1. 数据类型

基本数据类型

八个基本数据类型，四类八种

- 整数型：byte（字节，1B，默认0）、short（短整，2B，默认0）、int（整型，4B，默认0）、long（长整，8B，默认0L）；
- 浮点型：float（单精度浮点型，4B，默认0.0F）、double（双精度浮点型，8B，默认0.0D）
- 字符型：char（字符型，2B，默认'\u0000'）
- 布尔型：boolean（布尔型，1B，默认false）

数据类型转换

1. 自动类型转换　由小至大自动转换；多种类型运算结果自动转换为大数据类型；byte、short不能与char相互转换，char仅包含非负数；字符串与任何类型结合都转换为字符串
2. 强制类型转换　由大至小强制转换；　byte b = (byte)12;

变量

1. 实例变量

   任何方法、块之外的，不使用`static`关键字修饰的变量都属于实例变量，基于特定实例，不在实例间共享；可使用4种访问修饰符；可使用`transient、final`关键字；不可使用`abstract、synchronized、strictfp、native、static`关键字；实例变量带有默认值，即对应基本数据类型的默认值。

   ```
   class Fruits {
       public String fruitName;
       public static void main(String[] args) {
           Fruits fruits = new Fruits();
           fruits.fruitName = "Banana";     //实例变量调用方式
       }
   }
   ```

2. 静态变量

   属于该类的变量，由`static`关键字修饰，只能定义在类内部、方法外部；会在程序运行前初始化且只初始化一次；静态变量只有一个，为该类的所以实例共享；通过类名.变量名访问，且不需要创建对象；可以在非静态方法中使用静态变量。

   ```
   static String fruitType;
   Fruits.fruitType = "apple"; //静态变量调用方式
   ```

3. 常量

   使用final修饰常量，并使用大写命名；赋值后不可更改。

   ```
   final String TEST = "TEST";
   ```

2. 控制语句

控制语句有`if-else\while\do-while\for\return\break\switch`等。

```
//for-each
int array[] = {7,8,9};
for(int arr : array) {
    System.out.println(arr);
}
```

- break语句　终止循环，强行退出当前循环
- continue语句　跳出本轮循环，开始执行下一轮循环
- return语句　从一个方法返回到调用该方法的位置

3. 对象、类、接口、继承、多态

实例化对象

需要`new`关键字

```
Person p = new Person();
```

构造方法

java可根据需求定制不同的构造方法，当类中未定义构造方法时自动生成一个无参构造方法；当手动定义一个构造方法时，无参构造方法必须手动定义

```
class Apple {
    int sum;
    String color;
    public Apple() {}
    public Apple(int sum) {}
    public Apple(String color) {}
    public Apple(int sum, String color) {}
}
```

重载与重写

方法重写描述`子类与父类`间的关系，重载指`同一类中相同方法`的不同实现。

1. 方法重载的条件
    - 方法名称相同
    - 参数列表必须不同（个数、类型、顺序）
    - 返回类型可选，但不足以作为重载依据
    - 重载发生在编译期，根据参数类型选择使用某方法
2. 方法重写的规则
    - 方法必须与父类一致，包括返回类型、方法名、参数列表
    - 可使用@Override标识方法重写
    - 子类重写方法的访问权限不能低于父类方法的访问权限

类的初始化顺序

静态属性 > 静态方法块static{} > 普通属性 > 普通方法块{} > 构造函数 > 普通方法

this|super

- this：调用本类中的属性、构造函数、方法；必须出现在构造函数第一行；一个构造函数仅能调用一次。
- super：调用父类中属性、构造函数、方法；必须出现在构造函数第一行；一个构造函数仅能调用一次。

抽象类、接口

1. 继承父类使用`extends`关键字，实现接口使用`implements`关键字。

```
class Student extends Person implements Comparable { }
```

2. 抽象类使用`abstract`关键字定义，接口使用`interface`关键字定义.
    1. 接口是完全抽象类，不提供任何方法实现，只进行定义。抽象类可进行部分方法的实现。
    2. 接口只能使用public、default修饰符。
    3. 接口不能被实例化，接口中不能有构造方法。抽象类可定义全部方法、属性类型。
    4. 接口实现必须实现接口全部方法，否则需定义为抽象类。

