Interfaces

Lecture 8

Remember 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");
}
```

Issues

1. **`Pet`** subclasses might forget to override **`says()`**
2. What does it mean to have an instance of a **`Pet`**?

How did we fix this?

Getting Abstract

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

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

- If a class has 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

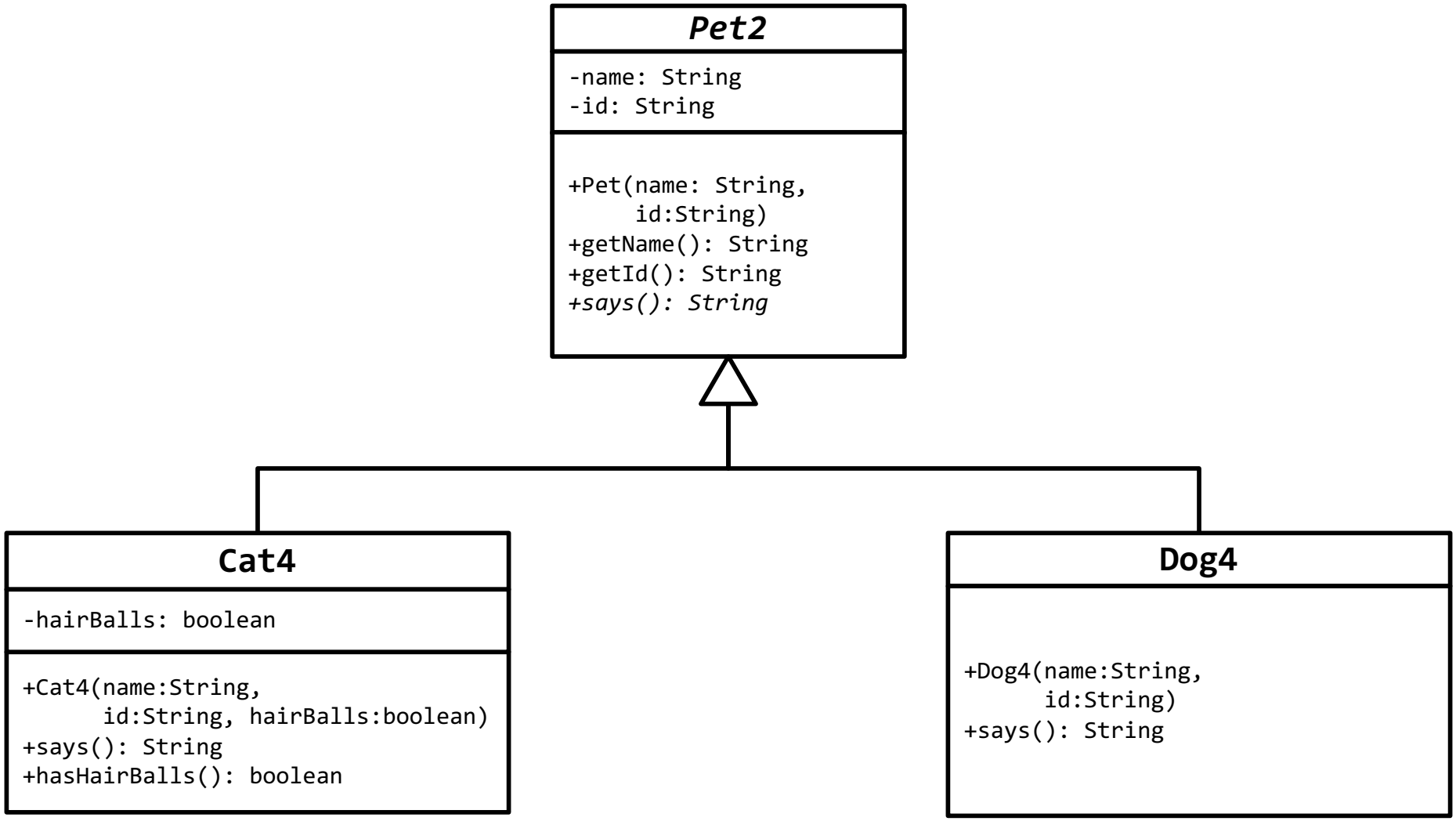
```
public class Cat4 extends Pet2 {
	final private boolean hairBalls;

	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



Notes on Abstract Classes (1)

- If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must be defined abstract

