1、运算符相关

```
What results from the following code fragment?



inti = 5;

intj = 10;

System.out.println(i + ~j);
```

- A、Compilation error because”~”doesn’t operate on integers
- B、-5
- C、-6
- D、15

正确答案：C

- 解法一：公式 -n=$_{n+1\text{ 可推出}}$ n=-n-1，所以~10=-11 再加 5 结果为-6
- 解法二：计算机中以补码存储。

正数的原码/反码/补码相同，所以
10 存储为 00000000 00000000 00000000 00001010
~10 的原码为 11111111 11111111 11111111 11110101（10 取反）
~10 的反码为 10000000 00000000 00000000 00001010（最高位符号位，不变，其余位取反）
~10 的补码为 10000000 00000000 00000000 00001011（负数的补码=反码+1)
所以~10 = -11

&&和&，||和|的区别：

- &&是逻辑与（短路与），当第一个判断条件不满足要求时（返回 false），第二个判断条件就不会执行；只有当两个判断条件都返回 true 时，整个逻辑运算才返回 true。
- &按位与，不论什么情况下，两边的判断条件都会执行，当两边都返回 true 时，按位与才返回 true。

- ||逻辑或，当第一个判断条件返回 true 时，逻辑或直接返回 true，第二个判断条件就不会执行了；
- |按位或，不论什么情况下，两边的判断条件都会执行，当有一个条件返回 true 时，按位或就返回 true。

注意：

逻辑与、逻辑或两边的运算符必须是 boolean 类型的，而按位与、按位或可以是 boolean 类型，两边也可以是 int 类型的。

当按位与、按位或两边是 int 类型时，将是通过二进制进行按位运算，规则就是：

- 按位与&：都为 1 时，返回 1，其他情况返回 0；
- 按位或|：有一个为 1 时，返回 1，都为 0 时才返回 0；

如下例子：

```
3&2=2,3|2=3

3-->0011

2-->0010

&-->0010=2

|-->0011=3
```

2、泛型相关

泛型在编译时期进行严格的类型检查，消除了绝大多数的类型转换。泛型在集合中使用广泛，在 JDK1.5 之后集合框架就全部加入了泛型支持。在没有使用泛型之前，我们可以往 List 集合中添加任何类型的元素数据，因为此时 List 集合默认的元素类型为 Object，而在我们使用的时候需要进行强制类型转换，这个时候如果我们往 List 中加入了不同类型的元素，很容易导致类型转换异常。如下例子：

```
List list = new ArrayList();

        list.add(18);
```

```
        list.add("lly");

        for(Object obj : list){

            int i = (int) obj;//此处运行后，将会报错

        }
```

上面在将“lly”转成 int 时，会报 ClassCastException，但是在编译时却不会出错。（在我们 JDK1.5 之后有了泛型之后，但是没有去使用泛型来定义集合，跟上面的一样效果。），同时，这也验证了我们前面总结异常类型时所说的，ClassCastException 是属于运行时异常，也即非检查性异常。

在我们使用泛型之后，可以避免不必要的转型，以及避免可能出现的 ClassCastException，如下：

```
List<Integer> list = new ArrayList<Integer>();

        list.add(18);

        list.add("lly");    //此时，编译时就不能通过，报错！！！
```

泛型允许我们在创建集合时就可以指定元素类型，当加入其他数据类型时，编译不能通过。泛型只作用于编译阶段，在编译阶段严格检查类型是否匹配，类型检查通过后，JVM 会将泛型的相关信息擦出掉（即泛型擦除），也就是说，成功编译过后的 class 文件中是不包含任何泛型信息的，泛型信息不会进入到运行时阶段。

如果像我们上面说的这样，那对于传进来的不同数据类型的对象也只会生成一个类型，而不是多种数据类型对象。下面我们可以验证一下：

```
class Person<T> {

    private T charac;//人物特征

    public Person(T ch){

        this.charac = ch;

    }
```

```
    public T getCharac() {

        return charac;

    }

    public void setCharac(T charac) {

        this.charac = charac;

    }

}
```

测试：

```
Person<String> p1 = new Person<String>("lly");

        Person<Integer> p2 = new Person<Integer>(18);

        System.out.println("p1--->"+p1.getClass());

        System.out.println("p2--->"+p2.getClass());
```

