1. Java의 장점
   1. 가장 많이 쓰이는 개발언어
   2. Write one, run anywhere
      1. Java는 공통 실행환경 JVM이 존재
         1. 기존의 언어는 한가지 운영체제에 맞춰 제작 -> 다른 운영체제에서 오류
         2. 운영 체제 위에서 작동
         3. 어느 환경에서도 동작 가능하게 작동한다
   3. 세상과 닮아 있는 개발 언어라 코드를 이해하기 쉽고 유지보수가 쉽다.
      1. 물건을 “객체”로 지정
      2. Class를 통해 객체를 생성
   4. 안정성이 높아 오류를 방지 & 보안상 안전
      1. Java로 작성된 문맥을 컴파일러가 먼저 오류체크
      2. Java에서 지원하는 보안 기술이 많이 있다.
   5. 대규모 앱 개발이 가능
      1. 네트워크, 데이터베이스 연결, 통신 작업을 처리하는 API가 존재
   6. 다양한 라이브러리와 개발 도구가 존재
      1. 앱 개발에 필요한 작업을 보다 쉽게 처리할 수 있다.
2. 객체지향
   1. 객체
      1. 특징 및 행동을 갖고 있다.
3. JVM
   1. Java Virtual Machine : 자바 가상 머신
   2. 여러가지 기기 위에 Java 프로그램을 실행시킬 수 있는 가상의 기기를 만들어주는 것을 의미
   3. PC 모바일 서버 환경에서도 JVM이 있다면 Java가 동작할 수 있는 환경이 된다.
   4. 구성
      1. 바이트 코드
         1. Java 컴파일러가 변환한 코드
            1. 컴파일러 : .java -> .class라는 파일로 변환하는 파일
      2. 인터 프리터
         1. Java .class 코드 해석기
         2. 운영체제가 읽은 바이트 코드를 기계가 실행할 수 있는 기계어로 변역
      3. JIT 컴파일러
         1. Just In Time
         2. 인터프리터 효율을 높여주는 서포터 해석기
      4. 메모리 영역
         1. Java 데이터를 저장하는 영역
         2. 운영체제로부터 JVM이 할당 받은 메모리 영역
      5. 클래스 로더
         1. Java .calss 바이트 코드를 메모리 영역에 담는 운반기
         2. JVM으로 class를 불러와서 메모리에 저장
      6. 가비지 컬렉터
         1. Java 쓰레기 청소기
         2. 메모리 영역에서 안 쓰는 데이터를 주기적으로 흡수해가는 청소기
   5. JDK( Java Development Kit )
      1. 자바 개발 키트
      2. JRE의 기능을 포함하고 있다.
         1. JRE( Java Runtime Environment )
         2. JVM은 JRE를 통해 설치
         3. 하지만, JRE만 있다면 Java 프로그램을 실행만 시킬 수 있다

.class 파일만 실행 가능

* + - 1. JDK가 javac명령을 통해 .java파일을 실행 가능한 .class파일로 변환
    1. JavaCompiler( javac )의 기능을 포함하고 있다
    2. 코드를 디버깅하는 jdb등의 기능을 갖고 있다.

요약

텍스트, 스크린샷, 도표, 폰트이(가) 표시된 사진

자동 생성된 설명

Java 문법

1. 변수와 상수
   1. 변수 : 변하는 수
   2. 상수 : 변하지 않는 수
      1. 앞에 final 키워드를 통해 상수로 지정 가능
2. 기본형 변수과 참조형 변수
   1. 기본형 변수
      1. 실제 값을 저장하는 저장공간
      2. 숫자, 문자, Boolean
      3. Float
         1. 부동 소수점
            1. 가수와 지수를 구분해서 저장하고, 이를 곱한 값을 저장
            2. Long 보다 더 넓은 범위를 표현할 수 있는 방법
      4. Stack에 원본 값이 저장
         1. Stack : 정적으로 할당된 메모리 영역
         2. 크기가 정해진 데이터를 저장하는데 사용( 메모리 주소도 포함 )
   2. 참조형 변수 ( 주소형 변수 )
      1. 값은 별도로 저장, 원본값의 주소값을 저장
      2. String, Object, Array, List…
      3. Heap에 원본 값이 저장
         1. Heap : 동적으로 할당된 메모리 영역
         2. 가변 데이터의 원본을 저장
         3. Stack에는 원본 값이 저장된 Heap 주소를 저장
   3. 래퍼 클래스 변수
      1. Wrapper Class
         1. 기본형 변수를 Class로 한번 랩핑한 변수
      2. Wrapper Class만 지원하는 Method 사용 가능
      3. 박싱 & 언박싱
         1. 박싱 : 기본 타입 -> 래퍼 클래스 변수로 변환
         2. 언박싱 :
      4. 왜 써야 하는가 ?
         1. Class를 사용하여 변수를 관리하면 객체지향의 많은 기능을 사용할 수 있게된다.
3. 얕은 복사 & 깊은 복사
   1. 얕은 복사
      1. 참조형 변수간의 ‘=’연산에서 발생
      2. 실제 값이 아닌 주소 값만 복사
      3. 복사한 데이터 변조시 원본 데이터 또한 같이 변조
   2. 깊은 복사
      1. 얕은 복사처럼 주소만 복사하는 것이 아닌 실제 값을 복사
      2. for문을 통해 하나하나 옮겨줄 수 있다.
         1. Clone()
            1. int[] b = a.clone() 형태로 사용

