Building Java Programs

Chapter 9
Inheritance and Polymorphism

reading: 9.1 - 9.2
Before class starts

- Interactive Activities
  - Go to pollev.com/cse143 on your phone
  - Type in your UW email
  - Don’t create account / type in password
  - Click link for single sign-on
  - Sign in using your UW credentials
  - Answer the question!
“Being given time to talk to my peers and TAs in lecture helps clarify concepts I might have been confused about.”

<table>
<thead>
<tr>
<th></th>
<th>6 respondents</th>
<th>2 %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strongly Disagree</td>
<td>27 respondents</td>
<td>7 %</td>
</tr>
<tr>
<td>Disagree</td>
<td>119 respondents</td>
<td>31 %</td>
</tr>
<tr>
<td>Neutral</td>
<td>149 respondents</td>
<td>38 %</td>
</tr>
<tr>
<td>Agree</td>
<td>82 respondents</td>
<td>21 %</td>
</tr>
<tr>
<td>Strongly Agree</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

“In general, I am attentive with what’s going on during lecture.”

“I feel comfortable asking questions in lecture.”

<table>
<thead>
<tr>
<th>Strongly Disagree</th>
<th>8 respondents</th>
<th>2 %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disagree</td>
<td>16 respondents</td>
<td>4 %</td>
</tr>
<tr>
<td>Neutral</td>
<td>52 respondents</td>
<td>13 %</td>
</tr>
<tr>
<td>Agree</td>
<td>192 respondents</td>
<td>49 %</td>
</tr>
<tr>
<td>Strongly Agree</td>
<td>114 respondents</td>
<td>29 %</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Strongly Disagree</th>
<th>31 respondents</th>
<th>8 %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disagree</td>
<td>103 respondents</td>
<td>27 %</td>
</tr>
<tr>
<td>Neutral</td>
<td>133 respondents</td>
<td>34 %</td>
</tr>
<tr>
<td>Agree</td>
<td>77 respondents</td>
<td>20 %</td>
</tr>
<tr>
<td>Strongly Agree</td>
<td>38 respondents</td>
<td>10 %</td>
</tr>
</tbody>
</table>
Asking Questions

• Asking questions is crucial to your learning
  • Goal: Make a classroom environment that welcomes (and encourages) asking questions
• Sometimes it can be a bit hard to ask questions in a 500 person lecture
• Some alternatives
  • Index cards (once a week)
  • While TAs are walking around
  • Have a TA ask a question for you
    • pollev.com/cse143questions
Recall: Inheritance

- **inheritance**: Forming new classes based on existing ones.
  - a way to share/reuse code between two or more classes
- **superclass**: Parent class being extended.
- **subclass**: Child class that inherits behavior from superclass.
  - gets a copy of every field and method from superclass
- **is-a relationship**: Each object of the subclass also "is a(n)" object of the superclass and can be treated as one.
Recall: Inheritance

public class A {
    public void m1() {
        S.o.pln(“A1”);
    }

    public void m2() {
        S.o.pln(“A2”);
    }
}

public class B extends A {
    public void m2() {
        super.method1();
        S.o.pln(“B2”);
    }
}

public class B extends A {
    public void m2() {
        super.method1();
        S.o.pln(“B2”);
    }
}

A a = new A();
B b = new B();
b.m1(); // A1
a.m2(); // A2
b.m2(); // A1 / B2
public class A {
    public void m1() {
        S.o.pln("A1");
    }

    public void m2() {
        S.o.pln("A2");
    }

    public void m3() {
        S.o.pln("A3");
    }
}

public class B extends A {
    public void m2() {
        S.o.pln("B2");
    }

    public void m3() {
        super.m1();  // A1
        S.o.pln("C3"); // C3
    }
}

public class C extends B {
    public void m1() {
        S.o.pln("C1");
    }

    public void m3() {
        super.m1();  // A1
        S.o.pln("C3"); // C3
    }
}

C c = new C();
c.m3();

What is the output?
• A1 / C3 ★
• B1 / C3
• C1 / C3
• C3
• Some kind of error
Why cover this again?

