accenture
Latvia ATC

Test Automation Engineering Fundamentals: Java

Module 6: Inheritance

GROW
CONFI
DENTLY

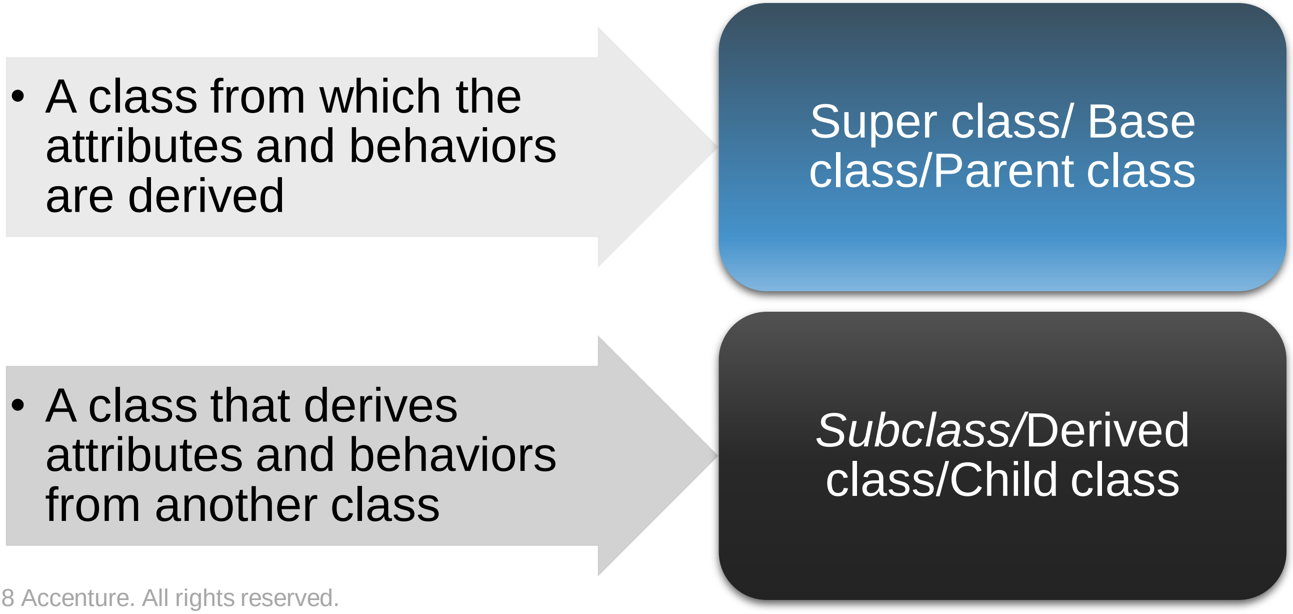
Module Objectives

accenture
Latvia ATC

- At the end of this module, participants will be able to:
  - ➢ Define inheritance
  - ➢ Describe the 'is-a' relationship
  - ➢ Explain how to use the 'extends' keyword to define an inheritance relationship
  - ➢ Identify the effects of access modifiers in an inheritance relationship
  - ➢ Explain how to override inherited methods
  - ➢ Define abstract classes and their use
  - ➢ Define interfaces and their use
  - ➢ Describe the difference between extending from a class and implementing interfaces
  - ➢ Discuss up-casting / down-casting references
  - ➢ Discuss virtual method invocation in Java

Defining Inheritance

- Inheritance is one of the language constructs that encourages the *re-use* of code by allowing the behavior of existing classes to be extended and specialized.
- ***Inheritance*** defines a hierarchical relationship among classes wherein one class shares the attributes and methods defined in one or more classes.