打印如下：

```
p1--->class com.scu.lly.Person

p2--->class com.scu.lly.Person
```

可以看到，虽然我们传入了两种数据类型，但是在编译时并没有生成这两种类型，而都是 Person 类型，这正是因为我们上面所说的，泛型在编译通过后，确保了类型正确，此后就擦除了相关泛型信息，把所有元素都作为 Person 数据类型。也就是说，泛型类型在逻辑上我们可以看成是多个不同的数据类型，但是在本质上它只是同一种数据类型。

类型通配符：

按我们上面的说法，泛型在编译成功后，泛型信息就被擦除，变成了同一个类型。那如何去区分原本具有父子类关系的泛型类型？如下例子：
我们在上面 Person 类的基础上，进行测试：

```java
public class CommonTest {

    public static void main(String[] args) {

        Person<Number> p1 = new Person<Number>(12);

        Person<Integer> p2 = new Person<Integer>(18);

        getCharac(p1);

        getCharac(p2);

        //报错！！！编译不能通过，提示参数类型不符合

    }

    public static void getCharac(Person<Number> person){

        System.out.println(person.getCharac());

    }

}
```

按照我们的想法，因为 Integer 是继承自 Number 的，根据 Java 多态的特性，我们调用 getCharac(p2);应该是没有问题的，但是正是因为泛型擦除的特点，导致了泛型在编译通过后被擦除了泛型类型，在运行时，JVM 根本不知道有 Number 和 Integer 这两个类型存在,内存中只会有 Person 对象存在。这也正是上面不能编译通过的原因。

为了解决这个问题，也就是说在使用泛型的时候为了能够体现出父子关系（或者说兼容多态特性），提出了类型通配符的概念。类型通配符用 ? 来代替参数类型，代表任何类型的父类，比如 Person<?>就是 Person<Number>和 Person<Integer>的父类了，而 Person<Number>和 Person<Integer>是体现不出父类关系的，现在就可以继续使用多态特性了，如下：

```java
public static void getCharac(Person<?> person){

    System.out.println(person.getCharac());

}
```

将上面的参数类型改成通配符的形式以后，我们调用 getCharac(p2);就不会出错了。

类型通配符上界和下界：

继续我们上面的例子，我们的本意是 getCharac 方法中只能传入数字类型的参数过来，也就是说希望传入的类型参数是 Number 类型或它的子类，类型通配符上界提供了这种约束。

类型通配符上界：可以这样定义：Person<? extends Number>，表示我们的参数类型最大是 Number 类型或者它的子类。继续修改我们的例子：

```
public static void getCharac(Person<? extends Number> person){

    System.out.println(person.getCharac());

}
```

测试：

```
Person<Number> p1 = new Person<Number>(12);

Person<Integer> p2 = new Person<Integer>(18);

Person<String> p3 = new Person<String>("lly");

getCharac(p1);

getCharac(p2);

getCharac(p3);

//报错！！！编译不能通过，提示参数类型不符合
```

因为 String 并不是 Number 的子类，因此上面也就不能编译通过了

类型通配符下界：可以这样定义：Person<? super Number>，表示我们的参数类型最小是 Number 类型，或者是它的超类类型。

下面我们来看一个题目：

```
Which three statements are true?

class A {}

class B extends A {}

class C extends A {}

class D extends B {}

Which four statements are true ?
```

- A、The type List<A>is assignable to List.
- B、The type List<B>is assignable to List<A>.
- C、The type List<Object>is assignable to List<?>.
- D、The type List<D>is assignable to List<? extends B>.
- E、The type List<?extends A>is assignable to List<A>.
- F、The type List<Object>is assignable to any List reference.
- G、The type List<?extends B>is assignable to List<?extends A>.

is assignable to（意为：赋给，转化为）

正确答案：ACDG

对于 A，任何使用了泛型的数据类型，都可以赋给没有使用泛型的数据类型，此时数据类型相当于是 Object，如上面我们的一个例子：

```
public static void getCharac(Person person){

    System.out.println(person.getCharac());

}
```

