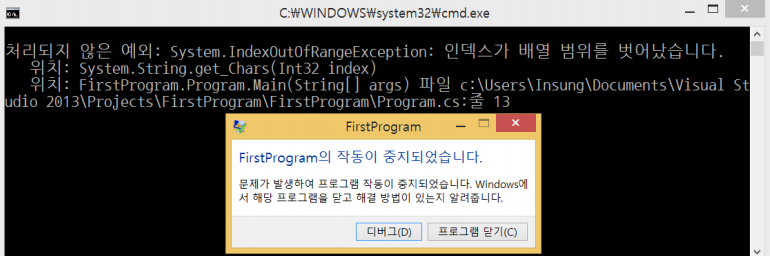
1장

* 플랫폼과 소프트웨어 플랫폼
* 플랫폼 : 소프트웨어 응용 프로그램의 실행에 사용되는 하드웨어와 소프트웨어 집합
* 소프트웨어 플랫폼 : 중간 레고 블록 역할을 함
* 라이브러리
* 코드를 쉽게 사용할 수 있게 미리 만들어준 코드
* 프로그램 소프트웨어를 만들 때 사용하는 클래스 또는 서브루틴의 집합
* 개발자가 사용해줘야만 하고 스스로는 아무것도 하지 못함
* 프레임워크
* 제어 역전이 있는 대규모의 라이브러리
* 제어역전:원래 개발자가 제어하던 코드를 프레임워크가 제어하는 것
* 프로그램의 초기화부터 종료까지의 흐름을 직접 관리
* 기본 틀을 모두 제공, 개발자는 개발 집중 가능
* 자동 완성 기능(인텔리센스) 과 보조기능
* Ctrl + Space 누르면 됨

2장

* 표현식
* 값을 만들어 내는 간단한 코드
* 식별자
* 괄호가 있는 식별자는 메서드, 이외의 것은 변수, 메서드 괄호 안에 넣는 것은 매개 변수
* 나머지 연산자와 부호
* 나머지 연산자의 부호는 왼쪽 피연산자의 부호를 따름
* 예외
* 런타임 에러, 예외 코드 실행 중 발생하는 오류



* 예외 발생(릴리즈 모드)
* 문자 덧셈 연산
* 문자열은 + 연산자로 연결 가능, 문자는 불가능
* Int 자료형
* 크기 : 4바이트(32비트), 범위: 2^32개의 숫자(-2,147,483,648 ~ 2,147,483,647)을 나타냄
* 오버플로우 : 범위를 넘는 현상
* Unsigned 자료형 (부호가 없는 자료형)
* 음수 사용을 위한 자료형
* Uint 와 ulong 키워드 사용
* MaxValue 와 MinValue
* Int.MaxValue //최댓값
* Int.MinValue //최솟값
* L
* 폰트에 따라 1과 소문자 l 혼동, 코드 작성 시 long 자료형 나타내는 대문자 L사용
* Float : 4바이트 double : 8바이트
* 문자열 말고 문자는 연산 가능하다.
* Sizeof 연산자와 string 자료형
* String 자료형은 sizeof 연산자로 자료형의 크기 구할 수 없음
* 이거는 class로 시작하기 때문에!!
* 자료형 검사 방법
* 1. 마우스 갖다대기
* 2. GetType()메서드 : 변수 뿐만 아니라 숫자 또는 문자열에 직접 적용 가능
* Ex) int.GetType()
* Var 키워드 사용
* 자료형을 자동으로 지정한다.
* 한 번 지정된 자료형은 계속 유지
* 지역 변수로 선언
* 지역 변수 : 메서드 내부에 선언되어 있는 변수
* 인스턴스 변수 즉, 함수 밖에 있는 변수는 var 사용 안됨
* 변수를 선언과 동시에 초기화!!
* Var 로 long 자료형, float 자료형 선언시 ,숫자 뒤에 L,F 붙여야한다.
* 자료형 변환
* 형변환 할 때 무조건 자료가 손상 되는 것은 아님
* Int.Parse() 다른 자료형을 int 로 바꾼다. (나머지도 마찬가지)
* 다른 자료형을 문자열로 반환
* C#의 모든 자료형은 ToString() 메서드를 가지고 있음
* 문자열로 변환해줌!
* 간단한 문자열 변환
* Char 형을 문자형으로 바꾸려면 char character 가 있을 때 character + “”을 해서 string에 저장해주면 됨!



