Inheritance and Polymorphism

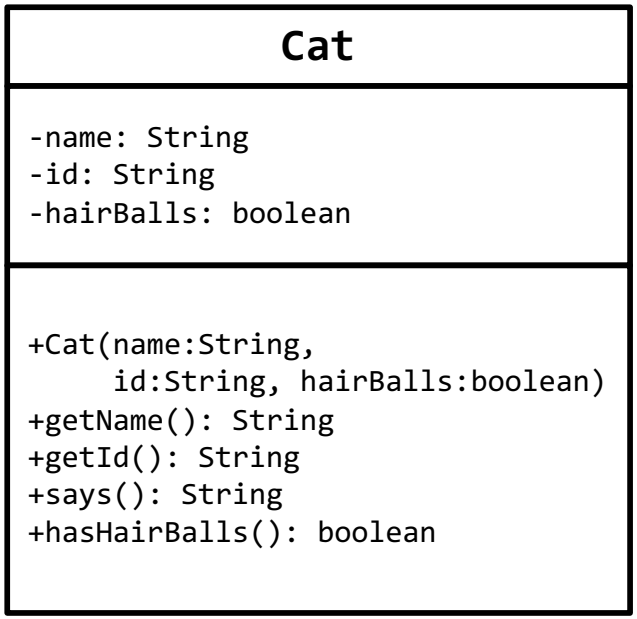
Lecture 4

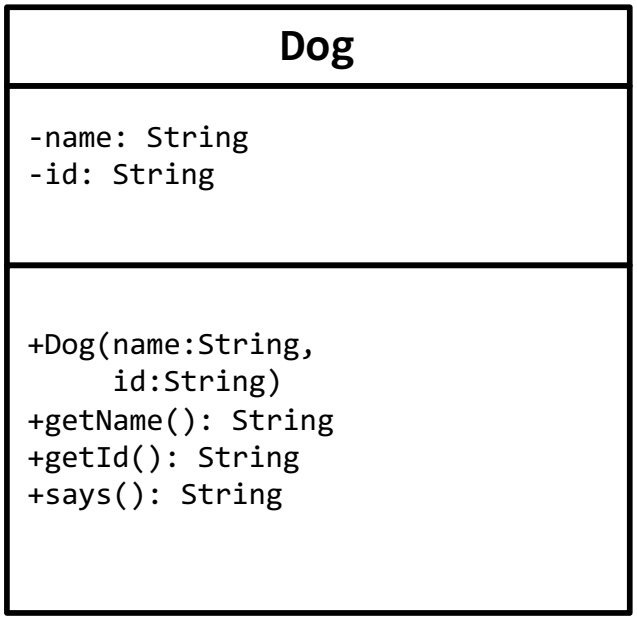
A Seemingly Simple Program

- Let's say we are writing a program to help a local vet track all their patients

- Being good OOPers, we make classes for cats and dogs, their main guests

The UML About Cats and Dogs





Code (1)

Cat.java

```
final private String name;
final private String id;
final private boolean hairBalls;

public Cat(String name, String id, boolean hairBalls) {
        this.name = name;
        this.id = id;
        this.hairBalls = hairBalls;
}

public String getName() {
        return name;
}

public String getId() {
        return id;
}

public String toString() {
        return String.format("%s (%s)",
                getName(), getId());
}

public String says() {
        return "meow";
}

public boolean hasHairBalls() {
        return hairBalls;
}
```

Dog.java

```
final private String name;
final private String id;


public Dog(String name, String id) {
        this.name = name;
        this.id = id;
}


public String getName() {
        return name;
}

public String getId() {
        return id;
}

public String toString() {
        return String.format("%s (%s)",
                getName(), getId());
}

public String says() {
        return "woof";
}
```

Code (2)

Vet.java

```
Dog[] dogs = {
      new Dog("Spot", "1234"),
      new Dog("Rover", "6789")
};

Cat[] cats = {
      new Cat("Mittens", "5432", true),
      new Cat("Garfield", "8765", false)
};

for (Dog d : dogs) {
      System.out.printf("%s says '%s'%n",
            d, d.says());
}

for (Cat c : cats) {
      System.out.printf("%s says '%s'",
            c, c.says());
      if (c.hasHairBalls()) {
            System.out.printf(" :: CLEARS THROAT ::");
      }
      System.out.printf("%n");
}
```

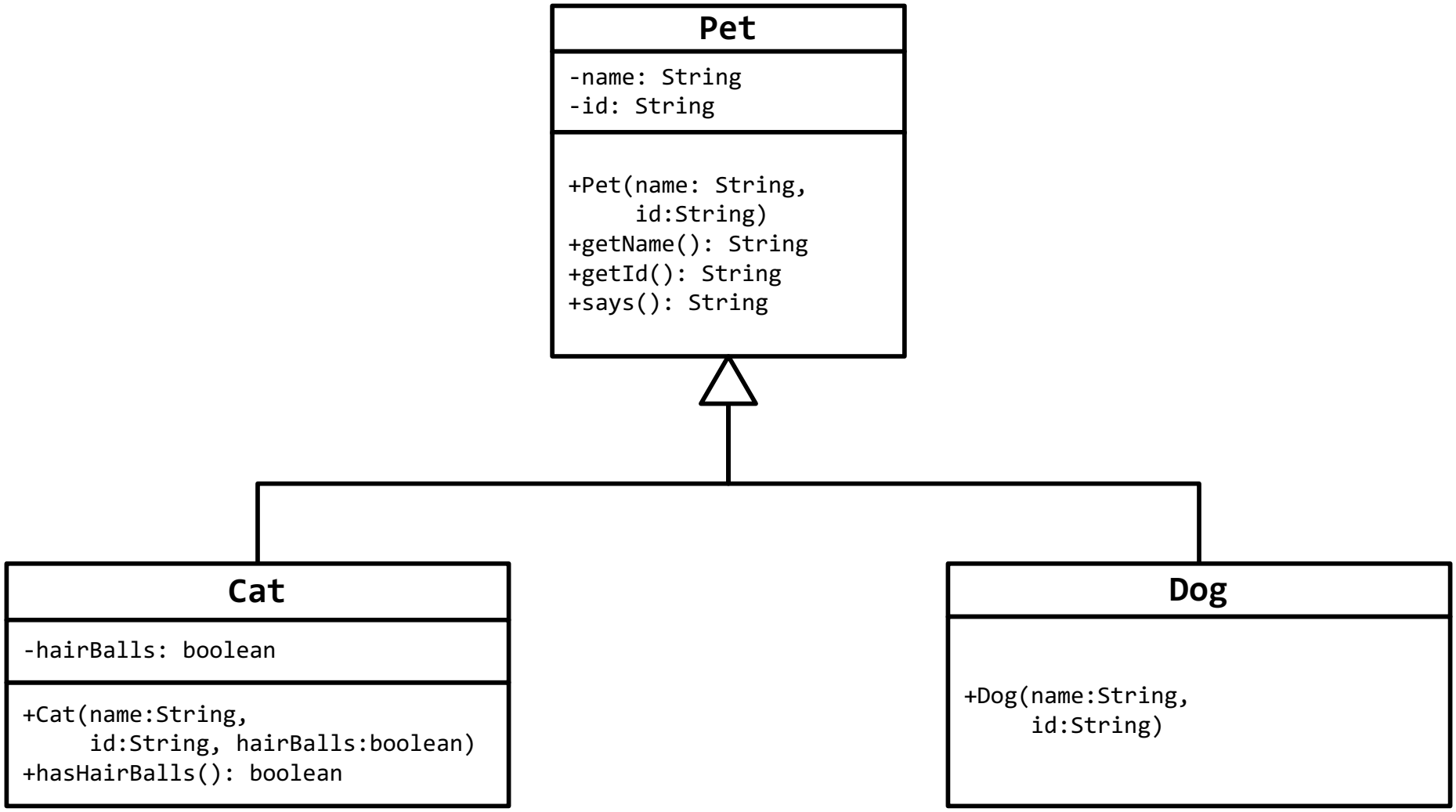
Output

```
Spot (1234) says 'woof'
Rover (6789) says 'woof'
Mittens (5432) says 'meow' :: CLEARS THROAT ::
Garfield (8765) says 'meow'
```

