

Inheritance and Polymorphism in Java

Subclasses and Superclasses

- If class B **extends** class A, then class B is a **SUBCLASS** of class A.
- Class A is Class B's **SUPERCLASS**.
- A class cannot extend more than one class. However, if class B extends class A and class C extends class B, then C is a subclass of both A and B.

Inheritance

- Subclasses **inherit** all of the methods and variables of the super class
- **Private variables and methods** are not directly accessible – must be accessed through **public accessor and mutator methods**
- **Constructors** are never inherited
- Inheritance goes all the way up the tree. If class B extends class A, and class C extends class B, then C inherits everything from A and B.

Inheritance

- A subclass can add new instance variables and methods.
- A subclass can also **override** inherited methods.

```
public class Performer {  
    private String name;  
    private int age;  
  
    public Performer() {  
        name = "Ima Performer";  
        age = 16;  
    }  
    public Performer(String n, int a) {  
        name = n;  
        age = a;  
    }  
    public String getName(){ ... }  
    public void practice(){ ... }  
    public void perform(){ ... }  
}
```

```
public class Musician extends Performer{  
    private String instrument;  
  
    public Musician() {  
        instrument = "Piano";  
    }  
  
    public Musician(String inst) {  
        instrument = inst;  
    }  
  
    public String getInstrument(){ ... }  
    public void playInstrument(){ ... }  
}
```

```
public class Performer {  
    public Performer(){... }  
    public Performer(String n, int a){... }  
}
```



```
public class Musician extends Performer{  
    public Musician(){ ... }  
    public Musician(String inst){ ... }  
}
```

Which of the following will compile?

```
Performer wynton = new Performer();
```

```
Musician branford = new Musician();
```

```
Performer wynton = new Performer("Wynton", 19);
```

```
Musician branford = new Musician("Branford", 20);
```


```
Musician branford = new Musician("Saxophone");
```

```
Performer wynton = new Performer("Trumpet");
```

```
Musician branford = new Musician("Branford", 20, "Saxophone");
```

```
public class Performer {  
    public Performer(){... }  
    public Performer(String n, int a){... }  
}
```

```
public class Musician extends Performer{  
    public Musician(){ ... }  
    public Musician(String inst){ ... }  
}
```



Performer wynton = new Performer(); ✓

Musician branford = new Musician(); ✓

Performer wynton = new Performer("Wynton", 19); ✓

Musician branford = new Musician("Branford", 20); ✗

Musician branford = new Musician("Saxophone"); ✓

Performer wynton = new Performer("Trumpet"); ✗


Musician branford = new Musician("Branford", 20, "Saxophone"); ✗

Implicit Calls to Default Constructors

A subclass's constructor **implicitly** calls the default constructor of its super class

```
public class Performer {  
    private String name;  
    private int age;  
  
    public Performer() {  
        name = "I'm a Performer";  
        age = 16;  
    }  
    public Performer(String n, int a) {  
        name = n;  
        age = a;  
    }  
    public String getName(){ ... }  
    public void practice(){ ... }  
    public void perform(){ ... }  
}
```

```
public class Musician extends Performer{  
    private String instrument;  
  
    public Musician() {  
        instrument = "Piano";  
    }  
    public Musician(String inst) {  
        instrument = inst;  
    }  
  
    public String getInstrument(){ ... }  
    public void playInstrument(){ ... }  
}
```



SUPER Keyword

* The super keyword can be used to **explicitly** call the superclass's constructors.

- This is very useful because you can't access private instance variables declared in your superclass!

- To call a super class's constructor from a subclass's constructor you use it like:

`super();` or `super(x, y, z);`

- **Nothing can precede this code!**

SUPER Keyword

- For example, you could make a musician constructor that specifies name, age, and instrument as follows:

```
public Musician (String name, int age, String instr) {  
    super(name, age);  
    this.instrument = instr;  
}
```

But can you do this?

```
public Musician (String name, int age, String instr) {  
    this.name = name;  
    this.age = age;  
    this.instrument = instr;  
}
```

Person

```
graph TD; Person --> Student; Person --> Employee; Student --> GradStudent; Student --> UnderGrad;
```

Student

Employee

GradStudent

UnderGrad

Declaring Subclass Objects

- Subclasses are considered instances of their superclasses as well as instances of themselves.

Example: a GradStudent object is a Student and a Person object as well. A Person object is NOT a Student, however.

Declaring Subclass Objects

Which of these will result in an error?

- a) `Person a = new Student();`
- b) `Person b = new Person();`
- c) `Student c = new UnderGrad();`
- d) `Student d = new Person();`
- e) `GradStudent e = new Student();`

Subclass Objects

These will result in an error!

- a) `Person a = new Student();`
- b) `Person b = new Person();`
- c) `Student c = new UnderGrad();`
- d) `Student d = new Person();`
- e) `GradStudent e = new Student();`

The variable's type must be at the same level or higher up the tree than the object it contains

Subclass Objects

Why might you want to do this?

```
Person[] people = new Person[10];
```

```
People[0] = new Student();
```

```
People[1] = new UnderGrad();
```

```
People[2] = new GradStudent();
```

```
People[3] = new Person();
```

This flexibility causes all sorts of ambiguous problems that are solved by Overriding, Polymorphism, Dynamic Binding, the SUPER keyword, and Downcasting!

Overriding

- When a subclass has a method that is the same name as a method in its super class
- You cannot override private or static methods.
- What method is being overridden in our examples?

Overriding vs. Overloading

- Easy to confuse!
- Overloading: When two methods in a class have the same name, but different parameter lists. Evaluated at compile time (Static Binding)
- Overriding: When a subclass has the same method as its super class. Evaluated at runtime (Dynamic Binding)

Polymorphism

Polymorphism describes the mechanism Java uses to choose which overridden method to call.

```
Student a = new Student();  
Student b = new UnderGrad();  
Student c = new GradStudent();
```

Which `calculateGrade()` method is called?

```
a.calculateGrade();  
b.calculateGrade();  
c.calculateGrade();
```

Polymorphism

It is based on the type of the actual object, not the type of the variable.

```
Student a = new Student();  
Student b = new UnderGrad();  
Student c = new GradStudent();
```

```
a.calculateGrade(); // uses Student's version  
b.calculateGrade(); // uses UnderGrad's version  
c.calculateGrade(); // uses GradStudent's version
```

Even though the variables a, b, and c are of type student.

Polymorphism

How about this example?

```
Student      a = new GradStudent();  
GradStudent b = new GradStudent();
```

```
int id1 = a.getID();  
int id2 = b.getID();
```

What happens, and why?

Polymorphism

How about this example?

```
Student      a = new GradStudent();  
GradStudent b = new GradStudent();
```

```
int id1 = a.getID(); //Doesn't compile  
int id2 = b.getID(); //Works fine
```

Why doesn't this work? `getID` isn't overridden, so polymorphism doesn't apply!

Downcasting

But if you want to get the ID of a, you can **Downcast!** This means casting down the tree (casting a superclass to a subclass).

```
Student a = new GradStudent();
```

```
int ID1 = ((GradStudent) a).getID();
```

Dynamic Binding

In Java, polymorphism happens at runtime not compile time. The term “Dynamic Binding” or “Late Binding” java determines which overridden method to call at run time rather than at compile time:

```
public void calc (Student s) {  
    s.calculateGrade();  
}
```

Static Binding

Also called “Early Binding”, java determines which overloaded method to call at compile time rather than at run time:

```
Fraction f1 = new Fraction();  
Fraction f2 = new Fraction(2,3);  
Fraction f3 = new Fraction("2/3");  
Fraction f4 = new Fraction(f3);
```


SUPER Keyword II

- To call methods specifically from the super class (mostly used to differentiate overridden methods), you can use it like this:

```
super.calculateGrade();
```

- An overridden method that uses the super keyword is said to be “partially overridden.”

SUPER Keyword II

- An overridden method that uses the super keyword is said to be “partially overridden.”
- If a constructor does not explicitly include a call to super(), it is implicitly included.

The Object Class

- The universal Superclass!
- All classes implicitly extend the Object class
- Includes:

```
public String toString()  
public boolean equals()
```

- Can be used to collect anything and everything! Consider:

```
Object[] objects;  
public void foo (Object obj) {}
```