# OOP: instance

## class and object

### objects

class is blueprint of objects. A class could be declared with some variables known as attributes/properties and methods. An object is instantiation of a class.

instantiation is instance of a class. A object would be created using new literal.

|  |
| --- |
| simplest class |
| class Test{ } // declare a class |
| Test t = new Test(); //instantiate class |

|  |
| --- |
| example class: properties, constructor and instance methods |
| class Test{  int age; //class properties  Test(int age){ // constructor with arguments  self.age = age;  }  void print(){ // instance methods  System.out.println(this.age);  }  } |
| int age =10;  Test t = new Test(age);  t.print(); |

|  |
| --- |
| certain class may consist of initializer block, which would be run before constructor. |
| public class ClassTest {  int age;  **{ //initializer block**  **age =10;**  **}**  static String name;  **static{ //initializer block**  **name = "na";**  **}**  public static void main(String[] args){  //wrong: System.out.println(ClassTest.age);  System.out.println(ClassTest.name);  ClassTest t = new ClassTest();  System.out.println(t.age);  System.out.println(t.name);  }  } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### garbage collection

objects without reference variables connected are eligible for garbage collection in JVM.

Garbage collection may not surely be executed.

|  |
| --- |
| two objects are created. |
| class A{  int a;  A(int a){ this.a = a; }  public static void main(String[] args){  A a = new A(1);  A b = new A(1);  }  } |
|  |

|  |
| --- |
|  |
| public class ReturnDemo {  public static String getName() {  String tmp = new String("unknown");  return tmp;  }  public static void main(String [] args) {  String result;  result = getName();  System.out.println(result);  **result = null; // the string object is ready for GC after this line.**  System.gc();  }  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
| one object is created. The references are all set to null by default |
| String[] s= new String[5]; |

|  |
| --- |
| three objects are created. object(2018,8,15) is eligible for garbage collection at Line X. d1 and d2 refer to object(2018,8,16), d3 refer to object(2018,8,17) |
| import java.time.\*;  public class Garbage1 {  public static void main(String[] args) {  LocalDate d1 = LocalDate.of(2018,8,15);  System.out.println(d1.hashCode());  d1 = d1.plusDays(1);    System.out.println(d1.hashCode());  LocalDate d2 = d1;  System.out.println(d2.hashCode());  LocalDate d3 = d1.plusDays(1);  System.out.println(d3.hashCode());  //Line x  }  } |
|  |

|  |
| --- |
| three objects are created. Three are eligible for garbage collection at lineX |
| public class Garbage2 {  public static void main(String[] args) {  new Wind(1).go();  //lineX  }  }  class Wind{  int id;  Wind(int i){  id=i;  }  void go(){  Wind w1=new Wind(1);  Wind w2=new Wind(1);  System.out.println(w1.hashCode());  System.out.println(w2.hashCode());  }  } |

|  |
| --- |
| Two objects are created. Two are eligible for garbage collection at lineX. |
| public class Garbage3 {  public static void main(String[] args) {  Wind w= new Wind(1);  System.out.println(w.hashCode());  Wind w2 = w.go(w);  System.out.println(w2.hashCode());  //lineX  }  }  class Wind{  int id;  Wind(int i){  id=i;  }  Wind go(Wind w){  w = new Wind(1);  Wind w1 = w;  w.id=2;  System.out.println(w.hashCode());  System.out.println(w1.hashCode());  return w1;  }  } |

|  |
| --- |
| 5 objects of Test (da refers to 3+1, d refers 1) are created. two objects that references da[0] and da[1]/d refer to are ready for garbage collection at lineX. |
| public class Garbage4 {  public static void main(String[] args) {  **Test[] da = new Test[3];**  da[0] = new Test();  Test d = new Test();  da[1] = d;  d = null;  da[1] = null;  // LineX;  }  }  class Test{} |
|  |

The finalize method can be invoked at most once.

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

## constructor

### declaration

Constructor is a special method, of which name must be identical with that of virtual class and has not return type. Constructor could initialize or update class/instant variables.

YES assertion:

* Constructor must have the same name with class name. Thus, constructor can't be overridden like common class methods.
* Constructor is automatically invoked when a class is instantiated. It can’t be executed separately like instance methods.
* Declaration of constructor is optional. A default constructor is automatically provided if none constructor is declared . For example, no arguments are passed or don't need to initialize class variables.
* It is ok to declare multiple constructors in one class. That is overloading. Overloading constructors must have the same name as class name and different arguments in type or number.
* For inheritance, constructor of parent class is executed firstly and then child's ones.

|  |
| --- |
| The three definitions are equal. constructor is optional. Default constructor would be invoked automatically |
| class A1{ }  //default constructor  class A2{ A2(){} }  //All objects belong to Object. super() refers to class java.lang.Object.  class A3{  A3(){ super(); }  } |

NO assertions:

* Don’t define return type in constructor . if there is, that should be method rather than constructor even if it method may have the same name as class name.
* Don't declare constructor in abstract class or static class or interface because they can't be instantiated as virtual class.
* Constructor is never overridden like instance methods because name of class name and constructor are identical. If there is inheritance, always execute super() firstly even if super() may be omitted.
* Don't forget super() if all constructors are arguments required in superclass.
* Don't place this() or super() in anywhere in constructor or other methods except the first statement of constructor.
* Don't use final modifier.
* Don't repeat declaration.

In some tricky source code, watch out to distinguish constructor with instance methods when method name is identical with class name. Of course, such approach for naming methods is not suggested.

|  |
| --- |
| Distinguish methods and constructors. The two classes below can be compiled. |
| class B1{  **B1(){ // this is constructor**  System.out.println("constructor B1");  }  **void B1(){ // this is method**  System.out.println("B1");  }  } |
| class B2{  **void B2(){ // this is method**  System.out.println("B2");  return;  }  } |
| class B3{  **B3(){ // this is constructor though there is "return;"**  System.out.println("constructor B3");  **return; //compiling is ok but that is not suggested.**  }  } |

|  |
| --- |
| output: 0 |
| public class Salmon {  int count;  **public void Salmon() { // this method**  count = 4;  }  public static void main(String[] args) {  Salmon s = new Salmon();  System.out.println(s.count);  }  } |

|  |
| --- |
| output: Color:null |
| public class Egret {  private String color;  public void Egret() { // this is method  Egret("white");  }  public void Egret(String color) { // that is method  color = color;  }  public static void main(String[] args) {  Egret e = new Egret();  System.out.println("Color:" + e.color);  }  } |

### this and super keywords

* "this." denotes this class variable. "super." denotes parent class variable. Class variables can't be overridden but hidden. For variables with same name, there are two different variable in sub-/super-class.
* this() and super() denote constructor of this class and its parent class, respectively. this() and super() **must be place in constructor, and the first statement**. They are used for calling a certain constructor of this class or any one of constructors of superclass.

Note: Java doesn't allow multiple inheritance, for example,

|  |
| --- |
| case 1 is wrong. In case 2, parent of A is B, C is parent of B rather than A. Therefore, super. or super() invoked in A would always refer to something in B. |
| //case 1  class C{}  class B{}  class A extends B, C{} |
| //case 2  class C {}  class B extends C{}  class A extends B{} |

|  |  |
| --- | --- |
| output |  |
| 10 | public class Test {  int age;  Test(){ **this.age = 10;** }  public static void main(String[] args) {  Test t = new Test();  System.out.println(t.age);  }  } |
| 10  this.age is equal to age if not other identical name variables | public class Test {  int age;  Test(){  **age = 10;**  }  public static void main(String[] args) {  Test t = new Test();  System.out.println(t.age);  }  } |
| 0  Test() is not constructor but methods | public class Test {  int age;  **void Test(){** this.age = 10; }  public static void main(String[] args) {  Test t = new Test();  System.out.println(t.age);  }  } |
| 0  the two age are all local variable.  correct:  Test(int age){  this.age = age;  } | public class Test {  int age;  **Test(int age){ age = age; }**  public static void main(String[] args) {  Test t = new Test(10);  System.out.println(t.age);  }  } |
| 1  class variable age is not updated in constructor but in initializer block. | public class Test {  int age;  Test(int age){ **age = this.age;** }  public static void main(String[] args) {  Test t = new Test(10);  System.out.println(t.age);  }  **{ age = 1; }**  } |

