Polymorphism

Binding

- Typically, we invoke a method through an object:

```
myObject.doSomething();
```

- At some point, this invocation is *bound* to the definition of the method that it invokes.

```
myObject.doSomething()
```

is connected to the method

```
public void doSomething() { … }
```

Binding

- If this binding occurred at compile time, then that line of code would call the same method every time.
- Java defers method binding until run time
  - *dynamic binding* or *late binding*
- Late binding provides flexibility
  - Different methods can be called by the same line of code.

Polymorphism

- *Polymorphism* means “having many forms”
- A *polymorphic reference* is a variable that can refer to different types of objects at different points in time
- The method invoked through a polymorphic reference can change from one invocation to the next
- In Java, all object references are potentially polymorphic

Object References

- You can declare an object to be of a type high up on the inheritance hierarchy.
- You can then instantiate that object to be of any type lower on the hierarchy.

Polymorphism

```
Employee e;
```

- Java allows the variable `e` to point to an object of type `Employee` or to an object of *any compatible type*
- A compatible type is a class *lower* on the inheritance tree.
- All of these are allowed:

```
Employee e = new SalariedPaidEmployee(…)
Employee e2 = new HourlyPaidEmployee(…)
Employee e3 = new UnpaidEmployee(…)
```

Polymorphism

- Why would we want to do that?
- What if we had an array or ArrayList of all different kinds of employees!

```
Employee[] eList = new Employee[10];
eList[0] = new SalariedPaidEmployee(…)
eList[1] = new HourlyPaidEmployee(…)
eList[2] = new UnpaidEmployee(…)
```

Declared Type vs Actual Type

- All object references really have *two* types!
- Declared type is determined at compile time
  - The type from the declaration statement
  - Controls which methods can be invoked
    - A method must exist in the declared class or you cannot compile!
- Actual type is determined at run time
  - The *real* type of object- based on the constructor
  - Controls which version of the method is invoked

Practice

- Declare two employee objects- one a salaried worked and one an hourly worker. Invoke the pay method on each.

Declared Type vs Actual Type

```
Employee e1 = new SalariedPaidEmployee(…)
Employee e2 = new HourlyPaidEmployee(…)
Employee e3 = new UnpaidEmployee(…)
```

Declared Type vs Actual Type

```
Employee e1 = new SalariedPaidEmployee(…)
Employee e2 = new HourlyPaidEmployee(…)
Employee e3 = new UnpaidEmployee(…)
```

- The compiler uses declared type to decide what is allowed.
- Any method invoked on e1, e2, and e3 must exist in Employee because it's the declared type.
- You cannot invoke vacation() on e1 because that method does not exist in Employee.
  - Even though it does exist in SalariedEmployee!

Declared Type vs Actual Type

```
Employee e1 = new SalariedPaidEmployee(…)
Employee e2 = new HourlyPaidEmployee(…)
Employee e3 = new UnpaidEmployee(…)
```

- The JVM uses the *actual type* to decide *which version* of a method to invoke!
- e1.pay() invokes the pay() method from `SalariedPaidEmployee` class.
- e2.pay() invokes the pay() method from `HourlyPaidEmployee` class.
- e3.pay() invokes the pay() method from `UnpaidEmployee` class.

Declared Type vs Actual Type

```
Employee e1 = new SalariedPaidEmployee(…)
Employee e2 = new HourlyPaidEmployee(…)
Employee e3 = new UnpaidEmployee(…)
```

- The JVM uses the *actual type* to decide *which version* of a method to invoke!
- This is polymorphism!
  - An employee object invoking pay could invoke different versions of the method depending on the *actual type* of the object.

Object References

- You can declare an object to be of a type high up on the inheritance hierarchy. You can then instantiate that object to be of any type lower on the hierarchy.

- When you declare an object reference, you specify the type.
  - The compiler only knows about this type.
  - The compiler only allows you to invoke methods associated with the declared type.
- At runtime, the JVM knows the *actual* type of the object.
  - It could be the declared type or any subclass of the declared type.
  - The actual type is used to invoke the correct method.

Practice

- Modify the MusicStore program to create a single AudioItem. Ask the user which type of item they want to create. Print out the objects
  - Notice that, at compile time, you don't know which type of object is created!
- Play a sample from the created objects.

Downcasting

- But what if I want to access a method in the child class?
  - Example: On my array of employees, print pay for every employee and vacation only for every salaried employee.
- You can do this with a *narrowing conversion*, also called *casting* or a *downcast.*

Downcasting

- Always use instanceof before you downcast to make sure it's the type you think it is!

```
for(int i=0; i<empList.length; i++) {
    Employee e = empList[i];
    e.pay();
    if(e instanceof SalaryEmployee) {
            SalaryEmployee s = (SalaryEmployee) e;
            s.vacation();
            // or without the local variable
            // (( SalaryEmployee) e).vacation();
    }
}
```

Downcasting

- Always use instanceof before you downcast to make sure it's the type you think it is!

```
for(int i=0; i<empList.length; i++) {
    Employee e = empList[i];
    e.pay();
    if(e instanceof SalariedPaidEmployee) {
        SalariedPaidEmployee s = (SalariedPaidEmployee ) e;
        s.vacation();
        // or without the local variable
        // ((SalariedPaidEmployee ) e).vacation();
    }
}
```

okay because pay is defined in Employee; the right version will be called (polymorphism!)

tells the compiler to temporarily treat e like a SalaryEmployee; does **not** change the type of e!

Downcasting

- instanceof is a safety check that you have the right type.
- The cast is the command to the compiler.

- You need to do both!

Practice

- Print the genres of only the music items created.

Practice

- Write a class for Customer and Preferred Customer. A customer is described by a name, id, and address.
- Write a method to invite the customer to a sale and a method to send the customer a birthday reward.
- Create a driver program to test the classes.