Java SL Review LIJIA XU (17)

* every line of code that can actually run needs to be inside a class. every Java program starts from the main method

```
class MyClass{
  public static void main(String[] args){
    System.out.println("Hi");
  }
}
```

// comments
/* */

```
String name="A";
int test=5;
++test; // test=6
int x=34;
int y=++x;
// y=35
int x=34;
int y=x++;
// y=34
```

use + for String conca

```
int
double
String
char
boolean
```

```
import java.util.Scanner;
class MyClass{
  public static void main(String[] args){
    Scanner myVar = new Scanner(System.in);
    System.out.println(myVar.nextLine());
  }
}
```

```
&& || !
for(int x=1; x<=5; x++){ }
use array literal to instantiate
String[] names = {"A","B","C"}; // most often
*String names[] // also okay
```

int[] arr = new int[5]; // declares an array of 5 integers (index up to 4)

Scanner obj is the most often

```
import java.util.Scanner
Scanner myVar = new Scanner(System.in);
```

```
nextByte()
nextShort()
nextInt()
nextLong()
nextFloat()
nextDouble()
nextBoolean()
nextLine()
next() // read word
```

```
> < != == <= >=
switch(){
  case val1:
    // break;
  default:
    //
}
```

```
for(int x=1; x<=5; x++){
  System.out.println(x);
}
int[] primes={2,3,5};
for(int t: primes){
  System.out.println(t);
}
```

multi dimensional Arrays ↘

can have more dimen

```
int[][] sample = {{1,2,3},{4,5,6}};
int x = sample[1][0] // 4
```

OOP

identities, attributes, behavior

object's current state capable of

Methods define behavior

collection of statements grouped together to perform operation

parameters

* define the return type before method name

void → method does not return any

```
static int sum(int val1, int val2){
  return val1 + val2;
}
```

class attribute Variables within a class

to use the method declare an object of Animal

```
public class Animal{
  void bark(){
    System.out.println("Woof");
  }
}
```

```
Animal dog = new Animal();
dog.bark();
```

Access Modifiers

* Best to keep vars in a class private
* modify using Getters and setters

default → any other class in same package
public → any other class
protected → same as default + subclasses can access superclass
private → Accessible only within in class

* best validate the value in the setter method

```
public class Vehicle{
  private String color;
  public String getColor(){
    return color;
  }
  public void setColor(String c){
    this.color = c;
  }
}
```

constructors: name must be same as the class name

No explicit return type

* think constructors as methods will set up class by default

multi

A single class can have constructors with different number of parameters

boolean char

value types: byte, short, int, long, float, double,

```
public class Vehicle{
  private String color;
  Vhicle(){
    this.setColor("red")
  }
  Vehicle(String c){
    this.setColor(c);
  }
  public void setColor(String c){
    this.color = c;
  }
}
```

```
Vehicle v1 = new Vehicle();
Vehicle v2 = new Vehicle("A");
```

Java auto provides a default constructor for all classes

Arrays and Strings are reference types

Math Class

```
int a = Math.pow(2,3); // 8
.abs();
.ceil();
//double
.floor();
.max();
.min();
.pow();
```

only one static member exists and will be shared by all objects

```
Counter c1 = new Counter();
Counter.COUNT // 2
c1.COUNT // 2
```

* best to use uppercase for static variables

the main method must always be static

final → var to constant
class can not subclassed
methods can not overridden

public static final double PI = 3.14;

Packages → avoid name conflicts and to control access to classes

on top of files: package sample;

import classes from packages: import samples.Vehicle;

4 core in OOP:

encapsulation → data hiding: var as private use setter and getter for values
control way data accessed or modified
more flexible and easily change
change one part and not influence other

inheritance → one class to acquire the properties of another

polymorphism

abstraction

inherits all non-private vars and Methods

constructor of the superclass is called

```
class A{
  public A(){
    System.out.println("New A");
  }
}
class B extends A{
  public B(){
    System.out.println("New B");
  }
}
class Program{
  public static void main(String[] args){
    B obj = new B();
  }
}
// new A
// new B
```

Polymorphism: call to a member method will cause a different implementation to be executed

depend on the type of method invoking the method

Method Overriding: have same return type and arguments * access level can Not be more restricted in the subclass

method declared as final or static can not be overridden *

constructors can Not be overriden

Overloading → Methods have the same name, different parameters

same method functionality for different types of parameters

* parameters should differ in type, number, or both

Method Overloading ⇓ compile-time polymorphism

Data abstraction → representing essential features without including implementation details

abstract class → Any class contains an abstract method should be define as abstract

interfaces

```
abstract class Animal{
  int legs = 0;
  abstract void makeSound();
}
```

Interface is a completely abstract class that contains only abstract methods

Define using interface keyword

May only have static final vars

can not contain constructor

can extend other interfaces

A class can implement any number of interfaces

interface and the methods init are implicitly abstract, Not need the abstract keyword

Methods in interface → implicitly public

```
interface Animal{
  public void eat();
  public void makeSound();
}
class Cat implements Animal{
  public void makeSound(){
    System.out.println("A");
  }
  public void eat(){
    System.out.println("B");
  }
}
```

Type Casting → assign one var of one type to another type

```
double a = 42.582;
int b = (int) a;
```

Up casting: Auto

```
Animal a = new Cat();
```

* never fail

Down casting

```
Animal a = new Animal();
((Cat)a).makeSound();
```

* may fail

Anonymous classes → extend the existing classes on the fly

only to the current object

```
class Machine{
  public void start(){
    System.out.println("A");
  }
}
public static void main(String[] args){
  Machine m = new Machine(){
    @Override public void start(){
      System.out.println("B");
    }
  }; // notice the ;
}
```

Java supports nesting classes

equals() for semantical testing

Enum → used to define collection of constants

* values are comma-seperated

```
enum Rank{
  SOLDIER, SERGEANT
}
Rank a = Rank.SOLDIER;
```

can check with a switch statement

API

```
import java.util.*
```

Exception

user error
programmer error
physical resource issue

Java is a multi-threaded programming language

Threads → notes 2

Application Program Interface provides special classes to store and manipulate groups of objects

ArrayList

```
import java.util.ArrayList
ArrayList Colors = new ArrayList();
Optional:
ArrayList<Integer> colors = new ArrayList<Integer>(10);
```

```
.contains()
.get(int index)
.size()
.clear()
index starts 0
.containsKey();
.containsValue();
```

```
try{
  int[] a = new int[2];
  System.out.println(a[5]);
}catch(Exception e){
  System.out.println("A");
}
```

throw ⇓ manually generate exceptions

```
int div(int a, int b) throws ArithmeticException{
  if(b==0){
    throw new ArithmeticException("Division by zero");
  }else{
    return a/b;
  }
}
```

Exception From specific to General

```
try{
  // code
}catch(ExceptionType e1){
  // catch 1
}catch(Exception e){
  // catch all others
}
```

can Not specify initial capacity for LinkedList

* ArrayList for storing and accessing
* LinkedList for multi inserts and deletes

HashMap → store data collection in key/value pairs

use .get(key) → value

```
import java.util.HashMap;
HashMap<String, Integer> points = new HashMap<String, Integer>();
```

It return null ⇐ the absence of a value