Alarms!

- This code *works*, but there is a great deal of duplication (which, aside from wasted effort, often leads to mistakes/bugs)
- **Inheritance** in OOP allows us to create general classes *(***superclass**), share this code amongst other classes (**subclass**), and then specialize them as needed
- **Polymorphism** allows us to write code that easily works with different subclasses that share a common superclass

Inheritance Visually



The `extends` Keyword

- In Java, classes can be *derived* from other classes, thereby inheriting fields and methods from those classes
- We begin with a more general **superclass**
- Then, any **subclass** that **extends** the superclass inherits any public or *protected* methods and/or variables of the class

Example (1)

Pet.java

```
final private String name;
final private String id;

public Pet(String name, String id) {
	this.name = name;
	this.id = id;
}

public String getName() {
	return name;
}

public String getId() {
	return id;
}

public String toString() {
	return String.format("%s (%s)",
		getName(), getId());
}

public String says() {
	return "???";
}
```

```
public class Dog2 extends Pet {
	public Dog2(String name, String id) {
		super(name, id);
	}
}
```

Because **Dog2** *extends* the **Pet** class, it gains the **getName()**, **getId()**, **toString()**, and **says()** methods!

Example (2)

Pet.java

```
final private String name;
final private String id;

public Pet(String name, String id) {
	this.name = name;
	this.id = id;
}

public String getName() {
	return name;
}

public String getId() {
	return id;
}

public String toString() {
	return String.format("%s (%s)",
		getName(), getId());
}

public String says() {
	return "???";
}
```

```
public class Cat2 extends Pet {
	final private boolean hairBalls;

	public Cat2(String name, String id,
			boolean hairBalls) {
		super(name, id);
		this.hairBalls = hairBalls;
	}

	public boolean hasHairBalls() {
		return hairBalls;
	}
}
```

Same for **Cat2**!

Example (3)

Pet.java

```
final private String name;
final private String id;

public Pet(String name, String id) {
	this.name = name;
	this.id = id;
}

public String getName() {
	return name;
}

public String getId() {
	return id;
}

public String toString() {
	return String.format("%s (%s)",
		getName(), getId());
}

public String says() {
	return "???”;
}
```

Test.java

```
Cat2 c = new Cat2("Mittens", "5432", true);
System.out.printf("%s%n", c);
System.out.printf("%b%n", c.hasHairBalls());
```

```
Mittens (5432)
true
```

The `super` Keyword

- Similar to the keyword **`this`**, which refers to the current instance, the keyword **`super`** refers to the superclass

- The keyword can be used in two ways:
  - To call a superclass constructor
  - To refer to a superclass method/variable

What About Constructors?

- Unlike properties and methods, a superclass's constructors are *not* inherited in the subclass
- They can only be invoked from the subclasses' constructors, using the keyword **`super`**
- If the keyword **`super`** is not explicitly used, the superclass's no-arg constructor is automatically invoked

Constructor Chaining

- A constructor may invoke an overloaded constructor (`this(…)`) or its superclass's constructor (`super(…)`)
  - Either of which must be the *first* line of the constructor
- If none of them is invoked explicitly, the compiler puts `super()` as the first statement in the constructor
  - Meaning, the superclass constructor is always invoked
- Thus, constructing an instance of a class invokes all the superclasses' constructors along the inheritance *chain* – this is known as **constructor chaining**
  - Intuition: set up the more general class first before initializing the specialization

Example (1)

Super.java

```
public class Super {
	public Super() {
		System.out.printf("Super()%n");
	}
}
```

Sub.java

```
public class Sub extends Super {
	public Sub() {
		this("");
		System.out.printf("Sub()%n");
	}

	public Sub(String s) {
		System.out.printf("Sub(s)%n");
	}

	public static void main(String[] args) {
		new Sub();
	}
}
```

```
Super()
Sub(s)
Sub()
```

Example (1a)

Super.java

```
public class Super {
	public Super() {
		System.out.printf("Super()%n");
	}
}
```

Sub.java

```
public class Sub extends Super {
	public Sub() {
		this("");
		System.out.printf("Sub()%n");
	}

	public Sub(String s) {
		// super();
		System.out.printf("Sub(s)%n");
	}

	public static void main(String[] args) {
		new Sub();
	}
}
```

```
Super()
Sub(s)
Sub()
```

Example (1b)

Super.java

```
public class Super {
	public Super() {
		System.out.printf("Super()%n");
	}
}
```

Sub.java

```
public class Sub extends Super {
	public Sub() {
		this("");
		System.out.printf("Sub()%n");
	}

	public Sub(String s) {
		super();
		System.out.printf("Sub(s)%n");
	}

	public static void main(String[] args) {
		new Sub();
	}
}
```

```
Super()
Sub(s)
Sub()
```

Example (1) – Trace (1)

Super.java

```
public class Super {
	public Super() {
		System.out.printf("Super()%n");
	}
}
```

Sub.java

```
public class Sub extends Super {
	public Sub() {
		this("");
		System.out.printf("Sub()%n");
	}

	public Sub(String s) {
		System.out.printf("Sub(s)%n");
	}

	public static void main(String[] args) {
		new Sub();
	}
}
```

Example (1) – Trace (2)

Super.java

```
public class Super {
	public Super() {
		System.out.printf("Super()%n");
	}
}
```

