* Identifiers must start with a letter or dolor ($) or underscore (\_). Identifiers cannot start with a digit!
* After the first character, identifiers can contain any combination of letters, currency characters, connecting characters, or numbers.
* In practice, there is no limit to the number of characters an identifier can contain.
* You can't use a Java keyword as an identifier.

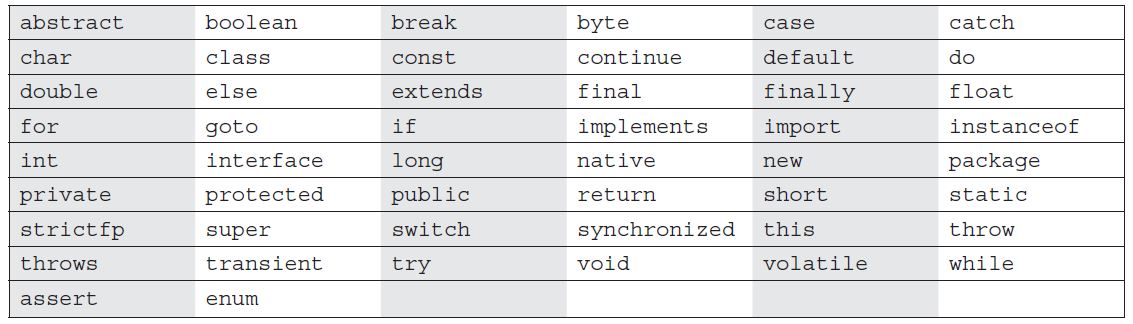
int \_a;

int $c;

int \_\_\_\_\_\_2\_w;

int \_$;

int this\_is\_a\_very\_detailed\_name\_for\_an\_identifier;



* There is never a case where an access modifier can be applied to a local variable, only one modifier that can ever be applied to local variable final.

class Foo {

void doStuff() {

private int x = 7;

}

}

* If two classes are in two different package and if any method or variable declared other than public using has relation we can’t access them. Same package we can access.
* There is never a case where an access modifier can be applied to a local variable, so watch out for code like the following:

class Foo {

void doStuff() {

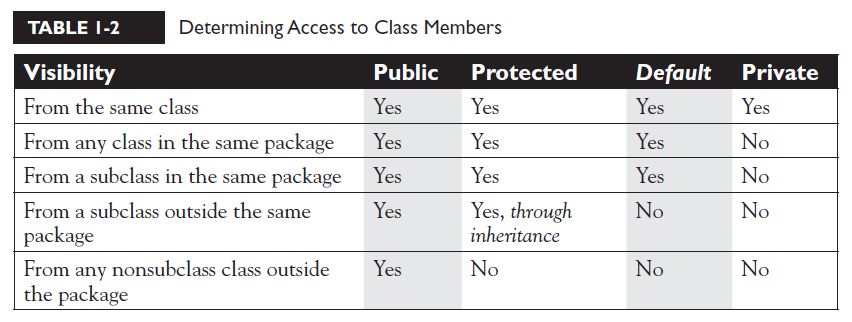
private int x = 7;

this.doMore(x);

}

}

there is only one modifier that can ever be applied to local variables—final.



* Check method parameter variable name and method local variable name and loop variable names if this are same code won’t compile.
* We can’t define abstract methods as final or private but we can define concrete methods in abstract class. If you define private concrete method in abstract class there no use with that method but you can use this method in another concrete method.
* If you define final concrete method in abstract class we can’t override in derived class.
* synchronized modifier can be applied only to methods—not variables, not classes, just methods. Constructor can’t be synchronized.
* The native modifier indicates that a method is implemented in platform-dependent code, often in C. native can be applied only to methods—not classes, not variables, just methods.
* strictfp is used for classes and methods not for the variables.
* Var-arg limits The var-arg must be the last parameter in the method's signature, and you can have only one var-arg in a method.
* Constructors can't be marked static or final or abstract

int[5] scores;

