COSC 121: Computer Programming II

Dr. Bowen Hui

University of British Columbia Okanagan

Quick Review

Representative example:

```
Animal[] myPets = new Animal[4];
myPets[0] = new Dog();
myPets[1] = new Cat();
myPets[2] = new Sheep();
myPets[3] = new Cow();
for( int i=0; i<myPets.length; i++ )
{
  System.out.println( myPets[i].talk() );
}
```

Output

```
Woof
Meow
Beh
Moo
```

- How are Animal, Dog, Cat, Sheep, Cow related?

Polymorphism via Interfaces

- Can also use interfaces to setup polymorphic references
- Follows same rules as inheritance
- Suppose we declare an interface `Speaker`:

```
public interface Speaker
{
   public void speak();
   public void announce( String str );
}
```

How to use it

- Interface used as type of object reference variable
  - Ex: `Speaker presenter;`
    `presenter.speak();`
  - `Presenter` reference used to point to any class that implements `Speaker`
  - Version of `speak()` invoked depends on type of object being referenced at runtime
- Recall, we cannot write:

```
Speaker presenter = new Speaker();
```

How to use it (cont.)

- Suppose we have two classes: `Philosopher`, `Dog`
  - Both implement `Speaker`
  - Both have different `speak()` methods
- Example:

```
Speaker guest = new Philosopher();
guest.speak();
guest = new Dog();
guest.speak();
```

In Detail

- Example:

  ```
  Speaker guest = new Philosopher();
  guest.speak();
  guest = new Dog();
  guest.speak();
  ```

- First call to `speak()`:
  - Calls definition in `Philosopher`
- Second call to `speak()`:
  - Calls definition in `Dog`
- Same reference variable (`guest`) used both times

Calling Class-specific Methods

- Compiler restricts calls to methods not in the interface
- Ex: Suppose `Philosopher` also had a method called `pontificate()`:

```
Speaker special = new Philospher();
special.pontificate();  // error
```

  - Causes a compiler error
- Reason: compiler bases its rules on the reference type
- How to fix this?

Using Casting

- Fixing previous example:

```
Speaker special = new Philosopher();
(( Philosopher )special).pontificate();
```

- Tells compiler `special` really is a `Philosopher`

Which way when?

- Polymorphism via inheritance

- Polymorphism via interfaces

Which way when?

- Polymorphism via inheritance
  - Extends an abstract class (or a parent class)
  - Inheritance necessarily means similar classes (IS-A)
  - Abstract class method definitions provide default behaviour (less redundant code)

- Polymorphism via interfaces
  - Implements an interface
  - No semantic relationship needed!
    - Example: `Human` and `Table` can both `stand()`

Example

- Which of these statements are valid?

```
1. Speaker first = new Dog();
2. first.speak();
3. Philosopher second = new Philosopher();
4. second.pontificate();
5. first = second;
6. first.speak();
7. Dog third = new Dog();
8. third.speak();
9. third = first;
10.third = second;
```

Example

- Which of these statements are valid?

```
1. Speaker first = new Dog();
2. first.speak();
3. Philosopher second = new Philosopher();
4. second.pontificate();
5. first = second;
6. first.speak();
7. Dog third = new Dog();
8. third.speak();
9. third = first;
10.third = second;
```

- Whose method is called on line 2?

Example

- Which of these statements are valid?

```
1. Speaker first = new Dog();
2. first.speak();
3. Philosopher second = new Philosopher();
4. second.pontificate();
5. first = second;
6. first.speak();
7. Dog third = new Dog();
8. third.speak();
9. third = first;
10.third = second;
```

- Whose method is called on line 4?

Example

- Which of these statements are valid?

```
1. Speaker first = new Dog();
2. first.speak();
3. Philosopher second = new Philosopher();
4. second.pontificate();
5. first = second;
6. first.speak();
7. Dog third = new Dog();
8. third.speak();
9. third = first;
10.third = second;
```

- Whose method is called on line 6?

Example

- Which of these statements are valid?

```
1. Speaker first = new Dog();
2. first.speak();
3. Philosopher second = new Philosopher();
4. second.pontificate();
5. first = second;
6. first.speak();
7. Dog third = new Dog();
8. third.speak();
9. third = first;
10.third = second;
```

- Whose method is called on line 8?

