Object Oriented Programming

ICT2122

Polymorphism

P.H.P. Nuwan Laksiri
Department of ICT
Faculty of Technology
University of Ruhuna

Lesson 03 - OOP Concepts - Part 02

Recap

- Inheritance
  - Examples
  - Handson
- Creating Sub Classes
- Behavior of Java Access Modifiers
- Types of inheritance in Java
  - Single Inheritance
  - Multilevel Inheritance
  - Hierarchical Inheritance
  - Hybrid Inheritance
  - Multiple Inheritance
- Overriding Methods
- Hiding Methods
- Hiding Fields
- Usage of this and super in Subclasses
- Constructors in Subclasses
- Usage of final keyword
- Casting Objects
- Determining Object's Type

Outline

- Polymorphism
- Method Overloading
- Method Overriding
- Dynamic Polymorphism
- Static Polymorphism

Object Oriented Concepts

- Object Oriented Programming simplifies the software development and maintenance by providing some concepts,
  - Object
  - Class
  - Inheritance
  - Polymorphism
  - Abstraction
  - Encapsulation

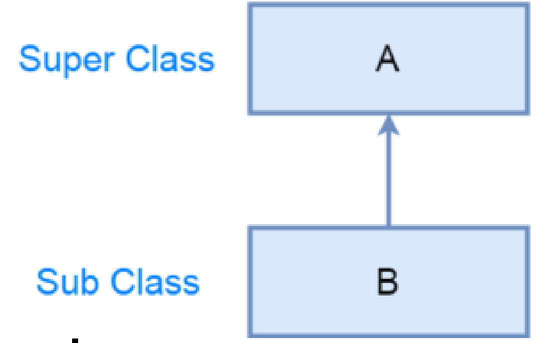
Classes and Objects

A class is like a cookie cutter; it defines the shape of objects

Objects are like cookies; they are instances of the class

Photograph courtesy of Guillaume Brialon on Flickr.