- A class from which the attributes and behaviors are derived

Super class/ Base class/Parent class

- A class that derives attributes and behaviors from another class

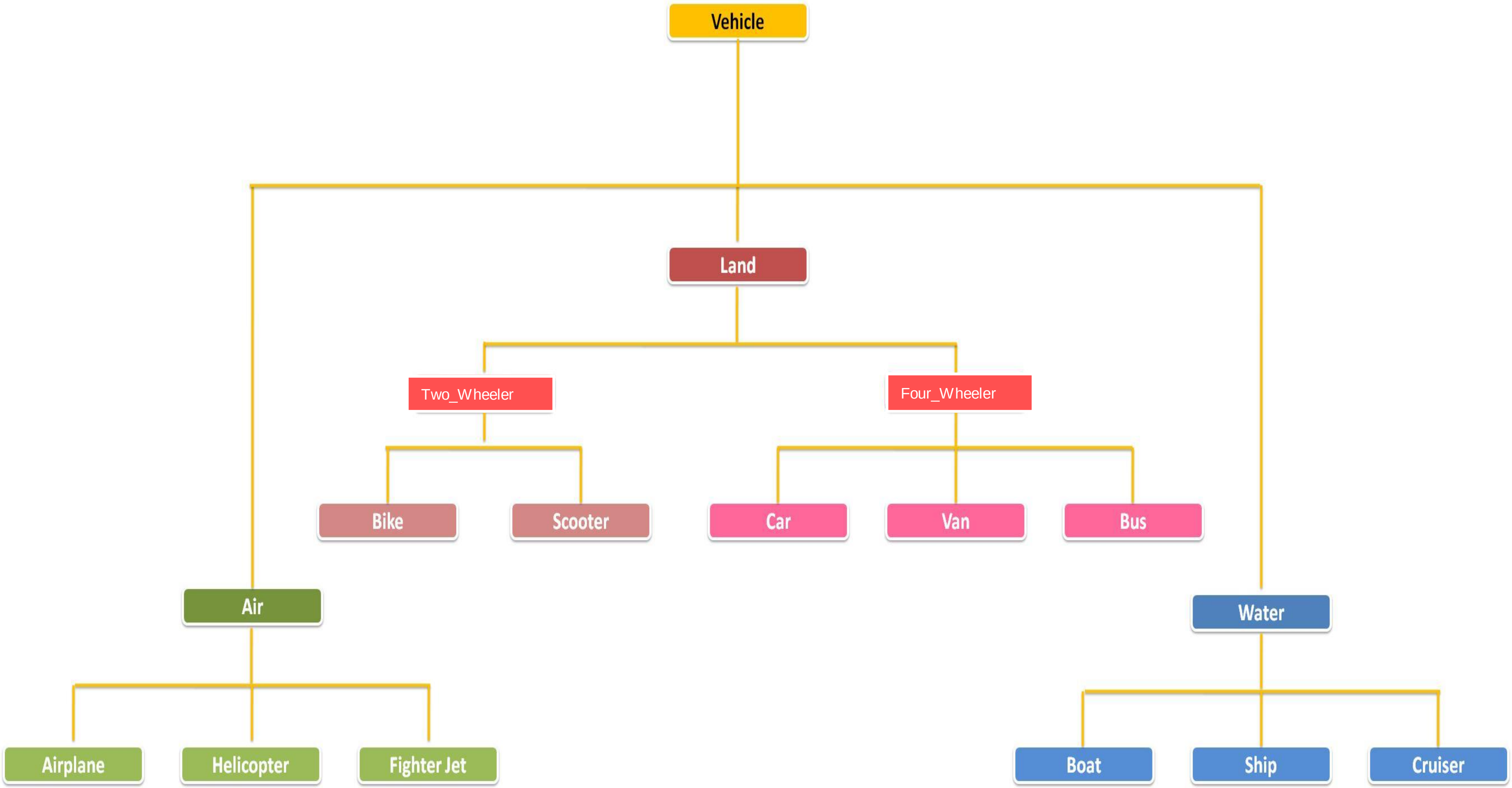
Subclass/Derived class/Child class

‘is a’ Relationship

- An inheritance relationship is described as an **“is-a”** relationship between two concepts
- Concept A “is a” Concept B means that Concept A is a specialization (subclass) of Concept B
- Concept A will have all the attributes and behavior of Concept B in addition to its own unique attributes and behavior

Inheritance Hierarchy

accenture
Latvia ATC



Inheritance

- Inheritance is implemented in Java with the keyword ***extends*** during class declaration
- By extending another class, all attributes and behavior of the parent class are automatically inherited by the child class

Example:

```
public class Cat extends Animal{
        //Define additional attributes that make a Animal into a Cat
}
```

Refer to example Student_I.java, Person_I.java, InheritanceSample.java inside package sef.module6.sample.

Inheritance and Access Modifiers

- Public and protected fields are inherited and are accessible by all subclasses
- Private fields are not inherited by a subclass
- Public and protected methods from superclass are inherited by subclasses.
- These public and protected methods in the superclass can be used to access private fields/methods of the superclass.

i Refer to example Student_I.java, Person_I.java, InheritanceSample.java inside package sef.module6.sample.