Example `Fish4` class (1)

```
abstract public class Fish4 extends Pet2 {
   public Fish4(String name, String id) {
      super(name, id);
   }

   // what sound does a fish make?
}
```

Example `Fish4` class (2)

```
public class Fish4 extends Pet2 {
   public Fish4(String id) {
      super("Fish", id);
   }

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

Example `Fish4` class (3)

```
abstract public class Fish4 extends Pet2 {
   public Fish4(String name, String id) {
      super(name, id);
   }

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

   // do not need abstract method(s) to
   // have an abstract class :)
}
```

Example `Fish4` class (4)

```
abstract public class Fish4 extends Pet2 {
   public Fish4(String name, String id) {
      super(name, id);
   }

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

   abstract public double idealSalinity();
}
```

Notes on Abstract Classes (2)

- A subclass of a concrete class can be abstract
  - Example: **`SkilledDog4`** needs to implement a **`doCoolTrick`** method

What do Abstract Classes by Us? (1)

- An abstract class creates a contract: any subclass that can be instantiated *will* have certain functionality (i.e. abstract methods define a contract)

```
public static void makeASound(Pet2 pet) {
    System.out.printf("%s%n", pet.says());
}

public static void makeAllTheSounds(Pet2[] pets) {
    for (Pet2 p : pets) {
        makeASound(p);
    }
}
```

What do Abstract Classes by Us? (2)

- Furthermore, concrete methods supply some nice default behavior (so subclasses can avoid code duplication)

- Any limitations?
  - How many "parent" classes can a class have…? *Multiple* inheritance?

Enter the Interface

- In Java, an **`interface`** is similar to an abstract class
  - Creates a "contract" (methods that must exist)
  - Cannot be instantiated

- However, whereas Java limits you to single inheritance, a class can *implement* any number of interfaces

- Because of this flexibility, it is used *extensively* in object-oriented code
  - Example: you'll need them for responding to events in your GUIs

What's in an Interface?

- Can contain only…
  - Constants, nested types (e.g. classes/interfaces)
