OOP with Java

Yuanbin Wu
cs@ecnu

OOP with Java

- Project 7 :6 月 30 日晚 9 点
- 6 月 17 日复习
- 考试时间： 6 月 24 日
- 考试地点：文史楼 203
- 答疑
  - 6 月 21 日 13 ： 00 – 14 ： 30
  - 教书院 230

OOP with Java

- 闭卷
- 题目类型
  - 选择 : 30%
  - 问答 : 30%
  - 编程 : 40%

OOP with Java

- 复习
  - 面向对象编程的要素
  - Java 类型
  - Java 控制结构
  - 类
  - Java 包
  - 访问控制
  - 类的复用
  - **Upcasting 和多态**
  - 抽象类和接口
  - 内部类
  - 异常
  - 容器
  - I/O

语言基础

类

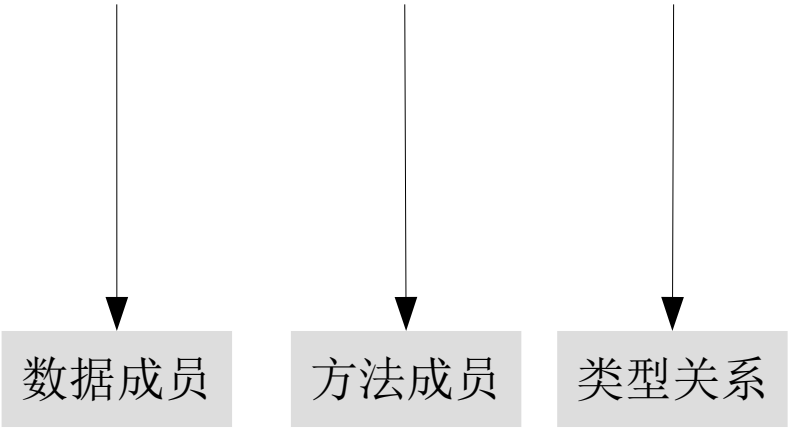
类型间关系

错误处理

应用

面向对象编程概述

对象的基本要素：状态，行为，类型

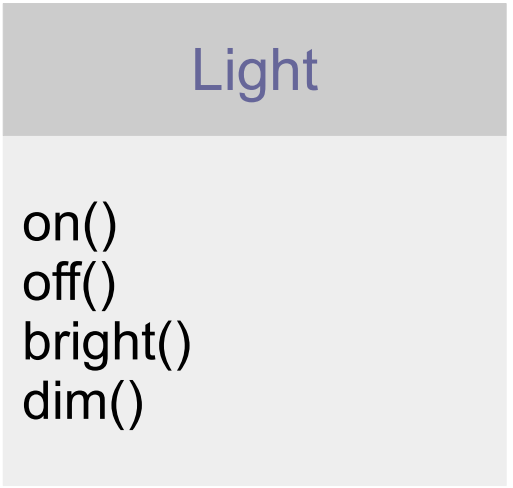


面向对象编程概述

- 对象的状态
  - 数据成员
- 对象的行为
  - 方法
  - 访问控制 (封装): public, private, protected
- 对象的类型
  - 类 与 类的对象
  - 基本类型 与 class

面向对象编程概述

- 对象 vs. 类型
  - 类型



  - 对象

```
Light m = new Light();
m.on();
Light n = new Light();
n.off();
```

面向对象编程概述

- 对象的类型
  - 如何通过已有的类型定义新的类型？
    - 组合 : has-a
    - 继承 : is
  - 不同类型间的关系
    - 基本类型之间的转换关系
    - Autoboxing, unboxing
    - Upcasting, downcasting

Java 类型

Java 类型：基本类型，数组，类

定义，创建，使用

Java 类型

- 基本类型
  - boolean, char, byte, short, int, long, float, double
- 基本类型的封装
  - Boolean, Character, Byte, Short, Integer, Long, Float, Double
- 定义
- 创建
  - int a = 1, double d = 1.0;
- 使用
  - int b = a * 2;

Java 类型

- class
  - 定义

```
class MyType {

   int i;
   double d;
   char c;

   void set(double x);
   double get();

}
```

创建

```
MyType a = new MyType();
```

使用

```
int b = a.i;
a.set();
a.get();
```

Java 类型

- 数组
  - 定义

```
int a[ ];
int [ ]a;

MyType [ ]m;
MyType [ ][ ]m;
```

初始化

```
// 静态初始化
int [ ]a = {1, 2, 3, 4, 5};
int [ ][ ] a= { {1,2,3}, {4, 5, 6} };

// 动态初始化 1
int [ ]a = new int[5];

// 动态初始化 2
MyType [ ]a = new MyType[] {
    new MyType(),
    new MyType(),
    new MyType()
};
```

使用

```
int b = a[1];
int len = a.length;
```

Java 类型

- 引用
  - 对象的名字
  - 同一个对象可以有不同的名字
  - 绝大多数使用指针实现

题目：

```
int [ ] a = new int[]{1,2,3};
int [ ] b;
b = a;
b[0] = 4;
System.out.println(a[0]);
```