Sub.java

```
public class Sub extends Super {
	public Sub() {
		this("");
		System.out.printf("Sub()%n");
	}

	public Sub(String s) {
		System.out.printf("Sub(s)%n");
	}

	public static void main(String[] args) {
		new Sub();
	}
}
```

Example (1) – Trace (3)

Super.java

```
public class Super {
	public Super() {
		System.out.printf("Super()%n");
	}
}
```

Sub.java

```
public class Sub extends Super {
	public Sub() {
		this("");
		System.out.printf("Sub()%n");
	}

	public Sub(String s) {
		System.out.printf("Sub(s)%n");
	}

	public static void main(String[] args) {
		new Sub();
	}
}
```

Example (1) – Trace (4)

Super.java

```
public class Super {
	public Super() {
		System.out.printf("Super()%n");
	}
}
```

Sub.java

```
public class Sub extends Super {
	public Sub() {
		this("");
		System.out.printf("Sub()%n");
	}

	public Sub(String s) {
		System.out.printf("Sub(s)%n");
	}

	public static void main(String[] args) {
		new Sub();
	}
}
```

```
Super()
```

Example (1) – Trace (5)

Super.java

```
public class Super {
	public Super() {
		System.out.printf("Super()%n");
	}
}
```

Sub.java

```
public class Sub extends Super {
	public Sub() {
		this("");
		System.out.printf("Sub()%n");
	}

	public Sub(String s) {
		System.out.printf("Sub(s)%n");
	}

	public static void main(String[] args) {
		new Sub();
	}
}
```

```
Super()
Sub(s)
```

Example (1) – Trace (5)

Super.java

```
public class Super {
	public Super() {
		System.out.printf("Super()%n");
	}
}
```

Sub.java

```
public class Sub extends Super {
	public Sub() {
		this("");
		System.out.printf("Sub()%n");
	}

	public Sub(String s) {
		System.out.printf("Sub(s)%n");
	}

	public static void main(String[] args) {
		new Sub();
	}
}
```

```
Super()
Sub(s)
Sub()
```

Example (2)

Super.java

```
public class Super {
	public Super() {
		System.out.printf("Super()%n");
	}
}
```

Sub.java

```
public class Sub extends Super {
	public Sub() {
		this("");
		System.out.printf("Sub()%n");
	}

	public Sub(String s) {
		System.out.printf("Sub(s)%n");
	}

	public static void main(String[] args) {
		new Sub("");
	}
}
```

```
Super()
Sub(s)
```

Exercise – Output?

```
public class Faculty extends Employee {
	public Faculty() {
		System.out.printf("Faculty()%n");
	}

	public static void main(String[] args) {
		new Faculty();
	}
}

public class Employee extends Person {
	public Employee() {
		this("Employee(s)");
		System.out.printf("Employee()%n");
	}

	public Employee(String s) {
		System.out.printf("%s%n", s);
	}
}

public class Person {
	public Person() {
		System.out.printf("Person()%n");
	}
}
```

```
Person()
Employee(s)
Employee()
Faculty()
```

Exercise – Trace (1)

```
public class Faculty extends Employee {
	public Faculty() {
		System.out.printf("Faculty()%n");
	}

	public static void main(String[] args) {
		new Faculty();
	}
}

public class Employee extends Person {
	public Employee() {
		this("Employee(s)");
		System.out.printf("Employee()%n");
	}

	public Employee(String s) {
		System.out.printf("%s%n", s);
	}
}

public class Person {
	public Person() {
		System.out.printf("Person()%n");
	}
}
```

Exercise – Trace (2)

```
public class Faculty extends Employee {
	public Faculty() {
		System.out.printf("Faculty()%n");
	}

	public static void main(String[] args) {
		new Faculty();
	}
}

public class Employee extends Person {
	public Employee() {
		this("Employee(s)");
		System.out.printf("Employee()%n");
	}

	public Employee(String s) {
		System.out.printf("%s%n", s);
	}
}

public class Person {
	public Person() {
		System.out.printf("Person()%n");
	}
}
```

Exercise – Trace (3)

```
public class Faculty extends Employee {
	public Faculty() {
		System.out.printf("Faculty()%n");
	}

	public static void main(String[] args) {
		new Faculty();
	}
}

public class Employee extends Person {
	public Employee() {
		this("Employee(s)");
		System.out.printf("Employee()%n");
	}

	public Employee(String s) {
		System.out.printf("%s%n", s);
	}
}

public class Person {
	public Person() {
		System.out.printf("Person()%n");
	}
}
```

Exercise – Trace (4)

```
public class Faculty extends Employee {
	public Faculty() {
		System.out.printf("Faculty()%n");
	}

	public static void main(String[] args) {
		new Faculty();
	}
}

public class Employee extends Person {
	public Employee() {
		this("Employee(s)");
		System.out.printf("Employee()%n");
	}

	public Employee(String s) {
		System.out.printf("%s%n", s);
	}
}

public class Person {
	public Person() {
		System.out.printf("Person()%n");
	}
}
```

Exercise – Trace (5)

```
public class Faculty extends Employee {
	public Faculty() {
		System.out.printf("Faculty()%n");
	}

	public static void main(String[] args) {
		new Faculty();
	}
}

public class Employee extends Person {
	public Employee() {
		this("Employee(s)");
		System.out.printf("Employee()%n");
	}

	public Employee(String s) {
		System.out.printf("%s%n", s);
	}
}

public class Person {
	public Person() {
		System.out.printf("Person()%n");
	}
}
```

Exercise – Trace (6)

```
public class Faculty extends Employee {
	public Faculty() {
		System.out.printf("Faculty()%n");
	}

	public static void main(String[] args) {
		new Faculty();
	}
}

public class Employee extends Person {
	public Employee() {
		this("Employee(s)");
		System.out.printf("Employee()%n");
	}

	public Employee(String s) {
		System.out.printf("%s%n", s);
	}
}

public class Person {
	public Person() {
		System.out.printf("Person()%n");
	}
}
```

```
Person()
```

Exercise – Trace (7)

```
public class Faculty extends Employee {
	public Faculty() {
		System.out.printf("Faculty()%n");
	}

	public static void main(String[] args) {
		new Faculty();
	}
}

public class Employee extends Person {
	public Employee() {
		this("Employee(s)");
		System.out.printf("Employee()%n");
	}

	public Employee(String s) {
		System.out.printf("%s%n", s);
	}
}

public class Person {
	public Person() {
		System.out.printf("Person()%n");
	}
}
```

```
Person()
Employee(s)
```