测试：

```
Person<Number> p1 = new Person<Number>(12);

Person<Integer> p2 = new Person<Integer>(18);
```

```
Person<String> p3 = new Person<String>("lly");

getCharac(p1);

getCharac(p2);

getCharac(p3);
```

这个时候都能通过。

由于使用泛型之后，父子关系必须要有通配符来解决。因此 CDG 正确。B 错。

对于 E，说反了，因为上面的 is assignable to 赋予关系，体现的是多态关系（父子关系）；

对于 F，和 B 一样的错误，虽然 Object 是任何类型的父类（需要跟 A 区别开来，A 是用没用泛型的比较，这里是都用了泛型之后的比较），但是用了泛型后，父子关系要有通配符来体现。同样 List<String>也不能转化为 List<Object>。

3、变量初始化问题

对于成员变量：
对于非 final 修饰的类的成员变量（包括 static 和非 static），如果开发者没有给其赋初值，在编译时，JVM 自动会给非 final 修饰的成员变量赋初值，我们在类的成员方法中就可以直接使用、运算了。

对于 final 修饰的成员变量，必须在

- 定义的时候初始化，
- 或者在构造方法中初始化（如果类中有多个构造方法，每个构造方法中都需要进行一次初始化），否则编译不通过。这是因为 final 类型的变量不能修改，必须在初始定义的时候或者 new 出对象时构造器里进行初始化，其他时候不能变更。

对于非成员变量，即方法中的临时变量：
方法中的临时变量，只需要在使用前保证了初始化就可以。不一定要在定义的时候就初始化，但必须要在开始使用这个变量前初始化。

如下例子：

```java
public class VarTest {

    final int i ;

    public VarTest(){

        //在构造方法中初始化了

        i = 3;

    }

    public VarTest(int n){

        //有多个构造方法，必须在每个构造方法中进行初始化

        i = n;

    }

    public void doSomething() {

        int j;

        j = 1;

        //对于临时变量，如果这里不进行初始化，下面使用++j 时编译不能通过

        System.out.println(++j + i);

    }

    public static void main(String[] args) {

        VarTest test = new VarTest();

        test.doSomething();

    }

}
```

4、suspend()和 resume()方法

Java.Thread 的方法 resume()负责重新开始被以下哪个方法中断的线程的执行（）？

- A、stop
- B、sleep
- C、wait
- D、suspend

正确答案：D

suspend() 和 resume() 方法：两个方法配套使用，suspend()使得线程进入阻塞状态，并且不会自动恢复，必须其对应的 resume() 被调用，才能使得线程重新进入可执行状态

5、几个需要注意的小知识点

（1）What is the result of compiling and executing the following fragment of code:

```
Boolean flag = false;

if(flag = true){

    System.out.println(“true”);

} else{

    System.out.println(“false”);

}
```

- A、The code fails to compile at the “if” statement.
- B、An exception is thrown at run-time at the “if” statement.
- C、The text“true” is displayed.
- D、The text“false”is displayed.
- E、Nothing is displayed.

正确答案：C

这里主要是要注意，if 条件判断中，flag = true 先是一个赋值语句，赋值完成后，flag 成为逻辑判断条件，会自动拆箱。

（2）What will happen when you attempt to compile and run the following code?

```
public class test{

    static{

        intx=5;

    }

    static int x,y;

    public static void main(String args[]){

        x--;

        myMethod( );

        System.out.println(x+y+ ++x);

    }

    public static void myMethod( ){

        y=x++ + ++x;

    }

}
```

- A、compiletime error
- B、prints:1
- C、prints:2
- D、prints:3
- E、prints:7
- F、prints:8

正确答案：D

注意静态代码块中的语句声明，重新定义了一个局部变量 int x = 5，执行完静态代码块后，局部变量就被销毁。此时全局变量 x 还是默认值 0，执行到打印语句时，x-1,y= 0，对于打印语句 System.out.println(x+y+ ++x);中的 x+y+ ++x 由于+是左结合的，因此 x+y+ ++x，即 1+0+（++x）.

