Java commands:

javac Test.java
java Test

Don't write .class in java command.
javac creates .class from .java file and then java command executes that .class file.

Variable Naming:

$ and _ are valid variables in java

Switch Statement

Data types supported by switch statements include the following:
■ int and Integer
■ byte and Byte
■ short and Short
■ char and Character
■ String
■ enum values

Note that boolean, long, float, double and their associated wrapper classes, are not supported by switch statements.
The syntax of default case is "**default**", not "case default".
Only constants can be used in switch cases

Primitive Data Type Rules:

Characters are initialized to Unicode '\u0000'
Characters can be used in array subscripts for index

The \u bits are in HEX, so the Unicode for 'a' is '\u0061'.

Order of chars in Unicode table:
048 – 057 : 0 –> 9
065 – 090: A -> Z
097 – 122: a -> z

Prefixes/Suffixes:

Binary: 0b	0b101
Hex: 0x	0xE
Long: L	20L
Float: f	10.5f

byte a = 40;
byte b = 50;

byte result = (byte) a + b; //Compile Error
cast operator has highest precedence

We can use class name as identifier but can't use reserved word.
String String = "test"; is valid statement

goto is a reserved word but is not used.

char cannot be implicitly converted to byte and short.
char c = 'a';
short s =c; //compile error
int i = c; //success

Numeric types cannot be converted to char type implicitly.
short s = 10;
char c = s; //compile error

A character can be explicitly casted to any primitive type other than short and byte.

Convert	Convert to:							
from:	boolean	byte	short	char	int	long	float	double
boolean	-	No	No	No	No	No	No	No
byte	No	-	Yes	Cast	Yes	Yes	Yes	Yes
short	No	Cast	-	Cast	Yes	Yes	Yes	Yes
char	No	Cast	Cast	-	Yes	Yes	Yes	Yes
int	No	Cast	Cast	Cast	-	Yes	Yes	Yes
long	No	Cast	Cast	Cast	Cast	-	Yes	Yes
float	No	Cast	Cast	Cast	Cast	Cast	-	Yes
double	No	Cast	Cast	Cast	Cast	Cast	Cast	-

A short CONSTANT can be assigned to a char only if the value fits into a char. short s = 1; byte b = s; => this will also not compile because although value is small enough to be held by a byte but the Right Hand Side i.e. s is a variable and not a constant. final short s = 1; byte b = s; => This is fine because s is a constant and the value fits into a byte. final short s = 200; byte b = s; => This is invalid because although s is a constant but the value does not fit into a byte.
Implicit narrowing occurs only for byte, char, short, and int.
Remember that it does not occur for long, float, or double. So, this will not compile: int i = 129L;

int x = (x=3) * 4; //valid statement.

i = i++
value of i will never change.
i++ increments the value of i but returns the old value of i. so the old value is assigned again.

Operators

a ^ b	Xor operator. False when both are same. Can work on binary and integral types.
a | b a & b	Bitwise operator. Can work on binary and integral types.
%	Can be used on float and doubles too
! && ||	Work with boolean only
~	Bitwise compliment. Works with integral types only.
a >> b	Right shift.
a << b	Left shift
a >>> b	Zero fill right shift

integral types means byte, short, int, long, and char

Note: There is no <<< operator in java.

The arithmetic operators *, / and % have the same level of precedence.
Assignment operator has least precedence of all operators.

Numeric Promotion

1. If two values have different data types, Java will automatically promote one of the values to the larger of the two data types.
2. If one of the values is integral and the other is floating-point, Java will automatically promote the integral value to the floating-point value's data type.
3. Smaller data types, namely byte, short, and char, are first promoted to int any time they're used with a Java binary arithmetic operator, even if neither of the operands is int.
4. After all promotion has occurred and the operands have the same data type, the resulting value will have the same data type as its promoted operands.

String

Important methods:

- indexOf(string, fromIndex)
- substring(startIndex, endIndex)
- concat(string)

String concatenation is left to right associative.

These are the six facts on Strings:

1. Literal strings within the same class in the same package represent references to the same String object.
2. Literal strings within different classes in the same package represent references to the same String object.
3. Literal strings within different classes in different packages likewise represent references to the same String object.
4. Strings computed by constant expressions are computed at compile time and then treated as if they were literals.
5. Strings computed at run time are newly created and therefore are distinct.
6. The result of explicitly interning a computed string is the same string as any preexisting literal string with the same contents. (So line 5 prints true.)

