Hàm trong C ( Function )

1. Định nghĩa hàm:

* Trong quá trình lập trình người ta thường chia các bài toán thành cái bài nhỏ. Mỗi bài toán như vậy giải quyết 1 vấn đề nhất định nào đó. Và mỗi bài toán nhỏ tổ chức thành các chương trình con nhằm giải quyết yêu cầu của bài toán lớn.
* Có những đoạn lệnh được lặp đi lặp lại nhiều lần để tránh nhàm chán trong lập trình người ta thường tổ chức thành các chương trình con.
* Hàm là đơn vị độc lập của chương trình. Chương trình viết theo ngôn ngữ C là 1 dãy các hàm, trong đó có 1 hàm chính có tên là main() để gọi các hàm khác và các hàm khác có thể gọi lẫn nhau.
* Các hàm thường được định nghĩa sau hàm main, vì vậy trước hàm main() phải khai báo nguyên mẫu hàm cho nó. Nguyên mẫu hàm chính là dòng đầu tiên của hàm được thêm vào dấu “;”. Trong khi khai báo nguyên mẫu hàm ta có thể bỏ qua tên của các biến mà chỉ giữ lại kiểu dữ liệu của các biến mà thôi.
* Cú pháp định nghĩa hàm như sau:

Kiểu dữ liệu + tên hàm (danh sách đối số)

{

Biến cục bộ

Các lệnh

Return + …

}

1. Lời gọi hàm:

* Hoạt động của hàm: hàm được sử dụng trong chương trình chính giống như 1 lệnh. Gọi hàm là gọi giá trị trả về của nó và cú pháp gọi hàm như sau:

Tên hàm (danh sách tham số thực)

* Khi gặp lời gọi hàm chương trình tạm thời vị trí đó và nhảy đến hàm tương ứng để thực thi quá trình được diễn ra với các bước sau:

1. Trừu tượng dữ liệu:

- xác định và tập hợp các tính chất và hành động của 1 thực thể có liên quan đến ứng dụng (tập trung vào vấn đề, loại bỏ những chi tiết không cần thiết, xác định những tính chất và hành động thiết yếu)

1. Lớp: 1 nhóm các đối tượng có chung những tính chất và hành động.
2. Đối tượng là 1 thể hiện của lớp (thuộc tính, hành động, phương thức, thông điệp, biến cố)
3. Constructor: cấp phát vùng nhớ, khởi gán các thuộc tính (nếu có), cho phép truy cập những thuộc tính và phương thức nếu có.
4. Tính lưu trữ: 1 đối tượng sau khi bị hủy, có thể được lưu lại dữ liệu
5. Đa hình: biên dịch (overloading: cùng tên hàm, khác đối số), thông dịch (overwriting: ghi đè)
6. Trải qua 2 quá trình: biên dịch -> bytecode (độc lập phần cứng) -> thông dịch
7. Kiểu dữ liệu cơ sở (primitive data types): double%int = …(float lỗi), phép + - cao hơn ++ --, vd: x=x++ + ++x, y=x => y=7 trong C và y=6 trong java,xor:khác ra 1, giống ra 0,
8. Kiểu dữ liệu tham chiếu: 3 kiểu: mảng, lớp, interface.
9. Khai báo mảng:

- Mảng 1 chiều:

+ KDL arr[];

+ KDL arr[] = new KDL[size]

+ KDL arr[]={x1,x2,…,xn}

- Mảng 2 chiều:

+ KDL arr[][];

KDL [][]arr = new KDL[size][size];

\*Ngoặc vuông để trước arr hoặc sau arr đều được

1. Thứ tự ưu tiên trong Java:

- Trong ngoặc đơn

- + - ++ --

- \* / + - << >>

- < > <= >= == !=

- && || & | ^

- =, \*=, /=, +=, ==

1. Lớp và phương thức:

Class tên lớp

{

//Thuộc tính:

Kiểu DL + tên biến;

//Phương thức

Quyền truy cập + KDL + tên pt()

:

Tạo đối tượng -> gọi các phương thức đáp ứng.

}

1. Phương thức:

- Cú pháp: quyền truy cập + static/non + KDL + tên hàm (DSTS)

1. Trong java không có biến toàn cục
2. Phương thức tĩnh không được truy cập 1 biến non-static hoặc gọi 1 hàm khác non-static.
3. Nạp chồng phương thức (overloading): Cùng ở trong 1 lớp, có cùng tên, khác nhau về ds tham số. Là quá trình đa hình trong biên dịch (compile).
4. Ghi đè phương thức (overriding): có mặt trong lớp cha + kế thừa, được định nghĩa lại trong lớp kế thừa. Là 1 hình thức đa hình trong quá trình thực thi (runtime).
5. Kế thừa: class subclass extends superclass
6. Constructor: dùng để khởi tạo giá trị cho các biến thành viên của lớp đối tượng. Có cùng tên với lớp và không có giá trị trả về. Được gọi khi đối tượng được tạo ra. Có 2 loại: tường minh và ngầm định.
7. Từ khóa static trong java được dùng chính để quản lí bộ nhớ. Không thuộc về 1 đối tượng cụ thể nào. Được áp dụng với biến, phương thức, khối, lớp lồng nhau.
8. Biến static để tham chiếu đến thuộc tính chung của tất cả các đối tượng (không là duy nhất của mỗi đối tượng). Biến static chỉ được cấp phất bộ nhớ 1 lần khi lớp đó được tạo.
9. Phương thức static: thuộc lớp chứ không phải đối tượng của lớp, được gọi mà không cần tạo ra 1 đối tượng của lớp. Phương thức static được tạo ra 1 lần khi lớp được tạo ra. Nếu trong cùng 1 lớp, thì gọi bằng tên phương thức, nếu ở ngoài lớp thì gọi bằng tên lớp.tên phương thức. Phương thức static có thể truy cập tới biến static và thay đổi giá trị của nó. Không được gọi 1 biến non-static hoặc gọi trực tiếp các phương thức non-static. Từ khóa this, super không thể dùng trong ngữ cảnh static.
10. Khối lệnh static: trong java, khối lệnh static chạy trước rồi mới tới hàm main. Vd: static {}. Version dưới 7, không có main, khối lệnh static vẫn chạy, trên 7 thì không có main không chạy.
11. Trong java, hàm main bắt buộc phải có static. Vì chương trình khi chạy sẽ chạy hàm main trước tiên.
12. Access modifier: không ghi 3 chế độ (private, public và protected) thì tự hiểu là default: ngầm định. Mức độ: private -> default -> protected -> public.
13. Cấu trúc lựa chọn: if, switch case, ?:

\* Nếu chỉ dùng if: lập trình giả sử

\* Đặc biệt:

Boolean X = true;

If (X=false) S.o.p(“false”);

Else S.o.p(“true”);

KQ: true. Điều kiện trong câu lệnh if dùng giá trị mà X được gán.

\* Switch case nếu không có break thì thực hiện từ cái đúng đến default. Nếu có break thì chỉ thực hiện case đúng.

1. While và for là một.
2. While (ĐK) lệnh
3. Do lệnh; while (ĐK);
4. For(DSKT; biểu thức kiểm tra; biểu thức tăng giảm) lệnh;
5. Continue: quay lại bắt đầu 1 chu trình mới.
6. Trong Java không được đa năng hóa toán tử, chỉ có đa năng hóa phương thức.
7. Xử lí biệt lệ (exception): lỗi đặc biệt xảy ra trong thông dịch, nếu không xử lí thì chương trình khi gặp lỗi sẽ dừng 1 cách đột ngột. Lớp “throwable” mà Java cung cấp là lớp trên nhất của lớp biệt lệ. Từ khóa: try, catch, throw, throws, finally. Cấu trúc của mô hình xử lí biệt lệ:

Try {…}

Catch(exception e1) {…}

Catch (exception e2) {…}

Catch (exception en) {…}

Finally {…}

Khối lệnh try tập hợp các lệnh thực thi, catch bắt các biệt lệ trước nó. Khối finally thực hiện tất cả các việc thu dọn khi biệt lệ xảy ra. Khối lệnh finally có thể dùng hoặc không dùng (nên dùng). Lệnh throw: phóng biệt lệ qua 1 đối tượng khác để xử lí.

Vd1: class ExceptionTest1

{

Static String s;

Public static void main(String[] args)

Try {

System.out.println(“chieu dai chuoi:” + s.length());

} catch (Exception e){}

System.out.println(“abc”);

System.out.println(“Chao ban! Hello you!”); }

Kết quả: chương trình chạy bình thường, in ra màn hình 2 dòng lệnh cuối. Bắt được lỗi NullPointException.

Vd2: class ExceptionTest1

{

Static String s;

Public static void main(String[] args)

Try {

System.out.println(“chieu dai chuoi:” + s.length());

}

Finally {

System.out.println(“Chao ban! Hello you!”); }

}

Kết quả: hiện ra màn hình “Chao…” rồi thông báo lỗi.

Vd3: numberformatexception.

class ExceptionTest2

{

Static String time = “2.020”;

Public static void main(String[] args)

Try {

System.out.println(“Nam den:” + (Integer.parseInt(time)+1));

} catch (Exception e){}

System.out.println(“Ban doi thanh cong chuoi->so”); }

Vd4: class ExceptionTest3 {

Public static void main(String[] args)

Stack st=new Stack();

St.push(“hello”);

St.push(“you”);

System.out.println(st.pop()); // lấy ra you

System.out.println(st.pop()); // lấy ra hello

System.out.println(st.pop()); // lỗi EmptyStackException

}

Vd5: class ExceptionTest3 {

public static void main(String[] args) throws MalformedURLException

{

try

{

URL url = new URL(“http://mail.yahoo.com”);

}

catch (Exception e) {

System.out.println(“Invalid URL”);

Throw e;

}

}

}

1. Interface là lớp trừu tượng thuần túy, trong Java không cho đa kế thừa nhưng interface thì có.
2. Các tính chất của OOP: kế thừa, đa hình, ghi đè phương thức, nạp chồng phương thức.
3. Có 3 kiểu kế thừa trong java đó là: đơn kế thừa, kế thừa nhiều cấp, kế thừa thứ bậc (nhiều lớp kế thừa từ 1 lớp).
4. Upcast: thành viên lớp cha muốn truy cập đến thuộc tính lớp con (thuộc tính đó phải có ở cả lớp cha và lớp con)
5. Ghi đè là giữ nguyên lại header của phương thức. Phương thức static không được ghi đè. Trong quá trình ghi đè, 1 phương thức được ghi đè được gọi thông qua biến tham chiếu của 1 lớp cha. Phương thức bị ghi đè không là thuộc tính (thành viên dữ liệu). Ghi đè thì không có được bởi dữ liệu thành viên của lớp con (chỉ ghi đè phương thức, không ghi đè thuộc tính), nếu khai báo static thì sẽ truy cập đến thuộc tính của lớp con.