Exercise – Trace (8)

```
public class Faculty extends Employee {
	public Faculty() {
		System.out.printf("Faculty()%n");
	}

	public static void main(String[] args) {
		new Faculty();
	}
}

public class Employee extends Person {
	public Employee() {
		this("Employee(s)");
		System.out.printf("Employee()%n");
	}

	public Employee(String s) {
		System.out.printf("%s%n", s);
	}
}

public class Person {
	public Person() {
		System.out.printf("Person()%n");
	}
}
```

```
Person()
Employee(s)
Employee()
```

Exercise – Trace (9)

```
public class Faculty extends Employee {
	public Faculty() {
		System.out.printf("Faculty()%n");
	}

	public static void main(String[] args) {
		new Faculty();
	}
}

public class Employee extends Person {
	public Employee() {
		this("Employee(s)");
		System.out.printf("Employee()%n");
	}

	public Employee(String s) {
		System.out.printf("%s%n", s);
	}
}

public class Person {
	public Person() {
		System.out.printf("Person()%n");
	}
}
```

```
Person()
Employee(s)
Employee()
Faculty()
```

Exercise – What is the Error?

```
public class Apple extends Fruit {
}

public class Fruit {
   public Fruit(String s) {
      System.out.printf("Fruit(s)%n");
   }
}
```

Exercise – Output?

```
public class COMP1050 {
     public static void main(String[] args) {
          System.out.printf("before");
          new CS();
          System.out.printf(".after");
     }
}

public class CS extends WIT {
     public CS() {
          System.out.printf(".cs");
     }
}

public class WIT {
     public WIT() {
          System.out.printf(".wit");
     }
}
```

Answer

```
before.wit.cs.after
```

Getting Back to the Pets…

Pet.java

```
final private String name;
final private String id;

public Pet(String name, String id) {
	this.name = name;
	this.id = id;
}

public String getName() {
	return name;
}

public String getId() {
	return id;
}

public String toString() {
	return String.format("%s (%s)",
		getName(), getId());
}

public String says() {
	return "???";
}
```

```
public class Dog2 extends Pet {
	public Dog2(String name, String id) {
		super(name, id);
	}

	public static void main(String[] args) {
		System.out.printf("%s%n",
		  new Dog2("Lassie", "124").says());
	}
}
```

???

Overriding Superclass Methods

- Sometimes it is necessary for the subclass to *modify* the implementation of a method defined in the superclass
  - This is referred to as **method overriding**
- This is done simply via creating a method in the subclass with the same name, parameters, and return type
- Basic idea:
  - Inherit what you can (`getName()`, `getID()`)
  - Add as needed (`Cat.hasHairBalls()`)
  - Override methods that need changing (`speak()`)

Java Annotations

- Annotations are a form of metadata – they provide data about a program that is not part of the program itself
- Annotations have no direct effect on the operation of the code they annotate
  - Useful for detecting/ignoring errors
- Examples…
  - **`@SuppressWarnings("resource")`**
    - Ignores warning on unclosed **`Scanner`** on **`System.in`**
  - **`@Override`**
    - Compiler error if not overriding an existing method

Overriding says()

Pet.java

```
final private String name;
final private String id;

public Pet(String name, String id) {
	this.name = name;
	this.id = id;
}

public String getName() {
	return name;
}

public String getId() {
	return id;
}

public String toString() {
	return String.format("%s (%s)",
		getName(), getId());
}

public String says() {
	return "???";
}
```

```
public class Dog3 extends Pet {
	public Dog3(String name, String id) {
		super(name, id);
	}

	@Override
	public String says() {
		return "woof";
	}

	public static void main(String[] args) {
		System.out.printf("%s%n",
		  new Dog3("Lassie", "124").says());
	}
}
```

```
woof
```

Don't Forget Cats!

Pet.java

```
final private String name;
final private String id;

public Pet(String name, String id) {
	this.name = name;
	this.id = id;
}

public String getName() {
	return name;
}

public String getId() {
	return id;
}

public String toString() {
	return String.format("%s (%s)",
		getName(), getId());
}

public String says() {
	return "???";
}
```

```
public class Cat3 extends Pet {
	final private boolean hairBalls;

	public Cat3(String name, String id,
			boolean hairBalls) {
		super(name, id);
		this.hairBalls = hairBalls;
	}

	@Override
	public String says() {
		return "meow";
	}

	public boolean hasHairBalls() {
		return hairBalls;
	}
}
```

Another Example: `toString()`

- In Java, all classes are subclasses of the class **`Object`**
  - Equivalent to **`public class MyClass extends Object`**

- Amongst other methods, **`Object`** has the method **`toString()`** with the following return…

```
getClass().getName() + "@" +
   Integer.toHexString(hashCode())
```

- When we write a **`toString()`** method for a class, we are overriding the base implementation to provide a more descriptive output

Overriding Note (1)

- An instance method can be overridden only if it is accessible
- Thus a *private* method cannot be overridden, because it is not accessible outside its own class
- If a method defined in a subclass is private in its superclass, the two methods are completely unrelated

Overriding Note (2)

- A subclass may override a **`protected`** method in its superclass and change its visibility to **`public`**
- However, a subclass cannot reduce the accessibility of a method defined in the superclass
- For example, if a method is defined as **`public`** in the superclass, it must be defined as **`public`** in the subclass.

Visibility Modifiers

Way of specifying what code can "see" (i.e. directly access) a variable/method

- By default, operate under a "need-to-know" basis (i.e. most constraining)
- No modifier = *package-private*
- More on protected/subclass soon!

	Class	Package	Subclass	World
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	
no modifier	✓	✓		
`private`	✓			

Overriding Note (3)

- Like an instance method, a *static* method can be inherited
- However, a static method cannot be overridden
- If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is *hidden*
  - Subtly different than overriding, since both sub/super can be accessed based on class

Example

Super.java

```
public static void m1() {
     System.out.printf("Super.m1()%n");
}

public static void m2() {
     System.out.printf("Super.m2()%n");
}
```

```
public class Sub extends Super {
     public static void m2() {
          System.out.printf("Sub.m2()%n");
     }

     public static void main(String[] args) {
          Super.m1();
          Super.m2();
          Sub.m1();
          Sub.m2();
     }
}
```

```
Super.m1()
Super.m2()
Super.m1()
Sub.m2()
```

Overriding vs. Overloading (1)

A.java

```
public int f(int x) {
      return x;
}
```

```
public class B extends A {
      public int f(double x) {
            return (int) (2 * x);
      }

      public static void main(String[] args) {
            final B b = new B();
            System.out.printf("%d %d%n",
                  b.f(10), b.f(10.0));
      }
}
```

```
10 20
```

Overriding vs. Overloading (2)

C.java

```
public int f(double x) {
      return (int) x;
}
```

```
public class D extends C {
      public int f(double x) {
            return (int) (2 * x);
      }

      public static void main(String[] args) {
            final D d = new D();
            System.out.printf("%d %d%n",
                  d.f(10), d.f(10.0));
      }
}
```

```
20 20
```

Exercise

- What are the effects of keeping all of a class' constructors private?

