**메서드**

|  |  |
| --- | --- |
| 1 | Product라는 이름의 클래스를 정의하고, 그 클래스안에 아래의 조건에 맞도록 메서드를 정의하세요  (1) return 값이 3인 getPrice() 메서드를 정의하세요.  (2) return 값이 “ java” 인 getName() 메서드를 정의하세요.  (3) return 값이 없는 setTest() 메서드를 정의하세요.  (4) return 값이 없으며 매개변수가 int a 와 int b인 setMoney() 메서드를 정의하세요.  (5) return 값이 false 이며 매개변수가 String s인 setStr() 메서드를 정의하세요. |
| 2 | 아래의 설명 중 맞는 것은?  **class** Product {    **int** price; //(가)    **public** **void** setPrice(**int** p){  p=500; //(라)  price = p;  }      **public** **static** **void** main(String[] args) {  **int** price=300; //(나)    Product p = **new** Product();  p.setPrice(price); //(다) price 가 들어가는 것이아니라, 300 이라는 값이 들어간다.    System.*out*.println(price); //(마)300  System.*out*.println(p.price); //(바)300  }  }  (1) (가)에서 개발자가 price 변수값을 초기화 하지 않았으므로 price에는 아무런 값이 들어있지 않다-컴파일 타이밍에 0으로 자동 초기화  (2) (나)에서 price는 Product 클래스의 price 변수를 가리킨다-main 의 지역변수  (3) (라)에서 매개변수 p가 전달받은 값은 300이지만, 500으로 변경이 일어나므로 main() 메서드에  선언된 price는 이 시점에 500으로 변경이 일어난다. 따라서 (마)의 출력 결과는 500이 된다.-300  (4) (다)에서 setPrice 호출시 전달되는 것은 price의 값인 300일 뿐이며, price변수 자체가 전달되는  것은 아니므로, (라)에서 매개변수의 값을 500으로 변경하는 행위는 main() 함수 (나)에 정의된  price 변수에 영향을 미치지 않는다.  (5) (4)번과 같이 변수자체가 아닌 변수의 값이 복사되어 전달되는 경우를 자바에서는 **값에 의한 호출**  **(call by value)** 이라고 한다 |
| 3 | 다음 설명 중 틀린 것은?  (1) 자바언어는 자료형이 총 4개이다.  (2) 사물을 표현하는 객체 자료형도 자료형이지만, 메서드의 리턴값이나 매개변수는 자바의 기본자료  형만 사용할 수 있다.  (3) 객체자료형은 리턴값은 가능하나, 매개변수에는 사용할 수 없다.  (4) 자바의 자료형에는 기본자료형 외에 객체자료형도 있으므로 메서드의 매개변수나, 리턴값에  객체자료형을 사용하는데는 아무런 문제가 없다. |
| 4 | **class** Printer {  String color="yellow";    }  **class** UsePrinter{  **int** k=3; //(가)    **public** **void** printerTest(Printer p){  p.color="red"; //(마)    }    **public** **static** **void** main(String[] args){  k=5; //(나)  Printer p = **new** Printer();//(바)  p.color="black";    UsePrinter up = **new** UsePrinter();  up.printerTest(p); //(다)  System.*out*.println(p.color);//(라)    }    }  다음 설명 중 맞는 것은?  (1) (나)에 의해 (가)의 값이 5로 변경될 것이다.  (2) (다)에서 p는 일반 변수가 아닌 레퍼런스 변수이므로 printerTest() 메서드에 주소값이 전달되는데,  자바에서는 이러한 호출 방식을 **주소값의 전달(call by reference)**이라 한다  (3) (마)에서 color 값이 red로 변경되더라도, (바)의 프린터 p의 color에는 영향을 미치지 않는다.  (4) (라)에서 출력되는 결과는 black 이다. |