  - Method signatures (no code, like abstract methods)
  - As of Java 8… **`static`** methods, **`default`** methods (like concrete methods in superclasses)
- All variables are **`public static final`**
  - So you may omit these
- All methods are **`public`**
  - So you may omit this

Example

Drawable.java

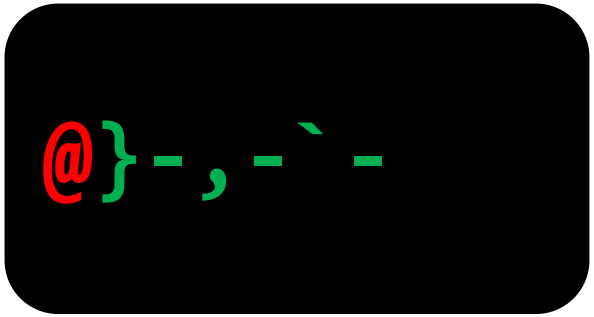
```
public interface Drawable {
     void draw();
}
```

Rose.java

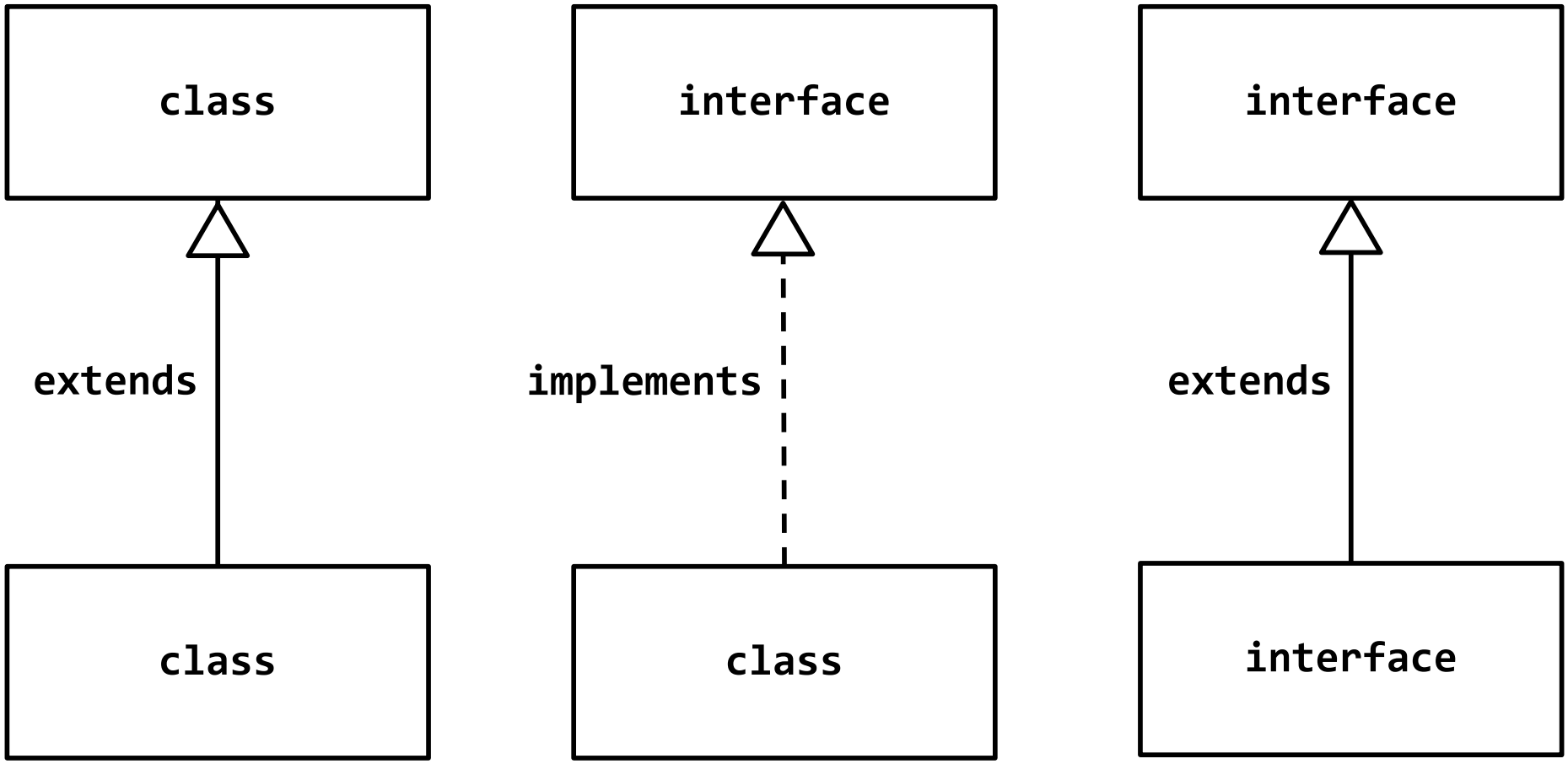
```
public class Rose implements Drawable {
     @Override
     public void draw() {
          System.out.printf("@}-,-`-");
     }
}
```

Foo.java

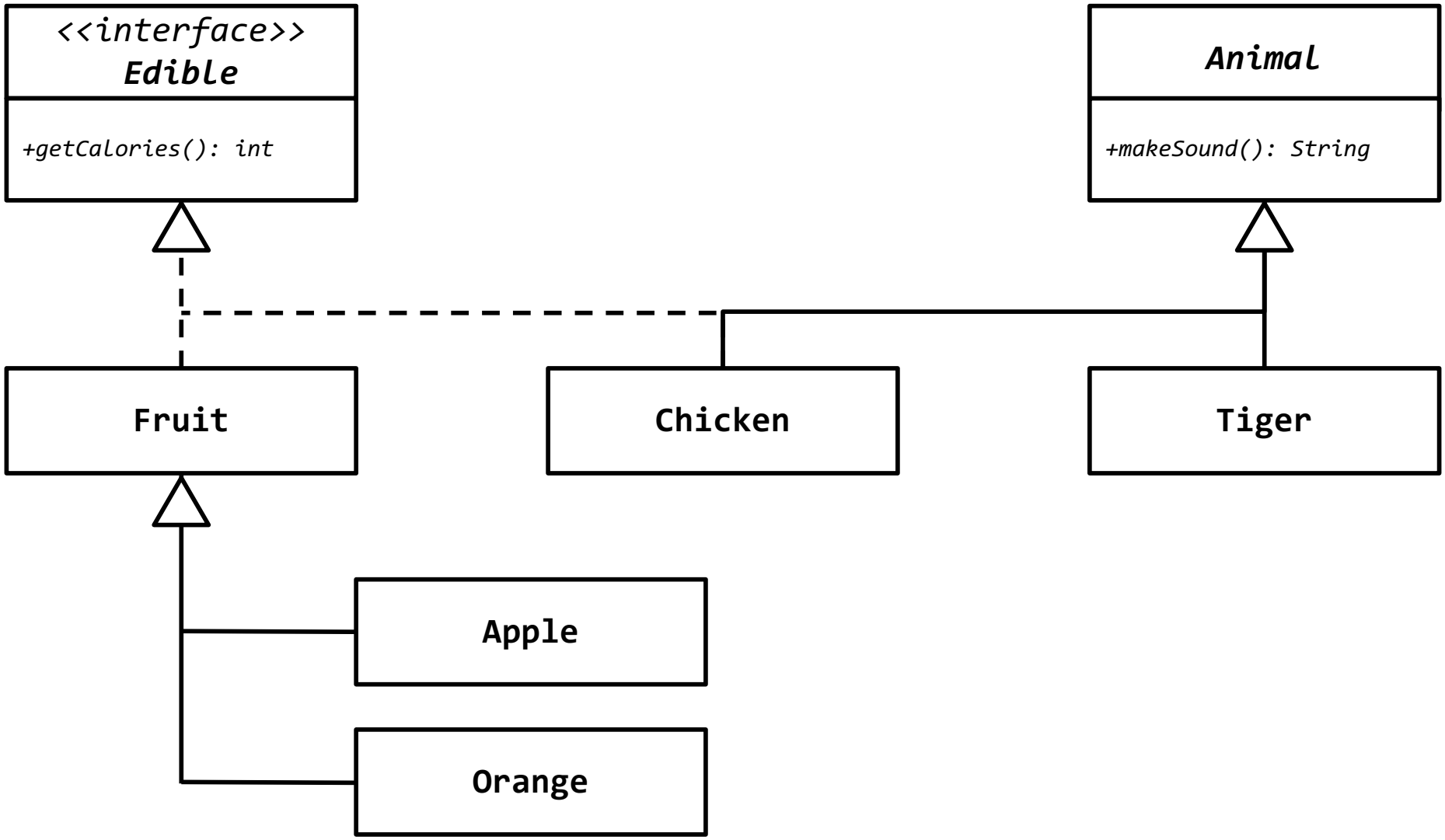
```
public class Foo {
     public static void main(String[] args) {
          final Drawable d = new Rose();
          d.draw();
     }
}
```



UML


class
interface
interface
extends
implements
extends
class
class
interface

A Complex Example



Exercise

- Create a **`Measurable`** interface, which specifies **`getPerimeter()`** and **`getArea()`** methods

- Create both **`Rectangle`** and **`Circle`** classes that implement `Measurable`

Answer

Measurable.java

