Recap: Inheritance, Encapsulation, and Exception Handling

This chapter contains a summary of slides from MIT OpenCourseWare and lecture notes of Peter Martin and Yves Lesperance

Objective

- Let’s wrap up what we have discussed so far.
  - Inheritance
  - Abstract classes
  - Encapsulation and Information hiding
  - Exception handling

Classes and Objects

- Classes model the real world
  - e.g. Animal, Person, Bicycle, etc.
- An object is an instance of a class
  - Person me = new Person (“Tuna”):
- The object referred to by the variable with the name “me” is an instance of the Person class

Classes

Avoid duplication
Inheritance

- The class Engineer and the class Manager inherit properties from a super-class (e.g. Employee)
- Write all the generic stuff in the Employee class
- Write manager specific code in the Manager class
- Write engineer specific code in the Engineer class
- Java does not allow multiple inheritance

```java
public class Employee {
    String name;
    int salary;
}

public class Manager extends Employee {
}

public class Engineer extends Employee {
}
```

Inheritance

- Classes inherit fields and methods from their parents

```java
public class Employee {
    String name;
    int salary;
    public static void printSalary () {
        System.out.println ("Salary of " + name + " is " + salary);
    }
}

public class World {
    public static void main (String[] args) {
        Manager m = new Manager ("Joshua", 4000);
        m.printSalary ();
    }
}
```

this

- The keyword "this" refers to the current object.

```java
public class Bicycle {
    int gear;
    public Bicycle (int gear) {
        this.gear = gear;
    }
}
```
Inheritance

• Subclasses inherit the default constructor automatically (i.e. the constructor with no arguments)

```java
public class Employee {
    String name;
    int salary;
    public Employee () {
        this.name = "Joe";
        this.salary = 10000;
    }
}
```

```java
public class Manager extends Employee {
    // no need for a constructor here
}
```

CMPE160: Introduction to Object Oriented Programming 8

Inheritance

• Subclasses do not inherit non-default constructors automatically (i.e. the constructor with arguments)

```java
public class Employee {
    String name;
    int salary;
    public Employee (String name) {
        this.name = "Joe";
        this.salary = 10000;
    }
}
```

```java
public class Manager extends Employee {
    // need a constructor here!!
}
```

CMPE160: Introduction to Object Oriented Programming 9

Inheritance

• If you define a constructor in a class, you must define it in all its subclasses.

```java
public class Employee {
    String name;
    int salary;
    // constructor
    public Employee (String name, int salary) {
        this.name = name;
        this.salary = salary;
        System.out.println("Created new employee " + name);
    }
}
```

```java
public class Manager extends Employee {
    String name;
    int salary;
    // constructor
    public Manager (String name, int salary) {
        super(name, salary);
        salary += 1000;
    }
}
```

CMPE160: Introduction to Object Oriented Programming 10

Inheritance

• You can reuse the super-class constructor using super.

```java
public class Manager extends Employee {
    String name;
    int salary;
    // constructor
    public Manager (String name, int salary) {
        super(name, salary);
        salary += 1000;
    }
}
```

CMPE160: Introduction to Object Oriented Programming 11
Abstract Classes

• Sometimes, the superclass should never be instantiated (i.e. no object of that class should exist)
  - e.g. in FooCorporation, you are either an Engineer or a Manager, but not just an Employee
• An abstract class is a class that can never be implemented.
• It may have abstract methods that have no body but also regular methods.
• An abstract method must be implemented in the subclasses.

```java
public abstract class Employee {
    String name;
    int salary;

    // regular method
    public void printSalary () {
        System.out.println("My salary is "+salary);
    }

    // abstract method
    public abstract void printPizzaCredits ();
}
```

```
public class Manager extends Employee {
    public Manager(String name, int salary) {
        super(name, salary);
    }

    // implements abstract method
    public void printPizzaCredits () {
        System.out.println("No pizza credit for managers!");
    }
}
```

Interface

• An interface is like an abstract class, but all methods are abstract and all fields are final.
• All methods must be implemented in the subclasses.
• You cannot change the value of an interface field.
Polymorphism