|  |  |
| --- | --- |
| output |  |
| 1  10  this() will be one of argument constructor | public class Test {  int age;  **Test(){ this(1); }**  Test(int age){ this.age = age; }  public static void main(String[] args) {  Test t1 = new Test();  Test t2 = new Test(10);  System.out.println(t1.age);  System.out.println(t2.age);  }  } |
| 1  null  10 | public class Test {  int age;  Integer a;  Test(){ this(1); }  Test(int age){ this.age = age; }  Test(Integer a){ this.a = a; }  public static void main(String[] args) {  Test t1 = new Test();  Test t2 = new Test(10);  System.out.println(t1.age);  System.out.println(t1.a);  System.out.println(t2.age);  }  } |
| 1  10 | public class Test extends Animal{  Test(){ super(1); }  Test(int age){ super(age); }  public static void main(String[] args) {  Test t1 = new Test();  Test t2 = new Test(10);  System.out.println(t1.age);  System.out.println(t2.age);  }  }  class Animal{  int age;  Animal(int age){ this.age = age; }  } |
| output:  33  1111  Two same instance variables in superclass.  this.age and super.age refer to the same value that refer to Tiger.age. I**nstance variable age in Animal is hidden.**  Java prohibit multiple inheritance. that means a subclass has only one parent, and one grandparent, and so on. | public class Test extends Tiger{  Test(){  super(2);  System.out.print(this.age);  **System.out.println(super.age);**  }  Test(int age){  super(age);  System.out.print(this.age);  **System.out.println(super.age);**  }  public static void main(String[] args) {  Test t1 = new Test();  Test t2 = new Test(10);  }  }  class Animal{  int age;  Animal(){ this.age = 1; }  Animal(int age){ **this.age = age;** }  }  class Tiger extends Animal{  int age;  Tiger(int age){ **this.age = age + 1;** }  } |
| output:  313  919 | public class Test extends Tiger{  Test(){ super(2); }  Test(int age){ super(age); }  public static void main(String[] args) {  Test t1 = new Test();  **System.out.print(t1.age);**  **System.out.print(((Animal) t1).age);**  **System.out.println(((Tiger) t1).age);**  Test t2 = new Test(8);  System.out.print(t2.age);  System.out.print(((Animal) t2).age);  System.out.println(((Tiger) t2).age);  }  }  class Animal{  int age;  Animal(){ this.age = 1; }  Animal(int age){ this.age = age; }  }  class Tiger extends Animal{  int age;  Tiger(int age){ this.age = age + 1; }  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### access privilege

Default access of constructor is "package default". So the same package class could instantiate this class.

|  |
| --- |
| default access |
| public class Test{  int age;  Test(){  this.age=1;  }  Test(int age){  this.age=age;  }  public static void main(String[] args) {  Test t1 = new Test();  Test t2 = new Test(8);  }  } |

class with private constructor must be instantiated within this class.

|  |
| --- |
| If constructor is marked as private, this class is only instantiated within this class. |
| public class Test{  int age;  **private Test(){**  this.age=1;  }  **private Test(int age){**  this.age=age;  }  public static void main(String[] args) {  Test t1 = new Test();  Test t2 = new Test(8);  }  } |

|  |
| --- |
|  |
| public class ConstructorTest{  int age;  **private ConstructorTest(int age){**  this.age = age;  }  public static void main(String[] args){  ConstructorTest c = new ConstructorTest(10);  System.out.println(c.age);  }  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### order of execution

Parent constructor and sub constructor would be called in order.

|  |
| --- |
| output:  animal mammal tiger |
| public class InitialOrder1 extends Tiger{  public static void main(String[] args){  InitialOrder1 d = new InitialOrder1();  }  }  class Tiger extends Mammal{  Tiger(){  System.out.print("tiger ");  }  }  class Mammal extends Animal{  Mammal(){  System.out.print("mammal ");  }  }  class Animal{  Animal(){  System.out.print("animal ");  }  } |

|  |
| --- |
|  |
|  |

## instance method

Class methods are categorized by virtual methods and other methods. The other methods could be final methods, static methods, private methods.

YES assertions

* Instance methods belongs to instances of a class (This is object). Instance methods could be invoked only if an object is created.
* Instance methods could access instance/class variables or methods. use this. or super. to invoke instance method or access instance variables within class.
* Instance methods may have multiple copies depending on objects. Instance methods could be overloaded or overridden.

NO assertions.

* Don't invoke instance methods in static context.

### define and invoke

|  |
| --- |
| define class methods and call them. |
| public class ClassMethod {  int age;  ClassMethod(){  age=10;  }  void setAge(int age){  this.age=age;  }  int getAge(){  return this.age;  }  public static void main(String[] args){  ClassMethod c = new ClassMethod();  c.setAge(20);  System.out.println(c.getAge());  }  } |

|  |
| --- |
| employ instance methods within instance methods  There is only one calAgeRate(). So "float age\_rate = this.calAgeRate();" is equal to " float age\_rate = calAgeRate();" |
| public class ClassMethod2 {  int age = 10;  float calAgeRate(){  return (float) this.age \* 10;  }  float getTotal(){  **float age\_rate = calAgeRate();**  return age\_rate \* .1f;  }  public static void main(String[] args){  ClassMethod2 c = new ClassMethod2();  System.out.println(c.getTotal());  }  } |

|  |
| --- |
| define instance methods, and invoke another instance methods. |
| public class Method2 {  int age = 10;  float calAgeRate(){  return (float) this.age \* 10;  }  float getTotal(){  **float age\_rate = this.calAgeRate();**  // float age\_rate = calAgeRate(); // they are equal  return age\_rate \* .1f;  }  public static void main(String[] args){  Method2 c = new Method2();  System.out.println(c.getTotal());  }  } |

overload instance methods

|  |
| --- |
| output:  go1  3  Test@2c7b84de |
| public class Test{  int age;  void go(){  System.out.println("go1");  }  void go(int a){  System.out.println(a);  }  void go(Object o){  System.out.println(o.toString());  }  public static void main(String[] args) {  Test t = new Test();  t.go();  t.go(3);  t.go(new Test());  }  } |

|  |
| --- |
| invoke method must follow the patterns of declaration. |
| public class MethodArgument {  **void test1(){**  System.out.println("print");  }  **void test2(int a, float b){**  System.out.println(a+b);  }  public static void main(String[] args){  MethodArgument C = new MethodArgument();  **C.test1(); //no arguments**  **C.test2(3, 3.4f); //same type and order**  }  } |
|  |

|  |
| --- |
| inheritance and override |
| public class Test extends Animal{  public static void main(String[] args) {  Test t = new Test();  t.go();  }  }  class Animal {  void go(){  System.out.println("animal");  }  } |
| public class Test extends Animal{  void go(){  System.out.println("test");  }  public static void main(String[] args) {  Test t = new Test();  t.go();  }  }  class Animal {  void go(){  System.out.println("animal");  }  } |

### pass arguments

* must pass the same type arguments
* For immutable type object namely primitive type or String, pass arguments by value.
* For mutable type object, pass reference only. Any calls to object methods within the method are reflected in the caller.

Data type between caller and declaration should be identical or compatible.

|  |
| --- |
| output: 33 |
| public class Test{  int age;  void go(int a){  System.out.print(a);  }  public static void main(String[] args) {  Test t = new Test();  short a =3;  t.go((a)); //automatic casting  t.go((int) 3.4f); //explicit casting  }  } |

The parameter obj support any variable. The class is java.lang.Object

|  |
| --- |
| output:  abc  23.0  Test@3fee733d |
| public class Test{  int age;  void go(Object a){  System.out.println(a.toString());  }  public static void main(String[] args) {  Test t = new Test();  t.go("abc");  t.go(23.f);  t.go(new Test());  }  } |
|  |

pass unknown number of arguments using "var-args".

|  |
| --- |
| four approaches below are good. |
| public class MethodArgument {  **void test3(int a, float... args){**  if (args.length > 0){  System.out.println(a+args[0]);  }  }  public static void main(String[] args){  MethodArgument C = new MethodArgument();  C.test3(3);  C.test3(3, 3.1f);  C.test3(3, 3.1f, 3.2f);  float[] a= {3.1f, 3.2f};  C.test3(3, a);  float[] b = new float[10];  C.test3(3, b);  }  } |

|  |
| --- |
| output:0120 |
| public class MethodVarargs {  static void test(boolean a, boolean... b){  System.out.print(b.length);  }  public static void main(String[] args){  test(true);  test(true, true);  test(true, true, false);  test(true, new boolean[0]);  // wrong: test(true, {true, false});  //wrong test(true, new boolean[]);  }  } |

pass argument by value

|  |
| --- |
| output: 63 |
| public class Test{  int age;  void go(int a){  a +=3;  System.out.print(a);  }  public static void main(String[] args) {  Test t = new Test();  int a =3;  t.go(a);  System.out.println(a);  }  }  class Animal {  void go(){  System.out.println("animal");  }  } |

pass object by reference

|  |
| --- |
| output:0122 |
| public class ObjectGarbage {  public static void main(String[] args){  Test d = new Test();  System.out.println(d.a); //==0  Test d1 = d.do1(d);  System.out.println(d1.a); //==1  Test d2 = d.do2(d);  System.out.println(d1.a); //==2  System.out.println(d.a); //==2  }  }  class Test{  int a =0;  Test do1(Test cb){  a =1;  return cb;  }  Test do2(Test cb){  cb = null;  a=2;  return cb;  }  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### return type

return type must be defined when a method is declared.

* **Compatible**: return type and return statement must be consistent or compatible.
* **Value return**: For immutable type object namely primitive type or String, return value.
* **Object return**: For mutable type object, keep an eye on how to return reference variable or object.
* **Covariant return type**: In case of overriding methods, the return type in child method is allowed to be a subclass of that defined in parent class.

