香港浸會大學
HONG KONG BAPTIST UNIVERSITY

COMP2026

Problem Solving Using Object Oriented Programming

Polymorphism

Part of the material refer from Oracle's Java Tutorial.

Overview

- Polymorphism
- @Override
- final method
- final class
- Upcasting and Downcasting

Polymorphism

- The dictionary definition of polymorphism refers to a principle in biology in which an organism or species can have many different forms or stages.
- In Java language, subclasses of a class can define their **own unique behaviors for the same method** and yet share some of the same functionality of the parent class.

Examples of Polymorphism

```java
class Animal {
  public void speak() {}
}
class Cat extends Animal {
  public void speak() { System.out.println("meow"); }
}

class Dog extends Animal {
  public void speak() { System.out.println("woof"); }
}
```

Example of Polymorphism

```java
class USBDevice {
  public void connect(Power p) {
    ...
  }
  public void writeData(byte[] b) {..}
  public byte[] readData() {...}
}
class USBThumb extends USBDevice {
  //read from storage
  public byte[] readData() {..}
  //write to storage
  public void writeData(byte[] b) {..}
}
class USBFan extends USBDevice {
  ..
}
class USBMouse extends USBDevice {
  ..
}
```

image credit: https://www.techrepublic.com/article/how-to-easily-share-usb-devices-using-usb-network-gate/

Example of Polymorphism

```
class Video {
  public void play() {..}
  public void pause() {..}
  public void skip() {..}
}
class AdVideo extends Video {
  public void skip() { return; } //not skippable
}
class LiveVideo extends Video {
  public void play() { playWithChat(); }
}
```

Polymorphism example

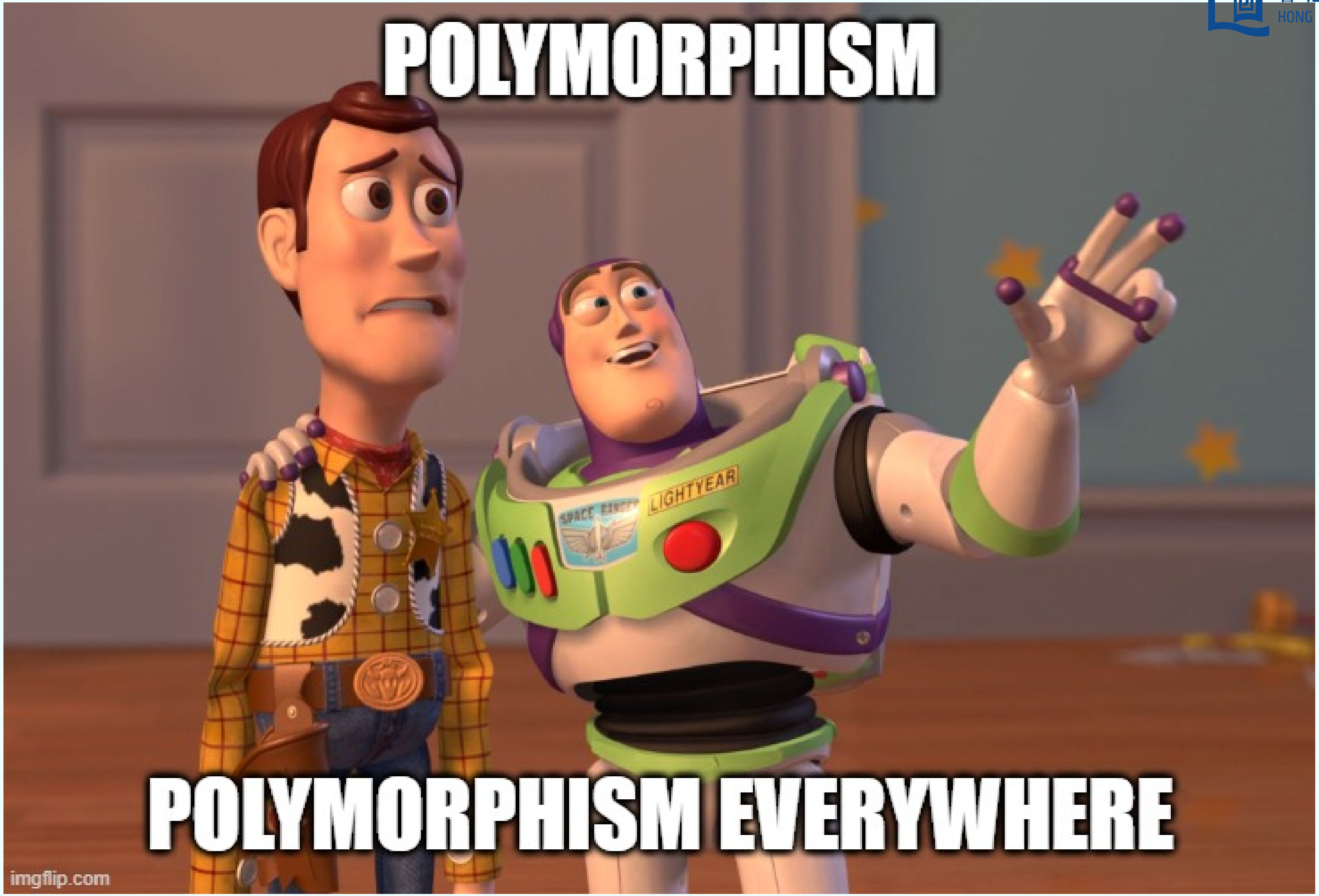
Different type of food eat differently, cook differently, keep differently...

