Abstract Classes and Interfaces

Ric Glassey

glassey@kth.se
Outline

• Survey results

• Abstract Classes
  – **Aim:** Explain how abstract classes increase flexibility within inheritance hierarchies
  – Motivation: Foxes and Rabbits Simulation
  – Abstract Classes & Methods in Java

• Interfaces
  – **Aim:** Demonstrate how interfaces are used to create contracts and support multiple inheritance
  – Motivation: Drawing (non-)visible animals (Ant)
  – Interfaces in Java
FOXES AND RABBIT SIMULATION
Simulation

• Common use of computers is to model complex systems (weather, populations, space)
  – Complex does not imply “complicated”
    • A watch is complicated, but its behaviour is predictable
    • An ant is simple, but the collective behaviour of an ant colony cannot be explained by studying individual ants
  – Actors in a simulation can be modelled by objects
    • We could design an Ant class in Java
      – model its basic lifecycle (from cradle to grave)
      – model its state (idle, foraging, attacking, etc)
      – model its response to the environment (chemical signals)
    • Then we can ‘simulate’ many within an environment
Objects First with Java: Rabbits and Foxes

- See Chapter 10 & online for source code

- Basic simulation of rabbits and foxes (actors) within a field (environment)
  - All are modelled in a moderately complex object oriented system
  - **Rabbit** class models:
    - A lifecycle governed by age and survival
    - They have once primary action: run( )
  - **Fox** models:
    - A lifecycle governed by age and hunger
    - They have one primary action: hunt( )
  - **Field**
    - models a random amount of rabbits and foxes
  - **Simulator**
    - manages each step (or move) in the simulation
Demo of simulation: initial randomised state
Demo of simulation: after n steps, patterns emerge
Room for improvement

• Consider the situation where we want to also include additional animals
  – There is already a lot of common fields and behaviour amongst Rabbit and Fox
    • age, alive, location, etc
  – but key differences
    • Run ( )
    • Hunt ( )
– What is the OOP design strategy we can use?
– What should we introduce?
This is a good start, but the client class Simulator will still have to cast to Fox or Rabbit to determine which method (hunt or run) to call depending on type of animal.
The more general form of `act()` can be overridden by subclasses and implemented as needed. However we are left with a curious question: *When would we ever need to create the Animal class?*
ABSTRACT CLASSES
Motivation

• Animal would never sensibly need to be instantiated as an object
  – It does specify common fields and methods
  – Its purpose is purely to exist as a superclass
  – It declares the act() method, but this method must be abstract because the superclass can have no meaningful understanding or contribution to the subclass implementation of act()
  – However, it is very useful to make it a strict requirement that all subclasses must override the act method with a specific implementation.
New Terminology

• **Abstract Class**
  – A class that cannot instantiate objects
  – Specifies what subclasses inherit
  – Specifies what subclasses must override

• **Abstract Method**
  – A method with no implementation
  – Denoted by using the ‘abstract’ keyword in Java

• **Concrete Class**
  – A regular class that we can instantiate objects from
  – All methods are either implemented or inherited
  – What we are used to creating
public abstract class Animal
{
    // Whether the animal is alive or not.
    private boolean alive;
    // The animal's field.
    private Field field;
    // The animal's position in the field.
    private Location location;

    /**
     * Create a new animal at location in field.
     *
     * @param field The field currently occupied.
     * @param location The location within the field.
     */
    public Animal(Field field, Location location)
    {
        alive = true;
        this.field = field;
        setLocation(location);
    }
}
Abstract method and concrete methods
Restrictions of Abstract Classes

• **No instances can be created** of abstract classes
  – A compiler error will be raised

• Only abstract classes can have **abstract methods**
  – Concrete classes should always provide an implementation for a method (or inherit one)

• Abstract classes with abstract methods **force subclasses to override and implement** methods declared as abstract

• Although we cannot instantiate an abstract class, we can use it as a type and take **advantage of variable polymorphism**

• Essentially we are formalising what is common and making constraints that all future subclasses must follow correctly
Updated class diagram
INTERFACES
Increasing the flexibility

We can move act() further up the chain

With Abstract Classes, we can further increase the flexibility

Depending on how complicated the simulation needs to be, designing for flexibility with inheritance can be achieved with inheritance.
What about the Ants?