Answer

- No one outside of the class can instantiate the class

- The class can not be inherited

The `final` Keyword

- We already know that the value of a **`final`** variable cannot be changed

```
final static double PI = 3.14159;
```

- A final class cannot be subclassed

```
final public class EndOfTheLine
```

- A final method cannot be overriden by subclasses

```
final public void lastWord()
```

Return of the Pets…

Pet.java

```
final private String name;
final private String id;

public Pet(String name, String id) {
	this.name = name;
	this.id = id;
}

public String getName() {
	return name;
}

public String getId() {
	return id;
}

public String toString() {
	return String.format("%s (%s)",
		getName(), getId());
}

public String says() {
	return "???";
}
```

Foo.java

```
public static void main(String[] args) {
	Pet p = new Pet("WhatAmI?", "-1");
}
```

Awkward...

Getting Abstract

- An **abstract** method is a method that is declared without an implementation

```
abstract public String says();
```

- If a class even one abstract method, then the class itself must be declared abstract

```
abstract public class Pet2
```

- An abstract class is a class that is declared **`abstract`** (it may or may not include abstract methods)
  - Abstract classes *cannot* be instantiated
  - Abstract classes *can* be subclassed
  - The opposite of an abstract class is a “concrete” class

`Pet2`: Now with More Abstract!

Pet2.java

```
abstract public class Pet2 {
	final private String name;
	final private String id;

	public Pet2(String name, String id) {
		this.name = name;
		this.id = id;
	}

	public String getName() {
		return name;
	}

	public String getId() {
		return id;
	}

	@Override
	public String toString() {
		return String.format("%s (%s)",
		getName(), getId());
	}

	abstract public String says();
}
```

Benefits

- Can't directly make a new **`Pet2`** object
- Any concrete subclass of **`Pet2`** must implement **`says()`**

Dog4/Cat4: Very Concrete

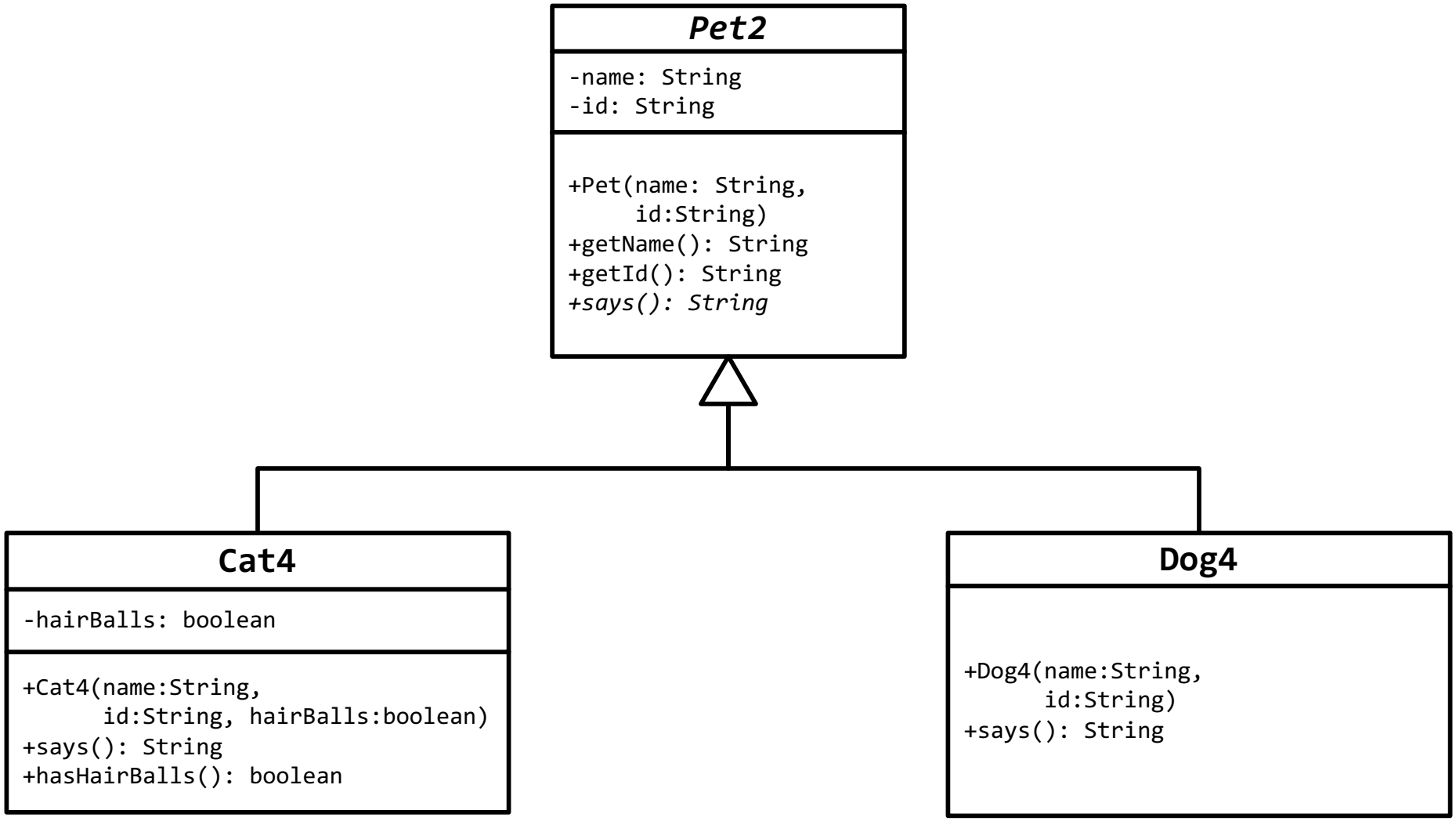
Dog4.java

```
public class Dog4 extends Pet2 {
	public Dog4(String name, String id) {
		super(name, id);
	}

	@Override  public
	String says() {
		return "woof";
	}
}
```

Cat4.java

```
public class Cat4 extends Pet2 {
	final private boolean hairBalls;

	public Cat4(String name, String id,
			boolean hairBalls) {
		super(name, id);
		this.hairBalls = hairBalls;
	}

	@Override
	public String says() {
		return "meow";
	}

	public boolean hasHairBalls() {
		return hairBalls;
	}
}
```

Abstract Visual



Revisiting the Vet

Vet.java

```
Dog[] dogs = {
      new Dog("Spot", "1234"),
      new Dog("Rover", "6789")
};

Cat[] cats = {
      new Cat("Mittens", "5432", true),
      new Cat("Garfield", "8765", false)
};

for (Dog d : dogs) {
      System.out.printf("%s says '%s'%n",
            d, d.says());
}

for (Cat c : cats) {
      System.out.printf("%s says '%s'",
            c, c.says());
      if (c.hasHairBalls()) {
            System.out.printf(" :: CLEARS THROAT ::");
      }
      System.out.printf("%n");
}
```

TODO

- Update for **`Cat4`**/**`Dog4`**
- Wouldn't it be nice if we didn't have to have as much separate code for cats/dogs?

