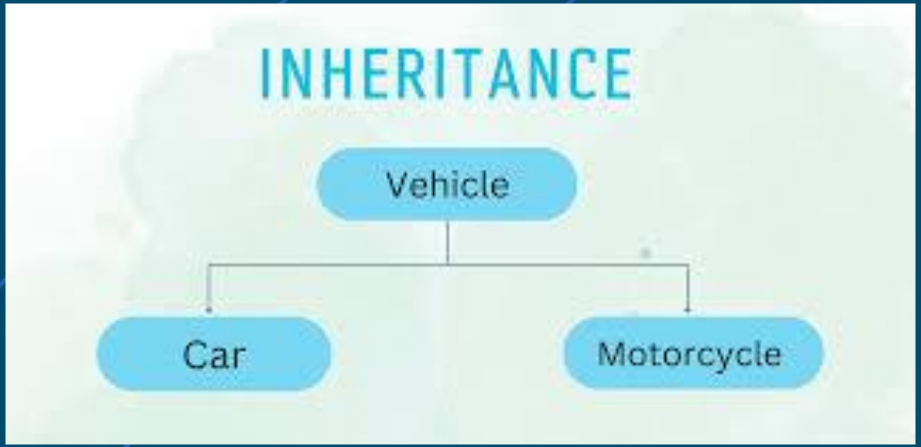
Féidearthachtaí as Cuimse
Infinite Possibilities



Inheritance Part 2

Object Oriented programming

OLLSCOIL TEICNEOLAÍOCHTA BHAILE ÁTHA CLIATH
TU DUBLIN
TECHNOLOGICAL UNIVERSITY DUBLIN

Inheritance

- Dictionary

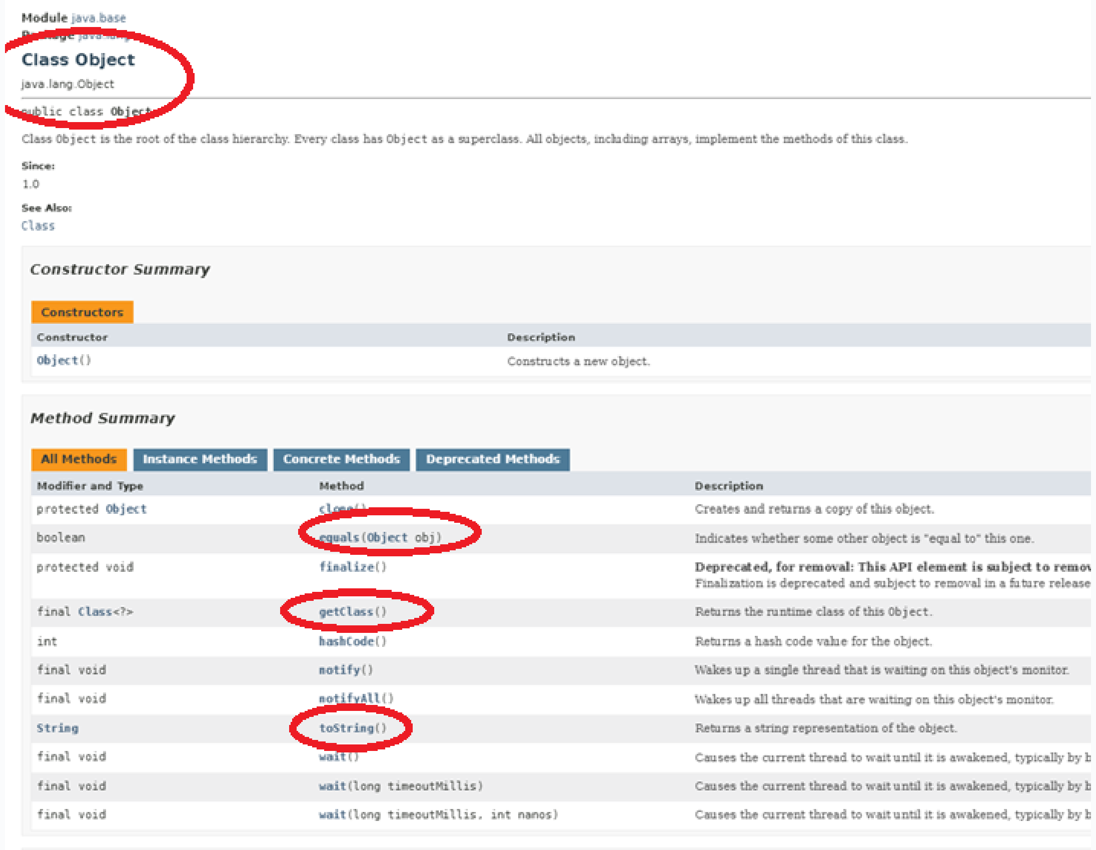
“To receive from predecessors”..