（3）以下代码将打印出

```
public static void main (String[] args) {

    String classFile = "com. jd. ". replaceA11(".", "/") + "MyClass.class";

    System.out.println(classFile);

}
```

- A、com. jd
- B、com/jd/MyClass.class
- C、///////MyClass.class
- D、com.jd.MyClass

正确答案：C

replaceAll 方法的第一个参数是一个正则表达式，而"."在正则表达式中表示任何字符，所以会把前面字符串的所有字符都替换成"/"。如果想替换的只是"."，那么久要写成"\."。和 replace 类似，split()中的参数也是正则表达式，“.”也是关键词，如果想要使用也是要转义。即写成 str.split("\.")进行转义处理。

（4）下面将打印出什么结果？

```
int b = 127;

            System.out.println(b);

            b = b++;

            System.out.println(b);
```

- A、127
- B、128

正确答案：A

要特别注意：b = b++;和 b = ++b;这种两种方式，实现 b 自增，在 Java 中是没有效果的。对于 b = b++;b 被赋予 b++的结果，之后 b 并不会再来自增，因此打印 127；对于 b = ++b;b 被赋予++b 的结果，之后 b 也并不会再来自增，因此打印 128。

6、自动拆箱、装箱问题

（1）下列 java 程序输出结果为_____。

```
inti=0;

Integer j = newInteger(0);

System.out.println(i==j);

System.out.println(j.equals(i));
```

- A、true,false
- B、true,true
- C、false,true
- D、false,false
- E、对于不同的环境结果不同
- F、程序无法执行

正确答案：B

对于==来说：

- 如果运算符两边有一方是基本类型，一方是包装类型，在进行==逻辑判断时，包装类型会自动进行拆箱操作，因此 i==j 返回 true；
- 如果都是包装类型，那么==就是按照正常判断逻辑来，==比较的是对象的地址，但是下面这种情况除外：

在-128 至 127 这个区间，如果创建 Integer 对象的时候（1）Integer i = 1;（2）Integer i = Integer.valueOf(1); 如果是这两种情况创建出来的对象，那么其实只会创建一个对象，这些对象已经缓存在一个叫做 IntegerCache 里面了，所以==比较是相等的。如果不在-128 至 127 这个区间，不管是通过什么方式创建出来的对象，==永远是 false，也就是说他们的地址永远不会相等。

举例测试如下：

```
Integer i1 = 8;

Integer i2 = 8;

Integer i3 = 300;

//超过了127 这个范围

Integer i4 = 300;

Integer i5 = Integer.valueOf(8);

Integer i6 = new Integer(8);

// System.out.println(i1 == i2);//true, 在-128 至127 这个区间，Integer i = 1;和Intege
r i = Integer.valueOf(1); 这两种方式创建的对象相同

// System.out.println(i3 == i4);//false ，超过这个区间，创建出的是不同对象

// System.out.println(i1 == i5);//true ， 在-128 至127 这个区间，Integer i = 1;和Inte
ger i = Integer.valueOf(1); 这两种方式创建的对象相同

System.out.println(i1 == i6);

//false ，在指定区间，只有上面两种方式创建出的对象才相同，通过new 出来的是不相同的
```

对于 equals 来说：

equals 不同的对象由不同的实现，对于 Integer 来说，equals 比较的是值。因此，j.equals(i);返回的是 true。

（2）代码片段：

```
byte b1=1,b2=2,b3,b6;
```

```
final byteb4=4,b5=6;

b6=b4+b5;

b3=(b1+b2);

System.out.println(b3+b6);
```

关于上面代码片段叙述正确的是（）

- A、输出结果：13
- B、语句：b6=b4+b5 编译出错
- C、语句：b3=b1+b2 编译出错
- D、运行期抛出异常

正确答案：C

被 final 修饰的变量是常量，这里的 b6=b4+b5 可以看成是 b6=10；在编译时就已经变为 b6=10 了

而 b1 和 b2 是 byte 类型，java 中进行计算时候将他们提升为 int 类型，再进行计算，b1+b2 计算后已经是 int 类型，赋值给 b3，b3 是 byte 类型，类型不匹配，编译不会通过，需要进行强制转换。