No return is allowed.

|  |
| --- |
| the below are equal |
| void test(){} |
| void test(){;} |
| void test(){return;} |

keep consistency of return type and return statement

|  |
| --- |
| output: 3  return int type |
| int test(){  return 3;  } |
| int test(){  return (int) 3.14; //casting  } |
| int test(){  short a = 3; return a;  } |
| int test(){  return (short) 3.14;  } |

Return type could be overridden if the return types of two methods are covariant and reference type. The method in subclass must declare more specific type than that in superclass. Both return type should be reference type to object rather than primitive type or String.

|  |
| --- |
| output: child child |
| public class MethodReturn extends Test {  **MethodReturn test5(){**  System.out.println("child");  return new MethodReturn();  }  **Test test6(){**  System.out.println("child");  return new Test();  }  public static void main(String[] args) {  MethodReturn c = new MethodReturn();  c.test5();  c.test6();  }  }  class Test{  **Test test5(){**  System.out.println("parent");  return new Test();  }  **Object test6(){**  System.out.println("parent");  return new Test();  }  } |

If reference type is passed into a method, reference variable within the method and outside of it would refer to the same object. Any update to this object within the method could be visible outside the method. Thus, reference type referring to the same object is not required in some cases.

|  |
| --- |
| output:  4066  4066  4066  [a, b, c]  [a, b]  126145  126145  Note: getList() return reference variable rather than object itself. list3, list2 and list are reference variables that refer to the same object created by constructor. |
| import java.util.ArrayList;  public class MethodReturn2 {  private ArrayList<String> list = new ArrayList<String>();  MethodReturn2(){  list.add("a");  list.add("b");  }  ArrayList getRef(){  ArrayList<String> list2 = list;  System.out.println(list.hashCode());  System.out.println(list2.hashCode());  return list2;  }  ArrayList getObj(){  ArrayList<String> list2 = new ArrayList<String>(list);  list2.add("c");  System.out.println(list2);  System.out.println(list);  System.out.println(list2.hashCode());  return list2;  }  public static void main(String[] args){  MethodReturn2 t = new MethodReturn2();  ArrayList<String> list3 = t.getRef();  System.out.println(list3.hashCode());  ArrayList<String> list4 = t.getObj();  System.out.println(list4.hashCode());  }  } |

If return type is reference type, object casting is allowed. That is one pattern of polymorphism. Moreover, return null is allowed.

|  |
| --- |
| output:  tiger  animal |
| public class MethodReturn3 extends Tiger{  **static Animal test1(){**  System.out.println("tiger");  **return new Tiger();**  }  **static Animal test2(){**  System.out.println("animal");  **return new Animal();**  }  public static void main(String[] args){  test1();  test2();  }  }  class Animal{}  class Tiger extends Animal {} |

|  |
| --- |
| output:  [java.lang.Object@5acf9800, Animal@4617c264, Tiger@36baf30c, null]  [java.lang.Object@5acf9800, Animal@4617c264, Tiger@36baf30c, null] |
| import java.util.ArrayList;  public class MethodReturn4 {  static ArrayList<Object> test(ArrayList<Object> x){  **x.add(new Object());**  **x.add(new Animal());**  **x.add(new Tiger());**  **x.add(null);**  return x;  }  public static void main(String[] args){  ArrayList<Object> list = new ArrayList<>();  ArrayList<Object> list2 = test(list);  System.out.println(list.toString());  System.out.println(list2.toString());  }  }  class Animal{}  class Tiger extends Animal {} |

|  |
| --- |
| output:  child |
| public class MethodReturn extends Test {  MethodReturn test5(){  System.out.println("child");  return new MethodReturn();  }  public static void main(String[] args) {  MethodReturn c = new MethodReturn();  c.test5();  }  }  class Test{  Test test5(){  System.out.println("parent");  return new Test();  }  } |

## instance variable

In class, Variable could be instance variable, or static variable (class variable), or local variable.

* Local variables and static variables are placed on the stack. Static variables are initialized before instance, and the order is static block, initialize block and constructor.
* If there is inheritance, variables declared in class can't be overridden but be hidden.

### instance variable

Instance variables are non-static variables and are declared in a class outside any method, constructor, or block.

* Instance variables should be declared in class outside class methods, constructor or other block.
* Instance variables are bound to objects. Instance variables like objects live on the heap. Instance variables are created when an object of the class is created and destroyed when the object is destroyed. They should be accessed by reference variable which refers to object/instance of class. Therefore, one object would have one set of instance variables.
* instance variables could be initialized when they are declared , or in instance initializer, or constructor, or omitted. Instance initializer are blocks of code that are run before the constructor. The order is initializer block, and then constructor. Differing from local variables, omit initialization for instance variables is ok.
* Use this. or super. if instance variables or those from superclass are called.
* In an encapsulated class, instance variables are of **private scope** and getters/setters are of **public scope**.

Distinguish instance variables from other variables in class.

|  |
| --- |
| name is instance variable. type is local variable. age is static/class variable. |
| class Test{  String name;  static int age = 1;  void go(){ String type; }  } |
|  |

|  |
| --- |
| default values of instance variables |
| String s == null  int num == 0  boolean b == false  float d == 0.0 |

The below shows 4 methods on initialization of instance variables.

|  |
| --- |
| output:  0  tiger  M  Animal |
| public class VariableInstance {  int age;  String name;  char gender = 'F';  String specie = "Animal";  {  gender = 'M';  }  VariableInstance(String name){  this.name = name;  }  void print(){  System.out.println(this.age);  System.out.println(this.name);  System.out.println(this.gender);  System.out.println(this.specie);  }  public static void main(String[] args){  new VariableInstance("tiger").print();  }  } |

Call instance variables using this. or super. in instance methods. Don't call instance variables in static methods.

|  |
| --- |
| output:  1  tiger  animal  this. could be omitted if there is no such local variables have the same name as instance variable. |
| public class VariableInstance2 extends Test{  int age = 1;  String name = "tiger";  void print(){ //call instance variables in instance methods only  System.out.println(age);  System.out.println(this.name);  System.out.println(super.name);  }  public static void main(String[] args) {  new VariableInstance2().print();  }  }  class Test{  String name = "animal";  } |

Moreover, instance variable could be reference variable referring to an object.

|  |
| --- |
| output: na |
| public class VariableInstance3{  **Test t1 = new Test();**  public static void main(String[] args) {  VariableInstance3 t = new VariableInstance3();  System.out.println(**t.t1.name**);  }  {  t1.name = "na";  }  }  class Test{  String name = "test";  } |

### local variable

Distinguish local variable, instance variable and static variable. See the example below

|  |
| --- |
|  |
| public class Type {  int integer; //instance variable. default is 0  static int num; // class variable or static variable. default is 0  public Type() {  int type; // local variable. No initialization  }  } |

|  |
| --- |
| a is instance variable. b and c is local variable |
| class Test{  int a;  {int b;}  void go(){ int c;}  } |
|  |

|  |
| --- |
| instance variable a would not be initialized in constructor because both of the variables of "a=a;" are local variable. |
| class Test{  int a;  Test(int a){ a=a;}  } |

Compare the three approaches below.

|  |
| --- |
| Color: null |
| public class Egret {  private String color;  public Egret() { this("white"); }  **public Egret(String color) { color = color; } //both variables are local**  public static void main(String[] args) {  Egret e = new Egret();  System.out.println("Color:" + e.color);  }  } |
| Color: White |
| public class Egret {  private String color;  public Egret() { this("white"); }  **public Egret(String color) { this.color = color; }**  public static void main(String[] args) {  Egret e = new Egret();  System.out.println("Color:" + e.color);  }  } |
| public class Egret {  private String color;  public Egret() { this("white"); }  **public Egret(String x) { color = x; }**  public static void main(String[] args) {  Egret e = new Egret();  System.out.println("Color:" + e.color);  }  } |
|  |

Watch out reference type local variable, which may refer to the same object as a certain instance variable. Therefore, any invoke may have impact on something outside of instance methods.

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### Order of initialization

* Keep in mind that only static blocks should be invoked before class instantiation. Only when a class is instantiated, firstly invoke instance initializer block, then constructor.
* **Static blocks would be invoked only if main() exists in this class.**
* **Instance initializer would be invoked only if class is instantiated.** The order is from the top to the end of class excluding instance methods.
* A certain constructor would be invoked only if class is instantiated.
* In case of inheritance, first initialize base class from the most base class then initialize child class. Within a certain class, order of initialization follows the above rule.

|  |
| --- |
| output: dacbe  initializer block -> constructor |
| public class ConstructorOrder {  {  **System.out.print("a"); //3**  }  **ConstructorOrder(){**  **System.out.print("b"); //5**  **}**  {  **System.out.print("c"); //4**  }  public static void main(String[] args){  **System.out.print("d"); //1**  new ConstructorOrder();  **System.out.print("e"); //2**  }  } |

What happens if there is static block? Static block will be executed before main() is executed. Note: instantiation of static block is not determined by instantiation of objects.

|  |
| --- |
| output: cd |
| public class ConstructorOrder {  static int a;  {  System.out.print("a");  }  ConstructorOrder(){  System.out.print("b");  }  static {  a=10;  **System.out.print("c"); //1**  }  public static void main(String[] args){  **System.out.print("d");//2**  }  } |

|  |
| --- |
| output: cdabe |
| public class ConstructorOrder {  static int a;  {  System.out.print("a");  }  ConstructorOrder(){  System.out.print("b");  }  **static {**  **a=10;**  **System.out.print("c");**  **}**  public static void main(String[] args){  System.out.print("d");  new ConstructorOrder();  System.out.print("e");  }  } |

|  |
| --- |
| output: fcd |
| public class ConstructorOrder extends Test{  static int a;  {  System.out.print("a");  }  ConstructorOrder(){  System.out.print("b");  }  static {  a=10;  **System.out.print("c");**  }  public static void main(String[] args){  **System.out.print("d");**  }  }  class Test{  static int b;  static {  b=10;  **System.out.print("f");**  }  } |

|  |
| --- |
| output: acbd |
| public class Letters {  { System.out.print("a"); }  public Letters() { { System.out.print("b"); } }  { System.out.print("c"); }  public static void main(String[] args) {  Letters a = new Letters();  { System.out.print("d"); }  }  } |

## Encapsulation

Encapsulation is to wrap up private instance variables and instance methods together as a single block. Here are some advantages of encapsulation

* Encapsulation help protection, details of data or behavior in class are hidden to callers. Callers can't access instance variables directly but through instance methods.
* Encapsulation would decouple components such read-only (getter) or write-only access (setter).
* Encapsulation would bind variable with behaviors together. That is highly integrated within instance methods.