2차원 배열에 사용할 경우 얕은 복사로 동작한다.

배열 자체는 복사해오지만, 배열 내 원소는 주소를 복사하기 때문

* + - 1. Array의 경우 Arrays.copyOf
         1. int[] b = Arrays.copyOf(a, a.length) 형태로 사용

java.util.Arrays를 import해야 사용 가능하다

1. Collection
   1. 자료구조 중 하나
   2. 배열보다 다수의 **참조형 데이터 & Wrapper**를 더 쉽고 효과적으로 처리할 수 있는 기능을 담고 있다.
      1. 크기 자동 조정 / 추가 / 수정 / 삭제 / 반복 / 순회 / 필터 / 포함확인 등
   3. 종류
      1. List
         1. 동적 배열 ( 가변형 배열 ) / 순서가 있는 데이터의 집합
         2. ArrayList, LinkedList로 구분
            1. ArrayList

순서가 정해져 있다.

메모리가 연속적이다.

데이터를 추가 / 삭제할 경우, 오버헤드가 발생한다.

참조가 잦은 상황에서 사용이 용이하다.

* + - * 1. LinkedList

순서가 정해져 있지 않다.

메모리가 불연속적이다.

데이터를 추가 / 삭제가 잦은 경우, 사용이 용이하다.

참조가 잦은 상황에서 사용하기엔 어려움이 있다.

* + - 1. 생성 시점에서 작은 연속된 공간에 **참조형 변수**를 담아놓는다.
      2. 값이 추가될 때, 공간이 필요하면 더 큰 공간을 받아서 저장
    1. Queue
       1. FIFO ( First in First out ) 먼저 들어간 순서대로 데이터를 뺄 수 있다.
       2. 대기열과 같은 순서대로 참조해야 할 때 용이하다.
       3. **생성자가 없는 인터페이스**
          1. New LinkedList와 같은 queue가 아닌 다른 collection으로 생성
       4. 대표 메소드
          1. Add, peek( 맨 앞에 있는 데이터 확인 ), poll( 데이터 꺼내기 )
    2. Stack
       1. FILO( First in Last out ) 늦게 들어간 순서대로 데이터를 뺄 수 있다.
       2. 데이터 중복 처리 및 최근 데이터를 확인할 때 용이하다.
    3. Set
       1. 중복을 허용하지 않는 데이터 집합 / 순서가 없다
       2. 생성자가 없는 껍데기 HashSet을 이용해서 생성가능
       3. 대표 메소드
          1. Add( 추가 ) , contains( 데이터 확인 )
    4. Map
       1. Key, Value 형태로 이루어진 데이터 집합 / 순서가 없다
          1. Key

배열의 인덱스와 같은 역할로 데이터를 찾을 때, Key를 기준으로 검색한다.

