Подготовка к Sun Certified Java Programmer(SCJP) exam.

Автор: ***Денис Жданов***
Enrelia

Опубликовано: *03.01.2007*
Исправлено: *15.04.2009*
Версия текста: *1.1*

Что это такое

Экзамен на SCJP – это тест, проводимый компанией Sun Microsystems, основная цель которого проверить базовые знания языка программирования Java.

Зачем

Минусы

- экзамен платный, стоимость составляет 150$;
- не самая простая и быстрая процедура сдачи;
- к нему надо готовиться;

> **ПРИМЕЧАНИЕ**
> Все нижесказанное является субъективным мнением автора.

Плюсы

- бонус при устройстве на работу;
- упорядочивание знаний у сдающего;
- повышение уверенности в своих знаниях;
- сдав на SCJP, можно сдавать Sun Certified Java Developer (SCJD), Sun Certified Web Component Developer (SCWCD), Sun Certified Business Component Developer (SCBCD), Sun Certified Developer For Java Web Services (SCDJWS), Sun Certified Mobile Application Developer (SCMAD);

Давайте поговорим подробнее об этом. С минусами понятно (про процедуру сдачи будет рассказано ниже). Давайте разберемся с плюсами. Сразу скажу, что являюсь сторонником сертификации, потому что она мне кажется достаточно объективным критерием оценки человека в профессиональном плане. Человек оценивается всегда, будь то устройство на работу или, например, взятие кредита в банке. Кажется, что дополнительный бонус при этом никогда не будет лишним. Также при подготовке происходит упорядочивание знаний, узнается что-то новое. SCJP необходим для сдачи SCJD, который, в свою очередь, проверяет умение разрабатывать реальные приложения.

Процедура сдачи

1. На адрес education@Russia.Sun.Com отправляется запрос о приобретении ваучера;
2. Выставляется счет;
3. Через некоторое время после оплаты счета(в моем случае примерно месяц) приходит ваучер;
4. На сайте http://www.2test.com находится наиболее удобный по расположению центр проведения экзамена;
5. Связываетесь с центром проведения, договариваетесь о времени;
6. Сдача экзамена ((SCJP 1.5) 72 вопроса, для успешной сдачи необходимо иметь не менее 59% правильных ответов). Каждый вопрос предлагается с вариантами ответов, задача выбрать правильный(ые);

Подготовка к SCJP 1.4

Статья не претендует на то, чтобы быть полным учебником, прочитать который достаточно для прохождения сертификации. Предполагается, что читатель знаком с основными концепциями Java, поэтому о них здесь говориться не будет. Я ставил целью рассказать о тех аспектах, которые не так часто встречаются в повседневной жизни, но знание которых проверяется во время сдачи. Также представлена интересная (на мой взгляд) информация, выходящая за рамки экзамена.

Exam objectives

- CX-310-035;

Language fundamentals

Keywords

Экзамен предполагает знание ключевых слов Java. Не требуется перечислить их все, но необходимо правильно отвечать на вопросы типа

пример

```
Which one of these lists contains only Java programming language keywords? (Choose one.)
A. class, if, void, long, Int, continue
B. goto, instanceof, native, finally, default, throws
C. try, virtual, throw, final, volatile, transient
D. strictfp, constant, super, implements, do
E. byte, break, assert, switch, include
```

СОВЕТ

Надо не забывать, что есть ключевое слово strictfp, которое означает, что над числами с плавающей точкой должны выполняться точные операции (IEE754). Также надо помнить, что const и goto - ключевые слова, которые не используются, но зарезервированы.

Ключевые слова нельзя использовать в качестве имен переменных и методов. Т.о. увидев пример кода наподобие следующего можно смело утверждать, что он не скомпилируется

пример

```
class Foo {

    public void go() {
        // complex code here
    }

    public int break(int b) {
        // code that appears to break something
    }
}
```

Хотя null, false и true, строго говоря, являются литералами, в рамках подготовки к экзамену их можно считать ключевыми словами.

СОВЕТ

На экзамене могут быть подводные камни типа использования в примере protect вместо protected, extend вместо extends и т.п.

Legal identifiers

В качестве имен переменных, методов, классов и т.д. можно использовать любую комбинацию символов unicode, чисел и знаков валюты ($) и подчеркивания (_). Причем имя не может начинаться с цифры. Т.о. следующие объявления не вызовут ошибки компиляции:

корректные объявления

```
int _a;
int $c;
int ______2_w;
int _$;
int this_is_a_very_detailed_name_for_an_identifier;
```

а эти вызовут:

некорректные объявления

```
int :b;
int -d;
int e#;
int .f;
int 7g;
```

Literals and ranges of all primitive data types

Здесь надо помнить

- размеры каждого типа;
- все, кроме char, знаковые;
- интегральные литералы бывают по основаниям 8, 10, 16;
- регистр символов в литералах не имеет значения (22l == 22L, 0xcafe == 0XCAFE и т.д.);
- по умолчанию литералы с плавающей точкой имеют тип double, т.о. следующий код не скомпилируется, потому что мы пытаемся присвоить переменной типа float (4 байта) значение типа double (8 байт)

```
float f = 0.1;
```

- для литералов с плавающей точкой можно не задавать целую часть, т.е. следующее правильно

```
double d = .5;
```

- можно использовать символьные литералы с кодом unicode

```
char c = '\u2122';
int i = '\u3100';
```

- при инициализации переменной литералом компилятор проверяет, попадает ли он в диапазон допустимых значений, т.е. следующее не скомпилируется

```
byte b = 128;
short s = '\u8fff';
```

- для символьных переменных можно использовать любое интегральное значение в диапазоне [0; 65536), т.о. следующий код скомпилируется:

```
char a = 0xFFFF;
char b = 10;
```

а этот нет

```
char c = -1;
char d = 0x10000;
```

Implicit widening

- надо помнить, что к char неявно не приводится переменная любого типа, т.е. следующее не скомпилируется:

пример

```
byte b = 1;
byte s = 2;
char c1 = b; // Compilation fails
char c2 = s; // Compilation fails
```

- любая переменная интегрального типа может быть неявно расширена до любого вещественного типа, т.о. следующий код скомпилируется:

пример

```
long var = Long.MAX_VALUE;
float f = var;
```

Array declaration, construction and initialization

- int[] i то же самое, что int i[] (хотя для лучшей читаемости рекомендуется использовать первый вариант);
- возможны записи вида

```
String[] names[] // двумерный массив String
```

СОВЕТ

Скобки после имени переменной относятся только к этой переменной, т.е. при объявлении вида
float[] f1[], f2;
f1 будет двумерный массив, а f2 одномерный.

- нельзя указывать размерность массива при его объявлении, т.е. следующее не скомпилируется

```
int[5] i;
```

- при создании массива его элементам присваиваются значения по умолчанию для соответствующего типа;
- при обращении к элементу массива в runtime производится проверка того, что индекс правильный и, в противном случае, возбуждается исключение ArrayIndexOutOfBoundsException;
- способы инициализации

пример

```
// агрегатная инициализация
int[][] scores = {{1, 2}, {1, 2, 3, 4}, {1, 2, 3}};

// создание анонимного массива
void test(String[] names) {}
…
test(new String[] {"1", "2"});
```

- массив можно инициализировать агрегатно только в момент объявления, т.е. следующее не скомпилируется:

пример

```
int[] i1 = {1}; // OK
int[] i2;
i2 = {1}; // Compilation fails
```

- при создании анонимного массива также нельзя явно указывать его размерность, т.е. следующее не скомпилируется

```
new int[2] {1, 2};
```

- массивы примитивов разных типов нельзя присваивать друг другу, т.е. следующее не скомпилируется

пример

```
int[] i;
char[] c = new char[2];
i = c;
```

- массивы ссылок разных статических типов можно присваивать друг другу, если между ними выполняется отношение IS-A. Следующий пример скомпилируется, потому что Sub IS-A Base.

пример

```
class Base {}
class Sub extends Base {}
...
Base[] b;
Sub[] s = new Sub[1];
b = s;
```

- хотя ссылки на массивы разных типов и можно присваивать друг другу в случае отношения IS-A, при попытке использовать значение неверного типа получим исключительную ситуацию времени выполнения:

пример

```
public class StartClass {
    public static void main(String[] args) {
        Integer[] intArray = {new Integer(1), new Integer(2)};
        Number[] numberArray = intArray;
        numberArray[0] = new Double(1.0); // java.lang.ArrayStoreException: java.lang.Double
    }
}
```

Uninitialized variables

- все поля класса инициализируются значениями по умолчанию;
- все локальные переменные требуют явной инициализации перед использованием;

Launching JVM

Экзамен предполагает, что испытуемый должен считать, что при запуске JVM должен быть указан класс, в котором затем ищется и запускается метод с сигнатурой

```
public static void main(java.lang.String[] args);
```

ПРИМЕЧАНИЕ

Вообще говоря, это не совсем так, потому что при работе, например, через JNI наличие main не требуется.

Имя переменной java.lang.String[] может быть любым, т.е. следующий метод будет найден и запущен при старте

```
public static void main(String[] aaa) {}
```

Исходя из вышесказанного логично, что при запуске следующего примера программа выведет в консоль сообщение об ошибке, потому что метод main, принимающий на вход массив из java.lang.String, в классе StrungOut не определен. Там присутствует метод main, принимающий на вход массив объектов класса String, объявленного в этом же файле:

```
Exception in thread "main" java.lang.NoSuchMethodError: main
```

пример

```
public class StrungOut {
    public static void main(String[] args) {
        String s = new String("Hello world");
        System.out.println(s);
    }
}

class String {
    private final java.lang.String s;

    public String(java.lang.String s) {
        this.s = s;
    }

    public java.lang.String toString() {
        return s;
    }
}
```

ПРИМЕЧАНИЕ

Задача выходит из круга рассматриваемых на экзамене.

Declaration and access control

Declarations and modifiers

- в файле может быть только один невнутренний public класс, причем имя файла должно совпадать с именем класса;
- если в файле нет ни одного public класса, его имя может быть любым;
- если в файле есть объявление package, то оно должно быть первой строкой кода;
- надо знать, какие бывают модификаторы доступа и что они означают применительно к классам и их членам;

СОВЕТ

Надо помнить, что если какой-то класс не видит другой, он не видит ни один из его методов независимо от модификатора доступа. Т.е. в следующем примере Bar не скомпилируется, потому что ему не виден весь класс Foo.

