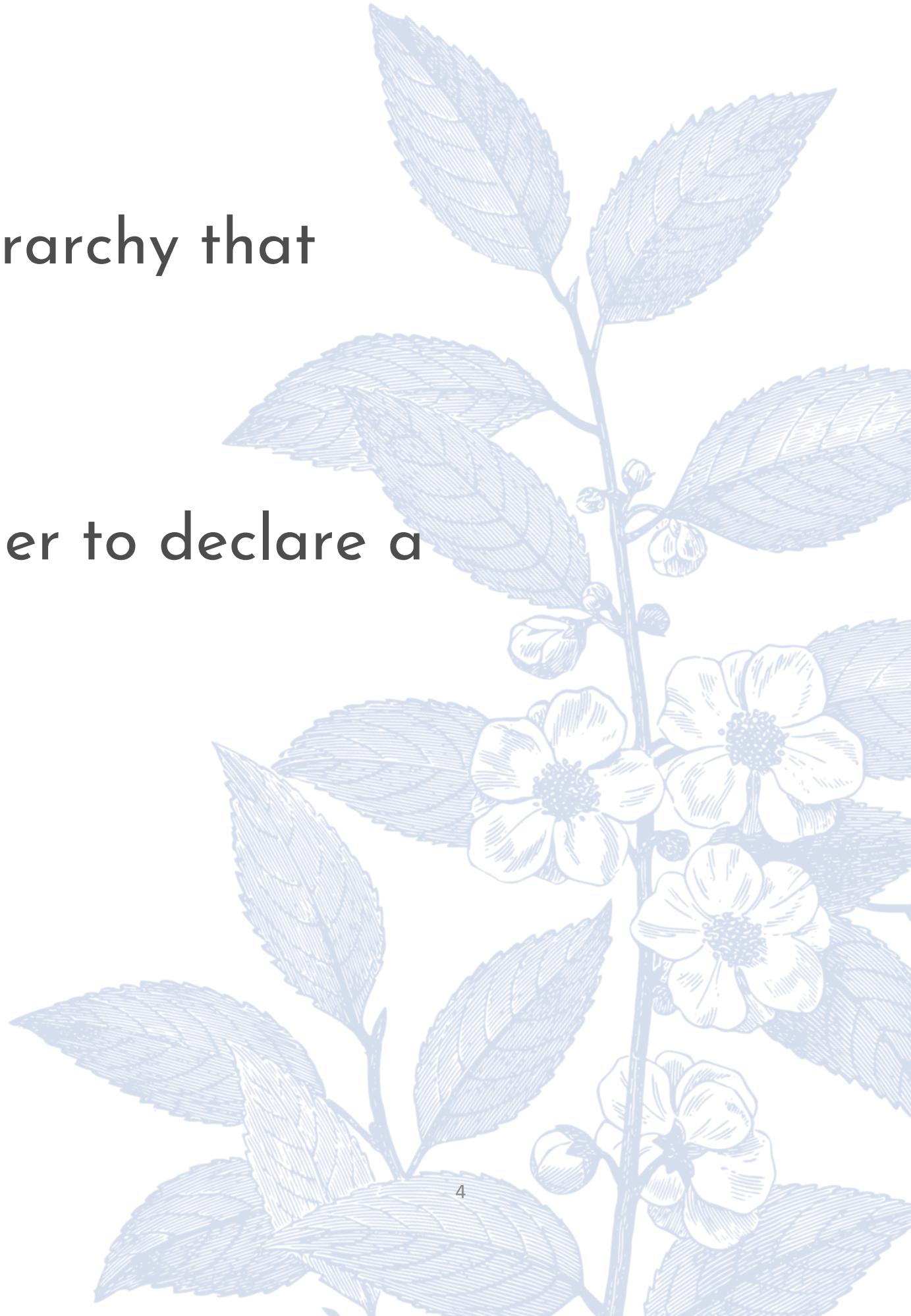
- ✓Abstract Classes and Abstract Methods

- ✓Interfaces

- ✓defining polymorphism and its benefits

- ✓using inheritance to create polymorphic references

- ✓using interfaces to create polymorphic references

- ✓using polymorphism to implement sorting and searching algorithms

- ✓Exception

Abstract Classes
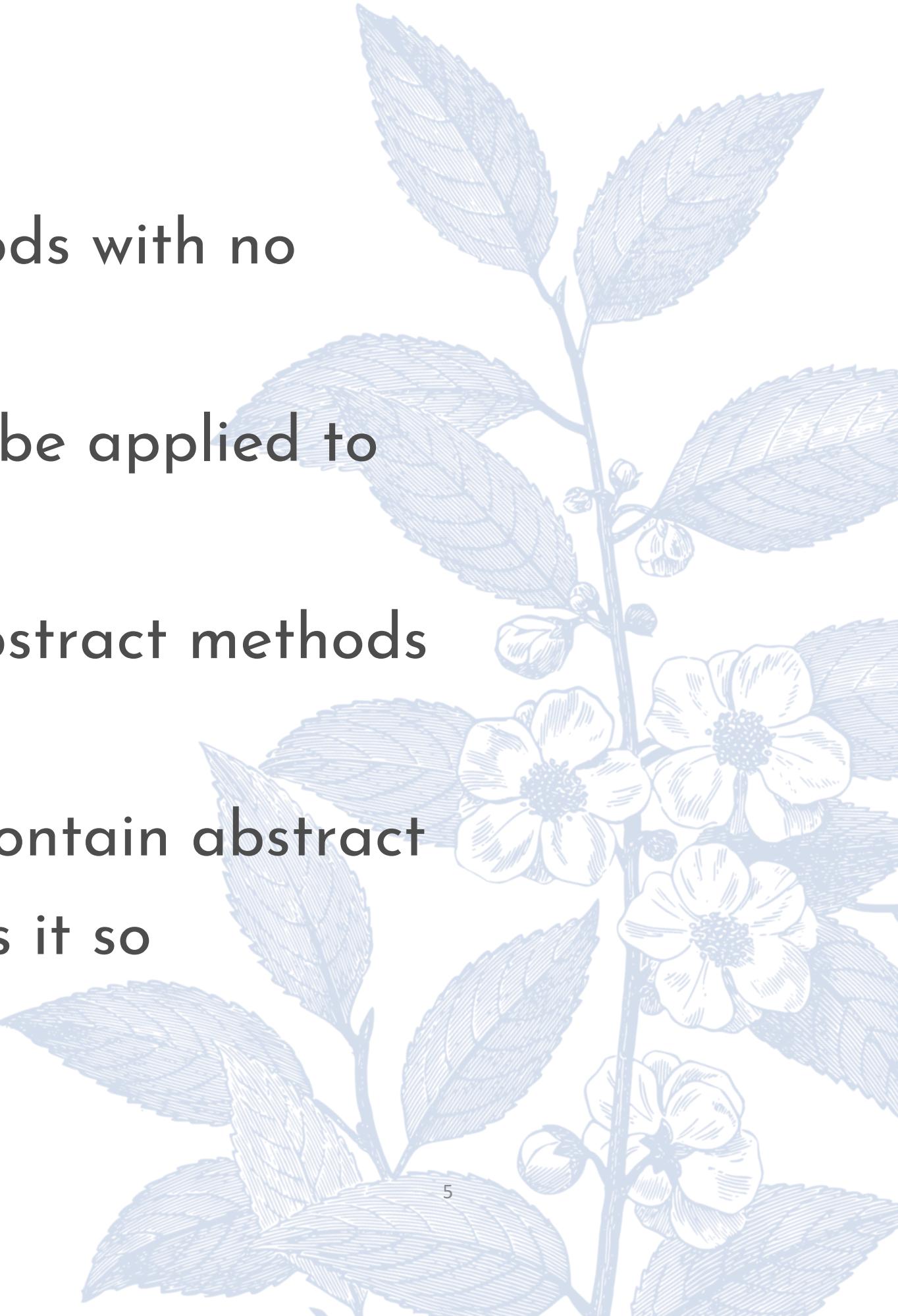
✓An abstract class is a placeholder in a class hierarchy that represents a generic concept

✓An abstract class cannot be instantiated

✓We use the modifier abstract on the class header to declare a class as abstract

```
public abstract class Product
{
    // class contents
}
```
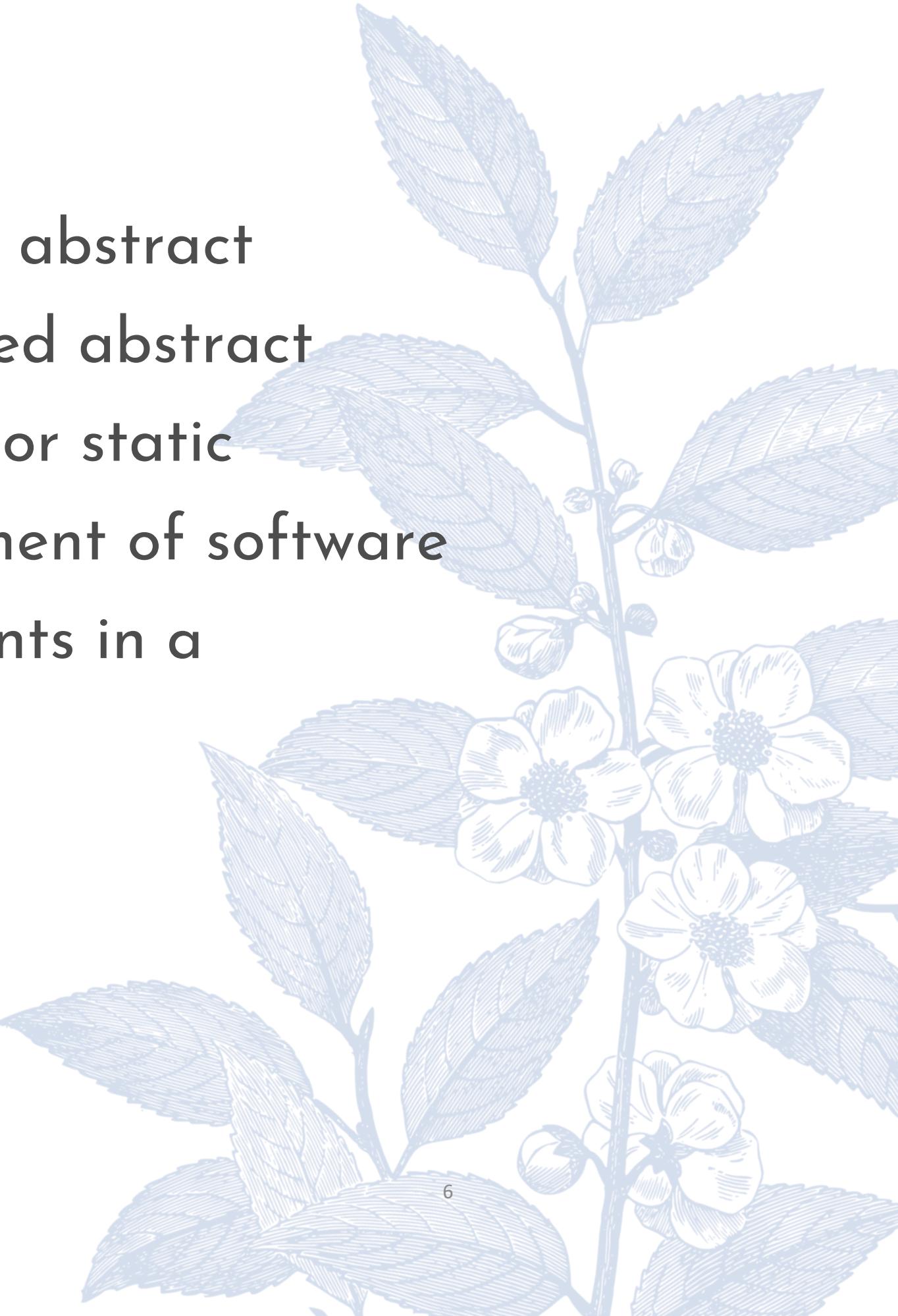
Abstract Classes

- ✓ An abstract class often contains abstract methods with no definitions *(like an interface)*
- ✓ Unlike an interface, the abstract modifier must be applied to each abstract method
- ✓ Also, an abstract class typically contains non-abstract methods with full definitions
- ✓ A class declared as abstract does not have to contain abstract methods -- simply declaring it as abstract makes it so
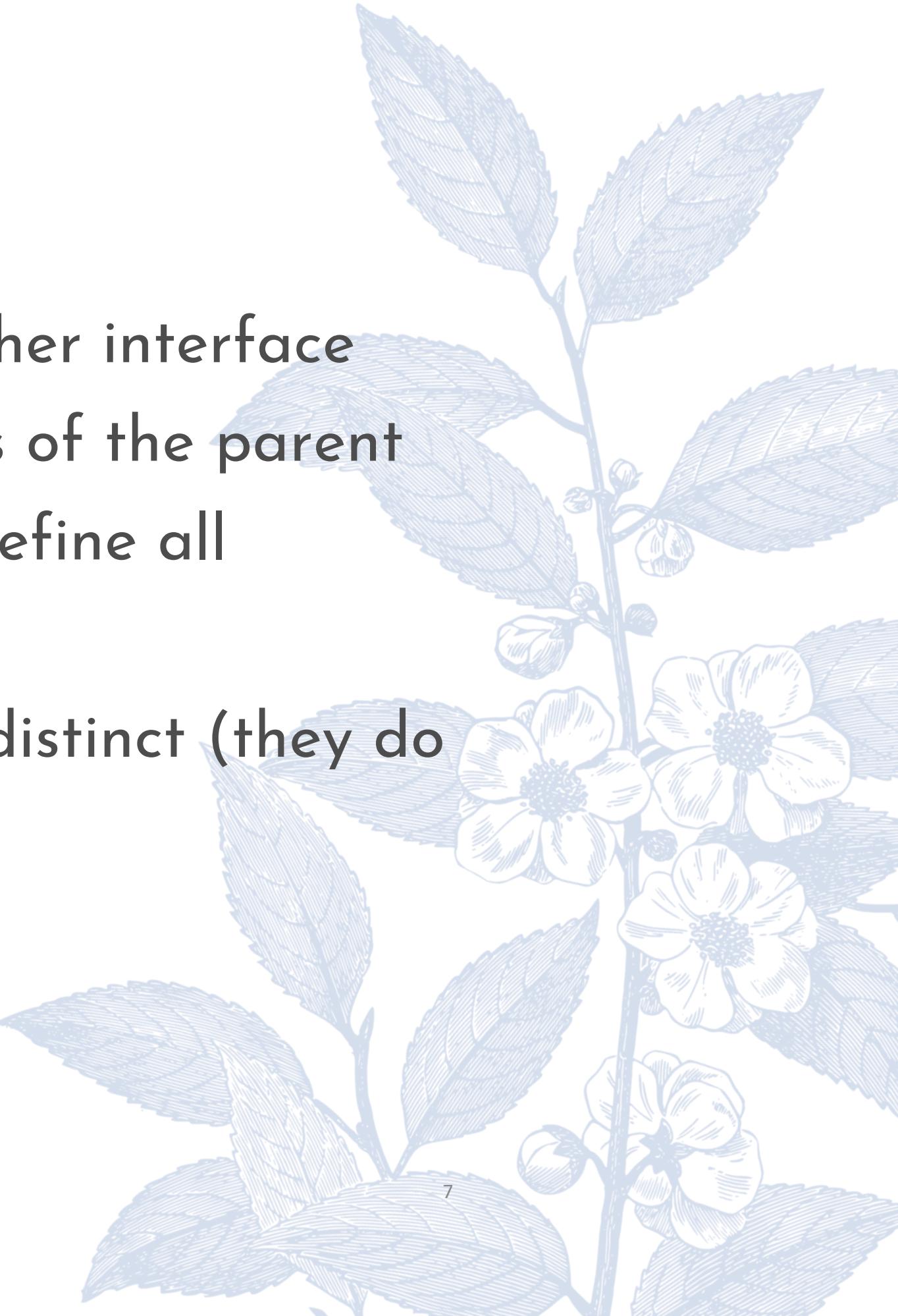
Abstract Classes