중복 허용 X

* + - * 1. Value

Key에 따른 값

중복 허용 O

* + - 1. 생성자가 없는 껍대기
         1. HashMap, TreeMap으로 응용

객체지향 프로그래밍

1. 특징
   1. 캡슐화, 상속, 다형성, 추상화
      1. 캡슐화
         1. 속성, 메서드를 하나로 묶어서 객체로 만드는 것
         2. 실제 내부 구현 내용은 외부에서 알 수 없게 감추는 것 또한 의미
         3. 외부에서 필드, 메서드를 잘 못 사용해서   
            변하지 않게 하기 위함
      2. 상속
         1. 부모 객체가 갖고 있는 필드, 메서드를 자식 객체 또한   
            갖고 있게 만들 수 있다.
         2. 상속관계로 묶으면서 객체 사이 구조를 파악하기 좋으며,   
            부모 객체를 변경할 경우, 자식 객체 또한 변경되기 때문에   
            일관성을 유지할 수 있다.
      3. 다형성
         1. 객체를 사용할 때, 객체가 갖고 있는 고유한 특성에 따라 다른 여러가지 형태로 재구성된다.
      4. 추상화
         1. 객체에서 공통된 부분들을 모아 상위 개념으로 새롭게 선언하는 것
2. 객체
   1. 세상에 존재하는 물체, 식별이 가능한 개념
   2. 속성( 필드 )과 행위( 메서드 )로 구성되어있다.
      1. 게임 캐릭터는 HP/능력치/이동속도와 같은 속성이 있다.
      2. 게임 캐릭터는 공격/회피/이동과 같은 행위를 할 수 있다.
   3. 객체 모델링
      1. 현실 세계에 있는 객체를 소프트웨어 객체로 설계하는 것
   4. 객체간의 상호작용
      1. 사람과 자동자가 상호작용을 하듯, 소프트웨어 속 객체 또한 메서드를 통해 서로 상호작용한다.
   5. 소프트웨어 객체 간의 관계
      1. 사용관계
         1. 차 <- 사람
      2. 포함관계
         1. 차 <- 핸들, 문, 타이어
      3. 상속관계
         1. 탈것 <- 차, 기차
   6. 객체와 클래스
      1. 객체를 생성하기 위해 필요한 설계도
      2. Class 기반으로 생성된 객체를 인스턴스라고 부른다.
3. Class
   1. 필드
      1. 객체의 고유한 데이터, 상태 데이터, 객체 데이터로 분류
         1. 컴퓨터 회사, 색상 / 작동 상태 / 그래픽카드, CPU 등등
      2. 필드의 기본값

|  |  |
| --- | --- |
| 데이터 타입 | 기본값 |
| byte | 0 |
| char | \u0000 (공백) |
| short | 0 |
| int | 0 |
| long | 0L |
| float | 0.0F |
| double | 0.0 |
| boolean | false |
| 배열 | null |
| 클래스 | null |
| 인터페이스 | null |

* 1. 메서드
     1. 객체의 행위, 객체 간의 협력을 위해 사용된다.
     2. 오버로딩 & 오버라이딩
        1. 오버로딩
           1. 메서드의 이름만 같고, 매개변수의 개수 타입 순서가 달라야 한다.
           2. 똑같은 이름에 다른 기능을 구현할 때 사용한다
           3. 접근 제어자 혹은 Return type만 다른 경우, 오버로딩이 불가능하다

int addData(int a)

char addData(ina a)

* + 1. 매개변수
       1. 기본형 & 참조형
          1. 기본형 매개변수

값 자체가 복사되어 넘어가기 때문에, 원본 값에 영향이 없다

* + - * 1. 참조형 매개변수

값의 주소 값이 넘어가기 때문에, 원본 값에 영향이 간다

* 1. 인스턴스 멤버 & 클래스 멤버
     1. 인스턴스 멤버
        1. 인스턴스 필드 + 인스턴스 메서드
        2. 객체를 생성해야만 사용할 수 있는 멤버
        3. 객체 **각각 별도의 인스턴스 필드**를 갖고 있다.
        4. 객체 각각 별도의 인스턴스 메서드를 갖고 있게 된다면, 메모리 효율이 떨어지기에 별도의 메서드 영역을 두고 인스턴스들이 공유한다.
     2. 클래스 멤버
        1. 클래스 필드 + 클래스 메서드
        2. 객체를 생성하지 않아도 사용할 수 있다.
        3. Class는 Java의 클래스 로더에 의해 메서드 영역에 저장되고 사용
        4. 클래스 멤버 = 메서드 영역의 클래스와 같은 위치에 고정적으로 위치하고 있는 멤버
        5. Static이라는 키워드를 사용한다.
           1. 인스턴스 모두 가지고 있어야만 하는 데이터를 저장하기에 용이
           2. 클래스 멤버로 선언된 메서드는 인스턴스 멤버 사용 X

Class 멤버는 객체가 생성되지 않아도 사용이 가능하지만,  
인스턴스 멤버는 객체가 생성돼야만 사용이 가능하기 때문

* + - * 1. 인스턴스 멤버로 선언된 메서드는 클래스 멤버 사용 O
  1. 생성자
     1. 객체가 생성될 때, 호출되며 객체를 초기화하는 역할
     2. 기본 생성자
        1. Car()와 같이 아무 매개변수가 없는 생성자를 의미
        2. 접근 제어자는 class의 접근 제어자를 따라간다.

