Lecture 3: Inheritance, Interfaces, and Generics

- Inheritance
- Interfaces
- Generics
- Enums

Some slides adopted from Algorithms, 4th Edition and Oracle tutorials
Inheritance

- When you want to create a new class and there is already a class that includes some of the code that you want, you can derive your new class from the existing class. → reuse code!

- Central concept in OOP.

- A class that is derived from another is called a **subclass** or **child class**.

- The class from which the subclass is derived is called a **superclass** or **parent class**.

- Single inheritance: A class can only extend ONE AND ONLY one parent class.

- Multilevel inheritance: A class can extend a class which extends another class etc.
INHERITANCE

Remember our Bicycle class?

```java
/**
 * Represents a bicycle
 * @author https://docs.oracle.com/javase/tutorial/java/concepts/class.html
 */

public class Bicycle {
    // instance variables
    private int cadence = 0;
    private int speed = 0;
    private int gear = 1;

    // the Bicycle class has one constructor
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    public void changeCadence(int newValue) {
        cadence = newValue;
    }

    public void changeGear(int newValue) {
        gear = newValue;
    }

    public void changeSpeed(int change) {
        speed = speed + change;
    }

    public int getCadence() {
        return cadence;
    }

    public void printGear() {
        System.out.println("Gear:" + gear);
    }

    public String toString() {
        return "cadence:" + cadence + " speed:" + speed + " gear:" + gear;
    }
}
```
A MountainBike is specialized type of Bicycle

/**
 * Demonstrates concept of inheritance
 * @author https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html
 */
public class MountainBike extends Bicycle {

    // the MountainBike subclass adds one field
    private int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int startHeight,
                            int startCadence,
                            int startSpeed,
                            int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }

    // the MountainBike subclass adds one method
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }
}
Inheritance

- The subclass inherits all the public and protected members.
  - Not the private ones, although it can access them with appropriate getters and setters.
- The inherited fields can be used directly, just like any other fields.
- You can declare a field in the subclass with the same name as one in the superclass, thus hiding it.
  - AVOID
- You can write a new instance method in the subclass that has the same signature as the one in the superclass, thus overriding it.
- You can write a new static method in the subclass that has the same signature as the one in the superclass, thus hiding it.
- You can write a subclass constructor that invokes either implicitly the default constructor of the superclass or by directly invoking it using the keyword super().
Polymorphism

- The ability of an object to take many forms.

- **Static Polymorphism**: Happens during method overloading, that is more than one method have the same name but different sets of parameters (signature).
  - Also known as Compile-Time Polymorphism, Static binding, Compile-Time binding, Early binding

- **Dynamic Polymorphism**: Happens during method overriding, that is a method with the same signature exists both in parent and child class. When a parent reference is used to refer to a child object, the method that will be executed with be defined at run-time, therefore will be the child’s overridden method.
  - Student student = new Student();
    Person person = new Student();
  - Also known as Run-Time Polymorphism, Dynamic binding, Run-Time binding, Late binding
Example: Animal

```java
public class Animal {
    public int legs = 2;
    public static String species = "Animal";
    public static void testClassMethod() {
        System.out.println("The static method in Animal");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Animal");
    }
}
```
Example: Cat

```java
public class Cat extends Animal {
    public int legs = 4;
    public static String species = "Cat";
    public static void testClassMethod() {
        System.out.println("The static method in Cat");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Cat");
    }
}
```
Hiding vs overriding

```java
public static void main(String[] args) {
    Cat myCat = new Cat();
    myCat.testClassMethod(); //invoking a hidden method
    myCat.testInstanceMethod(); //invoking an overridden method
    System.out.println(myCat.legs); //accessing a hidden field
    System.out.println(myCat.species); //accessing a hidden field
}
```

Output:

- The static method in Cat
- The instance method in Cat
- 4
- Cat

**WHAT YOU WERE EXPECTING, RIGHT?**
Hiding vs overriding

```java
public static void main(String[] args) {
    Animal yourCat = new Cat();
    yourCat.testClassMethod(); //invoking a hidden method
    yourCat.testInstanceMethod(); //invoking an overridden method
    System.out.println(yourCat.legs); //accessing a hidden field
    System.out.println(yourCat.species); //accessing a hidden field
}
```

Output:

The static method in Animal
The instance method in Cat
2
Animal

???
Hiding vs overriding

- **Hiding**: For fields (instance+static) and methods (static) the class is determined at compile-time. Here, the compiler sees that `yourCat` is declared as `Animal`.

- **Overriding**: For instance methods this is determined at run-time. At this point, we know that `yourCat` is of type `Cat`.

- One form of **polymorphism** (dynamic).

- You will get a compile-time error if you attempt to change an instance method in the superclass to a static method in the subclass and vice-versa.
**super** keyword

- Refers to the direct parent of the subclass.
- **super**.variable: for hidden fields, avoid altogether.
- **super**.instanceMethod(): for overridden methods.
- **super**(args): to call the constructor of the super class. First line in constructor of subclass.
All classes inherit class Object

- Directly if they do not extend any other class, or indirectly as descendants.
- Object class has built-in methods that are inherited.
  - `public boolean equals (Object other)`
    - Default behavior returns true only if same object.
  - `public String toString()`
    - Returns string representation of object - default is hexadecimal.
    - Does not print the string.
    - Typically needs to be overridden to be useful.
  - `public int hashCode()`
    - Unique identifier defined so that if `a.equals(b)` then `a, b` have same `hashCode`. 
**final** keyword

- Variable: only assigned once in its declaration or in constructor – its value cannot be changed after initialization.
  - E.g., `static final PI = 3.14;`
- Method: cannot be overridden by subclass.
- Class: cannot be extended.
INHERITANCE

Practice Time

```java
public class ClassA {
    public void methodOne(int i) {
    }
    public void methodTwo(int i) {
    }
    public static void methodThree(int i) {
    }
    public static void methodFour(int i) {
    }
}

public class ClassB extends ClassA {
    public static void methodOne(int i) {
    }
    public void methodTwo(int i) {
    }
    public void methodThree(int i) {
    }
    public static void methodFour(int i) {
    }
}
```

1. Which method overrides a method in the superclass?
2. Which method hides a method in the superclass?
3. What do the other methods do?
INHERITANCE

Answers

1. methodTwo.

