Activity 16: Polymorphism Revisited

Today, we revisit polymorphism in the context of inheritance and interfaces. As class hierarchies get more complex, it is important to be able to trace what happens when we call a method from a variable, especially if that variable's declared type and instantiated type are different.

Content Learning Targets

After completing this activity, you should be able to say:

- I can identify the declared type and instantiated type of a variable.
- I can trace polymorphic method calls in a class hierarchy.
- I can identify, and distinguish, compile-time and runtime errors in polymorphic method calls.
- I can apply polymorphism to avoid code duplication.

Process Skill Goals

During the activity, you should make progress toward:

- Predicting code output before running it and identifying patterns. (Critical Thinking)
- Forming shared understanding via team discussion. (Teamwork)

Facilitation Notes

First Hour: Model 1 explores the rules for tracing polymorphic method calls. Have students work through the puzzles in teams, then check their work and try to form rules for determining which method gets called. Go through the five-step process in the slides, then have students individually complete the Polymorphic Tracing Practice activity in Moodle.

Second Hour: Model 2 practices using polymorphism to avoid code duplication with live coding activities. Model 3 provides guidance for the open work time at the end.

Key questions: #6, #8, #12

Model 1 Tracing Polymorphic Method Calls

When we call a method from a variable whose type is in a complex class hierarchy, it can be tricky to sort out what actually happens. We start today with the following polymorphic method call "puzzles".

```
class A {
    public void print() {
        System.out.println("A");
    }
}

class B extends A {
    public void print() {
        System.out.println("B");
    }
}

class C extends B {
    public void print() {
        System.out.println("C");
    }
    public void printC() {
        print();
    }
}

class D extends C {
    public void print() {
        System.out.println("D");
    }
}
```

1. Using the space on the right, draw the UML diagram for these classes.

PlantUML link

2. The superclass of `A` is... `Object`

3. What method(s) is/are available to an object of type `D`?

`print()` (overridden) and `printC()` (inherited from `C`), as well as all methods inherited from `Object`

4. For each of the following code snippets, write down what is printed when the code is run. If there is a compile-time error, write "compile error". If there is a runtime error, write "runtime error". (If you're not sure, make your best guess.)
Some variables are reused, such as `x` from the first snippet. If an error happens in one snippet, assume it is commented out before running the next snippet.

a)
```
A x = new D();
x.print();
```

D

b)
```
D x2 = (D)x;
x2.print();
```

D

c)
```
x2.printC();
```

D

d)
```
B x3 = (B)x;
x3.print();
```

D

e)
```
x3.printC();
```

compile error (B has no method `printC()`)

f)
```
C x4 = new C();
x4.printC();
```

C

g)
```
D x5 = (D)x4;
x5.print();
```

runtime error (`ClassCastException`)

5. In `src/polymorphicMethods`, check your work by uncommenting each code snippet. Were there any surprises?

You may be surprised to see that casting `x` to a `B` does not change the method that gets called; we still use the `D` version of `print()`.

6. In your team, try to come up with a set of rules for determining which method ***actually*** gets called when a method is invoked on a polymorphic variable.

See the five-step process in the slides.

7. Individually, complete the Polymorphic Tracing Practice activity in Moodle.

Model 2 Using Polymorphism Effectively: Live Coding

We will practice together using polymorphism to avoid code duplication.

Bank Accounts: Reusing `toString()` Code with Polymorphism

In the `src/banking` folder, you will find several classes that represent different types of bank accounts. Each class has its own implementation of the `toString()` method, which returns a string representation of the account information.

8. How did we use polymorphism to avoid duplicating the `toString()` method?

We used `this.getClass().getSimpleName()` to get the class name dynamically, so the default `toString()` method in the `BankAccount` class works for the `CheckingAccount` class.

Chess Pieces: Refactoring from an Interface to an Abstract Class

This activity's starter files are in the `src/chessSupport` and `src/chessPieces` folders.

9. Take a minute to explore the code, starting from `src/chessSupport/ChessMain.java`.

10. The starter code roughly follows this UML class diagram (PlantUML link).

11. Let's add the `Queen` class. Like `King`, it will implement the `ChessPiece` interface. What problem do we run into?

We have to copy lots of code from the `King` class, including the `isWhite` field and most of the methods. And several of these methods will look identical for every chess piece!

12. How did we refactor to fix this problem?

We changed `ChessPiece` from an interface to an abstract class, and provided default implementations for several methods.

Model 3 Open Work Time

Use the remaining class time to work on any of the following:

- Continue working on the chess piece classes (in pairs or solo), using these movement rules as a guide. ***Save the `Pawn` class for last***, since it has some special rules. Suggested order: `Rook`, `Bishop`, `Knight`, `Pawn`.

- Individually, complete the Polymorphic Tracing Practice activity in Moodle.

- If you are confident in your understanding of today's topics, you and your HW9 partner can work together.