* The preceding code won't compile. Remember, the JVM doesn't allocate space until you actually instantiate the array object. That's when size matters.
* Don't forget that a static method can't directly access an instance variable.
* Classes can have only public or default access.
* A class with default access can be seen only by classes within the same package.
* A class with public access can be seen by all classes from all packages.
* Classes can also be modified with final, abstract, or strictfp.
* A class cannot be both final and abstract.
* A single abstract method in a class means the whole class must be abstract.
* It must not declare any new checked exceptions for an implementation method.
* It must not declare any checked exceptions that are broader(larger/parent) than the exceptions declared in the interface method or it can ignore the Exception.
* It may declare runtime exceptions on any interface method implementation regardless of the interface declaration.

**Note:** When taking the exam, verify that interface and class declarations are legal before verifying other code logic. Determine class visibility before determining member visibility.

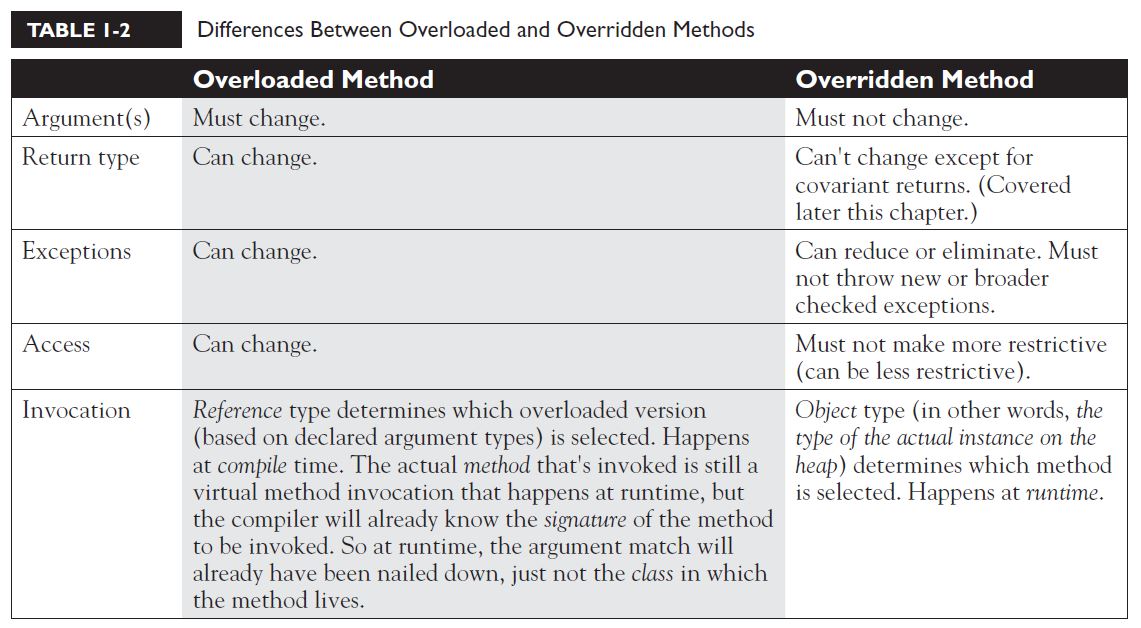
* Local variables don't get default values, so they must be initialized before use.
* The synchronized modifier applies only to methods and code blocks.
* transient and volatile modifier applies only to instance variables.
* An enum declared outside a class must NOT be marked static, final, abstract, protected, or private.
* enums can contain constructors, methods, variables, and constant-specific class bodies.
* enum constructors can NEVER be invoked directly in code. They are always called automatically when an enum is initialized.
* The semicolon at the end of an enum declaration is optional. These are legal:
* enum Foo { ONE, TWO, THREE}
* enum Foo { ONE, TWO, THREE};
* MyEnum.values() returns an array of MyEnum's values.
* The overriding method must NOT throw checked exceptions that are new or broader than those declared by the overridden method. For example, a method that declares a FileNotFoundException cannot be overridden by a method that declares a SQLException, Exception, or any other nonruntime exception unless it's a subclass of FileNotFoundException.
* Public void m1(Parent p)

Public void m1(Chaild c)

Parent p = new Chaild();

m1(p)call first one because overloaded method decided at compile time new Chaild runtime.