Main.java

```
package com.mycompany.first

public class Main {
}

class Foo {
    public static void callMe() {
    }
}
```

Bar.java

```
package com.mycompany.second

public class Bar {
    Bar() {
        Foo.callMe();
    }
}
```

- модификатор protected означает, что член класса виден только классам из этого же пакета и наследникам. Причем если наследник находится в другом пакете, он не имеет доступа к protected-члену через ссылку на объект суперкласса. Что это значит? Это значит, например, что следующий код не скомпилируется:

Base.java

```
package pkg1;

public class Base {
    protected int i;
}
```

Sub1.java

```
package pkg2;

import pkg1.Base;

public class Sub1 extends Base {

    void test(Base base) {
        System.out.println(base.i); // Compilation fails
    }
}
```

В то же время объект подкласса может обращаться к protected-полям суперкласса через цепочку наследования либо через ссылку на объект этого же класса, т.о. следующий класс скомпилируется:

Sub2.java

```
package pkg2;

import pkg1.Base;

public class Sub2 extends Base {

    void test1() {
        System.out.println(super.i);
    }

    void test2(Sub2 sub) {
        System.out.println(sub.i);
    }
}
```

- в объявлении класса кроме модификаторов доступа могут присутствовать следующие модификаторы

модификаторы

```
final
abstract
```

```
strictfp
```

- абстрактный класс может не иметь ни одного абстрактного метода, т.е. следующее скомпилируется

```
public abstract class Foo {
}
```

- на абстрактные методы помимо основных ограничений (не static, не private, не final) наложен запрет на использование следующих ключевых слов, относящихся к реализации метода

ключевые слова

```
native
synchronized
strictfp
```

- переопределяющий метод можно объявлять как abstract, т.е. следующий код скомпилируется

пример

```
abstract class Base {
    public abstract void meth1();
    public void meth2() {}
}

abstract class Sub extends Base {
    public void meth1() {}
    public abstract void meth2();
}
```

- объекты абстрактного класса нельзя создавать, а массивы можно:

пример

```
public class StartClass {
    public static void main(String[] args) {
        MyAbstractClass object = new MyAbstractClass(); // Compilations fails
        MyAbstractClass[] array = new MyAbstractClass[4]; // OK
    }
}

abstract class MyAbstractClass {
}
```

- члены-переменные не переопределяются, а перекрываются (shadowing), т.о. следующий код не скомпилируется, потому что мы пытаемся обратиться к закрытому члену класса (B.i).

пример

```
public class StartClass {
    public static void main(String[] args) {
        System.out.println(new B().i);
        // Исправляется так:
        // System.out.println(((A)new B()).i);
    }
}
class A {
    public int i = 1;
}

class B extends A {
    private int i = 2;
}
```

- к локальным переменным можно применять только модификатор final;
- из статического контекста нельзя обращаться к нестатическому, т.о. следующий пример не скомпилируется.

пример

```
public class StartClass {

    private final String DUMMY = "DUMMY";

    public static void main(String[] args) {
        System.out.println(DUMMY);
    }
}
```

- обращаться к статическим полям и методам можно и через объект класса. В данном случае это не больше, чем синтаксический сахар. По статическому типу ссылки объекта компилятор просто распознает нужный класс. Т.о. следующее будет работать правильно.

пример

```
public class StartClass {
    public static void main(String[] args) {
        ((Foo)null).test();
    }
}

class Foo {
    static void test() {
        System.out.println("Foo.test()");
    }
}
```

Interfaces

- для интерфейсов можно применять множественное наследование;
- необязательно указывать все ключевые слова при объявлении интерфейса, его полей и методов, т.о. все нижеследующие объявления одинаковы:

пример

```
public interface Testable {
    int FIELD = 1;
    void test();
}
public abstract interface Testable {
    int FIELD = 1;
    void test();
}
public interface Testable {
    public static final int FIELD = 1;
    void test();
}
public interface Testable {
```

```
    int FIELD = 1;
    public abstract void test();
}
```

- общие правила наследования и реализации интерфейсов;

СОВЕТ

Помните, что все переменные интерфейса public static final, а все методы public. Т.о. следующий код не скомпилируется, потому что пытается изменить терминальное поле counter

пример

```
interface Count {
    short counter = 0;
    void countUp();
}

public class TestCount implements Count {

    public static void main(String[] args) {
        TestCount t = new TestCount();
        t.countUp();
    }

    public void countUp() {
        for (int x = 6; x > counter; x--, ++counter) {
            System.out.print(" " + counter);
        }
    }
}
```

Operators and assignments

Operators

- надо знать, что делает каждый из операторов Java;
- результат выполнения любого оператора над интегральными переменными, меньшими, чем int, есть int. Т.о. следующий код не скомпилируется, потому что мы пытаемся присвоить переменной типа byte значение типа int.

пример

```
byte b1 = 1;
byte b2 = 2;
byte b3 = b1 + b2; // possible loss of precision
```

- compound assignment operators отличаются от обычных тем, что они выполняются по формуле

формула и пример

- все compound assignment operators требуют, чтобы операнды были примитивного типа кроме оператора +=. Он позволяет правому операнду быть любого типа, если левый String, т.е. следующее правильно

пример

```
String s = “abc”;
s += 7.0;
```

- оператор % может вернуть отрицательное число. Справедлива формула

```
(a / b) * b + (a % b) == a
```

- надо помнить о порядке выполнения операторов

пример

```
int a = 1;
int b = 2;
System.out.println("" + a + b); // prints ‘12’
System.out.println(a + b); // prints ‘3’
```

- JVM возбудит исключительную ситуацию java.lang.ArithmeticException при делении интегральной переменной на ноль;
- при делении числа с плавающей точкой на ноль (интегрального числа на 0.0) результатом будет бесконечность (Double.POSITIVE_INFINITY, Double.NEGATIVE_INFINITY, Float.POSITIVE_INFINITY, Float.NEGATIVE_INFINITY).

пример

```
double d1 = 1.5;
double d2 = d1 / 0;
System.out.println(Double.isInfinite(d2)); // true

int i1 = -2;
double d3 = i1 / 0.0;
System.out.println(Double.isInfinite(d3)); // true
```

- Double.NaN и Float.NaN не равны чему бы то ни было:

пример

```
double d1 = Double.NaN;
double d2 = Double.NaN;
System.out.println(d1 == d2); // prints false
System.out.println(Double.isNaN(Float.NaN)); // prints true
```

- при работе с примитивами в джаве возможно переполнение. Так, следующий пример напечатает 5, потому что хотя переменная MICROS_PER_DAY имеет тип long, ей присваивается результат выполнения (24 * 60 * 60 * 1000 * 1000), то есть int. Во время вычисления этого выражения и произойдет переполнение:

пример

```
public class StartClass {
    public static void main(String[] args) {
        final long MICROS_PER_DAY = 24 * 60 * 60 * 1000 * 1000;
        final long MILLIS_PER_DAY = 24 * 60 * 60 * 1000;
        System.out.println(MICROS_PER_DAY / MILLIS_PER_DAY);
    }
}
```

ПРИМЕЧАНИЕ

Задача выходит из круга рассматриваемых на экзамене.

- оператор instanceof требует, чтобы его первый операнд был ссылочной переменной, и чтобы либо статический тип первого операнда был приводим ко второму, либо наоборот, т.е. следующий код не скомпилируется, потому что ни List приводим к String, ни String к List:

пример

```
List string = new ArrayList();
System.out.println(string instanceof String);
```

- аналогично при компиляции проверяется, принадлежат ли операнды оператора приведения типа к одному дереву наследования:

пример

```
public class StartClass {

    public static void main(String[] args) {
        First first = new First();
        Second second = new Second();
        First anotherFirst = (First)second; // compilation fails
    }
}

class First {
}

class Second {
}
```

- если первым аргументом оператору instanceof передан null, результат всегда будет false

пример

```
String string = null;
System.out.println(string instanceof String); // false
System.out.println(null instanceof String); // false
```

- работа с операторами побитового сдвига происходит следуюшим образом: проверяется, что операнды интегрального типа, если левый операнд не шире int, к правому операнду применяется маска 0x1f, в противном случае 0x3f. После этого выполняется сдвиг на получившееся в результате применения маски число. Т.о. 1 << -30 эквивалентно 1 << 2.

задача

```
Может ли оператор >>> вернуть отрицательное число?
```

ответ

```
Может: (-1 >>> 32).
```

- надо помнить, что при использовании оперторов && и || правое выражение может не выполняться. На экзамене любят задачи вида

задача

```
Что напечатает программа?
public class StartClass {
    public static void main(String[] args) {
        int x = 0;
        int y = 0;
        for (int z = 0; z < 5; z++) {
            if ((++x > 2) || (++y > 2)) {
                x++;
            }
        }
        System.out.println(x + " " + y);
    }
}
```

- при применении оператора приведения типа к переменным, находящимся под воздействием другого оператора, к нужному типу приводится только первый операнд, т.о. следующий код не скомпилируется, потому что мы пытаемся присвоить переменной типа byte переменную, получившуюся в результате деления byte на long, т.е. long:

```
long x = 5;
long y = 2;
byte b = (byte) x / y;
```

Passing variables into methods

- надо помнить, что в Java все переменные передаются по значению, т.о. если метод принимает ссылочную переменную в качестве аргумента, на самом деле он получает на вход копию ссылки. Т.к. это копия, она указывает на тот же объект, что оригинал, но мы можем перенаправить ссылку на другой объект и изменять его, причем это никак не повлияет на оригинал. Пример задачи на эту тему:

пример

```
public class StartClass {
    public static void main(String[] args) {
        String s1 = "java";
        StringBuffer s2 = new StringBuffer("java");
        replaceString(s1);
        replaceStringBuffer(s2);
        System.out.println(s1); // prints 'java'
        System.out.println(s2); // prints 'javac'
    }

    static void replaceString(String s) {
        s = s.replace('j', 'l');
    }

    static void replaceStringBuffer(StringBuffer s) {
        s.append("c");
        s = new StringBuffer();
        s.append("d");
    }
}
```

Flow control, exceptions and assertions

if-else branching

- для ясности всегда лучше применять фигурные скобки. На экзамене попадаются вопросы вида

пример

```
// Что напечатает следующий фрагмент?
if (exam.done())
    if (exam.getScore() < 0.61)
        System.out.println(“Try again.”);
else
    System.out.println(“Java master!”);
```

Надо помнить, что else всегда относится к ближайшему if;

- if принимает булево выражение, поэтому на экзамены попадаются вопросы, где в секции условия оператора if находится присваивание булевой переменной.

пример

```
// Что напечатает следующая программа?
public class StartClass {
    public static void main(String[] args) {
        boolean foo = false;
        if (foo = true) {
            System.out.println("Never prints");
        }
    }
}
```

- если определение переменной происходит внутри if-блока, то при попытке использовать ее после if получим ошибку компиляции;

пример

```
public class StartClass {
    public static void main(String[] args) {
        int i;
        boolean b = true;
        if (b) {
            i = 1;
        }
        System.out.println(i);
    }
}
```

ПРИМЕЧАНИЕ

Это относится не только к инициализации переменной внутри if, но и вообще к использованию локальной переменной, которая может быть не инициализирована.

switch statement

- switch принимает на вход аргумент типа int (следовательно, из-за автоматического расширения подходят byte, char, short), т.о. следующий код не скомпилируется:

пример

```
long i = 1;
switch (i) {
    default:
        System.out.println("default");
        break;
}
```

ПРИМЕЧАНИЕ

Начиная с java 1.5 оператор switch работает также и с перечислениями.