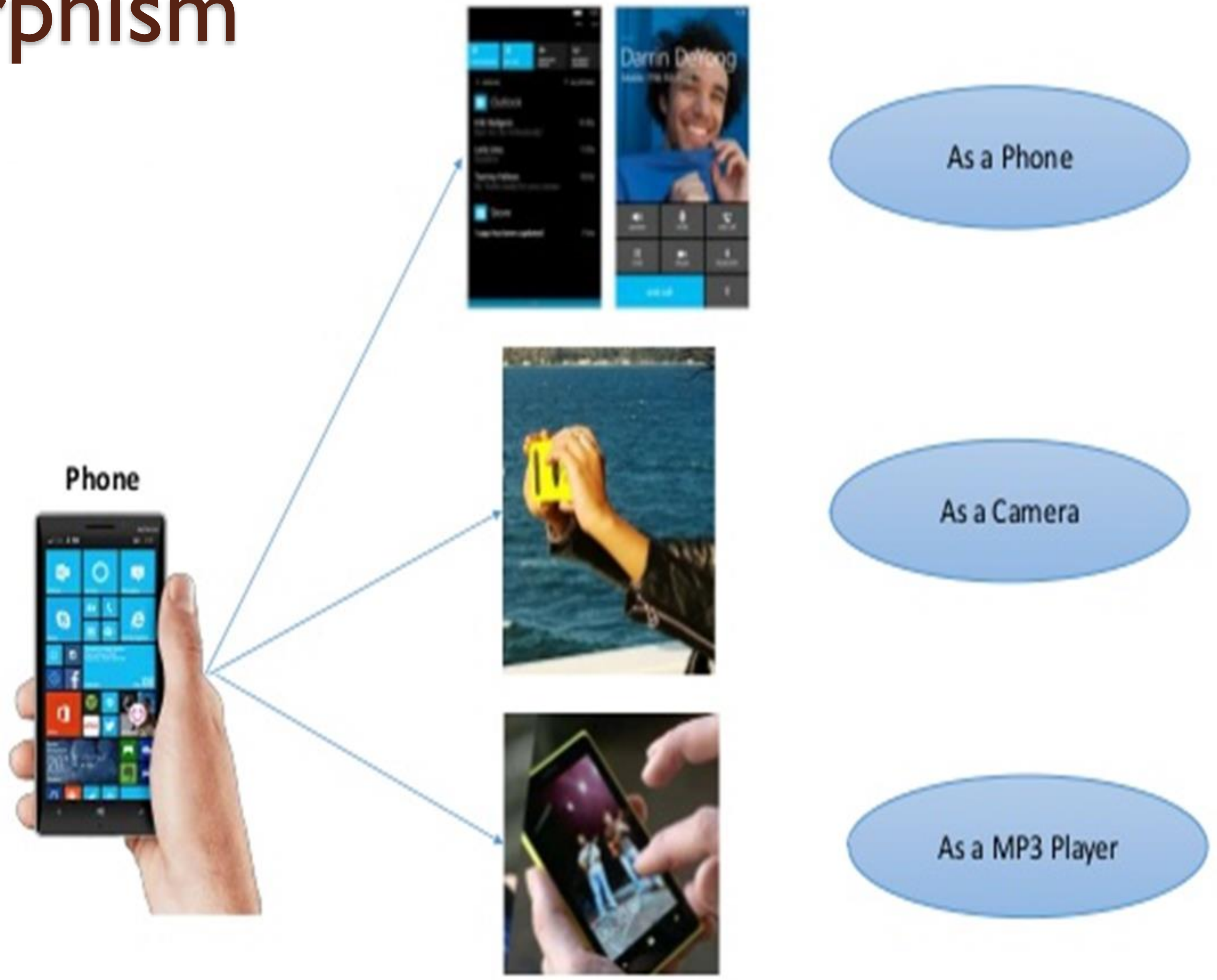
Inheritance



- Inheritance is a mechanism that allows
  - a subclass to inherit the properties and behaviors of a superclass.
- This means that the
  - subclass can access and use all the methods and variables of the superclass,
  - as well as add its own methods and variables.
- The subclass can also
  - override methods from the superclass to provide its own implementation.
- Inheritance enables
  - code reuse and makes it easier to manage and maintain complex systems
  - by reducing duplication and
  - providing a hierarchical structure for classes.
- It is a key feature of object-oriented programming and is widely used in Java