* overridden version of the method to call is decided at runtime based on object type, but overloaded version of the method to call is based on the reference type of the argument passed at compile time.



class Animal { }

class Dog extends Animal { }

class DogTest {

public static void main(String [] args) {

Animal animal = new Animal();

Dog d = (Dog) animal; // compiles but fails later(ClassCastException)

Dog d = animal;//compiler error

Animal animal2 = new Dog();

Dog d1 = (Dog) animal2;// work’s perfectly you can access the Dog specific methods and variables.

Or

((Dog) animal2).doDogStuff();

}

}

* All the compiler can do is verify that the two types are in the same inheritance tree, so that depending on whatever code might have come before the downcast, it's possible that animal is of type Dog. The compiler must allow things that might possibly work at runtime. However, if the compiler knows with certainty that the cast could not possibly work, compilation will fail. The following replacement code block will NOT compile:

Animal animal = new Animal();

Dog d = (Dog) animal;

String s = (String) animal; // animal can't EVER be a String

* Unlike downcasting, upcasting works implicitly

Dog d = new Dog();

Animal a1 = d; // upcast ok with no explicit cast (we can call Animal and overridden methods)

Animal a2 = (Animal) d;

Using a1 or a2 we can call overridden methods or Animal methods and variables.

* Both of the previous upcasts will compile and run without exception, because a Dog IS-A(n) Animal, which means that anything an Animal can do, a Dog can do. A Dog can do more, of course, but the point is that anyone with an Animal reference can safely call Animal methods on a Dog instance. The Animal methods may have been overridden in the Dog class, but all we care about now is that a Dog can always do at least everything an Animal can do. The compiler and JVM know it, too, so the implicit upcast is always legal for assigning an object of a subtype to a reference of one of its supertype classes (or interfaces). If Dog implements Pet, and Pet defines beFriendly(), then a Dog can be implicitly cast to a Pet, but the only Dog method you can invoke then is beFriendly(), which Dog was forced to implement because Dog implements the Pet interface. One more thing…if Dog implements Pet, then if Beagle extends Dog, but Beagle does not declare that it implements Pet, Beagle is still a Pet! Beagle is a Pet simply because it extends Dog, and Dog's already taken care of the Pet parts for itself, and for all its children. The Beagle class can always override any method it inherits from Dog, including methods that Dog implemented to fulfill its interface contract.
* Animal a = new Dog();

Dog d = (Dog) a; // we can call all Dog methods.

can be replaced with this easy-to-read bit of fun:

Animal a = new Dog();

((Dog)a).doDogStuff();

* Follow all the rules for legal overrides, such as the following:
* Declare no checked exceptions on implementation methods other than those declared by the interface method, or subclasses of those declared by the interface method.
* Maintain the signature of the interface method, and maintain the same return type (or a subtype). (But it does not have to declare the exceptions declared in the interface method declaration.)
* abstract class Ball implements Bounceable { }

class BeachBall extends Ball {}// we need to provide implementation for Bounceable methods.

class Alpha {

Alpha doStuff(char c) {

return new Alpha();

}

}

class Beta extends Alpha {

Beta doStuff(char c) { // legal override in Java 1.5

return new Beta();

}

}

* What happens if the super constructor has arguments?

Code fails to compile in derived class super() is calling implicitly so if you define parameterized constructor in parent it will fail.