Vd: class Bike {

void run() {

S.o.p(“running”);

}

Class Motor extends Bike {

void run() {

S.o.p(“running safely”);

}

Public static void main(“String args[]) {

Bike b = new Motor();

b.run(); // running safely

}

Vd: class Bike {

void run() {

Int speed = 70;

}

Class Motor extends Bike {

void run() {

Int speed = 120;

}

Public static void main(“String args[]) {

Bike b = new Motor();

S.o.p(speed); //70

}

1. Có 2 cách để nạp chồng phương thức trong java: số lượng tham số khác nhau, kiểu dữ liệu của các tham số. Không thể nạp chồng phương thức bằng cách thay đổi kiểu dữ liệu trả về của phương thức. Ta có thể nạp chồng n phương thức main, nhưng JVM chỉ gọi phương thức main có đối số là 1 mảng String.
2. Sum(int,int)(1), sum(int, long)(2), sum(long, int)(3), sum(long, long)(4). So (2)-(3) => compile error.

|  |  |
| --- | --- |
| **Overloading** | **Overriding** |
| Nạp chồng phương thức giúp code chương trình dễ đọc hơn | Được sử dụng để cung cấp cài đặt cụ thể cho phương thức được khai báo của lớp cha |
| Được thực hiện bên trong 1 class | Xảy ra trong 2 class có quan hệ kế thừa |
| Tham số phải khác nhau | Tham số phải giống nhau |
| Đa hình lúc biên dịch | Đa hình lúc runtime |
| Kiểu giá trị trả về có thể giống hoặc khác nhau nhưng tham số phải khác nhau | Giá trị trả về phải giống nhau |

1. Lớp abstract chứa phương thức abstract hoặc non-abstract.
2. Phương thức abstract không có hình thức triển khai body.

Ex: abstract void InitMethod();

1. Các trường của interface là public, static và final theo mặc định và các phương thức là public và abstract.
2. Một lớp mô tả các thuộc tính và hành vi của 1 đối tượng. Một interface chứa các hành vi mà một class triển khai.
3. Không thể khởi tạo 1 interface. Một interface không chứa bất cứ hàn constructor nào. Tất cả các phương thức của interface đều là abstract. Một interface không thể chứa 1 trường nào trừ các trường vừa static và final. Một interface không thể kế thừa từ lớp, nó được triển khai bởi 1 lớp. Một interface có thể kế thừa từ nhiều interface khác.
4. Ex:

interface printable()

{

void print();

}

class A implements printable

{

public void print()

{

System.out.println(“Hello”);

}

public static void main(String args[])

{

A.obj = new A();

obj.print();

}

}

1. Đa kế thừa: 1 lớp triển khai đa kế thừa, hoặc 1 interface kế thừa từ nhiều interface khác.
2. Ex:

interface B {

void print();

}

interface A {

void print(); // nếu ở đây trả về int => runtime error

}

class TestInterface implements B,C {

public void print() {

System.out.println(“Hello”);

}

public static void main(String args[])

{

TestInterface obj = new TestInterface();

obj.print(); // Hello

}

}

1. Lớp Vector + StringTokenizer
2. Container: frame (khung bao bọc bên ngoài), panel (phần bên trong của frame)
3. Dialog
4. Components khác: textfield, button
5. TextField() / TextField(String s) / TextField(int columns) / TextField(String s, int columns)
6. TextArea() / TextArea(String s) / TextArea(int rows, int cols) / TextArea(String s, int rows, int cols). Các phương thức: setText, getText, setEditable, isEditable, insertText,…
7. Button(), Button(String s)
8. Checkbox(), Checkbox(String text)
9. Để biến checkbox thành radiobutton: tạo ra 1 đối tượng checkboxgroup trước khi tạo button
10. Choice, list
11. Layout Manager: Flow Layout (trái - phải, căn giữa), Border Layout (N-E-W-S, center), Card Layout, Grid Layout (phân bố theo lưới có kích thước hình chữ nhật, mỗi component chứa trong 1 ô), GridBag Layout (như Grid Layout, nhưng mỗi component chứa trong nhiều ô)
12. Xử lí sự kiện: các bước để sử dụng mô hình Event Listener

- Cài đặt Listener tương ứng

- Nhận diện được tất cả các thành phần tạo sự kiện

- Nhận diện được tất cả các sự kiện được xử lí

- Cài đặt các phương thức của listener, và viết các đoạn mã để xử lí sự kiện trong các phương thức đó.

1. ActionEvent: button (nhấp 1 lần), list (nhấp 2 lần), menuitem (có 1 mục được chọn), textfield (enter trong trường textfield).
2. ItemEvent.
3. Lập trình cơ sở dữ liệu. Có 4 loại JDBC driver: loại 1(JDBC sử dụng cầu nối ODBC), loại 2(JDBC kết nối trực tiếp với các trình điều khiển), loại 3(JDBC kết nối thông qua các ứng dụng mạng trung gian), loại 4(JDBC kết nối thông qua các trình điều khiển đặc thù ở xa). Loại 2,3,4 nói chung được viết bởi nhà cung cấp csdl, hiệu quả hơn loại 1 nhưng thực hiện phức tạp hơn.
4. Các bước để kết nối cơ sở dữ liệu:

- Nạp trình điều khiển

- Tạo thông tin kết nối và đối tượng Connection

- Tạo đối tượng Statement để thực thi các lệnh truy vấn

- Xử lí dữ liệu

- Đóng kết nối.

1. getConnection(…) có 3 đối số: url, user, pass.

- Class.forName(String)

- Class.forName(“sun.jdbc.odbc.JdbcOdbcDriver”);

- java.sql.Statement: không có tham số

- java.sql.PrepareStatement: có tham số

- java.sql.CallableStatement: là 1 thủ tục

- Còn dữ liệu => hàm next() = true, ngược lại = false.

1. Luồng hướng Byte: InputStream/OutputStream (mỗi kí tự tương ứng với 1 Byte = 8 bit). Luồng hướng Char: Reader/Writer (mỗi kí tự tương ứng với 2 Byte = 16 bit (kí tự)).
2. Các luồng là những “đường ống” để gửi và nhận thông tin trong các chương trình java. Khi một luồng đọc hoặc ghi, các luồng các sẽ bị khóa.
3. Các lớp luồng I/O: System.out; System.in; System.err
4. Từ System.in -> tạo đường ống: luồng vào LV(InputStreamReader). Từ luồng vào LV -> tạo vùng đệm BR(BufferedReader). Dữ liệu vào vùng đệm BR -> đọc vào cho các biến chương trình.
5. Thread trong java là 1 tiến trình của hệ điều hành.
6. Java cung cấp 2 giải pháp tạo lập luồng:

C1: Thiết lập lớp con của lớp Thread.

C2: Cài đặt lớp xử lí luồng từ giao diện Runnable => để tránh cho việc thừa kế bội

1. C1: dùng class… extends Thread {

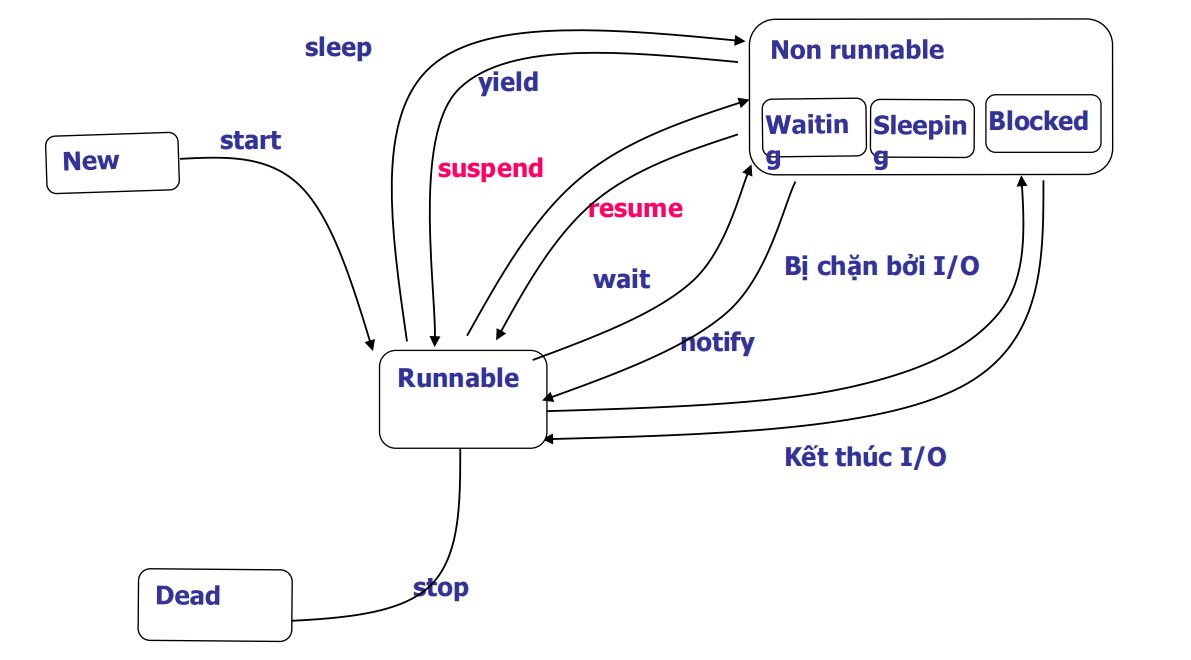
Public void run {các lệnh cần thực hiện theo luồng}

Một số hàm khác được viết đè hay được bổ sung

}

Khi chương trình chạy nó sẽ gọi một hàm đặc biệt đã được khai

báo trong Thread đó là start() để bắt đầu một luồng đã được tạo ra*.*



1. In Java, a protected member is accessible in all classes of same package and in inherited classes of other packages. Since Test and Main are in same package, no access related problem in the above program. Also, the default constructors initialize integral variables as 0 in Java (See [this](https://www.geeksforgeeks.org/g-fact-50/)GFact for more details).

|  |
| --- |
| // filename Test.java  **class** Test {  **public** **static** **void** main(String[] args) {  **for**(**int** i = 0; 1; i++) {              System.out.println("Hello");  **break**;          }      }  } |

Output: Compiler Error  
There is an error in condition check expression of for loop. Java differs from C++(or C) here. C++ considers all non-zero values as true and 0 as false. Unlike C++, an integer value expression cannot be placed where a boolean is expected in Java. Following is the corrected program.



|  |
| --- |
| // filename Main.java  **class** Main {  **public** **static** **void** main(String args[]) {          System.out.println(fun());      }  **int** fun() {  **return** 20;      }  } |

Output: Compiler Error  
Like C++, in Java, non-static methods cannot be called in a static method. If we make fun() static, then the program compiles fine without any compiler error. Following is the corrected program.



|  |
| --- |
| // filename Test.java  **class** Test {  **public** **static** **void** main(String args[]) {         System.out.println(fun());     }  **static** **int** fun() {  **static** **int** x= 0;  **return** ++x;     }  } |

Output: Compiler Error  
Unlike C++, static local variables are not allowed in Java. See [this](https://www.geeksforgeeks.org/g-fact-47/)GFact for details. We can have class static members to count number of function calls and other purposes that C++ local static variables serve. Following is the corrected program.



|  |
| --- |
| **package** main;    **class** Base {  **public** **void** Print()      {          System.out.println("Base");      }  }    **class** Derived **extends** Base {  **public** **void** Print()      {          System.out.println("Derived");      }  }    **class** Main {  **public** **static** **void** DoPrint(Base o)      {          o.Print();      }  **public** **static** **void** main(String[] args)      {          Base x = **new** Base();          Base y = **new** Derived();          Derived z = **new** Derived();          DoPrint(x);          DoPrint(y);          DoPrint(z);      }  } |

**Output:**

Base

Derived

Derived

Predicting the first line of output is easy. We create an object of type Base and call DoPrint(). DoPrint calls the print function and we get the first line.