Recap – Creating a Child class

Concept	What it Does	Example
extends	Makes one class inherit from another	class Student extends Person
super(...)	Calls the parent class constructor	super(name, age)
super.toString()	Calls the parent class method	super.toString()
@Override	Indicates that we’re replacing a parent method	@Override public String toString()

Object Class

- In Java, **all classes implicitly inherit from java.lang.Object**, even if you don't write extends Object.
- *root* of the class hierarchy
- Common methods such as toString(), equals(), hashCode(), and getClass() come from Object.
- In UML or design diagrams, you normally don't show Object



Module java.base
Package java.lang

Class Object

java.lang.Object

`public class Object`

Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class.

Since:
1.0

See Also:
`Class`

Constructor Summary

Constructors

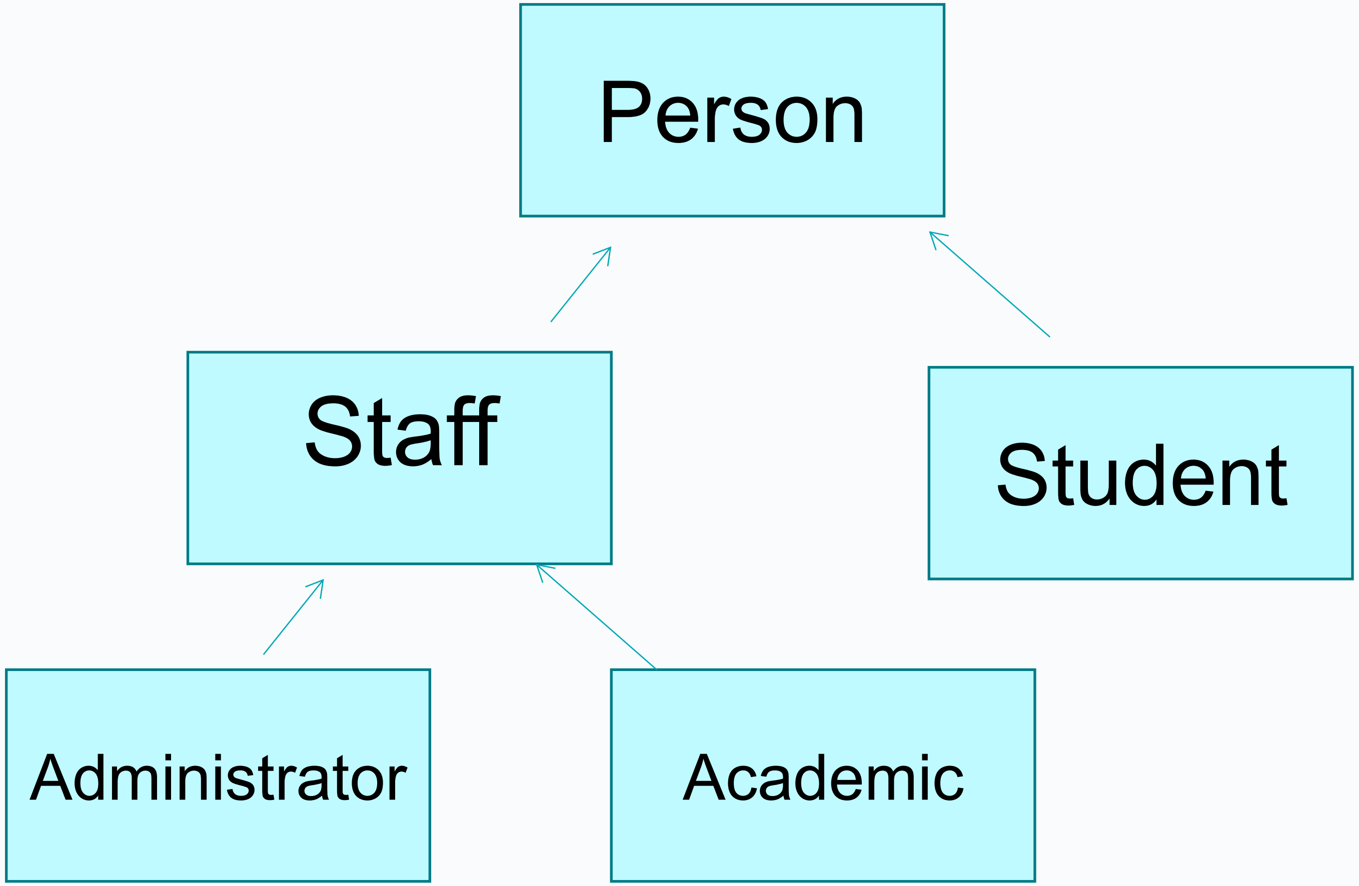
Constructor	Description
`Object()`	Constructs a new object.