内部类

定义在类内部的类被称为内部类，分为静态内部类、成员内部类、局部内部类、匿名内部类。

1. 静态内部类

```
public class Out{
    private static int a;
    public static class Inner {
        public void print() {
            sout(a);
        }
    }
}
```

    - 可访问外部类所有静态变量、方法，包括private
    - 可定义静态变量、方法、构造方法等
    - 其他类使用该类使用`Out.Inner inner = new Out.Inner();  inner.print();`的语法
    - Entry即是HashMap内部的静态内部类，为实际存放元素的数组对象。
2. 成员内部类

```
public class Out{
    public class Inner{
    }
}
```

    - 不能定义静态方法、变量，除非是final常量。（静态成员在成员之前初始化，若成员类中定义静态成员，则初始化顺序歧义）
3. 局部内部类

```
public void test(final int c) {
    final int d = 1;
    class Inner{
        public void print() {
            sout(c);
        }
    }
}
```

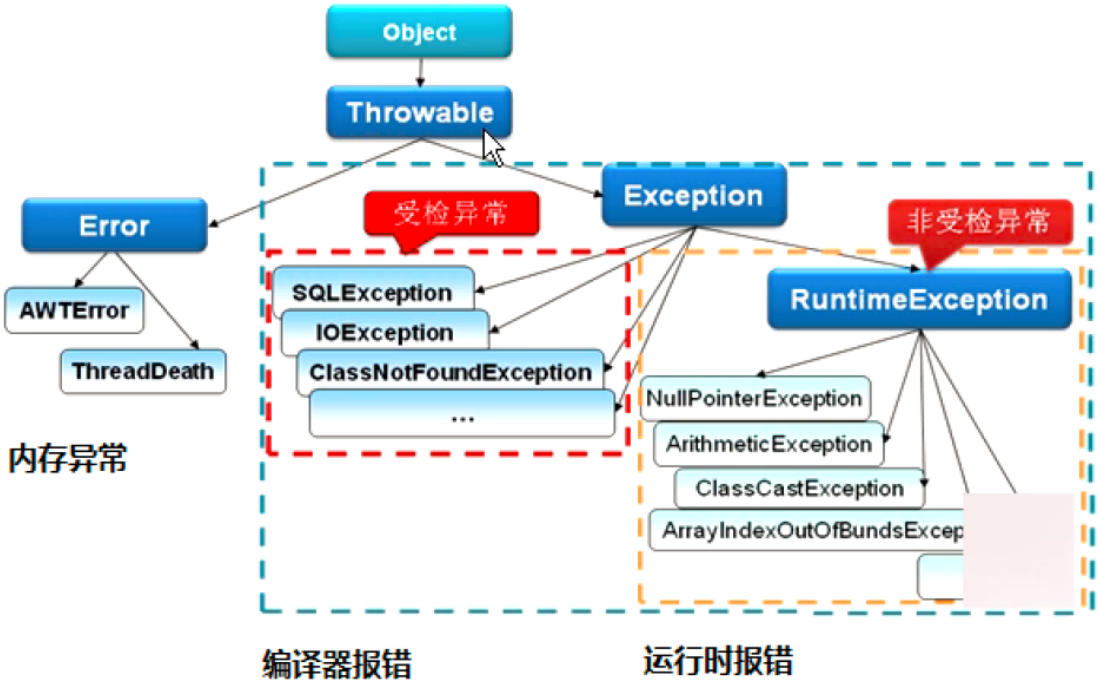
4. 匿名内部类

   匿名内部类必须要继承一个父类或实现一个接口，无class关键字，直接使用new生成一个对象引用，故称匿名

```
public abstract class Bird{
    public abstract int fly();
}
public void test(Bird bird) { ... }
public static void main(String[] args) {
    Test test = new Test();
    test.test(new Bird() {  //继承Bird类
        public int fly() {
            return 10000;
        }
    })
}
```