Java 中的 byte，short，char 进行计算时都会提升为 int 类型。

7、finally 语句的执行是在 return 前还是 return 后

在 try 的括号里面有 return 一个值，那在哪里执行 finally 里的代码？

- A、不执行 finally 代码
- B、return 前执行
- C、return 后执行

正确答案：C

finally 语句是在 try（或 catch）的 return 语句执行之后，return 返回之前执行。过程如下：在 try 中如果有 return 语句，他会首先检测是否有 fianlly，如果有的话，就保存 try 中 return 要返回的值，然后执行 finally 中的方法，如果 fianlly 没有返回值，则 finally 方法执行完毕之后，返回执行 try 中的 return 方法，他会取出之前保存的 return 值，进行返回。

如下例子：

```
public static void main(String[] args) {

    intk = f_test();

    System.out.println(k);

}

public static int f_test(){

    inta = 0;

    try{

        a = 1;

        returna;

    }

    finally{

        System.out.println("It is in final chunk.");

        a = 2;

        returna;

    }

}
```

输出：

```
It is in final chunk.
```

```
2
```

如果将 return a; 注释掉，尽管在 finally 中给 a 重新赋了值，但是结果将是如下输出

```
It is in final chunk.

1
```

说明，finally 是在 try 中执行完 return 后再执行的。只有当 finally 中也有 return 的时候，方法将直接返回，不再执行热河代码。

8、Java1.7 和 Java1.8 的新特性

一、对于 JDK7：

1、支持将整数类型用二进制来表示，用 0b 开头

```
// 所有整数 int， short，long，byte 都可以用二进制表示

// An 8-bit 'byte' value:

byte abyte = (byte) 0b00100001;

// A 16-bit 'short' value:

short ashort = (short) 0b1010000101000101;

// Some 32-bit 'int' values:

intanInt1 = 0b10100001010001011010000101000101;

intanInt2 = 0b101;

intanInt3 = 0B101;

// The B can be upper or lower case.

// A 64-bit 'long' value. Note the "L" suffix:

long along = 0b10100001010001011010000101000101101000010100010110100001010001011L;
```

```
// 二进制在数组等的使用

final int[] phases = { 0b00110001, 0b01100010, 0b11000100, 0b10001001,

    0b00010011, 0b00100110, 0b01001100, 0b10011000 };
```

2、switch 语句支持 String 参数

```
String str = "a";

switch (str) {

    case "a":

                    System.out.println("a---");

    break;

    case "b":

                System.out.println("b---");

    break;

}
```

注意：在把字符串传进 Switch case 之前，别忘了检查字符串是否为 Null。

3、数字类型的下划线表示 更友好的表示方式，不过要注意下划线添加的一些标准，可以参考下面的示例

```
 long creditCardNumber = 1234_5678_9012_3456L;

    long socialSecurityNumber = 999_99_9999L;

    float pi = 3.14_15F;

    long hexBytes = 0xFF_EC_DE_5E;

    long hexWords = 0xCAFE_BABE;

    long maxLong = 0x7fff_ffff_ffff_ffffL;
```

```
    byte nybbles = 0b0010_0101;

    long bytes = 0b11010010_01101001_10010100_10010010;

    //float pi1 = 3_.1415F;      // Invalid; cannot put underscores adjacent to a d
ecimal point

    //float pi2 = 3._1415F;      // Invalid; cannot put underscores adjacent to a d
ecimal point

    //Long socialSecurityNumber1= 999_99_9999_L;         // Invalid; cannot put und
erscores prior to an L suffix

    //int x1 = _52;              // This is an identifier, not a numeric literal

    int x2 = 5_2;              // OK (decimal literal)

    //int x3 = 52_;              // Invalid; cannot put underscores at the end of a
literal

    int x4 = 5_______2;        // OK (decimal literal)

    //int x5 = 0_x52;            // Invalid; cannot put underscores in the 0x radix
prefix

    //int x6 = 0x_52;            // Invalid; cannot put underscores at the beginnin
g of a number

    int x7 = 0x5_2;            // OK (hexadecimal literal)

    //int x8 = 0x52_;            // Invalid; cannot put underscores at the end of a
number

    int x9 = 0_52;             // OK (octal literal)

    int x10 = 05_2;            // OK (octal literal)

    //int x11 = 052_;            // Invalid; cannot put underscores at the end of a
number
```