Polymorphism

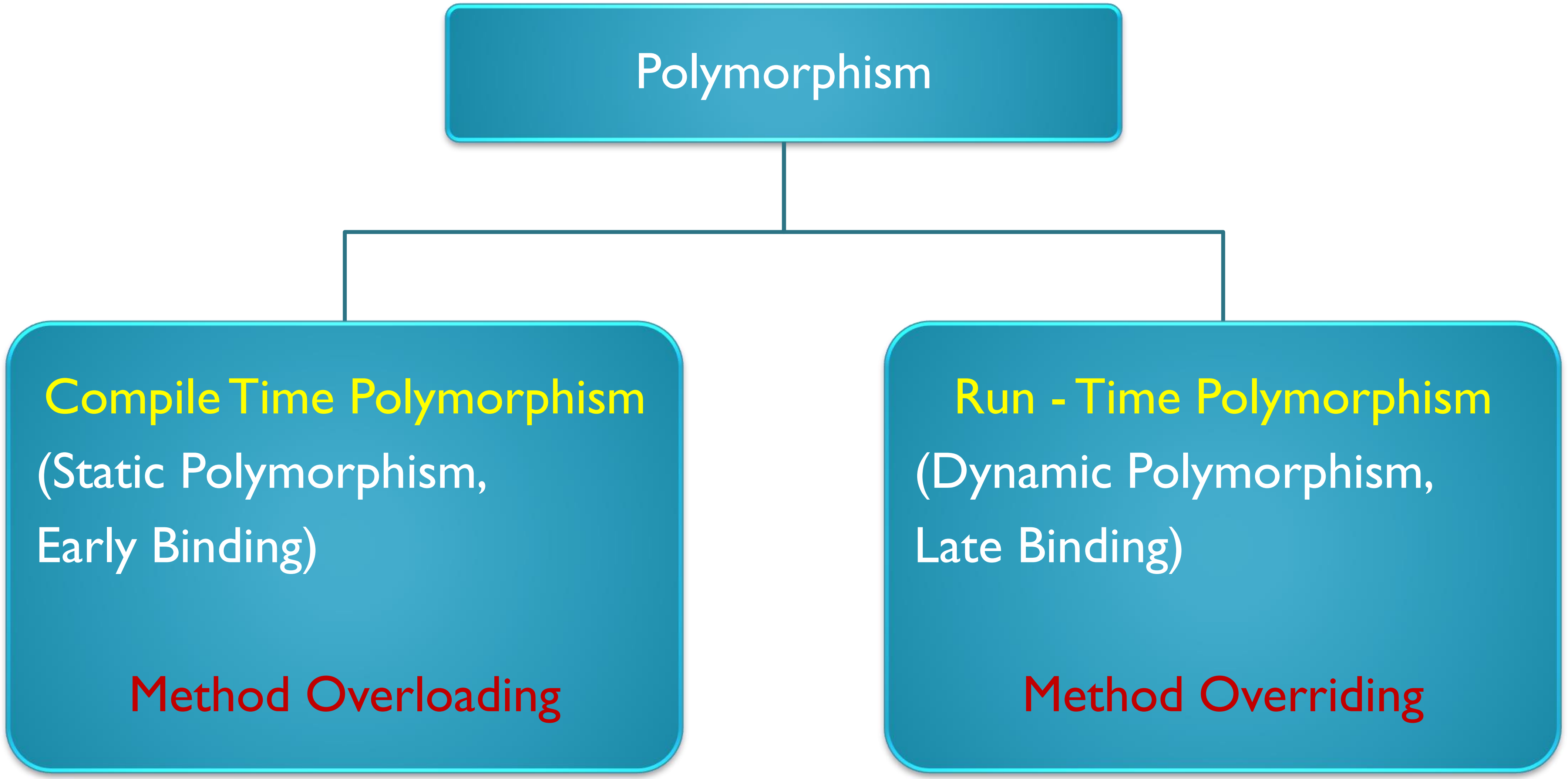
Polymorphism



Polymorphism

- Poly-Morphism-> ability to have multiple forms (shapes) of the same thing.
- Polymorphism is the capability of an action or method to do different things based on the object that it is acting upon.

Polymorphism in JAVA



Polymorphism in JAVA

- In Java, polymorphism is achieved through method overriding and method overloading.
  - Method overriding allows a subclass to provide a different implementation of a method that is already defined in its superclass.
    - This allows objects of different subclasses to respond differently to the same method call, based on their own implementation.
    - Resolved during Run Time
  - Method overloading allows multiple methods with the same name to exist in the same class, as long as they have different parameter lists.
    - This allows the same method name to be used in different contexts, providing a more concise and readable code.
    - Resolved during Compile Time

Method Overloading

Method Overloading

- Method overloading in Java is a technique for creating **multiple methods with the same name within the same class,** as long as they have **different parameter lists**.
- This allows the same method name to be used in different contexts, providing a more concise and readable code.
- If we have to perform only one operation, having same name of the methods increases the readability of the program.
- It is similar to the concept Constructor Overloading in JAVA.

Method Overloading

- There are three ways to overload a method in java,
  - By changing number of arguments

    (Different no of parameters)
  - By changing the data types of arguments

    (Same no of parameters)
  - By changing the sequence of data types of arguments

    (Same no of parameters)
- In java, Method Overloading is **not possible by changing the return type** of the method.

Method Overloading

by changing the no. of arguments

```
class Calculation
{
        void sum(int a,int b)
        {
                System.out.println(a+b);
        }
        void sum(int a,int b,int c){
                System.out.println(a+b+c);
        }

        public static void main(String args[])
        {
                Calculation obj=new Calculation();
                obj.sum(10,10,10);
                obj.sum(20,20);
        }
}
```

Method Overloading

by changing data type of argument

```
class Calculation
{
        void sum(int a,int b)
        {
                System.out.println(a+b);
        }

        void sum(double a,double b)
        {
                System.out.println(a+b);
        }

        public static void main(String[] args)
        {
                Calculation obj=new Calculation();
                obj.sum(10.5,10.5);
                obj.sum(20,20);
        }
}
```

