SWEN20003
Object Oriented Software Development

Revision - Questions

Bach Le
bach.le@unimelb.edu.au

University of Melbourne

Topics Covered So Far

- Subject Introduction
- A Quick Tour of Java (Java Introduction)
- Classes and Objects
- Software Tools and Bagel
- Arrays and Strings
- Input and Output
- Inheritance and Polymorphism
- Interfaces and Polymorphism

Lecture Objectives

- Respond to any mid-semester test related questions
- Review the subject learning outcomes so far

Mid-semester Test

- Will be a 40-minute test worth 30 marks:
  - 10 true/false questions 10 marks
  - 10 multiple choice questions 20 marks.
- The test has been designed to test your knowledge on the contents taught up to end of week 6, as we would test it in a non-online setting (under normal exam conditions), so you are expected to know the content without having to look up subject material or external resources during the exam.

Mid-semester Test

Cont...

- Recognizing the practical constraints in imposing exam conditions in the online context, to be fair to everybody, we have decided to make the exam open-resources (which means you are allowed to look up anything including searching the Internet), but if you do not understand/revise the material and choose to rely on referring to material during the exam, most probably you will run out of time to complete the exam, because the timing is set assuming you do not have to spend time referring to material.
- Even if you refer to material from other sources, you are not allowed do directly copy anything including from subject material – the answers must be your own.
- A practice mid-semester exam will be available.

What is *inheritance*?

What is *inheritance*?

Answer:
The idea behind inheritance is that you can create new classes that are built on existing classes. When you inherit from an existing class, you reuse (or inherit) its methods, and you can add new methods and fields to adapt your new class to new situations. This technique is essential in Java programming.

What is *inheritance*?

Answer:
The idea behind inheritance is that you can create new classes that are built on existing classes. When you inherit from an existing class, you reuse (or inherit) its methods, and you can add new methods and fields to adapt your new class to new situations. This technique is essential in Java programming.

What advantages does it give us as programmers?

What is *inheritance*?

Answer:
The idea behind inheritance is that you can create new classes that are built on existing classes. When you inherit from an existing class, you reuse (or inherit) its methods, and you can add new methods and fields to adapt your new class to new situations. This technique is essential in Java programming.

What advantages does it give us as programmers?

Answer:
This means we don't need to write the same code multiple times, makes our classes simpler, and is often a better representation of important data in a problem.

What *relationship* does inheritance represent?

What *relationship* does inheritance represent?

Answer:
Inheritance represents an *"Is A"* relationship, where it makes sense to say a subclass object *is a* subtype of the superclass; for example, a Dog is a subtype of Animal, a Car is a subtype of Vehicle, but a Table is not a subtype of Chair.

What *relationship* does inheritance represent?

Answer:
Inheritance represents an *"Is A"* relationship, where it makes sense to say a subclass object *is a* subtype of the superclass; for example, a Dog is a subtype of Animal, a Car is a subtype of Vehicle, but a Table is not a subtype of Chair.

What is the `super` keyword? Where do we typically use it?

What *relationship* does inheritance represent?

Answer:
Inheritance represents an *"Is A"* relationship, where it makes sense to say a subclass object *is a* subtype of the superclass; for example, a Dog is a subtype of Animal, a Car is a subtype of Vehicle, but a Table is not a subtype of Chair.

What is the `super` keyword? Where do we typically use it?

Answer:
We use `super` to reference an object's superclass, allowing us to access its protected or public attributes and methods. It can also be used to call the superclass' constructor, which is required if the superclass does not define a default constructor.

What is method overriding?

What is method overriding?

Answer:
Method overriding replaces a superclass' method with new functionality.

What is method overriding?

Answer:
Method overriding replaces a superclass' method with new functionality.

What class does every class inherit from?

What is method overriding?

Answer:
Method overriding replaces a superclass' method with new functionality.

What class does every class inherit from?

Answer: All Java classes implicitly inherit from the `Object` class.

What is method overriding?

Answer:
Method overriding replaces a superclass' method with new functionality.

What class does every class inherit from?

Answer: All Java classes implicitly inherit from the `Object` class.

What are some methods inherited from this class, and why do we generally replace them?

What is method overriding?

Answer:
Method overriding replaces a superclass' method with new functionality.

What class does every class inherit from?

Answer: All Java classes implicitly inherit from the `Object` class.

What are some methods inherited from this class, and why do we generally replace them?

Answer: Classes inherit the `toString` and `equals` methods (among others) from the `Object class`. The `toString` method should give a useful `String` representation of the object, and `equals` should be able to identify when objects are "identical"; neither of the inherited methods work "correctly" (or as expected) by default, so we almost always **override** them.

If you label a class or method as *abstract*, what does it do?

If you label a class or method as *abstract*, what does it do?

Answer: An *abstract class* cannot be instantiated, and an *abstract method* has no implementation.

If you label a class or method as *abstract*, what does it do?

Answer: An *abstract class* cannot be instantiated, and an *abstract method* has no implementation.

What is the conceptual meaning of abstract classes?

If you label a class or method as *abstract*, what does it do?

Answer: An *abstract class* cannot be instantiated, and an *abstract method* has no implementation.

What is the conceptual meaning of abstract classes?

Answer:
An *abstract class* defines common behaviours or properties of other classes, but doesn't have enough *concrete* information for it to be instantiated. For example, we know that all `Animal` objects make noise, but if you are asked what noise an animal makes, there's no "correct" answer; `Animal` is abstract.

How can we decide whether a class should be *abstract* or *concrete*?

How can we decide whether a class should be *abstract* or *concrete*?