DoPrint(y) causes the second line of output. Like C++, assigning a derived class reference to a base class reference is allowed in Java. Therefore, the expression Base y = new Derived() is a valid statement in Java. In DoPrint(), o starts referring to the same object as referred by y. Also, unlike C++, functions are virtual by default in Java. So, when we call o.print(), the print() method of Derived class is called due to run time polymorphism present by default in Java.



|  |
| --- |
| **package** main;    // filename Main.java  **class** Point {  **protected** **int** x, y;    **public** Point(**int** \_x, **int** \_y)      {          x = \_x;          y = \_y;      }  }    **public** **class** Main {  **public** **static** **void** main(String args[])      {          Point p = **new** Point();          System.out.println("x = " + p.x + ", y = " + p.y);      }  } |

**Output:**

Compiler Error

In the above program, there are no access permission issues because the Point and Main are in the same package and protected members of a class can be accessed in other classes of the same package. The problem with the code is: *there is no default constructor in Point.*

[Like C++](https://www.geeksforgeeks.org/does-c-compiler-create-default-constructor-when-we-write-our-own/), if we write our own parameterized constructor then Java compiler doesn’t create the default constructor. So there are following two changes to Point class that can fix the above program.

1. Remove the parameterized constructor.
2. Add a constructor without any parameter.

Java doesn’t support default arguments, so that is not an option.  
Please write comments if you find any of the answers/explanations incorrect, or want to share more information about the topics discussed above.



|  |
| --- |
| // file name: Main.java    **class** Base {  **protected** **void** foo() {}  }  **class** Derived **extends** Base {  **void** foo() {}  }  **public** **class** Main {  **public** **static** **void** main(String args[]) {          Derived d = **new** Derived();          d.foo();      }  } |

Output: Compiler Error  
foo() is protected in Base and default in Derived. Default access is more restrictive. When a derived class overrides a base class function,[more restrictive access can’t be given to the overridden function](https://www.geeksforgeeks.org/more-restrictive-access-is-given-to-a-derived-class-method-in-java/). If we make foo() public, then the program works fine without any error. The behavior in C++ is different. [C++ allows to give more restrictive access to derived class methods.](https://www.geeksforgeeks.org/what-happens-when-more-restrictive-access-is-given-in-a-derived-class-method-in-c/)



|  |
| --- |
| // file name: Main.java    **class** Complex {  **private** **double** re, im;  **public** String toString() {  **return** "(" + re + " + " + im + "i)";      }      Complex(Complex c) {          re = c.re;          im = c.im;      }  }    **public** **class** Main {  **public** **static** **void** main(String[] args) {          Complex c1 = **new** Complex();          Complex c2 = **new** Complex(c1);          System.out.println(c2);      }  } |

Output: Compiler Error in line “Complex c1 = new Complex();”  
In Java, if we write our own [copy constructor](https://www.geeksforgeeks.org/copy-constructor-in-java/) or parameterized constructor, then compiler doesn’t create the default constructor. This behavior is same as C++.



|  |
| --- |
| // Main.java  **public** **class** Main  {  **public** **static** **void** gfg(String s)      {          System.out.println("String");      }  **public** **static** **void** gfg(Object o)      {          System.out.println("Object");      }    **public** **static** **void** main(String args[])      {          gfg(**null**);      }  } //end class |

**Output**:

String

**Explanation** : In case of [method overloading](https://www.geeksforgeeks.org/overloading-in-java/), the most specific method is chosen at compile time. As ‘java.lang.String’ is a more specific type than ‘java.lang.Object’. In this case the method which takes ‘String’ as a parameter is choosen.



|  |
| --- |
| // Main.java  **public** **class** Main  {  **public** **static** **void** gfg(String s)      {          System.out.println("String");      }  **public** **static** **void** gfg(Object o)      {          System.out.println("Object");      }  **public** **static** **void** gfg(Integer i)      {          System.out.println("Integer");      }    **public** **static** **void** main(String args[])      {          gfg(**null**);      }  } //end class |

**Output:**

Compile Error at line 19.

**Explanation:** In this case of [method Overloading](https://www.geeksforgeeks.org/overloading-in-java/), the most specific method is choosen at compile time.  
As ‘java.lang.String’ and ‘java.lang.Integer’ is a more specific type than ‘java.lang.Object’,but between ‘java.lang.String’ and ‘java.lang.Integer’ none is more specific.  
In this case the Java is unable to decide which method to call.



|  |
| --- |
| // Main.java  **public** **class** Main  {  **public** **static** **void** main(String args[])      {          String s1 = "abc";          String s2 = s1;          s1 += "d";          System.out.println(s1 + " " + s2 + " " + (s1 == s2));            StringBuffer sb1 = **new** StringBuffer("abc");          StringBuffer sb2 = sb1;          sb1.append("d");          System.out.println(sb1 + " " + sb2 + " " + (sb1 == sb2));      }  } //end class |

Output:

abcd abc false

abcd abcd true

**Explanation :** In Java, String is immutable and string buffer is mutable.  
So string s2 and s1 both pointing to the same string abc. And, after making the changes the string s1 points to abcd and s2 points to abc, hence false. While in string buffer, both sb1 and sb2 both point to the same object. Since string buffer are mutable, making changes in one string also make changes to the other string. So both string still pointing to the same object after making the changes to the object (here sb2).



|  |
| --- |
| // Main.java  **public** **class** Main  {  **public** **static** **void** main(String args[])      {  **short** s = 0;  **int** x = 07;  **int** y = 08;  **int** z = 112345;            s += z;          System.out.println("" + x + y + s);      }  } //end class |

**Output:**

Compile Error at line 8

**Explanation:**  
1. In Line 12 The “” in the println causes the numbers to be automatically cast as strings. So it doesn’t do addition, but appends together as string.  
2. In Line11 the += does an automatic cast to a short. However the number 123456 can’t be contained within a short, so you end up with a negative value (-7616).  
(NOTE – short 2 bytes -32,768 to 32,767), Here the number 123456 doesn’t mean the Value of int z,it shows the length of the int value  
3. Those other two are red herrings however as the code will never compile due to line 8.  
Any number beginning with zero is treated as an octal number (which is 0-7).



|  |
| --- |
| **class** First  {  **int** i = 10;    **public** First(**int** j)      {          System.out.println(i);  **this**.i = j \* 10;      }  }    **class** Second **extends** First  {  **public** Second(**int** j)      {  **super**(j);          System.out.println(i);  **this**.i = j \* 20;      }  }    **public** **class** MainClass  {  **public** **static** **void** main(String[] args)      {          Second n = **new** Second(20);          System.out.println(n.i);      }  } |

Output:

10

200

400

**Explanation:**  
Since in ‘Second’ class it doesn’t have its own ‘i’, the variable is inherited from the super class. Also, the constructor of parent is called when we create an object of Second.



|  |
| --- |
| **import** java.util.\*;  **class** I  {  **public** **static** **void** main (String[] args)      {          Object i = **new** ArrayList().iterator();          System.out.print((i **instanceof** List) + ", ");          System.out.print((i **instanceof** Iterator) + ", ");          System.out.print(i **instanceof** ListIterator);      }  } |

Output:

false, true, false

**Explanation:**  
The iterator() method returns an iterator over the elements in the list in proper sequence, it doesn’t return a List or a ListIterator object. A ListIterator can be obtained by invoking the listIterator method.



|  |
| --- |
| **class** ThreadEx **extends** Thread  {  **public** **void** run()      {          System.out.print("Hello...");      }  **public** **static** **void** main(String args[])      {          ThreadEx T1 = **new** ThreadEx();          T1.start();          T1.stop();          T1.start();      }  } |

Output:

Run Time Exception

**Explanation:**  
Exception in thread “main” java.lang.IllegalThreadStateException at java.lang.Thread.start  
Thread cannot be started twice.



|  |
| --- |
| **public** **class** MyStuff  {      String name;        MyStuff(String n) {  name = n;  }    **public** **static** **void** main(String[] args)      {          MyStuff m1 = **new** MyStuff("guitar");          MyStuff m2 = **new** MyStuff("tv");          System.out.println(m2.equals(m1));      }        @Override  **public** **boolean** equals(Object obj)      {          MyStuff m = (MyStuff) obj;  **if** (m.name != **null**)  { **return** **true**;  }  **return** **false**;      }  } |

**Options :**  
A) The output is true and MyStuff fulfills the Object.equals() contract.  
B) The output is false and MyStuff fulfills the Object.equals() contract.  
C) The output is true and MyStuff does NOT fulfill the Object.equals() contract.  
D) The output is false and MyStuff does NOT fulfill the Object.equals() contract.

**Answer :** C) The output is true and MyStuff does NOT fulfill the Object.equals() contract.  
**Explanation :** As equals(Object obj) method in Object class, compares two objects on the basis of equivalence relation. But here we are just confirming that the object is null or not, So it doesn’t fulfill [Object.equals()](https://www.geeksforgeeks.org/overriding-equals-method-in-java/) contract. As m1 is not null, true will be printed.



|  |
| --- |
| **class** Alpha  {  **public** String type = "a ";  **public** Alpha() {  System.out.print("alpha "); }  }    **public** **class** Beta **extends** Alpha  {  **public** Beta()  {  System.out.print("beta ");  }    **void** go()      {          type = "b ";          System.out.print(**this**.type + **super**.type);      }    **public** **static** **void** main(String[] args)      {  **new** Beta().go();      }  } |

**Options :**  
A) alpha beta b b  
B) alpha beta a b  
C) beta alpha b b  
D) beta alpha a b

**Answer :** A) alpha beta b b  
**Explanation :** The statement **new Beta().go()**executes in two phases. In first phase Beta class constructor is called. There is no instance member present in Beta class. So now Beta class constructor is executed. As Beta class extends Alpha class, so call goes to Alpha class constructor as first statement by default(Put by the compiler) is **super()** in the Beta class constructor. Now as one instance variable(type) is present in Alpha class, so it will get memory and now Alpha class constructor is executed, then call return to Beta class constructor next statement. So alpha beta is printed.  
In second phase go() method is called on this object. As there is only one variable(type) in the object whose value is a. So it will be changed to b and printed two times. The [super keyword](https://www.geeksforgeeks.org/super-keyword/) here is of no use.