Method Overloading

by changing the sequence of data type of argument

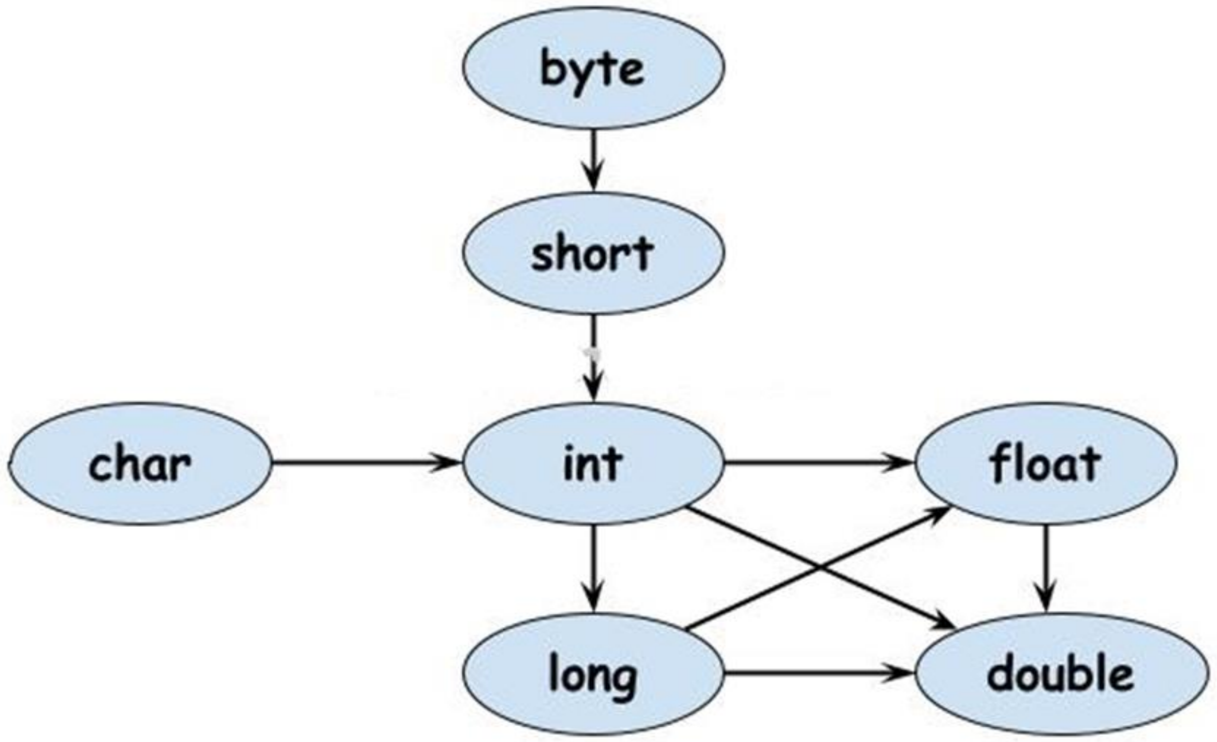
```
class Calculation
{
        void sum(double a,int b)
        {
                System.out.println(a+b);
        }

        void sum(int a,double b)
        {
                System.out.println(a+b);
        }

        public static void main(String[] args)
        {
                Calculation obj=new Calculation();
                obj.sum(10.5,2);
                obj.sum(1,20.5);
        }
}
```

Method Overloading - Highlights

- It's important to note that method overloading in Java is based on the number and types of parameters, not just the name of the method.
- This means that methods with the same name, but different return types are not considered overloaded methods.
- Additionally, methods with the same parameters but different return types are also not considered overloaded methods, as they would result in ambiguity in the code.

