COSC 121: Computer Programming II

Dr. Bowen Hui

University of British Columbia Okanagan

A1

- Posted over the weekend
- Two questions (still long questions)
  - Review of main concepts from COSC 111
  - Practice coding basic classes with inheritance
    - Use `extends`, `protected`, `super`, possibly `abstract`, `this`
    - Practice overriding
    - Don't use shadow variables (discuss today)
- Due: Jan 25th (about two weeks)
- Also: TA office hours posted on course website

Quick Review

- Inheritance is a new class relationship
  - Lets child class inherit all attributes and methods (except constructor)
  - Maximize code reuse
- Reserved words:
  - `extends`
  - `protected`
    - Visibility range: private – protected – public
  - `super`

Import vs. Extends

- Difference between importing a class (e.g., Scanner class) and extending a class?
- Imported class:
  - Lets you use someone else's code

- Extended class:
  - Lets you use someone else's code

Import vs. Extends

- Difference between importing a class (e.g., Scanner class) and extending a class?
- Imported class:
  - Lets you use someone else's code
  - Can access info that was declared public
- Extended class:
  - Lets you use someone else's code
  - Can modify it to fit your needs
  - Can access info that was declared protected/public

Main Concepts Today

- Overriding methods
  - Allows us to tailor child class method to our liking
- Shadow variables
  - Bad naming convention during inheritance
- Class hierarchy
  - Gives a big picture of how many classes are (already) related
- Abstract classes
  - Ability to model generic concepts without ever creating objects of those classes

Overriding Methods

- Sometimes, parent's method is not exactly what you want for the child class
- A child class can override the definition of an inherited method
- New method must have the same signature as the parent method

Text Example: Thought, Advice, Messages

```
// in Thought.java
public class Thought
{
  public void message()
  {
    System.out.println ("I feel diagonally parked");
  }
}
// in Advice.java
public class Advice extends Thought
{
  public void message()
  {
    System.out.println( ”Dates are closer than they seem” );
    super.message();
  }
}
```

Text Example: Thought, Advice, Messages

```
// in Thought.java
public class Thought
{
   public void message()
   {
      System.out.println ("I feel diagonally parked");
   }
}
// in Advice.java
public class Advice extends Thought
{
   public void message()
   {
      System.out.println( ”Dates are closer than they seem” );
      super.message();
   }
}
```

no attributes!

no constructor!

not OOP: just for illustration purposes

Text Example: Thought, Advice, Messages

```
// in Thought.java
public class Thought
{
  public void message()
  {
    System.out.println ("I feel diagonally parked");
  }
}
// in Advice.java
public class Advice extends Thought
{
  public void message()
  {
    System.out.println( ″Dates are closer than they seem″ );
    super.message();
  }
}
```

Text Example: Thought, Advice, Messages

```java
// in Messages.java (a test class)
public class Messages
{
  public static void main( String[] args )
  {
    Thought parked = new Thought();
    Advice  dates  = new Advice();

    parked.message();
    dates.message();
  }
}
```

in dates.message() which definition does Java call?

Text Example: Thought, Advice, Messages

```
// in Messages.java (a test class)
public class Messages
{
  public static void main( String[] args )
  {
    Thought parked = new Thought();
    Advice  dates  = new Advice();

    parked.message();
    dates.message();
  }
}
```

the child one, because it overrides the parent definition

Output

```
I feel diagonally parked
Dates are closer than they seem
I feel diagonally parked
```

Perks about Overriding

- A parent method declared as `final` cannot be overridden why not?
- Overloading vs. overriding

Perks about Overriding

- A parent method declared as `final` cannot be overridden why not?
- Overloading vs. overriding
  - Overloading – multiple methods …
    - With the same name
    - In the same class
    - With different signatures
  - Overriding – multiple methods …
    - With the same name
    - One in parent class, one in child class
    - With same signature

Perks about Overriding

- A parent method declared as `final` cannot be overridden why not?
- Overloading vs. overriding
  - Overloading
    - Purpose: lets you define similar operation using different input parameters
  - Overriding
    - Purpose: lets you define similar operation tailored for different class design/responsibility