Inheritance and Keyword "super"

- The *super* keyword allows a subclass to reference a field or method that belongs to its immediate parent class. The *super(<parameters>)* method can be called to refer to a parent class constructor.

```
            class Child extends Parent{
            public Child(){
                    //call parent constructor –can only be done from the Child constructor
                    super(“John Doe”);
            }
    }
```

- The *super.<field>* can be used to access a field or a method that belongs to the parent.

```
        super.aParentMethod();
```

Refer to example Student_I.java, Person_I.java, InheritanceSample.java inside package sef.module6.sample.

Activity 1 – Inheritance

- In this activity, you will:
  - Open the file ‘InheritanceActivity.java’ in the package sef.module6.activity.
  - Read the instructions and create the code to complete this program.

Overloading and Overriding

	Overloading	Overriding
Description	Method Overloading allows a subclass to redefine methods with the same method *name* but either a different number of parameters or different types of parameter in the parameter list.	Method Overriding allows a subclass to redefine methods of the same signature from the superclass.
Requirements	An overloading method must have: • The same name • Different number of parameters or types • The same or different return type	An overridden method must have: • The same name • The same number of parameters and types • The same return type

i Refer to the samples Person_P.java, Student_P.java and PolymorphismSample.java inside package sef.module6.sample.

Overloading and Overriding (cont.)

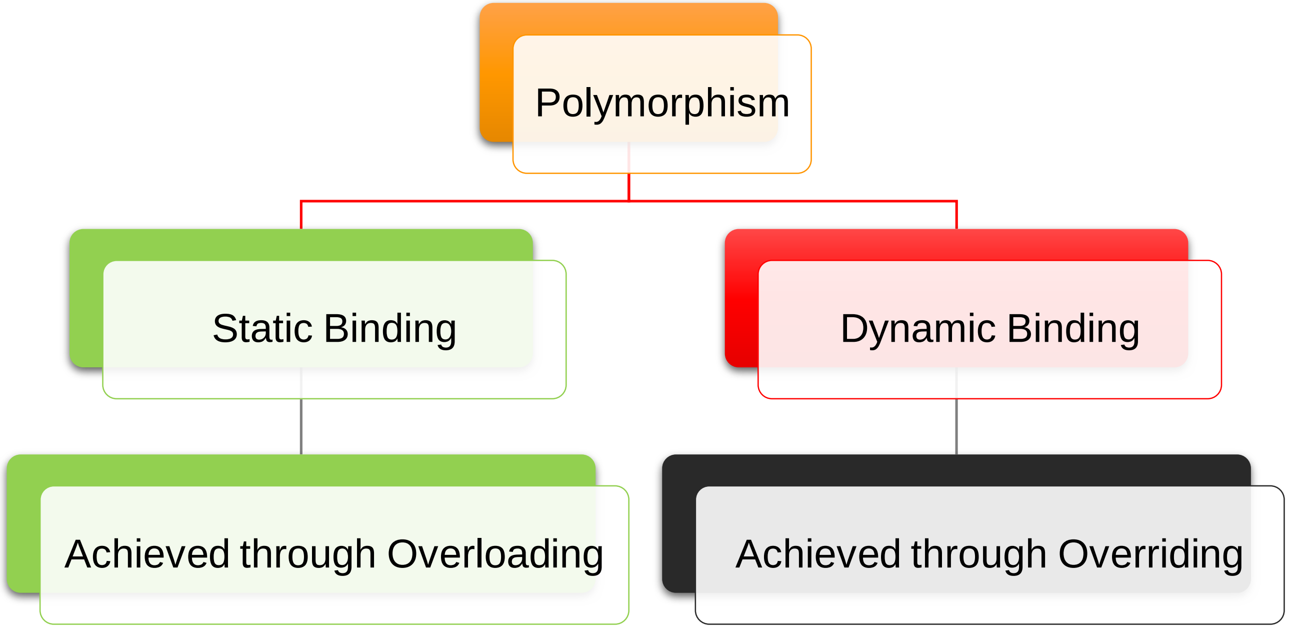
accenture
Latvia ATC

	Overloading	Overriding
Binding Type	Which overloading method to call is based on parameters type and number and decided at the compile time. This is called as Static Binding	Which overridden method to call is based on the actual *object type* and decided at runtime. This is called as dynamic Binding.
Sample	Refer to the samples Person_P.java, Student_P.java and PolymorphismSample.java inside package *sef.module6.sample.* *Explain to the participants how method* ***address****() is overloaded.*	Refer to the samples Person_P.java, Student_P.java and PolymorphismSample.java inside package *sef.module6.sample.* *Explain to the participants how method* ***announce****() is overridden.*

i Refer to the samples Person_P.java, Student_P.java and PolymorphismSample.java inside package sef.module6.sample.

