COSC 121: Computer Programming II

Dr. Bowen Hui

University of British Columbia Okanagan

Recall use of +

- Consider the following +:
  - 4 + 5
  - 3.14 + 2.9
  - str1 + “bar”
  - System.out.println(“the value is: “ + n);
- What does + mean in each case?
- How to achieve this?

Defining +

- How to achieve this?

```
public int +( int x, int y )
{ …
}
public double +( double x, double y )
{ …
}
```

What does this remind you of?
Note: different parameter lists

Applying to Other Methods

- What if related classes have methods of the same signature?

```
public String talk()
{
  return “beh”;
}
public String talk()
{
  return “meow”;
}
```

- Depending on caller object, different talk() would be called

Polymorphism

- Polymorphism is an OOP technique that allows us to reference different object types at different points in time
  - Literal meaning “having many forms”
- Achieved via inheritance or interface relationships

Late Binding

- Example:

```
myPet.talk();
```

- This call is bound to the definition of the method that it invokes
- If this binding occurred at *compile time*, then that line of code would call the same method every time
- However, Java defers method binding until *run time* – so it delays binding until as late as possible
  - This approach is called dynamic binding or late binding

Essence of Polymorphism

Representative example:

```
Animal[] myPets = new Animal[4];
myPets[0] = new Dog();
myPets[1] = new Cat();
myPets[2] = new Sheep();
myPets[3] = new Cow();
for( int i=0; i<myPets.length; i++ )
{
  System.out.println( myPets[i].talk() );
}
```

strange?

Essence of Polymorphism

Representative example:

```
Animal[] myPets = new Animal[4];
myPets[0] = new Dog();
myPets[1] = new Cat();
myPets[2] = new Sheep();
myPets[3] = new Cow();
for( int i=0; i<myPets.length; i++ )
{
  System.out.println( myPets[i].talk() );
}
```

`myPets` is an array of `Animal` objects

Essence of Polymorphism

Representative example:

```
Animal[] myPets = new Animal[4];
myPets[0] = new Dog();
myPets[1] = new Cat();
myPets[2] = new Sheep();
myPets[3] = new Cow();
for( int i=0; i<myPets.length; i++ )
{
  System.out.println( myPets[i].talk() );
}
```

Output

```
Woof
Meow
Beh
Moo
```

Essence of Polymorphism

Representative example:

```
Animal[] myPets = new Animal[4];
myPets[0] = new Dog();
myPets[1] = new Cat();
myPets[2] = new Sheep();
myPets[3] = new Cow();
for( int i=0; i<myPets.length; i++ )
{
  System.out.println( myPets[i].talk() );
}
```

Output

```
Woof
Meow
Beh
Moo
```

at i=0: talk() is the method from **Dog** class

Essence of Polymorphism

Representative example:

```
Animal[] myPets = new Animal[4];
myPets[0] = new Dog();
myPets[1] = new Cat();
myPets[2] = new Sheep();
myPets[3] = new Cow();
for( int i=0; i<myPets.length; i++ )
{
  System.out.println( myPets[i].talk() );
}
```

Output

```
Woof
Meow
Beh
Moo
```

at i=1: talk() is the method from **Cat** class

Essence of Polymorphism

Representative example:

```
Animal[] myPets = new Animal[4];
myPets[0] = new Dog();
myPets[1] = new Cat();
myPets[2] = new Sheep();
myPets[3] = new Cow();
for( int i=0; i<myPets.length; i++ )
{
  System.out.println( myPets[i].talk() );
}
```

Output

```
Woof
Meow
Beh
Moo
```

at i=2: talk() is the method from **Sheep** class

Essence of Polymorphism

Representative example:

```
Animal[] myPets = new Animal[4];
myPets[0] = new Dog();
myPets[1] = new Cat();
myPets[2] = new Sheep();
myPets[3] = new Cow();
for( int i=0; i<myPets.length; i++ )
{
  System.out.println( myPets[i].talk() );
}
```