- в качестве аргумента в case можно использовать переменную, но она должна быть объявлена с модификатором final, т.о. следующий код не скомпилируется

пример

```
public class StartClass {
    public static void main(String[] args) {
        int stage1 = 1;  // исправляется объявлением stage1 final.
        int i = 1;
        switch (i) {
            case stage1:
                System.out.println("first stage");
                break;
            default:
                System.out.println("default");
                break;
        }
    }
}
```

- компилятор проверяет, что значение каждого аргумента case не шире типа пременной, передаваемой в switch, т.о. следующий код не скомпилируется:

пример

```
public class StartClass {
    public static void main(String[] args) {
        char i = 1;
        switch (i) {
            case 0x7fffffff:
                System.out.println("max int");
                break;
            default:
                System.out.println("default");
                break;
        }
    }
}
```

- default case не обязательно должен находиться в последней позиции switch, т.е. расположение default по отношению к другим case не влияет на итоговый выбор:

пример

```
public class StartClass {
    public static void main(String[] args) {
        int i = 1;
        switch (i) {
            default:
                System.out.println("default");
                break;
            case 1:
                System.out.println("first"); // попадем сюда, а не в default
                break;
        }
    }
}
```

Loops

- в примерах на работу с циклами могут встречаться задания, где отсутствуют фигурные скобки и используется нарочито неправильное форматирование:

пример

```
// Что напечатает программа?
```

```
public class Test {
    public static void main(String [] args) {
        int i = 1;
        do while (i < 1)
            System.out.print("i is " + i);
        while (i > 1) ;
    }
}
```

Эта программа ничего не напечатает, потому что на самом деле данный код эквивалентен следующему

пример

```
public class Test {
    public static void main(String [] args) {
        int i = 1;
        do {
            while (i < 1) {
                System.out.print("i is " + i);
            }
        } while (i > 1) ;
    }
}
```

- в цикле do-while между do и while должен быть какой-то код иначе получим ошибку компиляции

пример

```
public class StartClass {
    public static void main(String[] args) {
        do while(false);
        // можно исправить, записав так
        // do ; while(false);
    }
}
```

- если между меткой и циклом есть код, то при отсутствии фигурных скобок получим ошибку компиляции

пример

```
loop : { // (1)
    System.out.println("test");
    for (;;) {
        break loop;
    }
} // (2)
//если бы фигурные скобки в строках (1) и (2) отсутствовали, была бы ошибка компиляции
```

- break и continue можно использовать вместе с метками:

пример break

```
public class StartClass {
    public static void main(String[] args) {
        boolean isTrue = true;
        outer:
        for (int i = 0; i < 5; i++) {
            while (isTrue) {
                System.out.println("Hello");
                break outer;
            } // end of inner while loop
            System.out.println("Outer loop."); // Won't print
        } // end of outer for loop
        System.out.println("Good-Bye"); // Will print
    }
}
```

пример continue

```
public class StartClass {
    public static void main(String[] args) {
        outer:
        for (int i = 0; i < 5; i++) {
            for (int j = 0; j < 5; j++) {
                System.out.println("Hello");
                continue outer;
            } // end of inner loop
            System.out.println("outer"); // Never prints
        }
        System.out.println("Good-Bye");
    }
}
```

- цикл for не требует наличия никакой из своих частей. По умолчанию условное выражение в нем всегда true, т.о. следующее является бесконечным циклом

```
for ( ; ; ) {}
```

Handling exceptions

- терминология: exception – это не наследник java.lang.Exception, а исключительная ситуация вообще. Т.о., например, объект java.lang.Error тоже является exception. Обобщая, можно сказать, что exception – любой класс, для которого выполняется отношение IS-A java.lang.Throwable. Исключительные ситуации делятся на контролируемые (checked) и неконтролируемые (unchecked). Неконтролируемая исключительная ситуация – любой класс, для которого выполняется IS-A java.lang.RuntimeException или IS-A java.lang.Error; контролируемая – все остальные. Контролируемая исключительная ситуация называется так, потому что компилятор следит за тем, чтобы программист ее обрабатывал или декларировал, что метод может пробросить ее дальше по стеку. Отсюда следует, что, увидев код наподобие нижеприведенного, можно утверждать, что он не скомпилируется, потому что контролируемая исключительная ситуация не обрабатывается и не декларируется:

пример

```
public class StartClass {
    public static void main(String[] args) {
        throw new Throwable();
    }
}
```

- компилятор следит за тем, чтобы не было недостижимого кода, т.е. кода вида

пример

```
public void test() {
    return;
    System.out.println("test"); // эта строка недостижима
}
```

Подобное может возникнуть и при обработке исключительных ситуаций. Приведенный ниже код не скомпилируется, потому что контролируемая исключительная ситуация не возбуждается нигде в блоке try, и, следовательно, блок catch по контролируемой исключительной ситуации недостижим

пример

```
public void test() {
    try {
        System.out.println("Inside empty try");
    } catch(IOException ignore) {
    }
}
```

Если мы обрабатываем несколько типов исключительных ситуаций, связанных отношением IS-A, надо следить за тем, чтобы обработчик исключительной ситуации общего типа не предшествовал остальным. Приведенный ниже фрагмент не скомпилируется, потому что после обработки java.lang.Exception управление никогда не сможет попасть в обработчик java.lang.IOException (IOException наследник Exception)

пример

```
try {
    throw new IOException();
} catch (Exception e) {
      // operate Exception
} catch (IOException e) {
      // operate IOException
}
```

СОВЕТ

Надо различать, какой код является достижимым, а какой нет. Например, следующий фрагмент кода скомпилируется, потому что в нем присутствует catch по java.lang.Exception, который сам по себе является контролируемой исключительной ситуацией, но в то же время является родителем неконтролируемой исключительной ситуации java.lang.RuntimeException(и всех потомков этого класса). Компилятор считает, что неконтролируемая исключительная ситуация может возникнуть всегда, и, следовательно, обработчик по java.lang.Exception является достижимым

пример

```
public void test() {
    try {
        System.out.println("Inside empty try");
    } catch(Exception ignore) {
    }
}
```

- блок finally выполняется всегда кроме случаев экстренного завершения программы. Т.о. следующий пример будет печатать 100

пример

```
public class StartClass {
    public static void main(String[] args) {
        System.out.println(getInt());
    }

    private static int getInt() {
        try {
            return 10;
        } finally{
            return 100;
        }
    }
}
```

- надо помнить, что между блоками try и catch, try и finally, catch и catch, catch и finally не может находиться никаких выражений. На экзамене встречаются вопросы наподобие нижеприведенного.

пример

```
Given the following,

System.out.print("Start ");
try {
    System.out.print("Hello world");
    throw new FileNotFoundException();
}
System.out.print(" Catch Here ");
catch(EOFException e) {
    System.out.print("End of file exception");
} catch (FileNotFoundException e) {
    System.out.print("File not found");
}

and given that EOFException and FileNotFoundException are both subclasses of IOException, and further assuming this block of code is
 placed into a class, which statement is most true concerning this code?
A. The code will not compile.
B. Code output: Start Hello world File Not Found.
C. Code output: Start Hello world End of file exception.
D. Code output: Start Hello world Catch Here File not found.
```

ответ

Working with assertions

- assertion может состоять из одного или двух выражений

одно выражение

```
public double sqrt(double base) {
    assert (0 <= base);
    // some stuff
}
```

два выражения

```
public double sqrt(double base) {
    assert (0 <= base) : "Can't calculate square root for the negative number(" + base + ")";
    // some stuff
}
```

Надо помнить, что второе выражение должно возвращать значение. Т.о. следующий пример не скомпилируется, потому что метод foo(), использующийся во втором выражении assert, ничего не возвращает

пример

```
public class Test {
    public static int y;

    public static void foo(int x) {
        System.out.print("foo ");
        y = x;
    }
```

```
    public static int bar(int z) {
        System.out.print("bar ");
        return y = z;
    }

    public static void main(String [] args) {
        int t = 0;
        assert t > 0 : bar(7);
        assert t > 1 : foo(8);  // Compilation fails
        System.out.println("done ");
    }
}
```

- по умолчанию assertions отключены, т.е. можно использовать assert как, например, имя переменной. Для того, чтобы скомпилировать код с использованием assert как ключевого слова, необходимо явно указать это с помощью специального ключа компилятора:.

```
javac -source 1.4
```

- следующие флаги позволяют регулировать обработку assertions во время выполнения программы. По умолчанию они не обрабатываются

флаги

```
-ea   // enables assertions
-enableassertions  // то же, что -ea
-esa // enables assertions for system classes
-enablesystemassertions // то же, что -esa
-da // disables assertions
-disableassertions  // то же, что -da
-dsa // disables assertions for system classes
-disablesystemassertions // то же, что -dsa
```

- при запуске виртуальной машины можно комбинировать использование assertions. Например, в следующем примере после старта JVM assertions будут обрабатываться для всех классов кроме системных и классов из пакета com.mycompany.myproduct.unchecked и всех его подпакетов

```
java -ea -dsa -da:com.mycompany.myproduct.unchecked...
```

Object orientation, overloading and overriding, constructors and return types

Overriden methods

ПРИМЕЧАНИЕ

Начиная с джава 1.5 с некоторым ограничением можно менять тип значения, возвращаемого переопределяющим методом. Это называется covariant returns.

- если метод не может быть наследован, его нельзя переопределять. Приведенный ниже пример не скомпилируется, потому что класс Sub не наследует метод method() из класса Base

пример

```
public class StartClass {
    public static void main(String[] args) {
        Sub sub = new Sub();
        sub.method();
    }
}

class Base {
    private void method() {}
}

class Sub extends Base {
}
```

- переопределяющий метод не может иметь в сигнатуре новую или более широкую по классу контролируемую исключительную ситуацию. Т.о. следующие примеры не скомпилируются

новая контролируемая исключительная ситуация

```
class Base {
    public void method() {}
}

class Sub extends Base {
    public void method() throws Exception {}
}
```

более широкий класс конторолируемой исключительной ситуации

```
class Base {
    public void method() throws Exception {}
}

class Sub extends Base {
    public void method() throws Throwable {}
}
```

- в переопределяющем методе можно указывать объявление более узкого класса контролируемой исключительной ситуации относительно переопределяемого метода. Т.о. следующий пример скомпилируется (IOException наследник Exception)

пример

```
class Base {
    public void method() throws Exception {}
}

class Sub extends Base {
    public void method() throws IOException {}
}
```

- статические методы не переопределяются. В совокупности с тем, что их можно вызывать через объектную переменную, могут попасться задания, в которых используется код подобный приведенному:

пример

```
public class StartClass {

    public static void main(String[] args) {
        Base sub = new Sub();
        sub.test(); // печатается 'Base.test()', потому что вызывается метод класса, определяемого статическим типом переменной (Base)
    }
}
```

```
class Base {
    public static void test() {
        System.out.println("Base.test()");
    }
}

class Sub extends Base {
    public static void test() {
        System.out.println("Sub.test()");
    }
}
```

Overloaded methods

- метод можно перегружать как в том же классе, так и в подклассе, поэтому надо различать перегрузку и переопределение. Следующий фрагмент является примером перегрузки, а не переопределения, поэтому на перегруженный метод не применяется ограничение переопределения (объявление новой контролируемой исключительной ситуации)

пример

```
class Base {
    public void doStuff(int y, String s) {}
    public void moreThings(int x) {}
}

class Sub extends Base {
    public void doStuff(int y, float s) throws IOException {}
}
```

- бывает, что одному вызову может соответствовать несколько перегруженных методов. В таком случае будет вызван наиболее специфичный из них. В приведенном ниже примере будет вызван метод, принимающий int, а не метод, принимающий long, потому что любую переменную типа int можно использовать как long, но не наоборот.