Java 类型

- 不可变类型
  - 一旦创建不能改变
  - String, Integer,…
- final 关键字
  - final 数据，方法，参数，类
  - static final
  - final 数据初始化

Java 控制结构

- 操作符
  - “==” 与 equals()
  - 字符串连接操作 : “+=”
- 控制语句
  - 条件 与 boolean
  - for each

类

- 普通成员
- 静态成员
  - 静态数据
    - 所有对象共享数据
  - 静态方法
    - main 函数

```
public class StaticTest {
        double d;
        static int i = 1;
        static void display() {
                System.out.println(“Hello”);
        }

        public static void main(String [ ]args) {
                        display();
                StaticTest.display();
                StaticTest s = new StaticTest();
                System.out.println(s.i);
                System.out.println(StaticTest.i)
    }
}
```

类

- 类的数据成员
  - this 关键字
    - 在类的非静态方法中，返回调用该方法的对象的引用
  - super 关键字
    - 在子类对象中包含的父类对象的引用
  - 数据成员的初始化
    - 初始化的值
    - 初始化的顺序
      - 静态成员 / 非静态成员
      - 子类成员 / 父类成员
      - 构造函数

类

- 类的方法
  - 构造函数
    - 名称与类名称相同，无返回值，为类的静态方法
    - 默认构造函数

```
public class MyType {
    int i;
    double d;
    char c;
    void set(double x) { d = x; }
    double get() { return d; }
    public static void main(String [ ]args) {
        MyType m = new MyType();
        MyType n = new MyType();
        m.set(1);
        n.set(2);
    }
}
```

类

- 类的方法
  - 重载
    - 方法名相同，参数类型 / 数量不同
    - 如何区分不同的函数？ 函数名 + 参数列表
  - 重写
    - 子类重写父类的方法

类

- 类的销毁
  - 垃圾回收

    1. 仅回收 new 创建的内存 .

    2. 是否回收，何时回收由 Java 虚拟机控制 .

