CS 213 – Spring 2016

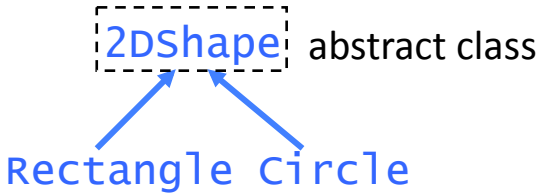
Lecture 9/10 – Feb 16/18

Abstract Classes

Abstract Classes – Introductory Examples

Rectangles and Circles have some common features:
- they can be drawn on the plane
- they have a perimeter and an area
- it can be checked whether a point is inside them or outside

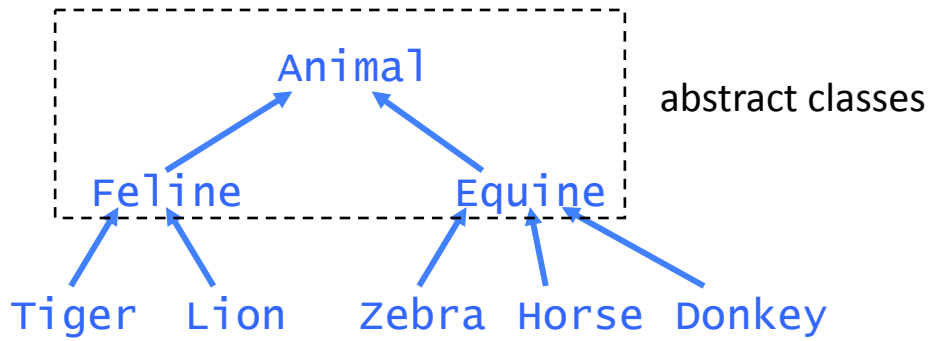
The common properties can be "abstracted" out into a superclass, say 2DShape.



This is called GENERALIZATION: gathering properties that are general to related classes into a superclass

But there is no actual 2DShape object:
only specific kinds of 2DShape objects.
So the generalized superclass is abstract

Abstract Classes – Introductory Examples



Tigers and Lions are both Felines.

Zebras, Horses, and Donkeys are Equines.

Felines and Equines are Animals.

But if you simulate, say, a forest, and populate it with specific kinds of animals, you cannot have an “Animal” or “Feline” or “Equine” object – you have to have Tigers or Lions, etc.

Abstract Class – Java Definition

When several classes have a common conceptual foundation,
i.e. common traits and behaviors,
they can be **generalized** into an abstract superclass.

A class is defined <u>abstract</u> in the class header

```
public abstract class 2DShape { … }
```

An abstract method is one that has no implementation

```
public abstract class 2DShape {
   public abstract void draw();
   public abstract float area();
   ...
}
```

Abstract Class – Java Definition

An abstract class may have <u>zero or more</u> abstract methods.

```
public abstract class Device {
   protected String name;
   protected int widthPixelDensity;
   protected int heightPixelDensity;

   public String getName() {
      return name;
   }
   public int getWidthPixelDensity() {
      return widthPixelDensity;
   }
   public int getHeightPixelDensity() {
      return heightPixelDensity;
   }
}
```

This class has NO abstract methods

Abstract Class – Java Definition

An abstract class cannot be instantiated even if all methods have been implemented.

```
public abstract class Device {
   protected String name;
   protected int horizontalResolution;
   protected int verticalResolution;

   public String getName() {
      return name;
   }
   public int getHorizontalResolution() {
      return horizontalResolution;
   }
   public int getVerticalResolution() {
      return verticalResolution;
   }
}
```

~~`Device device = new Device();`~~

Abstract Class – Java Definition

Is this an abstract class?

NO, because the class header does NOT have `abstract`

```
public class Vehicle {
   protected int numWheels;
   protected boolean hasMotor;

   public int getNumWheels() {
      return numWheels;
   }
   public boolean hasMotor() {
      return hasMotor;
   }
   public abstract int getWeight();
}
```

Abstract Class – Java Definition

An abstract class may implement non-default constructors

```
public abstract class Device {
   protected String name;
   protected int horizontalResolution;
   protected int verticalResolution;

   public Device(String name,
                 int hres, int vres) {
      this.name = name;
      horizontalResolution = hres;
      verticalResolution = vres;
   }
   ...
}
```

So what's the point of having constructors if you can't create objects?