✓The child of an abstract class must override the abstract methods of the parent, or it too will be considered abstract

✓An abstract method cannot be defined as final or static

✓The use of abstract classes is an important element of software design – it allows us to establish common elements in a hierarchy that are too general to instantiate
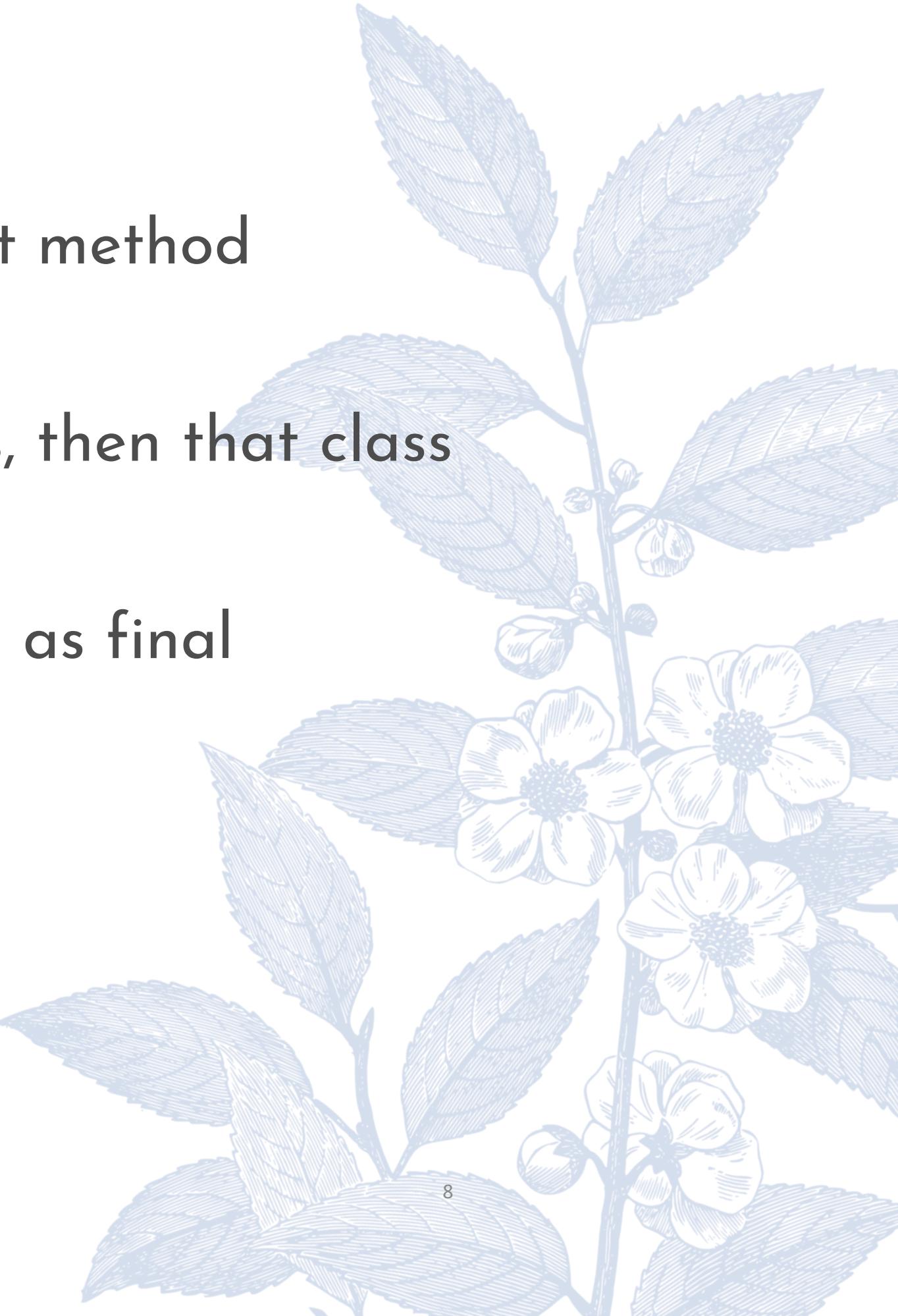
Interface Hierarchies

✓Inheritance can be applied to interfaces

✓That is, one interface can be derived from another interface

✓The child interface inherits all abstract methods of the parent

✓A class implementing the child interface must define all methods from both interfaces

✓Class hierarchies and interface hierarchies are distinct (they do not overlap)
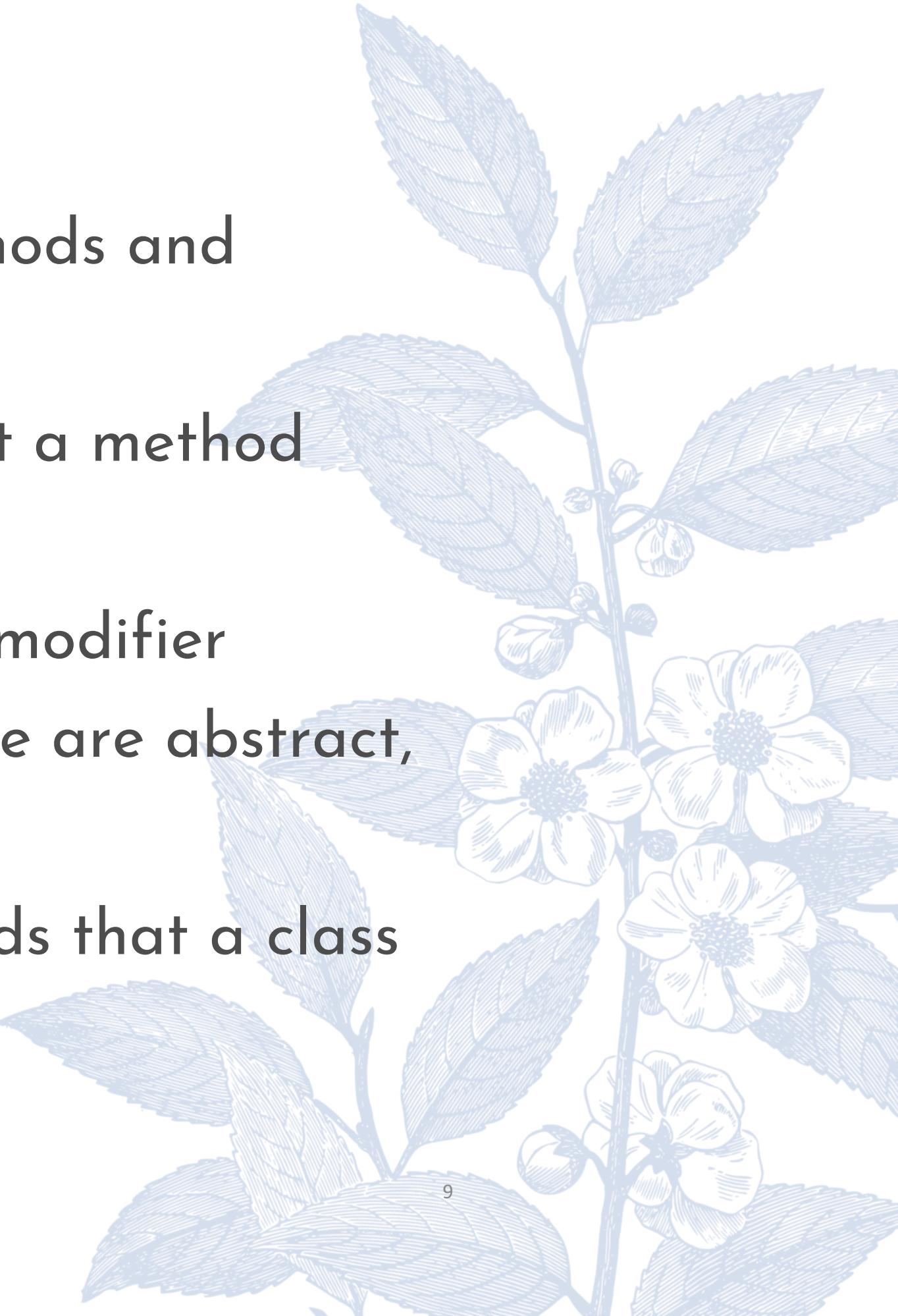
Restricting Inheritance

✓If the final modifier is applied to a method, that method cannot be overridden in any derived classes

✓If the final modifier is applied to an entire class, then that class cannot be used to derive any children at all

✓Therefore, an abstract class cannot be declared as final

Interfaces

- A Java interface is a collection of abstract methods and constants

- An abstract method is a method header without a method body

- An abstract method can be declared using the modifier abstract, but because all methods in an interface are abstract, usually it is left off

- An interface is used to establish a set of methods that a class will implement

Interfaces
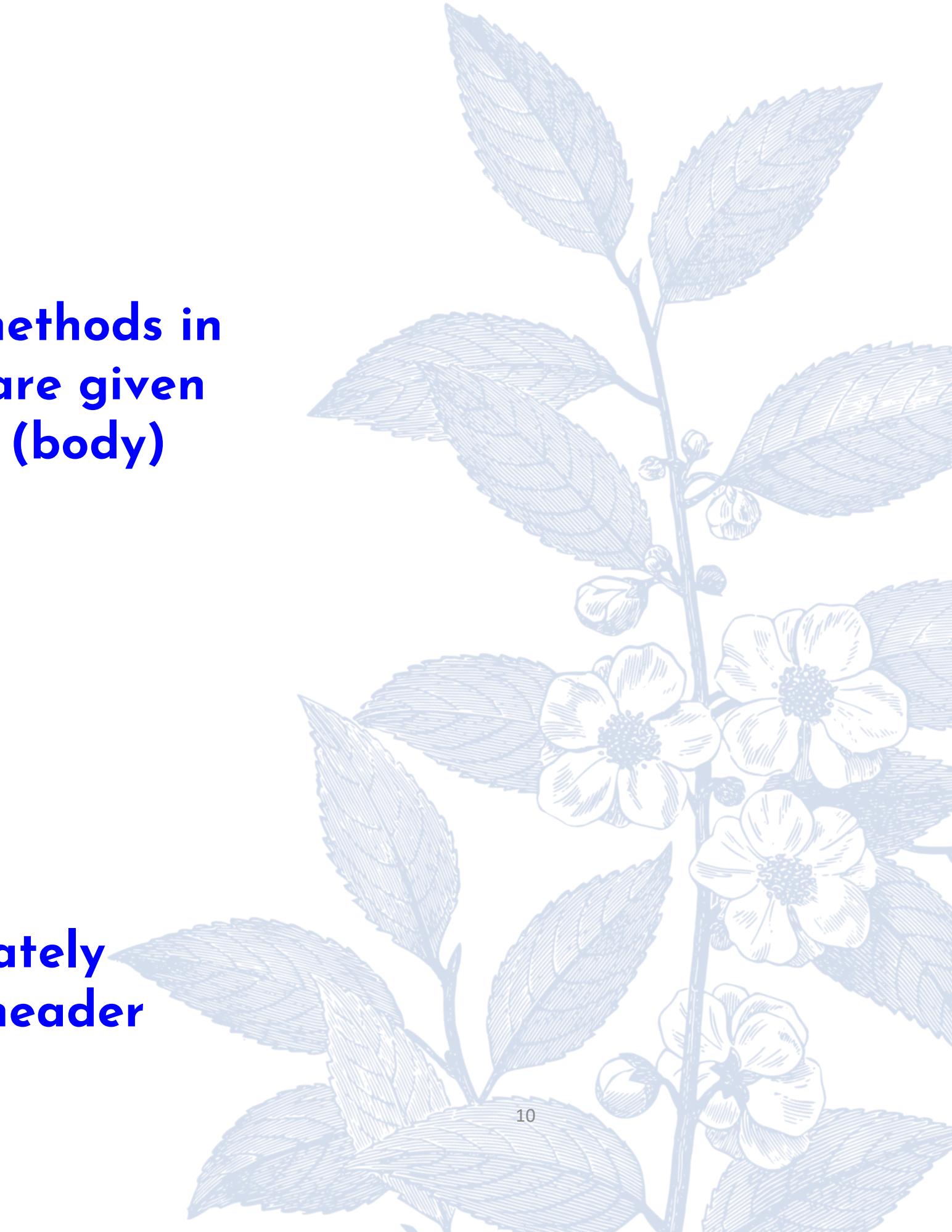
interface **is a reserved word**

None of the methods in an interface are given a definition (body)

```
public interface Doable
{
    public void doThis();
    public int doThat();
    public void doThis2 (float value, char ch);
    public boolean doTheOther (int num);
}
```

**A semicolon immediately follows each method header**

Interfaces

✓An interface cannot be instantiated

✓Methods in an interface have public visibility by default

✓A class formally implements an interface by:

- stating so in the class header
- providing implementations for each abstract method in the interface

✓If a class asserts that it implements an interface, it must define all methods in the interface