```
public interface Measurable {
	double getArea();
	double getPerimeter();
}
```

Circle.java

```
public class Circle implements Measurable {
	final private double radius;

	public Circle(double radius) {
		this.radius = radius;
	}

	@Override
	public double getArea() {
		return Math.PI*radius*radius;
	}

	@Override public double getPerimeter() {
		return 2.*Math.PI*radius;
	}
}
```

Rectangle.java

```
public class Rectangle implements Measurable {
	final private double h;
	final private double w;

	public Rectangle(double h, double w) {
		this.h = h;
		this.w = w;
	}

	@Override
	public double getArea() {
		return h*w;
	}

	@Override
	public double getPerimeter() {
		return 2*(h + w);
	}
}
```

Usage

```
public static void p(Measurable m) {
        System.out.printf("p=%.3f a=%.3f%n",
            m.getPerimeter(), m.getArea());
    }

    public static void main(String[] args) {
        final Measurable m1 = new Circle(1);
        p(m1); // p=6.283 a=3.142

        p(new Rectangle(2, 3)); // p=10.000 a=6.000
    }
```

Real Example: `Comparable`

The **`Comparable<T>`** interface allows you to specify that instances your class can be compared to instances of type **`T`** (could be the same) – creates a total ordering!

```
public interface Comparable<T> {
    // Returns a negative integer, zero, or a
    // positive integer as this object is less than,
    // equal to, or greater than the specified object.
    int compareTo(T o);
}
```

Example

Circle.java

```
public class Circle implements
                    Measurable, Comparable<Circle> {
      final private double radius;

      public Circle(double radius) {
            this.radius = radius;
      }

      @Override
      public double getArea() {
            return Math.PI*radius*radius;
      }

      @Override
      public double getPerimeter() {
            return 2.*Math.PI*radius;
      }

      @Override
      public int compareTo(Circle o) {
            return Double.compare(this.radius,
                                  o.radius);
      }
}
```

```
final Comparable<Circle> c1 = new Circle(1);

System.out.printf("%d%n",
      c1.compareTo((Circle) new Circle(2)));
```

Java 8: Method Implementations

- Java 8 allows interfaces to have implementations via **`default`** or **`static`**
- Use **`static`** to disallow overriding
  - Example: utility methods
- Use **`default`** to allow overriding
  - Example: adding a new interface method with existing implemented classes

Java 8: Functional Interfaces

- A "functional" interface is one that has a single abstract method
  - Makes it easy to treat a class like a method (or "function" in other languages)

- Many such interfaces exist already in the **`java.util.function`** package…
  - **`Predicate<T>`**: return a boolean given a single parameter of type **`T`**
  - **`BiPredicate<T,U>`**: return a boolean given two parameters of type **`T`** and **`U`**

Java 8: Lambda Expressions

- A lambda expression, also termed an *anonymous function*, allows you to very succinctly create and return a class that implements a functional interface

- Syntax:
  - Parameter list, types optional, in `()`'s
    - Can omit `()`'s if only one
  - Single block, in `{}`'s if not single expression
    - If just returning a value, can omit **`return`**

Example: Lambda

I1.java

```
@FunctionalInterface
public interface I1 {
	boolean test(int x);
}
```

Foo.java

```
public static void happyOrSad(I1 var) {
	for (int i=0; i<10; i++) {
		System.out.printf("%d :", i);
		if (var.test(i)) {
			System.out.printf(")");
		} else {
			System.out.printf("(");
		}
		System.out.printf("%n");
	}
}

public static void main(String[] args) {
	happyOrSad(p->p%2 == 0);
}
```

Example: Class

I1.java

```
@FunctionalInterface
public interface I1 {
	boolean test(int x);
}
```

IsEven.java

```
public class IsEven implements I1 {
	@Override
	public boolean test(int p) {
		return p%2 == 0;
	}
}
```

Foo.java

```
public static void happyOrSad(I1 var) {
	for (int i=0; i<10; i++) {
		System.out.printf("%d :", i);
		if (var.test(i)) {
			System.out.printf(")");
		} else {
			System.out.printf("(");
		}
		System.out.printf("%n");
	}
}

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

Example: Class in a Class!

I1.java

```
@FunctionalInterface
public interface I1 {
	boolean test(int x);
}
```

Foo.java

```
public class Foo {
	private static class IsEven implements I1 {
		@Override
		public boolean test(int p) {
			return p%2 == 0;
		}
	}

	public static void happyOrSad(I1 var) {
		for (int i=0; i<10; i++) {
			System.out.printf("%d :", i);
			if (var.test(i)) {
				System.out.printf(")");
			} else {
				System.out.printf("(");
			}
			System.out.printf("%n");
		}
	}

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

Example: Anonymous Class

I1.java

```
@FunctionalInterface
public interface I1 {
	boolean test(int x);
}
```

Notes
- Existed before Java 8
- Works to create any single-use class (e.g. sub class, non-functional interface)

Foo.java

```
public static void happyOrSad(I1 var) {
	for (int i=0; i<10; i++) {
		System.out.printf("%d :", i);
		if (var.test(i)) {
			System.out.printf(")");
		} else {
			System.out.printf("(");
		}
		System.out.printf("%n");
	}
}

public static void main(String[] args) {
	happyOrSad(new I1() {
		@Override
		public boolean test(int p) {
			return p%2 == 0;
		}
	});
}
```

Interface vs. Abstract Class

- Due to single inheritance, abstract classes should be used for strong *is-a* relationships
  - Example: **`Employee`** is-a **`Person`**
- Interfaces should be used for weaker relationships – a class can do something, or has a particular property
  - Example: **`MyClass`** can handle events, be compared to **`MyClass`**, has a favorite color, …
- Interfaces define data types, and thus can be used very similarly in polymorphic code

Take Home Points

- Both interfaces and abstract classes are used to express contracts (or APIs)
- Due to single inheritance, interfaces are quite commonly used for weaker class relationships
- Functional interfaces have a single abstract method, and provide support for lambda expressions