Output

```
Woof
Meow
Beh
Moo
```

at i=3: talk() is the method from **Cow** class

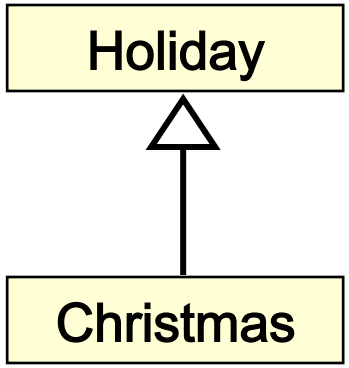
Polymorphism

- Recall: Polymorphism refers to different object types at different points in time
  - Done via a polymorphic reference – a variable that refers to different types
- Recall: Achieved via inheritance or interface relationships
  - In example, Dog, Cat, Sheep, Cow are must all either `extends` or `implements` Animal

Polymorphism via Inheritance

- An object reference can refer to an object of any class related to it by inheritance

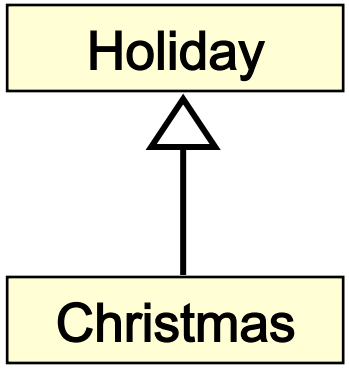
- For example, if `Holiday` is the superclass of `Christmas`, then a `Holiday` reference could be used to refer to a `Christmas` object




```
Holiday day;
day = new Christmas();
```

Polymorphism via Inheritance

- An object reference can refer to an object of any class related to it by inheritance
- For example, if `Holiday` is the superclass of `Christmas`, then a `Holiday` reference could be used to refer to a `Christmas` object




```
Holiday day;
day = new Christmas();
```

Okay to assign `Christmas` object to a `Holiday` reference because Christmas *is-a* Holiday

References and Inheritance

- Type compatibility rules are part of IS-A relationship established by inheritance
- To assign child object to parent reference:
  - Just do a simple assignment (=)
- To assign parent object to child reference:
  - Must use casting
  - Not recommended in practice
  - After all, not all holidays are Christmases

Method Invocation

- Suppose `Holiday` class has `celebrate()`, and `Christmas` overrides it
- Which method is invoked when:

```
day.celebrate();
```

Method Invocation

- Suppose `Holiday` class has `celebrate()`, and `Christmas` overrides it
- Which method is invoked when:

```
day.celebrate();
```

- Depends on what `day` is at time of method call
  - If `day` refers to a `Holiday` object, invoke the `Holiday` definition of `celebrate()`
  - If `day` refers to a `Christmas` object, invoke the `Christmas` definition of `celebrate()`

How it works

- Compiler restricts the method invocations based on the reference type
- Suppose `Christmas` had `getTree()` but `Holiday` didn't have this method, then:

```
day.getTree();  // compiler error
```

- Because the compiler doesn't "know" which type of `Holiday` object is being referenced

Calling Children Methods

- Compiler can only guarantee calls to methods defined within `Holiday` class
- One way to solve it is instead of:

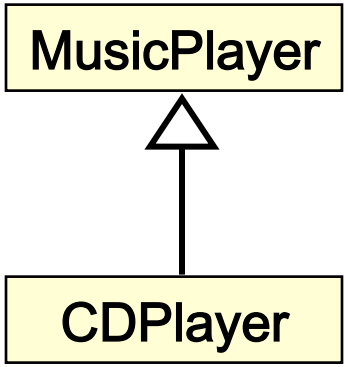
```
day.getTree();  // compiler error
```