|  |
| --- |
| **public** **class** Test  {  **public** **static** **void** main(String[] args)      {          StringBuilder s1 = **new** StringBuilder("Java");          String s2 = "Love";          s1.append(s2);          s1.substring(4);  **int** foundAt = s1.indexOf(s2);          System.out.println(foundAt);      }  } |

**Options :**  
A) -1  
B) 3  
C) 4  
D) A **StringIndexOutOfBoundsException** is thrown at runtime.  
**Answer :** C) 4  
**Explanation :** append(String str) method,concatenate the str to s1. The substring(int index) method return the String from the given index to the end. But as there is no any String variable to store the returned string,so it will be destroyed.Now indexOf(String s2) method return the index of first occurrence of s2. So 4 is printed as s1=”JavaLove”.



|  |
| --- |
| **class** Writer  {  **public**  **static** **void** write()      {          System.out.println("Writing...");      }  }  **class** Author **extends** Writer  {  **public**  **static** **void** write()      {          System.out.println("Writing book");      }  }    **public** **class** Programmer **extends** Author  {  **public**  **static** **void** write()      {          System.out.println("Writing code");      }    **public** **static** **void** main(String[] args)      {          Author a = **new** Programmer();          a.write();      }  } |

**Options :**  
A) Writing…  
B) Writing book  
C) Writing code  
D) Compilation fails

**Answer :** B) Writing book  
**Explanation :** Since static methods can’t be overridden, it doesn’t matter which class object is created. As a is a Author referenced type, so always Author class method is called. If we remove write() method from Author class then Writer class method is called, as Author class extends Writer class.



|  |
| --- |
| **class** GfG  {  **public** **static** **void** main(String args[])      {          String s1 = **new** String("geeksforgeeks");          String s2 = **new** String("geeksforgeeks");  **if** (s1 == s2)              System.out.println("Equal");  **else**              System.out.println("Not equal");      }  } |

Output: 

Not equal

**Explanation:** Since, s1 and s2 are two different objects the references are not the same, and the == operator compares object reference. So it prints “Not equal”, to compare the actual characters in the string .equals() method must be used.



|  |
| --- |
| **class** Person  {  **private** **void** who()      {          System.out.println("Inside private method Person(who)");      }    **public** **static** **void** whoAmI()      {          System.out.println("Inside static method, Person(whoAmI)");      }    **public** **void** whoAreYou()      {          who();          System.out.println("Inside virtual method, Person(whoAreYou)");      }  }    **class** Kid **extends** Person  {  **private** **void** who()      {          System.out.println("Kid(who)");      }    **public** **static** **void** whoAmI()      {          System.out.println("Kid(whoAmI)");      }    **public** **void** whoAreYou()      {          who();          System.out.println("Kid(whoAreYou)");      }  }  **public** **class** Gfg  {  **public** **static** **void** main(String args[])      {          Person p = **new** Kid();          p.whoAmI();          p.whoAreYou();      }  } |

Output: 

Inside static method, Person(whoAmI)

Kid(who)

Kid(whoAreYou)

**Explanation:** Static binding (or compile time) happens for static methods. Here *p.whoAmI()* calls the static method so it is called during compile time hence results in static binding and prints the method in Person class.   
Whereas *p.whoAreYou()* calls the method in *Kid* class since by default Java takes it as a virtual method i.e, dynamic binding.



|  |
| --- |
| **class** One **implements** Runnable  {  **public** **void** run()      {          System.out.print(Thread.currentThread().getName());      }  }  **class** Two **implements** Runnable  {  **public** **void** run()      {  **new** One().run();  **new** Thread(**new** One(),"gfg2").run();  **new** Thread(**new** One(),"gfg3").start();      }  }  **class** Three  {  **public** **static** **void** main (String[] args)      {  **new** Thread(**new** Two(),"gfg1").start();      }  } |

Output: 

gfg1gfg1gfg3

**Explanation :**Initially new Thread is started with name *gfg1* then in class Two the first run method runs the thread with the name *gfg1*, then after that a new thread is created calling run method but since a new thread can be created by calling start method only so the previous thread does the action and again *gfg1* is printed.Now a new thread is created by calling the start method so a new thread starts with *gfg3* name and hence prints *gfg3*.



|  |
| --- |
| **class** Gfg  {      // constructor      Gfg()      {          System.out.println("Geeksforgeeks");      }    **static** Gfg a = **new** Gfg(); //line 8    **public** **static** **void** main(String args[])      {          Gfg b; //line 12          b = **new** Gfg();      }  } |

Output:

Geeksforgeeks

Geeksforgeeks

**Explanation:**  
We know that static variables are called when a class loads and static variables are called only once. Now line 13 results to creation of object which inturn calls the constructor and “Geeksforgeeks” is printed second time.  
If in line 8 static variable would not have been used the object would have been called recursively infinitely leading to StackOverFlow error. See [this](https://ide.geeksforgeeks.org/wtntd4) for a sample run.



|  |
| --- |
| **class** Gfg  {  **static** **int** num;  **static** String mystr;        // constructor      Gfg()      {          num = 100;          mystr = "Constructor";      }        // First Static block  **static**      {          System.out.println("Static Block 1");          num = 68;          mystr = "Block1";      }        // Second static block  **static**      {          System.out.println("Static Block 2");          num = 98;          mystr = "Block2";      }    **public** **static** **void** main(String args[])      {          Gfg a = **new** Gfg();          System.out.println("Value of num = " + a.num);          System.out.println("Value of mystr = " + a.mystr);      }  } |

Output:

Static Block 1

Static Block 2

Value of num = 100

Value of mystr = Constructor

**Explanation:**  
Static block gets executed when the class is loaded in the memory. A class can have multiple Static blocks, which are executed in the same sequence in which they have been written into the program.  
**Note**: Static Methods can access class variables without using object of the class. Since constructor is called when a new instance is created so firstly the static blocks are called and after that the constructor is called. If we would have run the same program without using object, the constructor would not have been called.



|  |
| --- |
| **class** superClass  {  **final** **public** **int** calc(**int** a, **int** b)      {  **return** 0;      }  }  **class** subClass **extends** superClass  {  **public** **int** calc(**int** a, **int** b)      {  **return** 1;      }  }  **public** **class** Gfg  {  **public** **static** **void** main(String args[])      {          subClass get = **new** subClass();          System.out.println("x = " + get.calc(0, 1));      }  } |

Output:

Compilation fails.

**Explanation:**  
The method calc() in class superClass is final and so cannot be overridden.



|  |
| --- |
| **public** **class** Gfg  {  **public** **static** **void** main(String[] args)      {          Integer a = 128, b = 128;          System.out.println(a == b);            Integer c = 100, d = 100;          System.out.println(c == d);      }  } |

Output:

false

true

**Explanation:** In the source code of Integer object we will find a method ‘valueOf’ in which we can see that the range of the Integer object lies from IntegerCache.low(-128) to IntegerCache.high(127). Therefore the numbers above 127 will not give the expected output. The range of IntegerCache can be observed from the source code of the IntegerCache class. Please refer [this](https://blogs.oracle.com/darcy/entry/boxing_and_caches_integer_valueof) for details.



|  |
| --- |
| **public** **class** Test  {  **public** **static** **void** main(String[] args) **throws** InterruptedException      {          String str = **new** String("GeeksForGeeks");            // making str eligible for gc          str = **null**;            // calling garbage collector          System.gc();            // waiting for gc to complete          Thread.sleep(1000);            System.out.println("end of main");      }        @Override  **protected** **void** finalize()      {          System.out.println("finalize method called");      }  } |

Output:

end of main

**Explanation :** We know that [finalize()](https://www.geeksforgeeks.org/g-fact-24-finalfinally-and-finalize-in-java/) method is called by Garbage Collector on an object before destroying it. But here, the trick is that the str is String class object, not the Test class. Therefore, finalize() method of String class(if overridden in String class) is called on str. If a class doesn’t override finalize method, then by default Object class finalize() method is called.



|  |
| --- |
| **public** **class** Test  {  **public** **static** **void** main(String[] args) **throws** InterruptedException      {          Test t = **new** Test();            // making t eligible for garbage collection          t = **null**;            // calling garbage collector          System.gc();            // waiting for gc to complete          Thread.sleep(1000);            System.out.println("end main");      }        @Override  **protected** **void** finalize()      {          System.out.println("finalize method called");          System.out.println(10/0);      }    } |

Output:

finalize method called

end main

**Explanation :**  
When Garbage Collector calls finalize() method on an object, it **ignores** all the exceptions raised in the method and program will terminate normally.



|  |
| --- |
| **public** **class** Test  {  **public** **static** **void** main(String[] args)      {          // How many objects are eligible for          // garbage collection after this line?          m1();  // Line 5      }    **static** **void** m1()      {          Test t1 = **new** Test();          Test t2 = **new** Test();      }  } |

**Question :**  
How many objects are eligible for garbage collection after execution of line 5 ?  
**Answer :**

2

**Explanation :**  
Since t1 and t2 are local objects of m1() method, so they become eligible for garbage collection after complete execution of method unless any of them is returned.



|  |
| --- |
| **public** **class** Test  {  **public** **static** **void** main(String [] args)      {          Test t1 = **new** Test();          Test t2 = m1(t1); // line 6          Test t3 = **new** Test();          t2 = t3; // line 8        }    **static** Test m1(Test temp)      {          temp = **new** Test();  **return** temp;      }  } |

**Question :**  
How many objects are eligible for garbage collection after execution of line 8?  
**Answer :**

1

**Explanation :**  
By the time line 8 has executed, the only object without a reference is the one generated i.e as a result of line 6. Remember that “[Java is strictly pass by value](https://www.geeksforgeeks.org/passing-and-returning-objects-in-java/" \t "https://www.geeksforgeeks.org/output-of-java-programs-set-10-garbage-collection/_blank)” so the reference variable t1 is not affected by the m1() method. We can check it using finalize() method. The statement “System.out.println(this.hashcode())” in finalize() method print the object hashcode value on which finalize() method is called,and then just compare the value with other objects hashcode values created in main method.



|  |
| --- |
| **public** **class** Base  {  **private** **int** data;    **public** Base()      {          data = 5;      }    **public** **int** getData()      {  **return** **this**.data;      }  }    **class** Derived **extends** Base  {  **private** **int** data;  **public** Derived()      {          data = 6;      }  **private** **int** getData()      {  **return** data;      }    **public** **static** **void** main(String[] args)      {          Derived myData = **new** Derived();          System.out.println(myData.getData());      }  } |

a) 6  
b) 5  
c) Compile time error  
d) Run time error

Answer (c)  
**Explanation:** When overriding a method of superclass, [the method declaration in subclass cannot be more restrictive than that declared in the superclass](https://www.geeksforgeeks.org/more-restrictive-access-is-given-to-a-derived-class-method-in-java/).



|  |
| --- |
| **public** **class** Test  {  **private** **int** data = 5;    **public** **int** getData()      {  **return** **this**.data;      }  **public** **int** getData(**int** value)      {  **return** (data+1);      }  **public** **int** getData(**int**... value)      {  **return**  (data+2);      }    **public** **static** **void** main(String[] args)      {          Test temp = **new** Test();          System.out.println(temp.getData(7, 8, 12));      }  } |

a) Either Compile time or Runtime error  
b) 8  
c) 10  
d) 7

Answer (d)  
Explanation : [(int… values) is passed as parameter to a method when you are not aware of the number of input parameter but know that the type of parameter](https://www.geeksforgeeks.org/variable-arguments-varargs-in-java/)(in this case it is int). When a new object is made in the main method, variable data is initialized to 5. A call to getData() method with attributes (7, 8 ,12), makes a call to the third getData() method, which increments the value of data variable by 2 and return 7.



|  |
| --- |
| **public** **class** Base  {  **private** **int** multiplier(**int** data)      {  **return** data\*5;      }  }    **class** Derived **extends** Base  {  **private** **static** **int** data;  **public** Derived()      {          data = 25;      }  **public** **static** **void** main(String[] args)      {          Base temp = **new** Derived();          System.out.println(temp.multiplier(data));      }  } |

a) 125  
b) 25  
c) Runtime error  
d) Compile time error

Answer (d)  
**Explanation:** Since the method multiplier is marked as private, it isn’t inherited and therefore is not visible to the Derived.

