Topic 5: Abstract Classes & Interfaces

Classes seen so far: *concrete* classes -- completely specified
Abstract class: incomplete class, useless by itself, meant for use as a superclass.
Interface: a special limited kind of abstract class

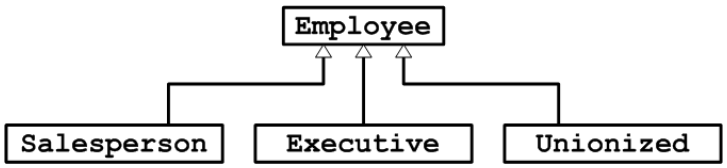
1. Abstract Classes

An abstract class is not completely specified.

- useless by itself
- can have subclasses

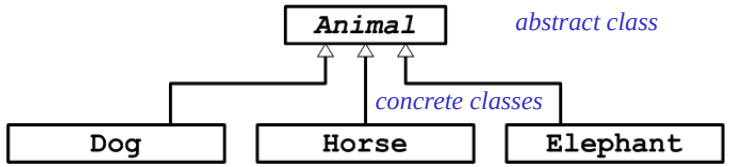
Employee Example

parent class is useful on its own
we instantiate **`Employee`** as well as sub-classes



Rationale For Abstract Classes

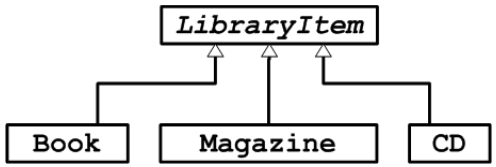
sometimes the parent class is more of a placeholder



"Animal" is a useful concept
no generic animals

Will never instantiate **`Animal`** (create a plain **`Animal`** object)
Some details left for subclasses to specify.

Another Example



UML For Abstract Classes

LibraryItem

(in italics)

or

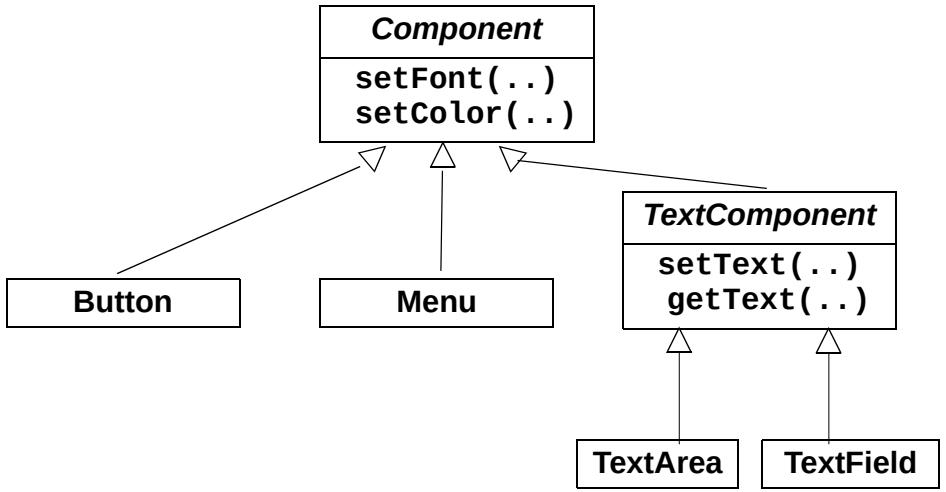
LibraryItem
{abstract}

Java For Abstract Classes

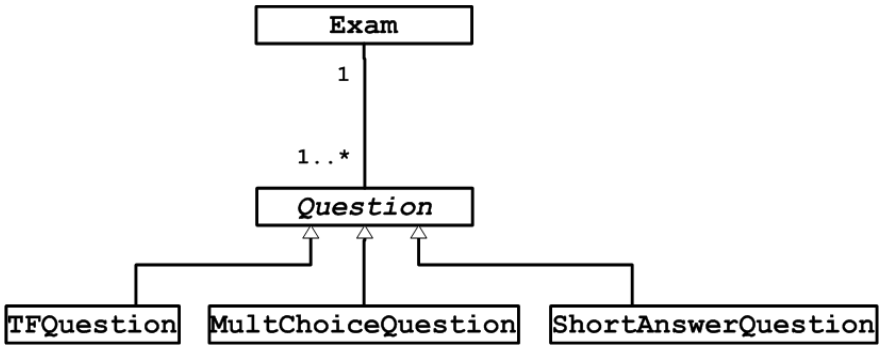
```
public abstract class LibraryItem {
   ...
}
```


