Building Java Programs

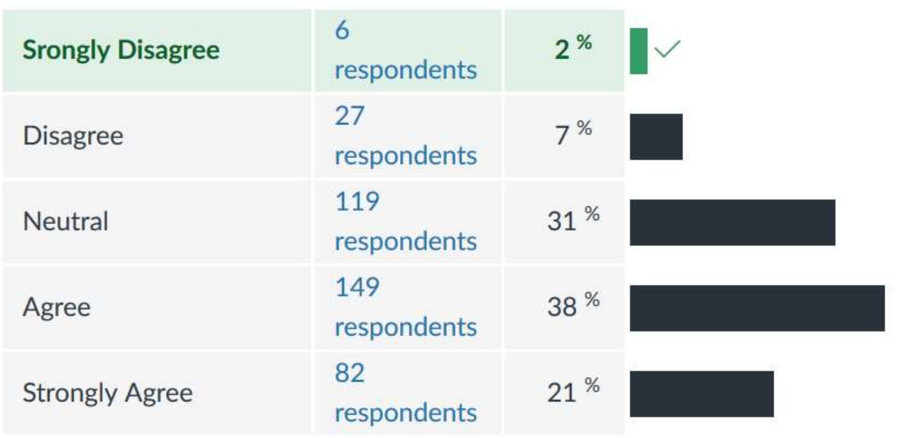
Chapter 9
Inheritance and Polymorphism

reading: 9.1 - 9.2

Before class starts

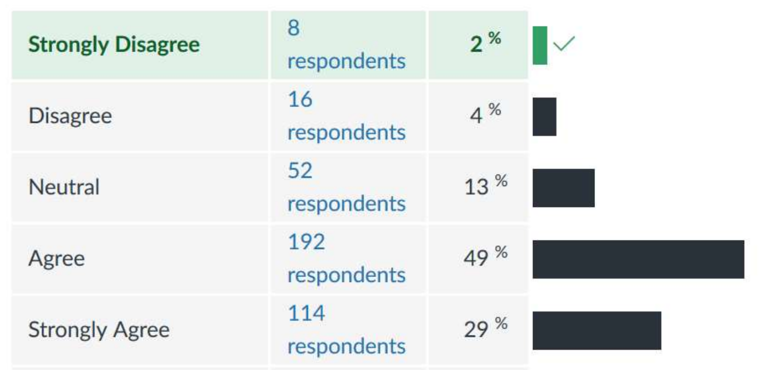
- Interactive Activities
  - Go to pollev.com/cse143 on your phone
  - Type in your UW email
  - **Don't create account / type in password**
  - Click link for single sign-on
  - Sign in using your UW credentials
  - Answer the question!

"Being given time to talk to my peers and TAs in lecture helps clarify concepts I might have been confused abotu."



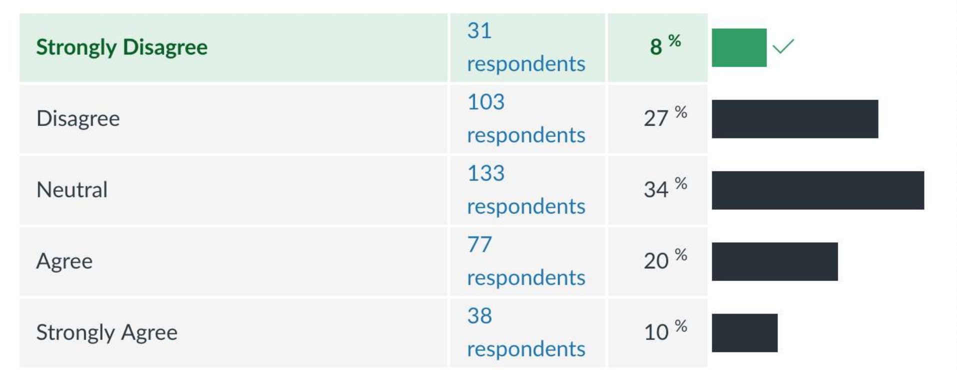
Srongly Disagree	6 respondents	2 %	✓
Disagree	27 respondents	7 %	
Neutral	119 respondents	31 %	
Agree	149 respondents	38 %	
Strongly Agree	82 respondents	21 %	

"In general, I am attentive with what's going on during lecture."