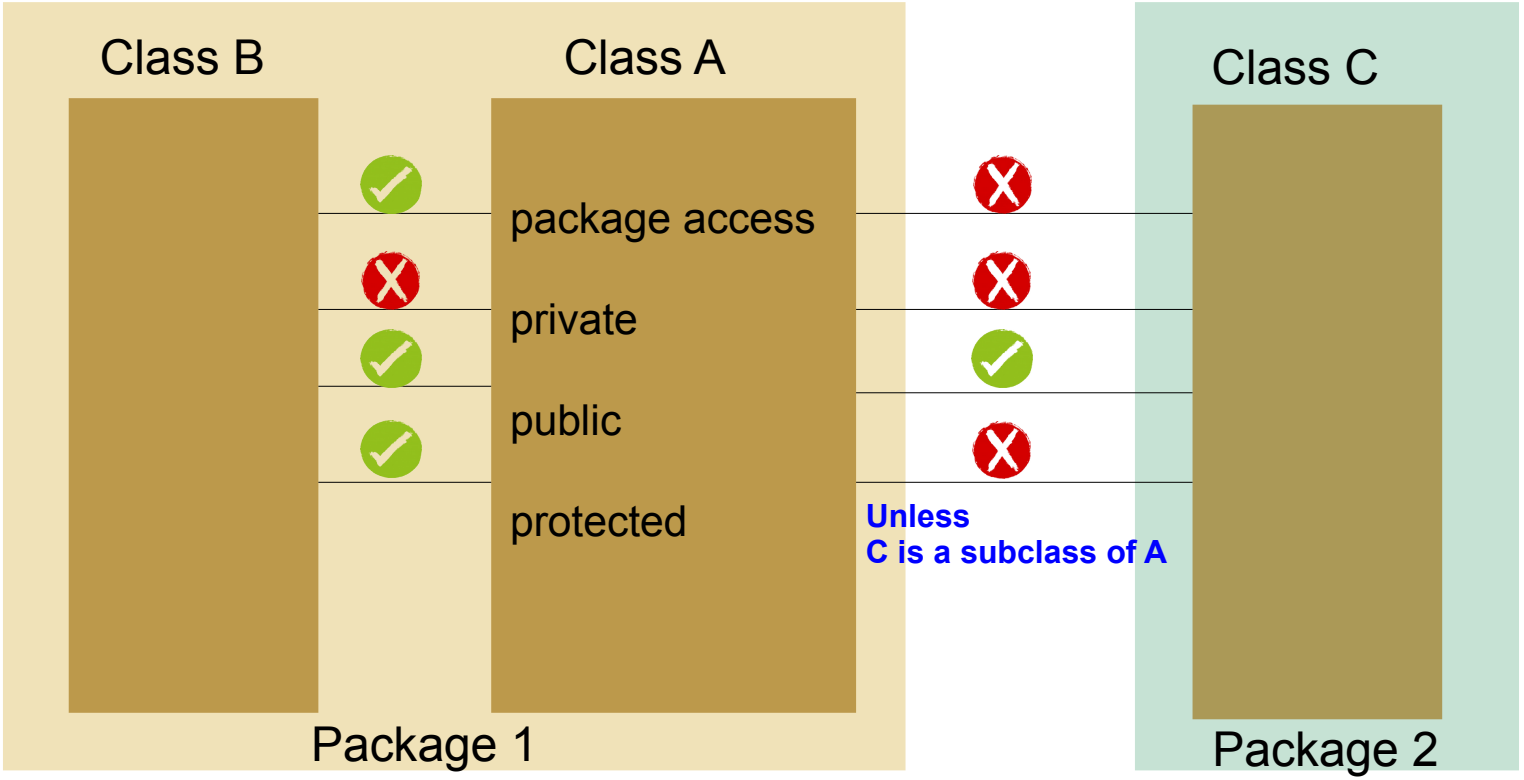
Java 包

- 什么是 Java 包
  - 一组类的集合，共享一个名字空间
- 如何使用包
  - import 语句
- 如何定义包
  - package 语句
  - 包结构与目录结构
- 如何编译包

Java 访问控制

- 四种访问控制类型
  - package access
    - 默认包
  - public
  - private
  - protected

Java 访问控制



Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

类的复用

- 如何通过已有类定义新的类？
  - 组合
    - 将已有的类作为新类的数据成员
  - 继承
    - 新类包含已有类的方法和数据
    - 并可修改 / 增添 （重写）
    - 如何实现？ 隐含一个父类对象

```
class MyType {
    public int i;
    public double d;
    public char c;
    public void set(double x) { d = x;}
    public double get() { return d; }
}
```

```
public class MyCompType {
    private MyType m = new MyType();
    private String s;
    public MyCompType(){
        s = new String("Hello");
    }
}
```

```
public class MySubType extends MyType{
    public static void main(String [ ]args){
        MySubType ms = new MySubType();
        ms.set(1.0);
        System.out.println(ms.get());
        System.out.println(ms.i);
    }
}
```

类的复用

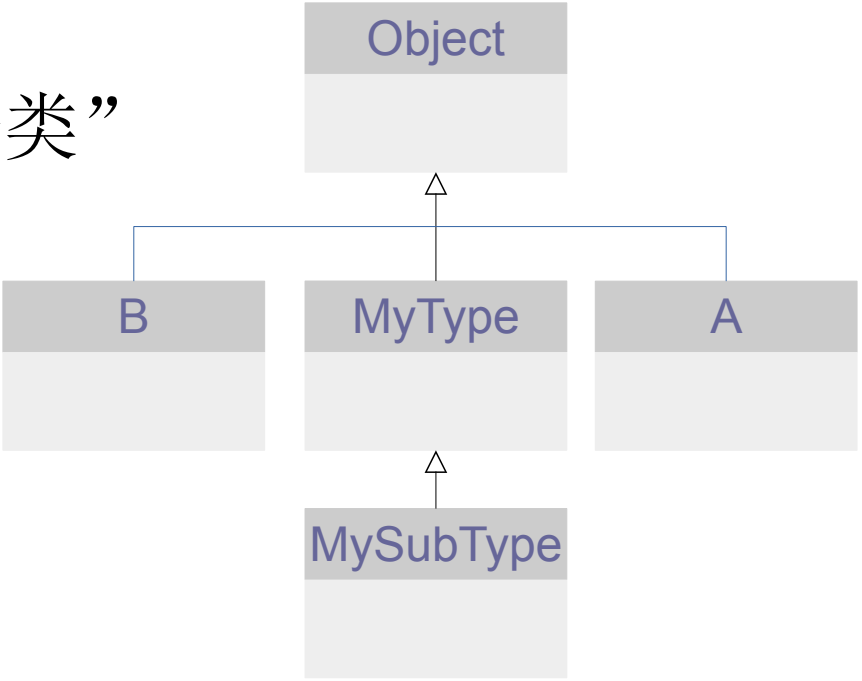
- 子类可以定义新的方法 / 数据
- protected 关键字
- 子类可以修改父类的方法： 重写

```
class MyType {
    public int i;
    public double d;
    public char c;
    public void set(double x) { d = x;}
    public double get() { return d; }
}
```

```
public class MySubType extends MyType{
    public void set(double x){ i = (int)x; }
    public double get() { return i; }
    public static void main(String [ ]args){
        MySubType ms = new MySubType();
        ms.set(1.0);
        System.out.println(ms.get());
        System.out.println(ms.i);
        System.out.println(ms.d);
    }
}
```

类的复用

- super 关键字
  - 子类的对象包含一个隐藏的父类对象
  - 在子类中，super 用来指代父类对象的引用
  - 子类，父类的构造函数调用顺序
- Object 类
  - “所有类都是 Object 类的子类”