* constructors are never inherited. They aren't methods. They can't be overridden (because they aren't methods, and only instance methods can be overridden).
* Instance init block (instance initialization block) code runs right after the call to super() in a constructor—in other words, after all super constructors have run.
* if a class has more than one instance initialization block, they will run in the order in which they appear in the class file.
* One of the mistakes most often made by new Java programmers is attempting to access an instance variable (which means nonstatic variable) from the static main() method.

class Foo {

int x = 3;

public static void main (String [] args) {

System.out.println("x is " + x);

}

}

class Foo {

int x = 3;

float y = 4.3f;

public static void main (String [] args) {

for (int z = x; z < ++x; z--, y = y + z)

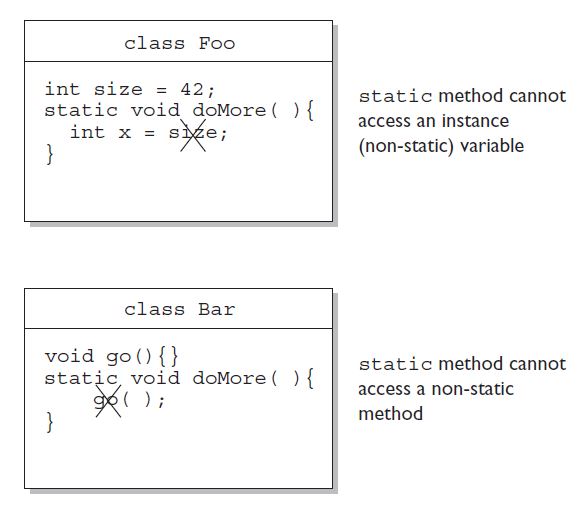
// complicated looping and branching code

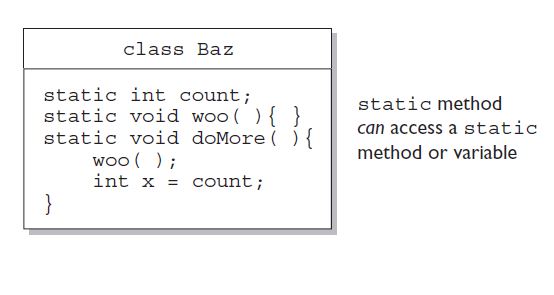
}

}

So while you're trying to follow the logic, the real issue is that x and y can't be used within main().

* Using reference/object variable we can access static variable and methods.





* Overloaded methods can change return types; overridden methods cannot, except in the case of covariant returns.
* Nothing can be returned from a void, but you can return nothing. You're allowed to simply say return.

**Initialization Blocks**

* Use static init blocks—static { /\* code here \*/ }—for code you want to have run once, when the class is first loaded. Multiple blocks run from the top down.
* Use normal init blocks—{ /\* code here }—for code you want to have run for every new instance, right after all the super constructors have run. Again, multiple blocks run from the top of the class down.
* Instance variables and objects live on the heap. Local variables live on the stack.

**Decimal Literals**

In the Java language, they are represented as is, with no prefix of any kind, as follows:

int length = 343;

**Binary Literals**

Also new to Java 7 is the addition of binary literals. Binary literals can use only the digits 0 and 1. Binary literals must start with either 0B or 0b, as shown:

int b1 = 0B101010; // set b1 to binary 101010 (decimal 42)

int b2 = 0b00011; // set b2 to binary 11 (decimal 3)

**Octal Literals**

Octal integers use only the digits 0 to 7. In Java, you represent an integer in octal form by placing a zero in front of the number, as follows:

int six = 06; // Equal to decimal 6

int seven = 07; // Equal to decimal 7

int eight = 010; // Equal to decimal 8

int nine = 011; // Equal to decimal 9

**Hexadecimal Literals**

Hexadecimal (hex for short) numbers are constructed using 16 distinct symbols. Because we never invented single-digit symbols for the numbers 10 through 15, we use alphabetic characters to represent these digits. Counting from 0 through 15 in hex looks like this:

0 1 2 3 4 5 6 7 8 9 a b c d e f

Java will accept uppercase or lowercase letters for the extra digits. You are allowed up to 16 digits in a hexadecimal number, not including the prefix 0x.