• The ability for a method to behave differently depending on the object it is called upon.
  - void spinning (Ball b);
  - void spinning (Image g);

Overloaded methods

```java
public class World {
    public static void fire (Employee e) {
        System.out.println ("Thank you!!");
    }
    public static void fire (Manager e) {
        System.out.println ("Here is $10,000");
    }
    public static void main (String[] args) {
    }
}
```

Override methods

```java
public class Manager extends Employee {
    public void getRaised (int raise) {  
        salary += 3 * raise;  
    }
}
```

Polymorphism

• Overloaded methods
  - Same name, but different input or output, e.g. public void spinning (Ball b); public void spinning (Image g);

• Overridden methods
  - Redefined in a subclass with the same signature (same input, same output)
Information hiding

```java
public class Guardian {
    private Key prisonKey;
    public Key getPrisonKey(Person requester) {
        if(requester.getType() == Person.PRISON_STAFF)
            return prisonKey;
        else
            return null;
    }
}
```

Encapsulation and information hiding

```java
public class Vehicle {
    private String registration;
    private int mileage = 0;

    public void addJourney(int miles) {
        mileage += miles;
    }

    public int getMileage() {
        return mileage;
    }

    public void setReg(String reg) {
        registration = reg;
    }

    public String getReg() {
        return registration;
    }
}
```

Encapsulation and information hiding

```java
public class UseVehicle {
    public static void main(String[] args) {
        Vehicle car = new Vehicle();
        car.registration = "34 YZ 1234";  // Register car
        car.mileage += 25;              // Drive
        car.mileage = 0;                // Reset mileage !!!
        car.registration = "34 AB 4321"; // Change registration !!!
    }
}
```
Encapsulation and information hiding

public class Vehicle {
    private String registration;
    private int mileage = 0;

    public void addJourney(int miles) {
        if (miles > 0)
            mileage += miles;
    }

    public int getMileage() {
        return mileage;
    }

    public Vehicle(String reg) {
        registration = reg;
    }

    public String getRegistration() {
        return registration;
    }
}

Exception handling

• A mechanism for making programs more robust, i.e. have them continue executing when errors, failures, or exceptional conditions occur, rather than crash.

• When an exception is thrown, execution of the try block is aborted and the corresponding catch handler block is executed.

• A constructor is used to set registration on instance creation.

• addJourney is modified to prevent rewinding.

Exception handling

• If there is no matching catch clause, the exception propagates out of the try block and may be caught by an outside try block. If the exception is never caught, the program’s execution will terminate abnormally.

• You can also throw exceptions yourself using the throw statement, e.g.

    throw new RuntimeException("Invalid stock symbol");

• The finally clause is used to indicate that some statements must be executed even when an exception is thrown, usually to do some cleaning up operations or ensure that an object’s state is consistent.

• The finally block is always executed, whether an exception is thrown or not, and if one is, whether it is caught or not.
try-catch control flow

```java
try {
    statements_try
} catch (C1 th) {
    statements_C1
} catch (C2 th) {
    statements_C2
} catch (C3 th) {
    statements_C3
}
... finally {
    statements_finally
}
```

- In normal control flow, when nothing is thrown in `statements_try`:
  - execute all of `statements_try`
  - then all of `statements_finally`

### Exception handling

- **Classes C1, C2, ...** must be subclasses of `Throwable`.
- The finally block is always executed:
  - after a normal try,
  - after a normal catch,
  - after a throw in a catch handler,
  - after an uncaught throwable.

- All exceptions except instances of `RuntimeException` are checked (by the compiler).

### Exception handling

- When you call a method that may throw a checked exception, your program must say what it will do if the checked exception is thrown, either catch it or declare that it may throw it.

  ```java
  new RunTimeException("too few arguments")
  ```

- To retrieve the message attribute of an exception:
  ```java
  e.getMessage()
  ```

- To print a stack trace:
  ```java
  e.printStackTrace()
  ```