Upcasting 和多态

- Upcasting
  - 子类是一种父类
  - 父类出现之处子类也可使用

```
class A{ … }
class B{ … }
A a = new A();
B b = new B();

// A a = new B(); compile error
```

```
class A{ … }
class B extends A{ … }
A a = new A();
B b = new B();

A a = new B(); // upcasting
```

Upcasting 和多态

```
class Instrument {
    public void play() {}
    static void tune(Instrument i) {
        // ...
        i.play();
    }
}
public class Wind extends Instrument {
    public static void main(String[] args) {
    Wind flute = new Wind();
    Instrument.tune(flute);
    }
}
```

Upcasting 和多态

- 多态
  - 子类重写了父类方法 f()
  - 当使用父类引用访问子类对象时，调用 f() 将绑定到子类的方法

Upcasting 和多态

```
class Instrument {
    public void play(int note) {
        System.out.println(“Instrument.play()” + n);
    }
}
```

```
public class Wind extends Instrument {
    public void play(int note) {
        System.out.println(“Wind.play()” + n);
    }
}
public class Stringed extends Instrument {
    public void play(int note) {
        System.out.println(“Stringed.play()” + n);
    }
}
public class Brass extends Instrument {
    public void play(int note) {
        System.out.println(“Brass.play()” + n);
    }
}
```

```
public class Music {
    public static void tune(Instrument i) {
        i.play();
    }
    public static void main(String []args){
        Wind flute = new Wind();
        Stringed violin = new Stringed();
        Brass frenchHorn = new Brass();
        tune(flute);
        tune(violin);
        tune(frenchHorn);
    }
}
```

Upcasting 与多态

```
interface DifferentiableFunction {
    double eval(double x);
    double diff(double x);
}
```

```
class Linear implements DifferentiableFunction {
    public double eval(double x) { //… }
    public double diff(double x) { //… }
}
```

```
class Sin implements DifferentiableFunction {
    public double eval(double x) { //… }
    public double diff(double x) { //… }
}
```

```
class Quadratic implements DifferentiableFunction
{
    public double eval(double x) { //… }
    public double diff(double x) { //… }
}
```

```
public class NewtonRoot {
    public static double findRoot(DifferentiableFunction f)
{
        // newton method...
    }
    public static void main(String []args){
        Linear l = new Linear();
        Sin s = new Sin();
        Quatratic q = new Quatratic();
        findRoot(l);
        findRoot(s);
        findRoot(q);
    }
}
```

Upcasting 与多态

- 例子 :
  - ArrayList a = new ArrayList(); a.add(“Hello”);
  - catch(Exception e)

接口与抽象类

- 抽象方法 (abstract method)
  - 仅提供方法的名称，参数和返回值 没有具体实现
  - 使用 abstract 关键字
- 抽象类 (abstract class)
  - 包含抽象方法的类称为抽象类
  - 不完整，不能用 new 实例化 (创建对象)
  - 子类中可以通过 super 实例化

```
class Instrument {
    public void play(int note) {
        System.out.println(“Instrument.play()” + n);
    }
}
```

普通方法

```
abstract class Instrument {
    public abstract void play(int note) ;
}
```

抽象方法

接口与抽象类

```
abstract class Instrument {
    public abstract void play(int note) ;
}
```

```
public class Wind extends Instrument {
    public void play(int note) {
        System.out.println(“Wind.play()” + n);
    }
}
public class Stringed extends Instrument {
    public void play(int note) {
        System.out.println(“Stringed.play()” + n);
    }
}
public class Brass extends Instrument {
    public void play(int note) {
        System.out.println(“Brass.play()” + n);
    }
}
```

```
public class Music {
    public static void tune(Instrument i) {
        i.play();
    }
    public static void main(String []args){
        Wind flute = new Wind();
        Stringed violin = new Stringed();
        Brass frenchHorn = new Brass();
        tune(flute);
        tune(violin);
        tune(frenchHorn);
    }
}
```

Upcasting 与多态

接口与抽象类

- 接口
  - 只有方法的名称，参数和返回值 没有方法的实现
  - interface/implement 关键字

接口与抽象类

```
abstract class Instrument {
    public abstract void play(int note);
    public abstract String what();
}
```

```
class Stringed extends Instrument {
    public void play(int note) {
        System.out.println(“Stringed.play()” + n);
    }
   public String what() {return “Stringed”;}
}
```