Example

- Which of these statements are valid?

```
1. Speaker first = new Dog();
2. first.speak();
3. Philosopher second = new Philosopher();
4. second.pontificate();
5. first = second;
6. first.speak();
7. Dog third = new Dog();
8. third.speak();
9. third = first;
10.third = second;
```

- How to fix line 9?

Fix for Line 9

- Options:
  - Change 9 to: third = (Dog)first;
  - Change 7 to: Speaker third = new Dog();
  - Change 1 to: Dog first = new Dog();
- Casting conflicts with lines 3-6 because Philosopher cannot be casted to a Dog:

```
3. Philosopher second = new Philosopher();
4. second.pontificate();
5. first = second;
6. first.speak();
```

Example

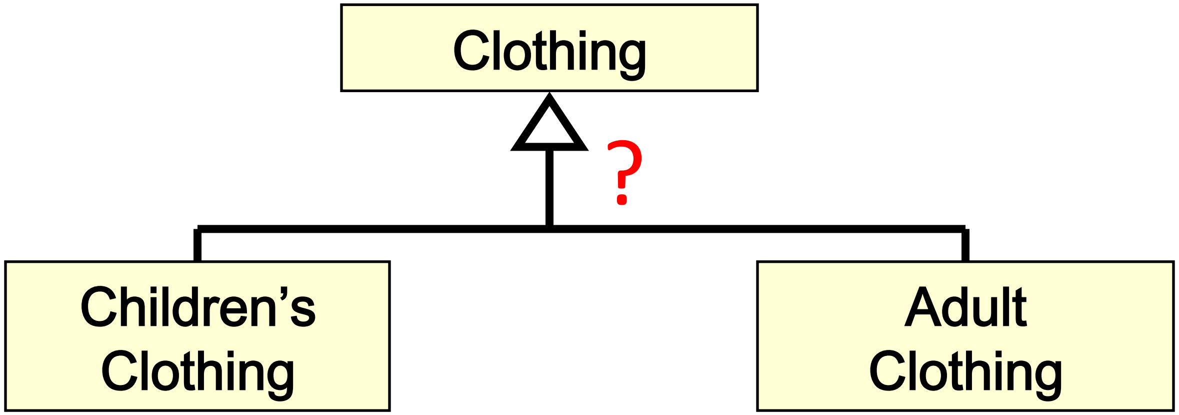
- Which of these statements are valid?

```
1. Speaker first = new Dog();
2. first.speak();
3. Philosopher second = new Philosopher();
4. second.pontificate();
5. first = second;
6. first.speak();
7. Dog third = new Dog();
8. third.speak();
9. third = first;
10.third = second;
```

- How to fix line 10?

Modeling Example

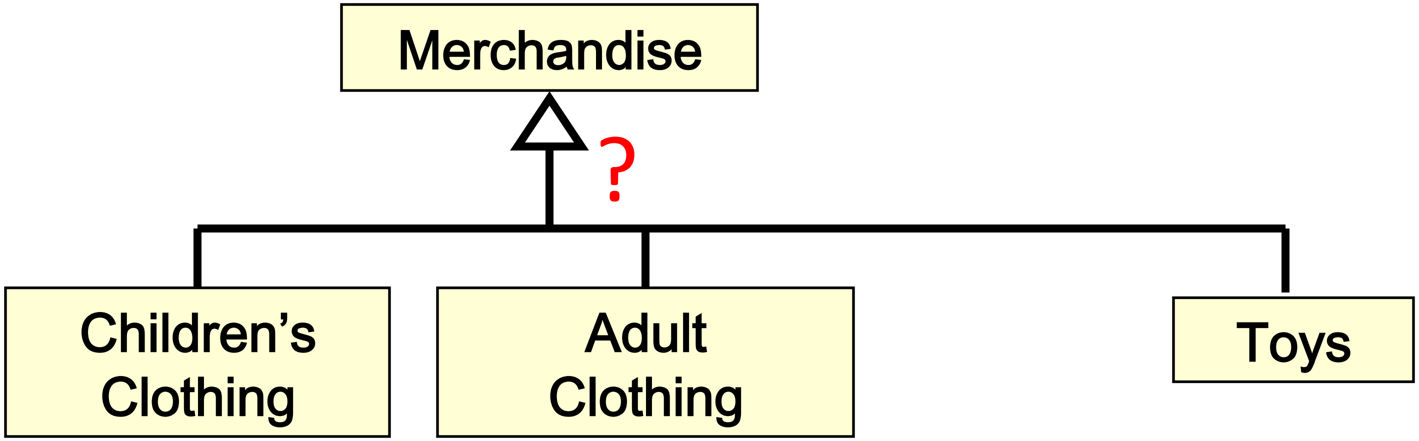
- A company sells merchandise (e.g., clothing items) to customers
- Since children's clothing are not tax applicable, two classes have been created



- In which class(es) would you put the method `computeTax()`?
- Would you implement `Clothing` as a regular class, abstract class, or an interface? Why?