image credit: Pallavi Damera

Polymorphism example

- `void flush()`

image credit: https://www.piqsels.com/en/search?q=toilet&page=2


POLYMORPHISM
POLYMORPHISM EVERYWHERE
LIGHTYEAR
SPACE RANGER
imgflip.com

A concrete example

```java
class Student {
  public void study() { System.out.println("Read book!");  }
}
class MusicStudent extends Student {
  public void study() { System.out.println("Practise Instruement!");  }
}
class CSStudent extends Student {
  public void study() { System.out.println("Code and Math!");  }
}
```

Mother

```
class Mother {
  Student son = ...;
  void eeoo() {
    System.out.println("Son, you need to study hard!");
    son.study();
  }
}
```

- In the method `eeoo`, mother does not really care what subject the son is studying, she just need to push him to study!
- Different instances of son will perform differently!

Mother

```
class Mother {
  Student son = new Student();
  void eeoo() {
    System.out.println("Son, you need to study hard!");
    son.study();
  }
}
```

```
Son, you need to study hard!
Read book!
```

Mother

```
class Mother {
  Student son = new MusicStudent();
  void eeoo() {
    System.out.println("Son, you need to study hard!");
    son.study();
  }
}
```

```
Son, you need to study hard!
Practise Instruement!
```

- `son` has the type of Student, but it is referring to the instance of a `MusicStudent`.
- The method is dynamically binded at run-time.

Mother

```
class Mother {
  Student son = new CSStudent();
  void eeoo() {
    System.out.println("Son, you need to study hard!");
    son.study();
  }
}
```

```
Son, you need to study hard!
Code and Math!
```

- Different types of son respond differently.
- How easy to code on mother with polymorphism!

Polymorphism - the wrong way

```
class Mother {
  Student son = ...
  void badEeoo() {
    System.out.println("Son, it is time to study");
    if (son instanceof CSStudent)
      System.out.println("Why you are not coding");
    else if (son instanceof MusicStudent)
      System.out.println("The instrustment costs me many money, lazy!");
    else
      System.out.println("Your book? Let's do some dictation now!");
  }
}
```

- This mother is so tired.
- Even worst, if we want to add more types of Student later.

🧑‍🏫 💬 Important note: Study is the responsibility of a Student!

Responsibility

- In OO design, object has its own responsibility.
- A class should know what its needs to do.
- Student should bear the responsibility to study. It knows what to study, not the mother!
- In principle, we code against **the general case**
- We **minimize the knowledge** of other classes.
    - Mother should know less about her son.
- Polymorphism allows each student to study on their own, in their own way.

Another example

```
class Student {
  public void doAssignment() {
    System.out.println("Write down the answer on paper");
  }
}
class MusicStudent extends Student {
  public void doAssignment() {
    System.out.println("Listen to some music first");
    System.out.println("Write down the answer on paper");
  }
}
class CSStudent extends Student {
  public void doAssignments() {
    System.out.println("Type in the computer");
  }
}
...
Student s = ...;
s.doAssignment();
```

Some problems in doAssignment

Problem 1. Code Repeated in Music Student

```
class MusicStudent extends Student {
  public void doAssignment() {
    System.out.println("Listen to some music first");
    System.out.println("Write down the answer on paper");
  }
}
```

- How come this is a problem? Just a copy-and-paste
- We need to minimize the number of code repeated

Problem 2. CSStudent's output is incorrect

```
Write down the answer on paper
```

Solution to Problem 1.

```
class MusicStudent extends Student {
  public void doAssignment() {
    System.out.println("Listen to some music first");
    super.doAssignment();
  }
}
```

- Call `doAssignment()` method of the superclass
- Imagine Student's `doAssignment()` involves many complicate instructions and fields.
- There is a good opportunity that you can make these fields private (if they are not used anywhere in the subclass of Students).