Interfaces

```
public class CanDo implements Doable
{
    public void doThis ()
    {
        // whatever
    }


    public void doThat ()
    {
        // whatever
    }

    // etc.
}
```

implements is a reserved word
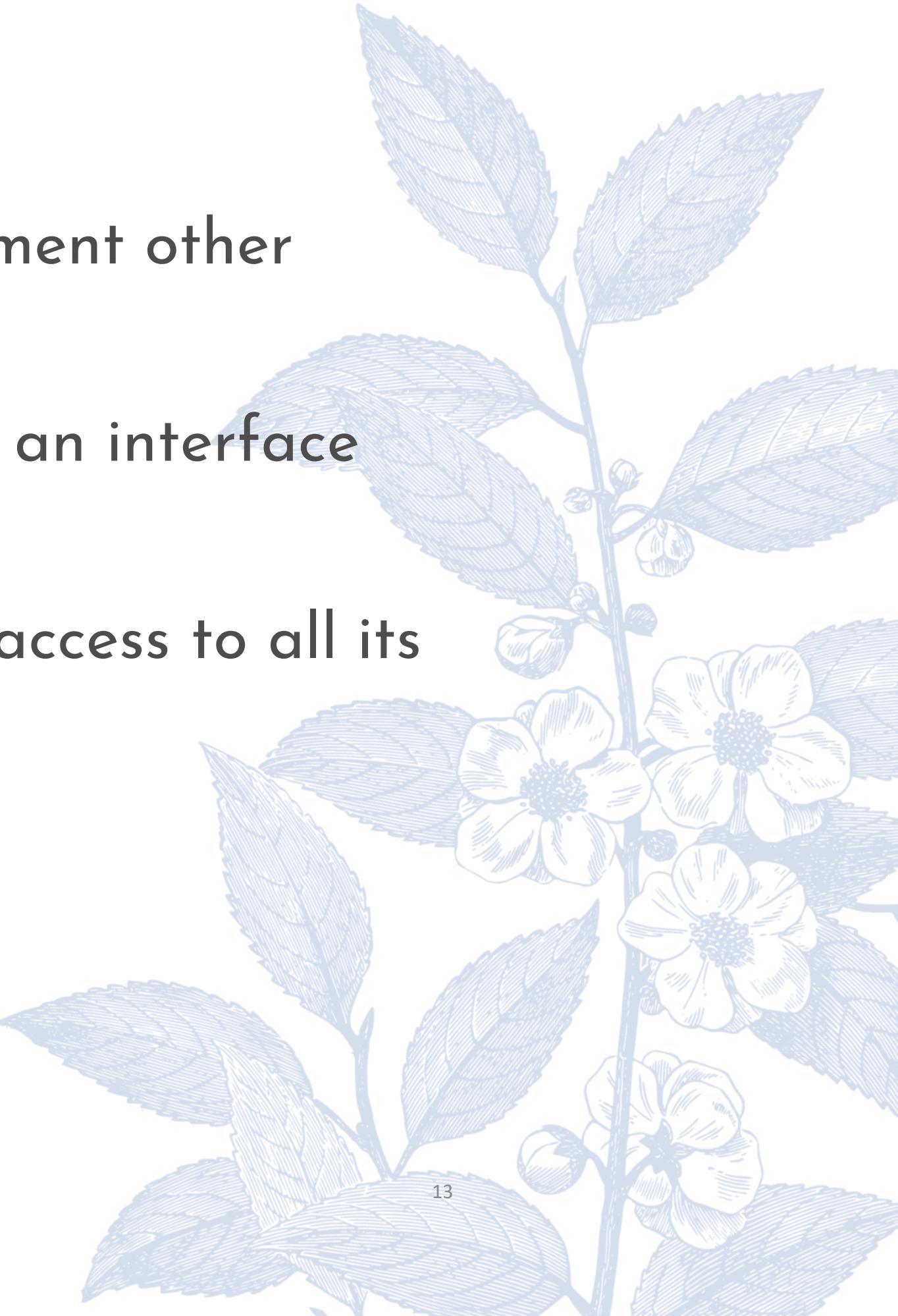
Each method listed in **Doable** is given a definition

Interfaces

✓A class that implements an interface can implement other methods as well

✓In addition to *(or instead of)* abstract methods, an interface can contain constants

✓When a class implements an interface, it gains access to all its constants

✓One class can implement several Interfaces

✓Can use this to "fake" multiple inheritance

Interfaces

✓A class can implement multiple interfaces

✓The class must implement all methods in all interfaces listed in the header
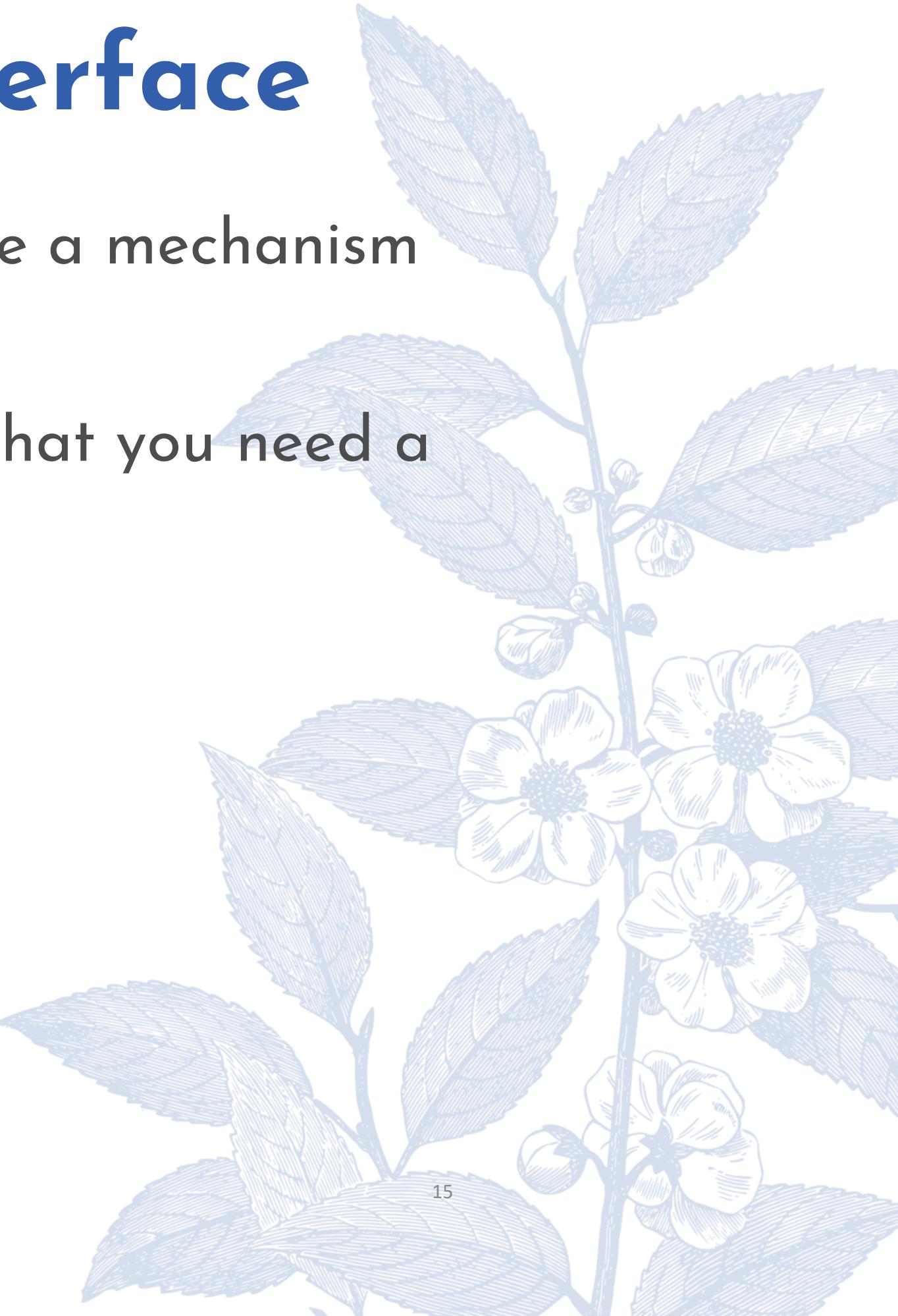
```
class ManyThings implements interface1, interface2
{
    // all methods of both interfaces
}
```

Example:The Comparable Interface

✓Any class can implement Comparable to provide a mechanism for comparing objects of that type

✓Specifically, implementing Comparable means that you need a method CompareTo

```
if (obj1.compareTo(obj2) < 0)
    System.out.println ("obj1 is less than obj2");
```
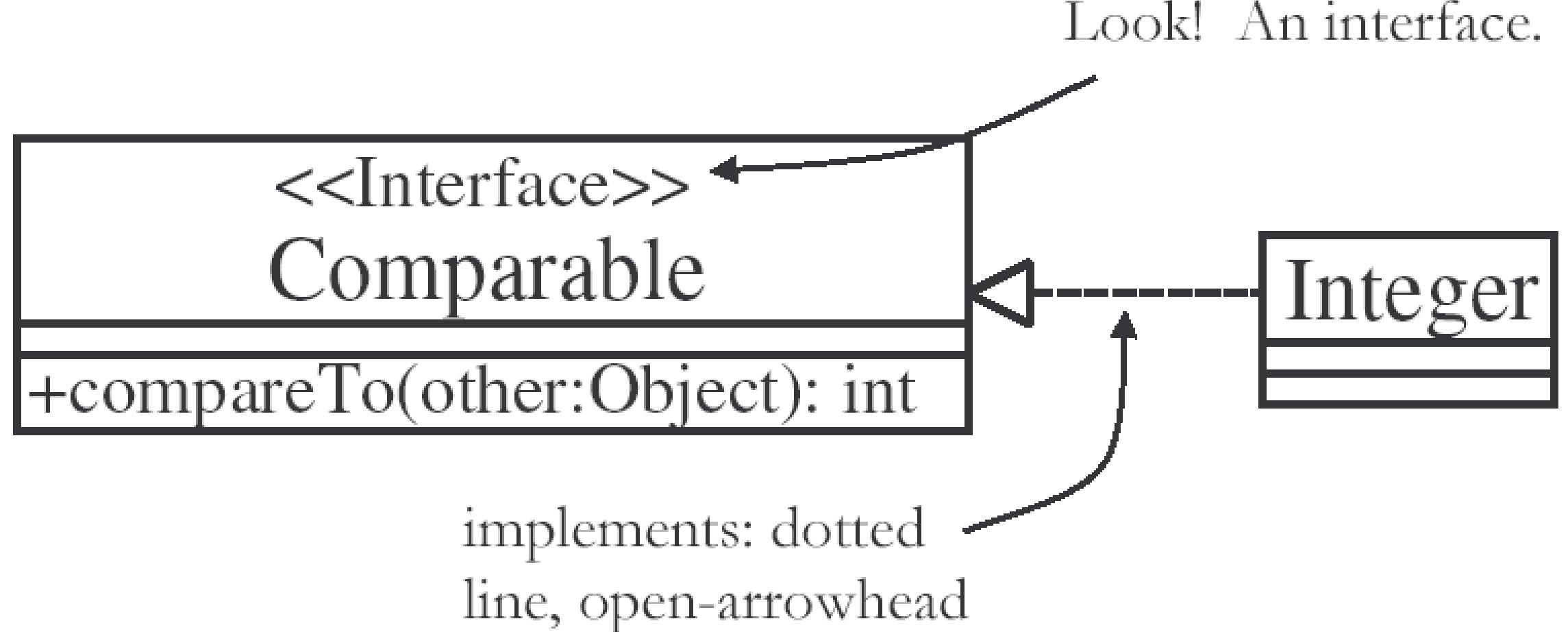
The Comparable Interface

✓It's up to the programmer to determine what makes one object less than another

✓For example, you may define the compareTo method of an Employee class to order employees by name (alphabetically) or by employee number

✓The implementation of the method can be as straightforward or as complex as needed for the situation
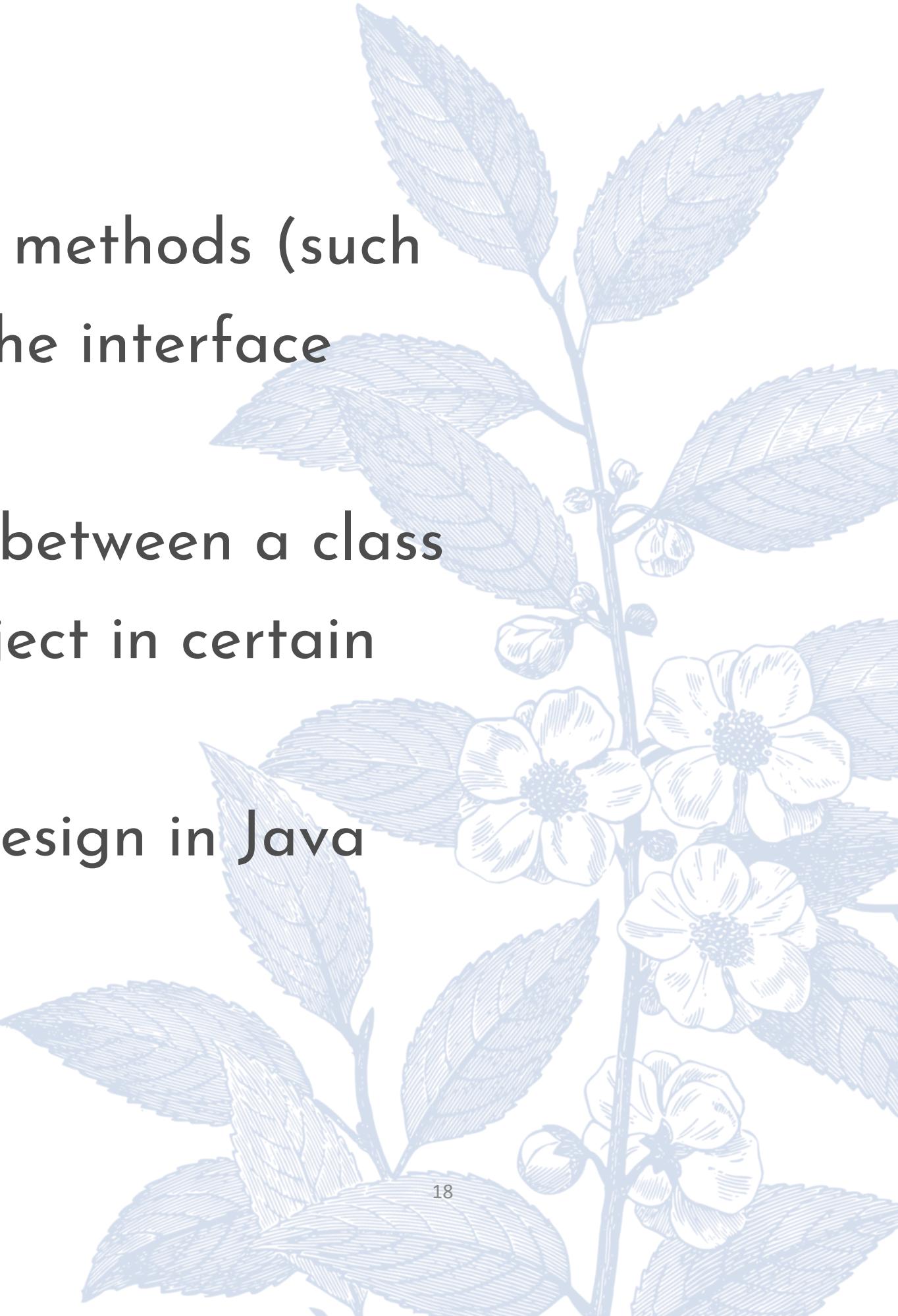
Interfaces in UML

✓Interfaces are easy to spot in class diagrams

Look! An interface.