For use by "concrete" subclasses!!

```
Device device = new Device("iPad Air",2048,1536);
```

Abstract Classes – Animal Hierarchy

```
public abstract class Animal {
    public void run() {
        System.out.println("run");
    }
}
```

```
public abstract class Feline
  extends Animal {
    public void purr() {
        System.out.println(”purr");
    }
}

public class Tiger extends Feline {
    public void purr() {
        System.out.print("Tiger: ");
        super.purr();
    }

    public void run() {
        System.out.print("Tiger: ");
        super.run();
    }
}
```

```
public abstract class Equine
  extends Animal {
    public void trot() {
        System.out.println("trot");
    }
}

public class Zebra extends Equine {
    public void trot() {
        System.out.print("Zebra: ");
        super.trot();
    }

    public void run() {
        System.out.print("Zebra: ");
        super.run();
    }
}
```

Inheritance Polymorphism

```
public class Forest {
    public static void main(String[] args) {

        Animal[] animals = new Animal[5];
        animals[0] = new Tiger();
        animals[1] = new Lion();
        animals[2] = new Zebra();
        animals[3] = new Horse();
        animals[4] = new Donkey();

        for (int i=0; i < 5; i++) {
            animals[i].run();
        }

        Equine[] equines = new Equine[3];
        equines[0] = (Equine)animals[2];
        equines[1] = (Equine)animals[3];
        equines[2] = (Equine)animals[4];

        for (int i=0; i < 3; i++) {
            equines[i].trot();
        }

    }
}
```

Static/ compile-time type is `Animal`

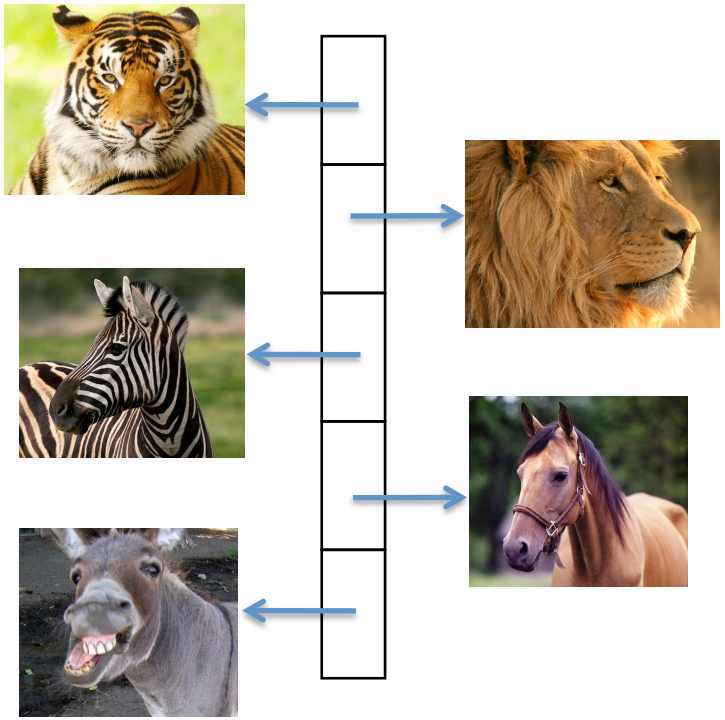
Dynamic/ run-time types are different

```
> java Forest
Tiger: run
Lion: run
Zebra: run
Horse: run
Donkey: run

Zebra: trot
Horse: trot
Donkey: trot
>
```

Polymorphism

Polymorphism



Java FX Example: Application Class

- `javafx.application.Application` is an abstract class with several non-abstract static and instance methods, and a single abstract method, namely `start`
  - `public abstract void start(Stage stage) throws Exception`

Non-GUI Abstract Classes

- Example: `java.util.Dictionary`
  - A dictionary is a data structure that allows insert, search, and delete
  - `java.util.Hashtable` is a concrete subclass of `Dictionary`
  - Any search structure (e.g. AVL tree) can implement Dictionary

- The `Dictionary` class is now obsolete, replaced by `java.util.Map` interface – why?

- The `java.util.HashMap` class implements the `Map` interface, but it also extends the `java.util.AbstractMap` abstract class – why?

Non-GUI Abstract Classes

- Example: `java.util.Calendar`
  - Provides methods to convert between specific instant in time and calendar attributes year, day, month, etc.
  - `java.util.GregorianCalendar` is a concrete subclass of `Calendar`
  - Static method `Calendar.getInstance()` returns calendar instance for current time with default locale (US – Gregorian calendar)

Default Methods in Interfaces (Java 8)

- Starting with Java 8, interfaces may have *default* methods – a default method is fully implemented. Why the need for default methods?

Default Methods in Interfaces (Java 8) Why?