Strongly Disagree	8 respondents	2 %	✓
Disagree	16 respondents	4 %	
Neutral	52 respondents	13 %	
Agree	192 respondents	49 %	
Strongly Agree	114 respondents	29 %	

"I feel comfortable asking questions in lecture."



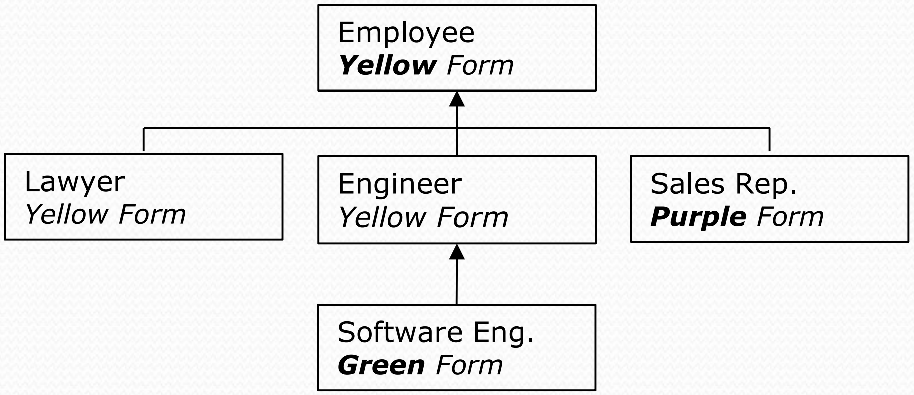
Strongly Disagree	31 respondents	8 %	✓
Disagree	103 respondents	27 %	
Neutral	133 respondents	34 %	
Agree	77 respondents	20 %	
Strongly Agree	38 respondents	10 %	

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”);
  }
}
```

```
A a = new A();
B b = new B();

b.m1(); // A1
a.m2(); // A2
b.m2(); // A1/ B2
```

	m1	m2
A	A1	A2
B	A1	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 class C extends B {
   public void m1() {
      S.o.pln(“C1”);
   }

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

Poll Everywhere

```
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

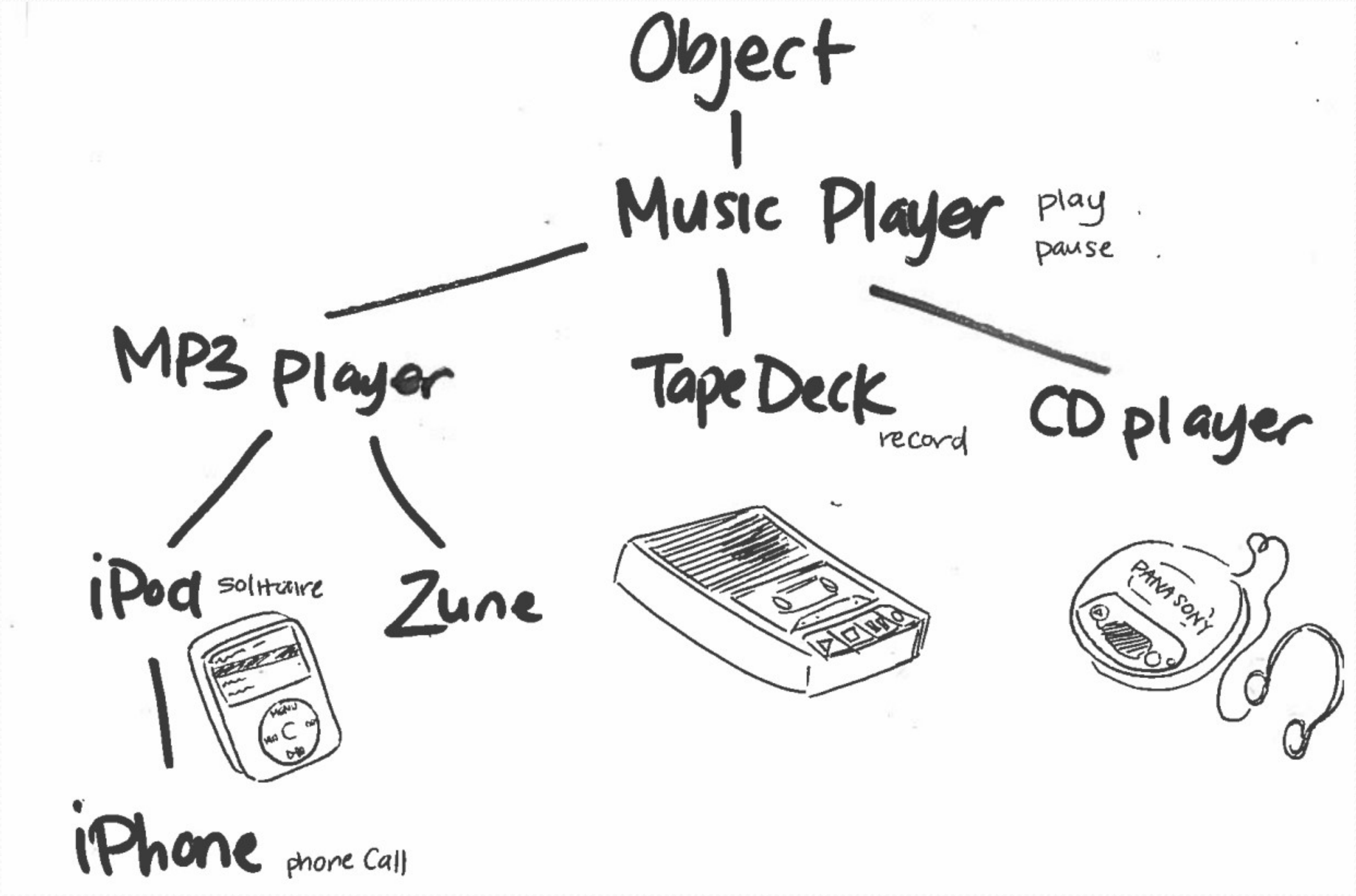
```
List<Integer> list = new ArrayList<Integer>();
list.add(1);
```