- New Topics
  - Polymorphism when calling other methods
  - Investigating Java’s type system
    - What happens when you using casting with objects?
    - What is and isn’t possible for the compiler to check?
- Motivation: We’ve been hand-waving what it means to say
  ```java
  List<Integer> list = new ArrayList<Integer>();
  list.add(1);
  ```
- Why allow different types on the left side vs. right side?
  ```java
  PromiseType variable = new ActualType();
  ```
- PromiseType can be a superclass that ActualType extends
  or an interface that ActualType implements
  - Restricts usage of the instance of ActualType to only
    PromiseType methods. Why is this useful?
Example: Music Players

- MP3 Player
  - iPod
  - Zune
  - iPhone

- Music Player
  - Tape Deck
    - record
  - CD Player
    - play
    - pause

- Object
  - iPoD solitaire
  - phone Call
MusicPlayer p = new Zune();

(((iPhone) p2).record());

What does this line do?
- Call record on Zune
- Call record on MusicPlayer
- Call record on iPhone
- Compiler Error
- Runtime Error

compile time happens first
```java
public class MusicPlayer {
    public void m1() {
        System.out.println("MusicPlayer1");
    }
}

public class TapeDeck extends MusicPlayer {
    public void m3() {
        System.out.println("TapeDeck3");
    }
}

public class IPod extends MusicPlayer {
    public void m2() {
        System.out.println("IPod2");
        m1();
    }
}

public class IPhone extends IPod {
    public void m1() {
        System.out.println("IPhone1");
        super.m1();
    }
    public void m3() {
        System.out.println("IPhone3");
    }
}
```

<table>
<thead>
<tr>
<th></th>
<th>m1</th>
<th>m2</th>
<th>m3</th>
</tr>
</thead>
<tbody>
<tr>
<td>MusicPlayer</td>
<td>MP1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TapeDeck</td>
<td>MP1</td>
<td></td>
<td>TD3</td>
</tr>
<tr>
<td>IPod</td>
<td>MP1</td>
<td>MP2</td>
<td></td>
</tr>
<tr>
<td>IPhone</td>
<td>MP1</td>
<td>MP2</td>
<td>IPhone3</td>
</tr>
</tbody>
</table>

Method calls: Write method call
Super calls: Write output of call
MusicPlayer var1 = new TapeDeck();
MusicPlayer var2 = new iPod();
MusicPlayer var3 = new IPhone();
IPod var4 = new IPhone();
Object var5 = new iPod();
Object var6 = new MusicPlayer();

var1.m1();
MusicPlayer1

var3.m1();
IPhone1 / MusicPlayer1

var4.m2();
IPod2 / IPhone1 / MusicPlayer1

var3.m2();
Compiler Error (CE)

var5.m1();
Compiler Error (CE)
```java
MusicPlayer var1 = new TapeDeck();
MusicPlayer var2 = new iPod();
MusicPlayer var3 = new IPhone();
IPod var4 = new IPhone();
Object var5 = new iPod();
Object var6 = new MusicPlayer();

((TapeDeck) var1).m2();
Compiler Error (CE)

((IPod) var3).m2();
IPod2 / IPhone1 / MusicPlayer1

((IPhone) var2).m1();
Runtime Error (RE)

((TapeDeck) var3).m2();
Compiler Error (CE)
```
**General Rule**

**PromiseType** var = new **ActualType**();
var.method() or ((**CastType** ) var).method();

**Compile Time**
if (involves casting) {
    check if **CastType** has method, if not fail with **CE**
} else {
    check if **PromiseType** has method, if not fail with **CE**
}

**RunTime (if compiles)**
if (involves casting) {
    check if **ActualType** can actually be cast to **CastType**, 
    if not fail with **RE**
}
call method on **ActualType**