4. 异常处理

若某方法不能按照正常途径完成任务，按照另一途径退出方法，且抛出封装了错误信息的对象，后续运行交给异常处理器。



异常分类

RuntimeException、CheckedException，两种都应该主动捕获。

1. RuntimeException：正常运行期间抛出的异常，为开发者错误，如数组越界、空指针、非法参数、数组长度为负、非法状态、类型转换。
2. CheckedException：编译阶段发现的异常，如无指定属性、无指定方法、不允许访问某类、未找到某类。

相关关键字

1. throws、throw

   异常也被看作对象，需要被自定义抛出或程序抛出，需借助`throws、throw`定义抛出异常。

   ```
   static void cacheException() throws Exception {
       throw new Exception();
   }
   ```

   throws用在函数上，用于声明异常，表示该方法可能出现某些异常；throw抛出具体问题对象，用在函数内。

2. try、catch、finally

   主要组合有try...catch、try...finally、try...catch...finally。

   ```
   try {
       System.out.println("1");
   }catch (Exception e) {
       e.printStackTrace();
   }finally {
       System.out.println("2");
   }
   ```

   try语句任何时候都会首先执行，catch语句在try语句内发生异常时执行，finally语句在最后会执行。

Error

Error是程序运行时系统内部错误或资源耗尽错误，表示出现较严重问题，不抛出对象，只作提醒，不可检查，如`OutOfMemoryError\StackOverflowError`。

5. 数组、容器

集合类

java.util包中，主要分为set\list\map三类，分别为集合、列表、键值对映射。

- Collection：集合顶层接口List、Set、Queue的基本接口
- Iterator：迭代器，可通过该类遍历集合中数据
- Map：键值对映射的顶层接口

1. List类
   1. ArrayList　按加入排序、可重复；底层使用动态数组；查询快、增删慢；线程不安全；容量不足时自动扩容为x*1.5+1；需要连续空间存储
   2. LinkedList　按加入排序、可重复；底层使用双向链表；查询慢、增删快；线程不安全；双向链表结构无需扩容；不需要连续空间存储

3. Vector　按加入排序、可重复；底层使用数组；查询快、增删慢；线程安全，`synchronized`锁；容量不足时自动扩容为x*2

2. Set类
    1. HashSet　无序排列、不可重复；底层使用Hash表；存取速度快；内部由屏蔽了value的HashMap支持；拉链法解决Hash冲突
    2. LinkedHashSet　采用Hash表+双向链表的结构，保证存取效率时兼顾插入顺序，内部为屏蔽了value的LinkedHashMap
    3. TreeSet　无序排列、不可重复；底层使用二叉树；按一定排序存储；
3. Map类
    1. HashMap　value可重复；底层使用Hash表（数组链表红黑树）；线程不安全；允许key有唯一的null；内部类Entry(key\value\hash\next*)
    2. Hashtable　value可重复；底层使用Hash表；线程安全；不允许key有null
    3. TreeMap　value可重复；底层使用二叉树
4. ConcurrentHashMap

    分为Segment片段，每个Segment对应HashMap的底层结构，Segment通过继承ReentrantLock加锁，每次锁住一个segment，此时进行写入的segment线程安全，且又不影响其他segment的写入，兼顾了效率。Segment默认有16个，初始化后不可扩容，最多支持16线程并发写入。

6. 泛型

泛型定义

泛型指一种参数化类型，所操作的数据类型被指定为一个参数，限制添加进集合的类型，可使方法、对象有最广泛的表达能力。可以为任意标识，常使用T\E\K\V等形式。

```
public class Generic<T> {
    private T value;
    public Generic(T value) {
        this.value = value;
    }
    public T getValue() {
        return value;
    }
    public void setValue(T v) {
        this.value = v;
    }
}
//泛型方法
public static <E> void printArray(E[] input) {
    for(E element: input) {
        sout("%s" ,element);
    }
}
```

泛型通配符

使用`?`表示，元素类型可匹配任何类型。

- 上界通配符`<? extends ClassType>`，表示的任何类型都是ClassType的子类。
- 下界通配符`<? super ClassType>`，表示的任何类型都是ClassType的父类。

```
public static void generic(List<?> data) {
    System.out.println(data.get(0));
}
```