* 음수밖에 없는 숫자 –output 으로 하면 음수로 나온다.
* 현재 시간 가져오는 메소드
* **DateTime.Now.Hour** : 현재 시간에서 시 를 뽑아내는 메소드
* String.Contains(string value) 함수 : 인자로 있는 값이 포함되어있는지를 판단한다.
* Console.ReadKey() 함수 : 사용자가 누른 기능 키를 가져옵니다. 누른 키는 선택적으로 콘솔 창에 표시된다.
* ConsoleKeyInfo info = Console.ReadKey();
* Info.KeyChar() 은 현재 입력한 키가 나타내는 유니코드 문자를 가져온다.
* Info.Key() 개체가 나타내는 콘솔키를 가져온다.
* ConsoleKey.A A키를 나타낸다.

3장

* 기본적인 배열 생성 방법
* 자료형[] 이름 = {자료,자료}
* 배열의 요소 : 배열[인덱스]
* Length 속성
* intArray.Length 이런식으로 길이를 나타낸다.
* IndexOutOfRangeException
* 잘못된 인덱스에 접근!
* 원하는 크기의 배열 생성
* Int[] array = new int[100];
* Do while 문
* Do{ }while(조건);
* 한글 다 출력
* 가부터 힣 까지 for 문 돌리면 됨 ex) i=’가’ …..
* Foreach 반복문
* 컬렉션에 쉽게 적용할 때 사용
* 컬렉션 : 여러 개체가 모여서 집합을 이룬 것
* Foreach( 자료형 변수 in 컬렉션) { }
* 코드 조각
* 자주 사용되는 코드 정의해서 사용하기 쉽게 만든 것
* For 를 입력후 tab 키 두번 누름
* 원하는 반복문 벗어나기
* 외부 반복문 => goto doNotUse;
* doNotUse: Console.WriteLine(“goto 키워드”);
* 문자열 처리
* 대문자화와 소문자화
* ToUpper() : 문자열의 모든 문자를 대문자로 변경
* ToLower() : 문자열의 모든 문자를 소문자로 변경
* C#의 모든 문자열 처리 메서드는 자신을 변경하지 않고 그냥 그 내용 자체를 반환
* 비파괴적 메서드 : 자신을 변경하지 않고 반환하는 메서드
* 문자열 자르기
* Split() : 문자열을 특정한 문자 또는 문자열로 자른다.
* 문자열 양 옆의 공백 제거
* Trim() : 문자열 양 옆의 공백을 제거한다.
* TrimStart() : 문자열 앞의 공백을 제거한다.
* TrimEnd() : 문자열 뒤의 공백을 제거한다.
* 배열을 문자열로 변환
* String.Join() : 배열의 요소를 뭉쳐 문자열로 변경한다.
* 이동하는 달팽이
* Console.Clear() : 콘솔 화면을 지운다.
* Console.SetCursorPosition() : 콘솔 화면의 특정한 위치로 커서를 옮긴다.
* Thread.Sleep() : 특정한 시간만큼 스레드를 정지한다.
* Switch 조건문과 무한 반복문
* ConsoleKey.UpArrow : 방향키 위 버튼 / RightArrow : 오른쪽 / DownArrow : 아래쪽

