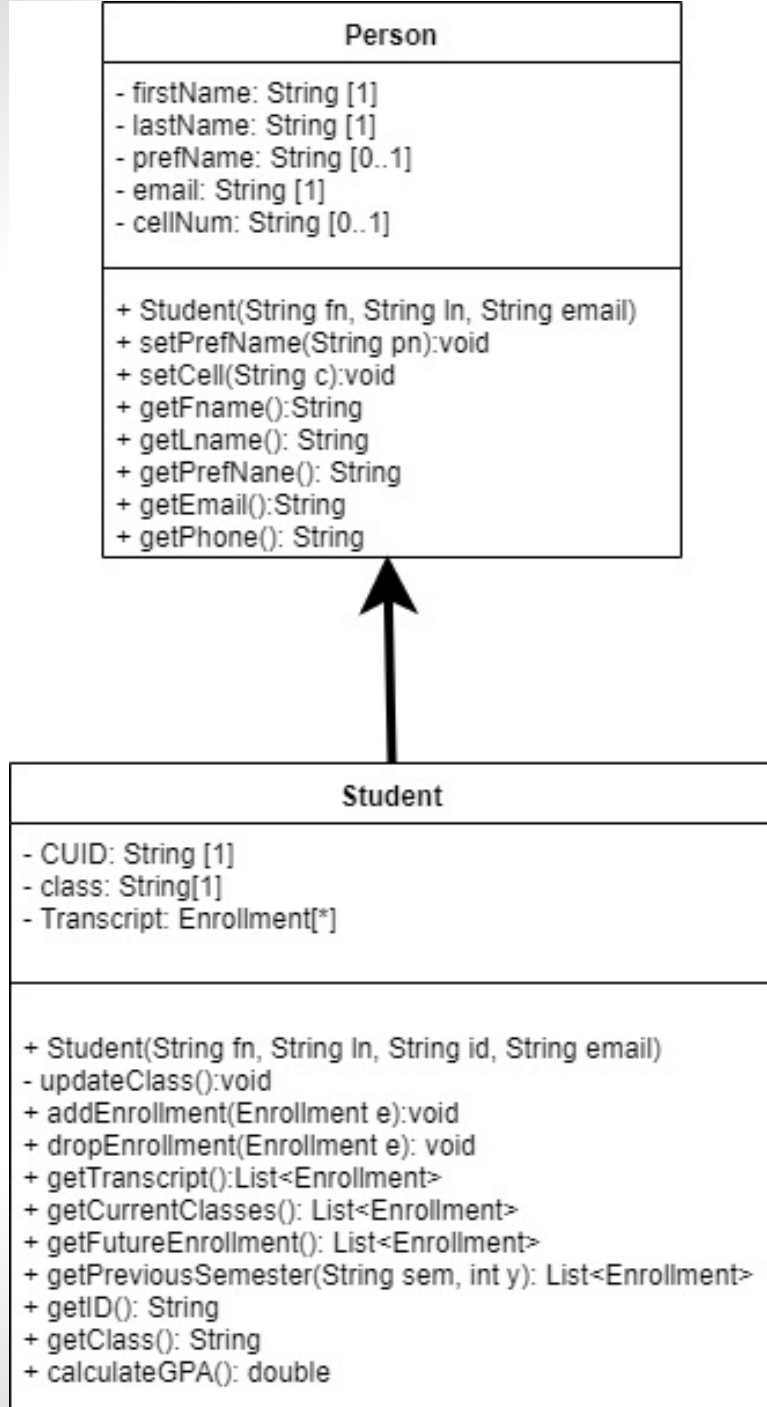


Inheritance

CPSC 2150

Inheritance: The “Extends” Relation

- The **extends** relation may hold between:
 - Two interfaces or
 - Two classes
- In either case, if **B extends A** then **B inherits** methods and data from **A**
 - **B** will have everything that **A** has, without having to actually type it into the class
- This is used to show subtyping of an object
 - **B** is a *type* of **A**



- Student **has** `firstName`, `lastName`, etc... data fields because it inherits it from `Person`
- Student has **ALL** the methods as `Person` as well
- It just saves us typing it out
- **Note:** this does not mean there is a person “contained” inside the student, this means that the `Student` *is a* `Person`

In our Student class

- When writing our `Student` class we can access data fields such as `firstName` and `lastName`
- They aren't explicitly declared inside of `Student`, they are inherited
- `Student` then adds on data fields such as `CUID` and `transcript`
- It's a shortcut:
 - `Student` will have all the same data and methods as a `Person`, plus more
 - I don't want to retype them, or copy and paste
 - I have `Student extend Person`, and the compiler does that for me
- By saying `Student extend Person`, we are saying "Take everything from `Person`, add these fields and methods, and call it a `Student`"

