Abstract Classes

by David Matuszek
Edited by Diana Palsetia

Abstract methods

- You can *declare* an object without *defining* it:
  ```java
  Person p;
  ```
- Similarly, you can declare a *method* without defining it:
  ```java
  public abstract void draw(int size);
  ```
  Notice that the body of the method is missing
- A method that has been declared but not defined is an abstract method

Abstract classes I

- Any class containing an abstract method is an abstract class
- You must declare the class with the keyword `abstract`:
  ```java
  abstract class MyClass {...}
  ```
- An abstract class is *incomplete*
  - It has “missing” method bodies
- You cannot *instantiate* (create a new instance of) an abstract class

Abstract classes II

- You can extend (subclass) an abstract class
  - If the subclass defines all the inherited abstract methods, it is “complete” and can be instantiated
  - If the subclass does *not* define all the inherited abstract methods, it too must be abstract
- You can declare a class to be abstract even if it does not contain any abstract methods
  - This prevents the class from being instantiated
Why have abstract classes?

- Suppose you wanted to create a class \texttt{Shape}, with subclasses \texttt{Oval}, \texttt{Rectangle}, \texttt{Triangle}, \texttt{Hexagon}, etc.
- You don’t want to allow creation of a “Shape”
  - Only \textit{particular} shapes make sense, not \textit{generic} ones
  - If \texttt{Shape} is abstract, you can’t create a new \texttt{Shape}
  - You \textit{can} create a new \texttt{Oval}, a new \texttt{Rectangle}, etc.
- Abstract classes are good for defining a general category containing specific, “concrete” classes

An example abstract class

- \texttt{public abstract class Animal { abstract String makeNoise(); \} }
- This class cannot be instantiated
- Any non-abstract subclass of \texttt{Animal} must provide the \texttt{makeNoise} method

Why have abstract methods?

- Suppose you have a class \texttt{Animal}, but it \textit{isn’t} abstract
  - \texttt{Animal} should \textit{not} have a \texttt{makeNoise()} method
  - Each subclass of \texttt{Animal} \textit{should} have a \texttt{makeNoise()} method
- Now suppose you have a variable \texttt{Animal \texttt{someAnimal};} where \texttt{someAnimal} contains some subclass object (such as a \texttt{Dog})
  - e.g. \texttt{Animal \texttt{someAnimal = new Dog();}}
  - It is a syntax error to say \texttt{someAnimal.makeNoise()}
- If \texttt{someAnimal} was an input parameter to method, compiler can’t tell in advance what kind of value will be in the variable (i.e. object it is referring to)

Solution to Problem

- \texttt{abstract class Animal { abstract void makeNoise(); \} }
- \texttt{class Dog extends Animal { void makeNoise() { … } \} }
- \texttt{class Cat extends Animal { void makeNoise() { … } \} }
- \texttt{Animal \texttt{someAnimal = new Dog();}}
- \texttt{this is legal, because a Dog \textit{is} a Animal}
- However, \texttt{Animal \texttt{someAnimal = new Animal();}} \textit{is no longer legal}
- \texttt{someAnimal.makeNoise();}
- \texttt{this is legal, because every actual instance \textit{must} have a \texttt{makeNoise()} method}
Example with Shapes

- abstract class Shape {
  abstract void draw();
}
- class Star extends Shape {
  void draw() { ... }
  ...
}
- class Crescent extends Shape {
  void draw() { ... }
  ...
}
- Shape someShape = new Star();
- This is legal, because a Star is a Shape
- However, Shape someShape = new Shape(); is no longer legal
- someShape.draw();
- This is legal, because every actual instance must have a draw() method