1. Which of the following is FALSE about finally block?  
   a) For each try block, there can be only 1 finally block.  
   b) finally block will not be executed if program exits by calling System.exit();  
   c) If an exception is not handled in the catch statement, before terminating the program, JVM executes the finally block  
   d) finally block contains important code segment and so the code inside finally block is executed with/without the presence of try block in java code.

Answer (d)  
**Explanation:**Statement (d) is false because finally blocks can exist only if it succeeds try or a catch block. Using a finally block without try block would give a compile time error.



|  |
| --- |
| **import** java.io.IOException;  **import** java.util.EmptyStackException;    **public** **class** newclass  {  **public** **static** **void** main(String[] args)      {  **try**          {              System.out.printf("%d", 1);  **throw**(**new** Exception());          }  **catch**(IOException e)          {              System.out.printf("%d", 2);          }  **catch**(EmptyStackException e)          {              System.out.printf("%d", 3);          }  **catch**(Exception e)          {              System.out.printf("%d", 4);          }  **finally**          {              System.out.printf("%d", 5);          }      }  } |

a) 12345  
b) 15  
c) 135  
d) 145

Answer (d)  
**Explanation:** The catch statements are written in the order: more specific to more general. In the code above, a new exception of type Exception is thrown. First the code jumps to first catch block to look for exception handler. But since the IOException is not  
of the same type it is moves down to second catch block and finally to the third, where  
the exception is caught and 4 is printed. Therefore, the answer is 145, as the order  
of execution in terms of blocks is: try->catch->finally.



|  |
| --- |
| **public** **class** javaclass  {  **static**      {          System.out.printf("%d", 1);      }  **static**      {          System.out.printf("%d", 2);      }  **static**      {          System.out.printf("%d", 3);      }  **private** **static** **int** myMethod()      {  **return** 4;      }  **private** **int** function()      {  **return** 5;      }    **public** **static** **void** main(String[] args)      {          System.out.printf("%d", (**new** javaclass()).function() + myMethod());      }  } |

a) 123  
b) 45  
c) 12345  
d) 1239

Answer (d)  
**Explanation:**static blocks in Java are executed even before the call to main is made by the compiler. In the main method, a new object of javaclass is made and its function() method is called which return 5 and the static method myMethod() returns 4 i.e., 4+5 = 9. Therefore, the output of the program is 1239, because 123 is printed on the console even before main method executes and main method on execution returns 9.



|  |
| --- |
| **public** **class** Test **implements** Runnable  {  **public** **void** run()      {          System.out.printf("%d",3);      }  **public** **static** **void** main(String[] args) **throws** InterruptedException      {          Thread thread = **new** Thread(**new** Test());          thread.start();          System.out.printf("%d",1);          thread.join();          System.out.printf("%d",2);      }    } |

a) 123  
b) 213  
c) 132  
d) 321

Ans: (c)

**Explanation:** The parent thread waits for the newly created thread to complete using join. [join()](https://www.geeksforgeeks.org/joining-threads-in-java/) method allows one thread to wait until another thread completes its execution. So, parent thread prints 1 and wait for the child thread to complete. The child thread prints 3 on console and finally the parent thread prints 2.



|  |
| --- |
| **public** **class** Test  {  **private** **static** **int** value = 20;  **public** **int** s = 15;  **public** **static** **int** temp = 10;  **public** **static** **class** Nested      {  **private** **void** display()        {            System.out.println(temp + s + value);        }      }    **public** **static** **void** main(String args[])      {        Test.Nested inner = **new** Test.Nested();        inner.display();      }  } |

a) Compilation error  
b) 1020  
c) 101520  
d) None of the above

Ans: (a)

**Explanation:** A non-static variable can not be accessed in [static nested inner class](https://www.geeksforgeeks.org/inner-class-java/). “Nested” cannot access non-static variables[variable s in this case]. Therefore the error:

10: error: non-static variable s cannot be referenced from a static context

System.out.println(temp + s + value);

^



|  |
| --- |
| **import** java.io.\*;  **public** **class** Test  {  **public** **void** display() **throws** IOException      {          System.out.println("Test");      }    }    **class** Derived **extends** Test  {  **public** **void** display() **throws** IOException      {          System.out.println("Derived");      }  **public** **static** **void** main(String[] args) **throws** IOException      {          Derived object = **new** Derived();          object.display();      }  } |

a) Test  
b) Derived  
c) Compilation error  
d) Runtime error

Ans: (b)

**Explanation:** If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.



|  |
| --- |
| **public** **class** Test **extends** Thread  {  **public** **void** run()      {          System.out.printf("Test ");      }  **public** **static** **void** main(String[] args)      {          Test test = **new** Test();          test.run();          test.start();      }  } |

a) Compilation error  
b) Runtime error  
c) Test  
d) Test Test

Ans: (d)

**Explanation:** test.run() executes the run method. test.start() creates a new thread and executes the overriden run method of the Thread class. The Thread.start() method always starts a new thread, and the entry point for this thread is the run() method. If you are calling run() directly it will execute in the same thread BUT it is **always recommendable** logically calling Thread.start() to start a new thread of execution followed by the run() method.



|  |
| --- |
| **public** **class** Test **extends** Thread  {  **public** **static** **void** main(String[] args)      {          String a = "GeeksforGeeks";          String b = **new** String(a);  **int** value = 0;          value = (a==b) ? 1:2;  **if**(value == 1)          {              System.out.println("GeeksforGeeks");          }  **else** **if**(value == 2)          {              System.out.println("Geeks for Geeks");          }  **else**          {              System.out.println("GFG");          }        }  } |

a) GeeksforGeeks  
b) Geeks for Geeks  
c) GFG  
d) None of the above

Ans: (b)

**Explanation:** == operator checks if two variable refer to the same object. Here a and b  
refers to two different objects. ?: is another form of if else statement that could be read as, condition : if true then do this : else do this.



|  |
| --- |
| **public** **class** Test  {  **try**      {  **public** Test()          {              System.out.println("GeeksforGeeks");  **throw** **new** Exception();          }      }  **catch**(Exception e)      {          System.out.println("GFG");      }  **public** **static** **void** main(String[] args)      {          Test test = **new** Test();      }  } |

a) GeeksforGeeks  
b) GFG  
c) Compilation error  
d) None of the above

Ans: (c)

**Explanation:** [Constructors](https://www.geeksforgeeks.org/constructors-in-java/) cannot be enclosed in try/catch block.



|  |
| --- |
| **public** **interface** Test  {  **public** **int** calculate();  **protected** **interface** NestedInterface      {  **public** **void** nested();      }  } |

a) Compile time error due to NestedInterface  
b) Compile time error due to access modifier of NestedInterface  
c) No Compile time error  
d) NestedInterface cannot hold any function declaration.

Ans: (b)

**Explanation:** Access modifier of NestedInterface can only be public. Therefore the error:

4: error: illegal combination of modifiers: public and protected

protected interface NestedInterface

^

1 error



a) Constructors can be declared final.  
b) Constructors can be surrounded by try/catch blocks.  
c) Constructor cannot throw exception.  
d) Constructors can hold synchronized code(so that each thread can access constructor sequentially).

Ans: (d)

**Explanation:** Constructors allows a sequential access of data between threads.



|  |
| --- |
| **import** java.util.\*;    **public** **class** priorityQueue {  **public** **static** **void** main(String[] args)      {          PriorityQueue<Integer> queue              = **new** PriorityQueue<>();          queue.add(11);          queue.add(10);          queue.add(22);          queue.add(5);          queue.add(12);          queue.add(2);    **while** (queue.isEmpty() == **false**)              System.out.printf("%d ", queue.remove());            System.out.println("\n");      }  } |

a) 11 10 22 5 12 2   
b) 2 12 5 22 10 11   
c) 2 5 10 11 12 22   
d) 22 12 11 10 5 2

**Ans. (c)**

**Explanation:** Priority queue always outputs the minimum element from the queue when remove() method is called, no matter what the sequence of input is.



|  |
| --- |
| **import** java.util.\*;    **public** **class** Treeset {  **public** **static** **void** main(String[] args)      {          TreeSet<String> treeSet = **new** TreeSet<>();            treeSet.add("Geeks");          treeSet.add("For");          treeSet.add("Geeks");          treeSet.add("GeeksforGeeks");    **for** (String temp : treeSet)              System.out.printf(temp + " ");            System.out.println("\n");      }  } |

a) Geeks For Geeks GeeksforGeeks   
b) Geeks For GeeksforGeeks   
c) For Geeks GeeksforGeeks   
d) For GeeksforGeeks Geeks

**Ans. (c)**

**Explanation:** A TreeSet sorts the data in ascending order that is inserted in it. Therefore, the output string contains all the strings arranged in ascending order. A TreeSet does not contain any duplicate element as it is a set. So in the output, there is just a single occurrence of string ‘Geeks’.



|  |
| --- |
| **import** java.util.\*;    **public** **class** linkedList {  **public** **static** **void** main(String[] args)      {          List<String> list1 = **new** LinkedList<>();          list1.add("Geeks");          list1.add("For");          list1.add("Geeks");          list1.add("GFG");          list1.add("GeeksforGeeks");            List<String> list2 = **new** LinkedList<>();          list2.add("Geeks");            list1.removeAll(list2);    **for** (String temp : list1)              System.out.printf(temp + " ");            System.out.println();      }  } |

a) For Geeks GFG GeeksforGeeks   
b) For GeeksforGeeks GFG   
c) For GFG for   
d) For GFG GeeksforGeeks

**Ans. (d)**

**Explanation:** list1.removeAll(list2) function deletes all the occurrence of string in list2 from list1. Here, the string ‘Geeks’ appears in list2, so all the nodes of linked list in list1 that contains ‘Geeks’ as its data is removed from list1.