4、泛型实例的创建可以通过类型推断来简化 可以去掉后面 new 部分的泛型类型，只用<>就可以了

一般使用泛型是如下情况：

```
List<String> strList = new ArrayList<String>();
```

在 JDK7 及以后，可以简化成如下：

```
List<String> strList = new ArrayList<>();
```

5、对资源的自动回收管理

平常我们在使用一些资源时，一般会在 finally 中进行资源的释放，如下形式：

```
BufferedReader br = new BufferedReader(new FileReader(path));

try {

    return br.readLine();

}

finally {

    br.close();

}
```

而 1.7 之后我们可以使用这种形式：

```
try (BufferedReader br = new BufferedReader(new FileReader(path)) {

    return br.readLine();

}
```

直接在 try 中进行声明，跟 finally 里面的关闭资源类似； 按照声明逆序关闭资源。这些资源都需要实现 java.lang.AutoCloseable 接口的资源。

二、对于 JDK1.8：

1、接口中可以定义默认非抽象的方法

Java 8 允许我们给接口添加一个非抽象的方法实现，只需要使用 default 关键字即可，这个特征又叫做扩展方法，示例如下：

代码如下:

```
interface Formula {

    double calculate(int a);

    default double sqrt(int a) {

        return Math.sqrt(a);

    }

}
```

Formula 接口在拥有 calculate 方法之外同时还定义了 sqrt 方法，实现了 Formula 接口的子类只需要实现一个 calculate 方法，默认方法 sqrt 将在子类上可以直接使用。

代码如下:

```
Formula formula = new Formula() {

    @Override

        public double calculate(int a) {

        return sqrt(a * 100);

    }

}

;

formula.calculate(100);

// 100.0

formula.sqrt(16);
```

```
// 4.0
```

2、Lambda 表达式

首先看看在老版本的 Java 中是如何排列字符串的：

代码如下:

```
List<String> names = Arrays.asList("peter", "anna", "mike", "xenia");

Collections.sort(names, new Comparator<String>() {

    @Override

        public int compare(String a, String b) {

        return b.compareTo(a);

    }

}

);
```

只需要给静态方法 Collections.sort 传入一个 List 对象以及一个比较器来按指定顺序排列。通常做法都是创建一个匿名的比较器对象然后将其传递给 sort 方法。

在 Java 8 中你就没必要使用这种传统的匿名对象的方式了，Java 8 提供了更简洁的语法，lambda 表达式：

代码如下:

```
Collections.sort(names, (String a, String b) -> {

    return b.compareTo(a);

}

);
```

看到了吧，代码变得更段且更具有可读性，但是实际上还可以写得更短：

代码如下:

```
Collections.sort(names, (String a, String b) -> b.compareTo(a));
```

对于函数体只有一行代码的，你可以去掉大括号{}以及 return 关键字，但是你还可以写得更短点：

代码如下:

```
Collections.sort(names, (a, b) -> b.compareTo(a));
```

Java 编译器可以自动推导出参数类型，所以你可以不用再写一次类型。

3、Date API

Java 8 在包 java.time 下包含了一组全新的时间日期 API。新的日期 API 和开源的 Joda-Time 库差不多，但又不完全一样，下面的例子展示了这组新 API 里最重要的一些部分：

Clock 时钟

Clock 类提供了访问当前日期和时间的方法，Clock 是时区敏感的，可以用来取代 System.currentTimeMillis() 来获取当前的微秒数。某一个特定的时间点也可以使用 Instant 类来表示，Instant 类也可以用来创建老的 java.util.Date 对象。

代码如下:

```
Clock clock = Clock.systemDefaultZone();

long millis = clock.millis();

Instant instant = clock.instant();

Date legacyDate = Date.from(instant);

// legacy java.util.Date
```

4、支持多重注解

9、异常抛出问题

以下关于 JAVA 语言异常处理描述正确的有？

- A、throw 关键字可以在方法上声明该方法要抛出的异常。
- B、throws 用于抛出异常对象。
- C、try 是用于检测被包住的语句块是否出现异常，如果有异常，则抛出异常，并执行 catch 语句。
- D、finally 语句块是不管有没有出现异常都要执行的内容。
- E、在 try 块中不可以抛出异常

正确答案：CD

throw 用于抛出异常。
throws 关键字可以在方法上声明该方法要抛出的异常，然后在方法内部通过 throw 抛出异常对象。