Polymorphism

Polymorphism: a variable of a supertype can refer to a subtype object

- A class defines a *type*
- A type defined by a subclass is called a *subtype*; a type defined by its superclass is called a *supertype*
  - Example: **`Dog4`** is a subtype of **`Pet2`**; **`Pet2`** is a supertype of **`Cat4`**
- **`Object`** is the supertype of all classes

Casting Objects (1)

- Casting can used to convert an object of one class type to another within an inheritance hierarchy

- Like with primitive variables, there is implicit and explicit casting

```
double x = 5; // implicit
double y = (double) 5; // explicit
int z = (int) 3.14; // explicit
```

- In general we must cast explicitly when moving from general to specific
  - All integers are doubles, not all doubles are integers
  - In the case of objects, from superclass to subclass

Casting Objects (2)

- The following implicit cast will compile

```
Object o1 = "foo"; // all strings are objects
```

- The following will fail to compile

```
String s1 = o1; // not all objects are strings
```

- So we must explicitly cast

```
String s1 = (String) o1;
```

- The conversion is checked at *runtime* – if it was invalid, a ClassCastException will be thrown

```
String s2 = (String) 5;
```

Updating Variables in `Vet2`

```
Pet2[] pets = {
   new Dog4("Spot", "1234"),
   new Dog4("Rover", "6789"),
   new Cat4("Mittens", "5432", true),
   new Cat4("Garfield", "8765", false)
};

for (Pet2 p : pets) {
}
```

Dynamic Binding

- When an object's method is invoked, which implementation is used will be determined by the JVM at runtime – this capability is known as **dynamic binding**
  - Whereas overloading can be achieved at compile-time, overriding must occur at run-time
- The JVM searches the implementations for the method from the most specific to the most general
- Once an implementation is found, the search stops and the **first-found implementation** is invoked

Example (1)

Polly.java

```
public class Polly {
}
```

```
public static void p(Object o) {
     System.out.printf("%s%n",
          o.toString());
}

public static void main(String[] args) {
     p(new Polly());
}
```

```
Polly@70dea4e
```

Example (2)

Polly.java

```
public class Polly {
     @Override
     public String toString() {
          return "wants a cracker";
     }
}
```

```
public static void p(Object o) {
     System.out.printf("%s%n",
          o.toString());
}

public static void main(String[] args) {
     p(new Polly());
}
```

```
wants a cracker
```

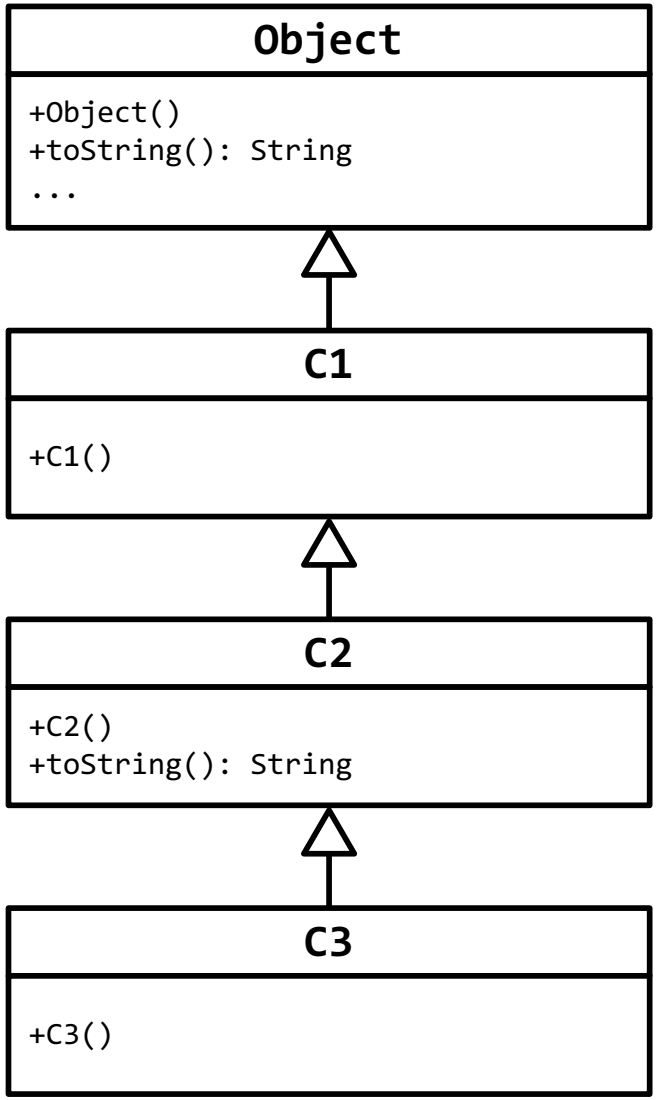
A More Complex Example

```
public class C1 {
}

public class C2 extends C1 {
	@Override
	public String toString() {
		return "C2.toString()";
	}
}

public class C3 extends C2 {
}
```

```
public static void main(String[] args) {
	final Object o1 = new C1();
	final Object o2 = new C2();
	final Object o3 = new C3();
	System.out.printf("%s%n%s%n%s%n",
		o1, o2, o3);
}
```

```
C1@70dea4e
C2.toString()
C2.toString()
```

Dynamic Binding Visually



- An instance of **`C1`** is an instance of **`C1`** and **`Object`**
  - **`C1`** is the most specific
  - **`Object`** is the most general
- An instance of **`C2`** is an instance of **`C2`**, **`C1`** and **`Object`**
  - **`C2`** is the most specific
  - **`Object`** is the most general
- An instance of **`C3`** is an instance of **`C3`**, **`C2`**, **`C1`**, and **`Object`**
  - **`C3`** is the most specific
  - **`Object`** the most general

Updating Method Calls in `Vet2`

```
Pet2[] pets = {
    new Dog4("Spot", "1234"),
    new Dog4("Rover", "6789"),
    new Cat4("Mittens", "5432", true),
    new Cat4("Garfield", "8765", false)
};

for (Pet2 p : pets) {
    System.out.printf("%s says '%s'%n",
        p, p.says());
}
```

```
Spot (1234) says 'woof'
Rover (6789) says 'woof'
Mittens (5432) says 'meow'
Garfield (8765) says 'meow'
```