LeftArrow : 왼쪽

5장

* 사용자 정의 자료형
* 클래스 : 객체 지향 언어, 원하는 새로운 자료형 정의
* 클래스와 인스턴스
* Car car = new Car() 순서대로 클래스, 인스턴스, new키워드, 생성자
* 인스턴스 : 클래스 자료형을 변수로 선언한 것
* Random 클래스
* Random random = new Random() 이런식으로 선언하면 됨
* 원하는 범위의 난수 생성
* 정수 난수 : random.Next(10,100)
* 실수 난수 : random.NextDouble()\*10; 10까지의 실수 난수 발생
* List 클래스
* 배열과 유사
* 제네릭 : 클래스 선언 시 어떤 자료형인지 알려주는 것
* List<int> list = new List<int>();
* List 인스턴스 생성과 동시에 요소 추가
* List<int> list = new List<int>() {52,273,32,64};
* Math 클래스
* 수학과 관련된 변수 또는 메서드 제공
* 인스턴스를 만들지 않고 사용
* Math.Abs() : 절대값 / Ceiling(숫자) 지정된 숫자보다 크거나 같은 최소 정수 / Floor(숫자) 지정된 숫자보다 작거나 같은 최대 정수
* 클래스 내부에도 클래스 생성 가능하고 한 파일에 여러 개의 클래스 생성 가능
* 클래스에서의 인스턴스 변수
* 접근제한자 자료형 이름 ex) public string name; public DateTime regDate;
* 초기화 : public string name = “default”; 이런식으로 초기화 시켜줌
* Ex) Product product = new Product() { name = “감자”, price = 2000 }; name 과 price 는 public 으로 선언된 변수임
* 클래스 변수
* 클래스 변수 : 접근제한자 static 자료형 이름
* 추상화
* 클래스 기반의 객체 지향 프로그래밍 언어의 특징
* 추상화, 캡슐화, 상속, 다형성
* 추상화 : 프로그램에 사용되는 핵심적인 부분을 추출하는 것
* List<Student>list = new List<Student>();
* List.Add(new Student() {name = “윤인성”, grade = 1}); 이런식으로 객체 만들어서 추가 시킬 수 있고 add 는 리스트에 대한 함수이다.
* List 클래스 요소 제거
* Remove( object element) : 특정 요소를 리스트에서 제거한다. (객체 지정)
* RemoveAt( int index ) : 특정 위치에 있는 요소를 리스트에서 제거한다.(인덱스 지정)