Polymorphism

- Polymorphism is one of the basic principles of Object-Oriented Programming
- Polymorphism means ‘many forms’
- It refers to the ability of a reference variable to change behavior according to what object instance it is holding



Polymorphism – Static Binding and Dynamic Binding

accenture
Latvia ATC

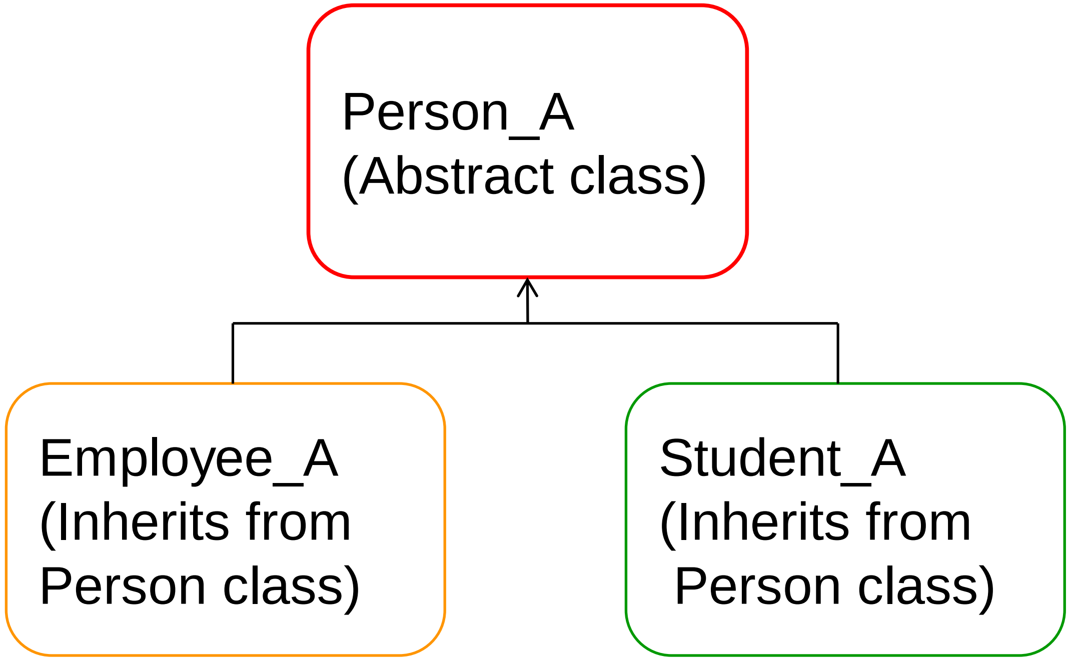
	Static Binding	Dynamic Binding
Definition	Ability to call specific behavior (method) at the compile time based on method signature	Ability to define/defer behavior specific to subclasses at run time.
How to Achieve	Can be achieved through Method Overloading	Can be achieved through Method Overriding

Defining Abstract Class

- An ***Abstract Class*** is a class that provides common behavior across a set of subclasses, but is not itself designed to have instances of its own
- An abstract class is designed as a template for other classes to follow by dictating behavior that must be implemented by its subclasses
- An abstract class can extend a class, an abstract class or implement an interface
- An abstract class can implement one or many interfaces
- An abstract class can extend only one abstract class
- Defined using the ‘abstract’ class modifier

```
public abstract Food{
        public abstract double calculateCalories();
}
```

Defining Abstract Class



Person_A
(Abstract class)