Method Summary

Modifier and Type	Method	Description
`protected Object`	`clone()`	Creates and returns a copy of this object.
`boolean`	`equals(Object obj)`	Indicates whether some other object is "equal to" this one.
`protected void`	`finalize()`	**Deprecated, for removal: This API element is subject to remov** Finalization is deprecated and subject to removal in a future release
`final Class<?>`	`getClass()`	Returns the runtime class of this `Object`.
`int`	`hashCode()`	Returns a hash code value for the object.
`final void`	`notify()`	Wakes up a single thread that is waiting on this object's monitor.
`final void`	`notifyAll()`	Wakes up all threads that are waiting on this object's monitor.
`String`	`toString()`	Returns a string representation of the object.
`final void`	`wait()`	Causes the current thread to wait until it is awakened, typically by b
`final void`	`wait(long timeoutMillis)`	Causes the current thread to wait until it is awakened, typically by b
`final void`	`wait(long timeoutMillis, int nanos)`	Causes the current thread to wait until it is awakened, typically by b

Object “Types”

An important concept in java



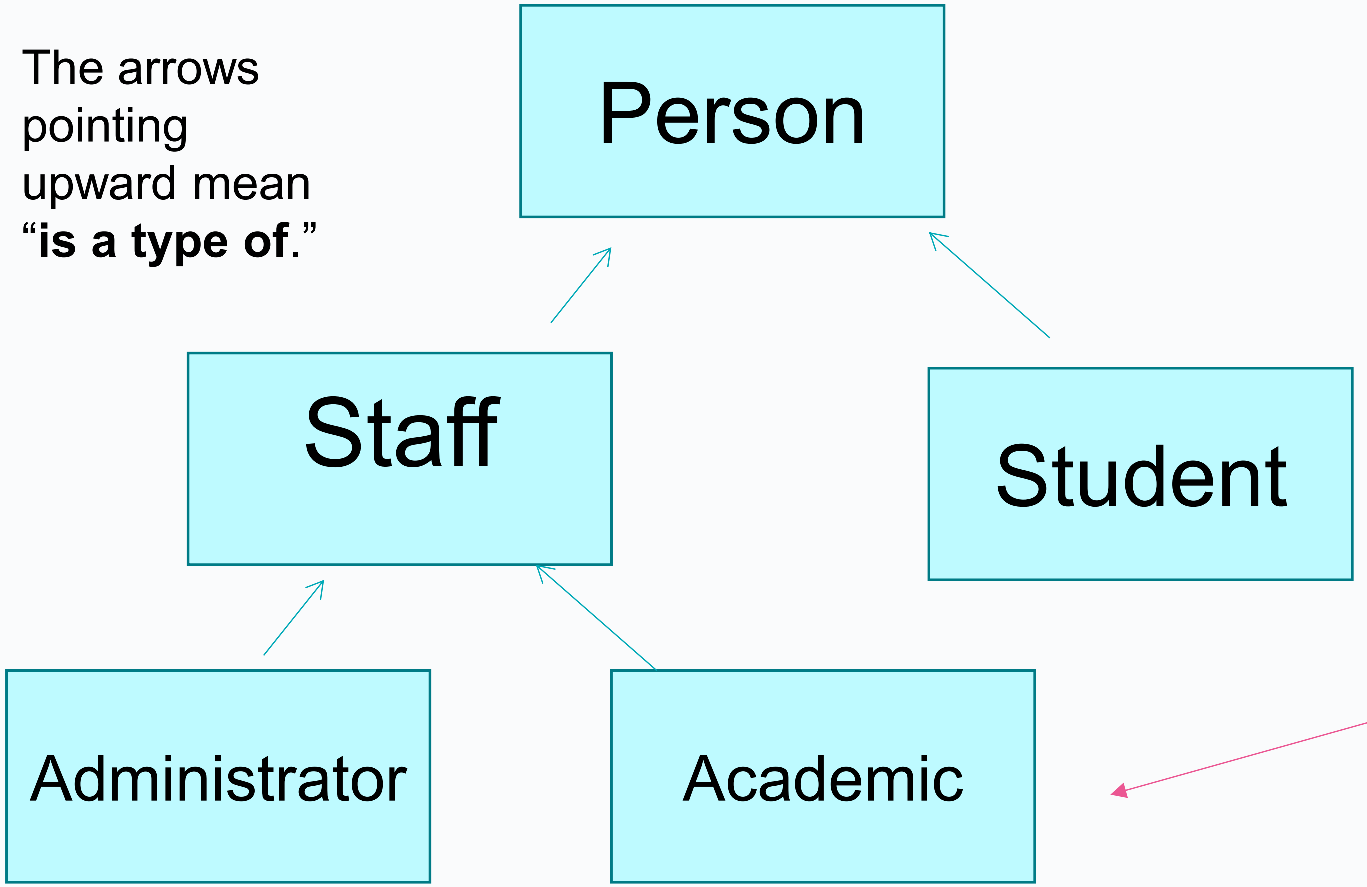
Objects created from subclasses can be treated as objects of their parent classes —
this is the basis of **polymorphism** and **type hierarchy** in object-oriented programming.

