Building Java Programs

Chapter 9
Inheritance and Polymorphism

reading: 9.1 - 9.2
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.

![Diagram showing the hierarchy of classes and their forms.](image-url)
public class A {
    public void m1() {
        m2();
        S.o.pln("A1");
    }

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

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

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

Poll Everywhere
B b = new B();
b.m1();

What is the output?
• A2 / A1
• B2 / A1
• Some kind of error
• I’m not sure
Why cover this again?

- New Topics
  - More practice with understanding polymorphism
  - 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 super-type 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

Object
- Music Player
  - MP3 Player
    - iPod
    - Zune
  - Tape Deck
    - play
    - pause
  - CD Player
    - phone Call
MusicPlayer p3 = new Zune();

((iPhone) p3).record();

What does this line do?
- Call record on Zune
- Call record on MusicPlayer
- Call record on iPhone
- Compiler Error
- Runtime Error
public class MusicPlayer {
    public void m1() {
        S.o.pln("MusicPlayer1");
    }
}

public class TapeDeck
    extends MusicPlayer {
    public void m3() {
        S.o.pln("TapeDeck3");
    }
}

public class IPod
    extends MusicPlayer {
    public void m2() {
        S.o.pln("IPod2");
        m1();
    }
}

public class IPhone
    extends IPod {
    public void m1() {
        S.o.pln("IPhone1");
        super.m1();
    }
    public void m3() {
        S.o.pln("IPhone3");
    }
}
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();
Compiler Error (CE)

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

((TapeDeck) var3).m2();
Compiler Error (CE)
```

<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>IPod2</td>
<td>/</td>
</tr>
<tr>
<td>IPhone</td>
<td>IPhone</td>
<td>IPod2</td>
<td>IPhone3</td>
</tr>
</tbody>
</table>
**General Rule**

```javascript
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