<<Interface>>
Comparable

+compareTo(other:Object): int

Integer

implements: dotted
line, open-arrowhead

Interfaces

✓You could write a class that implements certain methods (such as compareTo) without formally implementing the interface (Comparable)

✓However, formally establishing the relationship between a class and an interface allows Java to deal with an object in certain ways

✓Interfaces are a key aspect of object-oriented design in Java

Built-in Interfaces

✓The Java standard library includes lots more built-in interfaces

   ▪They are listed in the API with the classes

✓Examples:

   ▪Clonable – implements a clone() method

   ▪Formattable – can be formatted with printf
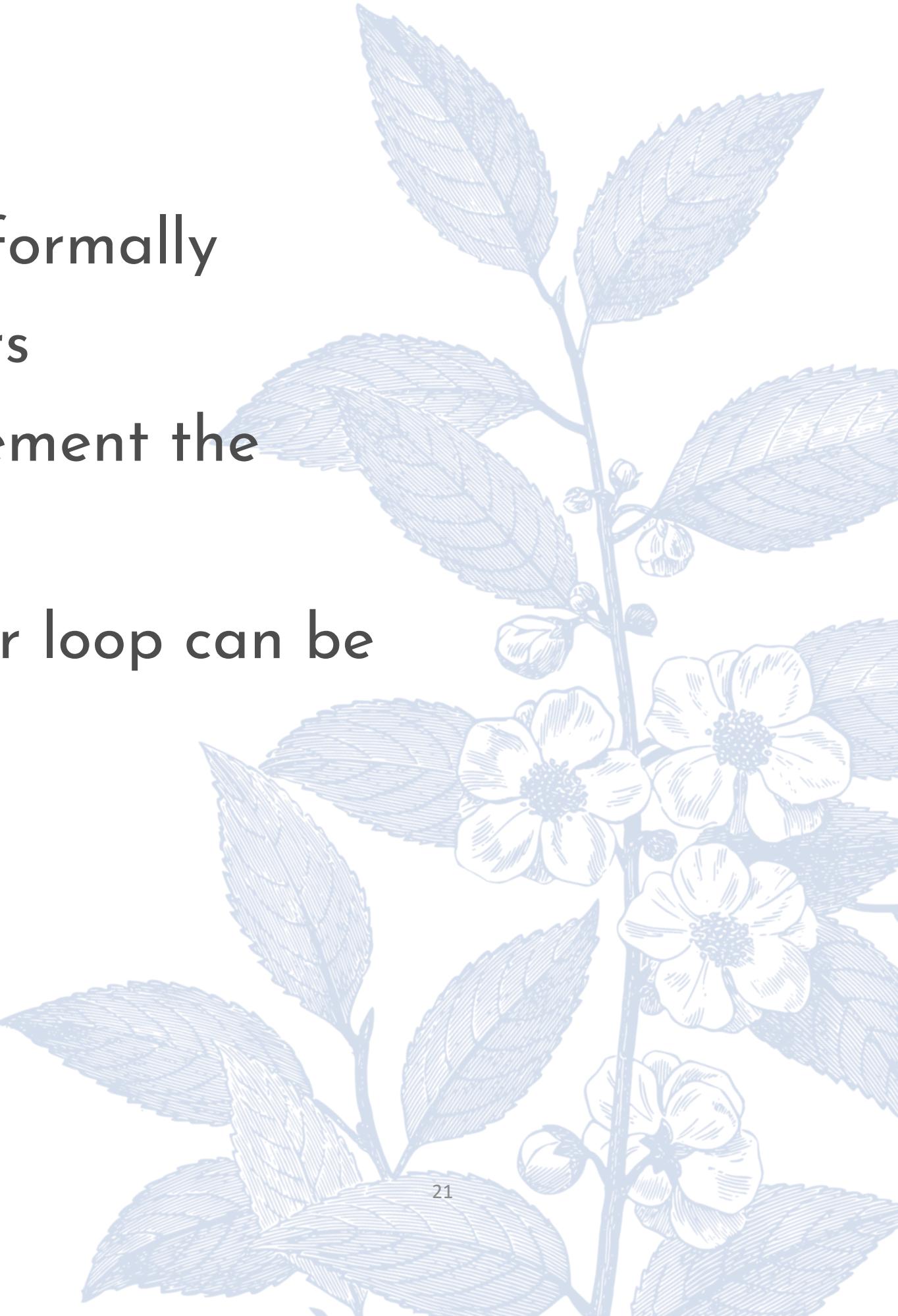
The Iterator Interface

✓ An iterator is an object that provides a means of processing a collection of objects one at a time

✓ An iterator is created formally by implementing the Iterator interface, which contains three methods

- The hasNext method returns a boolean result – true if there are items left to process
- The next method returns the next object in the iteration
- The remove method removes the object most recently returned by the next method
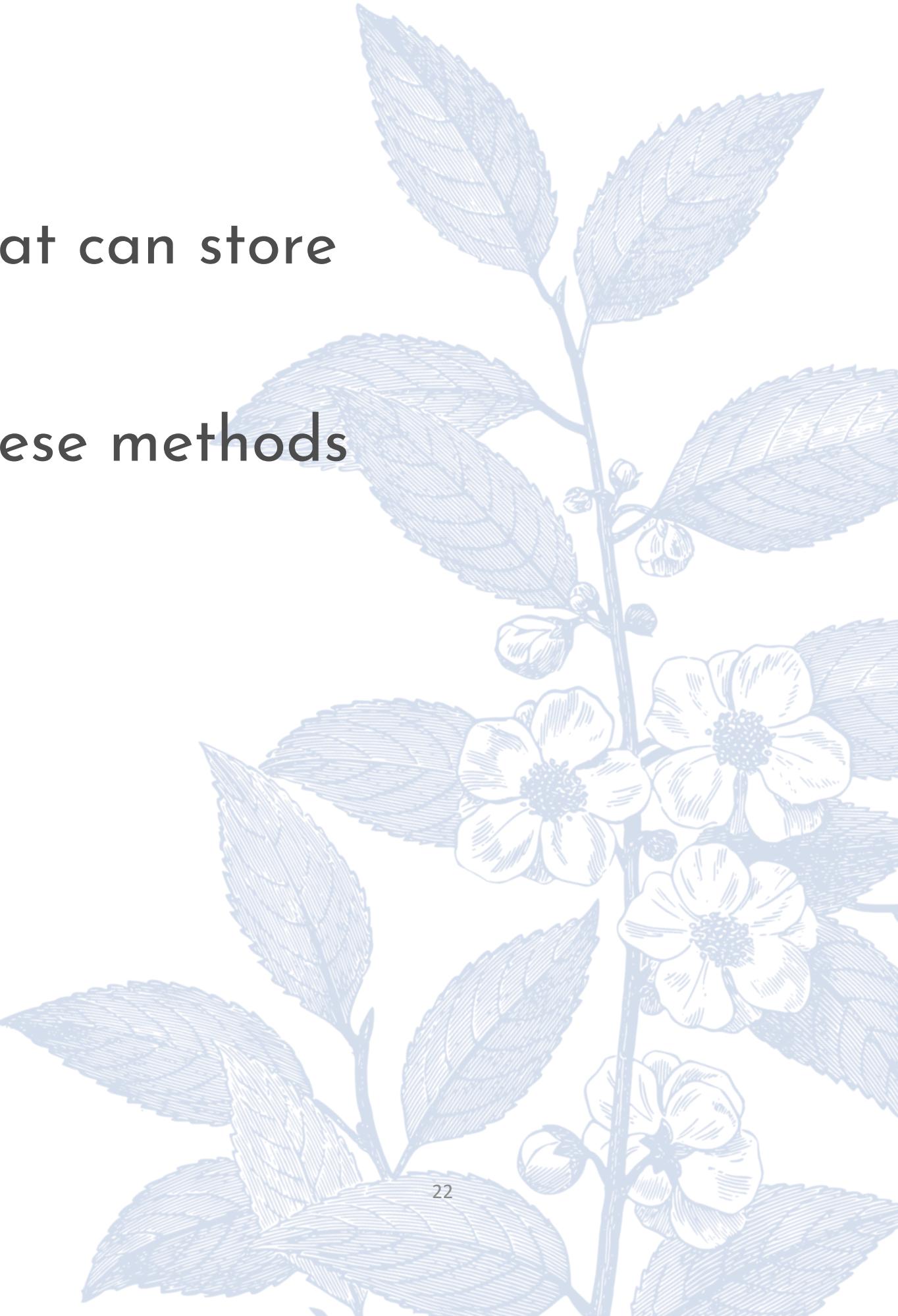
The Iterator Interface

✓By implementing the Iterator interface, a class formally establishes that objects of that type are iterators

✓The programmer must decide how best to implement the iterator functions

✓Once established, the for-each version of the for loop can be used to process the items in the iterator

Collections

✓Collection is a general interface for any type that can store multiple values

✓Any object c that implements Collections has these methods

- ▪ c.add(e)
- ▪ c.remove(e)
- ▪ c.size()

Collection Sub-Interfaces

✓Interfaces that are derived from Collection

Set: unordered, can't add the same object twice
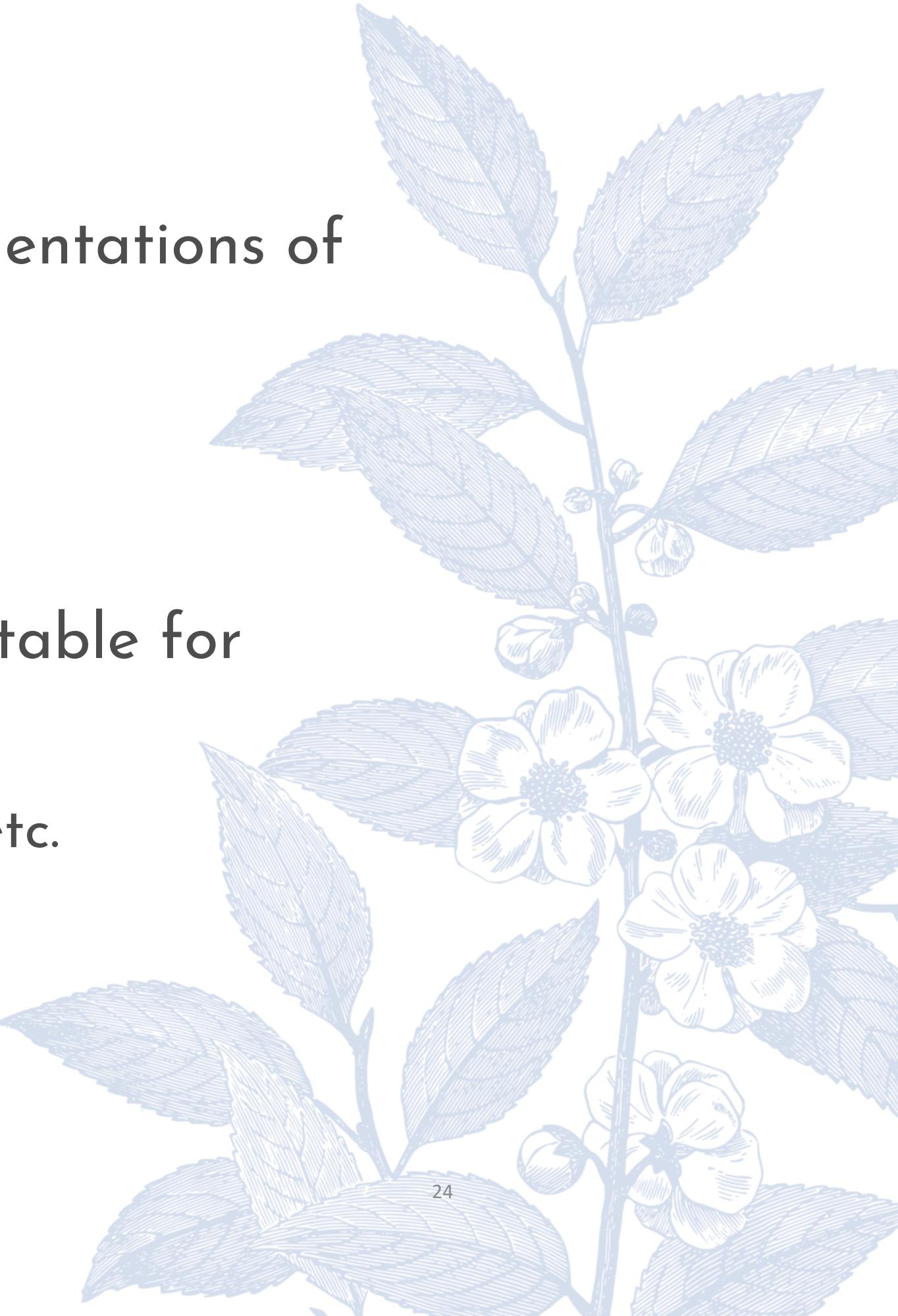
List: ordered, adds new methods

- get(i): get the ith element
- set(i,e): set the ith element to e

Collection Implementations

✓Also in the standard library: many good implementations of these interfaces

✓List: ArrayList, Stack, LinkedList

✓Sets: HashSet, TreeSet

✓Each implementation has some differences... suitable for particular problems

▪e.g. additional methods, different type restrictions, etc.