POLYMORPHISM

Polymorphism in Java is the ability of an **object or method** to take many forms, allowing the same method name to perform different actions depending on the object it belongs to.

Object Types

The arrows pointing upward mean “**is a type of**.”



Person

Staff

Student

Administrator

Academic

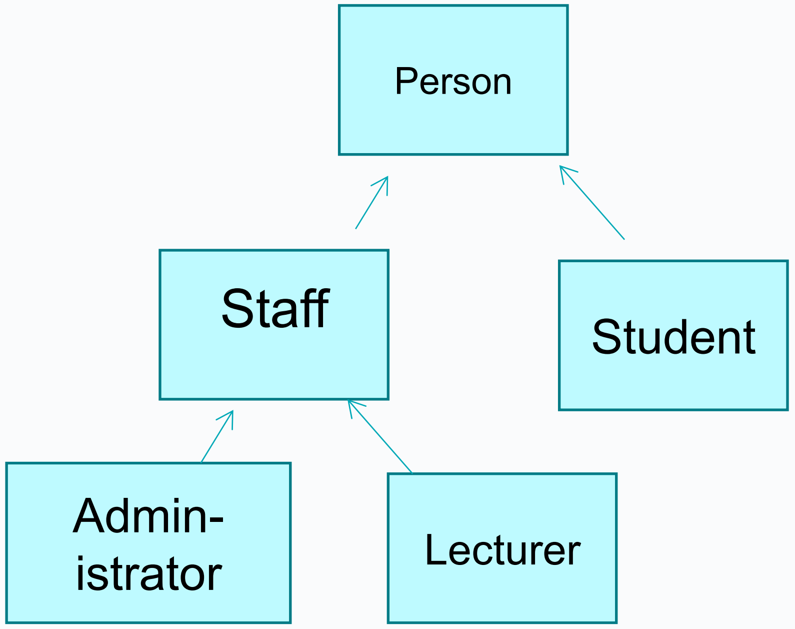
What is the object type(s) for an instance of this class

Or this class

Casting objects

"Casting" is taking an object of one type and converting into another type

In class hierarchies.. works a specific way:



Example

```
Person p1 = new Student();  // create a person object
Student s1 = (Student) p1;   // changes a person object called
p1 into a Student object
```

Or **upcasting**

```
Student s1 = new Student();    // A Student object
Person p1 = s1;                // Upcasting: Student → Person

Person p2 = new Staff();
Staff  a1 = (Staff) p2; ///downcasting
```

you should only downcast something that was previously upcasted

Polymorphism – Many Forms

```
Person p = new Student(); // Upcasting
p.printInfo();            // Prints: "This is a Student“
```