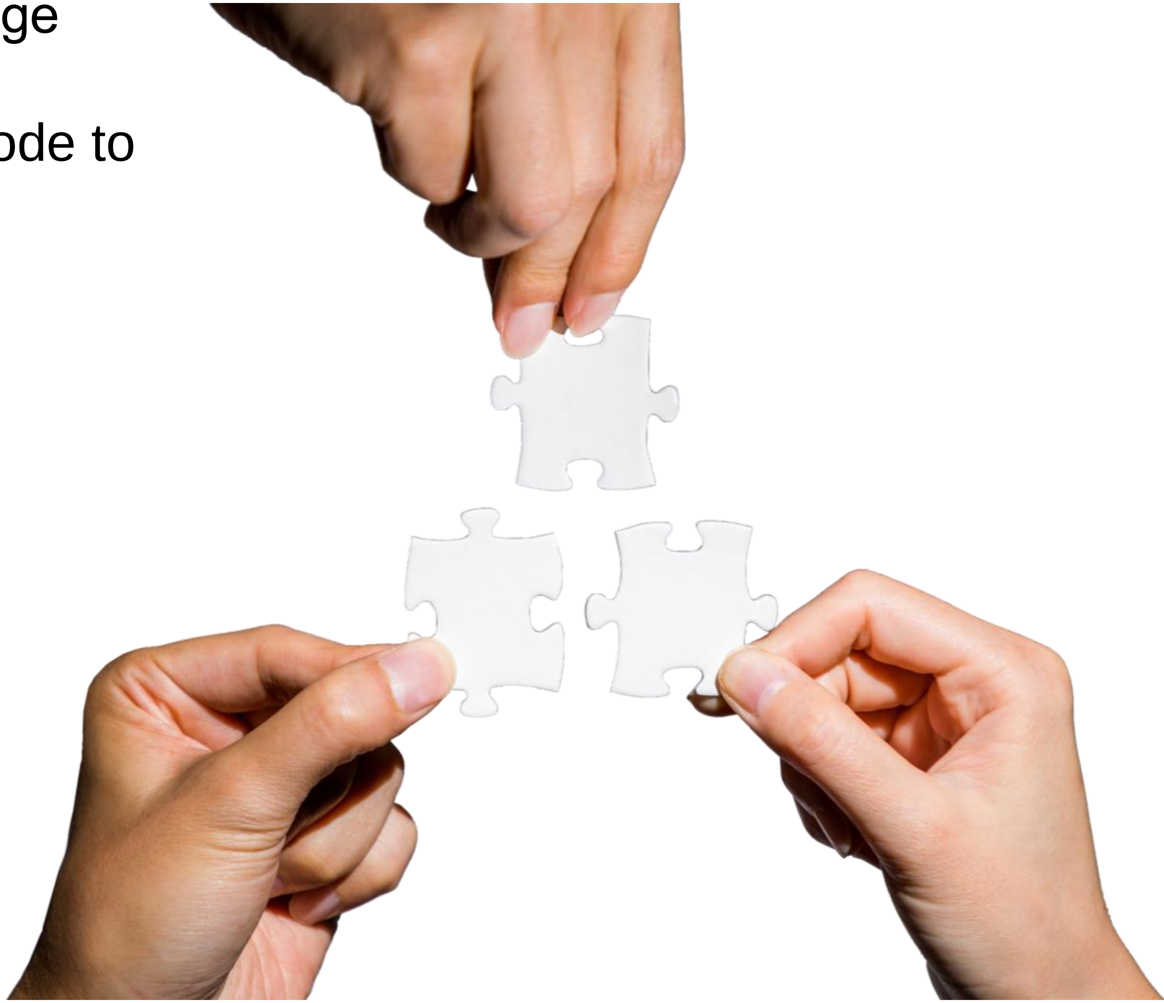
Employee_A
(Inherits from
Person class)

Student_A
(Inherits from
Person class)

i Refer to the sample inside package sef.module6.sample.

Activity 2 – Abstract Class

- In this activity, you will:
  - Open the file ‘Shape.java’ in the package sef.module6.activity.
  - Read the instructions and create the code to complete this program.