6장

* 메서드의 기본형태
* 접근제한자 반환형 메서드이름 매개변수{ 메서드 코드 }
* 메인 메서드는 클래스 메서드임
* 클래스 메서드 생성 방법
* 접근제한자 static 반환형 메서드이름 (매개변수){ }
* 클래스 메서드에서 사용할 수 있는 것
* 클래스 메서드에서는 메모리에 올라가지 않은 인스턴스 변수, 인스턴스 메서드 사용 못함
* 오버로딩
* 이름은 같고 매개변수는 다른 메서드를 만드는 것
* 매개변수의 타입이 다른 메서드를 만들어야 됨
* 똑같은 자료형으로 여러 개 이름이 같은 함수를 만들지 못함
* 접근 제한자의 형태
* 대표적인 접근 제한자 : public , private
* Private 접근 제한자
* 접근 제한자 입력하지 않으면 자동으로 private 접근 제한자로 설정된다.
* Private 적용 되면 자신의 클래스 내부에서만 해당 메서드 사용 가능하다.
* Public 접근 제한자
* Public 접근 제한자가 걸린 변수 또는 메서드는 모든 곳에서 접근 가능하다.
* 생성자
* 무언가를 생성할 때 자동으로 호출되는 메서드
* 이름이 클래스 이름과 같아야한다.
* 접근 제한자는 public
* 반환과 관련된 선언은 하면 안됨
* Public 클래스이름 (매개변수) { }
* 생성자의 인스턴스 변수 초기화
* This.price 라고 한다면 여기에서 말하는 this 는 해당 클래스가 된다
* Private 생성자
* 생성자로 클래스의 인스턴스를 만들 수 없게 할 때는 private 생성자 사용
* 정적 멤버만 가지고 있을 때
* 소멸자 (destructor)
* 인스턴스가 소멸될 때에 호출
* 변수의 불사용이 확실할 때 객체 소멸시키며 소멸자 호출 (객체 소멸시기 불명확)
* 이름은 클래스 이름 앞에 ~ 기호를 붙인거
* 접근 제한자 사용 않음
* 반환과 관련된 선언 없음
* 매개변수와 관련된 선언 하지 않음
* 하나의 클래스에는 하나의 소멸자만 가짐
* ~ 클래스이름(){ }
* 상수
* 일반적인 변수는 값 계속 변경 가능, 상수로 선언된 변수는 값 변경 불가능
* Math.PI 이렇게 상수로 지정된 값은 못 바꿈
* 상수 만들기
* Public const double PI = 3.141592; 이런식으로 상수 만들 수 있음
* 메서드 내부에서 상수 사용 가능
* Ex) cons tint value = 10; 이런식으로 함수 내부에 써주면 됨
* Readonly 키워드
* 읽기 전용 변수
* 변수 선언 시점과 생성자 메서드에서만 값 변경 가능
* 캡슐화
* 변수 width 와 height 에 음수가 들어갈 수 있는 위험성 원천 봉쇄
* class Box
* {
* private int width;
* private int height;
* public Box(int width,int height)
* {
* if (width > 0 || height > 0)
* {
* this.width = width;
* this.height = height;
* }
* }
* public int Area()
* {
* return this.width \* this.height;
* }
* }
* 겟터와 셋터
* 변수를 바로 수정할 수는 없지만 변수 변경 메서드를 만들고, 메서드를 호출해 변경
* Public int GetWidth(){ return width; }
* Public void SetWidth( int width ){ if ( width > 0 ){ this.width = width; }
* 일반적인 속성 생성 방법
* 속성 : 겟터와 셋터를 쉽게 만드는 법
* 속성의 일반적인 형태
* Private int 변수이름;
* Public int 속성이름{ get { return 변수이름; } set { 변수이름 = value; } }
* 속성의 사용 방법
* 인스턴스이름.속성이름 //겟터 호출
* 인스턴스이름.속성이름 = 값 //셋터 호출
* 간단한 속성 생성 방법
* Public int Width { get; set; }
* 속성 코드 조각
* Ctrl + space 또는 prop 입력 prop 잡고 tab 두번 누르면 간단한 겟터 셋터 생김
* Propfull 을 잡고 tab 키 두 번 누르면 전체 속성을 자동으로 작성해줌
* 값 복사와 참조 복사
* 값 또는 클래스로 만들어진 인스턴스는 참조
* 매서드의 매개변수로 값과 참조 전달 시 큰 차이 발생
* 값 복사
* 값을 복사하면 그 복사된 값을 저장한 변수는 그대로 이고 값을 변경한 변수만 바뀌게 된다.
* A = 10 input = 10 // a = 10 input = 20
* 참조 복사
* 참조 복사를 하게 되면 그 값을 복사해온 변수의 값이 변경되면 복사 당한 변수도 값이 변경되게 된다.
* 재귀 메서드
* 자기 자신을 호출 하는 거고 탈출 종료 조건 필요
* 메모화
* 한번 계산했던 값 저장해두는 것
* Private static Dictionary<int, long> memo = new Dictionary<int, long>();
* 이미 계산했던 값인지 확인한다. ->memo.ContainsKey(i)
* 값을 넣어줄 때도 memo[i] = value 이런식으로 넣어준다.
* 값에 의한 전달
* 값복사란 두개의 메모리가 존재하고 한쪽의 메모리에 들어 있는 값을 다른 쪽의 메모리로 그 값만큼 복사하는 행위 즉 값만 복사
* Ref에 의한 참조
* 참조형 매개변수는 메모리의 위치에 대한 참조이다. 위에서 함수를 호출하면 참조형 매개변수는 함수가 호출될 때 넘겨지는 변수의 메모리 주소에 대한 참조값을 가지게 된다.
* Out에 의한 참조
* Ref 키워드를 이용한 참조에는 반드시 참조할 변수가 초기화가 되어있어야 한다. 하지만 out 을 쓰면 굳이 호출하는 쪽의 인수를 초기화 할 필요가 없다.
* 깊은 복사와 얕은 복사
* 해당하는 변수의 전체 필드 값을 복사하는 것이 아니라 참조하고 있는 힙의 주소만 가지고 복사해 온 것을 얕은 복사라고 한다.
* 깊은 복사는 원본 객체와 별도의 객체를 할당하여 각 필드의 값을 복사해 넣는 것이다.
* 새로운 객체를 만들고 거기에 해당하는 필드들을 복사해서 넣어준다음에 return 해주는 함수를 사용하면 이거는 깊은 복사가 된다.
* Dictionary 클래스
* 키와 값으로 이루어진 해시테이블 형태의 제네릭 컬렉션
* 사전을 이용하면 내가 알고 있는 의미있는 키로 값을 찾을 수 있다.