Variable Naming Conventions

- Recall simple class example:

```
public class Dog
{
    private int size;
    private int age;

    public Dog( int sz, int years )
    {
        size = sz;
        age  = years;
    }
    // … rest of class
}
```

different names

Variable Naming Conventions

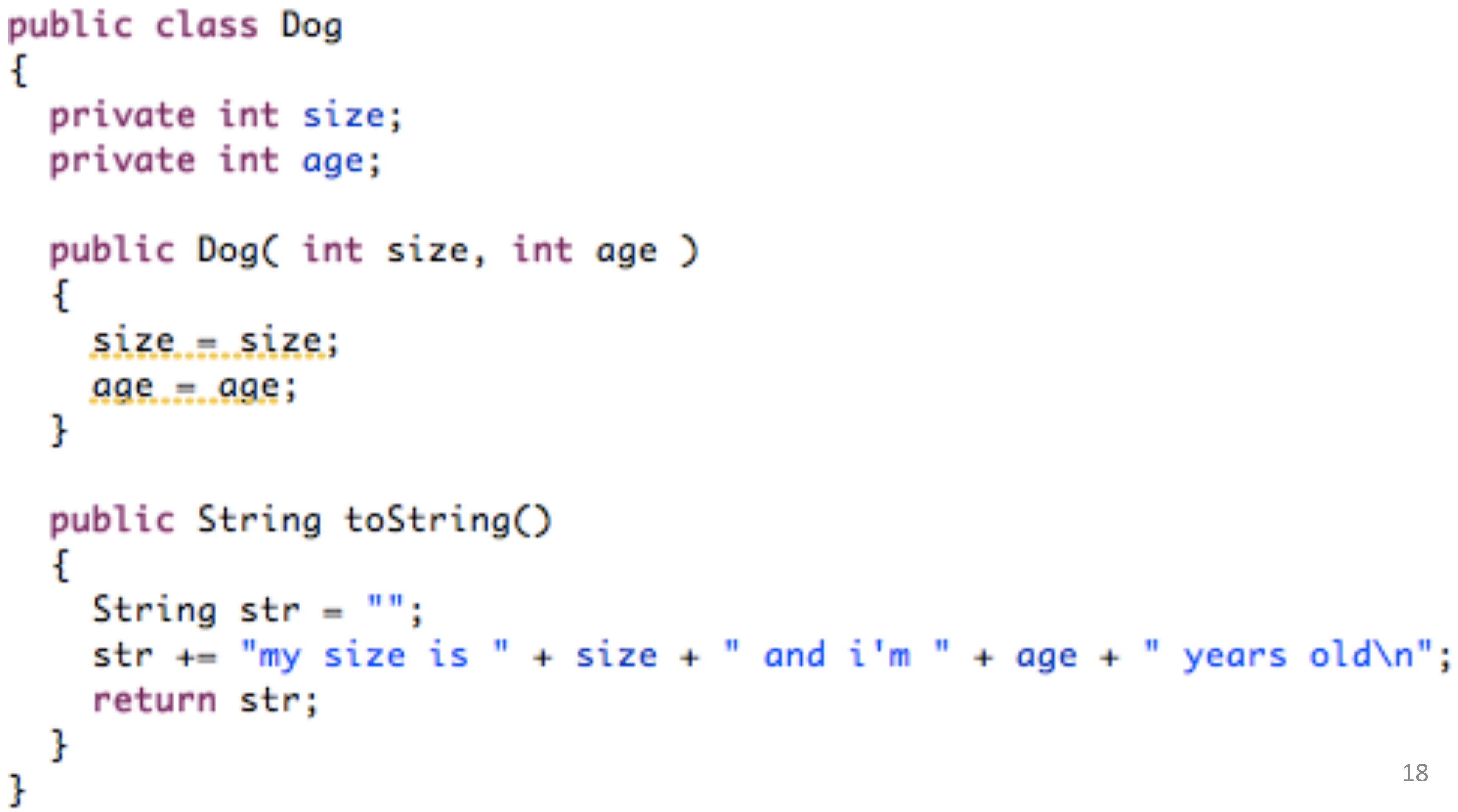
- Example with bad naming convention:

```
public class Dog
{
    private int size;
    private int age;

    public Dog( int size, int age )
    {
        size = size;
        age  = age;
    }
    // … rest of class
}
```

bad names

Variable Naming Conventions

- In fact, will this even work?



```
public class Dog
{
  private int size;
  private int age;

  public Dog( int size, int age )
  {
    size = size;
    age = age;
  }

  public String toString()
  {
    String str = "";
    str += "my size is " + size + " and i'm " + age + " years old\n";
    return str;
  }
}
```