Replace returns the same object if there is no change.

The charAt(i) method throws IndexOutOfBoundsException type if passed invalid value.
It mostly throws StringIndexOutOfBoundsException but it is free to choose.
Note: Always choose IndexOutOfBoundsException if it is in options of question

StringBuilder

StringBuilder is not immutable

Important methods:

- *insert()*
- *delete(from, to)*
- *deleteCharAt()*
- *reverse()*

- ***ensureCapacity(int)***
  increases the capacity to the given number if needed.

- ***setLength(int)***
  if lesser than current length, will truncate to new length.
  If greater than current length, will append \u0000 upto the new length.

- ***trimToSize(int)***
  if capacity is larger than size, will try to reduce capacity.

StringBuilder has contructor which take capacity as integer and another which takes string as value

StringBuffer is slow because it is thread safe

Methods not supported by StringBuilder:

- Clear()
- Empty()
- removeAll

- deleteAll()

Arrays:

int [] arr = new int[10];

initialized array:
int [] arr = new int[] {10, 20};

or

int [] arr = {10, 20}; //anonymous array as no size and type is specified

int [] arr1, arr2; //both are array type
int arr[], arr2; //only first is array

string [] arr = new [] String {“a”}; //Compile error. [] cannot be before Type.

If you give the elements explicitly you can't give the size.

Arrays.sort(arr); //comparator is optional. Also from and to index can be given
Arrays.binarySearch(arr, number);

if element is not found, the index of the position where element can be placed is negated and 1 is subtracted and then returned.

Collections.binarySearch can work on strings array also

VarArgs

Void someMethod(String …args)

Must be the last parameter of a method

Null can be passed as a parameter to var args.

Multi-dimensional arrays

int[][] vars1; // 2D array
int vars2 [][]; // 2D array
int[] vars3[]; // 2D array
int[] vars4 [], space [][]; // a 2D AND a 3D array

```
int[][] differentSize = {{1, 4}, {3}, {9,8,7}};
int [][] args = new int[4][];

for (int[] inner : twoD) {
        for (int num : inner)
                System.out.print(num + " ");
                System.out.println();
        }
}
```

ArrayList:

import java.util.ArrayList;

implements RandomAccess

boolean add(E element)

void add(int index, E element)

add always return true. Some other classes in the family use this return type.

add()

remove()

set()

clear()

contains

equals()

Dead code:

In case of if else, dead code will be warning and not an error.

But in case of loops it will be unreachable code error given by compiler.

Wrapper Classes

int p = Integer.parseInt(“123”);

Integer i = Integer.valueOf(“123”);

2 constructor for each wrapper class:

1. Takes same type primitive
2. Takes String

== operator compares references, not primitive value.

Short k = new Short(9); //Compile error because 9 is considered an int.
short s = 9; //Although 9 is considered an int, but since it will fit into short, it is casted implicitly.

The equals method of all wrapper classes first checks if the two object are of same class or not. If not, they immediately return false.

All the wrapper objects are immutable. When you do i++, what actually happens is something like this:

i = new Integer(i.intValue() + 1); As you can see, a new Integer object is

assigned back to i.

However, to save on memory, Java 'reuses' all the wrapper objects whose values fall in the following ranges:

All Boolean values (true and false)

All Byte values

All Character values from \u0000 to \u007f (i.e. 0 to 127 in decimal)

All Short and Integer values from -128 to 127

Character class doesn’t provide above 2 methods

Collections

list.toArray(T [] arr)

If the result fits in arr, it will be copied to arr and returned. Otherwise new array will be created and returned.

Specify 0 sized array to make sure new array is returned

Arrays.asList()

Returns fixed sized arrays and the arraylist created is backed by the given array. Change in one will update the other.

Collections.sort()

Both array and arraylist are ordered

Java 8 Date Time

Import java.time.*;

LocalDate

LocalDate date1 = LocalDate.of(2015, Month.JANUARY, 20);

LocalDate.now()

plusDays()

plusWeeks()

plusMonths

plusYears

minus versions for all

LocalTime

LocalDateTime

LocalDateTime ldt = LocalDateTime.of(ld, lt);

2015-01-20T12:45:18.401

Last value is nanoseconds, not milliseconds

	Old way	New way (Java 8 and later)
Importing	`import java.util.*;`	`import java.time.*;`
Creating an object with the current date	`Date d = new Date();`	`LocalDate d = LocalDate.now();`
Creating an object with the current date and time	`Date d = new Date();`	`LocalDateTime dt = LocalDateTime.now();`