Exercise – Output?

```
public class A {
	protected int i = 7;

	public A() {
		setI(20);
		System.out.printf("A(): i=%d%n", i);
	}

	public void setI(int i) {
		System.out.printf("A.setI(%d)%n", i);
		this.i = 2 * i;
	}
}

public class B extends A {
	public B() {
		System.out.printf("B(): i=%d%n", i);
	}

	public void setI(int i) {
		System.out.printf("B.setI(%d)%n", i);
		this.i = 3 * i;
	}
}
```

```
public static void main(String[] args) {
	new A();
	new B();
}
```

```
A.setI(20)
A(): i=40
B.setI(20)
A(): i=60
B(): i=60
```

Exercise – Trace (1)

```
public class A {
	protected int i = 7;

	public A() {
		setI(20);
		System.out.printf("A(): i=%d%n", i);
	}

	public void setI(int i) {
		System.out.printf("A.setI(%d)%n", i);
		this.i = 2 * i;
	}
}

public class B extends A {
	public B() {
		System.out.printf("B(): i=%d%n", i);
	}

	public void setI(int i) {
		System.out.printf("B.setI(%d)%n", i);
		this.i = 3 * i;
	}
}
```

```
public static void main(String[] args) {
	new A();
	new B();
}
```

Exercise – Trace (2)

```
public class A {
	protected int i = 7;

	public A() {
		setI(20);
		System.out.printf("A(): i=%d%n", i);
	}

	public void setI(int i) {
		System.out.printf("A.setI(%d)%n", i);
		this.i = 2 * i;
	}
}

public class B extends A {
	public B() {
		System.out.printf("B(): i=%d%n", i);
	}

	public void setI(int i) {
		System.out.printf("B.setI(%d)%n", i);
		this.i = 3 * i;
	}
}
```

```
public static void main(String[] args) {
	new A();
	new B();
}
```

Exercise – Trace (3)

```
public class A {
	protected int i = 7;

	public A() {
		setI(20);
		System.out.printf("A(): i=%d%n", i);
	}

	public void setI(int i) {
		System.out.printf("A.setI(%d)%n", i);
		this.i = 2 * i;
	}
}

public class B extends A {
	public B() {
		System.out.printf("B(): i=%d%n", i);
	}

	public void setI(int i) {
		System.out.printf("B.setI(%d)%n", i);
		this.i = 3 * i;
	}
}
```

```
public static void main(String[] args) {
	new A();
	new B();
}
```

```
A.setI(20)
```

Exercise – Trace (4)

```
public class A {
	protected int i = 7;

	public A() {
		setI(20);
		System.out.printf("A(): i=%d%n", i);
	}

	public void setI(int i) {
		System.out.printf("A.setI(%d)%n", i);
		this.i = 2 * i;
	}
}

public class B extends A {
	public B() {
		System.out.printf("B(): i=%d%n", i);
	}

	public void setI(int i) {
		System.out.printf("B.setI(%d)%n", i);
		this.i = 3 * i;
	}
}
```

```
public static void main(String[] args) {
	new A();
	new B();
}
```

```
A.setI(20)
```

Exercise – Trace (5)

```
public class A {
	protected int i = 7;

	public A() {
		setI(20);
		System.out.printf("A(): i=%d%n", i);
	}

	public void setI(int i) {
		System.out.printf("A.setI(%d)%n", i);
		this.i = 2 * i;
	}
}

public class B extends A {
	public B() {
		System.out.printf("B(): i=%d%n", i);
	}

	public void setI(int i) {
		System.out.printf("B.setI(%d)%n", i);
		this.i = 3 * i;
	}
}
```

```
public static void main(String[] args) {
	new A();
	new B();
}
```

```
A.setI(20)
A(): i=40
```

Exercise – Trace (6)

```
public class A {
	protected int i = 7;

	public A() {
		setI(20);
		System.out.printf("A(): i=%d%n", i);
	}

	public void setI(int i) {
		System.out.printf("A.setI(%d)%n", i);
		this.i = 2 * i;
	}
}

public class B extends A {
	public B() {
		System.out.printf("B(): i=%d%n", i);
	}

	public void setI(int i) {
		System.out.printf("B.setI(%d)%n", i);
		this.i = 3 * i;
	}
}
```

```
public static void main(String[] args) {
	new A();
	new B();
}
```

```
A.setI(20)
A(): i=40
```

Exercise – Trace (7)

```
public class A {
	protected int i = 7;

	public A() {
		setI(20);
		System.out.printf("A(): i=%d%n", i);
	}

	public void setI(int i) {
		System.out.printf("A.setI(%d)%n", i);
		this.i = 2 * i;
	}
}

public class B extends A {
	public B() {
		System.out.printf("B(): i=%d%n", i);
	}

	public void setI(int i) {
		System.out.printf("B.setI(%d)%n", i);
		this.i = 3 * i;
	}
}
```

```
public static void main(String[] args) {
	new A();
	new B();
}
```

```
A.setI(20)
A(): i=40
```

Exercise – Trace (8)

```
public class A {
	protected int i = 7;

	public A() {
		setI(20);
		System.out.printf("A(): i=%d%n", i);
	}

	public void setI(int i) {
		System.out.printf("A.setI(%d)%n", i);
		this.i = 2 * i;
	}
}

public class B extends A {
	public B() {
		System.out.printf("B(): i=%d%n", i);
	}

	public void setI(int i) {
		System.out.printf("B.setI(%d)%n", i);
		this.i = 3 * i;
	}
}
```

```
public static void main(String[] args) {
	new A();
	new B();
}
```

```
A.setI(20)
A(): i=40
```

Exercise – Trace (9)

```
public class A {
	protected int i = 7;

	public A() {
		setI(20);
		System.out.printf("A(): i=%d%n", i);
	}

	public void setI(int i) {
		System.out.printf("A.setI(%d)%n", i);
		this.i = 2 * i;
	}
}

public class B extends A {
	public B() {
		System.out.printf("B(): i=%d%n", i);
	}

	public void setI(int i) {
		System.out.printf("B.setI(%d)%n", i);
		this.i = 3 * i;
	}
}
```

```
public static void main(String[] args) {
	new A();
	new B();
}
```

```
A.setI(20)
A(): i=40
B.setI(20)
```

Exercise – Trace (10)

```
public class A {
	protected int i = 7;

	public A() {
		setI(20);
		System.out.printf("A(): i=%d%n", i);
	}

	public void setI(int i) {
		System.out.printf("A.setI(%d)%n", i);
		this.i = 2 * i;
	}
}

public class B extends A {
	public B() {
		System.out.printf("B(): i=%d%n", i);
	}

	public void setI(int i) {
		System.out.printf("B.setI(%d)%n", i);
		this.i = 3 * i;
	}
}
```

```
public static void main(String[] args) {
	new A();
	new B();
}
```

```
A.setI(20)
A(): i=40
B.setI(20)
```