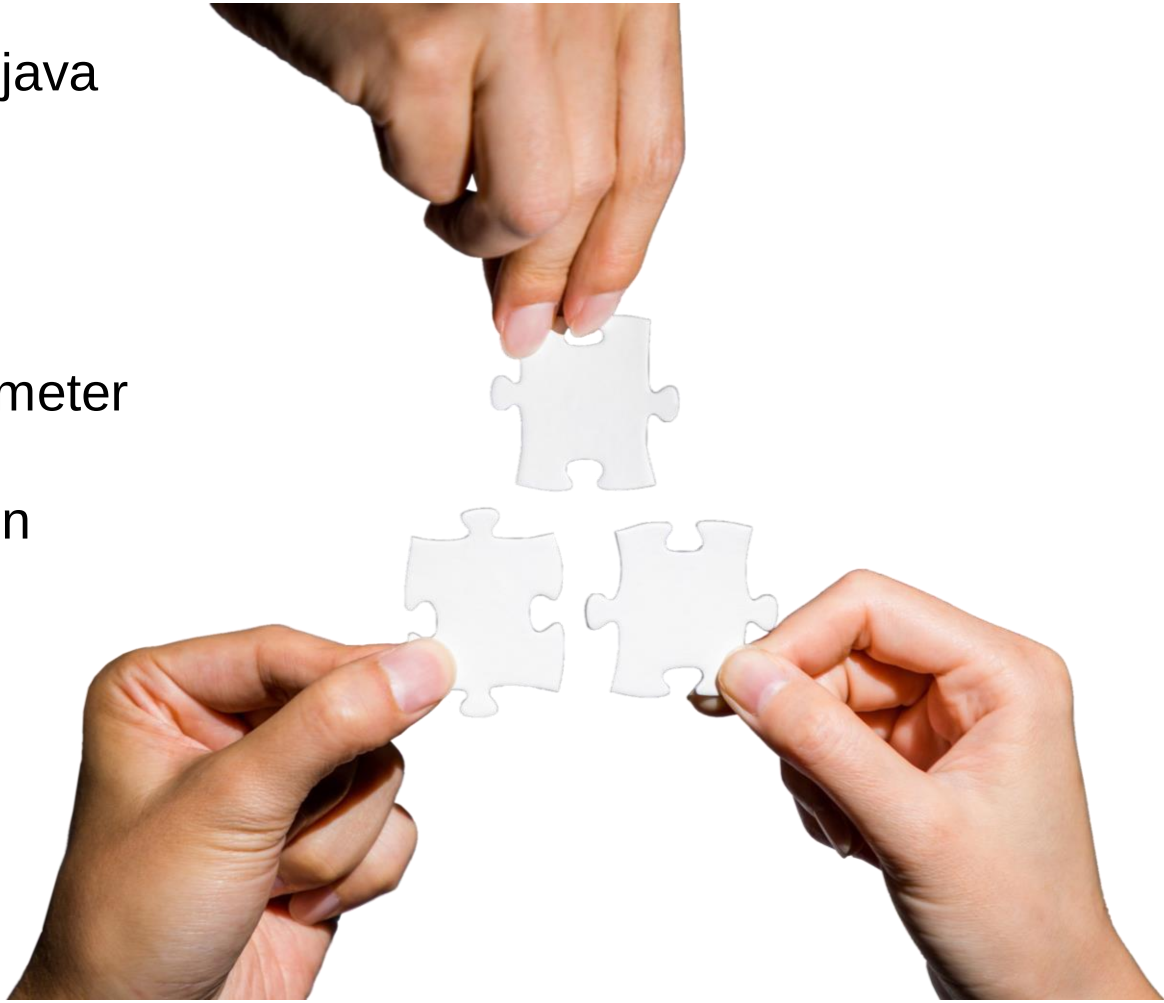


Activity 3 – Abstraction

accenture
Latvia ATC

- In this activity, you will:

1. Create a class Rectangle.java that extends Shape.java
2. Define double parameter ‘length’ and ‘breadth’
3. Write default and parameterized constructor
4. Define method caculateArea(). Note that area of Rectangle is ‘length x breadth’
5. Define method calculatePerimeter(). Note that perimeter of Square is ‘2 (length x breadth)’
6. Create a class AbstractionActivity.java with the main method
7. Write the code to print color, area and perimeter of Circle with side 5 and Rectangle with length 5 and breadth 6.
8. Print result.



