CS2312

Content

Topic1: Java Fundamentals

- Compiling and Running programs:
  - .java -> .class (after **compilation**, .class contains bytecodes, machine language of JVM)
    - Compiling
  - **JRE** runs **.class** with an instance of **JVM**
    - Running
- Errors:
  - Syntax Errors
  - Runtime Errors
  - Logic Errors

Topic2: Java Fundamentals

Topic3: Objects and Classes

Benefit of encapsulation:

- Protect the data from corruption by mistake
- Easier to find the bug
- Easier to change the implementation

Java uses **"call by value"**, the passed value is the reference of the object

field has default initialization, but **local variable** does not, we have to initialize them before usage.

Topic4: Inheritance

Inheritance:

- Reuse field
- Reuse method
- Redefine (Overriding) (Dynamic Binding)

Polymorphism and **Dynamic Binding**

- **Polymorphism**: An object variable can refer to different actual types [Compile time checking], this is checked by compiler
- **Dynamic Binding**: Automatically select the appropriate non-staic method [Run-time decision], this is checked by JVM

```
Employee h = new Manager(); // allowed, as manager is a special type of employee, super type has less data than sub class, so implicit upcast is allowed

Manager m = new Employee(); // not allowed, sub type has more data than the super class, so possible data loss, this is down-casting, implicit downcast is not allowed.
```

final class and method: avoid being inheritated or redefined

Casting: Consider the type of an object as a different type

```
// only if the object is actually a sub class instance, then there won't be run-time error (dynamic binding is checked during run-time)
for (Employee e: allEmployees) {
    Manager m;
    m = (Manager) e; // run-time error! not all the employees are managers
}
// could be solved using instanceOf, but instanceOf is often not needed
```

Abstract method:

- a method with the **abstract** keyword
- no implementation
- acts as a placeholder for concrete (i.e. nonabstract) methods

```
public abstract class Employee {
    public abstract double getPay(); // the implementation is done in its sub classes
}
```

Abstract class:

- cannot be instantiated // no new instance()
- a class with an abstract method must be declared as abstract

Override:

```
// correct way to override a method, e.g. equal, toString

@Override // override must happend after @Override annotation
public boolean equals(Object o) {

}
```

Generic ArrayList

```
ArrayList<Integer> arrlist = new ArrayList<> ();
```

Topic5: Interfaces and Inner class (inner class not tested)

Interface: A way to describe what classes should do (not how)

```
//Syntax
interface Interface_name {
    // static final fields
    // nonstatic methods
    public void f1();
    public void f2();
}

class Employee extends Person implements I1, I2 {
    // detailed implementation
}
```

Interface vs **Abstract class**
Similarities:

- Cannot instantiate them
- Contain methods which are to be implemented by their sub classes

Comparison

Abstract Class	**Interface**
Does not support multiple inheritance	Could implement 1 or more interfaces
Allow modifiers (private/protected/public)	All methods are public
Can provide shared method code	No shared method code
Has constructors	No constructors
Allow various kind of fields	No object fields, any field defined in interface is considered to be static and final

Using **abstract class** when:

- We want to **share code**
- Expect that the sub classes have many **common** methods or fields, or require non-public access modifiers
- Want to declare useful **object fields**

Using **interface** when:

- Unrelated classses would implement (e.g. cloneable and comparable)
- We want **multiple** inheritance

Summary:
Main benefits of **interface**: **multiple** inheritance, **irrelevant** inheritance
Main benefits of **abstract class**: need to **share code**, need **object fields**, **modifiers**

```
// Example for Collections.sort(arrlist);
class Employee implements Comparable<Employee> {
    @Override
    public int compareTo(Employee another) {
        if () return 0;
        else if () return 1;
        else return -1;
    }
}
ArrayList<Employee> arrlist = new ArrayList<>();
Collections.sort(arrlist);
```

Cloneable:

1. Implements **java.lang.Cloneable**
2. Redefine the method **public type clone()**
3. Object class does **shallow-copying**, but if one of the data field is reference (i.e. an object that you created), then you need to make sure that the cloned does not refers to the same object

```
class Employee implements Cloneable /*Comparable<Employee>*/  {
    @Override
    public Employee clone() {
        Employee copy = (Employee) super.clone(); // this does shallow-cloning
        copy.hireDay = new Day(__________); // this does deep-cloning
        return copy;
    }
}
```

Topic6: Exception Handling (Not tested)

Topic7: Generic Programming (Not tested)

Topic8: Collections

Topic9: OOP reviews

Todo:

- Recursive method to solve problem (midterm Q2)
- Some basic OO concepts
    - Benefits of Encapsulation (Topic 3)
    - Overloading concept (Topic 3)
    - Polymorphism and Dynamic Binding (Topic 4)
    - Up-casting and down-casting (Topic 4)
    - Interface vs Abstract Class (Topic 5)
    - Comparable and cloneable (Topic 5)
    - Collections?
- Cloneable and @Override

Reminder:

- Singleton Pattern

```
// Singleton pattern

public class Singleton {
    private static Singleton instance = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return this.instance;
    }
}
```