Real-Life Example (simplified)

component: something you can put in a GUI window



Extended Example: Exams



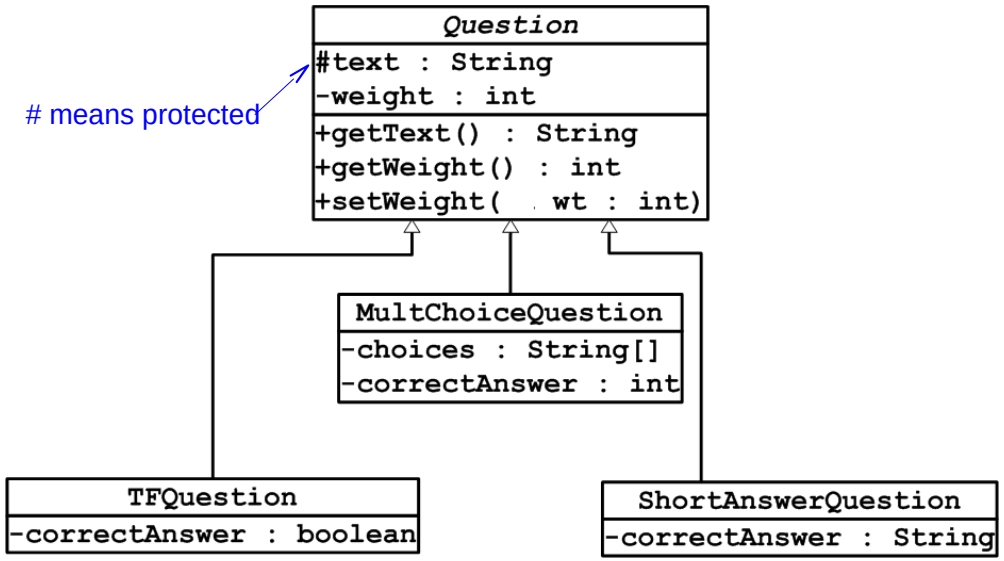
What's Common To All Questions?

- text of question
- weight (number of points)
- get & set methods for these

What's Different For Every Question Type?

1. how you specify the answers
2. how you ask the question (dialog with user)
3. additional feature for multiple choice: list of choices

Expanded UML



Asking a Question

True/False:

```
The capital of Quebec is Quebec City (enter T or F): t
correct!
```

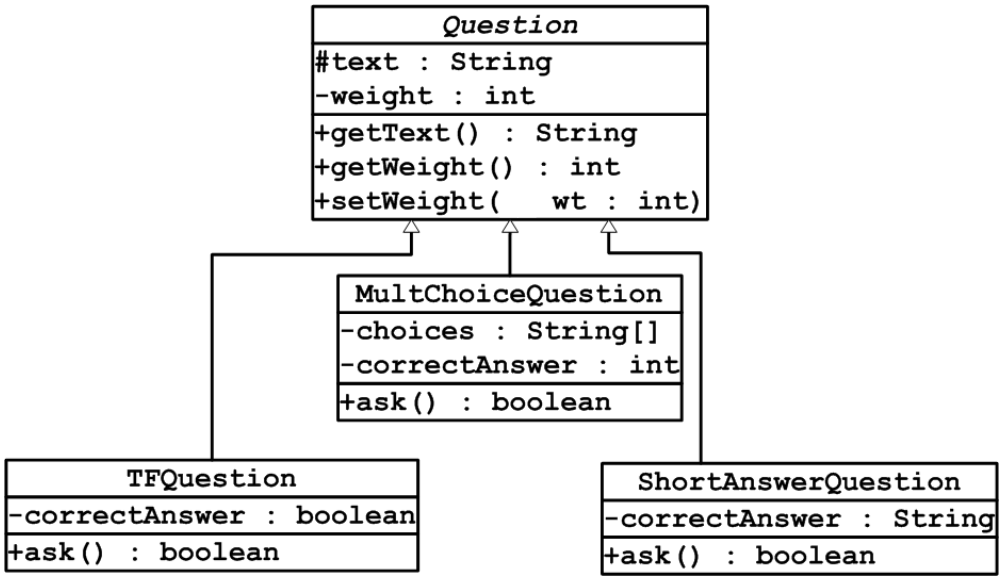
Short Answer:

```
What is the capital of Saskatchewan: Saskatoon
no, the answer is Regina
```

Multiple Choice:

```
What is the capital of Ontario?
    a. London
    b. Kingston
    c. Toronto
enter the letter of your choice: b
no, the answer is Toronto
```


ask methods



Problem

```
public class Exam {
  Question questions[];
  int numQuestions;
     ...
  public int giveExam() {
    int score = 0;
    for (int i = 0; i < numQuestions; i++) {
      if (questions[i].ask())
        score += questions[i].getWeight();
    } // end for
    return score;
  }
}
```


One More ask Method

Need to put an **`ask`** method into **`Question`** class.
What should it do??
It will never be called – just a placeholder
Every subclass will override.