Interfaces vs. Abstract Classes

✓Similarities

- neither can be instantiated
- both can be used as the starting point for a class

✓Differences

- A class can contain implementations of methods
- A class can implement many interfaces, but only one class

Comparison

In order of "abstractness":

✓Interface

- no method implementations
- can't be instantiated

✓Abstract class

- some method implementations
- can't be instantiated

✓Non-abstract class

- all methods implemented
- can be instantiated

Polymorphism
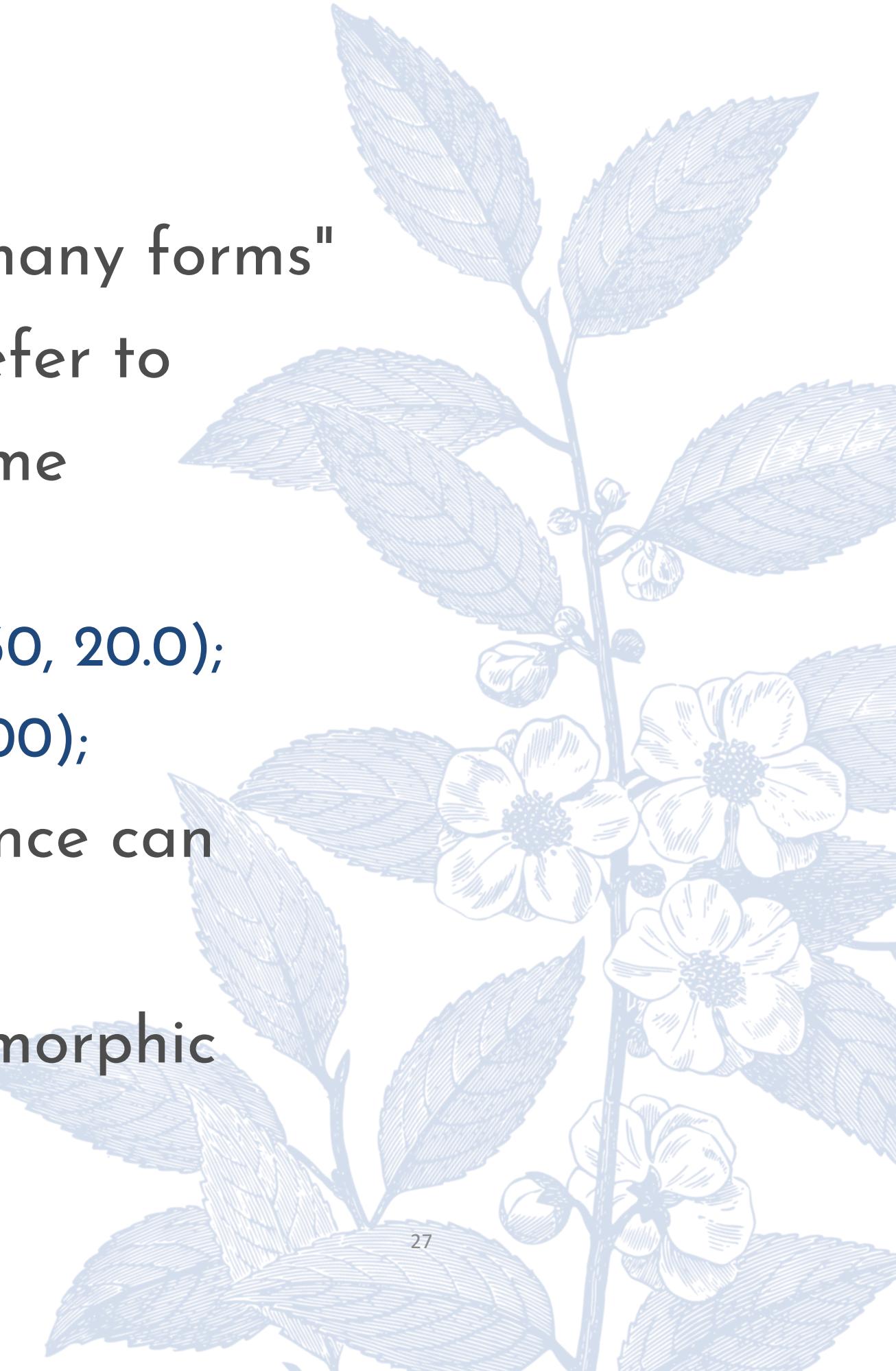
✓The term polymorphism literally means "having many forms"

✓A polymorphic reference is a variable that can refer to different types of objects at different points in time

    Employee emp = null;

    emp = new HourlyEmployee("1001", "Morgan, Harry", 30, 20.0);
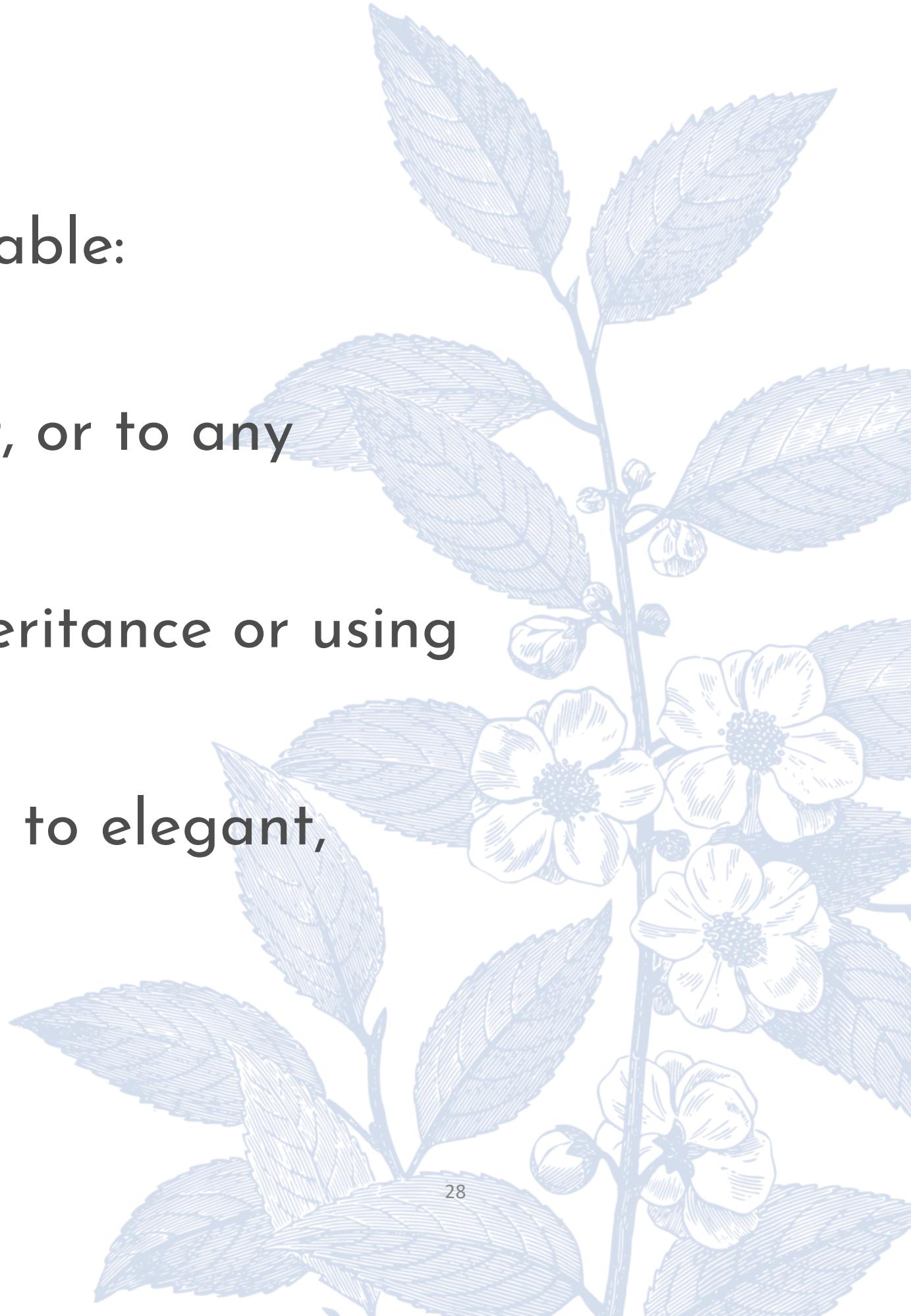
    emp = new SalariedEmployee("2001", "Lin, Sally", 52000);

✓The method called through a polymorphic reference can change from one invocation to the next

✓All object references in Java are potentially polymorphic
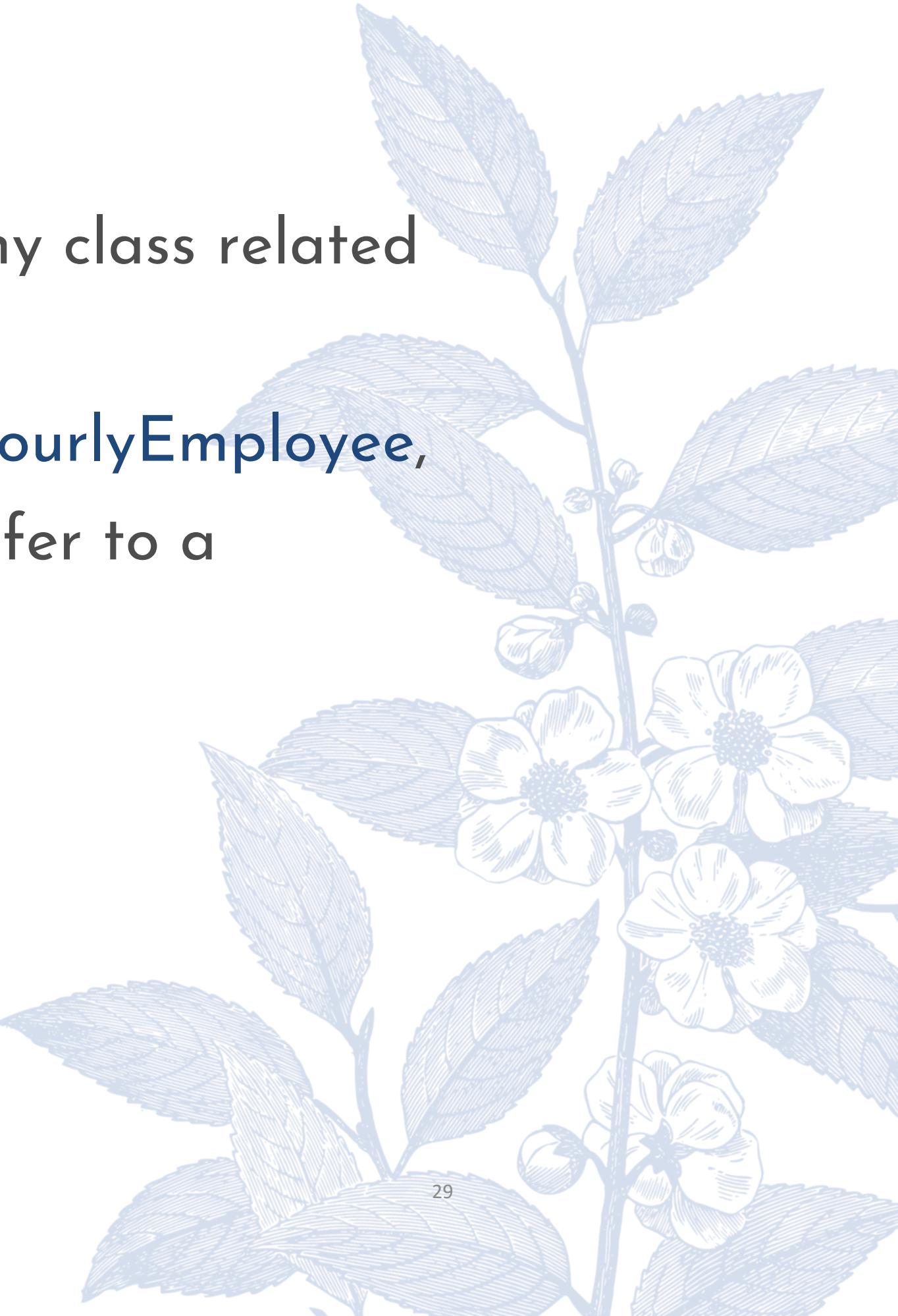
Polymorphism

✓Suppose we create the following reference variable:

   Employee emp

✓This reference can point to an **Employee** object, or to any object of any compatible type

✓This compatibility can be established using inheritance or using interfaces

✓Careful use of polymorphic references can lead to elegant, robust software designs

References and Inheritance

✓An object reference can refer to an object of any class related to it by inheritance

✓For example, if Employee is the superclass of HourlyEmployee, then an Employee reference could be used to refer to a HourlyEmployee object
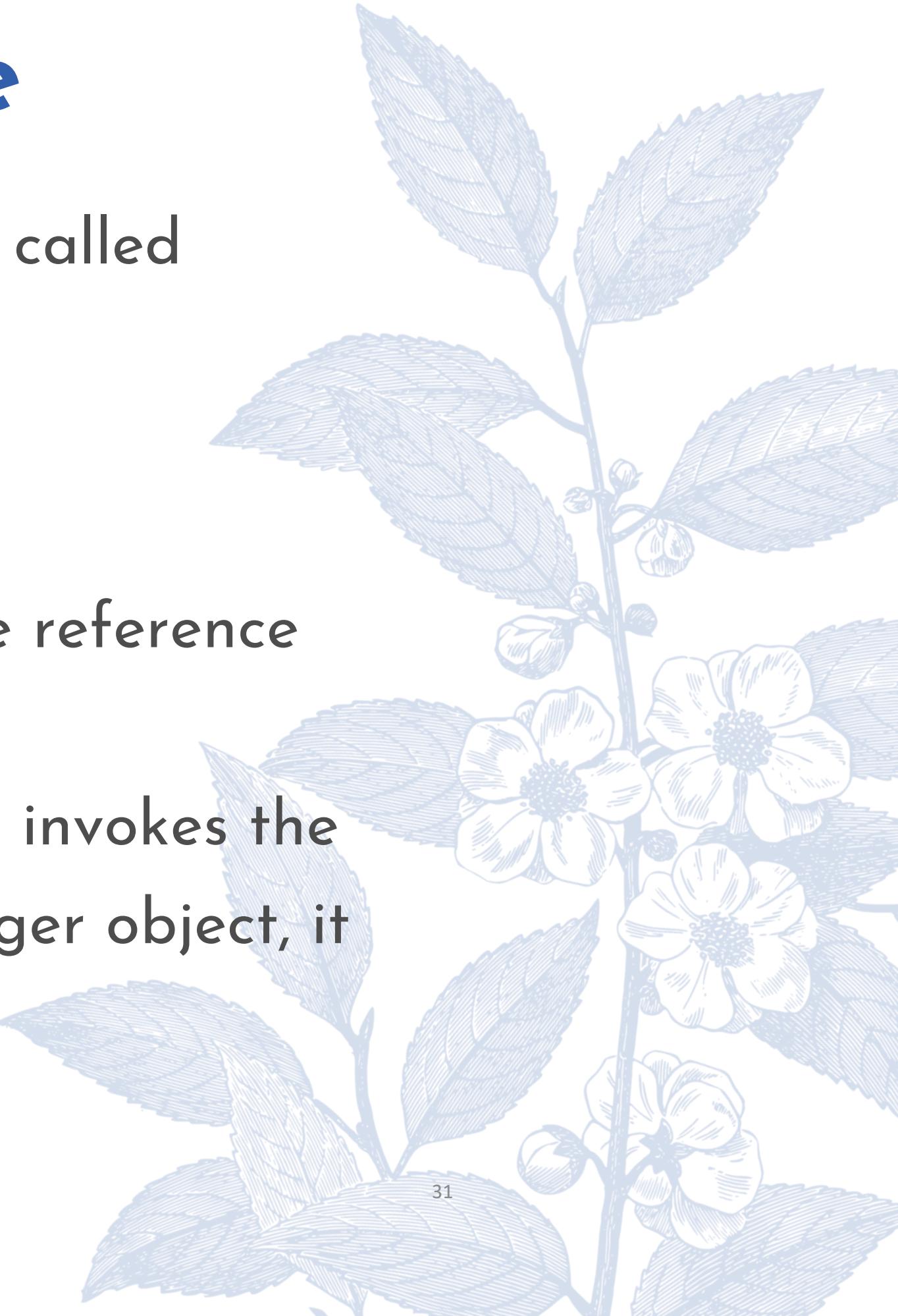
References and Inheritance

- ✓These type compatibility rules are just an extension of the is-a relationship established by inheritance
- ✓Assigning a HourlyEmployee object to an Employee reference is fine because HourlyEmployee is-a Employee
- ✓Assigning a child object to a parent reference can be performed by simple assignment
- ✓Assigning a parent object to a child reference can also be done, but it must be done with a cast
- ✓After all, HourlyEmployee is an Employee but not all Employees are HourlyEmployees

Polymorphism via Inheritance

✓Now suppose the Employee class has a method called weeklyPay, and the Manager overrides it

✓What method is invoked by the following?

   emp. weeklyPay();