Variable Naming Conventions

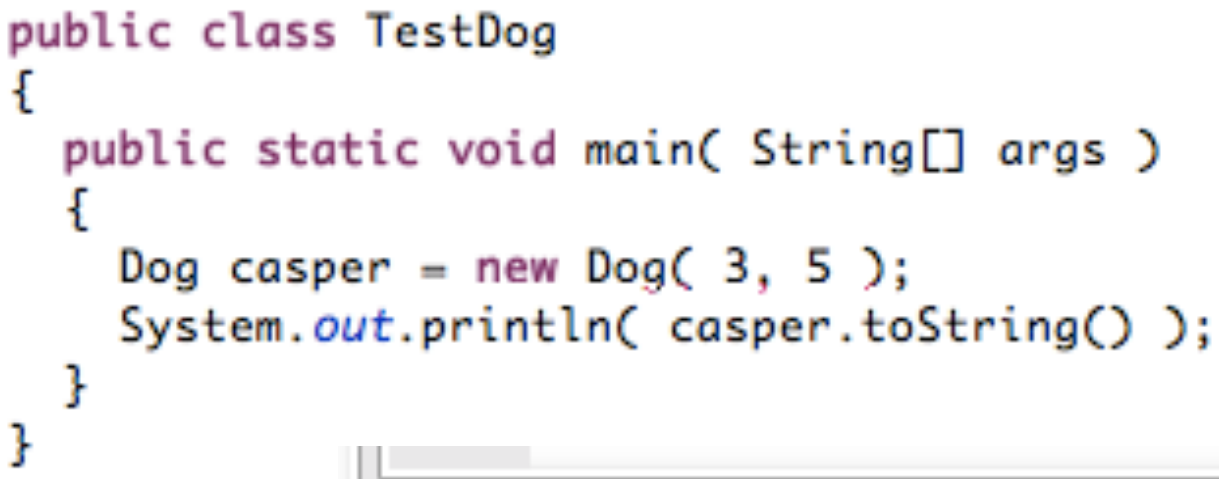
- In fact, will this even work?

```
public class Dog
{
  private int size;
  private int age;

  public Dog( int size, int age )
  {
    size = size;
    age = age;
  }

  public String toString()
  {
    String str = "";
    str += "my size is " + size + " and i'm " + age + " years old\n";
    return str;
  }
}
```

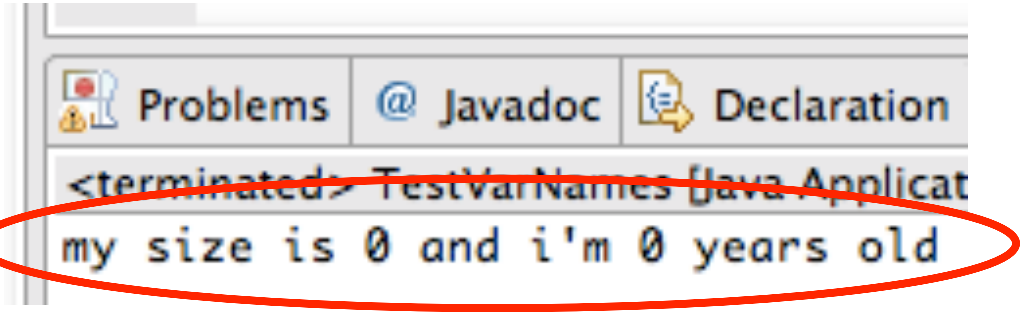
recall: scoping rule!

Variable Naming Conventions

- As it turns out, Eclipse will give default initializations for you



```
public class TestDog
{
  public static void main( String[] args )
  {
    Dog casper = new Dog( 3, 5 );
    System.out.println( casper.toString() );
  }
}
```



```
<terminated> TestVarNames [Java Applicat
my size is 0 and i'm 0 years old
```

not the values you want anyway

Similar Problem During Inheritance

- Child class inherits attributes from parent class
- Ensure you give different names
- A shadow variable is when child attribute has the same name as an inherited attribute
  - Not a scoping issue
  - Just really confusing – so don't do this!