|  |
| --- |
| **import** java.util.\*;    **public** **class** hashSet {  **public** **static** **void** main(String[] args)      {          HashSet<String> hashSet = **new** HashSet<>();          hashSet.add("Geeks");          hashSet.add("For");          hashSet.add("Geeks");          hashSet.add("GeeksforGeeks");            System.out.println(hashSet);      }  } |

a) [Geeks, For, Geeks, GeeksforGeeks]   
b) [GeeksforGeeks, Geeks, For]

**Ans. (b)**

**Explanation:** A HashSet is a set and as a set doesn’t contain any duplicate element therefore, the string ‘Geeks’ appears only once in the output.



|  |
| --- |
| **import** java.util.\*;    **public** **class** stack {  **public** **static** **void** main(String[] args)      {          List<String> list = **new** LinkedList<>();          list.add("Geeks");          list.add("For");          list.add("Geeks");          list.add("GeeksforGeeks");          Iterator<Integer> iter = list.iterator();    **while** (iter.hasNext())              System.out.printf(iter.next() + " ");            System.out.println();      }  } |

a) Geeks for Geeks GeeksforGeeks   
b) GeeksforGeeks Geeks for Geeks   
c) Runtime Error   
d) Compilation Error

**Ans. (d)**

**Explanation:** An iterator made for iterating over Integer cannot be used to iterate over String data type.



|  |
| --- |
| **class** Helper  {  **private** **int** data;  **private** Helper()      {          data = 5;      }  }  **public** **class** Test  {  **public** **static** **void** main(String[] args)      {          Helper help = **new** Helper();          System.out.println(help.data);      }  } |

a) Compilation error  
b) 5  
c) Runtime error  
d) None of these

**Ans.** (a)  
**Explanation:**A [private constructor](https://www.geeksforgeeks.org/private-constructors-and-singleton-classes-in-java/) cannot be used to initialize an object outside the class that it is defined within because it is no longer visible to the external class.



|  |
| --- |
| **class** Temp  {  **private** Temp(**int** data)      {          System.out.printf(" Constructor called ");      }  **protected** **static** Temp create(**int** data)      {          Temp obj = **new** Temp(data);  **return** obj;      }  **public** **void** myMethod()      {          System.out.printf(" Method called ");      }  }    **public** **class** Test  {  **public** **static** **void** main(String[] args)      {          Temp obj = Temp.create(20);          obj.myMethod();      }  } |

a) Constructor called Method called  
b) Compilation error  
c) Runtime error  
d) None of the above

**Ans.**(a)  
**Explanation:**When a constructor is marked as private, the only way to create a new object of that class from some external class is using a method that creates a new object, as defined above in the program. The method create() is responsible for creation of Temp object from some other external class. Once the object is created, its method can be invoked from the class in which the object is created.



|  |
| --- |
| **public** **class** Test  {  **public** Test()      {          System.out.printf("1");  **new** Test(10);          System.out.printf("5");      }  **public** Test(**int** temp)      {          System.out.printf("2");  **new** Test(10, 20);          System.out.printf("4");      }  **public** Test(**int** data, **int** temp)      {          System.out.printf("3");        }    **public** **static** **void** main(String[] args)      {          Test obj = **new** Test();        }    } |

a) 12345  
b) Compilation error  
c) 15  
d) Runtime error

**Ans.**(a)  
**Explanation:**[Constructors can be chained](https://www.geeksforgeeks.org/constructor-chaining-java-examples/) and overloaded. When Test() is called, it creates another Test object calling the constructor Test(int temp).



|  |
| --- |
| **class** Base  {  **public** **static** String s = " Super Class ";  **public** Base()      {          System.out.printf("1");      }  }  **public** **class** Derived **extends** Base  {  **public** Derived()      {          System.out.printf("2");  **super**();      }    **public** **static** **void** main(String[] args)      {          Derived obj = **new** Derived();          System.out.printf(s);      }  } |

a) 21 Super Class  
b) Super Class 21  
c) Compilation error  
d) 12 Super Class  
**Ans.** (c)  
**Explanation:**Constructor call to super class must be the first statement in the constructor of the Derived class.



|  |
| --- |
| **public** **class** Outer  {  **public** **static** **int** temp1 = 1;  **private** **static** **int** temp2 = 2;  **public** **int** temp3 = 3;  **private** **int** temp4 = 4;    **public** **static** **class** Inner      {  **private** **static** **int** temp5 = 5;    **private** **static** **int** getSum()          {  **return** (temp1 + temp2 + temp3 + temp4 + temp5);          }      }    **public** **static** **void** main(String[] args)      {          Outer.Inner obj = **new** Outer.Inner();          System.out.println(obj.getSum());      }    } |

a) 15  
b) 9  
c) 5  
d) Compilation Error

**Ans.** (d)  
**Explanation:** static inner classes cannot access non-static fields of the outer class.



|  |
| --- |
| **public** **class** Outer  {  **private** **static** **int** data = 10;  **private** **static** **int** LocalClass()      {  **class** Inner          {  **public** **int** data = 20;  **private** **int** getData()              {  **return** data;              }          };          Inner inner = **new** Inner();  **return** inner.getData();      }    **public** **static** **void** main(String[] args)      {          System.out.println(data \* LocalClass());      }  } |

a) Compilation error  
b) Runtime Error  
c) 200  
d) None of the above

**Ans.** (c)  
**Explanation:** LocalClass() method defines a local inner class. This method creates an object of class Inner and return the value of the variable data that resides within it.



|  |
| --- |
| **interface** Anonymous  {  **public** **int** getValue();  }  **public** **class** Outer  {  **private** **int** data = 15;  **public** **static** **void** main(String[] args)      {          Anonymous inner = **new** Anonymous()                  {  **int** data = 5;  **public** **int** getValue()                      {  **return** data;                      }  **public** **int** getData()                      {  **return** data;                      }                  };          Outer outer = **new** Outer();          System.out.println(inner.getValue() + inner.getData() + outer.data);      }  } |

a) 25  
b) Compilation error  
c) 20  
d) Runtime error

**Ans.** (b)  
**Explanation:** the method getData() is undefined in Anonymous class which causes the compilation error.



|  |
| --- |
| **public** **class** Outer  {  **private** **int** data = 10;    **class** Inner      {  **private** **int** data = 20;  **private** **int** getData()          {  **return** data;          }  **public** **void** main(String[] args)          {              Inner inner = **new** Inner();              System.out.println(inner.getData());            }      }  **private** **int** getData()      {  **return** data;      }  **public** **static** **void** main(String[] args)      {          Outer outer = **new** Outer();          Outer.Inner inner = outer.**new** Inner();          System.out.printf("%d", outer.getData());          inner.main(args);      }  } |

a) 2010  
b) 1020  
c) Compilation Error  
d) None of these

**Ans.**(b)  
**Explanation:** Inner class defined above though, have access to the private variable data of the Outer class, but declaring a variable data inside an inner class makes it specific to the Inner class with no conflicts in term of variable declaration.



|  |
| --- |
| **interface** OuterInterface  {  **public** **void** InnerMethod();  **public** **interface** InnerInterface      {  **public** **void** InnerMethod();      }  }  **public** **class** Outer **implements** OuterInterface.InnerInterface, OuterInterface  {  **public** **void** InnerMethod()      {          System.out.println(100);      }      **public** **static** **void** main(String[] args)      {          Outer obj = **new** Outer();          obj.InnerMethod();      }  } |

a) 100  
b) Compilation Error  
c) Runtime Error  
d) None of the above

**Ans.** (a)  
**Explanation:** [Nested Interfaces](https://www.geeksforgeeks.org/interface-nested-class-another-interface/) are defined in java. As both the interfaces has declaration of InnerMethod(), implementing it once works for both the InnerInterface and OuterInterface.



|  |
| --- |
| **public** **class** Test **implements** Runnable  {  **public** **void** run()      {          System.out.printf("GFG ");          System.out.printf("Geeks ");      }  **public** **static** **void** main(String[] args)      {          Test obj = **new** Test();          Thread thread = **new** Thread(obj);            thread.start();          System.out.printf("Geeks ");  **try**          {              thread.join();          }  **catch** (InterruptedException e)          {              e.printStackTrace();          }          System.out.println("for ");      }  } |

a) GFG Geeks Geeks for  
b) Geeks GFG Geeks for  
c) Either a or b  
d) Both a and b together

**Ans.** (c)  
**Explanation:**From the statement “thread.start()”, we have two threads [Main thread](https://www.geeksforgeeks.org/main-thread-java/) and “thread” thread. So either “GFG” can be printed or “Geeks”, depend on which thread, thread scheduler schedule.  
For (a), the parent thread after calling start() method is paused and the thread scheduler schedules the child thread which then completes its execution. Following this, the parent thread is scheduled. For (b), the parent thread calls start() method but continues its execution and prints on the console. When join() method is called, the parent thread has to wait for its child to complete its execution. Thread scheduler schedules child thread while the parent waits for the child to complete.



|  |
| --- |
| **public** **class** Test **implements** Runnable  {  **public** **void** run()      {          System.out.printf("GFG ");      }  **public** **static** **void** main(String[] args) **throws** InterruptedException      {          Thread thread1 = **new** Thread(**new** Test());          thread1.start();          thread1.start();          System.out.println(thread1.getState());      }  } |

a) GFG GFG TERMINATED  
b) GFG TERMINATED  
c) Compilation Error  
d) Runtime Error

**Ans.** (d)  
**Explanation:**Invoking start() method on a thread moves the thread to a RUNNABLE state. But invoking start() method on a thread that has already started throws a IllegalThreadStateException because the thread is already in RUNNABLE state.



|  |
| --- |
| **public** **class** Test **extends** Thread **implements** Runnable  {  **public** **void** run()      {          System.out.printf("GFG ");      }  **public** **static** **void** main(String[] args) **throws** InterruptedException      {          Test obj = **new** Test();          obj.run();          obj.start();      }  } |

a) Runtime error   
b) Compilation error  
c) GFG GFG  
d) None of the above  
**Ans.** (c)  
**Explanation:**Test class extends Thread class that has start() method implemented. So invoking start() on an object that extends Thread class invokes run() method defined in the program.



|  |
| --- |
| **class** myThread **implements** Runnable  {  **public** **void** run()      {          Test.obj.notify();      }  }    **public** **class** Test **implements** Runnable  {  **public** **static** Test obj;  **private** **int** data;  **public** Test()      {          data = 10;      }  **public** **void** run()      {          obj = **new** Test();          obj.wait();          obj.data += 20;            System.out.println(obj.data);      }  **public** **static** **void** main(String[] args) **throws** InterruptedException      {          Thread thread1 = **new** Thread(**new** Test());          Thread thread2 = **new** Thread(**new** myThread());            thread1.start();          thread2.start();            System.out.printf(" GFG - ");      }  } |

a) 30 GFG –  
b) GFG – 30  
c) GFG –  
d) Compilation error

**Ans.**(d)  
**Explanation:**An object must first acquire a lock before calling wait() method. Also wait() method throws [Checked](https://www.geeksforgeeks.org/checked-vs-unchecked-exceptions-in-java/) exception(InterruptedException), we must include it in a try-catch block or throws it.



|  |
| --- |
| **import** java.util.concurrent.\*;    **public** **class** Test **implements** Runnable  {  **public** **static** CyclicBarrier barrier = **new** CyclicBarrier(3);  **public** **void** run()      {          System.out.printf(" GFG ");  **try**          {              barrier.await();          } **catch** (InterruptedException | BrokenBarrierException e)          {              e.printStackTrace();          }      }  **public** **static** **void** main(String[] args) **throws** InterruptedException      {          Thread thread1 = **new** Thread(**new** Test());          Thread thread2 = **new** Thread(**new** Test());            thread1.start();          thread2.start();          System.out.printf(" GeeksforGeeks ");  **try**          {              barrier.await();          } **catch** (InterruptedException | BrokenBarrierException e)          {              e.printStackTrace();          }          System.out.printf(" End ");        }  } |

a) GeeksforGeeks GFG GFG End  
b) GFG GeeksforGeeks GFG End  
c) GFG GFG GeeksforGeeks End  
d) All the above

**Ans.**(d)  
**Explanation:**For (a), the parent thread keep executing until it reaches the barrier. The child thread are then scheduled. For (b), thread scheduler scheduler thread1. Once it reaches the barrier, the parent thread is scheduled. Once parent thread reached the barrier, thread2 is scheduled. For (c), Both the child thread are scheduled. Finally when each of the child threads reach the barrier, the parent thread is scheduled.