✓The type of the object being referenced, not the reference type, determines which method is invoked

✓If the employee refers to an Employee object, it invokes the Employee version of pay; if it refers to a Manager object, it invokes that version

Polymorphism via Inheritance
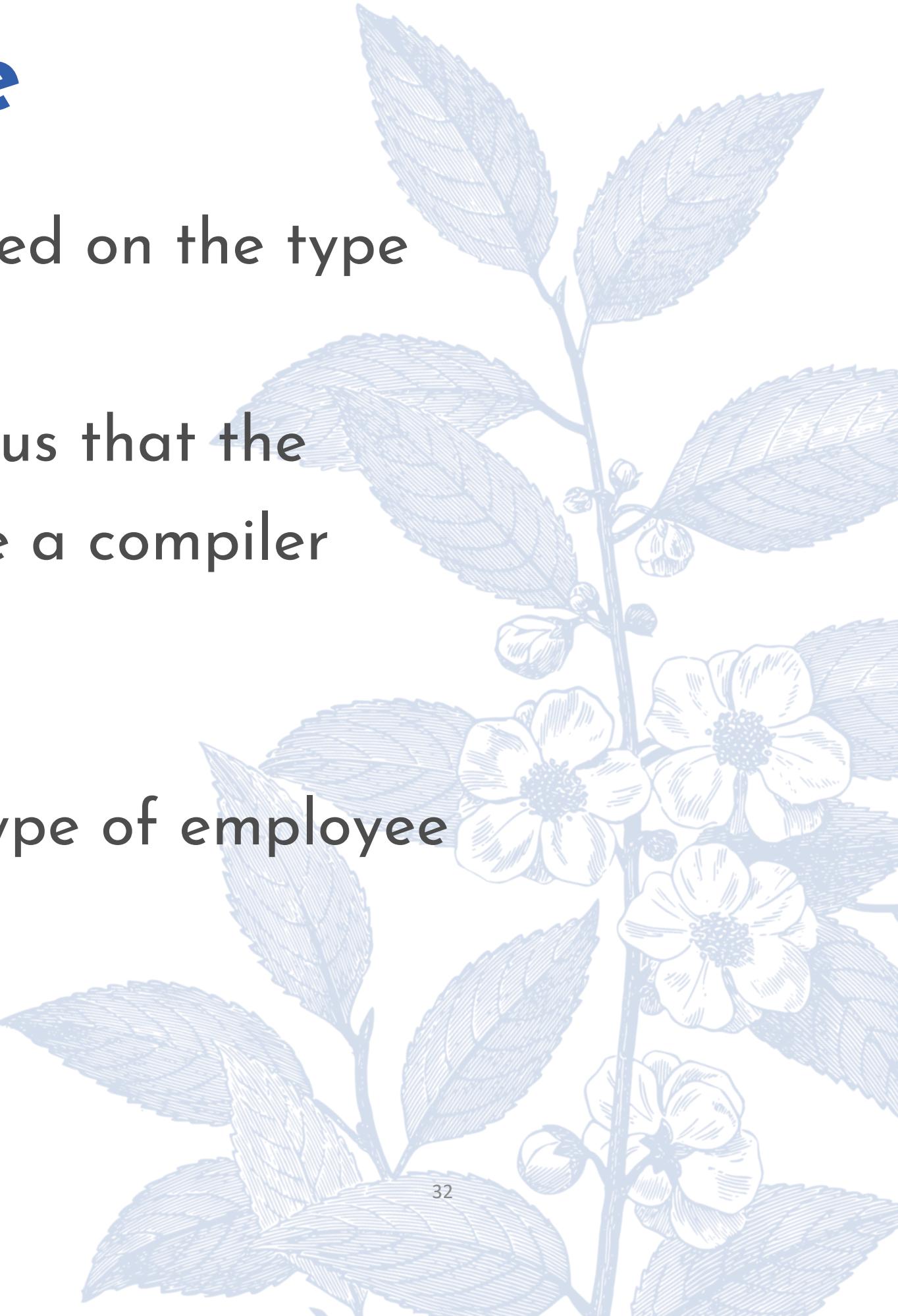
✓Note that the compiler restricts invocations based on the type of the reference

✓So if the Manager had a method called getBonus that the Employee didn't have, the following would cause a compiler error:

emp.getBonus(); // compiler error

✓Remember, the compiler doesn't "know" which type of employee is being referenced
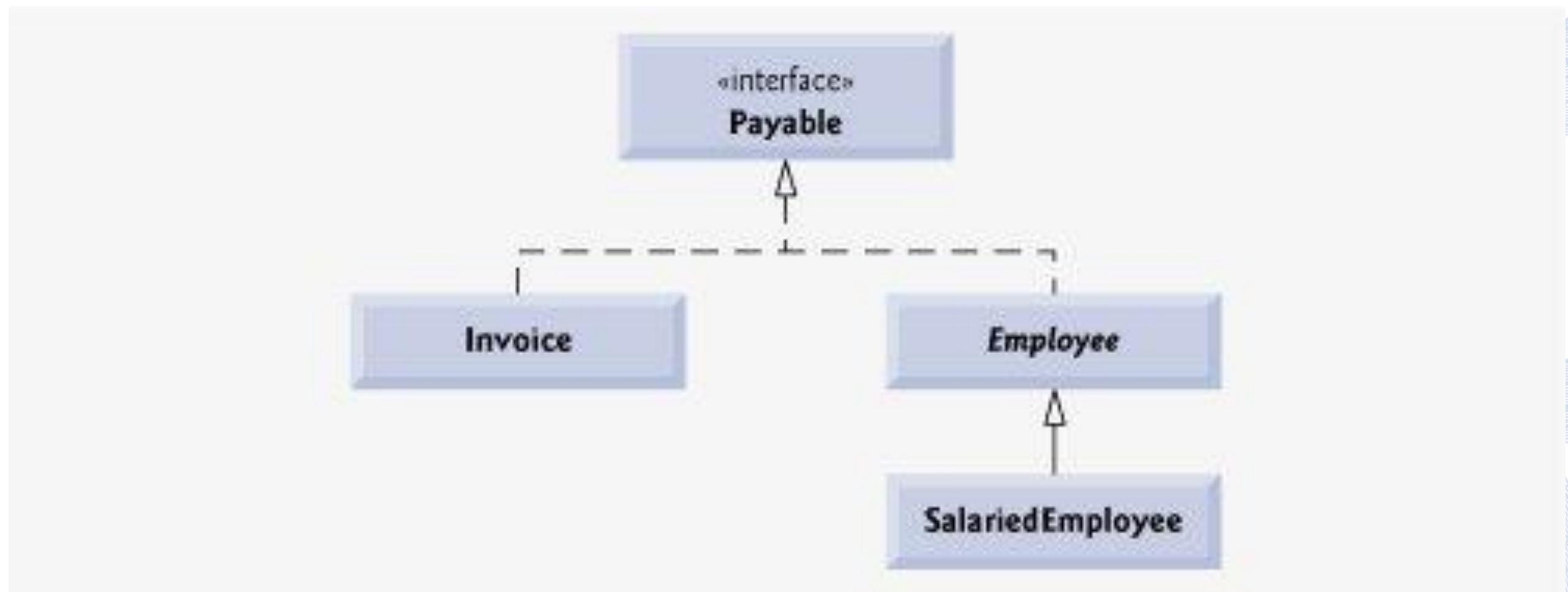
✓A cast can be used to allow the call:

((Manager)emp).getBonus();

Polymorphism via Interfaces

✓Interfaces can be used to set up polymorphic references as well

Exceptions

✓the purpose of exceptions

✓exception messages

✓the try-catch statement

✓propagating exceptions

✓the exception class hierarchy
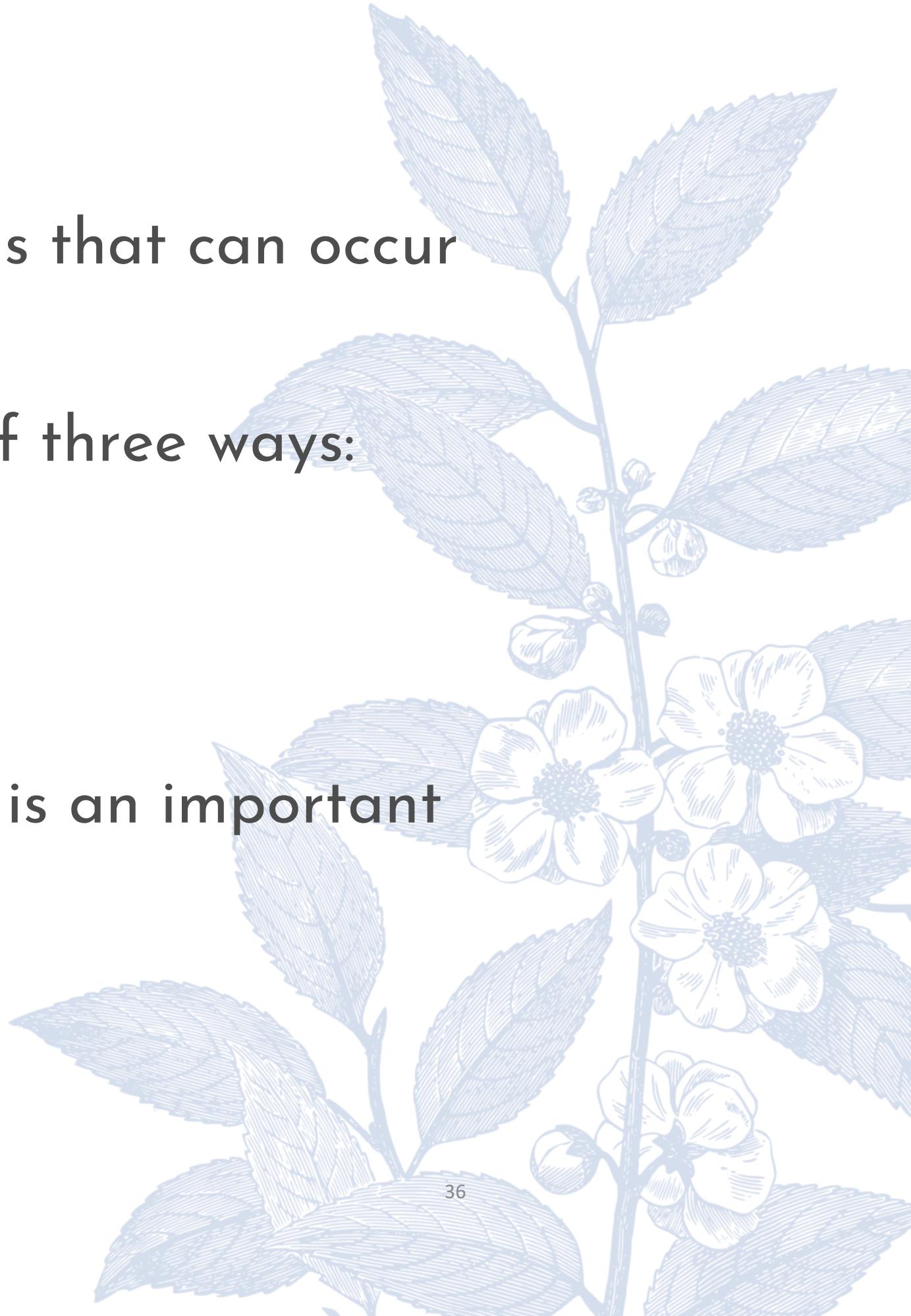
Exceptions

✓An exception is an object that describes an unusual or erroneous situation

✓Exceptions are thrown by a program, and may be caught and handled by another part of the program

✓A program can be separated into a normal execution flow and an exception execution flow

✓An error is also represented as an object in Java, but usually represents a unrecoverable situation and should not be caught
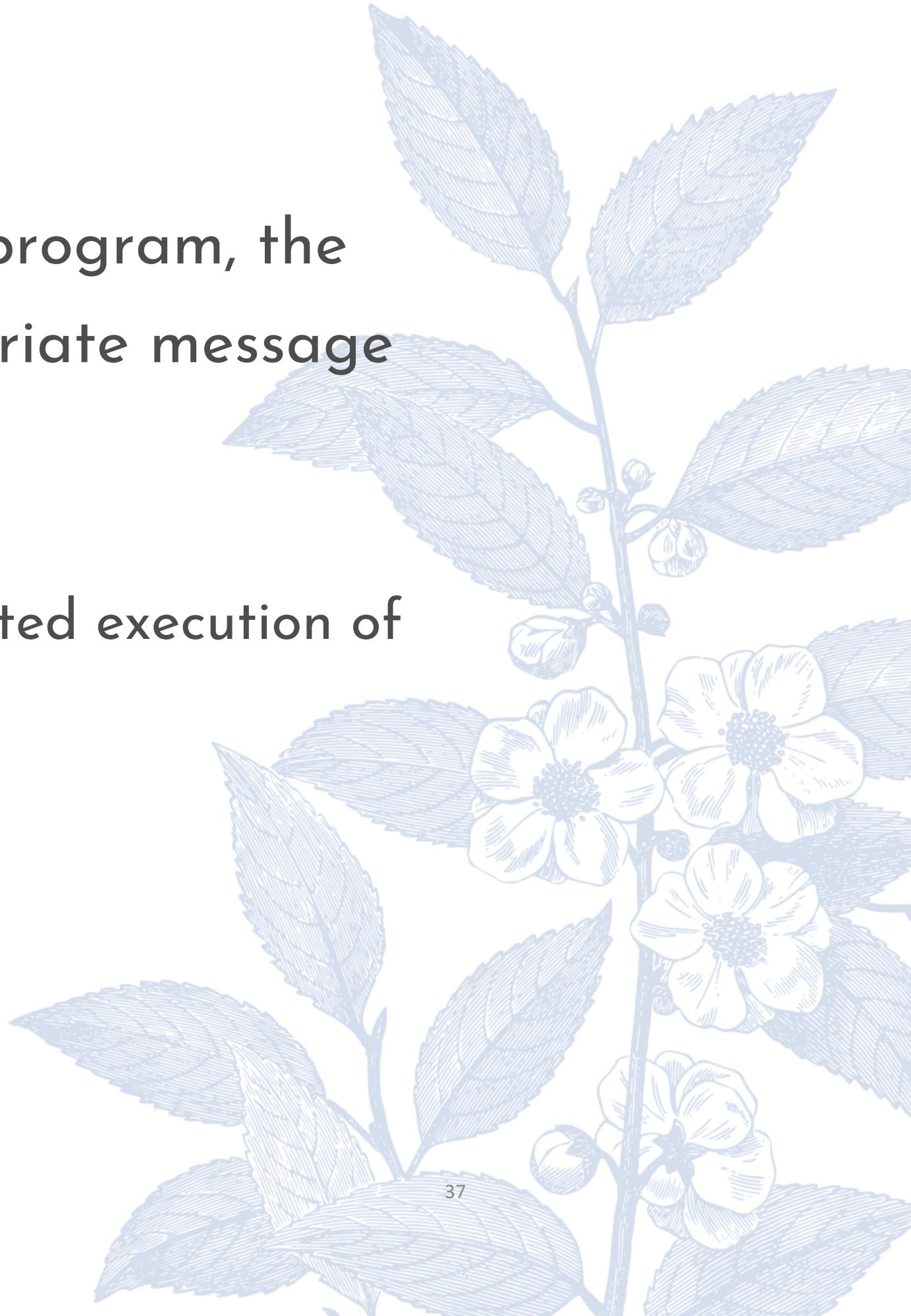
Exceptions