When a superclass reference points to a subclass object, calling an overridden method executes the subclass’s version — decided at runtime. This is **dynamic binding or late binding**

Method Overriding

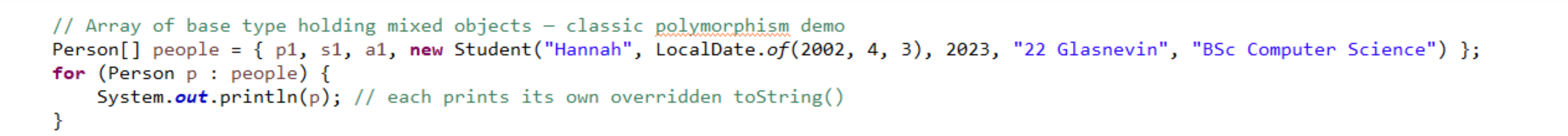
- Different classes in the hierarchy do things in “their own way” – i.e. have their own version of a method

- Note: Use **super.superclassmethod()** from the subclass method if the superclass does part of the work.
  - avoiding code repetition

- An example is the toString() method

Essentials of Method Overriding

- Same **method name**
- Same **parameter list**
- Same or **compatible return type**
- **Occurs between superclass and subclass**
- **@Override** annotation (recommended)
- **Access level** cannot be reduced
- **Static** and **final** methods **cannot** be overridden
- **Happens at runtime** (polymorphism)



```
// Array of base type holding mixed objects – classic polymorphism demo
Person[] people = { p1, s1, a1, new Student("Hannah", LocalDate.of(2002, 4, 3), 2023, "22 Glasnevin", "BSc Computer Science") };
for (Person p : people) {
    System.out.println(p); // each prints its own overridden toString()
}
```

Question

- What is the difference between method overriding and method overloading?

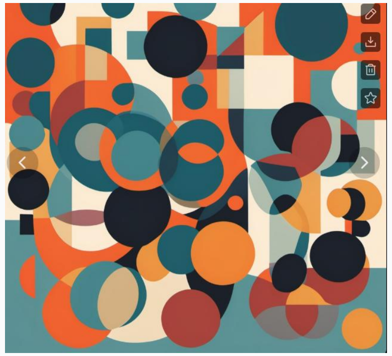
contrast with **overloading** (same name, different parameters, compile-time).