Modeling Example 2

- Similar setup
- Now, suppose there are many types of merchandise the company sells



- Would you implement `Merchandise` as a regular class, abstract class, or an interface? Why?

Polymorphic References as Input Parameters

- A powerful way to both specify the types of parameters to pass into a method
- Gives flexibility to the types of method parameters to accept
- Example:

```
public void sayIt( Speaker current )
{
   current.speak();
}
```

Polymorphic References as Input Parameters

- If types are related, overloading is not needed
- We do this:

```
public void sayIt( Speaker current ){ current.speak(); }
```

- Instead of:

```
public void sayIt( Philosopher current )
{
  current.speak();
}
public void sayIt( Dog current )
{
  current.speak();
}
// etc. one per each signature
```

Review

- Polymorphism is achieved by:

a. Overloading

b. Overriding

c. Embedding

d. Abstraction

e. Encapsulation

Review (cont.)

- What is the term to describe the technique that Java uses to determine the type of object a polymorphic reference is bound to at run time?

Review (cont.)

- A polymorphic reference can refer to different types of objects over time.
  - True?
  - False?

Review (cont.)

- Java allows us to create polymorphic references using inheritance but not using interfaces.
  - True?
  - False?

Review (cont.)

- A reference variable can refer to any object created from any class related to it by inheritance.
  - True?
  - False?

Review (cont.)

- In the following statement, what is the type of the reference variable?

```
    Speaker person;
    person = new Philosopher();
```

  - Speaker?
  - Philosopher?

Review (cont.)

- In the following statement, what is the type of the object?

```
Speaker person;
person = new Philosopher();
```

  - Speaker?
  - Philosopher?

Review (cont.)

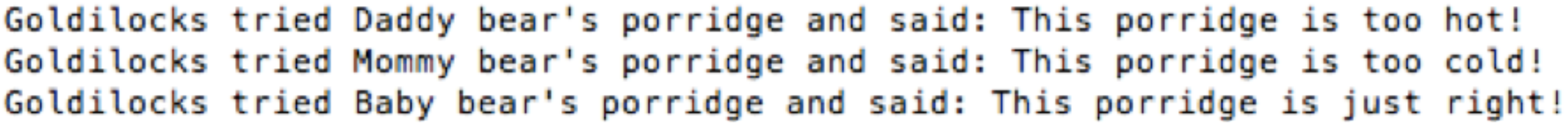
- The type of the reference variable, not the type of the object, is used to determine which version of a method is invoked in a polymorphic reference.
  - True?
  - False?

Review (cont.)