Method Overloading - Homework

- What is Type Promotion in java


$$\begin{tikzcd}
 & \text{byte} \arrow[d] & \\
 & \text{short} \arrow[d] & \\
\text{char} \arrow[r] & \text{int} \arrow[r] \arrow[d] \arrow[dr] & \text{float} \arrow[d] \\
 & \text{long} \arrow[r] \arrow[ur] & \text{double}
\end{tikzcd}$$

- What it has to do with method overloading? - Homework

Method Overriding

- An instance method in a subclass
  - **with the same signature** (name, plus the number and the type of its parameters) and
  - **return type**
  - as an instance method in the superclass

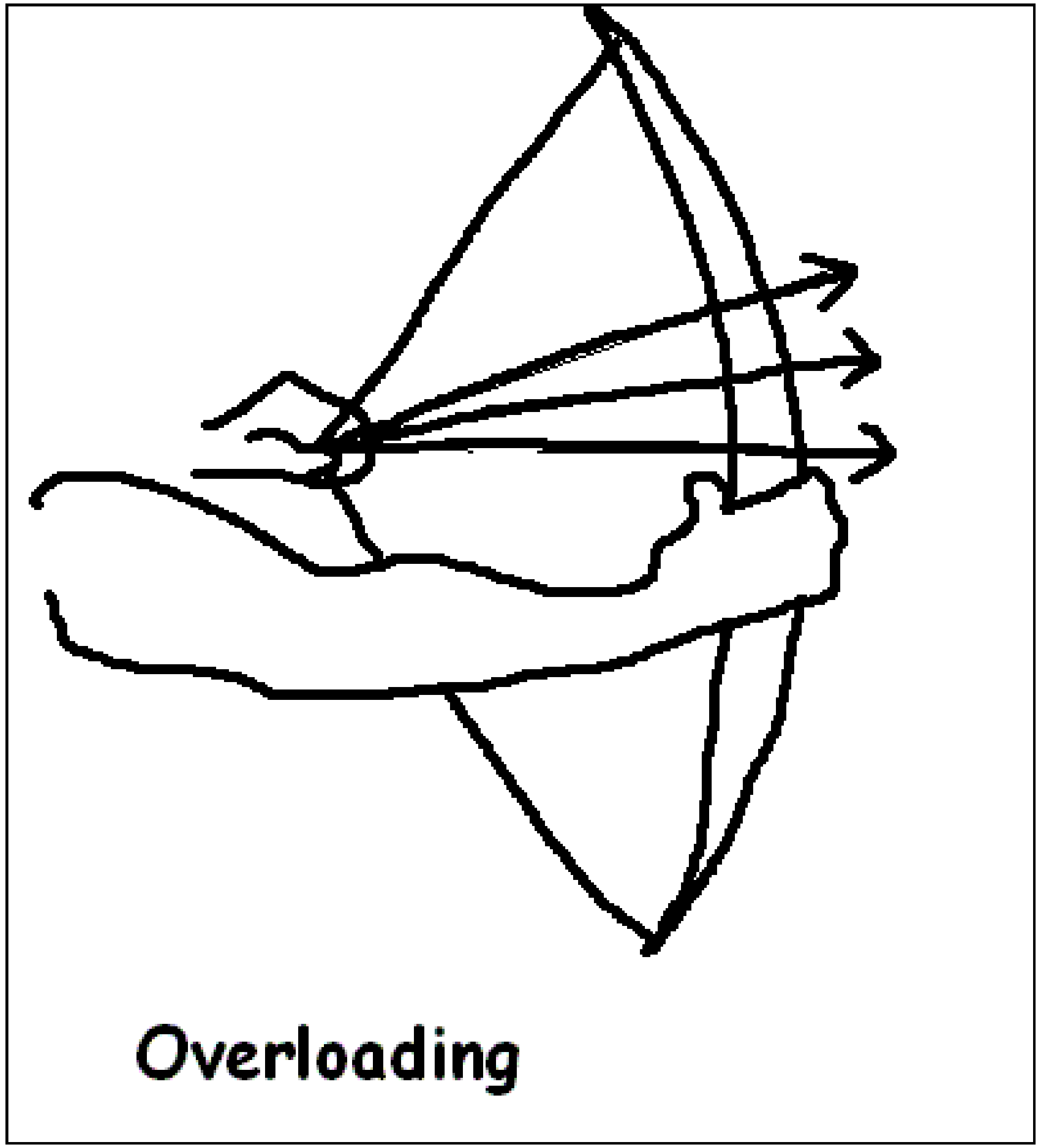
  overrides the superclass's method.