* Dictionary <string, string> hoons = new Dictionary <string, string>();
* 이런식으로 선언 하고
* 사전에 추가 : hoons.Add(“HS”,”HOONS”);
* 사전에서 제거 : hoons.Remove(“JS”);
* Try { } catch (Exception ex) { }
* Hoons.ContainKey (“DJ”) DJ가 포함되어 있는지
* Hoons.TryGetValues(“JS”,out js) JS 라는 게 들어있으면 js 로 나타낸다는거 같음!!
* using System;
* using System.Collections.Generic;
* namespace Jinsoo.Examples
* {
* class DictionaryTest
* {
* static void Main(string[] args)
* {
* Dictionary<string, string> hoons = new Dictionary<string, string>();
* Console.WriteLine("-------------------------------------------");
* // 사전에 추가
* hoons.Add("HS", "HOONS");
* hoons.Add("JS", "JINSU");
* hoons.Add("YS", "YOUNGSU");
* hoons.Add("SY", "SUYOUNG");
* Console.WriteLine(hoons["JS"]);
* // 키가 존재하면 값이 대체 됩니다.
* hoons["JS"] = "JISUN";
* Console.WriteLine(hoons["JS"]);
* Console.WriteLine("-------------------------------------------");
* // 사전에서 제거합니다.
* hoons.Remove("JS");
* // 지정한 키가 없다면 예외가 발생합니다.
* try
* {
* Console.WriteLine(hoons["JS"]);
* }
* catch (Exception ex)
* {
* Console.WriteLine(ex.Message);
* }
* // 키의 존재 우뮤 판단으로 안전하게 사전에 추가할 수 있습니다.
* if (!hoons.ContainsKey("DJ"))
* {
* hoons.Add("DJ", "DONGJIN");
* }
* else
* {
* hoons["DJ"] = "DONGJIN";
* }
* // 설정/검색이 가능한 키/값 쌍을 나열합니다.
* foreach (var h in hoons)
* {
* Console.WriteLine("Key = {0}, Value = {1}", h.Key, h.Value);
* }
* Console.WriteLine("-------------------------------------------");
* // 사전 키를 나열합니다.
* foreach (var hKey in hoons.Keys)
* {
* Console.WriteLine(hKey);
* }
* Console.WriteLine("-------------------------------------------");
* // 사전 값을 나열합니다.
* foreach (var hValue in hoons.Values)
* {
* Console.WriteLine(hValue);
* }
* Console.WriteLine("-------------------------------------------");
* }
* }
* }
* 상속과 다형성 : C#에서 반복을 줄이기 위해 사용
* 리스트에 객체 넣고 초기화
* List<Dog> Dogs = new List<Dog>() {new Dog(),new Dog(),new Dog()};
* 상속
* 클래스 사이에 부모 자식 관계를 정의하는 작업
* 부모 클래스 하나는 자식을 여러 개 가질 수 있음
* Class Dog : Animal => Animal 이라는 클래스를 상속 받음
* 클래스의 부모 자식관계가 형성되면 자식 클래스는 부모 클래스의 public 또는 protected 멤버에 접근 가능
* Base 키워드
* 자식 클래스에서 부모 클래스에서 정의한 멤버 사용 가능
* 이름이 겹치는 등 특수한 이유로 부모의 메서드에 접근 불가능 할 경우 this 키워드와 같은 형태로 base 키워드 사용 (this : 자신을 나타내는 키워드, base : 부모 나타내는 키워드)
* Protected 접근 제한자
* Private 와 비슷하지만 상속한 클래스(파생 클래스)에서는 접근 가능
* 다형성
* 하나의 클래스가 여러 형태로 변환될 수 있는 성질
* 자식 클래스가 부모 클래스로 위장
* Animal dog = new Dog();
* 변수 dog는 외관상으로 자료형 Animal 이지만 실제 내부에는 Dog가 들어있다.
* 외관상으로는 Animal 객체이므로 사용할 수 있는 멤버는 Animal 클래스의 멤버뿐임
* 자식 클래스에 있는 메서드 사용 위해 자식 클래스로 자료형 변환
* ((Dog)item) , ((Cat)item)
* 최상위 객체
* C#에서 만드는 모든 객체는 Object 라는 객체의 상속을 받음
* List<Object> listObject = new List<Object>();
* listObject.Add(new Dog());
* listObject.Add(52);
* listObject.Add(new Cat());
* is 키워드
* 특정 객체의 클래스 확인
* 변수 is 클래스
* Item is Cat // Cat 이라는 클래스이냐 이거를 보는거
* Item is Animal , item is Object
* 일반적인 자료형 변환
* (클래스) 변수 ex) ((Dog)item).Bark();
* As 키워드로 자료형 변환
* 변수 as 클래스
* 상속의 생성자
* 생성자 : 인스턴스 초기화할 때 사용
* 자식 인스턴스 생성하면, 부모의 멤버 초기화 위해 부모 생성자도 자동으로 호출된다.
* 부모 생성자 호출을 명시적으로 지정할 때
* 매개변수가 있는 메서드를 호출하고 싶을 때 base (10) base (input) 이런식으로 부모의 매개변수에 값을 넣어 줄 수 있다.
* 클래스 변수 상속
* 클래스 변수는 상속되어도 공유된다.
* 섀도잉과 하이딩
* 섀도잉 : 특정한 영역에서 이름이 겹쳐 다른 변수 가리는 것

