CS2510
MODERN PROGRAMMING LANGUAGES

Prof. Peter Edwards

p.edwards@abdn.ac.uk

Introduction

- Many object-oriented programming (OOP) languages.
  - Some support procedural and data-oriented programming (e.g. Ada 95+ and C++)
  - Some support functional programming (e.g. CLOS)
  - Newer languages don’t support other paradigms; use own imperative structures (e.g. Java & C#)
  - Some are pure OOP (e.g. Smalltalk & Ruby)

Object-Oriented Programming

- **Overview**
  - Key Principles of OOP
  - A Brief History of OOP
    - Simula
    - Smalltalk
  - Design Issues for OOP Languages
  - Object-Oriented Programming in:
    - C++
    - Java
    - Ruby
  - Closing Summary

UNIVERSITY OF
ABERDEEN

Key Principles of OOP

- Major language features:
  - Abstraction
  - Encapsulation
  - *Inheritance*
  - *Polymorphism*

Last lecture – ADTs!

Inheritance

- Productivity increases can come from re-use
  - ADTs are difficult to re-use – often need changes.
  - All ADTs are independent and at the same level.
- *Inheritance* allows new classes to be defined in terms of existing ones
  - by allowing them to inherit common parts.
- Inheritance addresses both of the above concerns:
  re-use ADTs after minor changes and define classes in a hierarchy.
- **BUT:**
  When we talk about *inheritance*
  - what do we really mean?

Inheritance

- The term, inheritance, is used in many object- oriented programming languages to describe the *subsumption* (**is-a**) relationship.
  - **A rose *is-a* flower.**
  - *Subsumption* provides the ability to create subclasses.
  - Subclasses share the structure and/or behaviour of the parent class.

- Inheritance from one class (*single inheritance*) or more than one (*multiple inheritance*).

- One disadvantage of inheritance:
  - Creates interdependencies among classes that can complicate maintenance.

Inheritance Example

```
public class Cake {

    public Cake() {}

    public void bake() {
       System.out.println("The cake is baking");
    }

    public void icing() {
       System.out.println("The cake now has icing");
    }
}
```

```
Cake normalCake = new Cake();
BirthdayCake birthCake = new BirthdayCake();

normalCake.bake();

birthCake.bake();
birthCake.putCandlesOnCake(50);
```

```
OUTPUT:
The cake is baking
The cake is baking
Putting 50 candles on the birthday cake.
```

```
public class BirthdayCake extends Cake {

    public BirthdayCake() {}

    public void putCandlesOnCake(int numberOfCandles) {
       System.out.println("Putting " + numberOfCandles +
     " candles on the birthday cake.");
    }
}
```

UNIVERSITY OF ABERDEEN

Object-Oriented Concepts

- ADTs are typically called ***classes***
- Class instances are called ***objects*** or ***instances***
- A class that inherits is a ***derived class*** or a ***subclass***
- The class from which another class inherits is a
  base class or ***superclass***
- Subprograms that define operations
  on objects are called ***methods***

Object-Oriented Concepts II

- Calls to methods are called ***messages***
- The entire collection of methods of an object is called its ***message protocol*** or ***message interface***
- Messages have two parts:
  - a *method name*
  - the *destination object*

Object-Oriented Concepts III

- There are two kinds of variables in a class:
  - – ***Class variables*** - one/class
  - – ***Instance variables*** - one/object

- There are two kinds of methods in a class:
  - – ***Class methods*** – accept messages to the class
  - – ***Instance methods*** – accept messages to objects

More on Inheritance

- Inheritance can be complicated by access controls to encapsulated entities:
  - A class can hide entities from its subclasses
  - A class can hide entities from its instances
  - A class can hide entities from its instances while allowing its subclasses to see them.
- Besides inheriting methods as is, a class can modify an inherited method
  - The new one overrides the inherited one – this is ***method overriding***.

Method Overriding Example

```
public class Cake {

.
.
.

   public void icing() {
      System.out.println("The cake now has icing");
   }
}
```

The overridden method

```
Cake normalCake = new Cake();
BirthdayCake birthCake = new BirthdayCake();

normalCake.icing();

birthCake.icing();
```

```
OUTPUT:
The cake now has icing
The cake now has gooey chocolate icing
```

```
public class BirthdayCake extends Cake {

    public BirthdayCake() {}

    public void icing() {
        System.out.println("The cake now has gooey chocolate icing");
    }

     public void putCandlesOnCake(int numberOfCandles) {
        System.out.println("Putting " + numberOfCandles +
     " candles on the birthday cake.");
     }
}
```

Types of Inheritance

Specification	The superclass defines behaviour that is implemented in the subclass but not in the superclass. Provides a way to guarantee that subclass implement the same behaviour.
Specialization	The subclass is a specialized form of the superclass that modifies (or overrides) some of the methods of the parent class.
Extension	The subclass adds new functionality (variables and/or methods) to the parent class, but does not change any inherited behaviour.
Limitation	The subclass restricts the use of some of the behaviour inherited from the superclass.
Combination	The subclass inherits features from more than one superclass (i.e. multiple inheritance).