1. Threads can be created by using two mechanisms :
2. Extending the Thread class
3. Implementing the Runnable Interface
4. **Thread Class vs Runnable Interface**
5. If we extend the Thread class, our class cannot extend any other class because Java doesn’t support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.
6. We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.
7. Using runnable will give you an object that can be shared amongst multiple threads.

|  |
| --- |
| **public** **class** Test  {  **private** **static** **float** temp()      {  **public** **static** **float** sum = 21;  **return**(--(sum));      }  **public** **static** **void** main(String[] args)      {          Test test = **new** Test();          System.out.println(test.temp());      }  } |

a) 21  
b) 20  
c) Compilation error  
d) Runtime error

Ans. (c)  
Explanation: [static variables](https://www.geeksforgeeks.org/variables-in-java/) are associated with the class and are therefore, not allowed inside a method body.



|  |
| --- |
| **public** **class** Test  {  **public** **static** **void** main(String[] args)      {  **int** value = 3, sum = 6 + -- value;    **int** data = --value + ++value / sum++ \* value++ + ++sum  % value--;          System.out.println(data);      }  } |

a) 1  
b) 2  
c) 0  
d) 3

Ans. (b)



|  |
| --- |
| **public** **class** Test  {  **public** **static** **void** main(String[] args)      {  **int** temp = 40;  **if**(temp == 30 && temp/0 == 4)          {              System.out.println(1);          }  **else**          {              System.out.println(2);          }      }  } |

a) 1  
b) 2  
c) Runtime Exception of java.lang.ArithmeticException  
d) Compilation error due to divisibility by 0

Ans. (b)  
Explanation: && operator is evaluated from left to right. If the first expression of && operator evaluates to false, then the second operator is not evaluated. There is no compilation error because divide by 0 is a runtime exception.



|  |
| --- |
| **public** **class** Test  {  **public** **static** **void** main(String[] args)      {  **int** temp = 9;  **int** data = 8;          System.out.println(temp & data);      }  } |

a) 9  
b) 8  
c) 1000  
d) 1001

Ans. (b)  
Explanation: & operator is logical bit-wise and operator in java. The and of 9(1001) and 8(1000) is 1000 which is 8.



|  |
| --- |
| **public** **class** Test  {  **public** **static** **void** main(String[] args)      {  **int** temp = **null**;          Integer data = **null**;          System.out.println(temp + " " + data);      }  } |

a) null null  
b) Compilation error due to temp  
c) Compilation error due to data  
d) Runtime error

Ans. (b)  
Explanation: temp is a primitive data type. Primitive [data types](https://www.geeksforgeeks.org/data-types-in-java/) cannot be assigned null values. data is an instance of class Integer and therefore can hold null values.



|  |
| --- |
| **class** Derived  {  **protected** **final** **void** getDetails()      {          System.out.println("Derived class");      }  }    **public** **class** Test **extends** Derived  {  **protected** **final** **void** getDetails()      {          System.out.println("Test class");      }  **public** **static** **void** main(String[] args)      {          Derived obj = **new** Derived();          obj.getDetails();      }  } |

a) Derived class  
b) Test class  
c) Runtime error  
d) Compilation error

**Ans.**(d)  
**Explanation:**Final and static methods cannot be overridden. For more details Check: [Can we Overload or Override static methods in java ?](https://www.geeksforgeeks.org/can-we-overload-or-override-static-methods-in-java/)



|  |
| --- |
| **class** Derived  {  **public** **void** getDetails(String temp)      {          System.out.println("Derived class " + temp);      }  }    **public** **class** Test **extends** Derived  {  **public** **int** getDetails(String temp)      {          System.out.println("Test class " + temp);  **return** 0;      }  **public** **static** **void** main(String[] args)      {          Test obj = **new** Test();          obj.getDetails("GFG");      }  } |

a) Derived class GFG  
b) Test class GFG  
c) Compilation error  
d) Runtime error

**Ans.**(c)  
**Explanation:**The overriding method must have same signature, which includes, the argument list and the return type. For details, See :[Overriding in Java](https://www.geeksforgeeks.org/overriding-in-java/)



|  |
| --- |
| **class** Derived  {  **public** **void** getDetails()      {          System.out.println("Derived class");      }  }    **public** **class** Test **extends** Derived  {  **protected** **void** getDetails()      {          System.out.println("Test class");      }  **public** **static** **void** main(String[] args)      {          Derived obj = **new** Test();  // line xyz          obj.getDetails();      }  } |

a) Test class  
b) Compilation error due to line xyz  
c) Derived class  
d) Compilation error due to access modifier

**Ans:**(d)  
**Explanation:**The overriding method can not have more restrictive access modifier.



|  |
| --- |
| **import** java.io.IOException;    **class** Derived  {  **public** **void** getDetails() **throws** IOException //line 23      {          System.out.println("Derived class");      }  }    **public** **class** Test **extends** Derived  {  **public** **void** getDetails() **throws** Exception //line 24      {          System.out.println("Test class");      }  **public** **static** **void** main(String[] args) **throws** IOException //line 25      {          Derived obj = **new** Test();          obj.getDetails();      }  } |

a) Compilation error due to line 23  
b) Compilation error due to line 24  
c) Compilation error due to line 25  
d) All the above

**Ans.**(b)  
**Explanation:**The exception thrown by the overriding method should not be new or more broader checked exception. In the code above, [Exception](https://www.geeksforgeeks.org/exceptions-in-java/) is more broader class of checked exception than IOException, so this results in compilation error.



|  |
| --- |
| **class** Derived  {  **public** **void** getDetails()      {          System.out.printf("Derived class ");      }  }    **public** **class** Test **extends** Derived  {  **public** **void** getDetails()      {          System.out.printf("Test class ");  **super**.getDetails();      }  **public** **static** **void** main(String[] args)      {          Derived obj = **new** Test();          obj.getDetails();      }  } |

a) Test class Derived class  
b) Derived class Test class  
c) Compilation error  
d) Runtime error

**Ans.**(a)  
**Explanation:**[super keyword](https://www.geeksforgeeks.org/super-keyword/)is used to invoke the overridden method from a child class explicitly.



|  |
| --- |
| **public** **class** RuntimePolymorphism  {  **public** **static** **void** main(String[] args)      {          A a = **new** B();          B b = **new** B();            System.out.println(a.c + " " + a.getValue()              + " " + b.getValue() + " " + b.getSuperValue());      }  }    **class** A  {  **protected** **char** c = 'A';  **char** getValue()      {  **return** c;      }  }    **class** B **extends** A  {  **protected** **char** c = 'B';  **char** getValue()      {  **return** c;      }  **char** getSuperValue()      {  **return** **super**.c;      }  } |

**Output:** A B B A  
**Explanation:** No magic of polymorphism here; the instance variable c from A is simply hidden in B  
– a.c is ‘A’ because it is so set in the class A  
– a.getValue() returns ‘B’ because the object is of type B



|  |
| --- |
| **public** **class** RuntimePolymorphism  {  **public** **static** **void** main(String[] args)      {          A a = **new** B();          B b = **new** B();          System.out.println(a.c + " " + a.getValue() +              " " + b.getValue() + " " + b.getSuperValue());      }  }    **class** A  {  **protected** **char** c = 'A';  **char** getValue()      {  **return** c;      }  }  **class** B **extends** A  {  **protected** **char** c = 'B';  **char** getSuperValue()      {  **return** **super**.c;      }  } |

**Output:** A A A A  
**Explanation:** Concept of polymorphism for methods cannot be used here, since in class B there is no function overloading a method in class A. Refer [run-time polymorphism](https://www.geeksforgeeks.org/dynamic-method-dispatch-runtime-polymorphism-java/) for more details



|  |
| --- |
| **class** test  {  **public** **static** **int** y = 0;  }    **class** HasStatic  {  **private** **static** **int** x = 100;    **public** **static** **void** main(String[] args)      {          HasStatic hs1 = **new** HasStatic();          hs1.x++;            HasStatic hs2 = **new** HasStatic();          hs2.x++;            hs1 = **new** HasStatic();          hs1.x++;            HasStatic.x++;          System.out.println("Adding to 100, x = " + x);            test t1 = **new** test();          t1.y++;            test t2 = **new** test();          t2.y++;            t1 = **new** test();          t1.y++;            System.out.print("Adding to 0, ");          System.out.println("y = " + t1.y + " " + t2.y + " " + test.y);      }  } |

**Output:**

Adding to 100, x = 104

Adding to 0, y = 3 3 3

**Explanation:** Properties of static are shown in this example. When a variable is declared as static, then a single copy of variable is created and shared among all objects at class level. Static variables are, essentially, global variables. All instances of the class share the same static variable. See more at: [static keyword](https://www.geeksforgeeks.org/static-keyword-java/)



|  |
| --- |
| **public** **class** Except  {  **public** **static** **void** main(String[] args)      {  **try**          {  **throw** **new** Error();          }  **catch** (Error e)          {  **try**              {  **throw** **new** RuntimeException();              }  **catch** (Throwable t)              {                }          }              System.out.println("phew");      }  } |

**Output:** phew  
**Explanation:** Its legal to throw and handle errors and runtime exceptions. RuntimeException is a sub-subclass of Throwable. See [exceptions](https://www.geeksforgeeks.org/checked-vs-unchecked-exceptions-in-java/) for more details



|  |
| --- |
| **public** **class** Boot  {  **static** String s;  **static**      {          s = "";      }      {          System.out.println("GeeksforGeeks ");      }  **static**      {          System.out.println(s.concat("practice.GeeksforGeeks "));      }      Boot()      {          System.out.println(s.concat("Quiz.GeeksforGeeks"));      }  **public** **static** **void** main(String[] args)      {  **new** Boot();          System.out.println("Videos.GeeksforGeeks");      }  } |