**byte** a = 3; // No problem, 3 fits in a byte

**byte** b = 8; // No problem, 8 fits in a byte

**byte** c = a + b;//will result in int so won’t compile use type cast.

byte c = (byte) (a + b);

* int a = 10; // Assign a value to a

System.out.println("a = " + a); // 10

int b = a;

b = 30;

System.out.println("a = " + a + " after change to b");//10

* Static variables have the longest scope; they are created when the class is loaded, and they survive as long as the class stays loaded in the Java Virtual Machine (JVM).
* Instance variables are the next most long-lived; they are created when a new instance is created, and they live until the instance is removed.
* Local variables are next; they live as long as their method remains on the stack. As we'll soon see, however, local variables can be alive and still be "out of scope."
* Block variables live only as long as the code block is executing.
* String x = "Java"; // Assign a value to x

String y = x; // Now y and x refer to the same

System.out.println("y string = " + y);

x = x + " Bean"; // Now modify the object using

System.out.println("y string = " + y);

OP: y string = Java

y string = Java

* void bar() {

Foo f = new Foo();

doStuff(f);

}

void doStuff(Foo g) {

g.setName("Boo");

g = new Foo();

}

reassigning g does not reassign f! At the end of the bar() method, two Foo objects have been created: one referenced by the local variable f and one referenced by the local (argument) variable g. Because the doStuff() method has a copy of the reference variable, it has a way to get to the original Foo object, for instance to call the setName() method. But the doStuff() method does not have a way to get to the f reference variable. So doStuff() can change values within the object f refers to, but doStuff() can't change the actual contents (bit pattern) of f. In other words, doStuff() can change the state of the object that f refers to, but it can't make f refer to a different object!

* class Foo {

static int size = 7;

static void changeIt(int size) {

size = size + 200;

System.out.println("size in changeIt is " + size);

}

public static void main (String [] args) {

Foo f = new Foo();

System.out.println("size = " + size);

changeIt(size);

System.out.println("size after changeIt is " + size);

}

}

OP: size = 7

size in changeIt is 207

size after changeIt is 7

* class Bar {

int barNum = 28;

}

class Foo {

Bar myBar = new Bar();

void changeIt(Bar myBar) {

myBar.barNum = 99;

System.out.println("myBar.barNum in changeIt is " + myBar.barNum);

myBar = new Bar();

myBar.barNum = 420;

System.out.println("myBar.barNum in changeIt is now " + myBar.barNum);

}

public static void main (String [] args) {

Foo f = new Foo();

System.out.println("f.myBar.barNum is " + f.myBar.barNum);

f.changeIt(f.myBar);

System.out.println("f.myBar.barNum after changeIt is "

+ f.myBar.barNum);

}

}

OP: f.myBar.barNum is 28

myBar.barNum in changeIt is 99

myBar.barNum in changeIt is now 420

f.myBar.barNum after changeIt is 99

* garbage collector uses a mark and sweep algorithm.
* From 1.7 with out main method java program won’t work it will throw Error.

Exception in thread "main" java.lang.NoClassDefFoundError:

**public** **class** Test2 {

**static**

{

System.***out***.println("hiiiiiii");

System.*exit*(0);

}

}

Java 7 looks for a main method before loading the class. This is a behavior change from previous java versions and hence your static block is not executing. In previous versions, the behavior was that JRE used to look for main method post loading the class and after executing the static blocks.

**Making Objects Eligible for GC**

1. **Nulling a Reference**

An object becomes eligible for garbage collection when there are no more reachable references to it.

StringBuffer sb = new StringBuffer("hello");

// The StringBuffer object with the value hello is assigned to the reference variable sb in the third line.

System.out.println(sb);

// The StringBuffer object is not eligible for GC

sb = null;

// Now the StringBuffer object is eligible for GC.