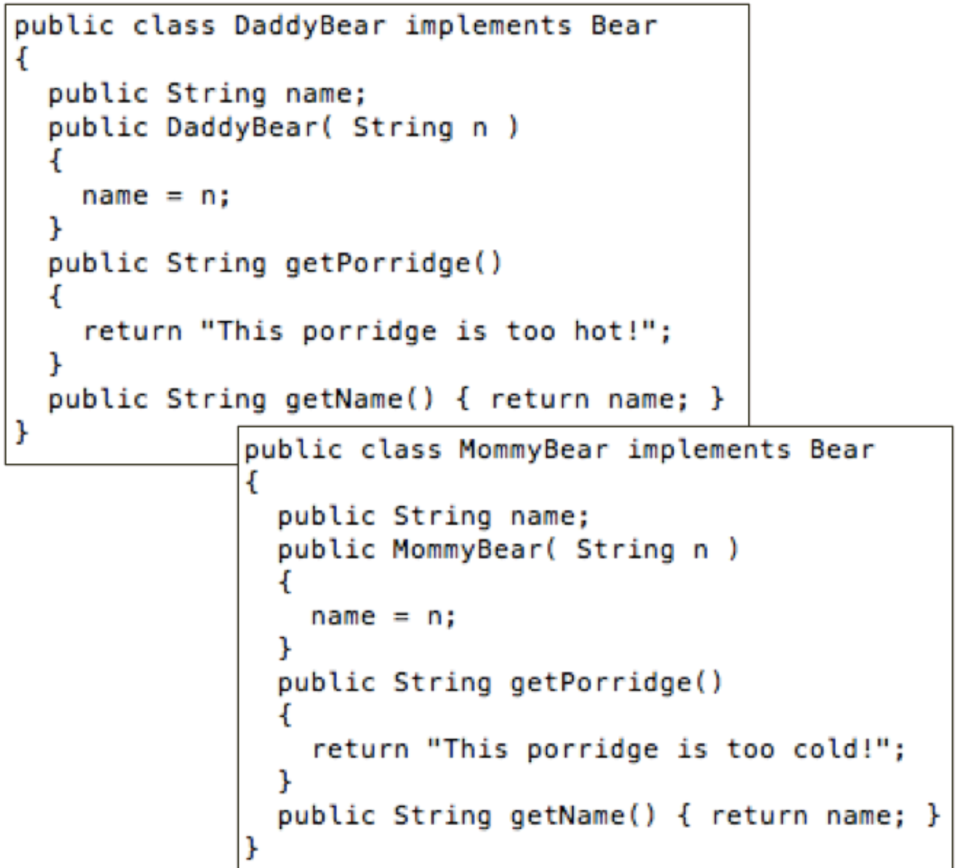
- `System.out.println()` is able to handle a variety of objects and print them correctly is an example of the polymorphic nature of `println()`.
  - True?
  - False?

Exercise

- Write `Bear` and `main()` to generate:

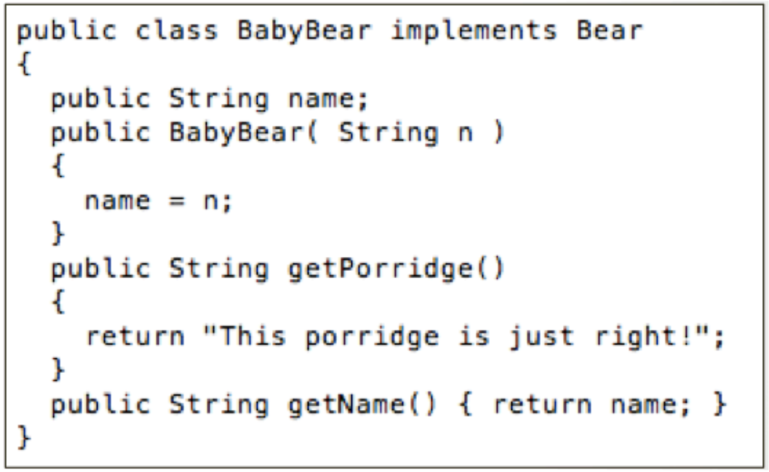


```
Goldilocks tried Daddy bear's porridge and said: This porridge is too hot!
Goldilocks tried Mommy bear's porridge and said: This porridge is too cold!
Goldilocks tried Baby bear's porridge and said: This porridge is just right!
```



```
public class DaddyBear implements Bear
{
  public String name;
  public DaddyBear( String n )
  {
    name = n;
  }
  public String getPorridge()
  {
    return "This porridge is too hot!";
  }
  public String getName() { return name; }
}
```

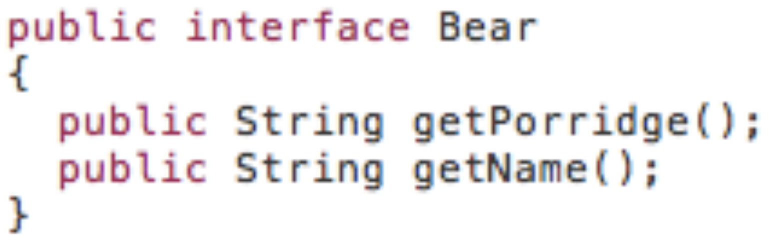
```
public class MommyBear implements Bear
{
  public String name;
  public MommyBear( String n )
  {
    name = n;
  }
  public String getPorridge()
  {
    return "This porridge is too cold!";
  }
  public String getName() { return name; }
}
```



```
public class BabyBear implements Bear
{
  public String name;
  public BabyBear( String n )
  {
    name = n;
  }
  public String getPorridge()
  {
    return "This porridge is just right!";
  }
  public String getName() { return name; }
}
```