For encapsulation, variables should be marked as "private". Therefore, those variables must be access by this class only. Any access, namely GET or SET, should be done from specific methods respectively. For example, when following JavaBean naming conventions, the accessor method for the property boolean b.

|  |
| --- |
|  |
| class Test{  **private boolean b;**  public boolean isB() { return b; }  public boolean getB(){ return this.b;}  public void setB(boolean b) { this.b =b;}  } |
|  |

Here are common rule:

* define a class which encapsulates attributes (variables) and methods.
* Attributes are marked as private, and all setter or getter methods are marked as public. They can't be accessed directly from outside but operated by certain methods. encapsulate variables as attributed and access variable by methods
* Access privilege of other methods are assigned with private/protected/<default>/public.
* With encapsulation, object become immutable where instance variables can't be changed once the object is created.

Here is one example of encapsulation that wrap up instance variables .

|  |
| --- |
| encapsulate attribute age, which is initialized by constructor. The methods setter and getter are defined. |
| public class Method1 {  **private int age;**  Method1(){ age=10; }  **public void setAge(int age){** this.age=age; }  **public int getAge(){** return this.age; }  public static void main(String[] args){  Method1 c = new Method1();  c.setAge(20);  System.out.println(c.getAge());  }  } |

Immutability of class does not allow setters.

|  |
| --- |
| In this case, the object c is immutable |
| public class Method{  **private int age;**  Method(){ age=10; }  **public int getAge(){** return this.age; }  public static void main(String[] args){  Method c = new Method();  c.setAge(20);  System.out.println(c.getAge());  }  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

# OOP: static

Java support static class, static instance variables, static methods, static variables, static block.

## static variable

* Class initialization initialize static variables. So, static variable is also known as class variable. Differing from instance variable, static variable belongs to a class rather than a particular instance.
* static variable is always initialized once with class initialization. That means a static variable could be used by multiple instances of the class.
* Static variable is accessed by class name rather than reference variables and don't need any object..
* That is stored in stack. The value is always the newest value.

Here are how do we define, implement a static variable.

|  |
| --- |
| // pattern 1: initialized in constructor  // Not suggested. usually remove static |
| public class StaticInitial {  static int a;  StaticInitial(){  a = 1;  }  } |

|  |
| --- |
| // pattern 2: initialized at static block |
| public class StaticInitial {  static int a =1;  static int b;  static {  b = 2;  // static variable can't be updated.  //No exception but does not work.  a=3;  }  } |

|  |
| --- |
| //pattern 3: initialized at declaration |
| class Test{  public static int c = 3;  } |

Compare the difference between instance and static variables in class.

|  |  |
| --- | --- |
| output |  |
| 10  age is instance variable | public class Test {  int age;  Test(){ **this.age = 10;** }  public static void main(String[] args) {  Test t = new Test();  System.out.println(t.age);  }  } |
| 20  age is static variable. main() will initialize the class Test ( age =0), and two instance will update age twice( age=10, age=20). {20, 10, 0} are stored in static stack. 20 is the newest one. | public class Test {  **static int age;**  Test(int age){ this.age += age; }  public static void main(String[] args) {  Test t1 = new Test(10);  Test t2 = new Test(10);  System.out.println(t2.age);  }  } |
| 11  initialize class Test: age=0;  t1 instance: age=1, age = 11  t2 instance: age =1, age =11  static stack {11,1,11,1,0} | public class Test {  static int age;  Test(int age){  this.age += age;  }  public static void main(String[] args) {  Test t1 = new Test(10);  Test t2 = new Test(10);  System.out.println(t2.age);  }  { age = 1; }  } |
| 21  initialize class Test: age=0;  static block: age = 1;  t1 instance: age = 11  t2 instance: age =21  static stack {21,11,1,0} | public class Test {  static int age;  Test(int age){  this.age += age;  }  public static void main(String[] args) {  Test t1 = new Test(10);  Test t2 = new Test(10);  System.out.println(t2.age);  }  static { age = 1; }  } |
| output: cbdada22  static stack {22,12,2,1,0} | public class Test extends Animal{  Test(int age){  System.out.print("a");  this.age += age;  }  public static void main(String[] args) {  Test t1 = new Test(10);  Test t2 = new Test(10);  System.out.println(t2.age);  }  static {  age = 2;  System.out.print("b");  }  }  class Animal{  static int age;  static {  age = 1;  System.out.print("c");  }  {  System.out.print("d");  }  } |

|  |
| --- |
|  |
|  |

static variable is only initialized one time.

Initialization of class static variable is not required but initialization is ok.

|  |
| --- |
| output: 12  class static variable age is not initialized |
| public class StaticInitial2 extends Test{  public static void main(String[] args) {  Test c1 = new Test();  System.out.print(c1.age);  Test c2 = new Test();  System.out.println(c2.age);  }  }  class Test{  **static int age;**  Test(){  this.age += 1;  }  } |
| output: 1 |
| public class StaticInitial3 extends Test{  static int age = 7;  public static void main(String[] args) {  new Test().go(age);  System.out.println(Test.age);  }  }  class Test{  static int age;  Test(){  this.age += 1;  }  void go(int age){  age++;  }  } |

|  |
| --- |
|  |
|  |

import static variables from other class.

|  |
| --- |
| stative variable LENGTH is stored in path a/b/c.java |
| import static a.b.c.LENGTH; |
| impot static a.b.c.\*; |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

## static methods

It is allowed to define static methods in abstract class or interface. Here are the rules:

* static methods must be marked as static and consist of method body.
* Defined in abstract class, static methods could be default/protected/public. private is not allowed.
* Defined in interface, static methods are public access by default. private/protected is not allowed.
* Static methods can't be marked as final or abstract or private.
* Static methods are invoked in static context or through instance methods.
* Static methods can't be overridden but hidden.

Here are examples for declaration static methods in abstract class and interface.

### Invoke static methods

invoke static methods in static context

invoke static methods in instance context

|  |
| --- |
|  |
| public class StaticMethod2 {  static void test(){  System.out.println("test1");  }  public static void main(String[] args) {  // invoke static methods  // method 1: class name+method name  **StaticMethod2.test();**  // method 2: method name within this class  **test();**  // method 3: instance method  **new StaticMethod2().test();**  }  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### In abstract class

|  |
| --- |
| The below are ok. |
| abstract class Test{  static void getName(){  System.out.println("name");  }  } |
| abstract class Test{  protected static void getName(){}  } |
| abstract class Test{  public static void getName(){;}  } |

|  |
| --- |
| output: name |
| public class InterfaceTest extends Test{  public static void main(String[] args){  **Test.getName();**  }  }  abstract class Test{  public static void getName(){  System.out.println("name");  }  } |

|  |
| --- |
| the access should be compatible. |
| public class MyTest extends Test{  **protected static void getName(){**  System.out.println("concrete");  }  public static void main(String[] args){  **Test.getName(); // static method in abstract class**  **getName(); // static method in virtual class**  }  }  abstract class Test{  **protected static void getName(){**  System.out.println("name");  }  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### In interface

|  |
| --- |
| They are equal |
| interface Test{  static void getName(){ }  } |
| interface Test{  public static void getName(){ }  } |
| interface Test{  static void getName(){ ; }  } |
| interface Test{  static void getName(){ return; }  } |

invoke static methods directly by add prefix of interface. Don’t invoke static method as instance methods.

|  |
| --- |
| output: animal |
| public class InterfaceStatic implements Animal{  public static void main(String[] args) {  **Animal.getName();**  }  }  interface Animal{  static void getName(){  System.out.println("animal");  }  } |

As the patterns of referencing static methods, overridden doesn't occur.

|  |
| --- |
| output:  animal  test |
| public class InterfaceStatic2 implements Animal{  static void getName(){  System.out.println("test");  }  public static void main(String[] args) {  **Animal.getName(); // reference interface static**  **getName(); //reference concrete class static**  }  }  interface Animal{  static void getName(){  System.out.println("animal");  }  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

## static class

basic rules:

* If a class is defined within another class, there are inner class and top-level class. Inner class could be declared as static class. Therefore, static classes are always nested classes.
* A top-level class can' be declared as static.
* Static nested class can be instantiated without instance of the outer class.