Public class : -> public car()

* + - 1. **생성자가 없다면 컴파일러는 기본 생성자를 자동으로 추가**한다.
      2. 생성자가 **1개라도 존재한다면, 컴파일러는 기본 생성자를 추가하지 않는다**.
  1. this & this()
     1. this
        1. 인스턴스 자신을 표현하는 키워드
        2. 객체 내부의 생성자나 메서드에서 객체 내부 멤버에 접근하기 위해 사용된다.
           1. Car() {  
               this.price = 100000;  
              }
        3. 메서드에서 매개변수명과 객체의 필드명이 동일한 경우에 발생하는 오류를 방지할 수 있다.
        4. 객체의 메서드에서 리턴 타입이 자기 Class인 경우, return this 형태로 자기를 반환할 수 있다.

Instance 디자인 패턴에서 사용 가능

* + 1. this()
       1. 인스턴스 자신의 생성자를 호출하는 키워드
       2. 객체 내부 생성자 및 메서드에서 자신의 생성자를 호출하기 위해 사용
       3. 텍스트, 스크린샷, 폰트이(가) 표시된 사진

          자동 생성된 설명중복 코드를 줄여줄 수 있다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

* + - 1. this() 키워드를 사용해서 다른 생성자를 호출할 경우, 꼭 첫번째 줄에 작성해야 한다.
  1. 제어자
     1. 접근 제어자
        1. 접근 제어자 종류
           1. Public

Class, 메서드, 멤버 변수 모두 사용가능

접근 제한이 없다

* + - * 1. Default

Class, 메서드, 멤버 변수 모두 사용가능

같은 패키지 내에서만 접근이 가능하다.

* + - * 1. Protected

메서드, 멤버 변수에만 사용가능

같은 패키지 내의 자식 Class에서만 접근이 가능하다.

* + - * 1. Private

메서드, 멤버 변수에만 사용 가능

같은 클래스 내에서만 접근이 가능하다.

* + - 1. 접근 제어자를 사용하는 이유
         1. 외부에서 접근하지 않도록 제한하기 위해 사용
         2. 캡슐화를 통해 데이터를 보호하기 위해 사용

값을 함부로 바꾸지 못하도록 접근 제한

* + - * 1. 생성자의 접근 제어자를 통해, 인스턴스 생성을 제한할 수 있다.

Ex) Protected를 통해 자손 클래스에서만 호출 가능

* + - 1. Getter & Setter
         1. 객체의 무결성

객체 변경이 없는 상태를 유지하기 위해 접근 제어자 사용

외부에서 접근할 수 없는 private 필드를 어떻게 변경해야 할까?

* + - * 1. Private 필드를 읽거나 변경하기 위해 사용되는 메서드
        2. Getter

Private 필드를 읽어올 때 사용

getValocity와 같은 메서드 명을 사용

* + - * 1. Setter

Private 필드를 변경할 때 사용

setValocity와 같은 메서드 명을 사용

* + 1. 그 외 제어자
       1. Final
          1. Class, 메서드, 멤버변수, 지역변수 모두 사용 가능
          2. 상수를 선언할 때 사용한다.
       2. Abstract
          1. Class, 메서드에만 사용이 가능하다
          2. 가상 함수 / 가상 Class를 사용할 때 사용
       3. Static
          1. Class, 메서드, 멤버 변수 모두 사용 가능
          2. 클래스 멤버를 선언할 때 사용
    2. 제어자 주의 사항
       1. 메서드 Static, abstract를 함께 사용할 수 없다
       2. 클래스 Abstract와 final을 함께 사용할 수 없다.
       3. Abstract메서드의 접근 제어가가 private일 수 없다.
          1. 자식 Class에서 선언해야 하기 때문
       4. 메서드에 private와 final을 같이 사용할 필요는 없다
  1. package와 import
     1. package
        1. 클래스 일부분이면서 클래스를 식별해주는 용도
        2. .을 통해 상위 패키지와 하위 패키지를 구분
        3. 같은 클래스 명이라도 패키지 경로에 따라 구분한다.
           1. myPackage.p1.Car, myPackage.p2.Car는 서로 다른 Class이다
     2. Import
        1. 다른 패키지에 있는 Class를 사용하기 위해 명시하는 키워드
           1. Import myPackage.\* 를 사용하여 패키지 아래에 있는 모든 Class사용 할 수 있다.
  2. 상속과 포함
     1. 상속
        1. 부모 클래스의 필드와 메서드를 자식 클래스에게 물려줄 수 있다.
        2. 적은 양의 코드로 새로운 클래스를 작성할 수 있고, 공통적인 코드를 관리하여, 코드의 추가와 변경이 수월해진다.