1. **Reassigning a Reference Variable**

StringBuffer s1 = new StringBuffer("hello");

StringBuffer s2 = new StringBuffer("goodbye");

System.out.println(s1);

// At this point the StringBuffer "hello" is not eligible

s1 = s2; // Redirects s1 to refer to the "goodbye" object

// Now the StringBuffer "hello" is eligible for collection

* Objects that are created in a method also need to be considered. When a method is invoked, any local variables created exist only for the duration of the method. Once the method has returned, the objects created in the method are eligible for garbage collection. There is an obvious exception, however. If an object is returned from the method, its reference might be assigned to a reference variable in the method that called it; hence, it will not be eligible for collection. Examine the following code:

public class GarbageFactory {

public static void main(String [] args) {

Date d = getDate();

doComplicatedStuff();

System.out.println("d = " + d);

}

public static Date getDate() {

Date d2 = new Date();

StringBuffer now = new StringBuffer(d2.toString());

System.out.println(now);

return d2;

}

}

In the preceding example, we created a method called getDate() that returns a Date object. This method creates two objects: a Date and a StringBuffer containing the date information. Since the method returns a reference to the **Date object and this reference is assigned to a local variable, it will not be eligible for GC** even after the getDate() method has completed. The StringBuffer object, will be eligible, even though we didn't explicitly set the variable to null.

1. **Isolating a Reference**

There is another way in which objects can become eligible for garbage collection, even if they still have valid references! We call this scenario "islands of isolation."

A simple example is a class that has an instance variable that is a reference variable to another instance of the same class. Now imagine that two such instances exist and that they refer to each other. If all other references to these two objects are removed, then even though each object still has a valid reference, there will be no way for any live thread to access either object. When the garbage collector runs, it can usually discover any such islands of objects and remove them.

public class Island (

Island n;

public static void main(String [] args) {

Island i2 = new Island();

Island i3 = new Island();

Island i4 = new Island();

i2.n = i3;

i3.n = i4;

i4.n = i2;

i2 = null;

i3 = null;

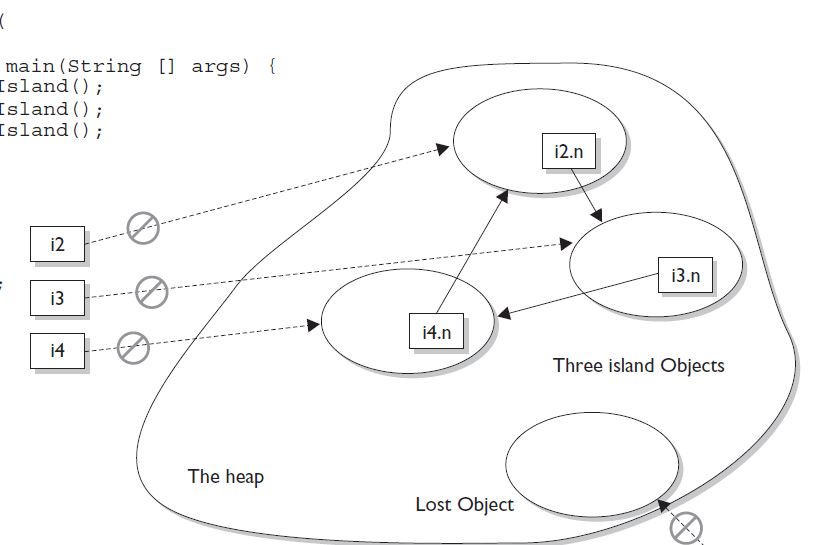
i4 = null;

doComplexStuff();

}

}

When the code reaches // doComplexStuff();, the three Island objects (previously known as i2, i3, and i4) have instance variables so that they refer to each other, but their links to the outside world (i2, i3, and i4) have been nulled. These three objects are eligible for garbage collection. This covers everything you will need to know about making objects eligible for garbage collection.



**Forcing Garbage Collection:**

The Runtime class is a special class that has a single object (a Singleton) for each main program. The Runtime object provides a mechanism for communicating directly with the virtual machine. To get the Runtime instance, you can use the method Runtime.getRuntime(), which returns the Singleton. Once you have the Singleton, you can invoke the garbage collector using the gc() method.

Or

System.gc()

**Note:**

Keep in mind that the behavior when gc() is called may be different for different JVMs, so there is no guarantee that the unused objects will be removed from memory. About the only thing you can guarantee is that if you are running very low on memory, the garbage collector will run before it throws an OutOfMemoryException.