继承：
1. extends 关键字
2. 父类，子类关系
3. class, extends

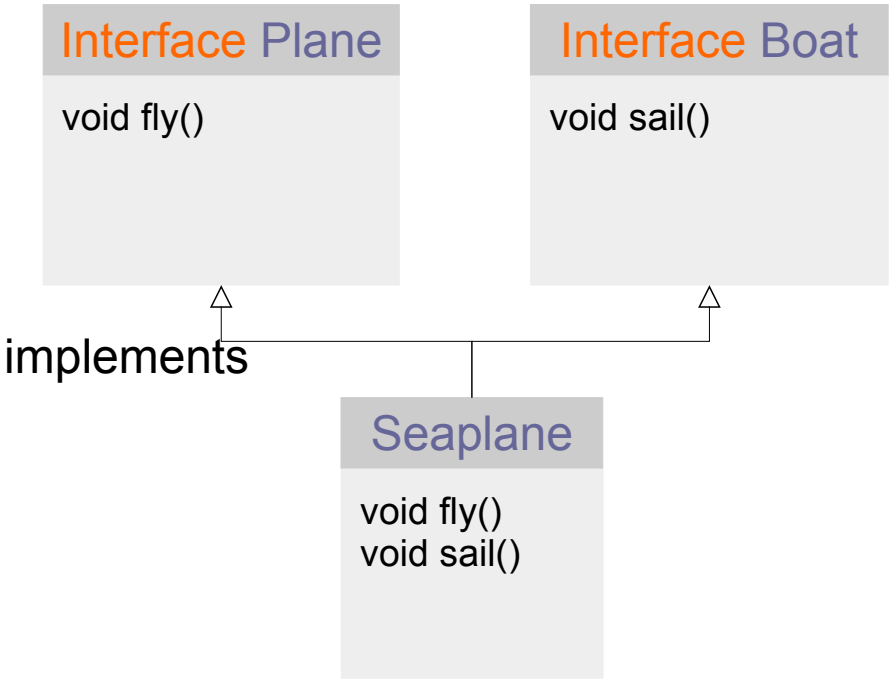
```
interface Instrument {
    void play(int note) ;
    String what();
}
```

```
class Stringed implements Instrument {
    public void play(int note) {
        System.out.println(“Stringed.play()” + n);
    }
   public String what() {return “Stringed”;}
}
```

接口：
1. implements 关键字
2. 接口，实现关系
3. interface, implements

接口与抽象类

- 一个类实现多个接口



接口与抽象类

```
interface Plane {
    void fly();
}
interface Boat {
    void sail();
}

class Seaplane implements Plane, Boat {
    public void fly(){
        System.out.println(“Fly!”);
    }
    public void sail(){
        System.out.println(“Sail!”);
    }
}
```

接口与抽象类

- Upcasting 与多态

```
interface CanFight {
    void fight();
}
```

```
interface CanSwim {
    void swim();
}
```

```
interface CanFly {
    void fly();
}
```

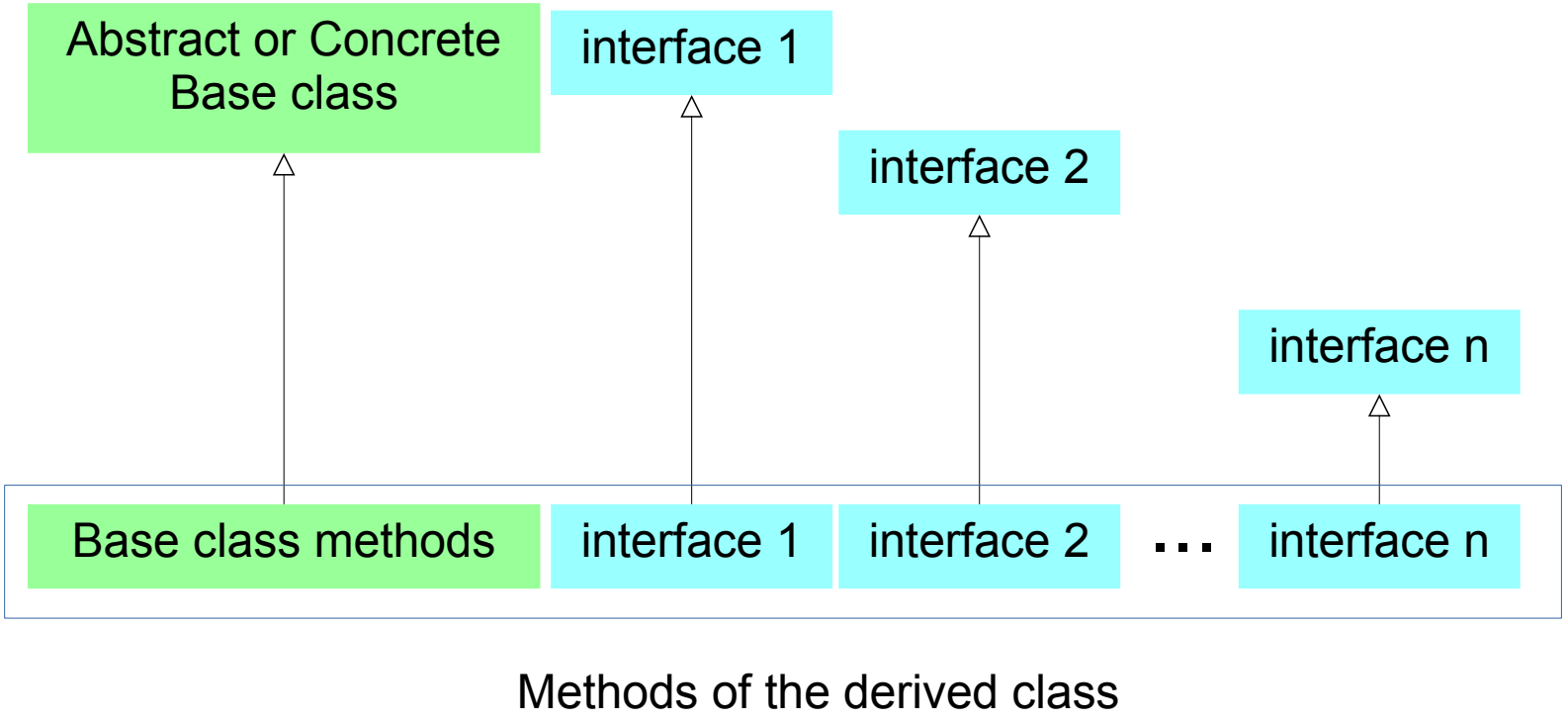
```
class ActionCharacter {
    public void fight() { }
}
```

```
class Hero extends ActionCharacter
        Implements CanFight, CanSwim, CanFly{
    public void fly() { }
    public void swim() { }
}
```

```
public class Adventure {
    public static void t(CanFight x) { x.fight();}
    public static void u(CanSwim x) { x.swim();}
    public static void v(CanFly x) { x.fly();}
    public static void w(ActionCharacter x) { x.fight();}
    public static void main(String []args) {
        Hero h = new Hero();
        t(h);   u(h);   v(h);   w(h);
    }
}
```