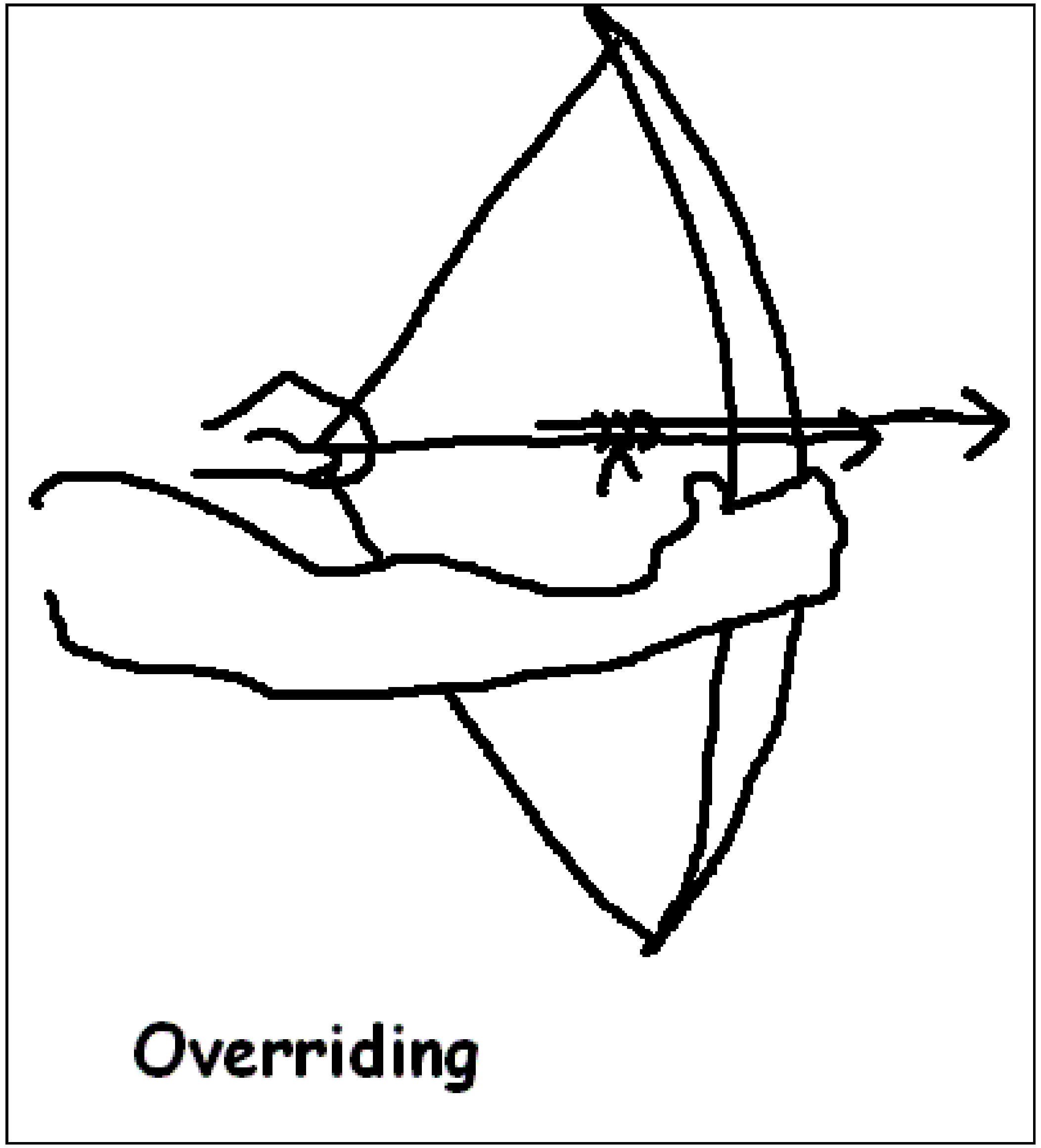
- Use **@Override** annotation

- Refer slide 26 to 30 in “Lesson 03 – OOP Concepts – Part 01”

Method Overriding

```
class Human
{
        public void eat()
        {
                System.out.println("Human is eating");
        }
}
class Boy extends Human
{
        public void eat()
        {
                System.out.println("Boy is eating");
        }
        public static void main( String args[])
        {
                Boy obj = new Boy();
                obj.eat();
        }
}
```


Overloading


Overriding

Dynamic Binding

- Dynamic binding and static binding have to do with inheritance.
- In Java, any derived class object can be assigned to a base class variable.
  - Vehicle v = new Car();
- The variable on the left is type Vehicle, but the object on the right is type Car.
- As long as the variable on the left is a base class of Car, you are allowed to do that.

