

자바 프로그램의 기본 구조

Hello : 클래스 이름

클래스
영역

```
public class Hello {  
    main() : 메소드  
    {  
        public static void main(String[] args) {  
            System.out.println("Hello, welcome to the java world!");  
        }  
    }  
}
```

main()
메소드
영역

파일명 : Hello.java

```
3 public class Hello {
4
5     public static void main(String[] args) {
6         System.out.println("Hello");
7     }
8
9 }
```

main() 메소드
매개변수 확인

오류 발생 시
어떤 오류인지 확인

```
3 public class Hello {
4
5     public static void main(String ... args) {
6         System.out.println("Hello");
7     }
8
9 }
```

```
3 public class Hello {
4
5     public static void main() {
6         System.out.println("Hello");
7     }
8
9 }
```



변수

- 변수 선언 / 사용법
- 변수 이름을 위한 명명 규칙
 - 변수명으로 사용할 수 없는 경우
- 다른 식별자(Identifier) 명명 규칙도 확인
 - 클래스 / 메소드

연산자

■ 증감 연산자 결과

```
int x = 0;
while(++x < 10){
    System.out.println(x);
}
```

```
int x = 0;
while(x++ < 10){
    System.out.println(x);
}
```

배열

■ 배열 선언 / 초기화 / 크기

```
int[ ] a = new int[5]; //초기값 : 0
boolean[ ] b = new boolean[5]; //초기값 : false
String[ ] s = new String[5]; //초기값 : null
String[ ] x = new String[0];
```

```
3 public class Test2 {
4
5     public static void main(String[] args) {
6         char[] x = {'J', 'a', 'v', 'a'};
7
8         System.out.print("출력 : ");
9         for(int i=0; i < x.length ; i++){
10             System.out.print(x[i]);
11         }
12
13     }
14
15 }
```



클래스

- 클래스 구성 요소

- 필드 / 메소드 / 생성자

- 클래스 상속 관계 : extends

- 자바는 다중 상속 안됨

- 인터페이스 구현 : implements

- 다중 인터페이스 구현 가능

- 여러 클래스와 인터페이스로 상속과 인터페이스 구현 코드 제시 – 잘못된 상속 관계와 구현 방법 구별



생성자

- 생성자 역할
- 생성자 오버라이딩
- 상속에서의 생성자
- `super` / `super()` 사용법
- 생성자 접근 제한자
- 코드에서 생성자 확인하고 잘못된 것 찾을 수 있어야 함



static

- static 개념

- 정적(static) 멤버

- 클래스에 고정된 필드와 메소드
- 클래스에 소속된 멤버 (클래스 당 하나만 생성)
 - 객체를 생성하지 않고 클래스로 바로 접근해 사용
- 클래스 내의 모든 객체들이 공유
- 프로그램이 시작될 때 이미 생성
 - 객체 보다 먼저 생성됨
- 프로그램이 종료될 때 사라짐

싱글톤 (Singleton)

- 싱글톤(Singleton) 클래스

- 하나의 애플리케이션 내에서 단 하나의 객체만 생성되는 클래스

- 싱글톤을 만드는 방법

- 외부에서 new 연산자로 생성자를 호출할 수 없도록 **private** 접근 제한자를 생성자 앞에 붙임
- 클래스 자신의 타입으로 정적(static) 필드 선언
 - 자신의 객체를 생성해 초기화
 - **private** 접근 제한자를 붙여 외부에서 필드 값 변경 불가하도록
- 외부에서 호출할 수 있는 정적(**static**) 메소드인 **getInstance()** 선언
 - 정적 필드에서 참조하고 있는 자신의 객체 리턴

```
1
2 public class Singleton {
3     // 클래스 자신의 타입으로 정적(static) 필드 선언
4     private static Singleton singleton = new Singleton();
5
6     private Singleton() {} // 외부에서 new 연산자로 생성자를 호출할 수 없도록 private
7
8     static Singleton getInstance() { // 외부에서 호출할 수 있는 정적(static) 메소드 사용
9         return singleton;
10    }
11 }
```



상속

- 강제 타입 변환(Casting)

- 부모 타입을 자식 타입으로 변환하는 것
- 모든 부모 타입을 자식 클래스 타입으로 강제 변환할 수 있는 것은 아니고,
- 조건
 - 자식 타입을 부모 타입으로 자동 변환 후, 다시 자식 타입으로 변환할 때

접근 제한자

접근 범위	멤버의 접근 제한자			
	public	private	protected	default
같은 패키지의 한 클래스 내에서	O	O	O	O
같은 패키지의 다른 클래스에서	O	X	O	O
다른 패키지의 클래스에서	O	X	X (상속 O)	X

접근 제한이 강한(점점 높아 지는) 순서
public -> protected -> default -> private

메소드 오버로딩

	오버로딩 (Overloading)	오버라이딩 (Overriding)
정의	동일한 이름의 메소드 중복	서브 클래스에서 슈퍼 클래스의 메소드를 동일한 이름으로 재정의
관계	동일 클래스 내 또는 상속 관계에서 발생	상속 관계에서 발생
목적	사용의 편리성	슈퍼 클래스에서 구현된 기능을 무시하고 서브 클래스에서 새로운 기능으로 재작성
조건	메소드 이름만 동일 매개변수의 개수나 자료형 다르게 리턴 타입 상관 없음	메소드의 이름, 반환형, 매개변수 리스트 모두 동일
바인딩	정적 바인딩 (컴파일 시에 중복된 메소드 중 호출되는 메소드 결정)	동적 바인딩 (실행 시간에 오버라이딩된 메소드 찾아서 호출)

중첩

- 로컬 클래스에서 매개변수와 로컬 변수는 **final** 인 경우만 사용 가능

```
public class Outer {  
  
    //자바8 이후  
    public void method2(final int arg1, int arg2) {  
        final int localVariable1 = 1;  
  
        class Inner {  
            public void method() {  
                int result = arg1 + localVariable1 + arg2;  
            }  
        }  
    }  
}
```

로컬 클래스 내부로 복사되어
final 특성을 가짐
자바8부터는 final 키워드 없어도
자동으로 final 특성 (오류 없음)



예외 처리

- **NullPointerException**

- 객체 참조가 없는 상태

- **ArrayIndexOutOfBoundsException**

- 배열에서 인덱스 범위 초과하여 사용할 경우 발생

- **ArithmeticException**

- 0으로 나눌때

스레드

■ 작업 스레드 생성 방법

① Thread 클래스로부터 직접 생성

- class Task implements Runnable { }
- Runnable을 매개값으로 갖는 생성자 호출
- Thread thread = new Thread(Runnable target);

② Thread 하위 클래스로부터 생성

- Thread 클래스 상속 후 run 메소드 재정의해 스레드가 실행할 코드 작성
- public class WorkerThread extends Thread

Runnable 구현 객체 대입 방법2


익명 구현 객체 사용

```
6 public static void main(String[] args) {
7     //Runnable 구현 객체 대입 방법1
8     /*Runnable beepTask = new BeepTask();
9     Thread thread = new Thread(beepTask); */
10
11     //Runnable 구현 객체 대입 방법2
12     //익명 구현 객체 사용 (BeepTask 사용 안 함)
13     Thread thread = new Thread(new Runnable() {
14         @Override
15         public void run() {
16             Toolkit toolkit = Toolkit.getDefaultToolkit();
17             for (int i = 0; i < 5; i++) {
18                 toolkit.beep();
19                 try {
20                     Thread.sleep(500);
21                 } catch (Exception e) {
22                 }
23             }
24         }
25     });
26
27     thread.start();
```

Runnable 구현 객체 대입 방법3

람다식 사용

```
19         try {
20             Thread.sleep(500);
21         } catch (Exception e) {
22         }
23     }
24 }
25 });*/
26
27 // Runnable 구현 객체 대입 방법3
28 // 람다식 사용
29 Thread thread = new Thread(() -> {
30     Toolkit toolkit = Toolkit.getDefaultToolkit();
31     for (int i = 0; i < 5; i++) {
32         toolkit.beep();
33         try {
34             Thread.sleep(500);
35         } catch (Exception e) {
36         }
37     }
38 });
39
40 thread.start();
```



```
2
3 @FunctionalInterface
4 public interface MyFunctionalInterface {
5     public void method();
6 }
```

@FunctionalInterface 어노테이션

함수적 인터페이스(추상 메소드 1개)임을 표시하는 어노테이션

스레드 동기화

- 동기화 메소드 및 동기화 블록 - **synchronized**
 - 단 하나의 스레드만 실행할 수 있는 메소드 또는 블록
 - 다른 스레드는 메소드나 블록이 실행이 끝날 때까지 대기
 - 동기화 메소드 : **synchronized** 키워드 붙임
 - (잠금(lock) 의미)
 - 코드 확인
 - 동기화 메소드
 - 동기화 블록

람다식

■ 람다식 기본 사용법

```
MyFunctionalInterface fi = () -> {  
    String str = "method call1";  
    System.out.println(str);  
};
```

■ 스레드 생성 시 람다식 사용

```
Thread thread = new Thread(() -> {  
    Toolkit toolkit = Toolkit.getDefaultToolkit();  
    for (int i = 0; i < 5; i++) {  
        toolkit.beep();  
        try {  
            Thread.sleep(500);  
        } catch (Exception e) {  
        }  
    }  
});
```