2. methodFour.

3. They cause compile-time errors.
   methodOne: “This static method cannot hide the instance method from ClassA.”
   methodThree: “This instance method cannot override the static method from ClassA.”
Lecture 3: Inheritance, Interfaces, and Generics

- Inheritance
- Interfaces
- Generics
- Enums
Interfaces

- Contracts of what a class must do, not how to do it, abstracting from implementation.
- Central concept in OOP.
- In Java, an interface is a reference type (like a class), that contains only constants, method signatures, default methods, and static methods.
- A class that implements an interface is obliged to implement its methods.
- Method bodies exist only for default methods and static methods.
- Interfaces cannot be instantiated (no `new` keyword). They can only be implemented by classes or extended by other interfaces.
Example

```java
public interface Moveable{
    int turn(Direction direction, double radius, double speed);

    default int stop(){
        speed=0;
    }
}

public class Car implements Moveable{
    int turn(Direction direction, double radius, double speed){
        //code goes here
    }
}

public class Bicycle implements Moveable{
    int turn(Direction direction, double radius, double speed){
        //code goes here
    }
}
```
Interfaces

- A class can implement multiple interfaces.
  ```java
  class A implements Interface1, Interface2{...}
  ```
- An interface can extend multiple interfaces.
  ```java
  public interface GroupedInterface extends Interface1, Interface2{...}
  ```
Lecture 3: Inheritance, Interfaces, and Generics

- Inheritance
- Interfaces
- Generics
Generics

- Compile-time errors can be easier to fix than run-time errors.

- Java introduced generics (similar to templates in C++) to help move more bugs to compile-time (easier to debug!), eliminate casting, and improve abstraction. E.g.,

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

Is now:

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0); // no cast
```

- Generics enable types (that is classes and interfaces) to be used as parameters when defining classes, interfaces, and methods.
Formal and actual type parameters

```java
public interface List<E> {
    void add(E x);
    Iterator<E> iterator();
}
```

```java
public interface Iterator<E> {
    E next();
    boolean hasNext();
}
```

> In the invocation (e.g., `List<Integer>`) all occurrences of the formal type parameters are replaced by the **actual type argument** (e.g., `Integer`).
Generic classes

class name <T1, T2, ..., Tn> {…}

- A type variable can be any non-primitive type (class, interface, array)
- E: element (common in data structures), T: type, K: key, V: value, N: number, etc.

/**
* Generic version of the Box class.
* [https://docs.oracle.com/javase/tutorial/java/generics/types.html](https://docs.oracle.com/javase/tutorial/java/generics/types.html)
* @param <T> the type of the value being boxed
*/

public class Box<T> {
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}

- Invocation: Box<Integer> integerBox = new Box<Integer>();
Multiple Type Parameters Example

```java
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}

public class OrderedPair<K, V> implements Pair<K, V> {
    private K key;
    private V value;

    public OrderedPair(K key, V value) {
        this.key = key;
        this.value = value;
    }
}
```

```java
Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);
OrderedPair<String, Box<Integer>> p = new OrderedPair<String, Box<Integer>>("primes", new Box<Integer>(...));
```
Generic methods

modifier (static) <T1, T2, ..., Tn> return-type name(list of type parameters){...}{}

- The type parameter’s scope is limited to the method which is declared.
- Static, non-static generic methods, generic class constructors are allowed.
- **Type inference**: allows you to invoke a generic method as an ordinary method, without specifying a type between angle brackets.
- E.g., className/objectName.genericMethod(arguments);
Example

- Generic method that swaps the elements of an array at two specified indices.

```java
public static <T> void swap(T[] a, int i, int j) {
    T temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```
Lecture 3: Inheritance, Interfaces, and Generics

- Inheritance
- Interfaces
- Generics
- Enums
The Enum type

- Special data type that enables for a variable to be a set of predefined constants.
- The variable needs to be equal to one of the predefined values of the set.
- You should use enum types any time you need to represent a fixed set of constants, e.g., days, months, compass directions, etc.
- Enum’s type fields are in ALL CAPS.

```java
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
```

- Then use as an object anywhere.
  - e.g., Day.values();
Readings:

- Oracle’s guides:
  - Interfaces and Inheritance: https://docs.oracle.com/javase/tutorial/java/landl/index.html
  - Generics: https://docs.oracle.com/javase/tutorial/java/generics/index.html
    https://docs.oracle.com/javase/tutorial/extra/generics/intro.html

- Textbook:
  - Pages 100-104, 122

- Textbook Website:
  - Generics: https://algs4.cs.princeton.edu/13stacks/

Practice Problems:

- If you want more practice with hiding vs overriding:
  http://javabypatel.blogspot.com/2016/04/java-interview-questions.html