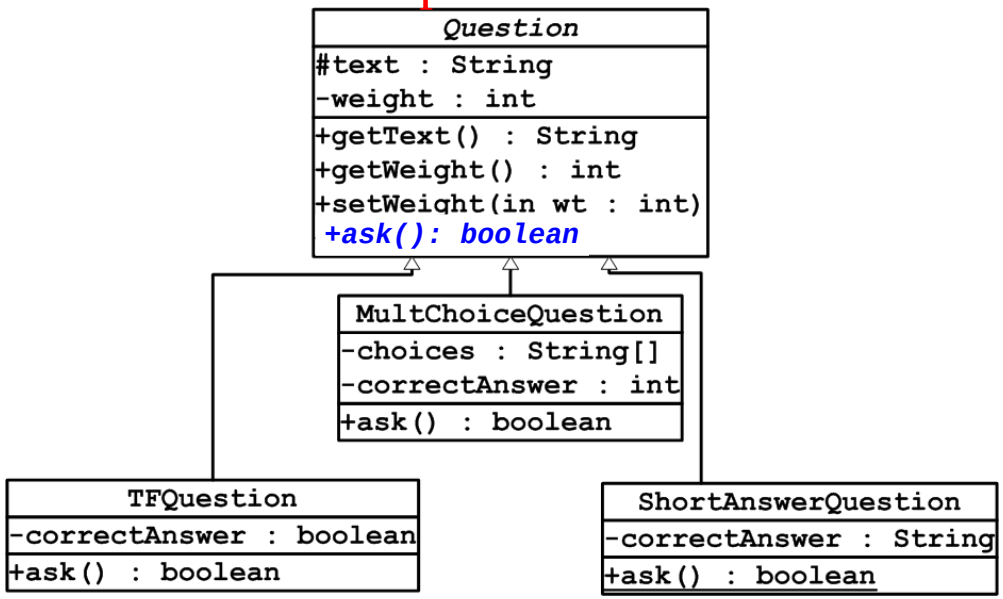
Solution: Abstract Method

```
public abstract class Question {
  ...
  public abstract boolean ask();
} // end class Question
```

Abstract method:
- no body
- will never be called
- must be overridden by every concrete subclass
- more of a placeholder than a method
- only legal inside an abstract class

The abstract ask method requires/guarantees that every subclass of **`Question`** will have an **`ask`** method.

Complete UML



Hierarchies of Abstract Classes

It is possible to have one abstract class extend another
Only the *concrete* classes must specify everything

```
public abstract class A {
  public abstract void one();
  public abstract int two();
}
```

```
public abstract class B extends A {
  public void one() {..};
  public abstract String three();
}
```

```
public class C extends B {
  public int two() {..}
  public String three() {..}
}
```


Abstract Classes: Summary

- An abstract class is not completely specified
- An abstract class can't be instantiated
- An abstract class may contain abstract methods (no bodies)
- Concrete child classes must override these abstract methods
- An abstract class may contain attributes & concrete methods

Concrete to Abstract

Abstract classes vary in how abstract they are.
One extreme: mostly concrete

- lots of data
- many concrete methods
- one abstract method to be filled in by subclasses

Other extreme: completely abstract

- no data
- only abstract methods

} Java calls this an *Interface*

Interfaces

an *Interface* is an abstract class that contains
only abstract methods – no data, no concrete methods
(may contain constants also)

an Interface is like a skeleton or template for a class

Example: Stacks

Recall from 121: Stack is an abstract data type
operations: **`push`**, **`pop`**, **`isEmpty`**
several ways to implement

Stack Interface

```
public interface Stack {
  public abstract void push(int i);
  public abstract int pop();
  public abstract boolean isEmpty();
} // end interface Stack
```

Note: all methods in an interface are public and abstract.
You can leave out those keywords

```
public interface Stack {
  void push(int i);
  int pop();
  boolean isEmpty();
} // end interface Stack
```

Terminology

You *extend* an abstract class:

```
public class TFQuestion extends Question {...
}
```

You *implement* an interface:

```
public class ArrayStack implements Stack {...
}
```


A Stack Implementation

```
public class ArrayStack implements Stack {
  private int elements[];
  private int topIndex;

 public ArrayStack() {
   elements = new int[100];
   topIndex = -1;
 } // end constructor


 public void push(int i) {...}
 public int pop() {...}

 public boolean isEmpty() {
   return topIndex == -1;
 } // end isEmpty
} // end class ArrayStack
```


Another Stack Implementation

```
public class LinkedStack implements Stack {
  // private instance variable: head of linked list


  public LinkedStack() {
    ...
  } // end constructor


  public void push(int i) {...}
  public int pop() {...}

  public boolean isEmpty() {
    ...
  } // end isEmpty

} // end class LinkedStack
```


Using Stacks (1)

```
Stack s = new ArrayStack();
   ...
s.push(x);
s.push(y);
z = s.pop();
```

constructor is only mention of which **`Stack`** implementation used

to change implementation, just change which constructor called

Using Stacks (2)

```
public static void push5(Stack theStack) {
  for (int i = 1; i <= 5; i++)
    theStack.push(i);
}
```

Parameter can be of any class that implements **`Stack`**.