Exercise – Trace (11)

```
public class A {
	protected int i = 7;

	public A() {
		setI(20);
		System.out.printf("A(): i=%d%n", i);
	}

	public void setI(int i) {
		System.out.printf("A.setI(%d)%n", i);
		this.i = 2 * i;
	}
}

public class B extends A {
	public B() {
		System.out.printf("B(): i=%d%n", i);
	}

	public void setI(int i) {
		System.out.printf("B.setI(%d)%n", i);
		this.i = 3 * i;
	}
}
```

```
public static void main(String[] args) {
	new A();
	new B();
}
```

```
A.setI(20)
A(): i=40
B.setI(20)
A(): i=60
```

Exercise – Trace (12)

```
public class A {
	protected int i = 7;

	public A() {
		setI(20);
		System.out.printf("A(): i=%d%n", i);
	}

	public void setI(int i) {
		System.out.printf("A.setI(%d)%n", i);
		this.i = 2 * i;
	}
}

public class B extends A {
	public B() {
		System.out.printf("B(): i=%d%n", i);
	}

	public void setI(int i) {
		System.out.printf("B.setI(%d)%n", i);
		this.i = 3 * i;
	}
}
```

```
public static void main(String[] args) {
	new A();
	new B();
}
```

```
A.setI(20)
A(): i=40
B.setI(20)
A(): i=60
B(): i=60
```

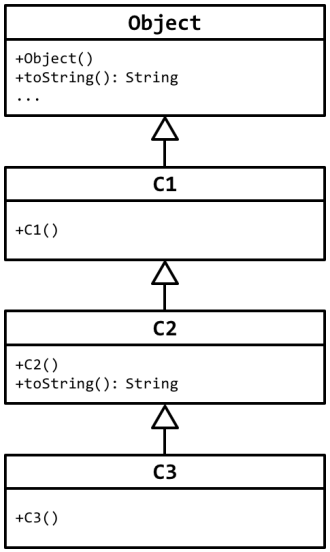
The `instanceof` Operator

Use the **`instanceof`** operator to test whether an object is an instance of a class

```
Object o1 = "howdy";
boolean isS = o1 instanceof String; // true
boolean isD = o1 instanceof Double; // false
```

A More Complex Example

```
public class C1 {
}

public class C2 extends C1 {
	@Override
	public String toString() {
		return "C2.toString()";
	}
}

public class C3 extends C2 {
}
```




```
public static void main(String[] args) {
	final Object o1 = new C1();
	final Object o2 = new C2();
	final Object o3 = new C3();
	System.out.printf("%b %b %b %b%n",
		o1 instanceof Object,
		o1 instanceof C1,
		o1 instanceof C2,
		o1 instanceof C3);
	System.out.printf("%b %b %b %b%n",
		o2 instanceof Object,
		o2 instanceof C1,
		o2 instanceof C2,
		o2 instanceof C3);
	System.out.printf("%b %b %b %b%n",
		o3 instanceof Object,
		o3 instanceof C1,
		o3 instanceof C2,
		o3 instanceof C3);

}
```

```
true true false false
true true true false
true true true true
```

Cat-Specific Method Calls in `Vet2`

```
Pet2[] pets = {
     new Dog4("Spot", "1234"),
     new Dog4("Rover", "6789"),
     new Cat4("Mittens", "5432", true),
     new Cat4("Garfield", "8765", false)
};

for (Pet2 p : pets) {
     System.out.printf("%s says '%s'", p, p.says());
     if (p instanceof Cat4) {
          final Cat4 c = (Cat4) p;
          if (c.hasHairBalls()) {
               System.out.printf(" :: CLEARS THROAT ::");
          }
     }
     System.out.printf("%n");
}
```

```
Spot (1234) says 'woof'
Rover (6789) says 'woof'
Mittens (5432) says 'meow' :: CLEARS THROAT ::
Garfield (8765) says 'meow'
```

Compare to…

Vet.java

```
Dog[] dogs = {
      new Dog("Spot", "1234"),
      new Dog("Rover", "6789")
};

Cat[] cats = {
      new Cat("Mittens", "5432", true),
      new Cat("Garfield", "8765", false)
};

for (Dog d : dogs) {
      System.out.printf("%s says '%s'%n",
            d, d.says());
}

for (Cat c : cats) {
      System.out.printf("%s says '%s'",
            c, c.says());
      if (c.hasHairBalls()) {
            System.out.printf(" :: CLEARS THROAT ::");
      }
      System.out.printf("%n");
}
```

Output

```
Spot (1234) says 'woof'
Rover (6789) says 'woof'
Mittens (5432) says 'meow' :: CLEARS THROAT ::
Garfield (8765) says 'meow'
```

The `equals` Method

- The **`equals`** method compares the contents of two objects

- The default implementation of the **`equals`** method in the **`Object`** class…

```
public boolean equals(Object obj) {
   return this == obj;
}
```

`equals` for the `Pet2` Class (1)

Pet2.java

```
// no equals method defined
```

```
final Dog4 d1 = new Dog4("d1", "1");
final Dog4 d2 = new Dog4("d2", "2");
final Dog4 lost = new Dog4("?", "1");

System.out.printf("%b%n",
    d1.equals(d2));
System.out.printf("%b%n",
    d1.equals(lost));
System.out.printf("%b%n",
    d2.equals(lost));
```


```
false
false
false
```

`equals` for the `Pet2` Class (2)

Pet2.java

```
@Override
public boolean equals(Object obj) {
    if (obj instanceof Pet2) {
        final Pet2 p = (Pet2) obj;
        return this.id.equals(p.id);
    } else {
        return false;
    }
}
```

```
final Dog4 d1 = new Dog4("d1", "1");
final Dog4 d2 = new Dog4("d2", "2");
final Dog4 lost = new Dog4("?", "1");

System.out.printf("%b%n",
    d1.equals(d2));
System.out.printf("%b%n",
    d1.equals(lost));
System.out.printf("%b%n",
    d2.equals(lost));
```

```
false
true
false
```

Take Home Points

- **Inheritance** in OOP allows us to create general classes (**superclass**), share this code amongst other classes (**subclass**), and then specialize them as needed
  - Constructor chaining
  - Method overriding (`@Override`) vs. overloading
  - Methods that are `final` cannot be overridden, `final` classes cannot be subclassed
  - Classes that are `abstract` cannot be instantiated

- **Polymorphism** allows us to write code that easily works with different subclasses that share a common superclass
  - Explicit casts are need from superclass to subclass
  - **Dynamic binding** results in calling the most specific version of a method implementation for an object
  - The `instanceof` operator allows you to determine at runtime if an object is of a particular class
  - The `equals` and `toString` methods have default implementations in the `Object` class (superclass to all classes) and should be overriden for your classes