Creating an object representing January 1, 2015	`Calendar c = Calendar.getInstance();`
	`c.set(2015, Calendar.JANUARY, 1);`
	`Date jan = c.getTime();`
	or
	`Calendar c = new GregorianCalendar(2015, Calendar.JANUARY, 1);`
	`Date jan = c.getTime();`
		`LocalDate jan = LocalDate.of(2015, Month.JANUARY, 1);`

Creating January 1, 2015 without the constant	`Calendar c = Calendar.getInstance();`
	`c.set(2015, 0, 1);`
	`Date jan = c.getTime();`
		`LocalDate jan = LocalDate.of(2015, 1, 1)`

Date and time classes are immutable.

Period

Period annually = Period.ofYears(1); // every 1 year
Period quarterly = Period.ofMonths(3); // every 3 months
Period everyThreeWeeks = Period.ofWeeks(3); // every 3 weeks
Period everyOtherDay = Period.ofDays(2); // every 2 days
Period everyYearAndAWeek = Period.of(1, 0, 7); // every year and 7 days

date.plus(period)

datetime.plus(period)

Cannot chain methods when creating periods. It is compile successfully however only the last statement in chain is effective.

DateTimeFormatter df = DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);

DateTimeFormatter df = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.SHORT);

DateTimeFormatter df = DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT);

DateTimeFormatter df = DateTimeFormatter.ofPattern("MM/dd/yyyy");

LocalDate.parse("10/10/2010")

localDt.format(df)

System.out.println(ld.format(DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT)));

11/5/16

System.out.println(ld.format(DateTimeFormatter.ofLocalizedDate(FormatStyle.LONG)));

November 5, 2016

Static

Static methods cannot hide instance methods and vice versa.

//both are legal. Method name and return type should be in order only.

Public final void foo();

Final public void foo();

Top level classes cannot be marked static or private.

Test t = null;

t.count //will not throw exception if count is static. This is because java will call count on the class, not on the object instance.

If static final variables are not initial, there will be compile error.

Static final variables must be set exactly one. It will give compile error, when they are set multiple times.

Instance final members also need to be set exactly once. Same rule as above.

import static class.*;

static import class.*; //does not compile

The word import should come before static.

Static imports doesn't import the class itself.

Static imports are for importing static members of classes. Normal imports are for importing classes.

Static imports can use wildcard (*)

Parameters names cannot be supplied with static import method name.

Access Modifiers

Private

Public

Default (only classes in same package can access) (Package Private)

Protected (classes in same package and for children outside package)

Default access is more restrictive than protected.

Always check for case sensitiveness

Protected rules apply under two scenarios:

1. A member is used without referring to a variable.
   In this case, we are taking advantage of inheritance and protected access is allowed.
2. A member is used through a variable.

In this case, the rules for the reference type of the variable are what matter. If it is a subclass, protected access is allowed.

It is legal to import a class in the same file in which class is declared. :O

Remember that you can never access a class that is defined in the default package (i.e. the package with no name) from a class in any other package.

Overloading Methods

void test(int []arr);

void test(int …arr) //will not compile

Both are same, so conflict

For overloading, the signatures must be same.

void test();

void test(int …arr) //will compile successfully

If argument is primitive, and primitive parameter method is present, it will have high priority.
Autoboxing is only done when no other option is available.

Java always picks the most specific overloaded method.

TABLE 4.4 Order Java uses to choose the right overloaded method

Rule	Example of what will be chosen for `glide(1,2)`
Exact match by type	`public String glide(int i, int j) {}`
Larger primitive type	`public String glide(long i, long j) {}`
Autoboxed type	`public String glide(Integer i, Integer j) {}`
Varargs	`public String glide(int... nums) {}`

Java will not convert to large primitive and autobox at the same time.

Void test(Long l) {}

test(12); //not compile

test(12L) //will compile

if null is passed, most specific method is selected.

If there are more than 1 specific, compile time errors occurs.

Constructors

Call to constructor using this() should be the first statement in the constructor.