| 5 | 아래 코드에 대한 설명 중 틀린 것은?  **class** Computer{  **int** speed=300;  }  **class** UseComputer {  **public** **void** setting(Computer c , **int** s){  c.speed=s;  s=50; //(다)  }    **public** **static** **void** main(String[] args) {  **int** k=500; //(가)    UseComputer uc = **new** UseComputer();  Computer com = **new** Computer();//(바)  com.speed=100;    uc.setting(com , k); //(나)    System.*out*.println(com.speed); //(라)500  System.*out*.println(k); //(마)500    }  }  (1) (가) 변수는 지역변수로서 stack 영역에 쌓인다.  (2) (나)에서 메서드 호출시 com변수에는 객체의 주소값이, k변수에는 변수의 값이 들어있다.  (3) (다)에 의해 (가)의 변수값이 50으로 변경될 것이다.  (4) (라)에서 출력되는 결과는 100 이다.  (5) (마)에서 출력되는 결과는 500 이다. |
| 6 | 아래의 클래스에 대한 설명으로 틀린 것은?  **class** Rose{  **int** leaf=0;  String color="red";    **public** **void** fall(){  leaf=0;  }  }  **class** UseRose{  **public** **static** **void** main(){  Rose r1=**new** Rose(); // (가) 생성자 메서드:클래스명과 일치한다.  r1.fall(); //(나)  r1.bloom(); //(다)  }  }  (1) (다)에서의 bloom() 은 Rose 클래스에 존재하지 않는 메서드이므로 컴파일 에러가 발생한다.  (2) (나)에서의 fall()은 Rose 클래스의 멤버메서드를 호출하고 있는 것이다.  (3) (가)에서 장미 인스턴스를 생성하기 위한 new 연산자의 뒤에오는 Rose() 는 괄호가 붙었으므로,  메서드로 보아야 한다.-생성자 메서드  (4) (가)에서 new 연산자뒤에 오는 Rose() 메서드는 개발자가 정의한 적이 없으므로 존재하지 않는  메서드이며 컴파일 에러가 발생할 것이다. |
| 7 | 생성자 메서드에 대한 설명 중 틀린 것은?  (1) 객체가 생성되어질때 무조건적으로 호출이 강제되는 메서드이다.  (2) 생성자메서드의 이름은 반드시 클래스명과 일치해야 한다.  (3) 생성자에는 반환형을 두어서는 않된다.-반환형을 두면 일반 메서드가 된다. New 가 안된다.  (4) 생성자는 개발자가 정의하지 않으면 객체생성시 존재하지 않는 메서드로 인한 에러가 발생하게  되므로, 컴파일러에 의해 자동으로 정의되어 진다.  (5) 개발자가 생성자를 정의해도 디폴트 생성자는 컴파일러에 의해 계속 존재하게 된다.  (6) Sun사가 객체의 생성시, 생성자 메서드를 무조건 적으로 강제하고 있는 이유는 개발자로 하여금  클래스로부터 만들어지는 모든 객체에 대해 초기화 작업의 기회를 주기 위해서이다. |
| 8 | 생성자 메서드와 일반메서드의 차이점에 대한 설명 중 틀린 것은?  (1) 생성자가 아니더라도 객체에 원하는 작업 내용이 있다면 일반 메서드로도 충분히 감당이 가능하다.  (2) 생성자도 메서드이다.  (3) 실행 시기적 관점에서 본다면 생성자와 일반 메서드의 차이점은 단지 작업의 착수 시점이 다를  뿐이며, 무조건적인 호출을 강제한 이유는 개발자로 하여금 객체생성 초기에 원하는 작업을  할 수 있도록 배려한 것이다.  (4) 생성자에 반환형을 둘 수 있다.-안된다. 일반 메서드가 됨.  (5) 현실에 비유하자면 생성자는 사물 제작 초기부터 해야 할 작업에, 일반메서드는 사물을 제작한  이후에 해야 하는 작업에 비유 할 수 있다. |
| 9 | 생성자 메서드에 대한 설명 중 틀린 것은?  (1) 현실에서 사물을 만들 때 고려할 사항이 있고, 만들어진 이후에 고려해야 할 사항이 있듯, 생성자  메서드는 객체지향 프로그래머가 사물을 생성할 때, 생성 초기에 해야 할 작업을 위해 기회를  준것이다.  (2) Sun에서 생성자 호출을 new 연산자 바로 뒤에 존재시키는 이유는 객체가 태어나는 타이밍을  놓치지 않기 위해서다.  (3) new 연산자 뒤에서 무조건적으로 호출을 강제하는 이유는 사물이 생성되는 최초 시점에서의  작업을 염두해 둔 것이다.  (4) 생성자는 메서드가 아니다.-메서드 이다.  (5) 생성자는 개발자가 정의할 수 없다.-있다 |
| 10 | 아래의 클래스를 생산 시점부터 빨간색인 하이힐을 만들수 있는 클래스로 완성하시오  **class** HighHeel{  String color="None Color";  public HighHeel(String color){ // 생성자에 void 를 주면 생성자로 간주하지 않는다.  // 에러는 안난다.  this.color=color;  }  } |
| 11 | 아래의 클래스를 생산 시점부터 검정색으로 만들수 있는 기능 및 생산한 이후에 언제든 색상을  빨간색으로 변경 할 수 있는 클래스로 완성하시오  **class** SportsCar{  String color="None Color";  public SportsCar(String color){  this.color=color;  }  Public void setColor(String color){  this.color=color;  }  } |