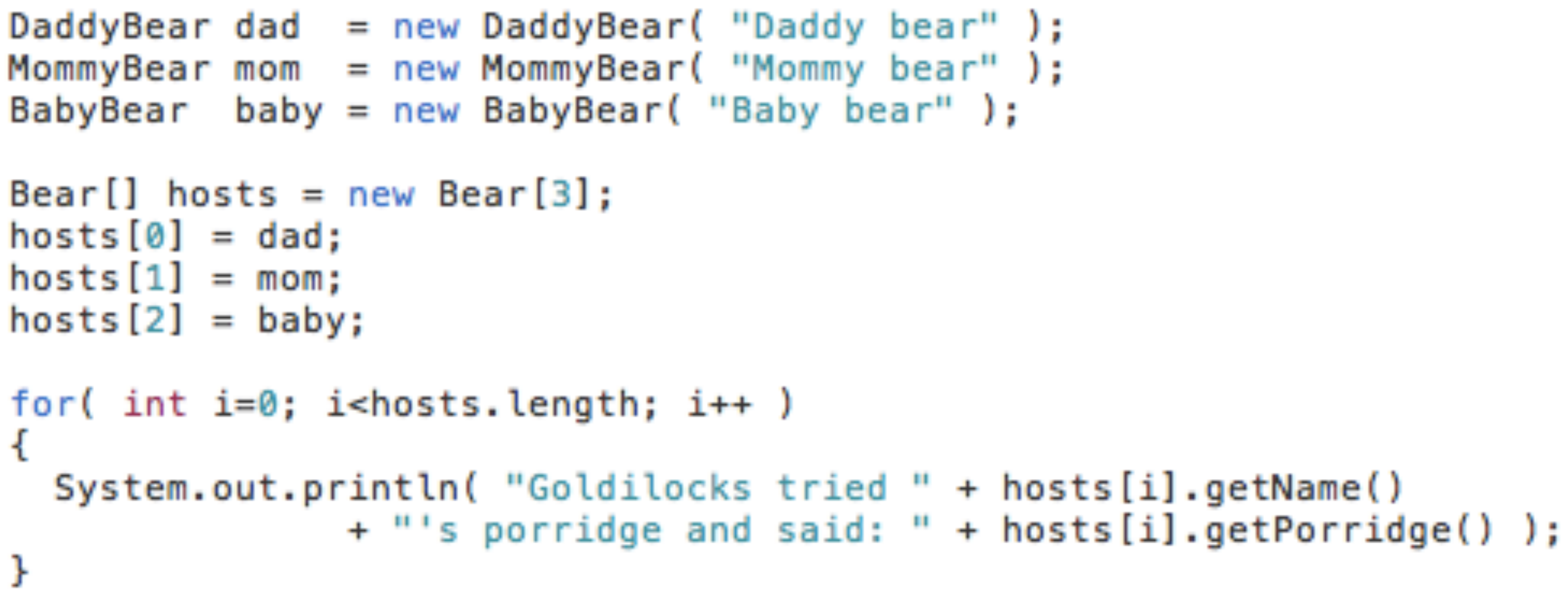
make sure solution uses polymorphism

Sample `Bear` and `main()`



```
public interface Bear
{
  public String getPorridge();
  public String getName();
}
```

// your main():



```
DaddyBear dad  = new DaddyBear( "Daddy bear" );
MommyBear mom  = new MommyBear( "Mommy bear" );
BabyBear  baby = new BabyBear( "Baby bear" );

Bear[] hosts = new Bear[3];
hosts[0] = dad;
hosts[1] = mom;
hosts[2] = baby;

for( int i=0; i<hosts.length; i++ )
{
  System.out.println( "Goldilocks tried " + hosts[i].getName()
               + "'s porridge and said: " + hosts[i].getPorridge() );
}
```

Summary of Polymorphism

- Polymorphism is an OOP technique that allows us to reference different object types at different points in time via late binding
- A reference variable:

  ```
  Occupation job;
  ```

  can point to an `Occupation` object, or any object of a *compatible* type
  - Established via inheritance or interface
- Polymorphism improves program design
  - More elegant and robust code