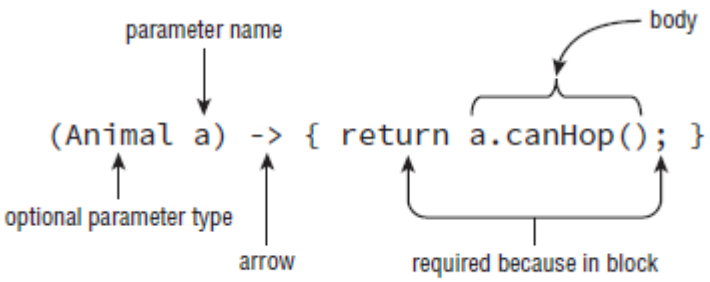
Default constructor is added by compiler if no constructor is provided.

The access type of a default constructor is same as the access type of the class. Thus, if a class is public, the default constructor will be public.

Lambdas

Functional programming uses lambda expressions to write code

Work with interfaces that have only one method.



Parentheses can be omitted if there is a single parameter and its type is not explicitly mentioned.

return keyword can be omitted ONLY if curly braces are NOT used.

```
3: print(() -> true);                                    // 0 parameters
4: print(a -> a.startsWith("test"));                     // 1 parameter
5: print((String a) -> a.startsWith("test"));            // 1 parameter
6: print((a, b) -> a.startsWith("test"));                // 2 parameters
7: print((String a, String b) -> a.startsWith("test"));  // 2 parameters
```

Parameters cannot be skipped (case 3)

(a, b) -> { int a = 0; return 5;} // DOES NOT COMPILE

a cannot be re-declared

Predicate is generic interface present in java.util.function

```
private static void print(List<Animal> animals, Predicate<Animal>
checker) {
   for (Animal animal : animals) {
     if (checker.test(animal))
        System.out.print(animal + " ");
    }
    System.out.println();
}
```

print(animals, a -> a.canHop());

ArrayList declares a removeIf() method that takes a predicate.

Inheritance

super() calls the parent constructor but super is used to access the members

super is a keyword and not an attribute.

super.super is not allowed.

Overriding

Rules for overriding methods:

1. Method in child class must have same signatures.
2. Method in child class must be at least as accessible or more accessible than the method in parent class.
3. Method in child class should not throw check excerption that is new or broader than the class of any exception thrown in the parent class. Child may eliminate parent's method exception.
4. If method returns value, it must be same or subclass of the method return type in the parent class.

Private methods cannot be overridden since they are not visible in child class. Child class can define its own method.

A method signature is the method name and the number and type of its parameters. Return types and thrown exceptions are not considered to be a part of the method signature.

If method is overridden, child version will be used always even if it is accessed in the parent class. This is because it is replaced with the child method.

Final methods cannot be overridden.

Hiding static methods

First 4 rules from method riding are also applicable

+

If method is static in parent, then it should be static in the child class. If method is instance in parent class, it cannot be static in child class.

Main method can be declared static like other static methods.

Hiding variables

Reference determines which variable will be accessed.

Both public and private variables **cannot** be overridden but hidden

Abstract Classes

Only an abstract class can have abstract methods.

Abstract class or method cannot be marked as final

Abstract methods cannot be marked **private**

Interfaces

Interfaces are abstract by default

Only final variables can be added

All interfaces must be public

Interfaces doesn't extend java.lang.object

Making interface as private, protected or final will trigger compile error.

All non-default methods in interfaces must be public

Making non-default methods private, protected or final will trigger compile error

Since all methods in interface are public, a class implementing that interface must declare methods as public. Otherwise it will be compile error because class will be trying to make them default access.

If 2 interfaces contains same constants, then class can still implement those interfaces. However using constant without interface name will result in compile time error due to ambiguity.

Unlike a class, an interface can extend from multiple interfaces.

If 2 interfaces contain a method with same signatures but different return type, class implementing both interfaces at a time will not compile due to conflict between 2 methods.

Default methods have implementation in the interface which can be overridden by the implementing class. Default methods must provide implementation in interface.

A default method cannot override a method from java.lang.Object.

If the class implements 2 interfaces that have same method signatures, and at least one of the interface has default implementation, compiler will give error. The class has to provide its own implementation to fix the error or remove the interface.

Functional Interfaces can contain more methods but only one abstract method

Static methods in interfaces now also possible. They must be public.

All non-final, non-static and non-private methods are virtual by default.

A class that implements two interfaces containing static methods with the same signature will still compile at runtime, because the static methods are not inherited by the subclass and must be accessed with a reference to the interface name.

Contrast this with the behavior you saw for default interface methods.

Child interface can override parent interface methods just like classes.

A class can implement parent and child interfaces both at the same time. It will not generate any conflict.