In our Student class

- Student also inherits all the methods from Person
 - Not the constructor!
- Can override them
 - In a person, MethodA does this, but in a student MethodA should do something else

Clients of Student

- Can call the methods specified in `Student`
- And also all the methods from `Person`
 - They inherit those as well
- `Student` *is a* `Person`
 - Any function that expects a `Person` as a parameter can accept a `Student` as well
 - `Student` has all the same methods as a `Person`, plus new ones
 - A method that is written to work for a `Person` will only call methods that belong to `Person`
 - Which `Student` inherited!
 - A `List<Person>` can include any type of `Person`, including `Students`

Inheritance vs Delegation

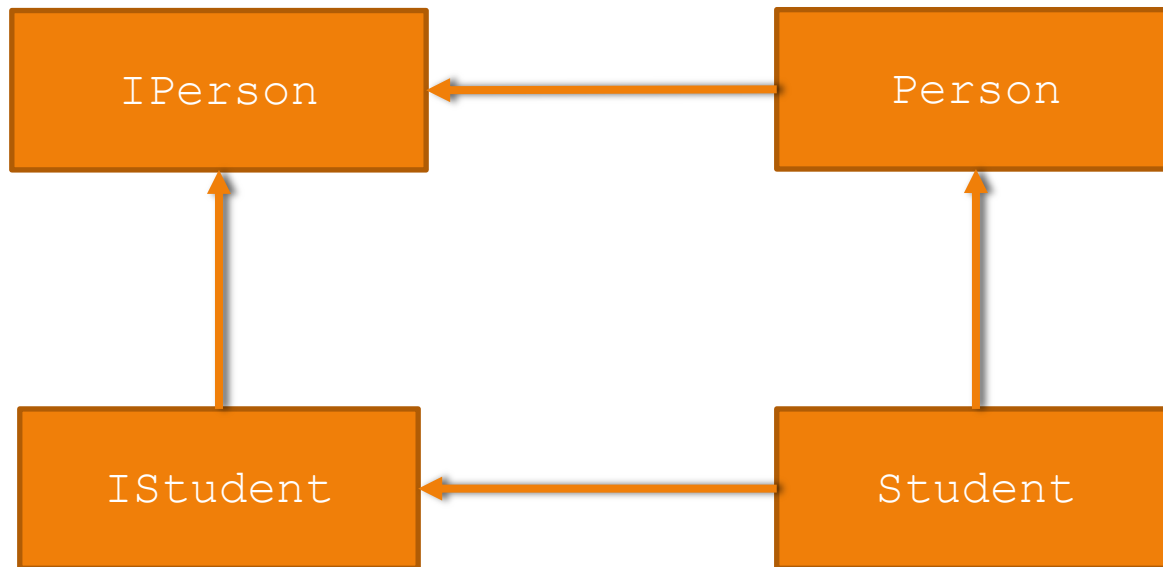
- Inheritance allows us to copy the code and data from another class
 - Reuse code!
- But we only use inheritance if we have an actual subtype relationship, not just because we want to reuse code
 - A Student *is a* Person!
- Don't use inheritance just because you want to reuse code
 - Making a Car class, and we need to know about the driver
 - Name, email, etc.
 - **BAD:** Have the Car class **extend** Person and inherit those fields and methods
 - Car *is not a* Person
 - **Good:** Car has a field called driver whose type is Person
 - Car *has a* Person
 - Person class still does all the work
 - We delegated the work to the Person class

Interfaces

- Interfaces can also take advantage of inheritance
- It works *exactly* the same way
- An interface is basically a class with no data fields, and only abstract and default methods
- When the `IStudent` interface extends the `IPerson` interface
 - Inherits all the methods
 - And then adds on the student specific ones

Implementation of an Extended Interface

- We need a class that implements the `IStudent` interface
- Must provide code for all the methods in `IStudent`
 - *Including* the methods inherited from `IPerson`
 - These methods don't appear in the code, but they still belong to `IStudent`



Abstract Classes

- Java permits you to write a kind of “partial” or “incomplete” class that contains bodies for *some but (typically) not all* of the methods of the interfaces it claims to implement
- Such a class is called an ***abstract class***:

```
public abstract class AC implements I {  
    ...  
}
```

Abstract

- Java permits a class that contains only abstract methods and no concrete methods.
- Such a class is called an abstract class.

```
public abstract class AC implements
```

```
...
```

```
}
```

