Programming Concepts Using Java

Week 3 Revision

W03:L01: The philosophy of OO programming

- Structured programming
  - The algorithms come first
    - Design a set of procedures for specific tasks
    - Combine them to build complex systems
  - Data representation comes later
    - Design data structures to suit procedural manipulations
- Object Oriented design
  - First identify the data we want to maintain and manipulate
  - Then identify algorithms to operate on the data
- **Designing objects**
  - **Behaviour** – what methods do we need to operate on objects?
  - **State** – how does the object react when methods are invoked?
    - State is the information in the instance variables
    - **Encapsulation** – should not change unless a method operates on it

W03:L01: The philosophy of OO programming (Cont.)

- Relationship between classes
  - Dependence
    - Order needs Account to check credit status
    - Item does not depend on Account
    - Robust design minimizes dependencies, or coupling between classes
  - Aggregation
    - Order contains Item objects
  - Inheritance
    - One object is a specialized versions of another
    - ExpressOrder inherits from Order
    - Extra methods to compute shipping charges, priority handling

W03:L02: Subclasses and inheritance

- A subclass extends a parent class
- Subclass inherits instance variables and methods from the parent class
- Subclass can add more instance variables and methods
  - Can also override methods
- Subclasses cannot see private components of parent class
- Use super to access constructor of parent class
- Manager objects inherit other fields and methods from Employee
- Every Manager has a name, salary and methods to access and manipulate these.

```
public class Employee{
  private String name;
  private double salary;

  // Some Constructors ...

  // "mutator" methods
  public boolean setName(String s){ ... }
  public boolean setSalary(double x){ ... }

  // "accessor" methods
  public String getName(){ ... }
  public double getSalary(){ ... }

  // other methods
  public double bonus(float percent){
     return (percent/100.0)*salary;
  }
}
public class Manager extends Employee{
     private String secretary;
     public boolean setSecretary(name s){ ... }
     public String getSecretary(){ ... }
}
```

W03:L03: Dynamic dispatch and polymorphism

- Manager can redefine bonus()

```
double bonus(float percent){
  return 1.5*super.bonus(percent);
}
```

  - Uses parent class bonus() via super
  - Overrides definition in parent class
- Consider the following assignment

```
Employee e = new Manager(...)
```

- Can we invoke e.setSecretary()?
  - e is declared to be an Employee
  - Static typechecking – e can only refer to methods in Employee

```
public class Employee{
  private String name;
  private double salary;

  // Some Constructors ...

  // "mutator" methods
  public boolean setName(String s){ ... }
  public boolean setSalary(double x){ ... }

  // "accessor" methods
  public String getName(){ ... }
  public double getSalary(){ ... }

  // other methods
  public double bonus(float percent){
     return (percent/100.0)*salary;
  }
}
public class Manager extends Employee{
     private String secretary;
     public boolean setSecretary(name s){ ... }
     public String getSecretary(){ ... }
}
```

W03:L03: Dynamic dispatch and polymorphism (Cont.)

- What about e.bonus(p)? Which bonus() do we use?
  - Static: Use Employee.bonus()
  - Dynamic: Use Manager.bonus()
- **Dynamic dispatch** (dynamic binding, late method binding, . . .) turns out to be more useful
- **Polymorphism**
  - Every Employee in emparray "knows" how to calculate its bonus correctly!

```
Employee[] emparray = new Employee[2];
Employee e = new Employee(...);
Manager e = new Manager(...);
emparray[0] = e;
emparray[1] = m;
for (i = 0; i < emparray.length; i++){
  System.out.println(emparray[i].bonus(5.0);
}
```

```
public class Employee{
  private String name;
  private double salary;

  // Some Constructors ...

  // "mutator" methods
  public boolean setName(String s){ ... }
  public boolean setSalary(double x){ ... }

  // "accessor" methods
  public String getName(){ ... }
  public double getSalary(){ ... }

  // other methods
  public double bonus(float percent){
     return (percent/100.0)*salary;
  }
}
public class Manager extends Employee{
     private String secretary;
     public boolean setSecretary(name s){ ... }
     public String getSecretary(){ ... }
}
```

W03:L03: Dynamic dispatch and polymorphism (Cont.)

- Signature of a function is its name and the list of argument types
- Overloading: multiple methods, different signatures, choice is static
- Overriding: multiple methods, same signature, choice is static
  - Employee.bonus()
  - Manager.bonus()
- Dynamic dispatch: multiple methods, same signature, choice made at run-time

```
double[] darr = new double[100];
int[] iarr = new int[500];
...
Arrays.sort(darr);
    // sorts contents of darr
Arrays.sort(iarr);
    // sorts contents of iarr
class Arrays{
    ...
    public static void sort(double[] a){..}
        // sorts arrays of double[]
    public static void sort(int[] a){..}
        // sorts arrays of int[]
    ...
}
```

W03:L03: Dynamic dispatch and polymorphism (Cont.)

Type casting

- Consider the following assignment

  `Employee e = new Manager(...)`
- e.setSecretary() does not work
  - Static type-checking disallows this
- Type casting — convert e to Manager

  `((Manager) e).setSecretary(s)`
- Cast fails (error at run time) if e is not a Manager
- Can test if e is a Manager

```
if (e instanceof Manager){
  ((Manager) e).setSecretary(s);
}
```

```
public class Employee{
  private String name;
  private double salary;

  // Some Constructors ...

  // "mutator" methods
  public boolean setName(String s){ ... }
  public boolean setSalary(double x){ ... }

  // "accessor" methods
  public String getName(){ ... }
  public double getSalary(){ ... }

  // other methods
  public double bonus(float percent){
     return (percent/100.0)*salary;
  }
}
public class Manager extends Employee{
     private String secretary;
     public boolean setSecretary(name s){ ... }
     public String getSecretary(){ ... }
}
```

W03:L04: The Java class hierarchy

- Java does not allow multiple inheritance
  - A subclass can extend only one parent class
- The Java class hierarchy forms a tree
- The root of the hierarchy is a built-in class called Object
  - Object defines default functions like equals() and toString()
  - These are implicitly inherited by any class that we write
- When we override functions, we should be careful to check the signature
- Useful methods defined in Object

```
public boolean equals(Object o) // defaults to pointer equality
public String toString()        // converts the values of the
                                // instance variables to String
```

- For Java objects x and y, x == y invokes x.equals(y)
- To print o, use System.out.println(o+"");
  - Implicitly invokes o.toString()

W03:L05: Subtyping vs inheritance

- Class hierarchy provides both subtyping and inheritance
- Subtyping
  - Capabilities of the subtype are a superset of the main type
  - If B is a subtype of A, wherever we require an object of type A, we can use an object of type B
  - Employee e = new Manager(...); is legal
  - Compatibility of interfaces
- Inheritance
  - Subtype can reuse code of the main type
  - B inherits from A if some functions for B are written in terms of functions of A
  - Manager.bonus() uses Employee.bonus()
  - Reuse of implementations
- Using one idea (hierarchy of classes) to implement both concepts blurs the distinction between the two

W03:L06: Java modifiers

- private and public are natural artefacts of encapsulation
  - Usually, instance variables are private and methods are public
  - However, private methods also make sense
- Modifiers static and final are orthogonal to public/private
- Use private static instance variables to maintain bookkeeping information across objects in a class
  - Global serial number, count number of objects created, profile method invocations, . . .
- Usually final is used with instance variables to denote constants
- A final method cannot be overridden by a subclass
- A final class cannot be inherited
- Can also have private classes