пример

```
public class StartClass {

    private static void method(int i) {
        System.out.println("method(int)");
    }

    private static void method(long i) {
        System.out.println("method(long)");
    }

    public static void main(String[] args) {
        byte b = 1;
        method(b);
    }
}
```

- выбор нужного перегруженного метода происходит во время компиляции, а не во время исполнения. Метод выбирается исходя из статического типа переданных ему аргументов. Т.о. в следующем примере программа вызовет method(Base), потому что, хотя действительный тип передаваемого аргумента Sub, его статический тип Base:

пример

```
public class StartClass {

    private static void method(Base b) {
        System.out.println("using Base");
    }

    private static void method(Sub s) {
        System.out.println("using Sub");
    }

    public static void main(String[] args) {
        Base sub = new Sub();
        method(sub);
    }
}

class Base {
}

class Sub extends Base {
}
```

- при вызове метода компилятор проверяет, что у класса, определяемого статическим типом переменной, действительно есть нужный метод. Т.о. следующий пример не скомпилируется, потому что у класса, определяемого статическим типом переменной (Animal) нет метода eat (String).

пример

```
public class StartClass {
    public static void main(String[] args) {
        Animal horse = new Horse();
        horse.eat("grass");
    }
}

class Animal {
    public void eat() {
        System.out.println("Generic Animal Eats Generically");
    }
}

class Horse extends Animal {
    public void eat() {
        System.out.println("Horse eats hay ");
    }

    public void eat(String s) {
        System.out.println("Horse eats " + s);
    }
}
```

- допустим, есть N независящих друг от друга интерфейсов, в которых объявлены методы с одинаковыми сигнатурами и типами возвращаемых значений, но разными декларациями контролируемых исключительных ситуаций. Класс, реализующий эти интерфейсы, может указать в сигнатуре указанного метода класс контролируемой исключительной ситуации не шире пересечения классов контролируемых исключительных ситуаций, объявленных в методах интерфейса. Давайте разберем это нечитаемое утверждение на примере:

пример

```
interface First {
    void method() throws IOException;
}

interface Second {
    void method() throws FileNotFoundException ;
}
```

```
class Impl implements First, Second {
    public void method() throws FileNotFoundException {
    }
}
```

FileNotFoundException потомок IOException, значит их пересечение есть FileNotFoundException. Поэтому метод method() класса Impl не может, например, указать в свой декларации IOException, т.о. следующий код не скомпилируется:

пример

```
// …
class Impl implements First, Second {
    public void method() throws IOException {
    }
}
```

Constructors and instantiation

- в каждом классе всегда есть конструктор. Если программист не объявит ни один конструктор явно, компилятор сгенерирует конструктор без аргументов, который будет обладать таким же модификатором видимости, что и его класс;
- первой строкой любого конструктора всегда идет вызов либо конструктора суперкласса, либо другого конструктора этого класса. Если программист не сделает этого явно, компилятор вставит первой строкой конструктора вызов конструктора без аргументов родительского класса. Исходя из этого, следующий пример не скомпилируется, потому что компилятор сгенерирует для класса Sub конструктор без аргументов, первой строкой которого будет вызов конструктора предка(Base) без аргументов. У класса Base нет конструктора без аргументов, поэтому получим ошибку:

пример

```
class Base {
    Base(int i) {}
}

class Sub extends Base {
}
```

- при попытке вызвать из конструктора этот же конструктор получим ошибку компиляции:

пример

```
class Base {
    Base(int i) {
        this(i);
    }
}
```

- при создании любого объекта сначала будут создаваться объекты всех его суперклассов, и, следовательно, вызываться их конструкторы. В следующем примере сначала будет вызван конструктор java.lang.Object, потом конструктор Base, потом конструктор Sub:

пример

```
public class StartClass {

    public static void main(String[] args) {
        Sub sub = new Sub();
    }
}

class Base {
    Base() {
        System.out.println("Base c-tor");
    }
}

class Sub extends Base {
    Sub() {
        System.out.println("Sub c-tor");
    }
}
```

- в качестве аргумента при вызове конструктора суперкласса нельзя передавать нестатические поля класса, результат выполнения нестатического метода или ссылку this. Т.о. в следующем примере вызовы конструктора предка (1) и (2) вызовут ошибку компиляции, а (3) и (4) будут работать правильно:

пример

```
class Base {
    public Base(String s) {
        System.out.println("Base(" + s + ")");
    }
}

class Sub extends Base {

    private static final String CLASS_DUMMY_STRING = "CLASS_DUMMY_STRING";
    private final String INSTANCE_DUMMY_STRING = "INSTANCE_DUMMY_STRING";

    public Sub() {
//        super(INSTANCE_DUMMY_STRING); (1)
//        super(getInstanceString());   (2)
//        super(CLASS_DUMMY_STRING);    (3)
        super(getClassString());     // (4)
    }

    private static String getClassString() {
        return CLASS_DUMMY_STRING;
    }

    private String getInstanceString() {
        return INSTANCE_DUMMY_STRING;
    }
}
```

- в классе можно определять методы, имя которых совпадает с именем класса. Надо отличать их от конструкторов. Пример подобной задачи:

пример

```
Given the following,

public class ThreeConst {
    public static void main(String [] args) {
        new ThreeConst();
    }

    public void ThreeConst(int x) {
        System.out.print(" " + (x * 2));
    }

    public void ThreeConst(long x) {
        System.out.print(" " + x);
```

```
    }

    public void ThreeConst() {
        System.out.print("no-arg ");
    }
}

what is the result?
A. no-arg
B. 8 4 no-arg
C. no-arg 8 4
D. Compilation fails.
E. No output is produced.
F. An exception is thrown at runtime.
```

Правильный ответ E, потому что в классе ThreeConst программистом явно не определен ни один конструктор. То, что похоже на конструкторы, на самом деле методы (у них указан тип возвращаемого значения void), поэтому компилятор сгенерировал конструктор без аргументов, который и был вызван при создании объекта;

java.lang – the Math class, Strings and wrappers

Strings

- надо помнить, что объекты String являются неизменяемыми, и, следовательно, вызов любых методов на объекте String не изменит его:

пример

```
String x = "Java";
x.concat(" Rules!");
System.out.println("x = " + x);  // the output is: x = Java
x.toUpperCase();
System.out.println("x = " + x);  // the output is still: x = Java
x.replace('a', 'X');
System.out.println("x = " + x);  // the output is still: x = Java
```

- при создании объекта String с помощью new всегда будет создан объект на куче, при использовании же литералов будет браться объект из пула строк, поэтому, в зависимости от создания, результат сравнения через метод equals() может совпадать с результатом сравнения через оператор ==

пример

```
public class StartClass {
    public static void main(String[] args) {
        String string1 = "Java";
        String string2 = "Java";
        String string3 = new String("Java");
        String string4 = new String("Java");
        System.out.println(string1 == string2); // true
        System.out.println(string1 == string3); // false
        System.out.println(string3 == string4); // false
        System.out.println(string1.equals(string3)); // true
        System.out.println(string3.equals(string4)); // true
    }
}
```

- у массивов есть поле length, у класса String есть метод length(), на экзамене бывают вопросы с вариантами их неправильного использования:

пример

```
String x = "test";
System.out.println(x.length);  // compiler error
String [] y = new String[3];
System.out.println(y.length());  // compiler error
```

- надо помнить, что метод equals() не переопределен для класса StringBuffer, т.о. следующий пример напечатает false:

пример

```
public class StartClass {
    public static void main(String[] args) {
        StringBuffer buffer1 = new StringBuffer("aa");
        StringBuffer buffer2 = new StringBuffer("aa");
        System.out.println(buffer1.equals(buffer2)); // Prints false.
    }
}
```

java.lang.Math class

- на экзамене проверяется знание того, что делают следующие методы класса java.lang.Math:

методы

```
ceil()
floor()
random()
abs()
max()
min()
round()
sqrt()
toDegrees()
toRadians()
tan()
sin()
cos()
```

Могут попасться вопросы наподобие нижеприведенного:

пример

```
public class Degrees {
    public static void main(String [] args) {
        System.out.println(Math.sin(75));
        System.out.println(Math.toDegrees(Math.sin(75)));
        System.out.println(Math.sin(Math.toRadians(75)));
        System.out.println(Math.toRadians(Math.sin(75)));
    }
}
what line will the sine of 75 degrees be output?
```

ответ

Wrapper classes

- у каждого класса-обертки за исключением Character есть два конструктора – один принимает в качестве параметра значение соответствующего примитивного типа, второй текстовое представление значения. В классе Character определен только конструктор, принимающий на вход переменную типа char.
- конструкторы классов-оберток целочисленных типов, принимающие на вход строку, рассматривают ее как десятичное число, т.о. при выполнении следующего примера возникнет NumberFormatException:

пример

```
Integer i = new Integer("0xff");
System.out.println(i.intValue());
```

- а следующий напечатает false:

пример

```
Integer i1 = new Integer("042");
Integer i2 = new Integer(042);
System.out.println(i1.equals(i2));
```

- текстовый аргумент конструктора Boolean является нечувствительным к регистру букв, т.о. следующий пример отработает нормально:

пример

```
Boolean b = new Boolean("TruE");
```

СОВЕТ

В 1.4 нельзя использовать объект Boolean в качестве условия в условном операторе, т.о. следующее вызовет ошибку компиляции:

пример

```
Boolean b = Boolean.TRUE;
if (b) {
    System.out.println("ok");
}
```

- при сравнении с помощью equals() объектов разных классов-оберток, результат всегда будет false:

пример

```
public class StartClass {
    public static void main(String[] args) {
        Integer var1 = new Integer(5);
        Long var2 = new Long(5);
        System.out.println(var1.equals(var2));  // Prints false
    }
}
```

Objects and collections

- надо помнить, что следующие методы принадлежат классу java.lang.Object, и, следовательно, всем создаваемым классам. Также надо знать, что они делают (это изложено в javadoc для java.lang.Object):

методы

```
boolean equals(Object obj)
void finalize()
int hashCode()
final void notify()
final void notifyAll()
final void wait()
String toString()
```

- надо помнить ограничения, действующие на методы equals() и hashCode(). Эта информация также доступна в javadoc к java.lang.Object. В частности, надо уметь отличать корректную реализацию от некорректной. На экзамене встречаются вопросы типа нижеприведенного:

вопрос

```
Does the following hashCode() implementation legal?

public int hashCode() {
    return 1;
}
```

ответ

- надо помнить основные интерфейсы, их методы и классы из java Collections framework;
- надо помнить, что интерфейс Map не наследует интерфейс Collection;
- на экзамене проверяется понимание того, что является упорядоченным (ordered: ArrayList, LinkedHashSet etc) и сортируемым (sorted: TreeSet, TreeMap etc);
- надо понимать, когда объект становится доступным для сборки garbage collector'ом. На эту тему встречаются вопросы типа нижеприведенного:

пример

```
Given the following,

1. public class X {
2.     public static void main(String [] args) {
3.             X x = new X();
4.             X x2 = m1(x);
5.             X x4 = new X();
6.             x2 = x4;
7.             doComplexStuff();
8.         }
9.         static X m1(X mx) {
10.        mx = new X();
11.     return mx;
12.    }
13. }

After line 6 runs. how many objects are eligible for garbage collection?
```

ответ

- надо внимательно следить за тем, сколько объектов действительно могут быть удалены сборщиком мусора:

пример

```
Given:
class CardBoard {

    StringBuffer story = new StringBuffer(“text”);

    CardBoard go(CardBoard cb) {
        cb = null;
        return cb;
    }

    public static void main(String[] args) {
        CardBoard c1 = new CardBoard();
```

```
        CardBoard c2 = new CardBoard();
        CardBoard c3 = c1.go(c2);
        c1 = null;
        // do stuff
    }
}

When // do stuff is reached, how many objects are eligible for GC?

A. 0
B. 1
C. 2
D. Compilation fails.
E. It is not possible to know.
F. An exception is thrown at runtime.
```

ответ

- объекты, в которых хранятся перекрестные ссылки друг на друга, недоступные из запущенного потока, могут быть собраны сборщиком мусора. Ниже приведен пример вопроса на эту тему:

пример

```
Given the following,

1.      class X {
2.          public X x;
3.          public static void main(String [] args) {
4.              X x1 = new X();
5.              X x2 = new X();
6.              x1.x = x2;
7.              x2.x = x1;
8.              x1 = new X();
9.              x2 = x1;
10.      // doComplexStuff();
11.     }
12. }

after line 9 runs, how many objects are eligible for garbage collection?
```

ответ

СОВЕТ

Такой набор объектов называется islands of isolated objects.

- про метод finalize() надо помнить, что в результате его выполнения объект может избежать удаления, т.е. станет снова strongly reachable. Также надо помнить, что finalize() определенного объекта вызвается только один раз, т.е. если некий объект был собран сборщиком мусора, был вызван его метод finalize(), и объект снова стал 'живым', в следующий раз, когда его соберет сборщик мусора, finalize() не будет вызван;

Inner classes

- внутренние классы бывают четырех видов:

```
Inner Classes (none-static inner classes)
Method-local Inner Classes
Anonymous Inner Classes
Static Nested Classes
```

- для создания объекта нестатического внутреннего класса всегда нужен объект внешнего класса. Из класса, в котором объявлен внутренний класс, его объект можно создавать, используя только имя класса; из внешнего кода всегда надо указывать объект внешнего класса. В следующем примере объекты внутреннего класса создаются в строках (1), (2) и (3):

пример

```
public class StartClass {
    public static void main(String[] args) {
        TestOuter myObject = new TestOuter();
        myObject.testOuter(1);
        myObject.new TestInner(2); // (2)
        new TestOuter().new TestInner(3); // (3)

    }
}

class TestOuter {
    public class TestInner {
        public TestInner(int i) {
            System.out.println("Hello from inner class (" + i + ")");
        }
    }

    public void testOuter(int i) {
        new TestInner(i); // (1)
    }
}
```

- для того, чтобы указать ссылку на объект окружающего класса из внутреннего класса, надо использовать комбинацию имени внешнего класса и ключевого слова this:

пример

```
public class StartClass {
    public static void main(String[] args) {
        new TestOuter().new TestInner().testInner();
    }

}

class TestOuter {
    public class TestInner {
        public void testInner() {
            System.out.println("Inner class reference: " + this);
            System.out.println("Outer class reference: " + TestOuter.this);
        }
    }
}
```

- во внутреннем нестатическом классе нельзя объявлять статические неконстантные поля и статические методы, т.о. если откомментировать строчку (1) или (2), получим ошибку компиляции:

пример

```
class TestOuter {
    public class TestInner {
        //static int i1 = 4; // (1)
        static final int i2 = 4;
        //static void test() {} // (2)
```

```
    }
}
```

- надо не забывать о том, что внутренний класс, объявленный в методе, может использовать только те аргументы метода и локальные переменные, которые объявлены с модификатором final. Т.о. следующий пример вызовет ошибку компляции:

пример

```
class TestOuter {
    void testOuter(int i) {
        class TestInner {
            void testInner() {
                System.out.println("parameter = " + i);
            }
        }
        new TestInner().testInner();
    }
}
```

- для того, чтобы создать объект статического внутреннего класса, надо указать только имя внешнего класса. Попытка использовать, например, объект внешнего класса вызовет ошибку компиляции:

пример

```
public class StartClass {
    public static void main(String[] args) {
        TestOuter.TestInner inner = new TestOuter.TestInner();
        TestOuter testOuter = new TestOuter();
        TestOuter.TestInner inner2 = testOuter.new TestInner(); // compilation fails
    }
}

class TestOuter {
    static class TestInner {
    }
}
```

Threads

- надо помнить, что у класса java.lang.Thread есть следующие конструкторы:

конструкторы

```
Thread()
Thread(Runnable target)
Thread(Runnable target, String name)
Thread(String name)
Thread(ThreadGroup group, Runnable target)
Thread(ThreadGroup group, Runnable target, String name)
Thread(ThreadGroup group, String name)
```

- поток нельзя сделать демоном, если он уже стартовал, но можно изменить его имя:

пример

```
public class StartClass {
    public static void main(String[] args) throws InterruptedException {
        Thread t = new Thread() {

            public void run() {
                try {
                    sleep(Long.MAX_VALUE);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };
        t.start();
        Thread.sleep(1000);
        t.setName("MyThread"); // Ok
        t.setDaemon(true); // produces IllegalThreadStateException
    }
}
```

- поток нельзя запустить дважды, т.о. следующие примеры вызовут исключительную ситуацию во время выполнения:

повторный запуск запущенного потока

```
public class StartClass {
    public static void main(String[] args) throws InterruptedException {
        Thread t = new Thread() {

            public void run() {
                try {
                    sleep(Long.MAX_VALUE);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };
        t.start();
        t.start(); // produces IllegalThreadStateException
    }
}
```

повторный запуск завершившегося потока

```
public class StartClass {
    public static void main(String[] args) throws InterruptedException {
        Thread t = new Thread();
        t.start();
        t.join();
        t.start(); // produces IllegalThreadStateException
    }
}
```

- потоку можно назначить приоритет из интервала [Thread.MIN_PRIORITY; Thread.MAX_PRIORITY]. При попытке использовать значение вне указанного интервала, получим исключительную ситуацию;
- нельзя синхронизироваться по примитивам, т.о. следующий код вызовет ошибку компиляции:

пример

```
int i = 1;
synchronized(i) { // compilation fails
    System.out.println("test");
}
```

- в случае объявления нестатического метода с ключевым словом synchronized, при его вызове лочится весь объект, в случае объявления статического метода с synchronized лочится соответствующий объект класса Class:

пример

```
class Test {

    static synchronized void test1() {} // lock on 'Test.class' reference

    synchronized void test2() {} // lock on 'this' reference
}
```

- если в потоке вызывается метод sleep() класса Thread, поток сохраняет владение всеми взятыми к этому моменту локами; если в потоке вызван метод wait() класса Object, поток освобождает все взятые к этому моменту локи;
- методы wait(), notify(), notifyAll() должны быть вызваны из синхронизированного по соответствующему объекту контекста, иначе получим исключительную ситуацию:

пример

```
public class StartClass {
    public static void main(String[] args) {
        args.notify(); // produces IllegalMonitorStateException
    }
}
```

- надо помнить, что в сигнатуре методов Thread.sleep() и wait() из Object присутствует объявление контролируемой исключительной ситуации InterruptedException, которую необходимо обработать. На экзамене попадаются вопросы вида:

пример

```
Given the following,
class Test {
    public static void main(String [] args) {
        printAll(args);
    }

    public static void printAll(String[] lines) {
        for(int i=0;i<lines.length;i++){
            System.out.println(lines[i]);
                Thread.currentThread().sleep(1000);
            }
    }
}
What is the result of this code?

A. Each String in the array lines will output, with a 1-second pause.
B. Each String in the array lines will output, with no pause in between because this method is
not executed in a Thread.
C. Each String in the array lines will output, and there is no guarantee there will be a pause
because currentThread() may not retrieve this thread.
D. This code will not compile.
```

ответ

- при попытке использования нулевой ссылки в блоке synchronized получим ошибку времени выполнения:

пример

```
public class StartClass {
    public static void main(String[] args) {
        Object o = null;
        synchronized (o) { // throws NullPointerException
            System.out.println("test");
        }
    }
}
```

Подготовка к SCJP 1.5

Exam objectives

- CX-310-055;

Var-args methods

- при объявлении метода с переменным числом аргументов число пробелов не регламентировано, т.е. все следующие объявления являются корректными:

пример

```
void test1(Object...args) {}
void test2(Object      ...args) {}
void test3(Object...      args) {}
void test4(Object      ...      args) {}
```

- в методе можно иметь как обычные параметры, так и параметр с переменным числом аргументов, но при этом var-arg параметр должен быть последним параметром метода:

пример

```
void test1(int i, Object...args) {} // OK
void test2(Object...args, int i) {} // compilation fails
```

- в методе может быть только один var-arg параметр:

пример

```
void test(int... i, float... f) {} // compilation fails, too many var-arg parameters
```

Enums

- перечисления нельзя объявлять в методах:

пример

```
public class StartClass {
    public static void main(String[] args) {
        enum Number {ONE, TWO}; // compilation fails
    }
}
```

- перечисления нельзя объявлять в нестатических внутренних классах:

пример

```
class Outer {
```

```
    private class Inner {
        enum Color {BLACK, WHITE}; // Compilation fails
    }
}
```

- точка с запятой после перечисления не является обязательной:

пример

```
public class StartClass {

    enum Number {ONE, TWO}; // OK
    enum AnotherNumber {THREE, FOUR} // OK
}
```

- в перечислениях можно объявлять поля, конструкторы, методы:

пример

```
public class StartClass {
    public static void main(String[] args) {
        System.out.println("BIG=" + Size.BIG.getSize() + ", SMALL=" + Size.SMALL.getSize());
    }
}

enum Size {
    BIG(10), SMALL(1);

    private int size;

    Size(int size) {
        this.size = size;
    }
    public int getSize() {
        return size;
    }
}
```

- конструктор перечисления может быть вызван только неявно, при объявлении члена перечисления:

пример

```
public class StartClass {
    public static void main(String[] args) {
        Size size = new Size(5); // compilation fails
    }
}

enum Size {
    BIG(10), SMALL(1);

    private int size;

    Size(int size) {
        this.size = size;
    }

    public int getSize() {
        return size;
    }
}
```

- подобно тому, как для доступа к статическим членам класса можно использовать объект этого класса, для доступа к членам перечисления можно использовать экземпляр перечисления:

пример

```
public class StartClass {

    private static Size size;

    public static void main(String[] args) {
        System.out.println("BIG=" + size.BIG.getSize() + ", SMALL=" + size.SMALL.getSize()); // OK
    }
}

enum Size {
    BIG(10), SMALL(1);

    private int size;

    Size(int size) {
        this.size = size;
    }

    public int getSize() {
        return size;
    }
}
```

- для отдельных элементов перечисления можно переопределять метод перечисления. Синтаксис похож на тот, что используется при объявлении анонимных внутренних классов. Это свойство называется *constant specific class body*:

пример

```
public class StartClass {

    public static void main(String[] args) {
        System.out.println(Size.BIG); // prints 'Ordinary size'
        System.out.println(Size.MEDIUM); // prints 'Ordinary size'
        System.out.println(Size.SMALL); // prints 'Really small'
    }
}

enum Size {
    BIG,
    MEDIUM,
    SMALL {
        public String toString() {
            return "Really small";
        }
    };

    public String toString() {
        return "Ordinary size";
    }
}
```

- обращение к статическим полям перечисления, объявленным без ключевого слова final, из конструктора, блока инициализации или при инициализации нестатической переменной вызывает ошибку компиляции:

пример

```
enum Test1 {
    FIRST, SECOND;
    private static int test = 1;

    private int i = test; // Compilation fails
}

enum Test2 {
    FIRST, SECOND;
    private static int test = 1;

    {
        System.out.println(test); // Compilation fails
    }
}

enum Test3 {
    FIRST, SECOND;
    private static int test = 1;

    Test3() {
        System.out.println(test); // Compilation fails
    }
}
```

- перечисления можно использовать в switch, т.о. следующий код скомпилируется и корректно отработает:

пример

```
public class StartClass {

    enum Size {BIG, SMALL};

    public static void main(String[] args) {
        Size size = Size.BIG;
        switch (size) {
            case BIG:
                System.out.println("Really big");
        }
    }
}
```

- при использовании перечисления в case-выражении оператора switch нельзя указывать полное имя перечисления, иначе получим ошибку компиляции:

пример

```
public class StartClass {

    enum Size {BIG, SMALL};

    public static void main(String[] args) {
        Size size = Size.BIG;
        switch (size) {
            case Size.BIG: // Compilation fails
                System.out.println("Really big");
        }
    }
}
```

Covariant returns

- переопределяющий метод может объявить тип возвращаемого значения, для которого выполняется отношение IS-A по отношению к типу возвращаемого значения переопределяемого метода:

пример

```
class Base {
    public Object test() {
        return new Object();
    }
}

class Sub extends Base {

    public Sub test() { // OK because Sub IS-A Object
        return this;
    }
}
```

Coupling and cohesion

SCJP 1.5 требует знания терминов coupling и cohesion и, соответственно, понимания, почему именно low coupling и high cohesion есть хорошо. На всякий случай привожу ссылки на эти определения в википедии:

- coupling: http://en.wikipedia.org/wiki/Coupling_(computer_science)
- cohesion: http://en.wikipedia.org/wiki/Cohesion_(computer_science)

Autoboxing / autounboxing

- надо помнить, что при использовании autoboxing/autounboxing код, который мы видим, может вести себя не так, как мы ожидаем. Например, можно ли представить, что при вызове приведенной функции случится NullPointerException?

функция

```
static void doStuff(int x) {
    System.out.println(x);
}
```

Ответ - можно, потому что при autounboxing компилятор генерирует за нас дополнительный код. Например, если мы пишем

```
Integer i1 = new Integer(100);
int i2 = i1;
```

то этот код на самом деле эквивалентен следующему:

```
Integer i1 = new Integer(100);
int i2 = i1.intValue();
```

возвращаясь к первоначальному примеру, NullPointerException может возникнуть, например, так:

пример

```
public class StartClass {

    static Integer i;

    public static void main(String[] args) {
        doStuff(i); // исключение возбуждается, потому что на самом деле здесь doStuff(i.intValue()), а i = null
    }

    static void doStuff(int x) {
        System.out.println(x);
    }
}
```

- что можно сказать о результате выполнения следующего кода?

пример

```
public class StartClass {
    public static void main(String[] args) {
        Integer i1 = 1000; // преобразовывается в Integer i1 = Integer.valueOf(1000);
        Integer i2 = 1000; // преобразовывается в Integer i2 = Integer.valueOf(1000);
        System.out.println(i1 == i2);
        System.out.println(i1 == 1000);
    }
}
```

ответ

С целью не потерять обратную совместимость с предыдущими версиями джавы при первом сравнении проверяется, ссылаются ли i1 и i2 на один и тот же объект. Не ссылаются. Второе же сравнение не вызовет ошибку компиляции только начиная с java 1.5, поэтому для этого случая (один операнд примитив, ко второму может быть применен autounboxing) возможно было реализовать сравнение действительных значений.

- принимая во внимание предыдущий пункт, что можно сказать о выполнении следующего кода?

пример

```
public class StartClass {
    public static void main(String[] args) {
        Integer i1 = 10;
        Integer i2 = 10;
        System.out.println(i1 == i2);
        System.out.println(i1 == 10);
    }
}
```

ответ

Здесь сравнение i1 с i2 дает положительный результат, потому что при autoboxing в случае, когда соотвествующее значение примитивного типа принадлежит определенному интервалу, для одинаковых примитивных значений возвращается один и тот же объект. 10 – число из этого интервала, поэтому при autoboxing ссылки i1 и i2 связываются с одним и тем же объектом Integer. Такой эффект происходит при autoboxing Boolean, autoboxing Byte, Short, Integer, Long для значений из [-128; 127], autoboxing Character для значений из ['\u0000'; '\u007F']. На Double и Float это не распространяется, поэтому, например, следующий код напечатает false:

пример

```
public class StartClass {
    public static void main(String[] args) {
        Double d1 = 1d;
        Double d2 = 1d;
        System.out.println(d1 == d2);
    }
}
```

- переменной примитивного метода не позволяется неявно расшириться, потом выполнить autoboxing, поэтому следующий пример не скомпилируется:

пример

```
public class StartClass {
    public static void main(String[] args) {
        byte b = 1;
        test(b); // compilation fails because b is not allowed to wide from byte to long and then make autoboxing
    }

    private static void test(Long var) {
    }
}
```

Т.к. здесь b не может быть неявно расширина, происходит autoboxing (byte -> Byte), потом проверяется, может ли переменная типа Byte использоваться как переменная типа Long (т.е. проверяется, верно ли, что Byte IS-A Long). Ответ нет, поэтому получаем ошибку компиляции;

- переменная, полученная в результате autoboxing, может использоваться вместо переменной другого типа, если между ними существует отношение IS-A:

пример

```
public class StartClass {
    public static void main(String[] args) {
        byte b = 1;
        test(b); // OK because Byte IS-A Object
    }

    private static void test(Object var) {
    }
}
```

- autounboxing работает в switch. Следующий фрагмент кода скомпилируется и корректно отработает:

пример

```
switch (new Integer(4)) {
    case 4:
        System.out.println("boxing is OK");
}
```

- в case-секциях оператора switch можно использовать переменные, если они объявлены как final. Это условие не выполняется в случае переменных, для которых нужен autounboxing, т.е. следующее не скомпилируется:

пример

```
final Integer i = 1;
switch (1) {
    case i: // compilation fails
        System.out.println("unboxing is OK");
}
```

Overloading rules

- при выборе перегруженного метода предпочтение отдается тому, для вызова которого надо выполнить неявное расширение типа, по сравнению с тем, для вызова которого надо

выполнить autoboxing:

пример

```
public class StartClass {
    public static void main(String[] args) {
        int i = 1;
        test(i); // prints 'test(long)'
    }

    private static void test(Integer i) {
        System.out.println("test(Integer)");
    }

    private static void test(long var) {
        System.out.println("test(long)");
    }
}
```

- неявное расширение типа предпочитается var-args:

пример

```
public class StartClass {
    public static void main(String[] args) {
        byte b = 1;
        test(b, b); // prints 'test(int, int)'
    }

    private static void test(int x, int y) {
        System.out.println("test(int, int)");
    }

    private static void test(byte ... b) {
        System.out.println("test(byte ...)");
    }
}
```

- autoboxing предпочитается var-args:

пример

```
public class StartClass {
    public static void main(String[] args) {
        byte b = 1;
        test(b, b); // prints 'test(Byte, Byte)'
    }

    private static void test(Byte x, Byte y) {
        System.out.println("test(Byte, Byte)");
    }

    private static void test(byte ... b) {
        System.out.println("test(byte ...)");
    }
}
```

Static import

- начиная с java 1.5 появилась возможность импортировать не только классы, но и их статические поля и методы:

пример

```
import static java.lang.Math.*;

public class StartClass {
    public static void main(String[] args) {
        System.out.println(max(1, 2)); // calls java.lang.Math.abs(int, int)
    }
}
```

- так же, как и с обычными импортами, в случае неопределенности при использовании статических импортов, ее надо устранить, иначе получим ошибку компиляции:

пример

```
import static java.lang.Integer.*;
import static java.lang.Long.*;

public class StartClass {
    public static void main(String[] args) {
        System.out.println(MAX_VALUE); // Compilation fails: both Integer.MAX_VALUE and Long.MAX_VALUE matches
    }
}
```

Bitwise operators

- в отличие от SCJP 1.4, в SCJP 1.5 не включены вопросы по операторам, работающим с битами (>>, >>>, <<, >>=, >>>=, <<=, &, |, ^, &=, |=, ^=). Причем операторы &, |, ^ рассматриваются, но только в примерах с булевыми выражениями;

Enchanced for loop (for-each)

- цикл foreach можно применять либо с объектами, классы которых реализуют интерфейс java.lang.Iterable, либо с массивами:

пример

```
public class StartClass {
    public static void main(String[] args) {
        List<String> list = new ArrayList<String>(); // List<E> extends Collection<E> & Collection<E> extends Iterable<E>
        list.add("a");
        list.add("b");
        for (String s : list) { // Compilation successful
            System.out.print(s + " "); // prints 'a b '
        }
        System.out.println("");

        MyIterable iterable = new MyIterable();
        for (Integer i : iterable) { // Compilation successful
            System.out.print(i + " "); // prints '0 1 2 '
        }
    }
}

class MyIterable implements Iterable<Integer> {

    private static class MyIterator implements Iterator<Integer> {

        private int counter = -1;
```

```
        public boolean hasNext() {
            return ++counter < 3;
        }

        public Integer next() {
            return counter;
        }

        public void remove() {
        }
    }

    public Iterator<Integer> iterator() {
        return new MyIterator();
    }
}
```

- при использовании for-each переменная, использующаяся для хранения значения из массива или коллекции, должна быть объявлена внутри цикла, иначе получим ошибку компиляции:

пример

```
int[] array = {1, 2};
int x;
for (x : array) { // compilation fails
    System.out.println(x);
}
```

- при обходе с помощью for-each тип обходимого массива или коллекции должен быть совместим с типом переменной, используемой при итерации:

пример

```
int[] array1 = {1, 2};
for (long x : array1) { // compilation successfully
    System.out.println(x);
}

long[] array2 = {3, 4};
for (int x : array2) { // compilation failes: 'possible loss of precision'
    System.out.println(x);
}
```

Compiling assertion-aware code

- компилятор 1.5 (в отличие от компилятора 1.4) по умолчанию рассматривает assert как ключевое слово. Если же запустить его с опцией –source 1.3, то при обработке кода, где assert используется не как ключевое слово, будет сгенерировано предупреждение;

StringBuilder class

- java.lang.StringBuilder это класс, предоставляющий те же возможности, что и java.lang.StringBuffer за искоючением того, что методы StringBuilder не синхронизированы;
- equals() для StringBuilder не переопределен (так же как и для StringBuffer):

пример

```
public class StartClass {
    public static void main(String[] args) {
        StringBuilder builder1 = new StringBuilder("test");
        StringBuilder builder2 = new StringBuilder("test");
        System.out.println(builder1.equals(builder2)); // prints false
    }
}
```

File navigation and I/O

- этот раздел не присутствовал в SCJP 1.4. Для SCJP 1.5 необходимо знать, для чего нужны и как работают нижеперечисленные классы и их методы:

File

```
File(File, String)
File(String)
File(String, String)
createNewFile()
delete()
exists()
isDirectory()
isFile()
list()
mkdir()
renameTo()
```

FileReader

```
FileReader(File)
FileReader(String)
read()
```

BufferedReader

```
BufferedReader(Reader)
read()
readLine()
```

FileWriter

```
FileWriter(File)
FileWriter(String)
close()
flush()
write()
```

BufferedWriter

```
BufferedWriter(Writer)
close()
flush()
newLine()
write()
```

PrintWriter

```
PrintWriter(File)
PrintWriter(String)
PrintWriter(OutputStream)
```

```
PrintWriter(Writer)
close()
flush()
format()
printf()
print()
println()
write()
```

СОВЕТ

Методы PrintWriter не пробрасывают IOException. Узнать, не произошла ли ошибка, можно только вызвав на нем метод checkError(). Поэтому (мое субъективное мнение) использовать его не стоит, достаточно BufferedWriter.

пример вопроса на эту тему

```
Given that bw is a reference to a valid BufferedWriter and the snippet:

15.  BufferedWriter b1 = new BufferedWriter(new File("f"));
16.  BufferedWriter b2 = new BufferedWriter(new FileWriter("f1"));
17.  BufferedWriter b3 = new BufferedWriter(new PrintWriter("f2"));
18.  BufferedWriter b4 = new BufferedWriter(new BufferedWriter(bw));

What is the result?

A. Compilation succeeds.
B. Compilation fails due only to an error on line 15.
C. Compilation fails due only to an error on line 16.
D. Compilation fails due only to an error on line 17.
E. Compilation fails due only to an error on line 18.
F. Compilation fails due to errors on multiple lines.
```

ответ

- надо не забывать, что многие методы классов из java.io объявляют конролируемую исключительную ситуацию IOException, обработка которой может быть пропущена в экзаменационных вопросах;
- файл на жестком диске создается при вызове File.createNewFile() либо при создании FileWriter или FileOutputStream, т.о. выполнении следующего примера создаст в текущем каталоге файл abcdefg.txt:

создание файла

```
public class StartClass {
    public static void main(String[] args) throws IOException {
        FileOutputStream fOut = new FileOutputStream(new File("abcdefg.txt"));
        fOut.close();
    }
}
```

Но это правило не распространяется на создание новых каталогов:

пример

```
public class StartClass {
    public static void main(String[] args) throws IOException {
        File myDir = new File("mydir");
        File myFile = new File(myDir, "myfile.txt");
        FileWriter writer = new FileWriter(myFile); // FileNotFoundException: mydir\myfile.txt (The system cannot find the path specified)
        writer.close();
    }
}
```

- при попытке прочитать из несуществующего файла, файл не создается;

пример

```
public class StartClass {
      public static void main(String [] args) throws IOException {
          FileReader reader = new FileReader(new File("abcdefg.txt")); // java.io.FileNotFoundException: abcdefg.txt (The system cannot find the file specified)
      }
}
```

- файлы нельзя создавать в несуществующем каталоге:

пример

```
public class StartClass {
      public static void main(String [] args) throws IOException {
          File dir = new File("mydir"); // assuming that no such dir exists
          File file = new File(dir, "myFile");
          file.createNewFile(); // java.io.IOException: The system cannot find the path specified
      }
}
```

- нельзя удалить непустой каталог:

пример

```
public class StartClass {
    public static void main(String[] args) throws IOException {
        File myDir = new File("mydir");
        myDir.mkdir();
        File myFile = new File(myDir, "myfile.txt");
        myFile.createNewFile();
        System.out.println(myDir.delete()); // prints false
        myFile.delete();
        System.out.println(myDir.delete()); // prints true
    }
}
```

- непустой каталог можно переименовывать;
- надо помнить, что в классе File есть открытые константы, при объявлении которых не соблюдены Sun's Code Conventions. Встретив их на экзамене, не надо думать, что это опечатка и пример не скомпилируется:

константы

```
File.separatorChar
File.separator
File.pathSeparatorChar
File.pathSeparator
```

Serialization

- при сериализации объекта также происходит сериализация всего графа объектов, на которые ссылается данный. Если какой-то из объектов графа не является сериализуемым, получим ошибку времени выполнения:

пример

```
public class StartClass {
    public static void main(String[] args) throws IOException {
        ByteArrayOutputStream bOut = new ByteArrayOutputStream();
        ObjectOutputStream oOut = new ObjectOutputStream(bOut);
        Whole object = new Whole();
        oOut.writeObject(object); //java.io.NotSerializableException: Part
    }
}

class Whole implements Serializable {
    private final Part part = new Part();
}

class Part {
}
```

- методы writeObjet() и readObject() должны быть объявлены с модификатором видимости private, иначе они не будут использоваться при сериализации/десериализации:

пример

```
public class StartClass {
    public static void main(String[] args) throws Exception {
        ByteArrayOutputStream bOut = new ByteArrayOutputStream();
        ObjectOutputStream oOut = new ObjectOutputStream(bOut);
        Whole object = new Whole();
        oOut.writeObject(object);
    }
}
class Whole implements Serializable {
    public void writeObject(ObjectOutputStream out) {
        System.out.println("Whole.writeObjet()"); // is never called since method is public
    }
}
```

- надо следить, чтобы десериализация происходила в том же порядке, что и сериализация:

пример

```
public class StartClass {
    public static void main(String[] args) throws Exception {
        ByteArrayOutputStream bOut = new ByteArrayOutputStream();
        ObjectOutputStream oOut = new ObjectOutputStream(bOut);
        oOut.writeObject(new Whole());

        ByteArrayInputStream bIn = new ByteArrayInputStream(bOut.toByteArray());
        ObjectInputStream oIn = new ObjectInputStream(bIn);
        oIn.readObject(); //java.io.StreamCorruptedException
    }
}
class Whole implements Serializable {

    transient Part part = new Part(1);
    int wholePart;

    private void writeObject(ObjectOutputStream out) throws IOException {
        out.writeInt(part.partField);
        out.defaultWriteObject();
    }

    private void readObject(ObjectInputStream in) throws Exception {
        // lines (1) and (2) must be changed
        in.defaultReadObject(); // (1)
        part = new Part(in.readInt()); // (2)
    }
}

class Part {
    int partField;

    public Part(int field) {
        this.partField = field;
    }
}
```

- при десериализации объекта класса, реализующего Serializable, его конструктор не выполняется, а поля не инициализируются значениями по умолчанию. Однако если создать сериализуемый (через Serializable) класс, который наследует несериализуемый класс, при десериализации его объекта происходит инициализация по умолчанию полей суперкласса и вызов его конструктора:

пример

```
public class StartClass {
    public static void main(String[] args) throws Exception {
        ByteArrayOutputStream bOut = new ByteArrayOutputStream();
        ObjectOutputStream oOut = new ObjectOutputStream(bOut);
        oOut.writeObject(new Sub()); // prints '1 Base.Base() 2 Sub.Sub()'

        System.out.println("");
        ByteArrayInputStream bIn = new ByteArrayInputStream(bOut.toByteArray());
        ObjectInputStream oIn = new ObjectInputStream(bIn);
        oIn.readObject(); // prints '1 Base.Base()'
    }
}
class Base {

    private int baseField = getInt(1);

    public Base() {
        System.out.print("Base.Base() ");
    }

    protected int getInt(int i) {
        System.out.print(i + " ");
        return i;
    }
}

class Sub extends Base implements Serializable {

    private int subField = getInt(2);

    public Sub() {
        System.out.print("Sub.Sub() ");
    }
}
```

- статические поля не сериализуются в процессе нормальной сериализации (без соответствующих writeObject() и readObject()) кроме специального поля serialVersionUID. Это связано с тем, что сериализация предназначена для сохранения состояния **объекта** в виде массива байтов. Статические же поля не входят в состояние объекта, они принадлежат **классу**;

Dates, Numbers and Currency

- этот раздел не присутствовал в SCJP 1.4. Для SCJP 1.5 необходимо знать, для чего нужны и как работают нижеперечисленные классы:

классы

```
java.util.Date
java.util.Calendar
java.text.DateFormat
java.text.NumberFormat
java.util.Locale
```

- при форматировании даты информация о часах и минутах теряется:

пример

```
public class StartClass {
    public static void main(String[] args) throws Exception {
        Date date = new Date(1000000000000L);
        System.out.println(date); // (1) prints 'Sun Sep 09 05:46:40 MSD 2001'

        DateFormat dateFormat = DateFormat.getDateInstance(DateFormat.LONG);
        String dateRepresentation = dateFormat.format(date);
        System.out.println(dateRepresentation); // prints '9 Сентябрь 2001 г.'

        Date newDate = dateFormat.parse(dateRepresentation);
        System.out.println(newDate); // prints 'Sun Sep 09 00:00:00 MSD 2001'. Compare with (1)
    }
}
```

- если NumberFormat должен укоротить слишком длинное вещественное число, он не обрезает его, а округляет:

пример

```
public class StartClass {
    public static void main(String[] args) throws Exception {
        float f = 123.45678f;
        NumberFormat numberFormat = NumberFormat.getInstance();
        numberFormat.setMaximumFractionDigits(2);
        System.out.println(numberFormat.format(f)); // prints '123,46'
    }
}
```

Parsing, Tokenizing and Formatting

- этот раздел не присутствовал в SCJP 1.4. Для SCJP 1.5 необходимо знать основы работы с регулярными выражениями, правила форматированного вывода и возможности их применения в следующих классах:

классы

```
java.util.regex.Pattern
java.util.regex.Matcher
java.lang.String
java.util.Scanner
java.util.Formatter
```

