3: Inheritance, Interfaces, and Generics

Tom Yeh
he/him/his
Lecture 3: Inheritance, Interfaces, and Generics

- Finish Java Basics
- Inheritance
- Interfaces
- Generics

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

- 1st quiz this Thursday
- 1st programming assignment to be released today
Practice Time:

1. The term "instance variable" is another name for ___.

2. The term "class variable" is another name for ___.

3. A local variable stores temporary state; it is declared inside a ___.

4. A variable declared within the opening and closing parentheses of a method signature is called a _____. The actual value passed is called an ___.

5. What are the eight primitive data types supported by the Java programming language?

6. Character strings are represented by the class ___.

7. An ___ is a container object that holds a fixed number of values of a single type.
Answers:

1. The term "instance variable" is another name for **non-static/member field**.

2. The term "class variable" is another name for **static field**.

3. A local variable stores temporary state; it is declared inside a **method**.

4. A variable declared within the opening and closing parentheses of a method is called a **parameter**. The actual value passed is called an argument.

5. What are the eight primitive data types supported by the Java programming language? **byte, short, int, long, float, double, boolean, char**

6. Character strings are represented by the class **java.lang.String**.

7. An **array** is a container object that holds a fixed number of values of a single type.

https://docs.oracle.com/javase/tutorial/java/nutsandbolts/QandE/answers_variables.html
Lecture3: Finishing Java Basics

- Methods
- Arrays
- Operators
- Control Flow
## Operator precedence

<table>
<thead>
<tr>
<th>Operators</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>postfix</td>
<td><code>expr++  expr--</code></td>
</tr>
<tr>
<td>unary</td>
<td><code>/++/+expr  -/----expr  !boolean</code></td>
</tr>
<tr>
<td>multiplicative</td>
<td><code>*  /  %</code></td>
</tr>
<tr>
<td>additive</td>
<td><code>+  -</code></td>
</tr>
<tr>
<td>relational</td>
<td><code>&lt;  &gt;  &lt;=  &gt;=  instanceof</code></td>
</tr>
<tr>
<td>equality</td>
<td><code>==  !=</code></td>
</tr>
<tr>
<td>logical AND</td>
<td><code>&amp;&amp;</code></td>
</tr>
<tr>
<td>logical OR</td>
<td>`</td>
</tr>
<tr>
<td>assignment</td>
<td><code>=  +=  -=  *=  /=</code></td>
</tr>
</tbody>
</table>
Assignment operator

- = assigns the value on its right to the operand on its left
  - e.g., `int cadence = 3;`
Arithmetic operators

```java
/**
 * Illustration of the arithmetic operators
 * @author https://docs.oracle.com/javase/tutorial/java/nutsandbolts/op1.html
 */
public class ArithmeticDemo {

    public static void main(String[] args) {

        int result = 1 + 2;
        // result is now 3
        System.out.println("1 + 2 = "+ result);
        int original_result = result;

        result = result - 1;
        // result is now 2
        System.out.println(original_result + " - 1 = "+ result);
        original_result = result;

        result = result * 2;
        // result is now 4
        System.out.println(original_result + " * 2 = "+ result);
        original_result = result;

        result = result / 2;
        // result is now 2
        System.out.println(original_result + " / 2 = "+ result);
        original_result = result;

        result = result + 8;
        // result is now 10
        System.out.println(original_result + " + 8 = "+ result);
        original_result = result;

        result = result % 7;
        // result is now 3
        System.out.println(original_result + " % 7 = "+ result);
    }
}
```

Output:

```
1 + 2 = 3
3 - 1 = 2
2 * 2 = 4
4 / 2 = 2
2 + 8 = 10
10 % 7 = 3
```
Unary operators require only one operand

```java
/**
 * Illustration of the unary operators
 * @author https://docs.oracle.com/javase/tutorial/java/nutsandbolts/op1.html
 */
public class UnaryDemo {
    public static void main(String[] args) {
        int result = +1;
        // result is now 1
        System.out.println(result);

        result--;
        // result is now 0
        System.out.println(result);

        result++;
        // result is now 1
        System.out.println(result);

        result = -result;
        // result is now -1
        System.out.println(result);

        boolean success = false;
        // false
        System.out.println(success);
        // true
        System.out.println(!success);
    }
}
```
The `++/--` operators can be applied pre or post operand