코드 중복 제거 & 재사용성 증가 = 유지보수 & 생산성

* + - 1. Extends 키워드를 통해 정의
      2. 자식 Class는 부모 Class로부터 데이터를 물려받기에 자식 Class의 부피가 더 크거나 같다.
      3. 다중상속
         1. 한 자식 Class가 다수의 부모 Class를 갖는 상황
         2. Java에서는 Class간의 다중상속을 허용하지 않는다.

Class간의 관계가 복잡해 진다.

부모 Class가 같은 이름의 멤버를 사용한다면, 구분 방법이 없다.

* + - 1. Final Class
         1. Class에서 Final을 사용할 경우 최종적인 Class가 된다.
         2. 더 이상의 상속은 불가능하다.
      2. 모든 Class는 Object로부터 상속받는다.
         1. Object는 최상위 Class
      3. Super & Super()
         1. Super

객체 내부의 생성자 및 메서드에서 부모 Class의 멤버에 접근하기위해 사용

자식 클래스 내부 멤버와 부모 클래스에서 상속받은 멤버를 구분하기 위해 사용

* + - * 1. Super()

부모 클래스의 생성자를 호출할 수 있는 키워드

자식 클래스의 객체가 생성될 때, 부모 클래스들이 모두 합쳐져서 하나의 인스턴스가 생성

부모 Class의 생성자가 호출되어야만 한다.

명시하지 않더라도 자식 클래스 생성자에는 Super()가 숨겨져 있다

* + 1. 포함
       1. A Class에서 필드로 B Class를 사용할 수 있다.
          1. 컴퓨터 Class 내부에 그래픽카드 Class를 사용
  1. 오버라이딩
     1. 부모 클래스로부터 상속받은 메서드를 재정의 하는 것
     2. 부모 클래스보다 접근 제어자를 좁게 선언할 수 없다
        1. 부모 public -> 자식 protected X
     3. 예외는 부모 클래스 메서드보다 많이 선언할 수 없다.
  2. 다형성
     1. 여러가지 형태를 가질 수 있는 능력
        1. 구성하고 있는 객체를 바꿨을 때 소프트웨어의 실행 성능 및 결과물이 다르게 나올 수 있다.
     2. 참조변수의 타입 변환
        1. 자동 타입 변환
           1. 자식 객체는 부모 객체의 멤버를 상속받기 때문에, 부모와 동일하게 취급될 수 있다.
           2. 부모타입 변수 = 자식 타입 객체는 자동으로 부모타입으로 변환
        2. 강제 타입 변환
           1. 부모 => 자식 타입은 자동으로 변환되지 않는다.
           2. 부모 타입 변수로는 사용할 수 없는, 자식 Class의 고유한 멤버를 사용하기 위해 사용
           3. 자식 타입 변수 = (자식 타입) 부모 타입 객체형태로 명시
           4. 강제 타입 변환이 불가능한 상황

자식 Class -> 부모 Class 자동 형 변환된 경우에만 사용이 가능하다.

부모 Class로 생성된 경우는 사용 불가능

* + - 1. 추가 정보
         1. 매개변수, 반환 타입에도 다형성이 적용될 수 있다
  1. 인터페이스
     1. 스펙이 정의된 메서들의 집합
     2. 상속관계가 없는 다른 클래스 들이 서로 동일한 메서드를 구현해야 할 때,   
        인터페이스를 통해 동일한 사용 방법과 행위를 보장해 줄 수 있다.

인터페이스에 다형성을 적용할 수 있게 만들어준다.

* + 1. 인터페이스 구현 클래스 들은 반드시 정의된 메서드를 구현해야만 한다.
    2. 선언
       1. Interface 키워드를 통해 인터페이스 선언
    3. 구성
       1. 모든 멤버 변수는 public static final 형태( 생략가능 )
       2. 모든 메서드는 public abstract 형태 ( 생략가능 )
       3. 생략한 제어자는 컴파일러가 자동으로 추가해준다.
    4. 구현
       1. 추상 메서드는 구현될 때, 반드시 오버라이딩 되어야 한다.
       2. 인터페이스의 추상 메서드 중 일부만 구현해야 한다면, 추상클래스로 변경해 주어야 한다.