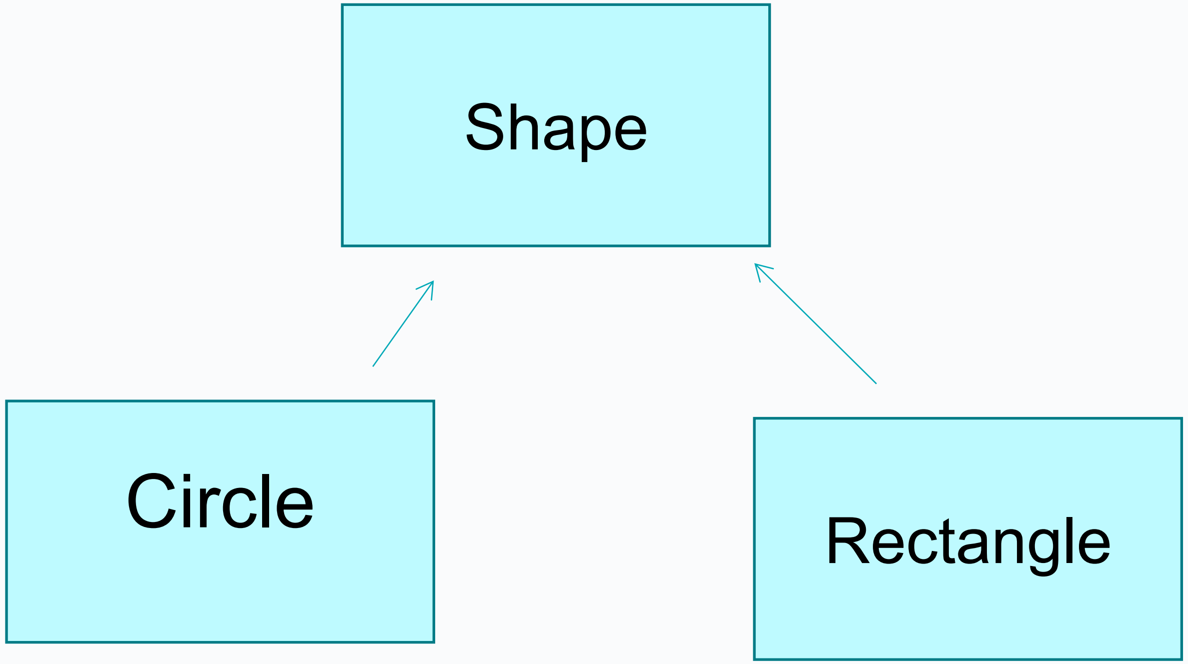
Another Scenario for inheritance

- Creating a game that will require different **shapes**:

- Circles, rectangles , etc

Another Scenario





Shape

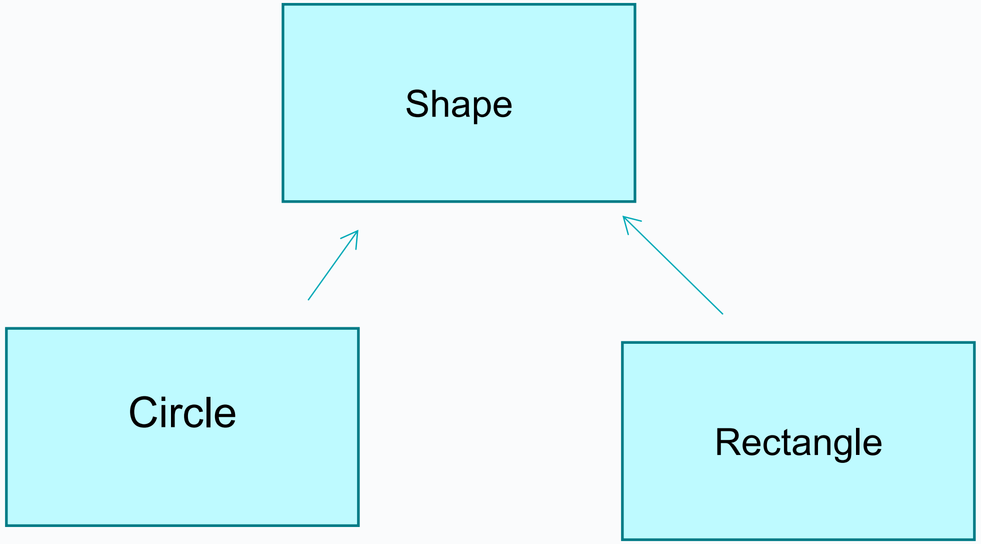
Circle

Rectangle

- Shape at the top of the hierarchy
- Circle/ Rectangle inherit from it.
- Each one has a method to **calculate its area**

  -

Another Scenario



- A class each..
- Constructor..
- Attributes..
- Its own calculateArea() method

An aside: Arrays

- In Java: Declare the type of objects it will hold – and either the length OR contents

```
String []  setOfWords= new String[4];
```

or

```
String[] setOfWords = {"enne", "meene", "miny",
"mo"};
```

Any type of object e.g

```
Person []  people = new Person[20];
```

Back to Shapes example

- In Java: Declare the type of objects it will hold – and either the length OR contents

```
Shape[]  setOfShapes= new Shape[4];
```

Set each entry to either a Circle or a Rectangle

And loop around calling calculateArea()..

Polymorphism

- In a class hierarchy – “**same**” method in different classes;
- The behaviour differs depending on the object type
  - E.g. calculateArea