/**
 * Illustration of the prefix/postfix unary operator
 * @author https://docs.oracle.com/javase/tutorial/java/nutsandbolts/op1.html
 */

public class PrePostDemo {
    public static void main(String[] args){
        int i = 3;
        i++;
        // prints 4
        System.out.println(i);
        ++i;
        // prints 5
        System.out.println(i);
        // prints 6
        System.out.println(++i);
        // prints 6
        System.out.println(i++);
        // prints 7
        System.out.println(i);
    }
}
Equality/Relational operators

/**
 * Illustration of the equality/relational operators
 * @author https://docs.oracle.com/javase/tutorial/java/nutsandbolts/op1.html
 */

public class ComparisonDemo {

    public static void main(String[] args) {
        int value1 = 1;
        int value2 = 2;
        if (value1 == value2) {
            System.out.println("value1 == value2");
        }
        if (value1 != value2) {
            System.out.println("value1 != value2");
        }
        if (value1 > value2) {
            System.out.println("value1 > value2");
        }
        if (value1 < value2) {
            System.out.println("value1 < value2");
        }
        if (value1 <= value2) {
            System.out.println("value1 <= value2");
        }
    }
}
Conditional operators

/**
 * Illustration of the equality/relational operators
 * @author https://docs.oracle.com/javase/tutorial/java/nutsandbolts/op1.html
 */

public class ConditionalDemo {

    public static void main(String[] args) {
        int value1 = 1;
        int value2 = 2;
        if ((value1 == 1) && (value2 == 2))
            System.out.println("value1 is 1 AND value2 is 2");
        if ((value1 == 1) || (value2 == 1))
            System.out.println("value1 is 1 OR value2 is 1");
    }
}

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A AND B</th>
<th>A OR B</th>
<th>NOT A</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>
Practice Time

1. Consider the following code:
   ```java
   arrayOfInts[j] > arrayOfInts[j+1]
   ```
   Which operators does the code contain?

2. Consider the following code snippet:
   ```java
   int i = 10;
   int n = i++%5;
   ```
   a. What are the values of `i` and `n` after the code is executed?
   b. What are the final values of `i` and `n` if instead of using the postfix increment operator (`i++`), you use the prefix version (`++i`))?

3. To invert the value of a boolean, which operator would you use?

4. Which operator is used to compare two values, `=` or `==`?
Answers:

1. >, +
2.
   a. i is 11, and n is 0
   b. i is 11, and n is 1.
3. The logical complement operator !
4. ==

https://docs.oracle.com/javase/tutorial/java/nutsandbolts/QandE/answers_operators.html
Lecture 3: Java Basics

- Control Flow
If-then statement

```java
public void applyBrakes() {
    // the "if" clause: bicycle must be moving
    if (isMoving) {
        // condition MUST be inside parens
        // the "then" clause: decrease current speed
        currentSpeed--;
    }
}
```
CONTROL FLOW

If-then-else statement

```java
/**
 * Illustration of the if then else control flow
 * @author https://docs.oracle.com/javase/tutorial/java/nutsandbolts/if.html
 */

public class IfElseDemo {
    public static void main(String[] args) {

        int testscore = 76;
        char grade;

        if (testscore >= 90) {
            grade = 'A';
        } else if (testscore >= 80) {
            grade = 'B';
        } else if (testscore >= 70) {
            grade = 'C';
        } else if (testscore >= 60) {
            grade = 'D';
        } else {
            grade = 'F';
        }
        System.out.println("Grade = " + grade);
    }
}
```

Once a condition is satisfied, the appropriate statements are executed and the remaining conditions are not evaluated.
While statement

/**
 * Illustration of the if then else control flow
 * @author https://docs.oracle.com/javase/tutorial/java/nutsandbolts/while.html
 */

public class WhileDemo {

    public static void main(String[] args){
        int count = 1;
        while (count < 11) {
            System.out.println("Count is: " + count);
            count++;
        }
    }
}
For statement

```java
for (initialization; termination; increment) {
    statement(s)
}

/**
 * Illustration of the for loop
 * @author https://docs.oracle.com/javase/tutorial/java/nutsandbolts/for.html
 */

public class ForDemo {
    public static void main(String[] args) {
        for (int i = 1; i < 11; i++) {
            System.out.println("Count is: " + i);
        }
    }
}
```
Enhanced for statement in most data structures