**Output:**

practice.GeeksforGeeks

GeeksforGeeks

Quiz.GeeksforGeeks

Videos.GeeksforGeeks

**Explanation:** static init blocks run before instance init blocks (in the order in which they appear, respectively). The order of initialization constructors and initializer block doesn’t matter, initializer block is always executed before constructor. See [static block](https://www.geeksforgeeks.org/g-fact-26-the-initializer-block-in-java/) for details



|  |
| --- |
| **class** A  {  **public** A(String s)      {          System.out.print("A");      }  }    **public** **class** B **extends** A  {  **public** B(String s)      {          System.out.print("B");      }  **public** **static** **void** main(String[] args)      {  **new** B("C");          System.out.println(" ");      }  } |

**Output:**Compilation fails

prog.java:12: error: constructor A in class A cannot be applied to given types;

{

^

required: String

found: no arguments

reason: actual and formal argument lists differ in length

1 error

**Explanation:** The implied super() call in B’s constructor cannot be satisfied because there isn’t a no-arg constructor in A. A default, no-arg constructor is generated by the compiler only if the class has no constructor defined explicitly.For detail See – [Constructors in Java](https://www.geeksforgeeks.org/constructors-in-java/)



|  |
| --- |
| **class** Clidder  {  **private** **final** **void** flipper()      {          System.out.println("Clidder");      }  }    **public** **class** Clidlet **extends** Clidder  {  **public** **final** **void** flipper()      {          System.out.println("Clidlet");      }  **public** **static** **void** main(String[] args)      {  **new** Clidlet().flipper();      }  } |

**Output:**

Clidlet

**Explanation:** Although a final method cannot be overridden, in this case, the method is private, and therefore hidden. The effect is that a new, accessible, method flipper is created. Therefore, no polymorphism occurs in this example, the method invoked is simply that of the child class, and no error occurs.



|  |
| --- |
| **class** Alpha  {  **static** String s = " ";  **protected** Alpha()      {          s += "alpha ";      }  }  **class** SubAlpha **extends** Alpha  {  **private** SubAlpha()      {          s += "sub ";      }  }    **public** **class** SubSubAlpha **extends** Alpha  {  **private** SubSubAlpha()      {          s += "subsub ";      }  **public** **static** **void** main(String[] args)      {  **new** SubSubAlpha();          System.out.println(s);      }  } |

**Output:**

alpha subsub

**Explanation:** SubSubAlpha extends Alpha! Since the code doesnt attempt to make a SubAlpha, the private constructor in SubAlpha is okay.



|  |
| --- |
| **public** **class** Juggler **extends** Thread  {  **public** **static** **void** main(String[] args)      {  **try**          {              Thread t = **new** Thread(**new** Juggler());              Thread t2 = **new** Thread(**new** Juggler());          }  **catch** (Exception e)          {              System.out.print("e ");          }      }  **public** **void** run()      {  **for** (**int** i = 0; i < 2; i++)          {  **try**              {                  Thread.sleep(500);              }  **catch** (Exception e)              {                  System.out.print("e2 ");              }              System.out.print(Thread.currentThread().getName()+ " ");          }      }  } |

**Output:** No Output  
**Explanation:** In main(), the start() method was never called to start ”t” and ”t2”, so run() never ran.  
For detail: See [Multithreading in Java](https://www.geeksforgeeks.org/multithreading-in-java/)



|  |
| --- |
| **class** Grandparent  {  **public** **void** Print()      {          System.out.println("Grandparent's Print()");      }  }    **class** Parent **extends** Grandparent  {  **public** **void** Print()      {          System.out.println("Parent's Print()");      }  }    **class** Child **extends** Parent  {  **public** **void** Print()      {  **super**.**super**.Print();          System.out.println("Child's Print()");      }  }    **public** **class** Main  {  **public** **static** **void** main(String[] args)      {          Child c = **new** Child();          c.Print();      }  } |

**Output:**Compiler Error in super.super.Print()  
**Explanation:** In Java, it is not allowed to do super.super. We can only access Grandparent’s members using Parent. See [Inheritance in Java](https://www.geeksforgeeks.org/inheritance-in-java/)



|  |
| --- |
| **public** **class** Test  {  **public** **static** **void** main(String[] args)      {  **int** value = 554;          String var = (String)value;  //line 1          String temp = "123";  **int** data = (**int**)temp; //line 2          System.out.println(data + var);      }  } |

a) 677  
b) Compilation error due to line 1  
c) Compilation error due to line 2  
d) Compilation error due to line 1 and line 2

**Ans.**(d)  
**Explanation:**Converting from int to String as well as converting from String to int is not allowed in java.



|  |
| --- |
| **public** **class** Test  {  **public** **static** **void** main(String[] args)      {  **double** data = 444.324;  **int** value = data;          System.out.println(data);      }  } |

a) 444.324  
b) 444  
c) Runtime error  
d) Compilation error

**Ans.**(d)  
**Explanation:**Converting from a bigger data type to a smaller data type is not allowed in java as it is a lossy conversion.



|  |
| --- |
| **public** **class** Test  {  **public** **static** **void** main(String[] args)      {  **double** data = 444.324;  **int** sum = 9;  **float** value = 5.1f;          System.out.println(data + sum + value);        }  } |

a) 444.32495.1  
b) 456  
c) 458.42399  
d) 458.4

**Ans.**(c)  
**Explanation:**If one of the operands is long, double or float, the entire expression is converted to long, double or float respectively.



|  |
| --- |
| **public** **class** Test  {  **public** **static** **void** main(String[] args)      {  **byte** var = 1;          var = (**byte**) var \* 0; //line 1  **byte** data = (**byte**) (var \* 0); //line 2          System.out.println(var);      }  } |

a) 0  
b) Compilation error due to line 1  
c) Compilation error due to line 2  
d) Compilation error due to line 1 and line 2

**Ans.**(b)  
**Explanation:**When the expressions are evaluated, the data type of the result is implicitly changed to a larger data type and therefore, explicit recasting has to be done as shown in line 2. On the other hand, line 1 shows compilation error because the expression on the right side has data type as int whereas left side it is byte.

|  |
| --- |
| **public** **class** Test  {  **public** **static** **void** main(String[] args)      {          System.out.println((100/25.0)\*Integer.parseInt("5") + 50);      }  } |

a) Compilation error  
b) 70  
c) 70.0  
d) Runtime error

**Ans.**(c)  
**Explanation:** If a double value is used in an expression then the output is returned in double format rather than an int.



**1. null is Case sensitive:** null is literal in Java and because keywords are **case-sensitive** in java, we can’t write NULL or 0 as in C language.

|  |
| --- |
| **public** **class** Test  {  **public** **static** **void** main (String[] args) **throws** java.lang.Exception      {          // compile-time error : can't find symbol 'NULL'          Object obj = NULL;            //runs successfully          Object obj1 = **null**;      }  } |

Output:

5: error: cannot find symbol

can't find symbol 'NULL'

^

variable NULL

class Test

1 error

**2. Reference Variable value:** Any reference variable in Java has default value null.

|  |
| --- |
| **public** **class** Test  {  **private** **static** Object obj;  **public** **static** **void** main(String args[])      {          // it will print null;          System.out.println("Value of object obj is : " + obj);      }  } |

Output:

Value of object obj is : null

**3. Type of null:** Unlike common misconception, null is not Object or neither a type. It’s just a special value, which can be assigned to any reference type and you can type cast null to any type  
Examples:

// null can be assigned to String

String str = null;

// you can assign null to Integer also

Integer itr = null;

// null can also be assigned to Double

Double dbl = null;

// null can be type cast to String

String myStr = (String) null;

// it can also be type casted to Integer

Integer myItr = (Integer) null;

// yes it's possible, no error

Double myDbl = (Double) null;

**4. Autoboxing and unboxing :** During auto-boxing and unboxing operations, compiler simply throws Nullpointer exception error if a null value is assigned to primitive boxed data type.

|  |
| --- |
| **public** **class** Test  {  **public** **static** **void** main (String[] args) **throws** java.lang.Exception      {              //An integer can be null, so this is fine              Integer i = **null**;                //Unboxing null to integer throws NullpointerException  **int** a = i;      }  } |

Output:

Exception in thread "main" java.lang.NullPointerException

at Test.main(Test.java:6)

**5. instanceof operator:** The java instanceof operator is used to test whether the object is an instance of the specified type (class or subclass or interface). At run time, the result of the instanceof operator is true if the value of the Expression is not null.  
This is an important property of instanceof operation which makes it useful for type casting checks.

|  |
| --- |
| **public** **class** Test  {  **public** **static** **void** main (String[] args) **throws** java.lang.Exception      {          Integer i = **null**;          Integer j = 10;            //prints false          System.out.println(i **instanceof** Integer);            //Compiles successfully          System.out.println(j **instanceof** Integer);      }  } |

Output:

false

true

**6. Static vs Non static Methods:** We cannot call a non-static method on a reference variable with null value, it will throw NullPointerException, but we can call static method with reference variables with null values. Since static methods are bonded using static binding, they won’t throw Null pointer Exception.

|  |
| --- |
| **public** **class** Test  {  **public** **static** **void** main(String args[])      {          Test obj= **null**;          obj.staticMethod();          obj.nonStaticMethod();      }    **private** **static** **void** staticMethod()      {          //Can be called by null reference          System.out.println("static method, can be called by null reference");      }  **private** **void** nonStaticMethod()      {          //Can not be called by null reference          System.out.print(" Non-static method- ");          System.out.println("cannot be called by null reference");        }    } |

Output:

static method, can be called by null referenceException in thread "main"

java.lang.NullPointerException

at Test.main(Test.java:5)

**7. == and !=** The comparison and not equal to operators are allowed with null in Java. This can made useful in checking of null with objects in java.

|  |
| --- |
| **public** **class** Test  {  **public** **static** **void** main(String args[])      {        //return true;      System.out.println(**null**==**null**);        //return false;      System.out.println(**null**!=**null**);        }  } |

Output:

true

False



|  |
| --- |
| / Java program to illustrate  // redeclaring a variable  // in initialization block  **public** **class** Example3  {  **public** **static** **void** main(String[] args)      {          // x is integer  **int** x = 0;            // redeclaring x as long will not work  **for**(**long** y = 0, x = 1; x < 5; x++)          {              System.out.print(x + " ");          }        }  } |

Example3.java:12: error: variable x is already defined in method main(String[])

for(long y = 0, x = 1; x < 5; x++)

Here, x was already initialized to zero as integer and is being re-declared in the loop with data type long.

But this problem can be fixed by slightly modifying the code. Here, the variables x and y are declared in a  
different way.

|  |
| --- |
| // Java program to illustrate  // redeclaring a variable  // in initialization block  **public** **class** Example3  {  **public** **static** **void** main(String[] args)      {          // x is integer  **int** x = 0;  **long** y = 10;    **for** (y = 0, x = 1; x < 5; x++)          {              System.out.print(x + " ");          }        }  } |

Output:

1 2 3 4