Multiple Inheritance Revisited

recall: a class can *extend* only one other class

but a class can *implement* many interfaces

distinction between abstract classes & interfaces is Java's answer to multiple inheritance

Syntax:

```
public class String extends Object
  implements Comparable, Serializable {....}
```


Example: Comparable Interface

To motivate, think about how to write a general-purpose sorting method (sorts any array of objects).

First try:

```
public static void sortArray(Object array[]) {
    // selection sort
    for (int i = 0; i < array.length-1; i++) {
        int minIndex = i;
        for (int j = i+1; j < array.length; j++) {
            if (array[j] < array[minIndex])
                minIndex = j;
        } // end for j
        // swap elements at minIndex and i
        Object temp = array[minIndex];
        array[minIndex] = array[i];
        array[i] = temp;
    } // end for i
} // end sortArray
```

Doesn't compile. Why? "<" operator not defined for objects

Second Try

Use **`compareTo`** instead of "<"

```
public static void sortArray(Object array[]) {
    // selection sort
    for (int i = 0; i < array.length-1; i++) {
        int minIndex = i;
        for (int j = i+1; j < array.length; j++) {
            if (array[j].compareTo(array[minIndex]) < 0)
                minIndex = j;
        } // end for j
        // swap elements at minIndex and i
        Object temp = array[minIndex];
        array[minIndex] = array[i];
        array[i] = temp;
    } // end for i
} // end sortArray
```

Doesn't compile: **`Object`** class doesn't have a **`compareTo`** method.

Solution: the Comparable Interface

Many classes have a **`compareTo`** method, but some don't.
No default **`compareTo`** method inside the **`Object`** class.

```
// part of java.lang package
interface Comparable {
   int compareTo(Object other);
}
```

Many (but not all) API classes implement **`Comparable`**.

BlueJ: version of **`Employee`** implementing **`Comparable`**
(gets a warning but runs anyway -- more details later in course)

Corrected Sorting Method

```
public static void sortArray(Comparable array[]) {
    // selection sort
    for (int i = 0; i < array.length-1; i++) {
        int minIndex = i;
        for (int j = i+1; j < array.length; j++) {
            if (array[j].compareTo(array[minIndex]) < 0)
                minIndex = j;
        } // end for j
        // swap elements at minIndex and i
        Object temp = array[minIndex];
        array[minIndex] = array[i];
        array[i] = temp;
    } // end for i
} // end sortArray
```


Compiler Warnings

```
public static void sortArray(Comparable array[]) {
    ....
    if (array[j].compareTo(array[minIndex]) < 0)
    ...
} // end sortArray
```

Java versions after 1.4 will give you a warning for using
`Comparable` like this
For now, you may ignore.
Later: generic version.

API Sorting Methods

API class **`Arrays`** provides useful static methods for arrays
(search, sort, fill with value, etc.)

includes:

```
static void sort(Object[] a)

static void sort(Object[] a,
                 int from,
                 int to)
```

(run-time check to make sure all objects implement **`Comparable`**)

Using Arrays.sort

```
import java.util.Arrays;
public class TrySort {
  public static void main(String a[]) {
    Employee payroll[] = ....;
    Arrays.sort(payroll);
} // end TrySort
```

works just like my sort method
(but faster -- merge sort)

Uses **`Employee`** class' **`compareTo`** method
(Java term: *natural ordering*)

Wrapper Classes

What if you want to sort an array of **`int`**s or **`char`**s? Not objects.

Solution: wrapper classes
Example: **`Integer`** (in **`java.lang`**)

```
public final class Integer implements Comparable {
    private int value;
    public Integer(int v) { value = v; }
    int intValue() { return value; }
    ....
    // plus useful static methods:
    public static int parseInt(String s) {
        // translates s into an int & returns it
    }
    ....
 }
```

Important note: all wrappers classes are immutable

Conversions (old way)

Before Java 1.5, all conversions had to be made explicitly:

```
int primInt;
Integer intObject;
...
intObject = new Integer(primInt); // "boxing"
...
primInt = intObject.intValue(); // "unboxing"
```


Conversions (new way)

Starting with Java 1.5, Java will do most conversions automatically

```
int primInt;
Integer intObject;
...
intObject = primInt; // automatic "boxing"
...
primInt = intObject; // automatic "unboxing"
...
intObject = 15;
primInt = 3;
Integer sum = intObject + primInt;
System.out.println(sum);
```

Other Wrapper Classes

`Character` for **`char`**
`Double` for **`double`**
`Boolean` for **`boolean`**
etc...
Documented in Java API