/**
 * Illustration of the enhanced for flow
 * @author https://docs.oracle.com/javase/tutorial/java/nutsandbolts/for.html
 * 
 */
class EnhancedFor {
    public static void main(String[] args){
        int[] numbers = {1,2,3,4,5,6,7,8,9,10};
        for (int item : numbers) {
            System.out.println("Count is: "+ item);
        }
    }
}
Break statement

- Use `break` to terminate a `for` or `while` loop.

```java
/**
 * Illustration of the break branch
 *
 * @author https://docs.oracle.com/javase/tutorial/java/nutsandbolts/branch.html
 *
 */
public class BreakDemo {
    public static void main(String[] args) {
        int[] arrayOfInts = {32, 87, 3, 589, 12, 1076, 2000, 8, 622, 127};
        int searchfor = 12;

        int i;
        boolean foundIt = false;

        for (i = 0; i < arrayOfInts.length; i++) {
            if (arrayOfInts[i] == searchfor) {
                foundIt = true;
                break;
            }
        }

        if (foundIt) {
            System.out.println("Found " + searchfor + " at index " + i);
        } else {
            System.out.println(searchfor + " not in the array");
        }
    }
}
CONTROL FLOW

Continue statement

Use **continue** to skip the current iteration of **for** or **while** loop.

```java
public class ContinueDemo {
    public static void main(String[] args) {
        String searchMe = "peter piper picked a " + "peck of pickled peppers";
        int max = searchMe.length();
        int numPs = 0;

        for (int i = 0; i < max; i++) {
            // interested only in p's
            if (searchMe.charAt(i) != 'p')
                continue; // What happens if we used a break here?

            // process p's
            numPs++;
        }
        System.out.println("Found " + numPs + " p's in the string.");
    }
}
```
Return statement

- The `return` statement exits from the current method, and control flow returns to where the method was invoked.
- Can return a value, e.g., `return counter++;`
- Or not, e.g., `return;`
Lecture 2: Java Basics

- Methods
- Arrays
- Operators
- Control Flow
Readings:

- Oracle’s guides:
  - Language Basics: [https://docs.oracle.com/javase/tutorial/java/nutsandbolts/index.html](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/index.html)

- Textbook:
  - Chapter 1.1 (Pages 8–35)
  - Chapter 1.2 (Pages 64–77, 84–88, 96–99)

Practice Problems:

- 1.1.1–1.1.5, 1.1.8–1.1.12, 1.2.4, 1.2.8
Lecture 3: Inheritance, Interfaces, and Generics

- Inheritance
- Interfaces
- Generics

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;
    }
}
```
INHERITANCE

A MountainBike is a specialized type of Bicycle

```java
/**
 * Demonstrates concept of inheritance
 * @author https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html
 */

public class MountainBike extends Bicycle {
    // MountainBike is subclass, Bicycle is superclass

    // 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);    // Calls constructor for superclass
        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()**.
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.
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

https://medium.com/@shanikae/polymorphism-explained-simply-7294c8deef7
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

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.
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.
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) {
    }
}

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
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.
  - `class A implements Interface1, Interface2 {...}

- An interface can extend multiple interfaces.
  - `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.,

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

Is now:

```java
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 \(<\text{T}_1, \text{T}_2, \ldots, \text{T}_n> \{\ldots}\)

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

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

public class Box<\text{T}> {
    private \text{T} t;

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

- Invocation: Box<\text{Integer}> \text{integerBox} = \text{new} \text{Box<\text{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;
    }
}

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;
}
```
Readings:

- Oracle’s guides:
  - Interfaces and Inheritance: [https://docs.oracle.com/javase/tutorial/java/landl/index.html](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/java/generics/index.html)  
    [https://docs.oracle.com/javase/tutorial/extra/generics/intro.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/](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](http://javabypatel.blogspot.com/2016/04/java-interview-questions.html)