**Cleaning Up Before Garbage Collection—the finalize() Method:**

Java provides a mechanism that lets you run some code just before your object is deleted by the garbage collector. This code is located in a method named finalize() that all classes inherit from class Object. On the surface, this sounds like a great idea; maybe your object opened up some resources, and you'd like to close them before your object is deleted. For any given object, finalize() will be called only once (at most) by the garbage collector.

**Stack and Heap**

❑ Local variables (method variables) live on the stack.

❑ Objects and their instance variables live on the heap.

❑ Compound assignments (such as +=) perform an automatic cast.

**Passing Variables into Methods (OCA Objective 6.8)**

❑ Method arguments are always copies.

❑ Method arguments are never actual objects (they can be references to objects).

❑ A primitive argument is an unattached copy of the original primitive.

❑ A reference argument is another copy of a reference to the original object.

❑ Shadowing occurs when two variables with different scopes share the same name. This leads to hard-to-find bugs and hard-to-answer exam questions.

**Garbage Collection (OCA Objective 2.4)**

❑ In Java, garbage collection (GC) provides automated memory management.

❑ The purpose of GC is to delete objects that can't be reached.

❑ Only the JVM decides when to run the GC; you can only suggest it.

❑ You can't know the GC algorithm for sure.

❑ Objects must be considered eligible before they can be garbage collected.

❑ An object is eligible when no live thread can reach it.

❑ To reach an object, you must have a live, reachable reference to that object.

❑ Java applications can run out of memory.

❑ Islands of objects can be garbage collected, even though they refer to each other.

❑ Request garbage collection with System.gc(); (for OCP 5 candidates only).

❑ The Class object has a finalize() method.

❑ The finalize() method is guaranteed to run once and only once before the garbage collector deletes an object.

❑ The garbage collector makes no guarantees; finalize() may never run.

❑ You can ineligible-ize an object for GC from within finalize().

* Test t= **new** Test();

Object o = **new** Object();

**if**(o **instanceof** Test1)//false

**if**(t **instanceof** Object)//true

* The null reference is an instance of a class. This will always result in false

String a = null;

boolean b = null instanceof String;

boolean c = a instanceof String;

System.out.println(b + " " + c);// false false

**instanceof Compiler Error:**

You can't use the instanceof operator to test across two different class hierarchies.

Dog d = new Dog();

System.out.println(d instanceof Cat);

* Remember that arrays are objects, even if the array is an array of primitives. Watch for questions that look something like this:

int [] nums = new int[3];

if (nums instanceof Object) { } // result is true

* Remember: Expressions are evaluated from left to right by default. You can change this sequence, or precedence, by adding parentheses. Also remember that the \*, /, and % operators have a higher precedence than the + and - operators.
* int b = 2;

System.out.println("" + b + 3);//23

System.out.println("" + (b + 3));//5

* The || and && operators work only with boolean operands. The exam may try to fool you by using integers with these operators:

if (5 && 6) { }

* int z = 5;

if(++z > 5 || ++z > 6) z++; // z = 7 after this code

if(++z > 6 || ++z > 6) z++; // z = 8 after this code

Note: left side is true won’t go farther exp, left side is false then will go.

if(++z > 6 && ++z > 6) z++; // z = 6 after this code

if(++z > 5 && ++z > 6) z++; // z = 8 after this code

Note: left side is false won’t go farther exp, left side is true then will go.

if(++z > 5 | ++z > 6) z++; // z = 8 after this code

if(++z > 5 & ++z > 6) z++; // z = 8 after this code

Note: it will go through entire expression.

* When comparing characters, Java uses the Unicode value of the character as the numerical value.

Ex: if(7>’a’)

* The \*, /, and % operators have higher precedence than + and –.
* The || does not evaluate the right operand if the left operand is true.
* The & and | operators always evaluate both operands.
* The ^ operator (called the "logical XOR") returns true if exactly one operand is true.