✓The Java API has a predefined set of exceptions that can occur during execution

✓A program can deal with an exception in one of three ways:
- ignore it
- handle it where it occurs
- handle it an another place in the program

✓The manner in which an exception is processed is an important design consideration
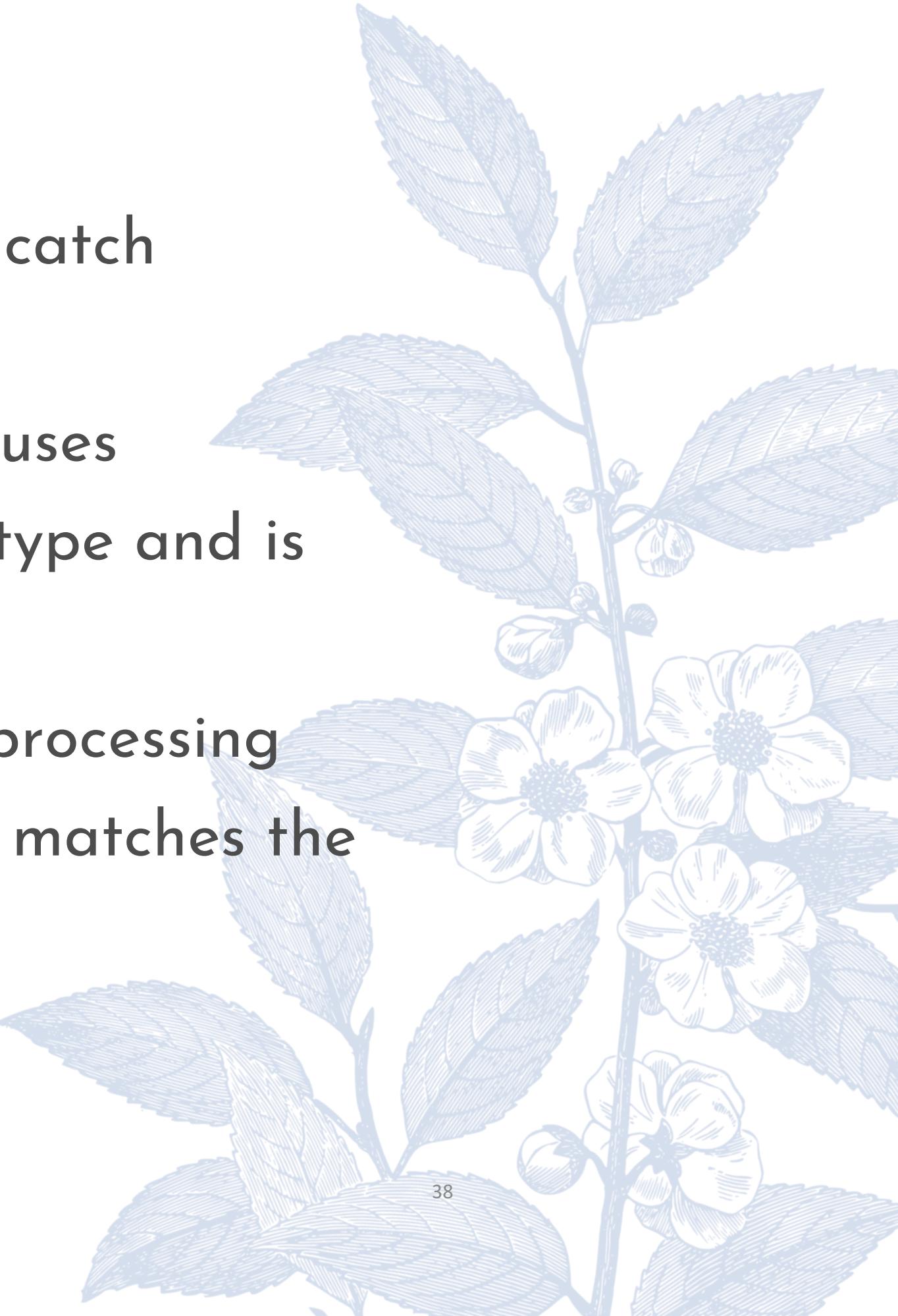
Exception Handling

✓If an exception is ignored (not caught) by the program, the program will terminate and produce an appropriate message

✓The message includes a call stack trace that:

- indicates the line on which the exception occurred
- shows the method call trail that lead to the attempted execution of the offending line

✓See Zero.java
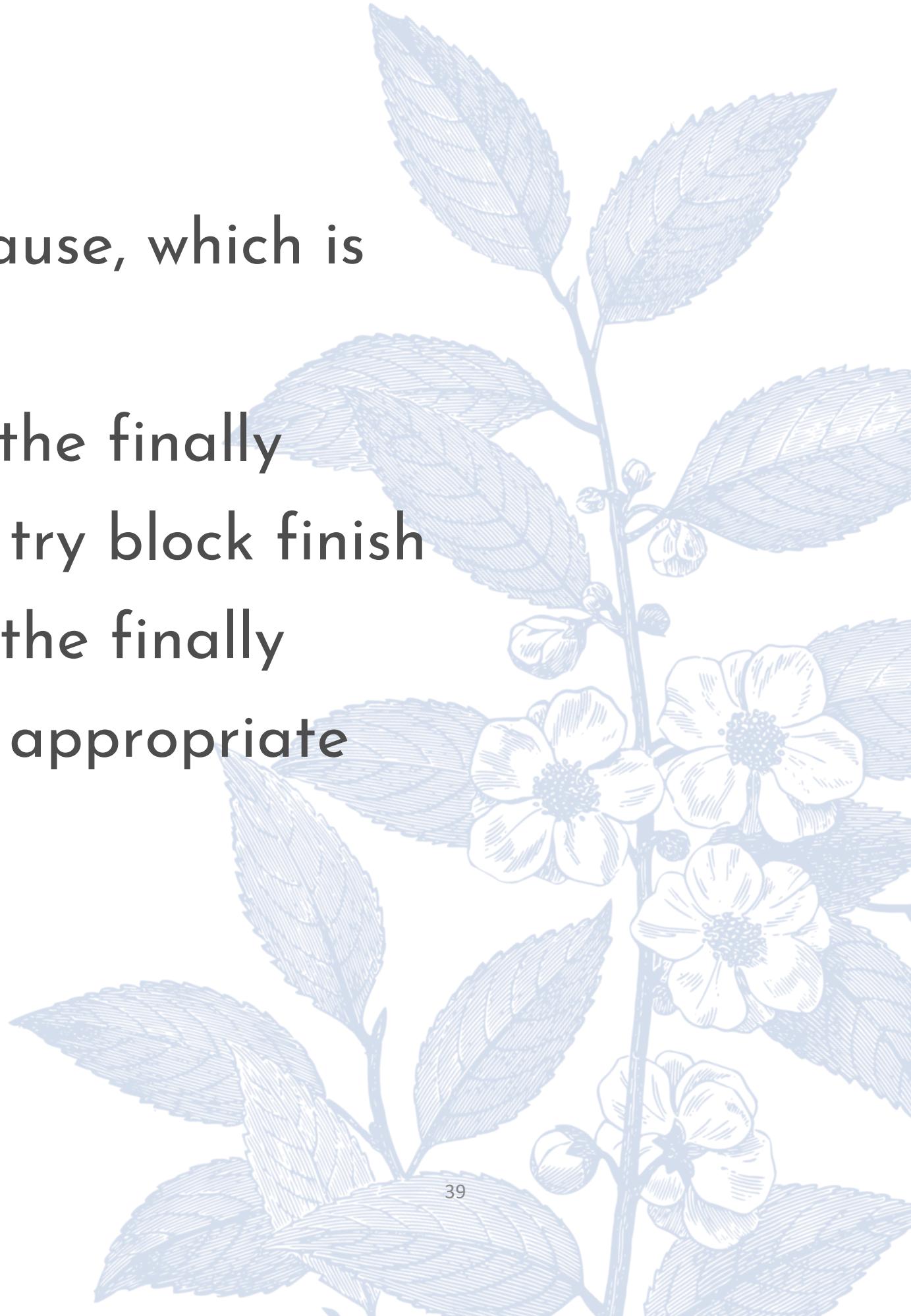
The try Statement

✓To handle an exception in a program, use a try-catch statement

✓A try block is followed by one or more catch clauses

✓Each catch clause has an associated exception type and is called an exception handler

✓When an exception occurs within the try block, processing immediately jumps to the first catch clause that matches the exception type
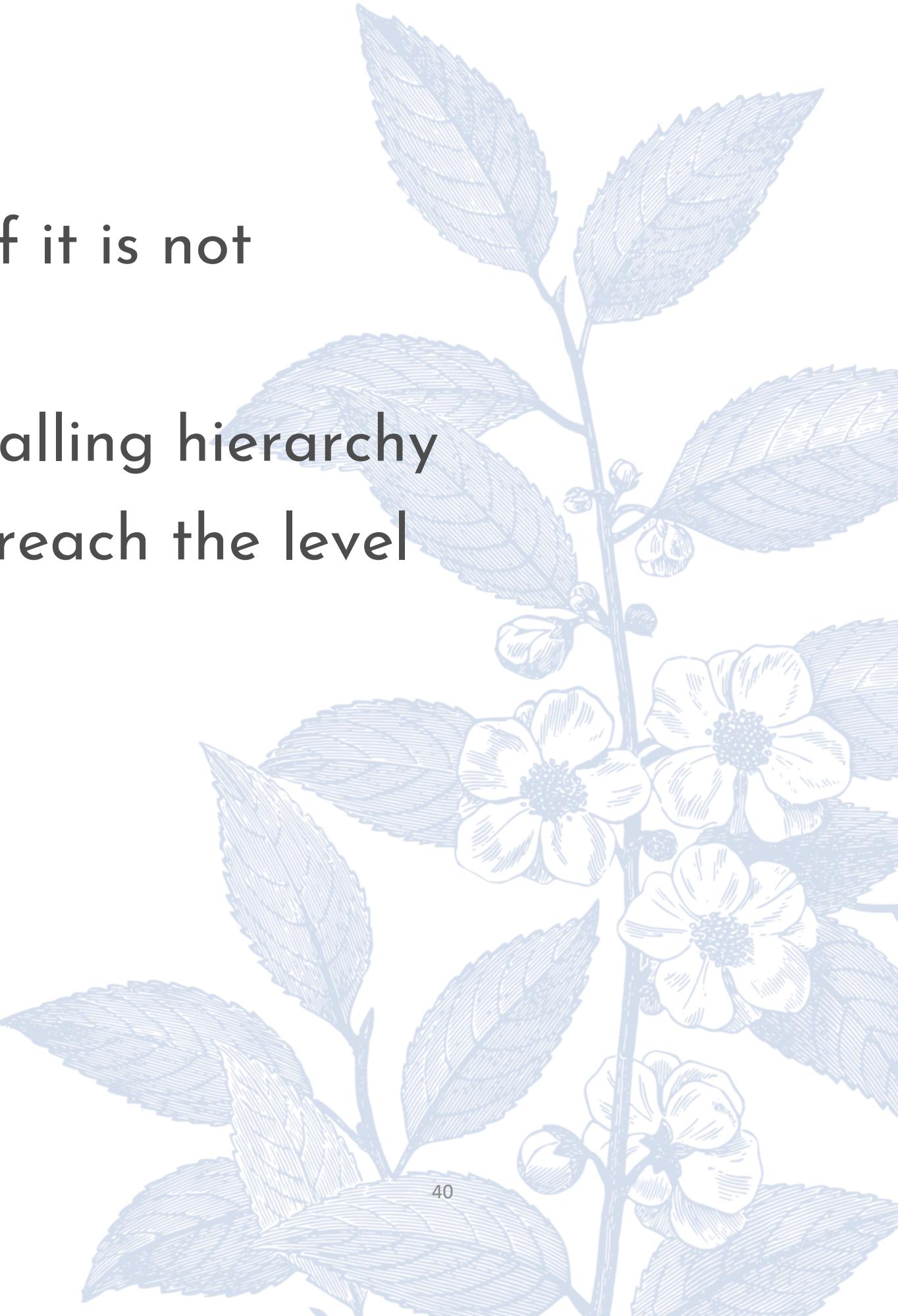
The finally Clause

✓A try statement can have an optional finally clause, which is always executed

✓If no exception is generated, the statements in the finally clause are executed after the statements in the try block finish

✓If an exception is generated, the statements in the finally clause are executed after the statements in the appropriate catch clause finish
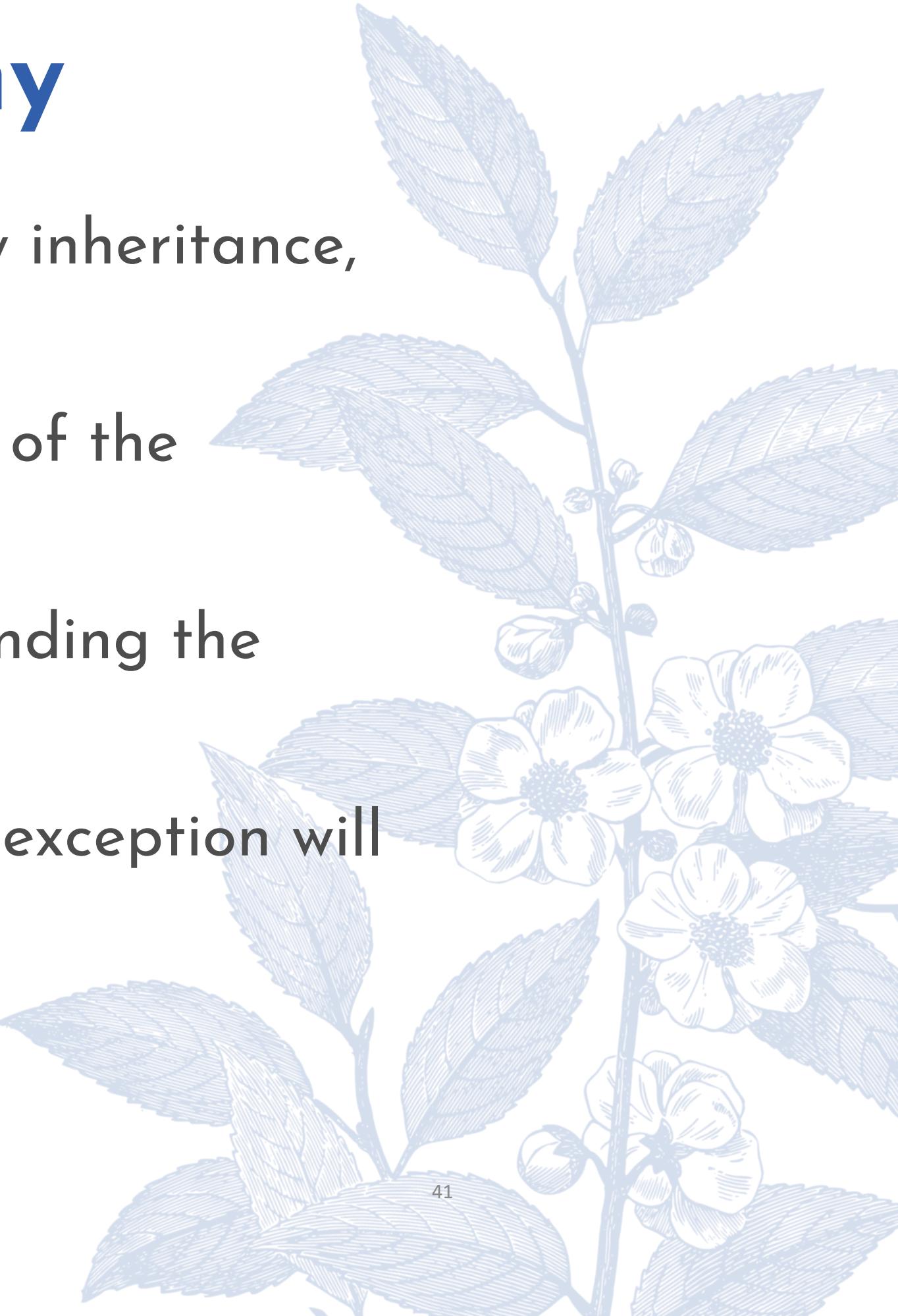
Exception Propagation