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### static object

|  |
| --- |
| output:  2  10 |
| import learn.Rope;  public class StaticClass {  **private static Rope r1 = new Rope();**  **private static Rope r2 = new Rope();**  // executed only when instantiation  {  System.out.println(r1.length);  }  public static void main(String[] args) {  r1.length += 2;  System.out.println(r1.length);  r2.length += 8;  System.out.println(r1.length);  }  } |

|  |
| --- |
|  |
|  |

## final identifier

* final instance methods can't be overridden.
* Any methods defined in abstract class or interface can't be marked as final.
* Final virtual class can't be extended. Either abstract class or interface can't be marked as final.
* final type of variables can only be set once. They would not be changed once they are initialized. Final variables acts like constants.
* Final variables could be initialized in declaration, or instance initializer or constructor.

declare and initialize final variables. Initialize final variables during declaration, or declaration of blank final variables followed by inside instance-initializer block or inside constructor or static block.

|  |
| --- |
| The three approaches for initialization is ok. |
| public class ClassTest {  final int age;  { //initializer block  age =10;  }  public static void main(String[] args){  ClassTest t = new ClassTest();  System.out.println(t.age);  }  } |
| public class ClassTest {  final int age=10; //initialized at declaration  public static void main(String[] args){  ClassTest t = new ClassTest();  System.out.println(t.age);  }  } |
| public class ClassTest {  final int age;  ClassTest(int a){  this.age = a; //initialized in constructor  }  public static void main(String[] args){  ClassTest t = new ClassTest(1);  System.out.println(t.age);  ClassTest t2 = new ClassTest(10);  System.out.println(t2.age);  }  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

# OOP: Inheritance

## Inheritance

relationship between class A and B: inherit, or import. Inheritance is stronger.

Java doesn't support multiple inheritance. Therefore, a certain class always has one parent if there is inheritance.

single inheritance:

|  |
| --- |
| B->A |
| class A{}  class B extends A{} |

multilevel inheritance

Note: multiple inheritance is wrong. For example, C->A,B

|  |
| --- |
| C->B->A |
| class A {}  class B extends A{}  class C extends B{} |

hierarchical inheritance

|  |
| --- |
| C->A and B->A. B and C are siblings (no inheritance) |
| class A{}  class B extends A{}  class C extends A{} |

|  |
| --- |
|  |
|  |

### inherit constructor

* In child class, constructor may be omitted. super() may be omitted if one parent constructor doesn't declare arguments.
* If parent class have argument constructor, child class must define constructor, and explicitly instantiate it using super(), and super() must be placed on the top of the child constructor.
* inherit various parent constructor if there are multiple parent constructor.

constructors are executed in order starting from the constructors of base, some derived and finally child class.

|  |
| --- |
| output:  animal mammal tiger  11 |
| public class ConstructorOrder {  public static void main(String[] args){  new Tiger();  }  }  class Animals{  int age;  Animals(){  this.age = 1;  System.out.print("animal ");  }  }  **class Mammals extends Animals{**  Mammals(){  **super(); //could be ommited**  System.out.print("mammal ");  }  }  **class Tiger extends Mammals{**  Tiger(){  System.out.println("tiger ");  // this. and super. are equal  System.out.print(super.age);  System.out.println(this.age);  }  } |

|  |
| --- |
| output:  1malenull  0NAliving |
| public class ConstructorSuper {  public static void main(String[] args){  Mammal b1 = new Mammal("male");  b1.Print();  Mammal b2 = new Mammal();  b2.Print();  }  }  class Mammal extends Animal{  String gender;  Mammal(String gender){  super(1);  this.gender = gender;  }  Mammal(){  super("living");  this.gender = "NA";  }  void Print(){  System.out.print(this.age);  System.out.print(this.gender);  System.out.println(this.status);  }  }  class Animal{  int age;  String status;  Animal(int age){  this.age = age;  }  Animal(String status){  this.status = status;  }  } |

|  |
| --- |
|  |
|  |

### inherit instance methods

|  |
| --- |
| call parent method from child object |
| public class ClassMethod4 extends Test{  int age = 10;  float calAgeRate(){  return (float) this.age \* 10;  }  float getTotal1(){  **float age\_rate = super.calAgeRate();**  return age\_rate \* .1f;  }  public static void main(String[] args){  ClassMethod4 c = new ClassMethod4();  System.out.println(c.getTotal1());  }  }  class Test{  int age = 10;  float calAgeRate(){  return (float) this.age \* 15;  }  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### inherit exceptions

|  |
| --- |
|  |
| //simple inheritance using default constructor  public class ConstructorException extends Parent{  public static void main(String[] args) {  new ConstructorException();  }  }  class Parent{} |
| //inheritance with constructor  public class ConstructorException extends Parent{  ConstructorException(){  System.out.println("child");  }  public static void main(String[] args){  new ConstructorException();  }  }  class Parent{  Parent(){  System.out.println("parent");  }  } |
|  |

For checked exceptions such as java.lang.Exception, java.io.IOException, or java.io.FileNotFoundException, child constructor and main() should throw exceptions if parent constructor declare exceptions, and their exception type should be consistent.

For unchecked exceptions, there is no such limits as checked exception.

|  |
| --- |
| output:  parent  child |
| //inherit constructor with exception  public class ConstructorException extends Parent{  ConstructorException() throws Exception{  System.out.println("child");  }  public static void main(String[] args) throws Exception{  new ConstructorException();  }  }  class Parent{  Parent() throws Exception {  System.out.println("parent");  }  } |
| //IOException  import java.io.IOException;  public class ConstructorException extends Parent{  **ConstructorException() throws IOException{**  System.out.println("child");  }  **public static void main(String[] args) throws IOException{**  new ConstructorException();  }  }  class Parent{  **Parent() throws IOException {**  System.out.println("parent");  }  } |
| //child class could miss RuntimeException.  public class ConstructorException extends Parent{  ConstructorException(){  System.out.println("child");  }  public static void main(String[] args){  new ConstructorException();  }  }  class Parent{  **Parent() throws RuntimeException {**  System.out.println("parent");  }  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

## overloading

In class, if two or more methods share the same name but differ in arguments, those methods are overloaded methods. Calling a method is method overloading. Overloading occurs in runtime rather than compiling process.

When overloaded methods are declared:

* method name must be identical
* arguments of methods must be different, name various number or data type.
* Return type could be identical, but it is ok if they are different.
* method could be overloaded within one class or subclass.
* method could declare new or wider exceptions.
* method could be marked as different access identifier.

NO assertions:

* don't overload operators as in C++

### overload constructor

constructor can be overloading but can't be override

overloading constructor:

1. same name and privilege but different arguments

|  |
| --- |
| three constructors are defined. |
| public class ConstructorThis {  int a=-1;  static int b=-1;  String c;  **public ConstructorThis(){**  this.a=1;  b=2;  }  **public ConstructorThis(int a){**  a = a+1;  }  **public ConstructorThis(String c){**  c=this.c;  }  public static void main(String[] args){  ConstructorThis c1 = new ConstructorThis();    int a=3;  ConstructorThis c2 = new ConstructorThis(a);  String c="abc";  ConstructorThis c3 = new ConstructorThis(c);  }  } |

employ constructor chaining when overload constructors.

|  |
| --- |
| output:  #2  #0  NA10  #2  #1  single10  #2  two34 |
| public class ConstructorChaining {  private String name;  private int age=10;  // constructor chaining  public ConstructorChaining(String name, int age){  this.name = name;  this.age = age;  System.out.println("#2");  }  public ConstructorChaining(String name){  this.name = name;  this.age = 10;  }  public ConstructorChaining(){  this.name = "NA";  this.age = 0;  System.out.println("#0");  }  public void Print(){  System.out.print(this.name);  System.out.println(this.age);  }  public static void main(String[] args){  new ConstructorChaining().Print();  new ConstructorChaining("single").Print();  new ConstructorChaining("two", 34).Print();  }  } |
| public class ConstructorChaining {  private String name;  private int age=10;  public ConstructorChaining(String name, int age){  this.name = name;  this.age = age;  System.out.println("#2");  }  public ConstructorChaining(String name){  this(name, 10);  System.out.println("#1");  }  public ConstructorChaining(){  this("NA", 10);  System.out.println("#0");  }  public void Print(){  System.out.print(this.name);  System.out.println(this.age);  }  public static void main(String[] args){  new ConstructorChaining().Print();  new ConstructorChaining("single").Print();  new ConstructorChaining("two", 34).Print();  }  } |

|  |
| --- |
|  |
|  |

### overload methods

for overloading methods, return type, access modifier, and exception list may vary. If the method name was different, it would be a different method rather than an overloaded method.

same method name and return type but different arguments.