- Why allow different types on the left side vs. right side?

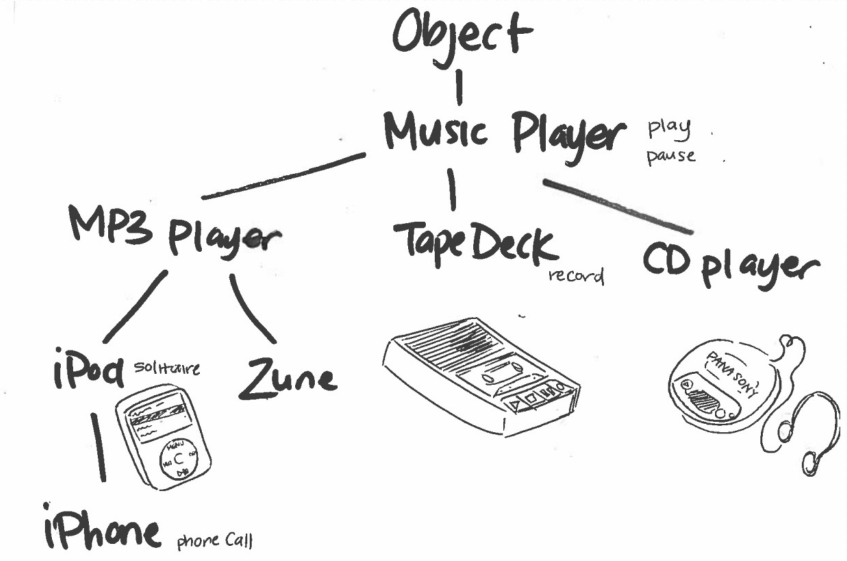
```
PromiseType variable = new ActualType();
```

- `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



Poll Everywhere



```
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

```
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");
  }
}
```

	m1	m2	m3
MusicPlayer	MP1	/	/
TapeDeck	MP1	/	TD3
IPod	MP1	IPod2 m1()	/
IPhone	IPhone1 MP1	IPod2 m1()	IPhone3

Method calls: Write method call

Super calls: Write output of call

	m1	m2	m3
MusicPlayer	MP1	/	/
TapeDeck	MP1	/	TD3
IPod	MP1	IPod2 **m1**()	/
IPhone	IPhone1 MP1	IPod2 **m1**()	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)

	m1	m2	m3
MusicPlayer	MP1	/	/
TapeDeck	MP1	/	TD3
IPod	MP1	IPod2 **m1**()	/
IPhone	IPhone1 MP1	IPod2 **m1**()	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();
```

```
((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
```