Example:

Library designer ships this interface:

```
public interface Stack<T> {
   void push(T item);
   T pop() throws
       NoSuchElementException;
   boolean isEmpty();
   int size();
   void clear();
}
```

Applications build Stack implementations for this interface

Library designer decides to add a peek function:

```
public interface Stack<T> {
   ...
   T peek() throws
       NoSuchElementException;
}
```

What happens to applications that want to update to the new library?

Default Methods in Interfaces (Java 8) Why?

Example: Library updates an interface with new functionality. What happens to applications that update to the new library?

```
public class MyStack<T> implements Stack<T> {
   ...
   public void push(T item) {...}
   public T pop() throws NoSuchElementException {...}
   public boolean isEmpty() {...}
   public int size() {...}
   public void clear() {...}
}
```

This original implementation in the application will no longer compile because the peek method is not implemented

Default Methods in Interfaces (Java 8) Why?

Example: Library updates an interface with new functionality. Old code that implements this interface will no longer compile

Application has two choices:

1. Get the updated library binaries and run original implementation without recompiling (binary compatibility)

   Too restrictive, ultimately impractical

2. If other code in application changes, recompiling may be necessary, in which case implement peek, even if it is not needed (source incompatibility)

   Forces application to do unnecessary code rewrite

Default Methods in Interfaces (Java 8) Why?

Example: Library updates an interface with new functionality. Old code that implements this interface will no longer compile, UNLESS interface can provide a default implementation

```
public interface Stack<T> {
   void push(T item);
   T pop() throws NoSuchElementException;
   boolean isEmpty();
   int size();
   void clear();

   default T peek() throws NoSuchElementException {
       T temp = pop();
       push(temp);
       return temp;
   }
}
```

Default Method in Java 8 Library: Example

Previous to Java 8, the way to sort a `List` was to call static method sort in the `java.util.Collections` class, with optionally a `Comparator`

```
List<T> list = ...
MyComparator myComparator = ...
Collections.sort(list, myComparator);
```

In Java 8, the `List` interface has been updated to include a sort method so applications can sort a List by invoking it directly:

```
list.sort(myComparator);
```

The sort method is declared default (with full implementation) so that legacy code can still compile and run with previous List implementations

Default Methods and Multiple Inheritance

Since interfaces can now implement default methods, what happens if a class implements multiple interfaces that share default methods with the same signature?

```
public interface Lion {
   default void roar() {
      System.out.println
          (“Lion: roar”);
   }
}
```

```
public interface Tiger {
   default void roar() {
      System.out.println
          (“Tiger: roar”);
   }
}
```

Default Methods and Multiple Inheritance

```
public class Liger implements Lion, Tiger {
   public static void main(String[] args) {
       new Liger().roar();
   }
}
```

Will this code compile?

NO

Default Methods and Multiple Inheritance

```
public interface Lion {
   default void roar() {
      System.out.println
          (“Lion: roar”);
   }
}
```

```
public interface Tiger {
   default void roar() {
      System.out.println
          (“Tiger: roar”);
   }
}
```

Resolution: In `Liger`, override the common method, and have it explicitly call one of the default methods:

```
public class Liger implements Lion, Tiger {
   public void roar() {
       Lion.super.roar();
   }
   public static void main(String[] args) {
       new Liger().roar();
   }
}
```

Default Methods and Multiple Inheritance General Resolution Rules

Rules in order of highest to lowest priority:

1. Classes come first: A method declaration in a class takes priority over a default method declaration in an interface

```
public class Lion {
   public void roar() {
      System.out.println
          ("Lion: roar");
   }
}
```

```
public interface Tiger {
   default void roar() {
      System.out.println
          ("Tiger: roar");
   }
}
```

```
public class Liger extends Lion implements Tiger {
   public static void main(String[] args) {
      new Liger().roar();
   }
}
```

What is printed? `Lion: roar`

Default Methods and Multiple Inheritance General Resolution Rules

2. If there are only interface implementations (no subclassing), then the conflicting default method in the most specific sub-interface is used.

```
public interface Piece {
   default void move() {
      System.out.println
          (“Piece: move”);
   }
}
```

```
public interface FlexiblePiece
  extends Piece {
   default void move() {
      System.out.println
          (“FlexiblePiece: move”);
   }
}
```

```
public class SlowFlexiblePiece implements FlexiblePiece {
   public static void main(String[] args) {
       new SlowFlexiblePiece().move();
   }
}
```

What is printed? `FlexiblePiece: move`

Default Methods and Multiple Inheritance General Resolution Rules

3. If neither of the previous rules can be applied, then the class implementing the interfaces with the conflicting default methods has to explicitly pick which default method to use by:

- overriding it
- calling the desired method (as in the earlier example with `Lion.super.roar()`)

Abstract Class vs Interface

- Compare and contrast