The below overloading methods are legal.

|  |
| --- |
| public void test(){} |
| public void test(String sound){} |
| public void test(boolean soft){} |
| public void test(int… time){} //different arguments |
| public static void test(){} //static |
| private void test(Integer a){} //different access |

being done during execution time.

could occur within one class or inheritance

The order of overloading: exact match, widening primitives, autoboxing, varargs.

|  |
| --- |
| overload methods |
| public class ClassMethod4 extends Test{  int age = 10;  float calAgeRate(){  return (float) this.age \* 10;  }  **float getTotal(){**  float age\_rate = calAgeRate();  return age\_rate \* .1f;  }  **float getTotal(float rate){**  float age\_rate = this.calAgeRate();  return age\_rate \* rate;  }  public static void main(String[] args){  ClassMethod4 c = new ClassMethod4();  System.out.println(c.getTotal());  System.out.println(c.getTotal(.15f));  }  } |
| public class ClassMethodOverload extends Test{  int age = 10;  **float calAgeRate(){**  return (float) this.age \* 10;  }  float getTotal(Integer rate){  **float age\_rate = rate==null ? calAgeRate() : calAgeRate(rate);**  return age\_rate \* .1f;  }  public static void main(String[] args){  ClassMethodOverload c = new ClassMethodOverload();  System.out.println(c.getTotal(null));  System.out.println(c.getTotal((Integer)16));  }  }  class Test{  int age = 10;  **float calAgeRate(Integer rate){**  return (float) this.age \* rate;  }  } |

|  |
| --- |
| output:10.0. only call the first one if two overloading methods with same arguments  the compiling is ok but that is coding error. One of them should deleted. |
| public class ClassMethod4 extends Test{  int age = 10;  float calAgeRate(){  return (float) this.age \* 10;  }  **float getTotal(){**  float age\_rate = calAgeRate();  return age\_rate \* .1f;  }  **float getTotal(){**  float age\_rate = calAgeRate();  return age\_rate \* .5f;  }  public static void main(String[] args){  ClassMethod4 c = new ClassMethod4();  System.out.println(c.getTotal());  }  } |

|  |
| --- |
| output:  many 1 one 3 one 2 |
| public class Test{  private void layEggs(int... eggs) {  System.out.print("many " + eggs[0] + " ");  }  private void layEggs(int eggs) {  System.out.print("one " + eggs + " ");  }  public static void main(String[] args) {  Test c = new Test();  c.layEggs(1, 2);  c.layEggs(3);  **c.layEggs(new Integer(2)); //invoke layEgg(int eggs)**  }  } |

|  |
| --- |
|  |
|  |

### return type

return type of overloading methods is usually identical, but different return type of overloading methods is ok.

|  |
| --- |
|  |
| public class OverloadReturn {  **static void go(){**  System.out.println("no return");  }  **static int go(int a){**  System.out.println("return " + a);  return a;  }  public static void main(String[] args) {  go();  go(3);  }  } |

|  |
| --- |
|  |
| public class OverloadReturn extends Test{  **void go(){**  System.out.println("no return");  }  public static void main(String[] args) {  OverloadReturn t = new OverloadReturn();  t.go();  t.go(10);  }  }  class Test{  **int go(int a){**  System.out.println("return " + a);  return a;  }  } |

### Cast argument

Match arguments for overloading method:

1. detect exact match.

2. larger primitive type by casting: automatic convert short bytes to longer bytes

3. autoboxing: convert primitive type to wrapper class

4. varargs:

|  |
| --- |
| output:  int  int  double |
| public class OverloadArgument1 {  static void go(int a){  System.out.println("int");  }  static void go(double a){  System.out.println("double");  }  public static void main(String[] args){  go(3); // exact match  short a =3;  go(a); //convert short to int  go(3.3f); // convert folat to double;  }  } |

|  |
| --- |
|  |
| public class OverloadArgument2 {  static void go(Integer a){  System.out.println("Integer");  }  static void go(float a){  System.out.println("float");  }  static void go(Object a){  System.out.println("Object");  }  public static void main(String[] args){  go(3); // autoboxing  short a =3;  go(a); //autoboxing  go(3.3); // convert double to object;  go("abc"); // convert double to object;  }  } |

|  |
| --- |
| output:  Integer  Integer  Object  Object  Integer |
| public class OverloadArgument2 {  static void go(Object a){  System.out.println("Object");  }  static void go(Integer a){  System.out.println("Integer");  }  static void go(float a){  System.out.println("float");  }  public static void main(String[] args){  Integer a= Integer.valueOf(3);  go(a); // Integer  short b =3;  go(a); //convert short to int and then autoboxing  go(3.3); // convert double to object;  go("abc"); // convert double to object;  **go(null); //Integer**  }  } |

|  |
| --- |
| output:  varargs  int  varargs |
| public class OverloadArgument3 {  static void go(int a){  System.out.println("int");  }  **static void go(Object... args){**  System.out.println("varargs");  }  public static void main(String[] args){  go();  go((short)45);  go(true, 34f, null);  }  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

## override

For inheritance relationship, child methods override methods in parent class if those methods have the same signatures. Overridden methods are replaced at runtime in all places it is defined.

Compared with overridden method in super class, the methods in subclass

* have identical method name and argument list.
* access identifier should be equivalent or wider than that in super class. For example, protected in super-class, and public in subclass
* return type should be covariant - should be equivalent or narrower than that in super class.
* Throwed exception should be equivalent or narrower than that in super class.

**Overloaded methods** share the same name but different signatures and have no polymorphic relationship.

### between virtual class

|  |
| --- |
| child method overrides calAgeRate(). |
| public class ClassMethod3 extends Test{  int age = 10;  **float calAgeRate(){**  return (float) this.age \* 10;  }  float getTotal1(){  **float age\_rate = calAgeRate();**  return age\_rate \* .1f;  }  float getTotal2(){  **float age\_rate = this.calAgeRate();**  return age\_rate \* .1f;  }  public static void main(String[] args){  ClassMethod3 c = new ClassMethod3();  System.out.println(c.getTotal1());  System.out.println(c.getTotal2());  }  }  class Test{  int age = 10;  **float calAgeRate(){**  return (float) this.age \* 15;  }  } |

|  |
| --- |
| If child instance method could override parent method, child method is default invoked or parent method could be invoked with "super.". |
| public class Method3 extends Test{  int age = 10;  float calAgeRate(){  return (float) this.age \* 10;  }  float getTotal1(){  **float age\_rate = calAgeRate();**  return age\_rate \* .1f;  }  float getTotal2(){  **float age\_rate = super.calAgeRate();**  return age\_rate \* .1f;  }  public static void main(String[] args){  Method3 c = new Method3();  System.out.println(c.getTotal1());  System.out.println(c.getTotal2());  }  }  class Test{  int age = 10;  float calAgeRate(){  return (float) this.age \* 15;  }  } |

If parent reference variable is casted to child object, child method would override parent method. That is polymorphism.

|  |
| --- |
| output:  DeerReindeer,true  parent reference refers to child object and invokes child method |
| public class Deer {  public Deer() {  System.out.print("Deer");  }  public Deer(int age) {  System.out.print("DeerAge");  }  **public boolean hasHorns() {**  return false;  }  public static void main(String[] args) {  **Deer deer = new Reindeer(5);**  System.out.println(","+deer.hasHorns());  }  }  class Reindeer extends Deer {  public Reindeer(int age) {  System.out.print("Reindeer");  }  **public boolean hasHorns() {**  return true;  }  } |

But if the parent method is private method, which can't be overridden but by hidden.

|  |
| --- |
| output:  DeerReindeer,false |
| public class Deer {  public Deer() {  System.out.print("Deer");  }  public Deer(int age) {  System.out.print("DeerAge");  }  **private boolean hasHorns() {**  return false;  }  public static void main(String[] args) {  **Deer deer = new Reindeer(5);**  System.out.println(","+deer.hasHorns());**//That is hidden**  }  }  class Reindeer extends Deer {  public Reindeer(int age) {  System.out.print("Reindeer");  }  **public boolean hasHorns() {**  return true;  }  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### access identifier

Access of method that override the method in super class should be wider than access of that method.

* Overridden methods declared as public in a superclass must be public in all subclasses.
* Overridden methods declared as protected in a superclass must be either protected or public in subclasses. They cannot be private in subclasses.
* Overridden methods declared as private or final can't be inherited at all. There are no overridden methods even if they looks similar. They are hidden to all subclasses. Therefore, methods in subclasses could be marked as any access identifier.

|  |
| --- |
| ouput: child  public >protected. |
| public class MethodFinal extends Test{  **public void test(){**  System.out.println("child");  }  public static void main(String[] args){  MethodFinal d = new MethodFinal();  d.test();  }    }  class Test{  **protected void test(){**  System.out.println("parent");  }  } |

|  |
| --- |
|  |
|  |

### interface methods

For single or multi-level inheritance, a->b or a->b->c

for example, child interface extends another interface, child default methods could override parent default methods. The rule follows that which is applied in inheritance between virtual class.

|  |
| --- |
| output: interface |
| public class InterfaceTest implements MyInterface{  public static void main(String[] args){  InterfaceTest c = new InterfaceTest();  System.out.println(c.getName());  }  }  interface MyInterface extends Test{  **default String getName() { //override parent default method**  return "inteface";  }  }  interface Test{  **default String getName() {**  return "test";  }  } |
| output: main |
| public class InterfaceTest implements MyInterface{  **public String getName(){**  **return "main";**  **}**  public static void main(String[] args){  InterfaceTest c = new InterfaceTest();  System.out.println(c.getName());  }  }  interface MyInterface extends Test{  **default String getName(){**  **return "inteface";**  **}**  }  interface Test{  **default String getName(){**  **return "test";**  **}**  } |

For branch inheritance, a->b, c (b and c is not inheritance but sibling.). Interface default methods can't be overridden between sibling interface.

but it is ok that abstract method is overridden by another abstract method, which is implemented by a concrete class. OR instance methods of concrete class overrides interface default methods.

|  |
| --- |
|  |
| **public class InterfaceTest implements MyInterface, Test{**  public String getName() {  return "inteface";  }  public static void main(String[] args){  InterfaceTest c = new InterfaceTest();  System.out.println(c.getName());  }  }  interface MyInterface{  **String getName();**  }  interface Test{  **String getName();**  } |
| public class InterfaceTest implements MyInterface, Test{  **public String getName() {**  **return "inteface";**  **}**  public static void main(String[] args){  InterfaceTest c = new InterfaceTest();  System.out.println(c.getName());  }  }  interface MyInterface{  **default String getName() {**  **return "test";**  **}**  }  interface Test{  **String getName();**  } |

|  |
| --- |
|  |
|  |

## hide

**Hidden methods** are share the same signature but are only replaced in the subclasses for which they are defined.

Private instance methods, final methods, static methods and all class variable can't be overridden but only be hidden because there is no inheritance for those cases.