Defining Java Interface

- An interface is 100% abstract class
- An Interface specifies a set of method or templates that an implementing class needs to follow
- An interface provides only a form for a class but no implementation
- An interface defines *what* a class can do but not *how* the class will do it

Implementing Interfaces

- A class implementing interfaces is required to override the inherited methods
- Interfaces are implemented using the keyword **implements**
- Rules on implementing the interface methods
  - ➢ Must have the same method signature and return type
  - ➢ Cannot narrow the method accessibility
  - ➢ Cannot specify broader checked exceptions
- Interface variables are implicitly public final static
- Interface methods are implicitly public abstract

Rules on Interface

- An interface can extend several interfaces
- Interfaces can be implemented by any class
- A class can implement several interfaces
- A class that implements an interface partially must be declared abstract
- An interface can be declared as a reference variable
- An interface cannot be instantiated

Reference Casting

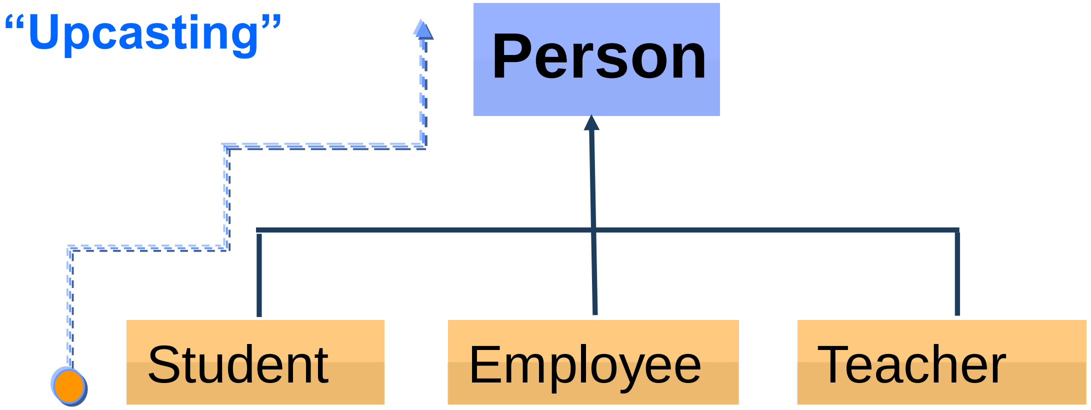
- Reference casting is converting a reference data type to another.
- Reference casting is only allowed between classes that belong to an inheritance chain.
- Reference casting is useful in many programming situations as it allows a reference variable of a specific type to point to different subtypes of objects.
- A reference type gives us a ‘view’ of an object. An object is perceived depending on what reference type is used.

i Refer to the ReferenceCastingSample.java sample code.

Reference Casting Flow

- *Upcasting* is conversion up the inheritance hierarchy.
- To upcast a **Student** object, all you need to do is assign the object to a reference variable of type **Person.** Note that the **p** reference variable cannot access the members that are only available in **Student.**

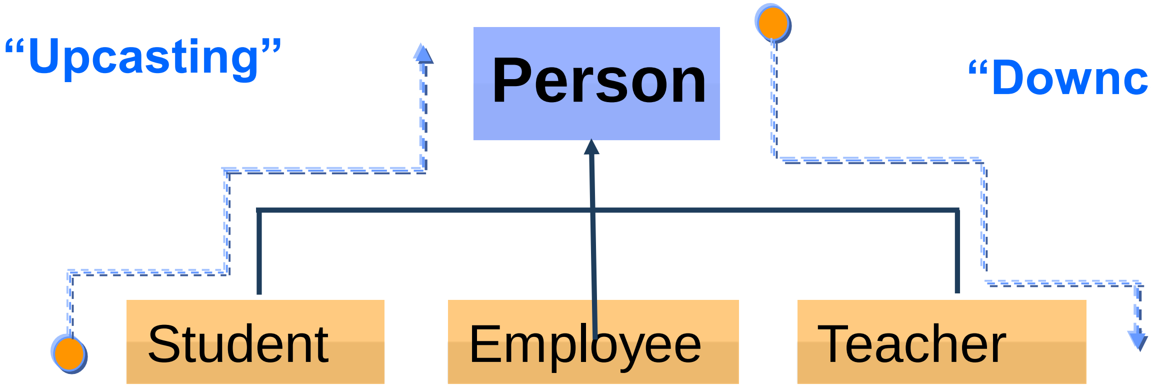
```
// * Assuming Student is a subclass of Person
       Person p = new Student(); //upcasting
```



Reference Casting Flow (cont.)