Answer:
Deciding whether to make a class abstract is both easy, and hard. Concrete classes represent a *thing* in your problem; a database, a flying robot, a toaster. Abstract classes are... abstract. They are usually created because *concrete* classes share attributes, or behaviour, so we use it as a design tool to make classes neater. Some good rules of thumb:

- Do any of these classes have common attributes or methods?
- If they both inherited from the same class, would that make sense?
- Would it make sense to create an object of that superclass? If not, then it's *abstract*.

Define *polymorphism*.

Define *polymorphism*.

Answer:
Polymorphism in general is the ability of an object or method to be used in many different ways.

Define *polymorphism*.

Answer:
Polymorphism in general is the ability of an object or method to be used in many different ways.

In what ways does Java allow polymorphism?

Define *polymorphism*.

Answer:
Polymorphism in general is the ability of an object or method to be used in many different ways.

In what ways does Java allow polymorphism?

Answer:
Java allows polymorphism through:

- **Overloading** method used depends on the **signature**
- **Overriding** method used depends on the **class** that was instantiated
- **Substitution** subclasses taking the place of superclasses
- **Generics** defining parametrised methods/classes

What is *upcasting*, and why is it useful to be able to write code like:

```
Piece[] pieces = new Piece[]{new Rook(), new King(), new Queen()}
```

What is *upcasting*, and why is it useful to be able to write code like:
`Piece[] pieces = new Piece[]{new Rook(), new King(), new Queen()}`

Answer:

Polymorphism allows us to store *collections* of objects that are similar, by allowing subclass objects to be stored in variables of their superclass. This means we can store (for example) a 2D array of `Piece` objects, and treat them all the same, even though *under the hood* the objects behave differently.
Upcasting is the process of storing an instance of a subclass in a reference of the superclass type. Importantly, if an object has overridden methods, the overridden version is still used when the object is upcasted.

What is *downcasting*? What do you need to be aware of when using it?

What is *downcasting*? What do you need to be aware of when using it?

Answer:
Downcasting refers to the opposite: converting a reference of a superclass type to one of its subclasses. To do this safely, we need to use `instanceof` to make sure the object is of the correct type.

What is an interface?

What is an interface?

Answer:
An interface is used to define the *behaviour* or *actions* of several classes that may not have anything else in common, and represent a "can do" relationship; `Houses` and `Furniture` can both be *built*, but don't have much in common to warrant inheritance.

What is an interface?

Answer:
An interface is used to define the *behaviour* or *actions* of several classes that may not have anything else in common, and represent a "can do" relationship; `Houses` and `Furniture` can both be *built*, but don't have much in common to warrant inheritance.

How is implementing an interface different to inheriting from a class?

What is an interface?

Answer:
An interface is used to define the *behaviour* or *actions* of several classes that may not have anything else in common, and represent a "can do" relationship; `Houses` and `Furniture` can both be *built*, but don't have much in common to warrant inheritance.

How is implementing an interface different to inheriting from a class?

Answer:
Interfaces are the preferred abstraction when there are no (or few) common properties, or no logical parent-child relationship, between classes that share behaviours.

In what situations would we use one (or more) interfaces over inheritance?

In what situations would we use one (or more) interfaces over inheritance?

Answer:
The general rule of thumb for determining if the the relationship between a parent and child class is an inheritance-based or interface-based is that:

- The implementation of an interface indicates a **can-do** relationship.
- The inheriting of a class indicates an **is-a** relationship.

Define an interface called `Transferable` that allows objects to be represented in a network-friendly format (e.g. for sending to other computers).
What are some classes that might use this interface? Why are we using an interface instead of inheritance?

Define an interface called `Transferable` that allows objects to be represented in a network-friendly format (e.g. for sending to other computers).
What are some classes that might use this interface? Why are we using an interface instead of inheritance?

Answer:
A simple Transferable interface could look like:

```
public interface Transferable {
        public String encode();
}
```

Some common objects we may want to transfer include `AudioFile`, `HttpMessage`, `Image`, etc. These have little in common other than the fact they can be encoded in a network-friendly format, so inheritance is not useful.

How do polymorphism and interfaces relate?

How do polymorphism and interfaces relate?

Answer:
Interfaces are only really useful when upcasting is used so that a method does not need specific knowledge of the underlying implementation, and can instead rely upon abstraction. One of the core benefits of interfaces is that you can leverage the polymorphism that an interface enables to make code more maintainable. This is clearly shown in the solution for the design problem.

What is the `Comparable` interface? What implementations of `compareTo` might we use to compare classes:

- `Student`
- `Fruit`
- `MusicalArtist`

What is the `Comparable` interface? What implementations of `compareTo` might we use to compare classes:

- `Student`
- `Fruit`
- `MusicalArtist`

Answer:
The `Comparable` interface allows its implementors to indicate that there is a *natural ordering* to its instances.

An example for Student:

```
public class Student implements Comparable<Student> {
        private final String firstName;
        private final String lastName;
        private final int age;
        public Student(String firstName, String lastName, int age){
                this.firstName = firstName;
                this.lastName = lastName;
                this.age = age;
        }
        @Override
        public int compareTo(Student other) {
                // We leverage the String class' natural ordering
                int comparedFirstName = this.firstName.compareTo(other.firstName);
                // Different first name
                if (comparedFirstName != 0){
                        return comparedFirstName;
                }
                int comparedLastName = this.lastName.compareTo(other.lastName);
                // Different last name
                if (comparedLastName != 0){
                        return comparedLastName;
                }
                return this.age - other.age;
        }
}
```