Inheritance vs Composition

- Other relationships:
  - Composition(**has-a**)
    - **A rose *has-a* a petal**.
  - Composition allows us to construct new classes using existing ones.
  - Containing object is responsible for the lifetime of the object it holds.

```
public class Engine {
     …
}

public class Car {

   private Engine engine;

   public Car() {
      this.engine = new Engine();
   }
}
```

Often more appropriate to compose an object from its parts (has-a) than extend what it is (is-a).

Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, 1994: "Favor 'object composition' over 'class inheritance'".

Polymorphism

- From the Greek, meaning *many* (poly) *shapes* (morph).
- Polymorphism enables programmers to deal in generalities and let the execution-time environment handle the specifics.
  - Promotes extensibility and re-use.
- Several different kinds of polymorphism exist.
- Here we will consider:
  - ***method overloading*** *(ad hoc polymorphism)*
  - ***subtyping***
  - ***parametric polymorphism***

Polymorphism

- ***Method overloading***
  - A class can have two or more methods having the same name, if their argument lists are different.
  - ***Static (early) binding*** : polymorphic operation is selected at compile time.

```
public class Cake {
    public Cake() {}
    public void bake() {
       System.out.println("The cake is baking");
    }
    public void bake(int temp) {
       System.out.println("The cake is baking at ” + temp + “ C");
    }
    public void bake(char oven, int temp) {
       System.out.println("The cake is baking at ” + temp + “ C in oven” + oven);
    }
}
```

Three different variants of the ‘bake’ method.

Polymorphism

- ***Subtyping (Inclusion Polymorphism)***
  - Allows a method to be written to take an object of a certain type B, but also work correctly if passed an object that belongs to a type S that is a subtype of B.
  - *Dynamic (late) binding*: polymorphic operation is selected at run time.

Four classes, each with their own 'icing' method.

```
public class Cake {
// icing
}
public class BirthdayCake extends Cake {
// gooey chocolate icing
}
public class WeddingCake extends Cake {
// white royal icing
}
public class FiftyBirthdayCake extends BirthdayCake {
// 50th Birthday letters and icing
}
```

```
BirthdayCake susan = new BirthdayCake();
WeddingCake edwards = new WeddingCake();
FiftyBirthdayCake john = new FiftyBirthdayCake();

ArrayList<Cake> cakes = new ArrayList<Cake>();
cakes.add(susan);
cakes.add(edwards);
cakes.add(john);
for ( Cake a : cakes) { a.icing();}
```

Polymorphism

- ***Subtyping*** **(*Inclusion Polymorphism*)**
  - Relies on *upcasting* - a form of casting where we cast up the inheritance hierarchy from a subtype to a a supertype.
  - Consider this example:

```
Cake susansCake = new BirthdayCake();

susansCake.bake();
susansCake.bake(128);
susansCake.icing();
```

Perfectly legal, as BirthdayCake isa Cake.

```
OUTPUT:
The cake is baking
The cake is baking at 128 C
The cake now has gooey chocolate icing
```

Exactly the behaviour we expect for a BirthdayCake object.

  - BUT: What about this?

```
susansCake.putCandlesOnCake(49);
```

```
OUTPUT:
Error: cannot find symbol -
method putCandlesOnCake(int)
```

`putCandlesOnCake()` method does not exist in Cake class - so fails at compile time.

Polymorphism

- ***Parametric polymorphism***
  - A method or data type is written in a generic fashion - so values can be handled regardless of their type.
  - Example:
    - List with elements of arbitrary type.
  - Several OO languages provide support:
    - templates in C++,
      generics in Java

- **More on this in a later lecture!**

Overriding vs Overloading

- ***Overriding***
  - When you redefine a method that has already been defined in a base class (using the same method signature).
    - Signature - method name, number and types of parameters.
  - Resolved at runtime (*dynamic*).
- ***Overloading***
  - When you define two (or more) methods with the same name, distinguished by their method signatures.
  - Resolved at compile time (*static*).