> 🧑‍🏫 💬 Take away: Try to reuse superclass's method!

More on reusing

```java
class superclass {
  void method() {
    { //Block A
      //Code that does not
      //repeat in subclass
    }
    { //Block B
      //Code that repeat
    }
  }
}
```

```java
class subclass extends superclass {
  void method() {
    { //Block B
      //Code that repeat
    }
    { //Block C
      //Additional Code
      //in subclass
    }
  }
}
```

- In this case directly calling parent method is not possible.
- Refactor the code to extract Block B as a protected method.

More on reusing

```
class superclass {
  protected void B() { ... }
  void method() {
    { //Block A
      //Code that does not
      //repeat in subclass
    }
    B();
  }
}
```

```
class subclass extends superclass {
  void method() {
    B();
    { //Block C
      //Additional Code
      //in subclass
    }
  }
}
```

🧑‍🏫 💬 Take away: Call your parent class more often!

Problem 2

```java
class Student {
  public void doAssignment() {
    System.out.println("Write down the answer on paper");
  }
}
class CSStudent extends Student {
  public void doAssignments() {
    System.out.println("Type in the computer");
  }
}
```

```
Write down the answer on paper
```

- Reason?

Problem 2

```
class CSStudent extends Student {
  public void doAssignments() {
    System.out.println("Type in the computer");
  }
}
```

- Compiler cannot differentiate you are spelling it wrong or just want to add a new method
- The annotation `@Override` helps you to label a method that is designed for override
- Place the annotation `@Override` one-line above **subclass method**.
- `@Override` is optional, but useful.

▶

Solution to Problem 2

- This work the same without `@Override`.

```java
class CSStudent extends Student {
  @Override
  public void doAssignment() { System.out.println("Type in the computer"); }
}
```

❌ This will not compile.

```java
class CSStudent extends Student {
  @Override
  public void doAssignments() { //typo
    System.out.println("Type in the computer");
  }
}
```

🧑‍🏫 💬 Take away: Make a habit to use `@Override` in all overriden methods.

Cheating

```
class Student {
  public boolean isCheating() {
    //I don't cheat, so return false
    return false;
  }
}

...
Student son = ...;
if (son.isCheating())
  System.out.println("Impossible, my son would not cheat!");
```

- Is it really impossible?

Unwanted overriding

```
class Student {
  public boolean isCheating() {
    //I don't cheat, so return false
    return false;
  }
}
class CSStudent extends Student {
  @Override
  public boolean isCheating() {
    return true;
  }
}
```

- Oh.

Unwanted overriding

- We can use `final` to stop further overriding.
- A method labeled with `final` cannot be overriden by its subclasses.

```java
class Student {
  public final boolean isCheating() {
    return false;
  }
}
```

❌ This would not compile

```java
class CSStudent extends Student {
  @Override
  public boolean isCheating() {
    return true;
  }
}
```

final in multilayer hierarchy

```java
class Human {
  public boolean isCheating() {
    return true;
  }
}
class Student extends Human {
  @Override
  public final boolean isCheating() {
    return false;
  }
}
```

- Student add `final` to `isCheating()` stop its subclass to further overriding it.
- While it's siblings (other child of Human) may override it.

final class

- When a class is labeled as `final`, it cannot be further inherited

```java
final class BCDAStudent extends CSStudent {

}
```

❌ This would not compile

```java
class MinorBCDAStudent extends BCDAStudent {

}
```

The final keyword

- `final` in front of a field to indicate this value cannot be changed.

```java
final String name;
```

- `final` in front of a static variable to indicate this is a constant, to avoid hardcode.
  - All CAPITAL letters by convention.

```java
public static final int ROWS_FOR_SUDOKU = 9;
```

- `final` in front of a method indiciates this cannot be further overriden.
- `final` in front of a class indiciates this class cannot be further inherited.

> 🧑‍🏫 💬 Take away: `final` method stops further overriding, `final` class stops further inheritance.

Revisit Casting

- Type casting changes the data type of a value from its normal type to some other type.

Two type of casting:

- Widening (automatic): changes a smaller type to a bigger/more precise type
  - byte ➡ short ➡ char ➡ int ➡ long ➡ float ➡ double
- Narrowing (manual): changes a bigger/more precise type to a smaller type
  - double ➡ float ➡ long ➡ int ➡ char ➡ short ➡ byte

Widening

```
float f = 1.2345f; //to specify a number literal as float, add f after it
double d;
d = f;
```

- The value 1.2345 will be stored in double without any precision lost.
- No problem will happen for sure.

```
int i = 439234;
long l;
l = i;
```

- The variable `l` has a type `long` which support a larger range than `int`.
- No problem will happen for sure.