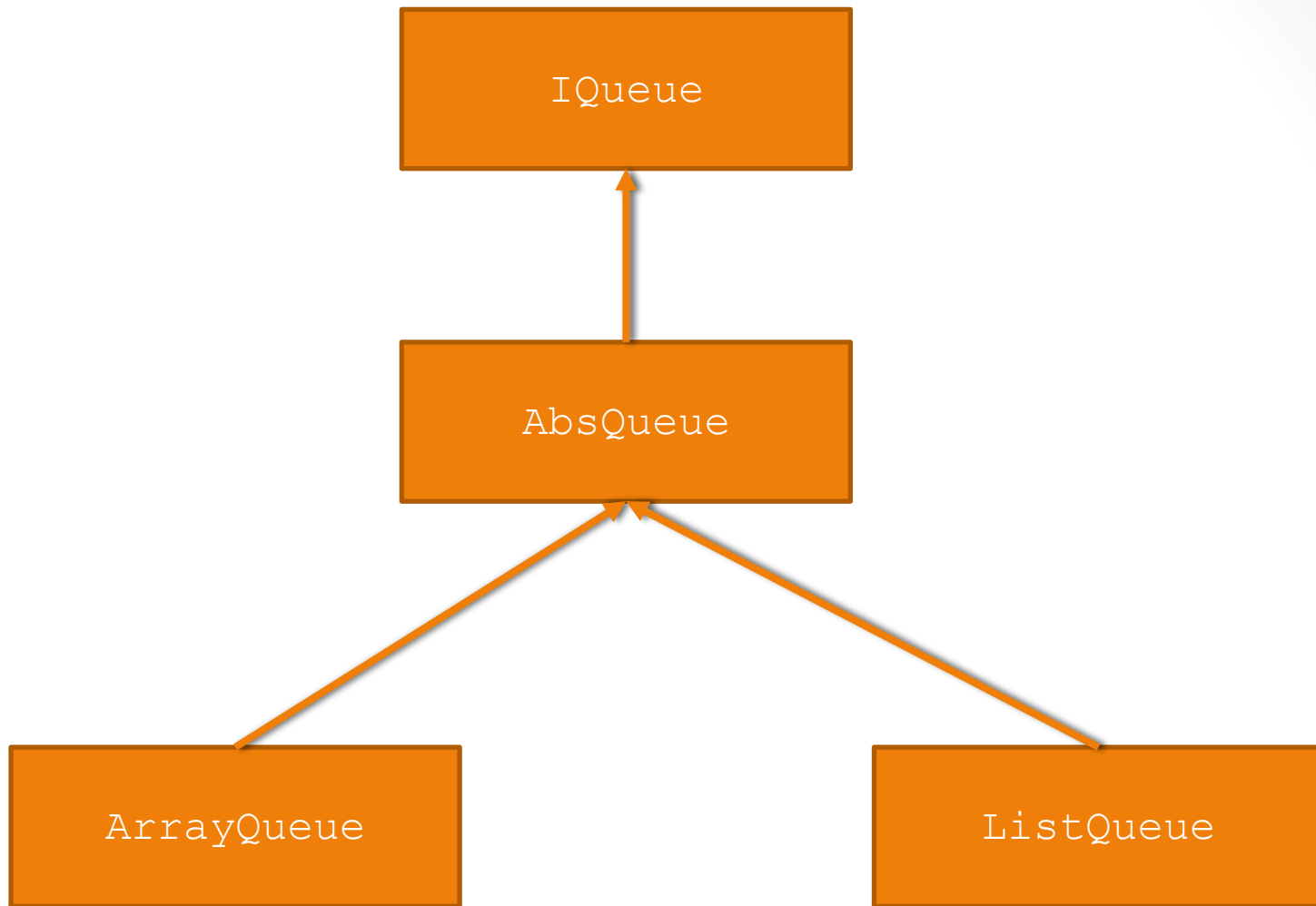
Because some methods still might not have bodies, Java will *not* let you **instantiate** an abstract class; that is, you cannot use an abstract class like a normal class and create a new object from it.

Abstract Classes

- Abstract classes can include data fields, and code for methods
 - Even non-default methods
- You can then extend from the abstract class to inherit it's data and methods like any other class

When to use Abstract Classes

- If you wanted to define data fields that every subclass must contain
- Provide methods that use those data fields that every subclass can inherit
- Provide overridden implementations for methods inherited from `Object` class
 - Can't go in the interface
 - Add it to an abstract class that implements the interface, and implement it by calling methods that belong to the interface
 - Any implementation of the interface can extend the abstract class and inherit the overridden methods