接口与抽象类

- 实现多个接口
  - 父类只能有一个普通类 / 抽象类



接口与抽象类

- 扩展接口

```
interface A {
    …
}
interface B extends A{
    …
}

interface D {
    …
}
interface D extends A, C{
    …
}
```

内部类

- 内部类 (Inner class)
  - 定义在一个类的内部
  - 与组合不同
  - 帮助隐藏实现细节

Inner class

```
class Outer{
    …
    class Inner{
        ...
    }
    ...
}
```

Composition

```
class Outer{
    …
    Inner in = new Inner();
    ...
}
class Inner{
    ….
}
```

```
public class Sequence{
    private Object[] items;
    private int next = 0;
    public Sequence (int size) {items = new Object[size];}
    public void add(Object x){
        if (next < items.length)
            items[next++] = x;
    }
    private class SequenceSelector implements Selector{
        private int i = 0;
        public boolean end() {return i == items.length;}
        public Object current () {return items[i];}
        public void next() { if(i < items.length) i++; }
    }
    public Selector selector(){
        return new SequenceSelector(s);
    }
    public static void main(String []args){
        Sequence seq = new Sequence(10);
        for (int i = 0; i < 10; ++i)
            seq.add(Integer.toString(i));
        Selector s = seq.selector();
        while(!s.end()) {
            System.out.println(s.current() + " ");
            s.next();
        }
    }
}
```

```
interface Selector{
    boolean end();
    Object current();
    void next();
}
```

1. Sequence 类包含内部类 SequenceSelector
2. 内部类实现接口 Selector
3. 内部类能访问 Sequence 的 private 成员
4. 内部类为 private
5. 内部类的对象隐藏包含一个外部类对象的引用
   - 多数情况下由编译器自动完成

upcasting: Object / selector()

内部类

- 内部类和外部类的关系
  - 内部类的对象隐含了一个引用，指向包含它的外部类对象
  - 如何在内部类中访问外部类对象的引用？
    - OuterClassName.this
  - 如何创建内部类的对象
    - 在外部类的方法中： 直接创建
    - 其他地方 : OuterClassObject.new

匿名类

- 匿名内部类 (匿名类)
  - 没有名字的内部类，没有构造函数，同时定义和创建
  - 必须继承某个类，或实现某个接口

```
public class Parcel{

    public Contents contents(){
        return new Contents() {
            // anonymous inner class definition
            private int i = 11;
            public int value() {return i;}
        };
    }

    public static void main(String []args){
        Parcel p = new Parcel();
        Contents c = p.contents();
    }
}
```

```
public interface Contents{
    int value();
}
```

“ 创建一个实现 Contents 的匿名类”

语法解释
1. “;” 为 return 语句的分号
2. 在 return 语句中定义匿名类
   - 实现 Contents 接口
   - 花括号内部
3. 创建一个改匿名类的对象
   - new Content () { }

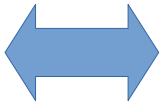
匿名类

- 匿名类

```
public class Parcel{

    public Contents contents(){
        return new Contents() {
            // anonymous inner class definition
            private int i = 11;
            public int value() {return i;}
        };
    }

    public static void main(String []args){
        Parcel p = new Parcel();
        Contents c = p.contents();
    }
}
```