Narrowing

```
double d = 1.23456;
float f;
f = d; //error!
```

- The assign has an error because it is possible that some digits in `d` can't be stored in `f`
- **Lost of precision**

```
long l = 123456789;
int i;
i = l; //error!
```

- It is possible that `l` has a value large than what `int` can support (± 2147483647)

Narrowing

- You can suppress the error by casting if you are sure the value are compatible

```
double d = 1.23456;
float f;
f = (float) d; //casting
```

```
long l = 123456789;
int i;
i = (int) l; //casting
```

- Both examples compile

Narrowing

- However, what happen if the value is *incompatible*?

```
double d = 1.23456789123456789;
float f = (float) d;
System.out.println(d + ":" + f);
```

```
1.2345678912345678:1.2345679
```

- Things get worst for integer

```
int i = 1234567;
short s = (short) i; //short support -32768 to 32767
System.out.println(i + ":" + s);
```

```
123456:-7616
```

Casting in OO

- Similar concept but different terminology here
- Suppose we have the class hierarchy

(superclass) A ⬅ B ⬅ C ⬅ D (subclass)

```
A objA = new __?__();
```

- We can fill A/B/C/D in (?)
- This is automatic.
- This is called **Upcasting**.
- Upcasting is always safe.

Casting in OO

(superclass) A ⬅ B ⬅ C ⬅ D (subclass)

```
A objA = new C();
```

- Now we see `objA` is holding `C`'s instance.
- ❌ But this does not compile

```
objA.cMethod();
```

- The compiler just don't know if this is a `C` or not in compile time.

Casting in OO

(superclass) A ⬅ B ⬅ C ⬅ D (subclass)

❌ Similarly this does not work

```
A objA = new C();
C objC = objA;
```

- The compiler just don't know if this is a `C` or not in compile time.

Casting in OO

(superclass) A ⬅ B ⬅ C ⬅ D (subclass)

- If you are 100% sure that `objA` is pointing to a `C` object, you can perform **downcasting**.
- Downcasting means to force this object as a specific subclass.
- Downcasting is not automatic, must be done explicitly.
- Downcasting can throw `ClassCastException`.

```
A objA = new C();
C objC = (C)objA;
((C)objA).cMethod();
```

Casting in OO

To check the class of an object in run-time, use `instanceof`

```java
A objA = new C();
if (objA instanceof C) {
  C objC = (C)objA;
  ((C)objA).cMethod();
  ...
}
```

> 🧑‍🏫 💬 Most of the time our students will misuse downcasting! Rethink carefully if this can be done via polymorphism first.

The Object class

- There is a class called `Object`.

```
Object obj = new Object();
```

- This `Object` class is the superclass of all Java classes

The Object class

- Don't be confuse with an object.
- This is a class, just its name is called `Object.`
- All classes, even if you do not declare explicitly, it inherits object anyway.

```
public class MyClass {}
```

is same as

```
public class MyClass extends Object {}
```

The Object class

`Object` class has the following methods.

Modifier	Return Type	Method Name
protected	Object	`clone()`
public	boolean	`equals(Object obj)`
protected	void	`finalize()`
public	Class<?>	`getClass()`
public	int	`hashCode()`
public	void	`notify()`
public	void	`notifyAll()`
public	String	`toString()`
public	void	`wait()`
public	void	`wait(long timeout)`
public	void	`wait(long timeout, int nanos)`

The Object class

> 🧑‍🏫 💬 Recall we can actually call `toString()` and `equals(Object obj)` method! Because your father has it!

```java
public class MyClass { /* empty */ }
public class MyClassWithtoString {
  @Override
  public String toString() { return "override toString!";}
}
System.out.println(new MyClass());
System.out.println(new MyClassWithtoString());
```

```
MyClass@1b6d3586
override toString!
```

The Object class

- Since all objects are in fact an Object class, you can actually store different object inthe same List!

```
List<Object> handbag = new ArrayList<>();
handbag.add(myString);
handbag.add(myContact);
handbag.add(myPhone);
handbag.add(myLipstick);
```

- But it creates a problem when you want to retrieve it

```
Contact c = handbag.get(1); //error!
Phone p = handbag.get(2); //error!
Object lipstick = handbag.get(3); //OK
```

The Object class

- Downcasting is needed in this case

```java
List<Object> handbag = new ArrayList<>();
handbag.add(myString);
handbag.add(myContact);
handbag.add(myPhone);
handbag.add(myLipstick);

...

Contact c = (Contact)handbag.get(1);
Phone p = (Phone)handbag.get(2);
Phone s = (Phone)handbag.get(0); //crash
```

> 🧑‍🏫 💬 At the first place `handbag` is a bad design!

Errata

- Page 26,27 CSStudent should `extends` Student