类型擦除

Java泛型基本在编译器层次实现，生成的**字节码不包含泛型的类型信息**。使用泛型时加上的类型参数会在编译时被去掉。如源码中`List<Object>|List<String>`类型，编译后会变为List。类型擦除首先找到用来替换类型参数的具体类，常为通用父类Object，若指定上界则为该上界，随后类型参数都被替换为具体的类。

7. 反射机制

动态语言：运行时可改变其结构，可引入新函数，可删除旧函数。

反射机制：运行状态中对于任意一个类都能够知道该类所有的属性、方法，都能调用其任意方法。

反射API

1. Class类：可获取类的属性、方法等信息

```
Person p = new Person();
Class clazz = p.getClass(); //1.调用对象的getClass方法
Class clazz = Person.class; //2.调用类的class属性
Class clazz = Class.forName("类的全路径")    //3.使用Class.forName静态方法，最常用
```

2. Field类：表示类的成员变量，可用来获取和设置类中的属性值

```
Field[] fields = clazz.getDeclaredFields(); //获取类的所有成员属性信息
```

3. Method类：表示类的方法，可用来获取类中方法信息或执行方法

```
Method[] methods = clazz.getDeclaredMethods();  //获取类的所有方法信息
```

4. Constructor类：表示类的构造方法

```
Constructor[] constructor = clazz.getDeclaredConstructors();    //获取类的所有构造方法信息
```

5. 创建对象的方法

```
Class clazz = Class.forName("reflection.Person");   //获取Person类的clazz对象
Person p = (Person) clazz.newInstance();    //使用Class.newInstance创建对象
Constructor c = clazz.getDeclaredConstructor(String.class, String.class, int.class);    //获取(String,String,int)参数的构造方法
Person p1 = (Person) c.newInstance("张仨","女",22);    //使用Constructor.newInstance(param)创建对象并初始化属性。
```

使用步骤

1. 获取欲操作的类的Class对象，通过该对象可任意调用类方法
2. 调用Class类中方法，进行使用
3. 使用反射API操作信息

8. 编程规范

1. 类名使用UpperCamelCase风格，除了DTO、UID等约定俗成简称外。

```
JavaServerlessPlatform/XmlService/TaPromotion
```

2. 方法名、参数名、成员变量、局部变量统一使用lowerCamelCase风格。

```
localValue/getHttpMessage()/inputUserId
```

3. 常量命名全部为大写，单词下划线分隔。

```
MAX_STOCK_COUNT/CACHE_EXPIRED_TIME
```

4. 变量命名不以下划线/美元符开始、结束。

```
不可使用： _name/$name/name_/name$
```

5. 抽象类以Abstract/Base开头，异常类以Exception结束，测试类以Test结尾，枚举类以Enum结尾

```
AbstractClassName/MyTestException/ClassNameTest/MyEnum
```

6. 避免在子父类成员变量间、不同代码块局部变量间使用相同命名变量降低可读性。
7. 使用到设计模式的模块、接口、类、方法命名时体现出设计模式。

```
OrderFactory/LoginProxy/ResourceObserver
```

8. 所有Integer对象之间值的比较，全部使用equals

> Integer var = ? 在-128~127之间赋值时，会复用IntegerCache.cache内已有对象，该范围内==可直接判断值的相等；但范围外使用==则比较引用。

9. hashCode/equals
    1. 只要重写equals就必须重写hashCode
    2. 由于Set以hashCode/equals判断不重复对象，因此Set存储对象必须重写两方法
    3. 若自定义对象作为Map的键，则必须重写两方法保证判断不重复正确
10. 使用集合转数组方法必须使用toArray(T[] array)，传入类型一致、长度为0的空数组

> 直接使用toArray无参方法只返回Object[]类，强转可能报错ClassCastException

```
List<String> list = new ArrayList<>(2);
list.add("a"); list.add("b");
String[] array = list.toArray(new String[0]);
```