Dynamic Binding

- Being able to do assignments like that sets up what is called "**polymorphic behavior**": if the Vehicle has a method that is the same as a method in the Car class, then the version of the method in the Car will be called.
  - v.start();
- The version of start() in the Car will be called. Even though you are using a Vehicle variable type to call the method start() , the version of start() in the Vehicle class won't be executed. Instead, it is the version of start() in the Car that will be executed.
- The type of the object that is assigned to the Vehicle variable determines the method that is called.

Dynamic Binding

- When the compiler scans the program and sees a statement like this:

  v.start();

  - it knows that "v" is of type Vehicle,
  - but the compiler also knows that "v" can be a reference to any class derived from Vehicle.
  - Therefore, the compiler doesn't know what version of start() that statement is calling.
  - It's not until the assignment:

    Vehicle v = new Car();

  - is executed that the version of start() is determined. Since the assignment doesn't occur until runtime, it's not until runtime that the correct version of start() is known.

Dynamic Binding

- This is known as "dynamic binding" or "late binding“
- It's not until your program performs some operation at runtime that the correct version of a method can be determined. In Java, most uses of inheritance involve dynamic binding.
- Dynamic binding is deciding at run time which method to invoke.
- With dynamic binding, the method that gets invoked is determined by the class of the object.
- In Java, instance methods (with few exceptions) are dynamically bound.
  - Exceptions are
    - private methods, <init>methods(Constructors), super invocations, final methods

Dynamic Binding – Hands - On

- Add following codes to Demo class and test.

```
Car c = new Car(); //
c.start();

Vehicle v = new Car(); //
v.start();
```

Static Binding

- "Static binding“ or "early binding" occurs when the compiler can readily determine the correct version of something during compile time, i.e. before the program is executed.
- All the instance method calls are always resolved at runtime, but **all the static method calls are resolved at compile time** itself and hence we have static binding for static method calls.
- In Java, **member variables have static binding** because Java does not allow for dynamic binding with member variables.

Static Binding

- If both the Vehicle class and the Car class have a member variable with the same name, then it's the base class version that is used.
- Because the value of member variable is determined in compile time, not in runtime.

Static Binding – Hands - On

- Hands –On
  - Add String color field to both Vehicle (white) and Car (Red)
  - Then check followings in Democlass
    - Car c = new Car(); → c.color // ??
    - Vehicle v = new Car(); → v.color // ??

Summary

- Polymorphism is a fundamental concept in object-oriented programming that allows objects of different classes to respond to the same method call in different ways.
- There are two types of polymorphism in Java:
  - static polymorphism (method overloading) and
  - dynamic polymorphism (method overriding).
- Method overloading
  - is a form of static polymorphism
  - allows multiple methods with the same name to exist in a single class, as long as they have different parameter lists.
  - The method to be called is determined at compile time based on the number and type of arguments passed to the method.
- Method overriding
  - is a form of dynamic polymorphism
  - allows a subclass to provide its own implementation of a method that is already defined in its superclass.
  - The method to be called is determined at runtime based on the actual type of the object, rather than the reference type.

Summary

- Dynamic binding allows
  - objects of different subclasses to respond differently to the same method call, based on their own implementation.
  - This makes the program more flexible and dynamic, as the behavior of the objects can change based on their actual type, rather than being limited by the reference type.

- The @Override annotation is used to indicate that a method in a subclass is intended to override a method in the superclass.
  - This can help to prevent mistakes and improve code readability.

- Polymorphism is a powerful tool for creating more flexible and reusable code. By using polymorphism, it is possible to write code that can handle objects of different types in a generic way, without having to know the exact type of the object.

References

- https://docs.oracle.com/javase/tutorial/java/landI/polymorphism.html

- How To Program (Early Objects)
  - By H .Deitel and P. Deitel
- Headfirst Java
  - By Kathy Sierra and Bert Bates

Questions ???

Thank You