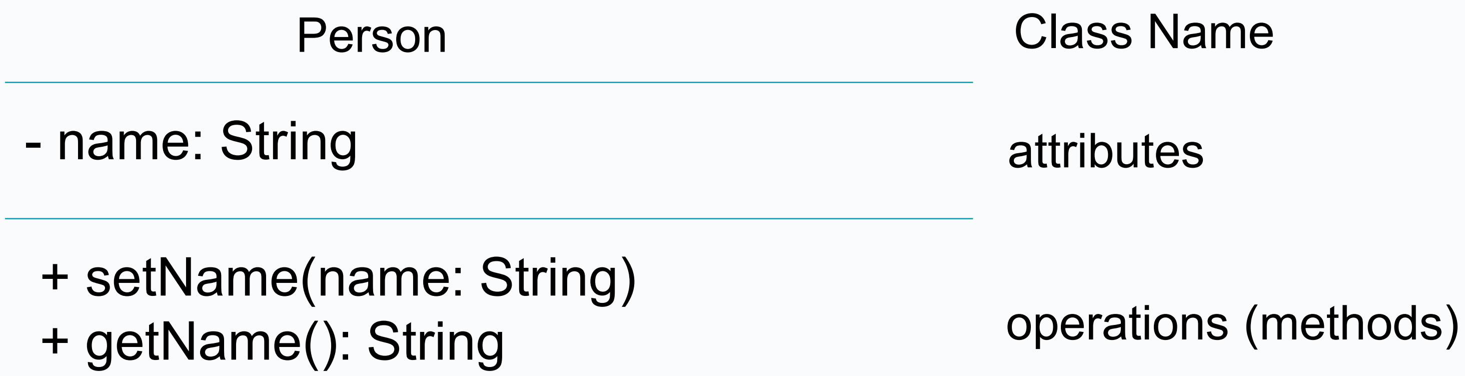
- Demo.. Using Shapes array

- **Dynamic binding**: the correct method (in this case, calculateArea()) is called depending on the object type..

UML

- In industry, Unified Modelling Language used to specify design.
- UML class diagrams often used

Note: +s/-s,
method
signature,
parameter list,
return types



Person

- name: String

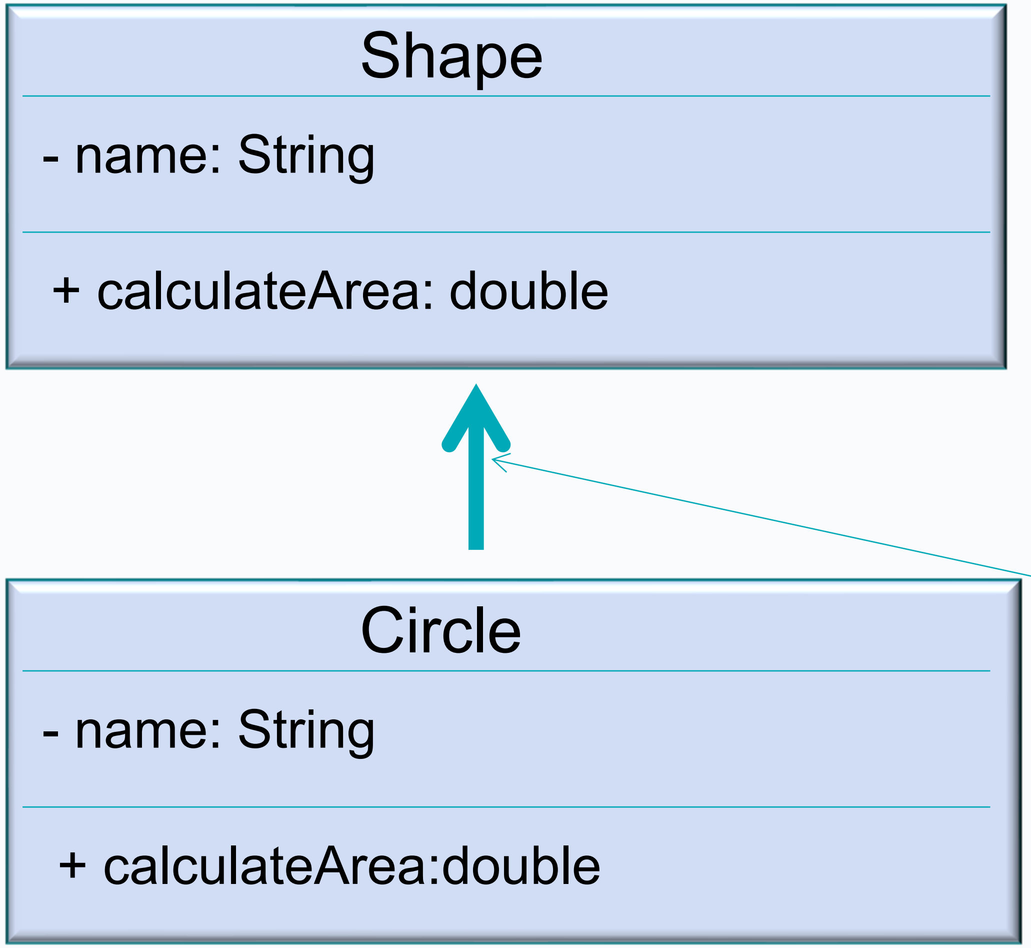
+ setName(name: String)
+ getName(): String

Class Name

attributes

operations (methods)

UML : class inheritance



Shape

- name: String

+ calculateArea: double

Circle

- name: String

+ calculateArea:double

Solid
arrow indicates
class (or abstract
class) inheritance

Dictionary

- A **shape** is a form or outline of an object — something that has boundaries, dimensions, and area.