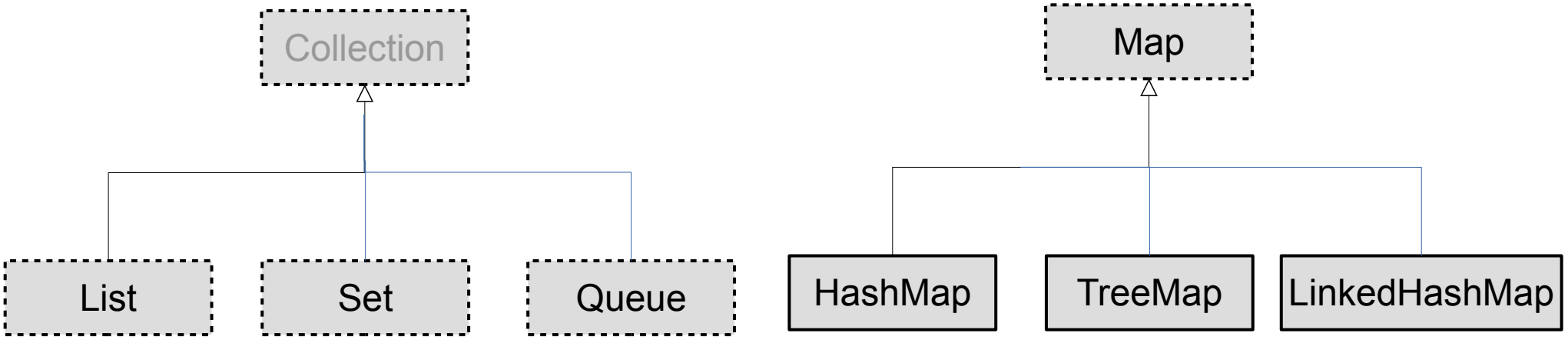


```
public class Parcel{
     class PContents implements Contents{
        private int i = 11;
        public int value() {return i;}
    }

    public Contents contents(){
        return new PContents() ;
    }

    public static void main(String []args){
        Parcel p = new Parcel();
        Contents c = p.contents();
    }
}
```

容器


Collection
List
Set
Queue
Map
HashMap
TreeMap
LinkedHashMap

容器

- 容器与泛型

```
public class ApplesAndOrangesWithGenerics {
public static void main(String[] args) {
    ArrayList<Apple> apples = new ArrayList<Apple>();
    for(int i = 0; i < 3; i++)
        apples.add(new Apple());

    // Compile error!
    // apples.add(new Orange());
    for(int i = 0; i < apples.size(); i++)
        apples.get(i).id();
    }
    for(Apple c: apples)
        System.out.println(c.id());
}
```

容器

- upcasting 和多态

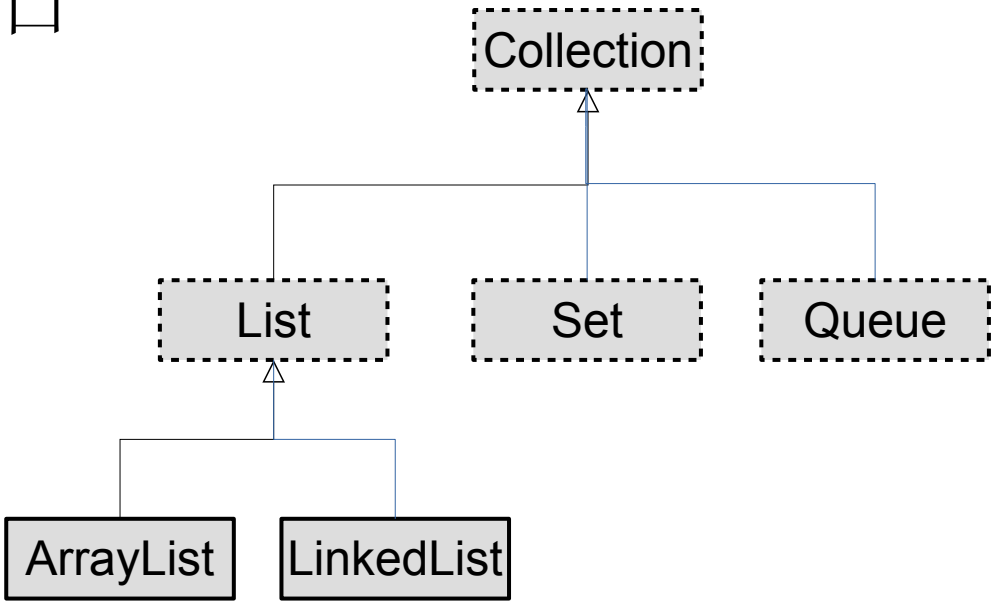
```
class GrannySmith extends Apple {}
class Gala extends Apple {}
class Fuji extends Apple {}
class Braeburn extends Apple {}
```

```
public class GenericsAndUpcasting {
    public static void main(String[] args) {
        ArrayList<Apple> apples = new ArrayList<Apple>();

        apples.add(new GrannySmith());
        apples.add(new Gala());
        apples.add(new Fuji());
        apples.add(new Braeburn());
        for(Apple c : apples)
            System.out.println(c);
    }
}
```

容器

- List
  - 实现原理
  - 接口 : add(), remove(), get()…
  - 迭代器
  - LinkedList 与 Queue 接口