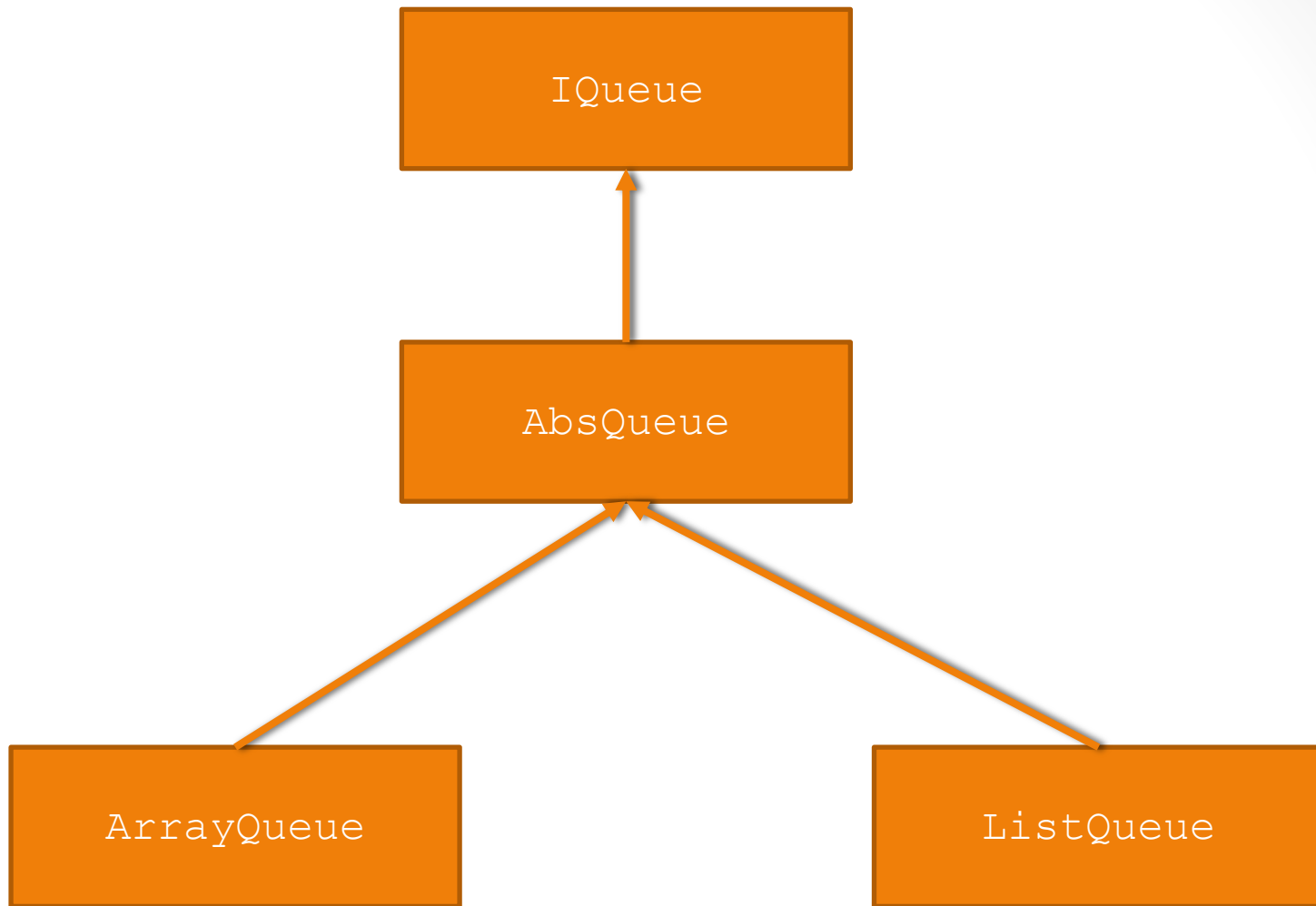


Polymorphism

- Java and other object-oriented languages decide which method body to use for any call to an instance method based on the *dynamic type* of the *receiver*
 - This type, because it is the *class* of the constructor, is always a *class* type
- This behavior for calling methods is known as *polymorphism*: “having many forms”

Declared and Dynamic Type

- Declared Type
 - Set in the declaration
 - Compiler checks the declared type to see what methods are available to be called at compile time
 - The higher up the inheritance or implementation tree we can be, the better
 - **Programming to the Interface**
- Dynamic Type
 - Set dynamically, in an assignment statement
 - Not known at compile time
 - When code is running and a method is called compiler will check the dynamic type and run that version of the method
 - Not defined? Move up the inheritance/implementation tree until you find one



Why use inheritance?

- It allows us to reuse existing code
- It allows us to use **Programming to the Interface** in languages that don't have interfaces
 - An interface is an abstract class with no data fields
- It allows us to show subtyping of data types we create

Why use Interfaces

- Allows us to separate contracts from code
- Allows us to program to the interface
 - Don't need implementation details to write code
 - Can swap out implementations at any time