- *Downcasting* is conversion down the inheritance hierarchy.
- *Continuing from previous example: Now, to change back the **Student** object, change the* **Person** reference back to **Student.** This time, it is called downcasting because you are casting an object to a class down the inheritance hierarchy.
- Downcasting requires that you write the Student type in brackets.

```
// *Assuming p is actually pointing to a Student object (as in previous slide)
// p is Person reference of an object of type Student i.e. Person p = new Student();
Student s = (Student)p;
```

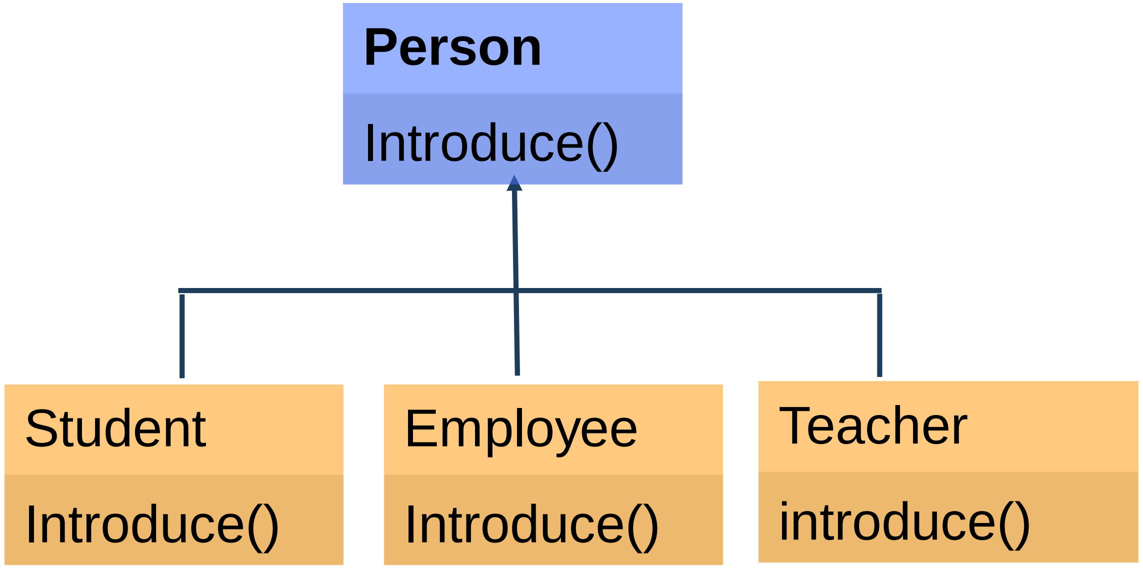


i Refer to the ReferenceCastingSample.java sample code.

Virtual Methods

accenture
Latvia ATC

- A virtual method is a method whose actual implementation is dynamically determined during runtime
- All java methods are ‘virtual’ and can be overridden by methods that belong to the sub class

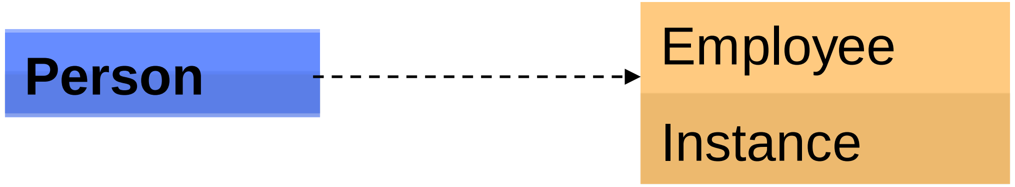


i Refer to the VirtualMethodSample.java sample code sample code.

Virtual Method Invocation

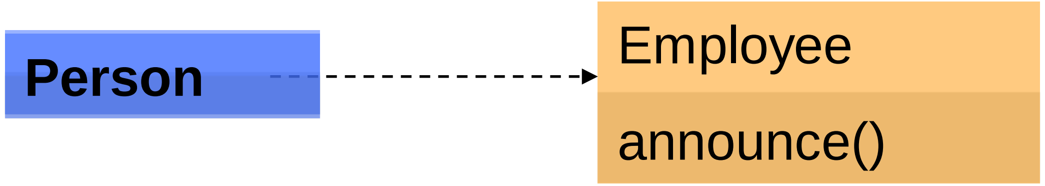
- A reference type variable can point to instances of its own type, or its subtypes through casting.

  Person p = new Employee()



- When calling an object’s methods through a reference variable, the implementation called is the one used by the object and not necessarily the reference variable.

  p.announce();



i Refer to the VirtualMethodSample.java sample code sample code.

Questions and Comments

accenture
Latvia ATC

- What questions or comments do you have?