Finalize method doesn't take any parameters.

Casting is only allowed between parent child classes.

String o = (String) new TestClass(); //does not compile

Exceptions:

Throwable is subclass of object

Runtime exceptions can be caught but they are not required.

Checked exceptions must be caught or declared.

try statement must have curly brackets even if there is single statement in it.

Runtime exception extends **RuntimeException**.

Runtime Exceptions:

NullPointerException

ClassCastException

ArithmeticException

NumberFormatException

SecurityException: Thrown by JVM securitymanager upon security violation

IllegalStateException: Thrown by application e.g when some method is invocated on a thread that is not in appropriate state.

Checked Exception:

FileNotFoundException

IOException

Errors:

ExceptionInInitializerError: Thrown when there is exception in static initializer

NoClassDefFoundError: Class present at compile time but not at runtime

StackOverflowError:

NumberFormatException and IOException are not thrown by JVM. They are thrown by wrapper/utility classes in java.

Even if you are throwing custom runtime exception, it is not needed to catch or declare it.

For checked exceptions, catch statements must declare only those check exception which are thrown in the code. Compiler doesn't allow unreachable catch blocks.

If exception is thrown in finally block, that exception will be thrown and any exception thrown from try or catch will be ignored.

Errors are allowed to be handled or declared.

If the exception object thrown is null, jvm will generate NullPointerException.

exception.toString() doesn't print exception stack. The output format is

<class name>: <message>

e.g java.lang.Exception: some error occurred.

When no main method is found, JVM throws java.lang.NoSuchMethodError

Use of uninitialized local variables in any statement will create compile error.

Remember to count exception object when counting total number of objects created.

Java Exceptions is a mechanism ..

1. that you can use to determine what to do when something unexpected happens.
2. for logging unexpected behavior

The expression `x>a?y>b?y:b:x>z?x:z;` should be grouped as -

```
x > a  ? (y>b ? y : b)  :  (x>z ? x : z);
```

Labelled loops:

If label is put on statement other than loop label, it cannot be used in break.

```
public static void main(String[] args) {
        int c = 0;
        JACK: while (c < 8) {
                JILL: {
                        System.out.println(c);
                        if (c > 3)
                                break JACK;
                        else
                                c++;
                }
        }
}
```

Break JILL will not break the loop but it will compile.

A break statement with no label attempts to transfer control to the innermost enclosing switch, while, do, or for statement; this statement, which is called the break target, then immediately completes normally. If no switch, while, do, or for statement encloses the break statement, a compile-time error occurs.
A break statement with label Identifier attempts to transfer control to the enclosing labeled statement that has the same Identifier as its label; this statement, which is called the break target, then immediately completes normally. In this case, the break target need not be a while, do, for, or switch statement.

HashCode

The hashCode() method of objects is used when you insert them into a HashTable, HashMap or HashSet. It is used as key in hashing.

Rules:

1. If object1 and object2 are equal according to their equals() method, they must also have the same hash code.
2. If object1 and object2 have the same hash code, they do NOT have to be equal too.

Number:

The abstract class Number is the superclass of classes BigDecimal, BigInteger, Byte, Double, Float, Integer, Long, and Short.

Important Points:

For encapsulation, data members must be private. Although setters are allowed but they are not required.

String replace method replaces all instances of given substring :D

Child class cannot access private methods of parent using super()
hashCode can be overridden

synchronized can only be applied to methods or blocks.

List of final classes:

- String
- StringBuilder
- StringBuffer
- All wrapper class (Integer, Long …)
- java.lang.System

All non-private method calls are polymorphic.

```
int i = 3;
int [][][] arr = new int[i][i=4][i];
```

```
System.out.println(arr.length + ", " + arr[0].length + ", " + arr[0][0].length);
```
Output: 3, 4, 4

```
int i = 4;
int result = i + (i=3) + i;
System.out.println(result);
```
Output: 10

```
int k = 1;
k += (k = 4) * (k + 2);
System.out.println(k);
```
Output: 25

Object can be instantiated:
Object o = new Object();

System.out.println('a' + 1);
Output: 98
Both operands are promoted to integer.
System.out.println(100/10.0);
Output: 10.0

interface Flyer{ }
class Bat { }

if(b instanceof Flyer) System.out.println("f is a Bird");

Note that there is no compilation issue with b instanceof Flyer because Flyer is an interface and it is possible for b to point to an object of a class that is a sub class of Bat and also implements Flyer. So the compiler doesn't complain. If you make Bat class as final, b instanceof Flyer will not compile because the compiler knows that it is not possible for b to point to an object of a class that implements Flyer.