| 12 | 다음 설명 중 틀린 것은?  (1) 자바에서는 하나의 클래스내에 메서드의 이름을 중복 정의해서는 않된다.-된다(overloading)  (2) 메서드의 이름은 중복되어도 인수의 자료형과 개수를 달리하면 메서드간 구분이 가능하므로  이러한 코드 작성법은 자바에서 허용된다.  (3) 중복된 메서드명은 하나의 클래스내에 2개까지만 허용된다.  (4) 중복 메서드를 허용하는 자바의 코드 기법을 오버로딩(Overloading)이라고 한다.  (5) 개발자는 변수나 메서드명에 직관성을 부여하는 것이 대부분인데, 어렵게 지어놓은 메서드명을  중복금지라는 규칙 때문에 다시 사용할 수 없다면 프로그램 개발시 많은 불편함을 초래한다,  또한 상당히 비슷한 기능을 갖는 메서드들 임에도 불구하고, 이름을 달리 정의하게 될 경우 ,  메서드 이름으로만으로도 그 수행 내용을 예측할 수 있었던 직관성을 잃게되어 많은 불편함을  초래할 수 있다.  따라서 오버로딩을 사용하면 개발자는 메서드명의 중복을 피하기 위해 명명작업 때문에 고민하지  않아도 되며, 단지 인수의 자료형과 개수로 메서드들간의 구분이 가능하도록 할 수 있다. |
| 13 | **OCJP 시험문제 - 오버로딩**  Given:  1. public class A {  2. public void doit() {  3. }  4. public String doit() { // 반환형이 다르기 때문에 overloading 이 아니다. 에러난다.  5. return “a”;  6. }  7. public double doit(int x) { // 반환형이 지정되어 overloading 아님.  8. return 1.0;  9. }  10.}  What is the result?  A. An exception is thrown at runtime.  B. Compilation fails because of an error in line 7.  C. Compilation fails because of an error in line 4.  D. Compilation succeeds and no runtime errors with class A occur. |
| 14 | 아래의 클래스에 대한 설명 중 맞는 것은?  **class** Doll{  **static** String *name*="라라";  **int** price;    **public** Doll(**int** p){  price=p;  }  **public** **static** **void** main(String[] args){  Doll.*name*="테디베어"; //(가) static 이므로 가능  Doll d=**new** Doll(); //(나) 생성자에 인수가 있는데 지정이 안되어 에러남.  d.*name*="미미"; //(다)  }  }  (1) 이 클래스의 멤버변수는 2개이다.  (2) 개발자가 생성자를 정의하지 않았으므로 디폴트 생성자만 존재한다.  (3) (가) 에서 아직 인형의 인스턴스가 생성되지 못한 상태에서 name변수를 접근하려 했으므로,  컴파일 에러가 발생한다.  (4) (나) 는 인형을 메모리에 올리는 코드이므로 잘못된 부분이 없다.-인수 없이 넘겨서 에러  (5) static 으로 선언된 name 변수는 (다) 와 같은 접근방법으로 사용이 불가능하다.-static 이라 가능  (6) 이 클래스는 개발자가 생성자를 정의하였으므로, (나)에서 인형의 인스턴스를 생성하려할 때 Doll()  은 존재하지 않는 메서드이므로, 컴파일 에러가 날 것이다. |
| 15 | 아래의 클래스에 대한 설명 중 맞는 것은?  **class** Movie{  String title;  **int** price;  **public** Movie(**int** p){  price=p;  }  **public** Movie(String t){  title=t;  }  **public** **static** **void** show(){  System.out.println(“movie start!!”);  }  **public** **static** **void** main(String[] args){  Movie mv1=**new** Movie(); //(가) 에러  Movie.*show*(); //(나)  }  }  (1) 이 클래스는 생성자가 2개 정의되었으므로 컴파일 에러가 발생할 것이다.-생성자도 메서드로 오버로딩가능  (2) (가)에서 Movie 클래스의 인스턴스가 오류없이 생성될 것이다.-인수 없어서 에러남.  (3) Movie 클래스에는 컴파일러에 의한 디폴트 생성자가 존재한다.-존재하지 않는다.  (4) 생성자 오버로딩이 적용된 클래스이며, 디폴트 생성자는 존재하지 않는다.  (5) (나)의 show() 메서드는 멤버메서드이므로 클래스명으로 접근할 수 없다.-static 이므로 클래스명으로 접근가능 |