Example

```
public class Dog
{
  private   int size;
  protected int age;
  // methods
}

public class Terrier extends Dog
{
  private   int size;
  protected int colour;
  // methods

  // if your methods make use of size in this class
  // you will access size in Terrier correctly
  // because size in Dog is private anyway
  // so this works as intended, but it's just confusing
}
```

size in Terrier shadows size in Dog

Accessing Parent Information

- Recall: all attributes and methods are inherited by children classes
- Recall use of `super`
  - Explicit reference to parent object
- Can also use `this`
  - Explicit reference to current object
  - E.g.:
    this.toString();
    this.size = super.getSize();

Accessing Parent Information (cont.)

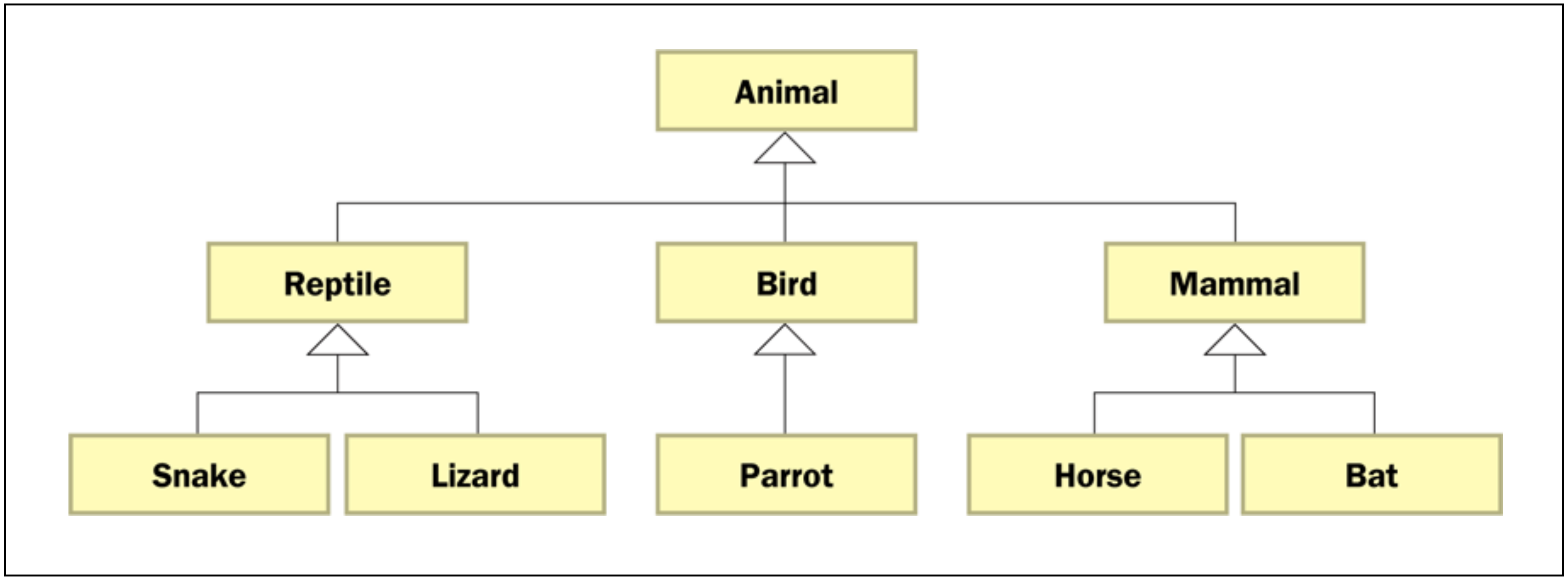
- What if parent attributes and methods are private?
  - Access indirectly: via another method

Accessing Parent Information (cont.)