- Write:

```
(( Christmas ) day).getTree();
```

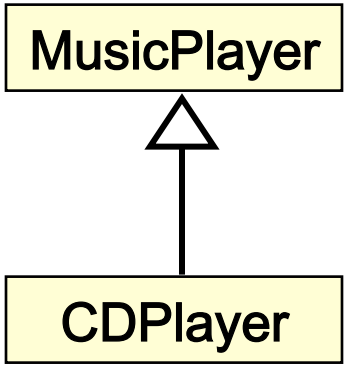
- Only if you're absolutely sure `day` is a `Christmas` object

Another Example



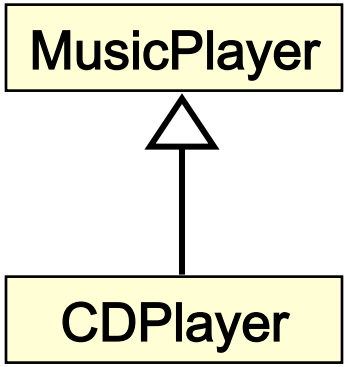
- Given the diagram, are the following valid?
  - `MusicPlayer mplayer = new CDPlayer();`
  - `CDPlayer cdplayer = new MusicPlayer();`

Another Example



- Given the diagram, are the following valid?
  - `MusicPlayer mplayer = new CDPlayer();`
  - Yes: `CDPlayer` is-a `MusicPlayer`
  - `CDPlayer cdplayer = new MusicPlayer();`