public static int number = 10;

static void Main(string[] args)

{

int number = 20;

Console.WriteLine(number);

* }
* 하이딩 : 부모 클래스와 자식 클래스에 동일 이름으로 멤버 만들 때 발생

class Parent

{

public int variable = 273;

}

class Child :Parent

{

public string variable = "Shadowing";

* }
* 부모에 있는 int 자료형의 변수 사용할 때
* 부모로 자료형 변환 하고 사용
* 메서드는 변수와 다르게 충돌이 발생할 때 하이딩 할 지 오버라이딩 할 지 결정 가능
* Public new void Method(){ Console.WriteLine(“자식의 메서드”);
* Public override void Method(){ Console.WriteLine(“자식의 메서드”);
* 오버라이딩 : 부모 클래스의 메서드 자식 클래스에서 재구현
* 하이딩의 형태로 메서드 작성 후 앞에 virtual 이라는 키워드 붙임
* 하이딩은 멤버 전체 에서 발생
* 오버라이딩은 메서드 관련해서만 발생
* New 메서드
* 하이딩한다는 표시를 위해 메서드 이름 앞에 new 키워드 붙임

class Program

{

class Parent

{

public int variable = 273;

public void Method()

{

Console.WriteLine("부모의 메서드");

}

}

class Child : Parent

{

public new string variable = "hiding";

public new void Method()

{

Console.WriteLine("자식의 메서드");

}

}

static void Main(string[] args)

{

Child child = new Child();

child.Method();

((Parent)child).Method();

}

* }
* 오버라이딩 하면 클래스형을 어떻게 변환해도 자식에서 다시 정의한 메서드가 호출된다.
* 부모에는 virtual 이런식으로 쓰고 그 override 하려는 함수를 public override void Method() 이런식으로 써준다.
* Sealed 메서드 : 클래스 적용(상속 제한),메서드 적용(오버라이딩 제한)
* Sealed class parent 이런식으로 선언된 클래스가 있으면 그걸 상속 받을 수 없다.
* 메서드에 sealed 를 붙이면 그 메서드를 오버라이딩 할 수 없다.
* Sealed public override void Test(){ } 이러면 이 메서드를 다른 곳에서 오버라이딩 하거나 그런거를 할 수 없다.
* Abstract 키워드 : 무조건 상속 또는 메서드 반드시 오버리이딩
* Abstact class Parent 라고 선언된 클래스는 객체를 생성하지 못하고 무조건 상속 되어야지 그 안에 있는 메서드나 변수를 사용할 수 있다.
* 메서드 오버라이딩
* Abstract class Parent{ public abstract void Test();} abstract 를 쓰려면 클래스에 붙여줘야 메서드에서도 abstract 를 붙여서 사용이 가능하다.
* Abstract 메서드에서 오버라이딩을 하지 않으면 오류가 생긴다
* 해결하려면 abstract class Parent{ public abstract void Test();} class Child : Parent{ public override void Test(){ }
* Enum 자료형(열거자) : 숫자에 특정한 이름을 붙여주는 방법
* 자료형 비교를 통해 숫자를 조금 더 안전하게 입력
* 열거자는 클래스, 인터페이스 등과 같은 위치에 생성
* 메서드의 매개변수 또는 반환형에 orderState 이런 것들이 자주 활용 된다