제네릭

■ 제네릭

- 클래스(인터페이스)나 메소드를 타입 파라미터를 이용하여 선언하는 기법
- 제네릭을 사용하는 코드의 이점
 - 컴파일 시 강한 타입 체크 가능
 - 컴파일 시에 미리 타입을 강하게 체크해서 에러를 사전 방지 (안전)
 - 강제 타입 변환 제거 가능 (프로그램 성능 향상)

제네릭 (계속)

■ 와일드카드 타입

- 제네릭 타입을 매개값이나 리턴 타입으로 사용할 때 구체적인 타입 대신에 와일드카드(?) 형태로 사용
- 타입 파라미터를 제한할 목적

■ 와일드카드 타입의 세가지 형태

- 제네릭타입 <?>
- 제네릭타입 <? extends 상위타입>
- 제네릭타입 <? super 하위타입>

제네릭 (계속)

■ 제네릭타입 <?>

- Unbounded Wildcards (제한없음)
- 타입 파라미터를 대치하는 구체적인 타입으로 모든 클래스나 인터페이스 타입이 올 수 있음

■ 제네릭타입 <? extends 상위타입>

- Upper Bounded Wildcards (상위 클래스 제한)
- 타입 파라미터를 대치하는 구체적인 타입으로 상위 타입이나 하위 타입만 올 수 있음



제네릭 (계속)

■ 제네릭타입 <? super 하위타입>

- Lower Bounded Wildcards (하위 클래스 제한)
- 타입 파라미터를 대치하는 구체적인 타입으로 하위 타입이나 상위 타입만 올 수 있음



WildcardEx 예제 이해할 것

```
public class WildCardExample {  
    // 제한 없는 경우  
    // 수강생은 모든 타입 가능 : Person, Worker, Student, HighStudent  
    public static void registerCourse(Course<?> course) {  
        System.out.println(course.getName() + " 수강생: " +  
            Arrays.toString(course.getStudents()));  
    }  
  
    // 상위 클래스 제한 경우  
    // 수강생은 Student, HighStudent만 가능  
    public static void registerCourseStudent(Course<? extends Student> course) {  
        System.out.println(course.getName() + " 수강생: " +  
            Arrays.toString(course.getStudents()) );  
    }  
  
    // 하위 클래스 제한 경우  
    // 수강생은 Worker, Person만 가능  
    public static void registerCourseWorker(Course<? super Worker> course) {  
        System.out.println(course.getName() + " 수강생: " +  
            Arrays.toString(course.getStudents()));  
    }  
}
```



스트림

■ 기본 타입 입출력 보조 스트림

- **DataInputStream / DataOutputStream**
- **기본 데이터 타입으로 입출력 가능**
- **FileInputStream / DataOutputStream 메소드**
 - read기본타입() : readInt(), readDouble(), ...
 - write기본타입() : writeInt(), writeDouble(), ...
 - 문자열 (String) 경우 : readUTF(), writeUTF()
 - readString(), writeString()은 없음

소켓

- 서버가 클라이언트와 통신하는 과정
 - 서버 소켓 생성
 - 서버는 접속을 기다리는 포트로 선택
 - 클라이언트로부터 접속 기다림
 - `accept()` 메소드는 연결 요청이 오면 새로운 `Socket` 객체 반환
 - 접속 후 새로 만들어진 `Socket` 객체를 통해 클라이언트와 통신
 - 네트워크 입출력 스트림 생성
 - `Socket` 객체의 `getInputStream()`과 `getOutputStream()` 메소드를 이용하여 입출력 데이터 스트림 생성

ServerSocket

① // 서버 소켓 생성 (클라이언트 기다림)
serverSocket = new ServerSocket(9999);

③ // 클라이언트로부터 요청이 오면 통신용 소켓 생성
socket = serverSocket.accept();

Socket

//소켓 스트림을 이용하여 데이터 입출력 진행

④ // 데이터 수신
InputStream is = socket.getInputStream()
//보조 스트림 사용 가능

⑤ // 데이터 전송
OutputStream os = socket.getOutputStream()
//보조 스트림 사용 가능

⑥ socket.close()

서버

Socket

② // 통신용 클라이언트 소켓 생성
socket = new Socket("localhost", 9999);

//소켓 스트림을 이용하여 데이터 입출력 진행

④' // 데이터 수신
InputStream is = socket.getInputStream()
//보조 스트림 사용 가능

⑤' // 데이터 전송
OutputStream os = socket.getOutputStream()
//보조 스트림 사용 가능

socket.close()

클라이언트

접속