容器

- Set, Queue
  - 基本接口及使用
  - HashSet
- Map
  - 基本接口及使用
  - HashMap

异常

- 抛出异常
  - throw
- 处理异常
  - try, catch, finally
- 异常对象
  - Exception 类的子类

异常

- Upcasting 与多态

```
class SimpleException extends Exception { }

public class InheritingExceptions {

    public static void main(String[] args) {
        try {
            System.out.println("Throw SimpleException from f()");
            throw new SimpleException();
        } catch(Exception e) {
            System.out.println("Caught it!");
            System.out.println(e);
            System.out.println(e.printStackTrace(System.out));
        }
    }
}
```

异常

- 从方法中抛出异常
  - 方法的异常说明 :throws
  - 中断当前方法的执行，返回抛出的异常对象，在该方法的调用路径上寻找合适的 catch.

异常

```
bar() throws Type1Exception, Type2Exception{
    …
    throw new Type1Exception ();
    …
    throw new Type2Exception ();
}
```

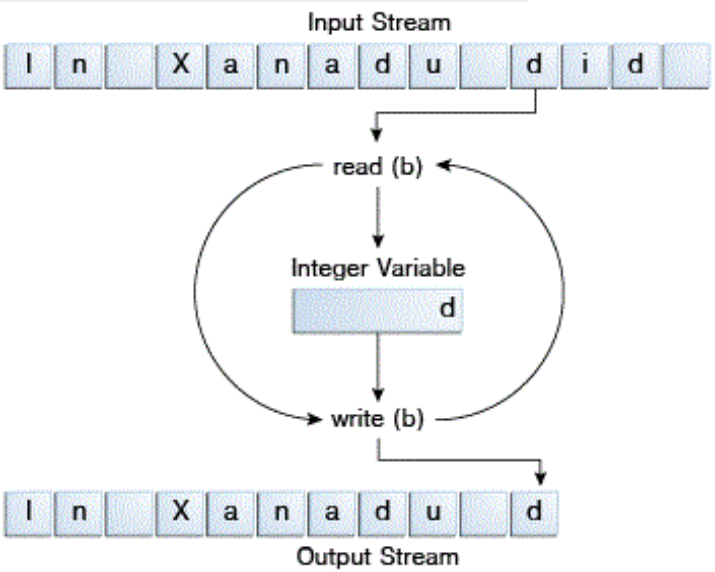
```
foo() {
    try{
        …
        bar();
    }
    catch (Type1Exception e){
        ...
    }
    catch (Type2Exception e){
        ...
    }
}
```

I/O

- I/O 流
  - InputStream/Reader
    - read()
  - OutputStream/Writer
    - write()
  - 抽象：数据的来源 / 数据的目的地
    - ByteArrayStream, FileStream...

I/O

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class CopyBytes {
    public static void main(String[] args) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;
        try {
            in = new FileInputStream("xanadu.txt");
            out = new FileOutputStream("outagain.txt");
            int c;
            while ((c = in.read()) != -1)
                out.write(c);
        } finally {
            if (in != null)
                in.close();
            if (out != null)
                out.close();
        }
    }
}
```



I/O

- 装饰器
  - FilterInputStream/FilterOutputStream
    - BufferedInputStream
    - DataInputStream

I/O

- Upcasting 和多态

```
FileInputStream fin = new FileInputStream("xanadu.txt");
BufferedInputStream bf = new BufferedInputStream(fin);
DataInputStream din = new DataInputStream(bf);

din.read();  din.readInt();  din.readDouble();
```

```
FileOutputStream fout = new FileOutputStream("xanadu.txt");
BufferedOutputStream bf = new BufferedOutputStream(fout);
DataOutputStream dout = new DataOutputStream(bf);

dout.write(1);  dout.writeInt(10);  dout.writeDouble(3.14);
```

泛型

- 如何定义泛型类，泛型接口

```
class Apple { }

public class Holder<T> {
    private T a;
    public Holder(T a) { this.a = a; }
    public void set(T a) { this.a = a; }
    public T get() { return a; }

    public static void main(String[] args) {
        Holder<Apple> h = new Holder<Apple>(
                new Apple());
        Apple a = h.get();
    }
}
```

```
public interface Generator<T> { T next(); }

public class Fibonacci implements Generator<Integer> {
    private int count = 0;
    public Integer next() { return fib(count++); }
    private int fib(int n) {
        if(n < 2) return 1;
        return fib(n-2) + fib(n-1);
    }
    public static void main(String[] args) {
        Fibonacci gen = new Fibonacci();
        for(int i = 0; i < 18; i++)
            System.out.print(gen.next() + " ");
    }
}
```

考试内容将不包含

- char 的 unicode 表示
- 赋值左值表达式，右值表达式
- 垃圾回收的实现原理
- 嵌套类，工厂模式
- TreeSet, LinkedHashSet, TreeHashMap, LinkedHashMap 的实现原理
- RTTI, type erasure，被限定类型的泛型，通配符