### private methods

Any private methods can't be overridden but be hidden. If there is inheritance, private methods in superclass is invisible to subclass. So there is no overridden. Therefore, overridden rules should not be applied in such cases.

|  |
| --- |
| output: child  There is no overridden. Private method test() in Class Test is invisible to the subclass. Method test() in class MethodFinal could be default/protected/public. |
| public class MethodFinal extends Test{  **void test(){**  System.out.println("child");  }  public static void main(String[] args){  new MethodFinal().test();  }  }  class Test{  **private void test(){**  System.out.println("parent");  }  } |

|  |
| --- |
|  |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### final methods

Any final methods can't be overridden but be hidden. If there is inheritance,

|  |
| --- |
| output: child |
| public class MethodFinal extends Test{  void test(){  System.out.println("child");  }  public static void main(String[] args){  Test d = new MethodFinal();  d.test();  }    }  class Test{  final void test(){  System.out.println("parent");  }  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

## Mixin

Mixin also known as Injection is to implement another class within one class without inheritance. Mixin denote **softer connection** between two classes than inheritance. Mixin makes code reusable.

### same package

For same package,

imported class should not be marked as pulic.

import statement is not required.

package declaration is not required.

the same file.

|  |
| --- |
| output: animal |
| public class Injection1 {  void test(){  **new Animal().go();**  }  public static void main(String[] args) {  Injection1 t = new Injection1();  t.test();  }  }  class Animal {  void go(){  System.out.println("animal");  }  } |

different file but in the same directory: don't need to use import statements

|  |
| --- |
|  |
| **//package src/Injection2.java**  public class Injection2 {  void test(){  new Animal().go();  }  public static void main(String[] args) {  Injection2 t = new Injection2();  t.test();  }  } |
| **//package src/Animal.java**  class Animal {  void go(){  System.out.println("animal");  }  } |

The next example, what happens if two classes are defined in different packages?

path:

F:\Java\src\Injection4.java

F:\Java\learn2\Animal.java.

compiling:

>cd F:\Java

>javac src\Injection4.java learn2\Animal.java

execution:

>java src\Injection4.java

|  |
| --- |
|  |
| **package src;**  **import learn2.Animal;**  public class Injection4 {  public static void main(String[] args){  new Animal().go();  }  } |
| **package learn2;**  public class Animal {  public void go(){  System.out.println("java.animal");  }  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### keyword: import

import class if they are in different package.

* format: import <package name>.<class name>;
* package name must be declared on the first statement.
* both class and methods should be marked as public.

|  |
| --- |
| ouput: learn.animal  Here are the path: src/Injection3.java and src/learn/Animal.java.  after compiling, src/Injection3.class and src/learn/Animal.class are created. |
| **import learn.Animal;**  public class Injection3 {  public static void main(String[] args){  new Animal().go();  }  } |
| **package learn;**  public class Animal {  public void go(){  System.out.println("learn.animal");  }  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### naming conflicts

For example:

path:

\src\Injection5.java

\src\learn\Animal.java

\src\learn2\Animal.java

|  |
| --- |
| package learn;  public class Animal {  public void go(){  System.out.println("learn.animal");  }  } |
| package learn2;  public class Animal {  public void go(){  System.out.println("learn2.animal");  }  } |

Here are three approaches when both of them are referenced in a class.

Note: Java doesn't support class alias as in Python.

|  |
| --- |
| **//method 1: exact import one and import others using wild card**  **import learn.Animal;**  **import learn2.Animal.\*;**  public class Injection5 {  public static void main(String[] args){  **new Animal().go();**  **new learn2.Animal().go();**  }  } |
| **//method 2: remove import statements. specify path when calling**  public class Injection5 {  public static void main(String[] args){  **new learn.Animal().go();**  **new learn2.Animal().go();**  }  } |
| **//methods 3: extends class**  **import learn.Animal;**  **class Animal2 extends learn2.Animal{}**  public class Injection5 {  public static void main(String[] args){  new Animal().go();  new Animal2().go();  }  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

# OOP: Polymorphism

Polymorphism is the ability of an object to take on variant forms.

Advantages:

* A reference to super class is used to refer to the instance of one subclass and invoke instance methods if those methods are overridden by parent methods.
* polymorphism is based on objects. The feature may help to decrease occurrence of the instance of a class. One parent reference could refer to parent instance or child instance.
* polymorphism enables various method passing through objects because some behaviors are determined by runtime rather than compiling. That results in flexible and expandable coding work. If various methods are encapsulated into class, the instance bound with those methods could be invoked according to real context.
* polymorphism is dynamic binding. polymorphism occurs in runtime.
* Decrease work load of developers. In agile project, the features of polymorphism enable developers to focus on base class at early stage. with the decreasing codes, developers could work on certain derived class without updating base class.

Weakness:

* polymorphism would work only if there is inheritance and instance methods are overridden.
* casting objects can't invoke instance methods which are specific in subclass. The reference must have to be casted explicitly.

## applications

### casting objects

Case I: A reference to superclass could be automatically converted to a reference to a superclass without an explicit cast if the reference refers to the instance of its subclass.

|  |
| --- |
| output:  super  sub  super  Therefore, reference a refers to the instance of class A. The object of B() is ready for garbage collection. |
| public class Test{  public static void main(String[] args) {  **A a = new B();**  **a = new A(); //implicit casting**  }  }  class A{  A(){ System.out.println("super"); }  }  class B extends A{  B(){ System.out.println("sub"); }  } |

Care II: A reference to a super class should be explicitly casted as a reference to its subclass if this reference refer to the instance of the subclass. By explicitly casting an object to a subclass, you can gain access to methods and variables that were hidden from access.

|  |
| --- |
| output:  super  sub  Therefore, two references a and b, refer to the instance of B() |
| public class Test{  public static void main(String[] args) {  **A a = new B();**  **B b = (B) a; // explicit casting**  }  }  class A{  A(){ System.out.println("super"); }  }  class B extends A{  B(){ System.out.println("sub"); }  } |

The two examples below shows how reference variables vary if a certain object is created, or objects vary if a certain reference variable is determined.

Let's say: Given class inheritance: C->B->A

"A a = new A();", "A a = new B();", "A a = new C();", or "Object o = new A();" is allowed.

"B b = new B();", "B b = new C();", or "Object o = new B();" is allowed.

"C c = new C();", or "Object o = new C();" is allowed.

"A a = null;", "B b = null;", or "C c = null;" is also allowed

Given class inheritance: B->A. Here are common instantiation.

A a = new A(); B b = new B();

It is ok to cast objects:

A a = new B(); A a1 = (A) a;

That is wrong:

B b = new A(); child reference can't refer to parent object

B b = (B) a; parent reference can’t be casted to child reference

|  |
| --- |
| the reference variable tadpole could be Amphibian, CanSwim, or Object type.  The object is Tadpole type. Class Tadpole extends Amphibian, and implements CanSwim. Therefore, Tadpole type object could be referred to reference variables with Tadpole, Amphibian, or CanSwim or Object type reference variables. |
| import java.util.ArrayList;  import java.util.List;  interface CanSwim {}  class Amphibian implements CanSwim {}  class Tadpole extends Amphibian {}  public class ClassFindAll{  public static void main(String[] args) {  List<Tadpole> tadpoles = new ArrayList<Tadpole>();  for(Amphibian amphibian : tadpoles) {  **Amphibian tadpole = amphibian; //this class**  **CanSwim tadpole = amphibian; // interface extended by this class**  **Object tadpole = amphibian; // object**  }  }  } |
| In this case, the reference variable tadpole could be Tadpole(this class), Amphibian(superclass), CanSwim(interface), or Object type. |
| import java.util.ArrayList;  import java.util.List;  interface CanSwim {}  class Amphibian implements CanSwim {}  class Tadpole extends Amphibian {}  public class ClassFindAll{  public static void main(String[] args) {  List<Tadpole> tadpoles = new ArrayList<Tadpole>();  for(Tadpole amphibian : tadpoles) {  **Tadpole t1 = amphibian;**  **Amphibian t2 = amphibian;**  **CanSwim t3 = amphibian;**  **Object t4 = amphibian;**  }  }  } |

|  |
| --- |
| output:  test  casting |
| public class Casting2 extends Test{  int age = 10;  Casting2(){  System.out.println("casting");  }  public static void main(String[] args){  **Test c = new Casting2();**  }  }  class Test{  int age = 1;  Test(){  System.out.println("test");  }  } |

### Inheritance: invoke instance methods in subclass

Parent reference could point to child object and call child methods. In this case, the child methods must have the same signatures with parent methods.