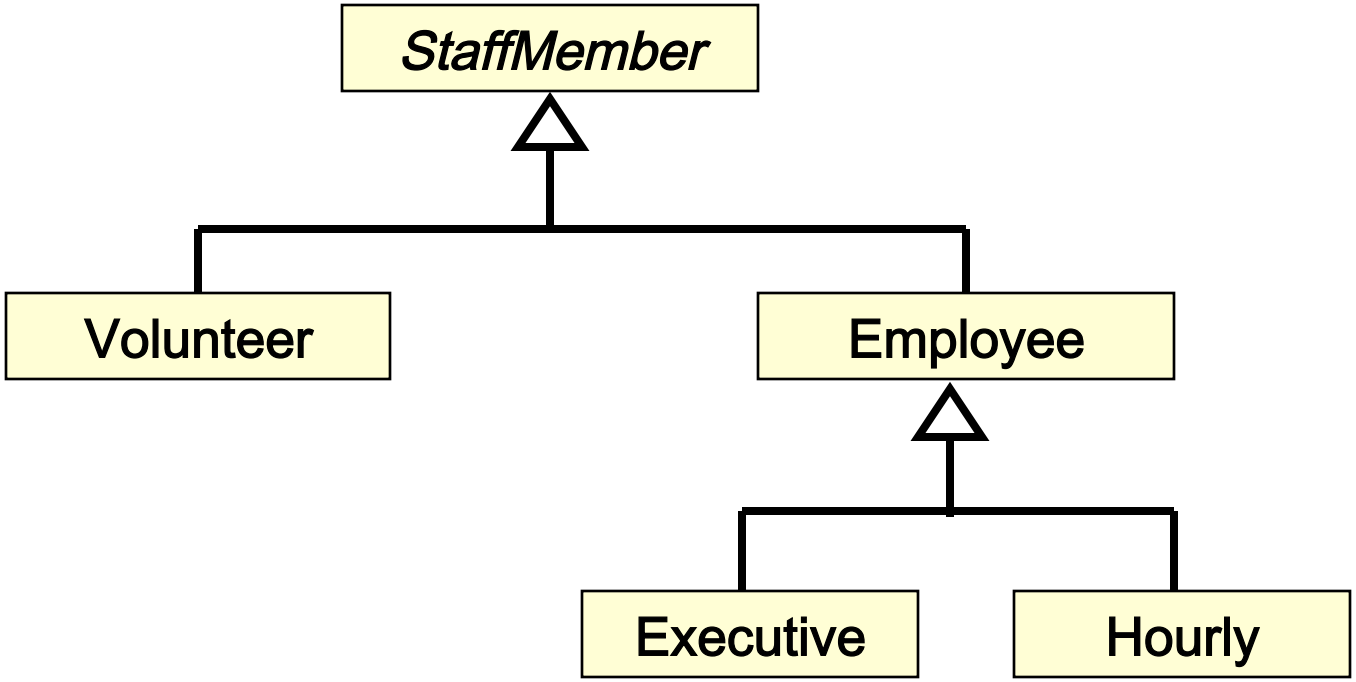
Another Example



- Given the diagram, are the following valid?
  - `MusicPlayer mplayer = new CDPlayer();`
  - Yes: `CDPlayer` is-a `MusicPlayer`
  - `CDPlayer cdplayer = new MusicPlayer();`
  - No: not all `MusicPlayer` are `CDPlayer`
  - You could force this to work by casting

Example from Text

- Given partial class hierarchy:



```java
abstract public class StaffMember
{
    protected String name;
    protected String address;
    protected String phone;

    public StaffMember( String eName, String eAddress, String ePhone )
    {
        name = eName;
        address = eAddress;
        phone = ePhone;
    }

    public String toString()
    {
        String result = "Name: " + name + "\n”;
        result += "Address: " + address + "\n";
        result += "Phone: " + phone;
        return result;
    }

    public abstract double pay();
}
```

Why bother?

```java
public class Volunteer extends StaffMember
{
   public Volunteer( String eName, String eAddress, String ePhone )
   {
      super( eName, eAddress, ePhone );
   }

   // doesn't override toString()

   public double pay()
   {
      return 0.0;
   }
}
```

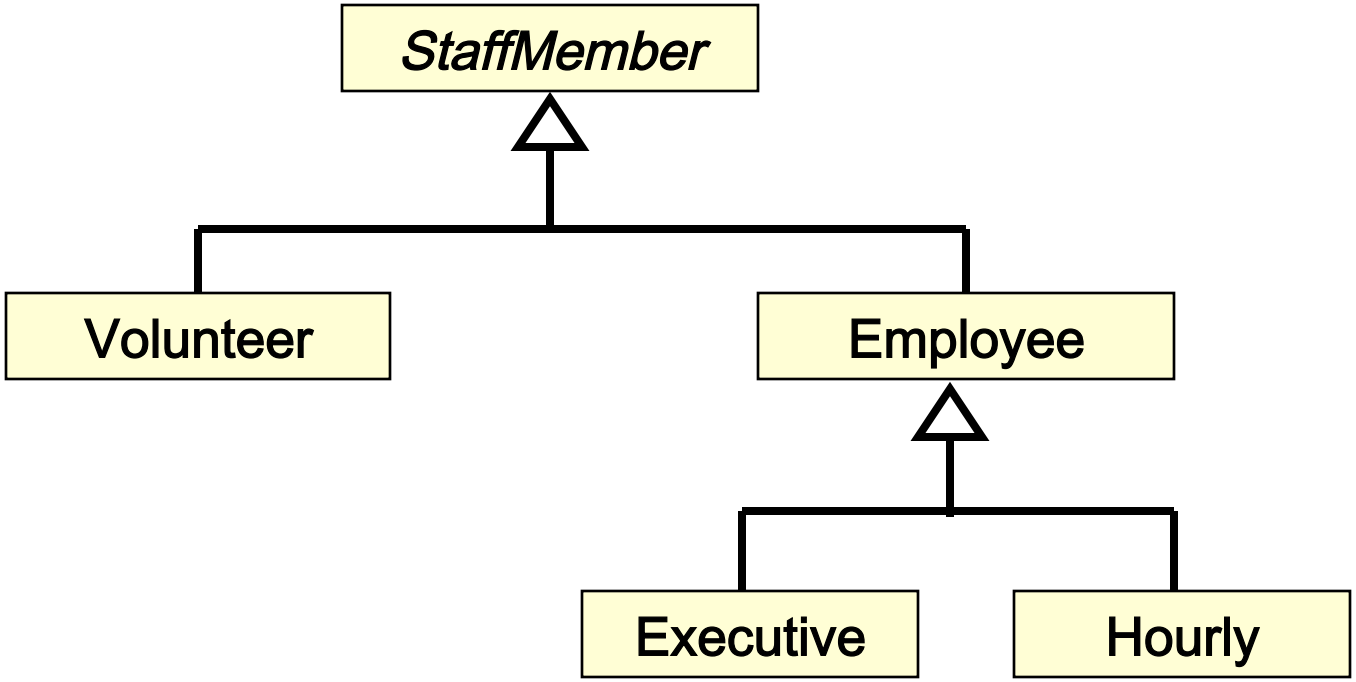
What does this do?

```java
public class Employee extends StaffMember
{
   // additional attributes
   protected String ssn;
   protected double payRate;

   public Employee( String eName, String eAddress, String ePhone,
                    String socSecNumber, double rate )
   {
      super( eName, eAddress, ePhone );
      ssn = socSecNumber;
      payRate = rate;
   }

   public String toString()
   {
      String result = super.toString();
      result += "\nSocial Security Number: " + ssn;
      return result;
   }

   public double pay() { return payRate; }
}
```