Classes:

A class or interface type T will be initialized at its first active use, which occurs if: T is a class and a method actually declared in T (rather than inherited from a superclass) is invoked. T is a class and a constructor for class T is invoked, or U is an array with element type T, and an array of type U is created. A non-constant field declared in T (rather than inherited from a superclass or superinterface) is used or assigned. A constant field is one that is (explicitly or implicitly) both final and static, and that is initialized with the value of a compile-time constant expression . Java specifies that a reference to a constant field must be resolved at compile time to a copy of the compile-time constant value, so uses of such a field are never active uses. All other uses of a type are passive A reference to a field is an active use of only

the class or interface that actually declares it, even though it might be referred to through the name of a subclass, a sub interface, or a class that implements an interface.

An instance member belongs to a single instance, not the class as a whole. An instance member is a member variable or a member method that belongs to a specific object instance. All non-static members are instance members.

transient and volatile modifiers are only valid for member field declarations. abstract and native are only valid for member methods.
native method cannot be abstract.

```
String s = "Hello";
String s2 = "He" + "llo";
System.out.println(s == s2);
```

Although there is more to it that the following sequence, for the purpose of exam, this is all you need to know:
1. Static blocks of the base class (only once, in the order they appear in the class).
2. Static blocks of the class.
3. Non-static blocks of the base class.
4. Constructor of the base class.
5. Non-static blocks of the class.
6. Constructor of the class.

ArrayCopy

public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)

Parameters:
src - the source array.
srcPos - starting position in the source array.
dest - the destination array.
destPos - starting position in the destination data.
length - the number of array elements to be copied.
Throws:
IndexOutOfBoundsException - if copying would cause access of data outside array bounds.
ArrayStoreException - if an element in the src array could not be stored into the dest array because of a type mismatch.
NullPointerException - if either src or dest is null.

Note that if the src and dest arguments refer to the same array object, then the copying is performed as if the components at positions srcPos through srcPos+length-1 were first copied to a temporary array

with length components and then the contents of the temporary array were copied into positions destPos through destPos+length-1 of the destination array.

ASCII table:

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`	128	80	Ç	160	A0	á	192	C0	└	224	E0	α
1	01	Start of heading	33	21	!	65	41	A	97	61	a	129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
2	02	Start of text	34	22	"	66	42	B	98	62	b	130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
3	03	End of text	35	23	#	67	43	C	99	63	c	131	83	â	163	A3	ú	195	C3	├	227	E3	π
4	04	End of transmit	36	24	$	68	44	D	100	64	d	132	84	ä	164	A4	ñ	196	C4	─	228	E4	∑
5	05	Enquiry	37	25	%	69	45	E	101	65	e	133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
6	06	Acknowledge	38	26	&	70	46	F	102	66	f	134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
7	07	Audible bell	39	27	'	71	47	G	103	67	g	135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
8	08	Backspace	40	28	(	72	48	H	104	68	h	136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i	137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j	138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k	139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l	140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m	141	8D	ì	173	AD	¡	205	CD	═	237	ED	ø
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n	142	8E	Ä	174	AE	«	206	CE	╬	238	EE	ε
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o	143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
16	10	Data link escape	48	30	0	80	50	P	112	70	p	144	90	É	176	B0	░	208	D0	╨	240	F0	≡
17	11	Device control 1	49	31	1	81	51	Q	113	71	q	145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
18	12	Device control 2	50	32	2	82	52	R	114	72	r	146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
19	13	Device control 3	51	33	3	83	53	S	115	73	s	147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
20	14	Device control 4	52	34	4	84	54	T	116	74	t	148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u	149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v	150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
23	17	End trans. block	55	37	7	87	57	W	119	77	w	151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
24	18	Cancel	56	38	8	88	58	X	120	78	x	152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
25	19	End of medium	57	39	9	89	59	Y	121	79	y	153	99	Ö	185	B9	╣	217	D9	┘	249	F9	∙
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z	154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{	155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|	156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}	157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~	158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	⌂	159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	

Practice Questions

```
1: package animal;
2: public class Frog {
3:   protected void ribbit() { }
4:   void jump() { }
5: }
```

```
1: package other;
2: import animal.*;
3: public class Tadpole extends Frog {
4:   public static void main(String[] args) {
5:     Tadpole t = new Tadpole();
6:     t.ribbit();
7:     t.jump();
8:     Frog f = new Tadpole();
9:     f.ribbit();
10:    f.jump();
11:   } }
```

- A. 5
- B. 6 ✗
- C. 7 ✗
- D. 8
- E. 9

Explanation

Close

Answer: C;E;F

`jump()` has default (package private) access, which means it can only be accessed from the same package. `Tadpole` is not in the same package as `Frog`, causing lines 7 and 10 to give a compiler error. `ribbit()` has protected access, which means it can only be accessed from a subclass reference or in the same package. Line 6 is fine because `Tadpole` is a subclass. Line 9 does not compile because the variable reference is to a `Frog`. This is the trickiest question you can get on this topic on the exam.

41. Which of the following are true about ternary operators? (Choose all that apply)

- A. The left-hand side of the expression must evaluate to a `boolean` expression or numeric value of 0 or 1 at runtime.
- ✗ B. The data types of the two expressions on the right-hand side of the equation must match or be able to be numerically promoted to one another.
- ✗ C. All ternary operators can be rewritten as `if-then-else` statement.
- ✗ D. If the expression on the left-hand side evaluates to `true`, then the rightmost expression will not be evaluated.
- E. The terms must be compile-time constant values.

Explanation

Close

Answer: C;D

Option A is incorrect, because only `boolean` expressions are allowed, and `0` and `1` cannot be parsed as `boolean` values in Java. Option B is incorrect, because the data types of the expression can be completely different, such as `String` and `int`. That said, if the ternary operator is used on the right-hand side of an assignment operator, then the values do need to match or be able to be numerically promoted to one another. That limitation is introduced by the assignment operator, though, not the ternary operator. Option C is correct because ternary operators are just glorified `if-then-else` statements. As of Java 7, option D is correct, since only one of the right-hand side expressions of the ternary operation is evaluated at runtime. Option E is incorrect, because this phrase relates to the data type of `case` statement values within `switch` statements.

48. Which of the following compile? (Choose all that apply)

- A. `List<Integer> l1 = new ArrayList();`
- ✗ B. `List<Integer> l2 = new ArrayList<>();`
- ✗ C. `List<Integer> l3 = new ArrayList<Integer>();`
- ✗ D. `List<> l4 = new ArrayList<Integer>();`
- E. `List<Integer> l5 = new List<Integer>();`
- F. `ArrayList<int> l6 = new List<int>();`

Explanation Close

Answer: A;B;C

Option A compiles since it is allowed to use generics on just one side in a declaration. Option B compiles using the diamond operator. Option C is a longer form of option B; it spells out the generics type. Option D does not compile because the diamond operator is only allowed on the right side. Option E does not compile because `List` is only allowed on the left side since it's an interface rather than a concrete type. Option F does not compile because primitives are not allowed to be `ArrayList` types. Autoboxing only works when working with the `ArrayList`, not when creating it.

49. Which of the following statements about objects and references are true in Java? (Choose all that apply)

- ✗ A. By explicitly casting an object to a subclass, you can gain access to methods and variables that were hidden from access.
- B. If you compare two distinct references to the same object with ==, the result will evaluate to `false`.
- C. All objects in memory can be referenced using a reference of type `java.lang.Object`.
- D. When you create an object in Java, you get direct access to the object in memory.
- ✗ E. Removing all references to an object deletes the object from memory.

Explanation

Close

Answer: A;C

It is true that casting an object to a subclass may grant access to new methods and variables that were not previously available in the superclass reference, so option A is correct. Option B is incorrect—comparing two references with == that point to the same object results in a `true` value. All objects inherit from `java.lang.Object`, so all objects in memory can be access with a reference of that type, so option C is true. All objects are accessed via a reference in Java, so option D is incorrect since you can never get direct access. Finally, removing all references to an object makes the object eligible for garbage collection, but the rules of garbage collection do not guarantee when this deletion from memory will occur.

55. What is the result of the following code snippet? (Choose all that apply)

```
3: final int movieRating = 4;
4: int badMovie = 0;
5: switch(badMovie) {
6:   case 0:
7:   case badMovie: System.out.println("Awful"); break;
8:   case 4:
9:   case movieRating:  System.out.println("Great"); break;
10:   default:
11:   case (int)'a':
12:   case 1*1: System.out.println("Too be determined"); break;
13: }
```

- A. The code will not compile because of line 6.
- ✗ B. The code will not compile because of line 7.
- C. The code will not compile because of line 8.
- D. The code will not compile because of line 9.
- E. The code will not compile because of line 10.
- F. The code will not compile because of line 11.

Explanation Close

Answer: B;C;D

Clearly, there are some problems with this `switch` statement! First off, lines 6 and 7 both contain the same value of `0`, but `badMovie` is not a `final` constant variable; therefore, it is line 7 that will not compile, so option B is correct. Note that if `badMovie` was made a `final` constant variable with the same value of `0`, then option A would be considered correct too. Since it is not `final`, though, option A is not correct.

Next, lines 8 and 9 both have the same value of `4`, and they are both constant values. Therefore, they will cause the code not to compile, so options C and D are both correct. If either of the lines were removed, then the code would compile.

Finally, options E and F, although expressed a little oddly, both evaluate to constant integer values at compile

40. Which of the following are correct differences between abstract class and interfaces? (Choose all that apply)

- A. Abstract classes can declare static methods, but interfaces cannot.
- B. Interfaces can declare `public static final` variables, but abstract classes cannot.
- C. An interface may extend another interface but not an abstract class.
- D. Interfaces support multiple inheritance, whereas abstract classes support single inheritance.
- E. An abstract class can be declared in a separate file, whereas an interface must be bundled within an existing class file.
- ✕ F. Only abstract classes can include abstract methods.

Answer: C;D

As of Java 8, an interface can include static methods, so option A is incorrect. Both interfaces and abstract classes can declare `public static final` variables, so option B is incorrect. Interfaces can only extend other interfaces, whereas classes can only extend other classes, so option C is correct. A class may implement multiple interfaces but only extend one class, so option D is correct. Interfaces can be in their own file with no class defined, so option E is incorrect. Finally, both abstract classes and interfaces can define abstract methods, so option F is incorrect.

27. What is the result of the following code snippet?

```
3: String tiger = "Tiger";
4: String lion = "Lion";
5: final String statement = 250 > 338 ? lion : tiger = " is Bigger";
6: System.out.println(statement);
```

A. `Tiger`

Explanation Close

Answer: F

The code does not compile, because the assignment operator has the highest order of precedence in this expression. It may be helpful to see what the compiler is doing by adding optional parentheses:

```
final String statement = (250 > 338 ? lion : tiger) = " is Bigger";
```

This expression is then reduced to:

```
final String statement = "tiger" = " is Bigger";
```

This expression is invalid, as the left-hand side of the second assignment operator is not a variable, so the

39.

Which of the following statements can be inserted in the blank so that the code will compile successfully? (Choose all that apply)

```
public interface WalksOn4Legs {}
public abstract class Mammal {
  public int numberOfOffspring;
}
```

Explanation Close

Answer: B;D;F

Option A is incorrect, since `Mammal` is declared abstract and cannot be instantiated. Likewise, `WalkOn4Legs` is an interface, which is inherently abstract; therefore, option C is incorrect. Option B is correct as `Antelope` is a subclass of `Mammal` and as a polymorphic parameter can be passed without issue to a method accepting a reference of type `Mammal`. Option D is correct, since `Object` is a superclass of `Mammal` and the compiler allows the cast, even though this would produce a `ClassCastException` at runtime. Option E is incorrect, since `String` is not a superclass of `Mammal` and the compiler detects this and throws a compilation error. Finally, option F is correct—a `null` value can always be passed as a reference. Although this will compile without issue, it will produce a `NullPointerException` at runtime.

25. What is the result of the following?

```
List<String> hex = Arrays.asList("30", "8", "3A", "FF");
Collections.sort(hex);
int x = Collections.binarySearch(hex, "8");
int y = Collections.binarySearch(hex, "3A");
int z = Collections.binarySearch(hex, "4F");
 System.out.println(x + " " + y + " " + z);
```

- A. 0 1 –2
- B. 0 1 –3
- C. 2 1 –2
- D. 2 1 –3
- E. None of the above
- F. The code doesn't compile.

Explanation Close

Answer: D

After sorting, `hex` contains `[30, 3A, 8, FF]`. Remember that numbers sort before letters and strings sort alphabetically. This makes 30 comes before 8. A binary search correctly finds 8 at index 2 and 3A at index 1. It cannot find 4F but notices it should be at index 2. The rule when an item isn't found is to negate that index and subtract 1. Therefore we get –2–1, which is –3.