пример на использование регулярных выражений

```
Given:
public class StartClass {
    public static void main(String[] args) throws Exception {
        String source = "ab34ef";
        String regex = "\\d*";
        Pattern p = Pattern.compile(regex);
        Matcher m = p.matcher(source);
        while (m.find()) {
            System.out.print(m.start() + m.group());
        }

    }
}
What is the result?
A. 234
B. 334
C. 334
D. 0123456
E. 01234456
F. 12334567
G. Compilation fails.
```

ответ

- при разбиении текста повторяющийся символ-разделитель предстает как пустая строка:

пример

```
public class StartClass {
    public static void main(String[] args) throws Exception {
        String source = "a 3 bc 23 d";
        Scanner scanner = new Scanner(source);
        scanner.useDelimiter("\\d");
        while (scanner.hasNext()) {
            System.out.println(">" + scanner.next() + "<");
        }
    }
}
```

результат

Тот же результат мы получим и, например, если будем использовать String.split():

пример

```
public class StartClass {
    public static void main(String[] args) {
        String source = "a 3 bc 23 d";
        String[] tokens = source.split("\\d");
        for (String token : tokens) {
            System.out.println(">" + token + "<");
        }
    }
}
```

- PrintWriter.format() работает так же, как PrintWriter.printf();
- преобразование булева выражения %b возвращает true для любых не-null или не-boolean аргументов:

пример

```
public class StartClass {
    public static void main(String[] args) {
        System.out.printf("%b\n", 1.2); // prints 'true'
        System.out.printf("%b", new Object()); // prints 'true'
    }
}
```

- в случае несоответствия переданного аргумента флагу форматирования возбуждается исключительная ситуация:

пример

```
public class StartClass {
    public static void main(String[] args) {
        System.out.printf("%c", "c"); // java.util.IllegalFormatConversionException: c != java.lang.String
    }
}
```

- флаг форматирования %s можно использовать с любым типом аргументов:

пример

```
public class StartClass {
    public static void main(String[] args) {
        System.out.printf("%s\n", 1); // prints '1'
        System.out.printf("%s\n", 2.3); // prints '2.3'
        System.out.printf("%s\n", true); // prints 'true'
        System.out.printf("%s", new Test()); // prints 'Test.toString()'
    }
}

class Test {
    public String toString() {
        return "Test.toString()";
    }
}
```

Generics

- из существования отношения IS-A между типами параметризованного класса не следует, что между объектами, параметризованными этим типами, также выполняется IS-A:

пример

```
public class StartClass {
    public static void main(String[] args) {
        ArrayList<Base> list = new ArrayList<Sub>(); // Compilation fails
    }
}

class Base {
}

class Sub extends Base {
}
```

- при определении объекта параметризованного типа необходимо, чтобы тип-параметр был одинаковым в обеих частях выражения:

пример

```
public class StartClass {
    public static void main(String[] args) {
        Set<List<String>> set1 = new HashSet<List<String>>(); // OK
        Set<List<String>> set2 = new HashSet<ArrayList<String>>(); // Compilation fails
    }
}
```

- нельзя объявлять перегруженные методы, различающиеся только типом параметризации:

пример

```
class Test {
    void test(List<Integer> list) { // Compilation fails
    }

    void test(List<String> list) { // Compilation fails
    }
}
```

- при объявлении маски для типа-параметра (например, <? extends MyClass>) всегда используется ключевое слово extends, даже если в выражении находится имя интерфейса, а не класса:

пример

```
class Test {
    void test1(List<? extends Serializable> list) { // OK
    }

    void test2(Set<? implements Serializable> set) { // Compilation fails
    }
}
```

- в маске для типа-параметра нельзя использовать примитивы и массивы:

пример

```
class Test {

    <T extends long> void test1(T t) { // Compilation fails
    }

    <T extends Object[]> void test2(T t) { // Compilation fails
    }
}
```

- при создании объекта типизированного класса нельзя использовать маску для типа-параметра:

пример

```
public class StartClass {
    public static void main(String[] args) {
        List<?> list1 = new ArrayList<? extends Object>(); // Compilation fails
        List<?> list2 = new ArrayList<?>(); // Compilation fails
    }
}
```

Из этого следует, например, что следующая попытка создать копию списка не скомпилируется:

invalid copying

```
class Test {
    void testInvalidCopy(List<?> list) {
        List<?> copy = new ArrayList<?>(list); // Compilation fails
    }
}
```

Надо делать так:

valid copying

- параметризовать можно любой метод, в том числе и конструктор (хотя это и бессмысленно):

пример

```
class Test {
    public <T> Test(T t) {
    }
}
```

- имя типа-аргумента может сопадать с именем класса, в этом случае оно перекрывается:

пример

```
class T {
    public <T> T(T t) { // t's type is constructor's type parameter, not T.class
    }
}
```

- знак ? нельзя использовать в маске типа-параметра при объявлении интерфейсов, классов или методов:

пример

```
class Test<?> { // Compilation fails
}

class MyList extends ArrayList<?> { // Compilation fails
}
```

- на объекте параметризованного класса, в статическом типе которого указана маска <?>, нельзя вызывать методы, принимающие аргумент типа-параметра. Единственное исключение – передача null:

пример

```
public class StartClass {
    public static void main(String[] args) {
        List<?> list = new ArrayList<String>();
        list.add("a"); // Compilation fails
        list.add(null); // OK

        Test<?> testObject = new Test<Integer>();
        testObject.test(1); // Compilation fails
        testObject.test(null); // OK
    }
}

class Test<T> {
    void test(T t) {
    }
}
```

- результат вызова метода параметризованного класса на объекте, в статическом типе которого указана маска <?>, и возвращающего значение типа-параметра можно безопасно приводить к Object:

пример

```
public class StartClass {
    public static void main(String[] args) {
        List<?> list = new ArrayList<String>();
        String firstElement = list.get(0); // Compilation fails
        Object secondElement = list.get(1); // OK

        Test<?> testObject = new Test<Integer>();
        Integer var1 = testObject.get(); // Compilation fails
        Object var2 = testObject.get(); // OK
    }
}

class Test<T> {
    T get() {
        return null;
    }
}
```

- при объявлении параметризированного метода объявление типа-параметра должно присутствовать перед типом возвращаемого значения:

пример

```
class Test {
    <T> void method1(T t) { // OK
    }

    void <T> method2(T t) { // Compilation fails
    }
}
```

- один и тот же параметризированный тип нельзя использовать в маске больше одного раза:

пример

```
class ObjectStore<T extends Comparable<T> & Comparable<String>> { // Compilation fails
}
```

- параметризированный метод не может переопределять непараметризированный:

пример

```
class Base {
    public void test() {
    }
}

class Sub extends Base {
    public <T> void test() { // Compilation fails
    }
```

```
}
```

- тип-параметр нельзя использовать в статическом контексте:

пример

```
class Test<T> {
    static void method(T t) { // Compilation fails
    }
}
```

- перечисления нельзя параметризовать:

пример

```
enum Color<T> { // Compilation fails
    WHITE, BLACK
}
```

- методы перечисления можно параметризовать:

пример

```
enum Shape {
    TRIANGLE, SQUARE;

    public <T> T test(T t) { // OK
        return t;
    }
}
```

- типизированный класс с определенным типом-параметром нельзя использовать с оператором instanceof:

пример

```
public class StartClass {
    public static void main(String[] args) {
        test1(new ArrayList<String>());
        test2(new ArrayList<String>());
    }

    private static void test1(List<?> list) {
        System.out.println(list instanceof LinkedList<String>); // Compilation fails
    }

    private static void test2(List<?> list) {
        System.out.println(list instanceof LinkedList); // OK
    }
}
```

- нельзя создавать массивы параметризованного типа или типа-параметра:

пример

```
class Test<T> {
    void test(T t) {
        Object array1 = new T[5]; // ?The component type of an array object may not be a type variable
        T[] array2 = {t}; // Even agregate initialization is not allowed
        Class<T>[] array3 = new Class<T>[5]; // ?The component type of an array object may not be parameterized type
    }
}
```

- можно создавать массивы параметризированного типа, используя маску <?>:

пример

```
public class StartClass {
    public static void main(String[] args) {
        List<?>[] array = new List<?>[2]; //OK
        List<Integer> list1 = new ArrayList<Integer>();
        list1.add(1);
        List<String> list2 = new ArrayList<String>();
        list2.add("2");
        array[0] = list1; // OK
        array[1] = list2; // OK
        Integer i = (Integer) array[0].get(0); // OK but explicit case is necessary
        String s = (String) array[1].get(0); // OK but explicit case is necessary
    }
}
```

- во время вызова параметризованного метода тип-параметр может быть выведен из аргумента, т.о. нет необходимости задавать его явно, хотя и можно. В случае явного задания надо сначала явно указать, на чем вызывается метод:

пример

```
public class StartClass {
    public static void main(String[] args) {
        String s = test("test"); // OK
        Integer i1 = <Integer>test(1); // Compilation fails
        Integer i2 = StartClass.<Integer>test(1); // OK
    }

    private static <T> T test(T t) {
        return t;
    }
}
```

- при использовании масок для типа-параметра можно задавать несколько условий. При этом после *erasure* типом параметра будет являться тип, использующийся в первом условии. Во всех условиях кроме первого должны использоваться интерфейсы:

пример

```
public class TestClass {
    private <T extends Interface1 & Interface2> void test1(T t) { // OK
    }

    private <T extends Class1 & Interface1 & Interface2> void test2(T t) { // OK
    }

    private <T extends Interface1 & Class1> void test3(T t) { // Compilation fails
    }
}

interface Interface1 {
}
```

```
interface Interface2 {
}

class Class1 {
}

class Class2 {
}
```

- попытка создания параметризованного класса, который явно или неявно наследуется от java.lang.Throwable, вызвает ошибку компиляции:

пример

```
class MyException<T> extends Exception { // Compilation fails
}
```

- тип-параметр можно указывать в объявлении throws, если используется маска с участием класса из иерархии дерева исключительных ситуаций (IS-A Throwable):

пример

```
class TestClass {
    <T extends Exception> void test(T t) throws T {
        throw t;
    }
}
```

- тип-параметр нельзя указывть в секции catch:

пример

```
class TestClass {
    <T extends Exception> void test(T t) {
        try {
        } catch (T e) {
        }
    }
}
```

Collections changes

- начиная с 1.5 LinkedList реализует интерфейс Queue;

Материалы

- JLS 3.0: http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf
- SCJP 1.5 guide
- 1.4 javadoc: http://java.sun.com/j2ee/1.4/docs/api/
- 1.5 javadoc: http://java.sun.com/j2se/1.5.0/docs/api/
- Java Puzzlers
- serialization specification: http://java.sun.com/j2se/1.5.0/docs/guide/serialization/spec/serialTOC.html
- Sun's code conventions: http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html
- Sun's internationalization tutorial: http://java.sun.com/docs/books/tutorial/i18n/index.html
- Regexp resource: http://www.regular-expressions.info/
- Sun's regexp tutorial: http://java.sun.com/docs/books/tutorial/essential/regex/intro.html
- Format string syntax: http://java.sun.com/j2se/1.5.0/docs/api/java/util/Formatter.html#syntax
- Sun's generics tutorial: http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf
- Generics tutorial from Angelika Langer: http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html