What does this do?

Example from Text

- Recall class hierarchy:



```java
public class Executive extends Employee
{
   private double bonus;

   public Executive( String eName, String eAddress, String ePhone,
                     String socSecNumber, double rate )
   {
      super( eName, eAddress, ePhone, socSecNumber, rate );
      bonus = 0;  // bonus has yet to be awarded
   }

   public void awardBonus( double execBonus )
   {
      bonus = execBonus;
   }

   // doesn't override toString()

   public double pay()
   {
      double payment = super.pay() + bonus;
      bonus = 0;
      return payment;
   }
}
```

```java
public class Hourly extends Employee
{
   private int hoursWorked;

   public Hourly (String eName, String eAddress, String ePhone,
                  String socSecNumber, double rate)
   {
      super (eName, eAddress, ePhone, socSecNumber, rate);
      hoursWorked = 0;
   }

   public void addHours( int hours ) { hoursWorked += hours; }

   public double pay()
   {
      double payment = payRate * hoursWorked;
      hoursWorked = 0;
      return payment;
   }

   public String toString()
   {
      String result = super.toString();
      result += "\nCurrent hours: " + hoursWorked;
      return result;
   }
}
```

```
public class Staff // manages all StaffMembers
{
   private StaffMember[] staffList;

   public Staff ()
   {
      staffList = new StaffMember[6];
```

Is this valid?

continue

Are we instantiating `StaffMember` objects?

continue in Staff class

instantiations

```
    staffList[0] = new Executive ("Sam", "123 Main Line",
       "555-0469", "123-45-6789", 2423.07);

    staffList[1] = new Employee ("Carla", "456 Off Line",
       "555-0101", "987-65-4321", 1246.15);
    staffList[2] = new Employee ("Woody", "789 Off Rocker",
       "555-0000", "010-20-3040", 1169.23);

    staffList[3] = new Hourly ("Diane", "678 Fifth Ave.",
       "555-0690", "958-47-3625", 10.55);

    staffList[4] = new Volunteer ("Norm", "987 Suds Blvd.",
       "555-8374");
    staffList[5] = new Volunteer ("Cliff", "321 Duds Lane",
       "555-7282");

    (( Executive )staffList[0]).awardBonus( 500.00 );

    (( Hourly )staffList[3]).addHours( 40 );
}
```

why is casting needed?

continue in Staff class

continue in Staff class

```
   //  Pays all staff members.
   public void payday ()
   {
      double amount;

      for( int count=0; count < staffList.length; count++ )
      {
         System.out.println( staffList[count] );

         amount = staffList[count].pay();  // polymorphic

         if( amount == 0.0 )
            System.out.println( "Thanks for volunteering!" );
         else
            System.out.println( "Paid: " + amount );

         System.out.println ("--------------------------------");
      }
   }
}
```

What is this printing?

```java
public class Firm // test class
{
   public static void main (String[] args)
   {
      Staff personnel = new Staff();
      personnel.payday();
   }
}
```

```java
public class Firm // test class
{
   public static void main (String[] args)
   {
      Staff personnel = new Staff();
      personnel.payday();
   }
}
```

Output

```
Name: Sam
Address: 123 Main Line
Phone: 555-0469
Social Security Number: 123-45-6789
Paid: 2923.07
-----------------------------------
Name: Carla
Address: 456 Off Line
Phone: 555-0101
Social Security Number: 987-65-4321
Paid: 1246.15
-----------------------------------
Name: Woody
Address: 789 Off Rocker
Phone: 555-0000
Social Security Number: 010-20-3040
Paid: 1169.23
-----------------------------------
```

Output (continued)

```
Name: Diane
Address: 678 Fifth Ave.
Phone: 555-0690
Social Security Number: 958-47-3625
Current hours: 40
Paid: 422.0
-----------------------------------
Name: Norm
Address: 987 Suds Blvd.
Phone: 555-8374
Thanks!
-----------------------------------
Name: Cliff
Address: 321 Duds Lane
Phone: 555-7282
Thanks!
-----------------------------------
```

Use of Abstract Class in Polymorphism

- Recall:

```
private StaffMember[] staffList;
...
staffList = new StaffMember[6];
staffList[0] = new Executive ...
staffList[5] = new Volunteer ...
```

- Array is declared to hold `StaffMember` references
  - Actually filled with objects of subclasses
  - Another good use of abstract classes

Review

- How does inheritance support polymorphism?

Review

- How does inheritance support polymorphism?
  - A reference variable of class X can be used to refer to an object of class Y if Y is a descendent of X
  - If both classes contain the same method (i.e., same signature), the parent reference can be polymorphic

Review (cont.)

- What is the difference between overriding and polymorphism?

Review (cont.)

- What is the difference between overriding and polymorphism?
  - When a child class overrides the definition of a parent's method, two versions of that method exist
  - A single polymorphic reference can be used to invoke the child or parent method
    - Method version invoked is determined at runtime
    - Contrast to the use of `super`

Last Example

```
public class Figure
{
  public void display() { System.out.println( “Figure” );
}
public class Rect extends Figure
{
  public void display() { System.out.println( “Rectangle” );
}
public class Box extends Figure
{
  public void display() { System.out.println( “Box” );
}
public class TestDisplay
{
  public static void main( String[] args )
  {
    Figure f = new Figure();
    Rect    r = new Rect();
    Box     b = new Box();
    f.display();
    f = r;
    f.display();
    f = b;
    f.display();
  }
}
```

Output: ?

Last Example

```
public class Figure
{
  public void display() { System.out.println( “Figure” );
}
public class Rect extends Figure
{
  public void display() { System.out.println( “Rectangle” );
}
public class Box extends Figure
{
  public void display() { System.out.println( “Box” );
}
public class TestDisplay
{
  public static void main( String[] args )
  {
    Figure f = new Figure();
    Rect    r = new Rect();
    Box     b = new Box();
    f.display();
    f = r;
    f.display();
    f = b;
    f.display();
  }
}
```

Output:

Figure
Rectangle
Box

Summary of Polymorphism

- Polymorphism is an OOP technique that allows us to reference different object types at different points in time
- Makes use of late binding
- Possible when objects related via inheritance
  - Use of casting to explicitly refer to specific object type
  - Use of abstract class as polymorphic reference
- Polymorphism ≠ overriding
- Next class: Polymorphism via interfaces