- Modelling terms: A **Shape** is a *general concept* that represents any geometric figure that has an area and possibly other shared features (like colour, position, circumference or name).

Abstraction

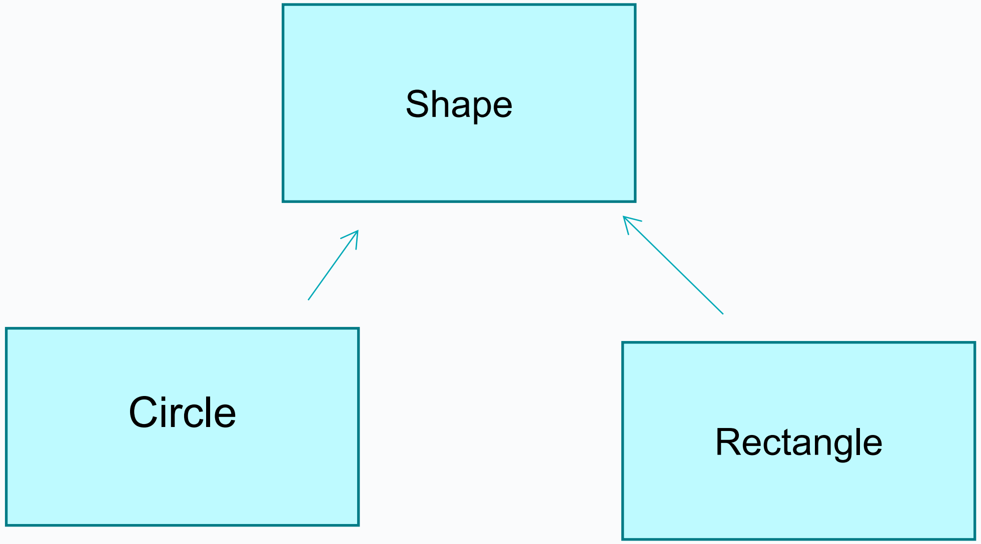
Abstraction in Java is the process of **hiding the complex details** of how something works and **showing only the essential features**.

Abstract Classes

- An abstract class or method is defined by the keyword abstract

  -
    ```
    abstract class Shape {
            …
      abstract double calculateArea();
      abstract double
      calculateCircumference();
            …
    };
    ```
- Any class with an abstract method is automatically abstract and must be declared as such
- Opposite of concrete classes (which can be instantiated)

Scenario



- Shape wasn’t ever a “real” thing – no attributes (WELL NO geometric ATTRIBUTES).
  Only its subclasses make good objects
- Can make it abstract – and enforce the calculateArea() method to be implemented

Abstract Class

- An **abstract class** in Java implements **abstraction** by providing a **blueprint** for other classes — it defines **what should be done**, but not necessarily **how it’s done**.
- Here’s how:
- 🧩 **Abstract methods** in the class have **no body** — they only declare the method name and parameters.
    - This hides the internal implementation (the *“how”*) and focuses on the *“what”*.
    - Subclasses must **implement** these methods, providing the actual behavior.
- 🧱 The **abstract class itself** can also have **regular (concrete) methods** — these show shared functionality that all subclasses can use, without needing to rewrite it.

Q Why would we use abstract classes?

“final” keyword

- Final attributes - can’t be changed
  - Used for constant values e.g. ?

```
public final double xPos;
```

- A Final class - can’t be subclassed

```
public final class Person..
```

- A final method – can’t be overridden

```
public final void someMethod()
```

What we covered

- Inheritance
  - Why it’s used - No 1 reason: code re-use
  - How it’s used - “extends”

- “Object” class

- Object types / Casting
- Method overriding
- Polymorphism
- Abstract classes
- “final” keyword