- What if parent attributes and methods are private?
  - Access indirectly: via another method
  - Note: Slight wording "contradiction" in text
    Sec 9.1: all info is inherited by children
    Sec 9.4: private info not inherited by children
    Reality: Everything is inherited
    - Children classes just can't access private info from parents
    - But memory space and names are in fact created

Class Hierarchy

- Class hierarchy is the set of classes related through inheritance
- Example:



Inheritance Transitivity

- Common features (attributes/methods) should be defined as high up in the hierarchy as is reasonable
- Transitivity:
  - Inherited member is passed continually down the line
  - Example: Dog is a Mammal, Mammal is an Animal, so Dog is an Animal

The `Object` Class

- In `java.lang` (implicitly imported always), there is a class called `Object`
- All classes inherit from the `Object` class automatically
- Even basic class definitions without `extends` all are children of `Object`
- `Object` is the root of all class hierarchies

Methods in `Object`

- All children of `Object` have the following:
  - `toString()`
    - When you define `toString()` in your classes, you are overriding the inherited definition!
    - If you don't define `toString()` in your own class, you can still call the inherited definition (hard to read)
  - `equals()`
    - Inherited definition returns true if two references are aliases of the same object
    - `String` class overrides `equals()` by checking if two `String` objects have the same characters

Example

- You define:

```
public class Name
{
    private String title;
    private String firstName;
    private String middleName;
    private String lastName;
    // … rest of class
}
```

- E.g., Sir Arthur Conan Doyle
- `Name` inherits `equals()` from `Object`
- Should we override it?

Abstract Classes

- Sometimes you might want a parent class to represent a generic concept
- Use an abstract class for this purpose
  - Serves as a placeholder in class hierarchy
  - An abstract class **cannot** be instantiated!
  - Use `abstract` modifier in class template:

```
public abstract class ClassName
{
    // class contents
}
```

Defining Abstract Classes

- Option 1:
  - Attributes and methods as usual
  - Only different is class is explicitly `abstract`
- Option 2:
  - Some/all methods can also be explicitly `abstract`
  - Class is also explicitly `abstract`
  - Abstract methods have no body definitions
  - Forces children classes to either
    - Define those methods
    - Declare those methods as `abstract` too
  - Abstract methods cannot be `final` or `static`   why?

When to create abstract classes?

- When you have inheritance relationship
- When you want all subclasses to have a default behaviour
- Examples:
  - All animals can walk
  - All animals can run
  - All animals can eat
  - Even if the specific animal (a subclass) can walk, run, eat differently

Example

```
public abstract class Animal
{
  protected int numLegs;
  protected int speed;
  public abstract void walks();
  public abstract void eats();
  public int runs()
  {
    // statements to define how fast it runs based on speed
  }
}
```

can leave some methods abstract if wanted

Example

```java
public abstract class Animal
{
   protected int numLegs;
   protected int speed;
   public abstract void walks();
   public abstract void eats();
   public int runs()
   {
      // statements to define how fast it runs based on speed
   }
}
```

no constructor

- Abstract methods end with semicolon
- All subclasses must define walks() and eats() or declare them abstract too

Summary of Inheritance Concepts

- Inheritance models **IS-A** relationship

Summary of Inheritance Concepts

- Inheritance models **IS-A** relationship
- Class hierarchy has `Object` as the root
  - Push common info as high up as appropriate
  - `toString()` and `equals()` are automatically inherited as part of `Object` class

Summary of Inheritance Concepts

- Inheritance models **IS-A** relationship
- Class hierarchy has `Object` as the root
  - Push common info as high up as appropriate
  - `toString()` and `equals()` are automatically inherited as part of `Object` class
- Overriding methods (as opposed to overloading)

Summary of Inheritance Concepts

- Inheritance models **IS-A** relationship
- Class hierarchy has `Object` as the root
  - Push common info as high up as appropriate
  - `toString()` and `equals()` are automatically inherited as part of `Object` class
- Overriding methods (as opposed to overloading)
- Shadow variables are bad

Summary of Inheritance Concepts

- Inheritance models **IS-A** relationship
- Class hierarchy has `Object` as the root
  - Push common info as high up as appropriate
  - `toString()` and `equals()` are automatically inherited as part of `Object` class
- Overriding methods (as opposed to overloading)
- Shadow variables are bad
- New reserved words:
  - `protected` (in between `private` and `public`)
  - `super` vs. `this`
  - `abstract`

Summary of Inheritance Concepts

- Inheritance models **IS-A** relationship
- Class hierarchy has `Object` as the root
  - Push common info as high up as appropriate
  - `toString()` and `equals()` are automatically inherited as part of `Object` class
- Overriding methods (as opposed to overloading)
- Shadow variables are bad
- New reserved words:
  - `protected` (in between `private` and `public`)
  - `super` vs. `this`
  - `abstract`
- Rules for combined use with `final` and `static`