✓An exception can be handled at a higher level if it is not appropriate to handle it where it occurs

✓Exceptions propagate up through the method calling hierarchy until they are caught and handled or until they reach the level of the main method
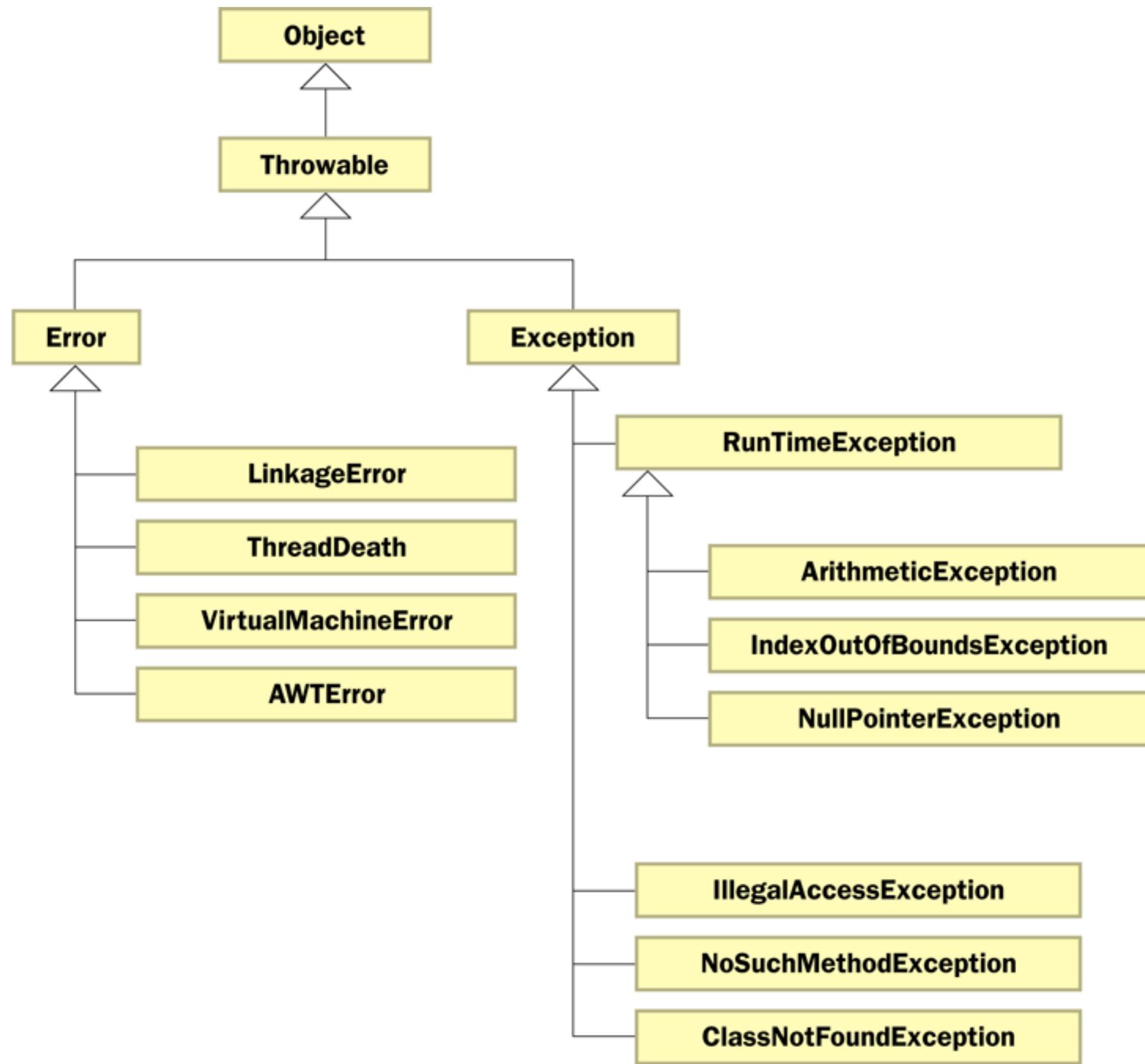
The Exception Class Hierarchy

✓Exception classes in the Java API are related by inheritance, forming an exception class hierarchy

✓All error and exception classes are descendents of the Throwable class

✓A programmer can define an exception by extending the Exception class or one of its descendants

✓The parent class used depends on how the new exception will be used

The Exception Class Hierarchy

Checked Exceptions
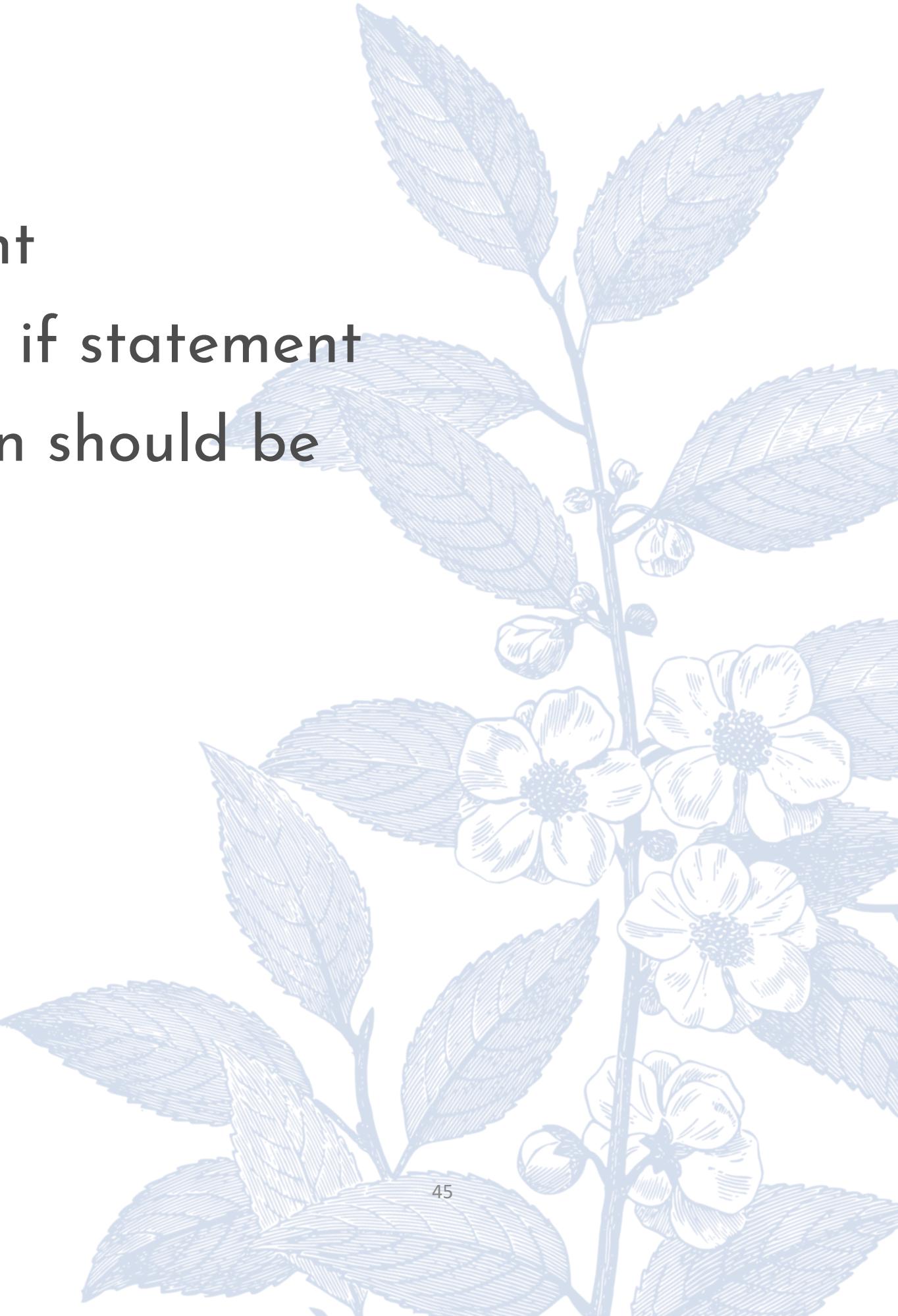
✓An exception is either checked or unchecked

✓A checked exception must either be caught or must be listed in the throws clause of any method that may throw or propagate it

✓A throws clause is appended to the method header

✓The compiler will issue an error if a checked exception is not caught or listed in a throws clause

Unchecked Exceptions

✓An unchecked exception does not require explicit handling, though it could be processed that way

✓The only unchecked exceptions in Java are objects of type RuntimeException or any of its descendants

✓Errors are similar to RuntimeException and its descendants in that:
  ▪Errors should not be caught
  ▪Errors do not require a throws clause

44
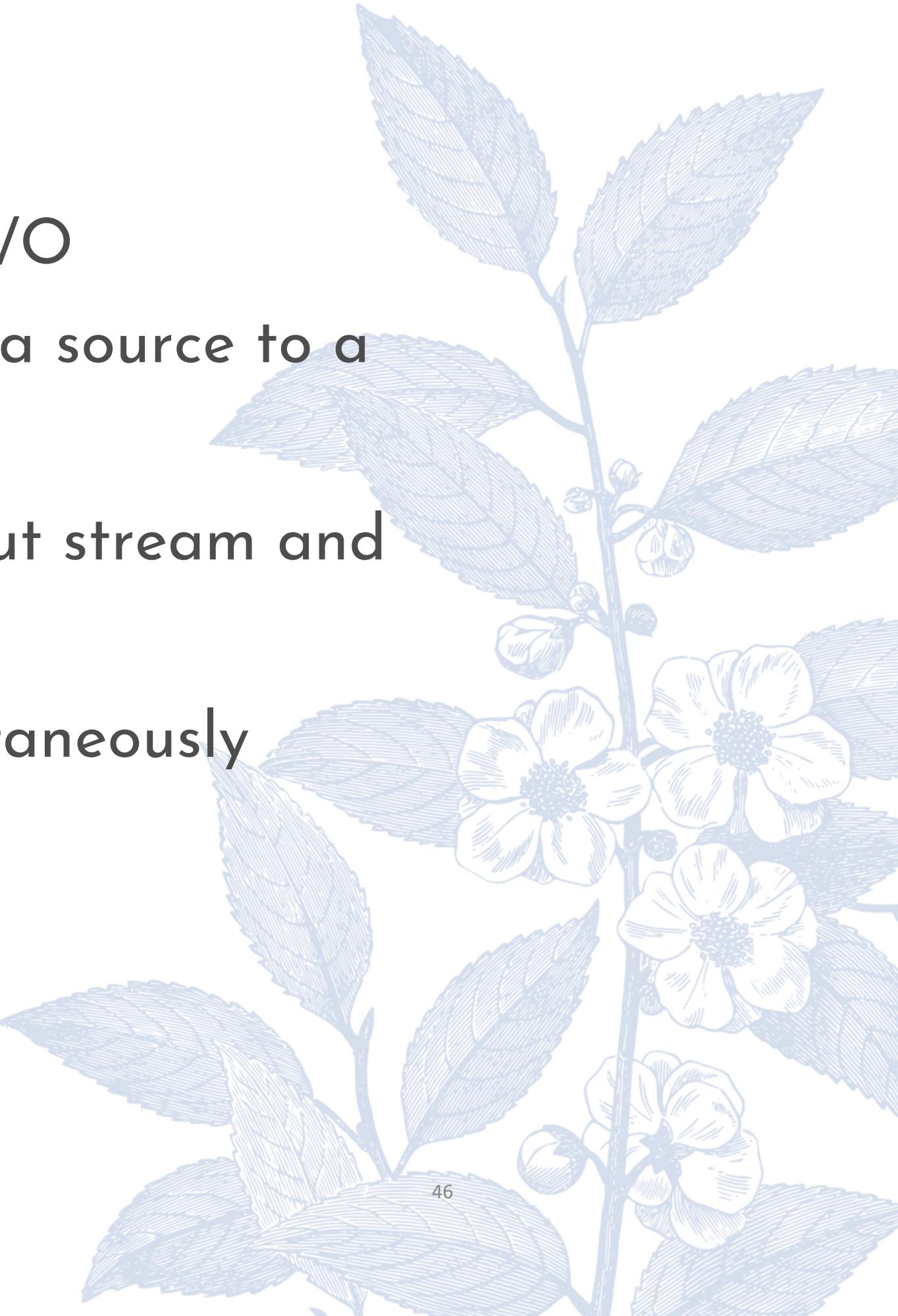
The throw Statement

✓Exceptions are thrown using the throw statement

✓Usually a throw statement is executed inside an if statement that evaluates a condition to see if the exception should be thrown

I/O Exceptions

✓Let's examine issues related to exceptions and I/O

✓A stream is a sequence of bytes that flow from a source to a destination

✓In a program, we read information from an input stream and write information to an output stream

✓A program can manage multiple streams simultaneously

Standard I/O

✓There are three standard I/O streams:

- standard output – defined by System.out
- standard input – defined by System.in
- standard error – defined by System.err

✓We use System.out when we execute println statements

✓System.out and System.err typically represent the console window

✓System.in typically represents keyboard input, which we've used many times with Scanner

The IOException Class

✓Operations performed by some I/O classes may throw an IOException

- A file might not exist
- Even if the file exists, a program may not be able to find it
- The file might not contain the kind of data we expect

✓An IOException is a checked exception