- The simulation was updated to include ants, but they are many more ants than other actors...
- Implications?
  - All actors are drawn on the field in each step
  - This scales well for rabbits, foxes and hunters, but thousands of ants will **create a performance bottleneck**
  - This is a single threaded application so essentially we have a very long list of things to draw
- Ideally, we want to have actors that are **drawable** (rabbits, foxes and hunters) and other actors that are **non-drawable** (ants, etc)
  - What we are saying is that we want our actors to inherit different behaviours (despite belonging in the same inheritance hierarchy)
The need for multiple inheritance

- Drawable
- <<Abstract>> Actor
- <<Abstract>> Animal
- Hunter
- Fox
- Rabbit
- Ant

1) Concrete classes should inherit these properties/behaviours
2) They also should inherit these properties/behaviours
3) Ants do not require to be drawn on the field
Multiple Inheritance

• Different languages implement multiple inheritance in different ways
  – Multiple inheritance (C++, Lisp, Python, etc)
  – Single inheritance (Java, C#, Ruby, etc)
    • Multiple form possible with interfaces

• Can lead to...
  – Class A **declares** method X( )
  – Class B **overrides** method X( )
  – Class C **inherits** method X( )
  – Problem?
Interfaces

• Java supports multiple inheritance through the use of interfaces
• Interfaces create a specification or contract, but provide no implementation details
  – compare to abstract and concrete classes
• Only allows:
  – Constants
  – Method signatures
  – Default methods (& method body)
  – Static methods (& method body)
• As with abstract classes, interfaces cannot be instantiated, but can be used as a static type to promote polymorphism
Example Interface: Comparable

• Very common interface to implement if you want to compare the natural ordering of objects in a class

• The interface only contains one method signature:
  – public int compareTo(Obj o)
  – returns negative integer for less than, 0 for equal, and positive integer for greater than

• If we have a Person class, and we have decided that should a group of person objects be naturally ordered by age when sorted...
public interface Comparable<T> {
    public int compareTo(T o);
}

public class Person implements Comparable<Person> {

    public int age;
    // ...

    public int compareTo(Person other) {
        if (this.age == other.age) {
            return 0;
        } else if (this.age > other.age) {
            // greater than
            return 1;
        } else {
            return -1;
        }
    }
}

Person p1 = new Person(36);
Person p2 = new Person(24);
Person p3 = new Person(16);
//..
Person p100 = new Person(45);

ArrayList<Person> alist = new ArrayList<Person>();
alist.add(p1);
alist.add(p2);
alist.add(p3);
//...
alist.add(p100);

// sort the list with a built-in sort function
// this will expect subtypes of "Comparable"
// and will use the compareTo method
Collections.sort(alist);
Chose abstract classes when...

• You want to **share code** among several closely related classes (think animals)

• You expect that classes that extend your abstract class have **many common methods or fields**

• You want to declare **non-static or non-final fields**. This enables you to define methods that can access and modify the state of the object to which they belong
When to choose interfaces...

• You expect that **unrelated classes** would implement your interface
  – Think about comparable and cloneable being implemented by many unrelated classes

• You want to **specify the behaviour** of a particular data type, but not concerned about who implements its behaviour
  – List Interface (can be implemented as ArrayList or LinkedList)

• You want to take advantage of **multiple inheritance** of several types
Readings

• Objects First with Java 5th Ed. **required**
  – Chapter 10: Further Abstraction Techniques
  – Foxes & Rabbits simulator code available online:
    • All book projects 97MB in size

• The Java Tutorial
  – Trail: Learning the Language
    • Interfaces and Inheritance
      – [http://docs.oracle.com/javase/tutorial/java/landl/](http://docs.oracle.com/javase/tutorial/java/landl/)