Note:

* The method name, return type, and arguments are identical, and access identifiers are compatible.
* Child reference can't point to parent object.

|  |
| --- |
| output:  1  20  Note: int getAge() must co-exist in parent and child class, or compiling cause error. |
| public class ClassMethodCast extends Test{  int age = 10;  **int getAge(){**  return this.age\*2;  }  public static void main(String[] args){  **Test c = new Test();**  System.out.println(c.getAge());  **c = new ClassMethodCast();**  System.out.println(c.getAge());  }  }  class Test{  int age = 1;  **int getAge(){**  return this.age;  }  } |

Let's say: Y->X.

Compare the difference of the codes below.

|  |  |
| --- | --- |
| output |  |
|  |  |
| YYY  Note: method X.do1() is overridden by Y.do1() | public class CastingMethod {  public static void main(String[] args) {  **X x = new Y();**  **x.do1();**  **((X)x).do1(); // the same as the above**  **((Y)x).do1();**  }  }  class X{  void do1(){ System.out.print("X"); }  }  class Y extends X{  void do1(){ System.out.print("Y"); }  } |
| XXY  There is no overridden but inheritance. | public class CastingMethod {  public static void main(String[] args) {  X x = new Y();  x.do1(); //inherit  ((Y)x).do1();  ((Y)x).do2();  }  }  class X{  void do1(){ System.out.print("X"); }  }  class Y extends X{  void do2(){ System.out.print("Y"); }  } |
| error: cannot find symbol  No inheritance and No overriding, so the approach of invoking is not working. | public class CastingMethod {  public static void main(String[] args) {  **X x = new Y();**  x.do1();  }  }  class X{}  class Y extends X{  void do1(){ System.out.print("Y"); }  } |

### Mixins: pass reference as arguments

Object is passed by reference in methods. Therefore, a certain type of reference could be casted into **this type or all other child type object**.

Note: reference variable could refer to null.

Here is an example. There is inheritance: Cobra -> Snake.

|  |
| --- |
| The below three bold statments are allowed. |
| class Snake {}  class Cobra extends Snake {}  class GardenSnake {}  public class Casting2 {  private Snake snake;  **public void setSnake(Snake snake) {**  this.snake = snake;  }  public static void main(String[] args) {  **new Casting2().setSnake(new Snake());**  **new Casting2().setSnake(new Cobra());**  **new Casting2().setSnake(null);**  }  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

**Additionally, polymorphism is only for methods, not for variables.**

|  |
| --- |
| output:  furry bray |
| public class CastingVariable {  public static void main(String[] args){  new CastingVariable().go();  }  void go() {  Mammal m = new Zebra();  **System.out.print(m.name); //parent variable**  System.out.println(m.noise()); //child method  }  }  class Mammal{  String name = "furry ";  String noise(){  return "generic";  }  }  class Zebra extends Mammal{  String name = "stripse ";  String noise(){  return "bray";  }  } |

### invoke interface default methods

Interface can't be instantiated like virtual class. Default methods would acts like instance methods. but concrete class could inherit default methods defined in interface. Therefore, casting could invoke default methods.

interface type reference variable refers to the object of concrete class.

|  |
| --- |
| output: interface  "Test t = (Test) new InterfaceTest();" explicit casting is allowed, too. |
| public class InterfaceTest implements Test{  public static void main(String[] args){  **Test t = new InterfaceTest();**  t.getName();  }  }  interface Test{  **default void getName(){**  System.out.println("interface");  }  } |

However, interface default methods might be overridden by instance methods in concrete class. If it is, the instance methods must be marked as public.

|  |
| --- |
| output:  class  class  default method is overridden by virtual method defined in concrete class |
| public class InterfaceTest implements Test{  **public void getName(){**  System.out.println("class");  }  public static void main(String[] args){  **Test t = new InterfaceTest();**  **t.getName();**  **new InterfaceTest().getName();**  }  }  interface Test{  **default void getName(){**  System.out.println("interface");  }  } |

|  |
| --- |
|  |
|  |

### invoke non-abstract methods

Abstract class can't be instantiated. reference of insect type points to the object of non-abstract class.

|  |
| --- |
| Insect is abstract class. Note: |
| public class AbstractTest{  public static void main(String[] args){  **Insect c = new Beetle();**  c.getLegs();  }  }  abstract class Insect{  abstract int getLegs();  }  class Beetle extends Insect{  int getLegs(){  System.out.println("Number of legs is ");  return 1;  }  } |

|  |
| --- |
|  |
| public abstract class AbstractCast {  void fly(){  System.out.println("abstract");  }  public static void main(String[] args){  **AbstractCast c = new Concrete();**  c.fly();  }  }  class Concrete extends AbstractCast{  void fly(){  System.out.println("concrete");  }  } |

|  |
| --- |
| the reference variables s and obj refer to the same object. obj2 refers to another object. |
| String[] s = {"good"};  **Object[] obj = s; //cast reference variable**  Object[] obj2 = {s};  **String[] s3 = (String[]) obj; // cast reference variable** |

## Data type

Everything in Java is object - "Object". The base class is "java.lang.Object".

There are eight primitive types in Java. Each of them are declared in the package "java.lang.\*". Moreover, those primitive types have corresponding wrapper class. For example, int type is declared in the class "java.lang.int". The wrapper type is "Integer" declared in the class "java.lang.Integer".

### primitive casting

casting convert one type of a primitive type to another type.

implicit casting is ok if primate type is compatible: char/bytes/short->int, char/bytes/short/int->long, char/bytes/short/int/long->float, char/bytes/short/int/long/float->double,

Casting should be explicit if bits of type should be narrowed down.

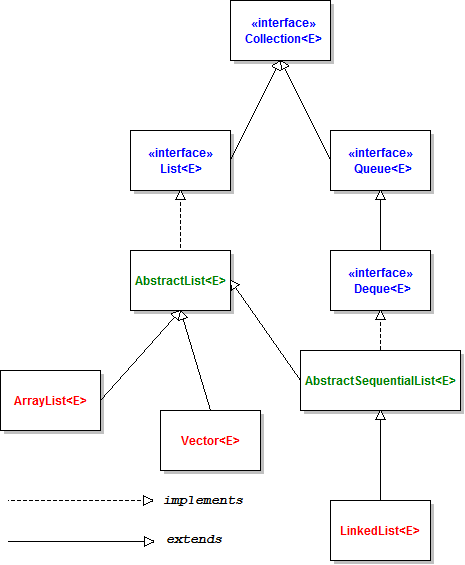
|  |
| --- |
|  |
| int[] nums = new int[] { 1, 4, 6};  Object p = nums; |

|  |
| --- |
| 512 |
| int x = 4;  long y = x \* 4 - x++;  System.out.print(x);  System.out.println(y); |
| 12.05 |
| byte x = 4;  float y = x \* 4 - x++;  System.out.print(y);  System.out.println(x); |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### Collection



List is declared in interface "java.util.List".

Arraylist is declared in class "java.util.ArrayList" which implements "java.util.List".

Theoretically, ArrayList<Object> could contain any type of elements.

|  |
| --- |
|  |
| import java.util.List;  import java.util.ArrayList;  public class Casting3 {  public static void main(String[] args){  **List<Object> list = new ArrayList<>();**  **list.add(new Tiger());**  **list.add(new Animal());**  **list.add(null);**  }  }  class Animal {}  interface Test{}  class Tiger extends Animal {} |
|  |

|  |
| --- |
|  |
| import java.util.List;  import java.util.ArrayList;  public class Casting3 {  public static void main(String[] args){  **List<Whale> list = new ArrayList<>();**  **list.add(new Whale());**  **list.add(new Narwhal());**  **list.add(null);**  }  }  interface HasTail {}  class Mammal {}  class Whale extends Mammal implements HasTail {}  class Narwhal extends Whale {} |
| output:  whale general narwhal specie |
| import java.util.List;  import java.util.ArrayList;  public class Casting3 {  public static void main(String[] args){  **List<Whale> list = new ArrayList<>();**  **list.add(new Whale());**  **list.add(new Narwhal());**  for( Whale t: list){  System.out.print(t.getName() + " ");  System.out.print(t.getType() + " ");  }  }  }  interface HasTail {  String getName();  }  class Mammal {  **String getName(){ return "mammal"; }**  **String getType(){ return "general"; }**  }  class Whale extends Mammal implements HasTail {  **public String getName(){ return "whale"; }**  }  class Narwhal extends Whale {  **public String getName(){ return "narwhal"; }**  **String getType(){ return "specie"; }**  } |

List could be used with interface or abstract class. It is good idea to invoke methods in lists in practice. Let's say, in one project, as described in context, several entities are defined in the system. They have similar behaviors on the same

one interface defines blueprint. Then several concrete class concrete all methods defined in the interface.

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### exception casting

|  |
| --- |
| pass reference variable of class exception or subclass as arguments. |
| public class ExceptionLogger {  **public static void logException(RuntimeException e) {**  System.out.println("Logging " + e);  }  public static void main(String [] args) {  **logException(new NullPointerException());**  **logException(new RuntimeException());**  }  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

## example

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |