Java Programming

CPT111 – Lecture 10
Erick Purwanto

Xi'an Jiaotong-Liverpool University
西交利物浦大学

CPT111 Java Programming

Lecture 10

OOP Principles, Polymorphism and Exception Handling

Welcome!

- Welcome to Lecture 10 !
- Last week, we have learned about Class, Object, Inheritance
  - constructors, instance/class variables/methods, extends, overloading vs overriding, polymorphism/dynamic method selection
- In this lecture we are going to review and learn about
  - OOP Principles
    - Encapsulation
    - Inheritance
    - Polymorphism
  - Exception
    - Throwing Exception
    - Handling Exception

Part 1: OOP Principles

- Let us review what we have learned so far in Week 7 and Week 9, have a closer look from the lenses of Object-Oriented Programming

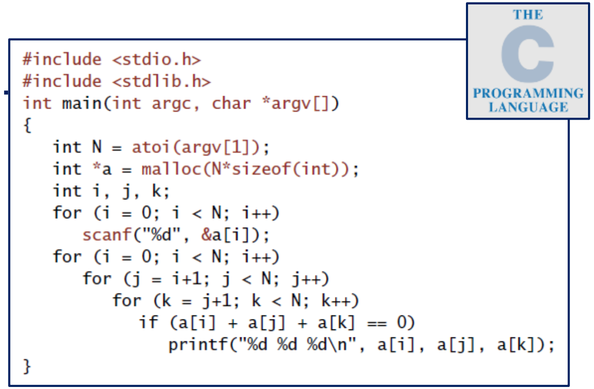
Object-oriented Programming

- Goal: design software to model and simulate the real world
    - because we know how the real world works

- Object-oriented programming (OOP)
    - Programming based on *data types* as classes
        - classes are the template for instances/objects
    - Identify things that are parts of the instances:
        - instances in the world *have* or *know* something: *instance variables*
        - instances in the world *do* something: *instance methods*

Procedural Programming

- Before OOP, we have *procedural programming*
  - tell the computer to do this
  - then tell the computer to do that
  - ...



```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
   int N = atoi(argv[1]);
   int *a = malloc(N*sizeof(int));
   int i, j, k;
   for (i = 0; i < N; i++)
      scanf("%d", &a[i]);
   for (i = 0; i < N; i++)
      for (j = i+1; j < N; j++)
         for (k = j+1; k < N; k++)
            if (a[i] + a[j] + a[k] == 0)
               printf("%d %d %d\n", a[i], a[j], a[k]);
}
```

- Procedural programming is VERB- oriented
  - Object-oriented programming is NOUN-oriented

Features of OOP

- Features of OOP
  - *type checking* (*W1*) makes it easier to avoid and find errors
  - *encapsulation* (*W6*) hides information to make programs more robust
  - *inheritance* and *polymorphism* (*W8*) enable code reuse

- Other features of (non-OOP) programming:
  - *recursion* is a programming technique where a function calls itself (*W13*)
  - *immutability* guarantees stability of program data
    - when you declare an instance variable as `final` (*W9*),
      you promise to assign it a value *only once*
    - more in *CPT204 Advanced OOP* course!

鬼滅の刃 KIMETSU NO YAIBA

Demon Slayer Corps

- Let us continue using Demon Slayer Corps as our running example in this lecture!

- Recall that we have built four classes in Lecture 8:
  - `Swordsman` class to model a regular *swordsman*
  - `Successor` class to model a *stronger swordsman*
  - `Pillar` class to model a *much more stronger swordsman*
  - `DemonSlayerCorps` class to instantiate *a troop of swordsmen*

- We are going to add more elements to review and illustrate *OOP principles*

Review: Encapsulation

- The process of separating clients from implementations by *hiding information* is known as encapsulation

- We use encapsulation to
  - enable modular programming
  - facilitate debugging
  - clarify program code

Encapsulation: Private

- When you declare an instance variable (or method) to be private
    - you are making it impossible for any client (code in another class) to *directly* access that instance variable (or method)

- Private methods are also called *helper methods*
    - used to help writing instance methods
    - not to be used by client classes

In-Class Quiz 10.1: Instance Variable

- We want to keep track the health point of a Swordsman
- Add that information in the `Swordsman` class as an instance variable:

```
public class Swordsman {
    // instance variables
    ...
    // constructors, instance methods
    ...
}
```

?

○ `double healthPoint;`

○ `public double healthPoint;`

○ `private double healthPoint;`

○ `private final double healthPoint;`

add getters and setters yourself

Encapsulation: Limiting the potential for error

- Encapsulation also helps programmers ensure that their code operates as intended
- For example, the health point of a swordsman:
    - starts from 100.0
    - always between 0.0 and 100.0
    - reduced by receiving some damage point
    - if it ever reaches 0.0, he dies and stays 0.0 forever

In-Class Quiz 10.2: Constructor

- The health point of a Swordsman starts from 100.0
- Add that initialization in the Swordsman constructor:

```
public class Swordsman {

    public Swordsman(String name) {
        this.name = name;
        ...
    }
}
```

?

- ○ `healthPoint = 100.0;`
- ○ `double healthPoint = 100.0;`
- ○ `private double healthPoint = 100.0;`
- ○ `private double healthPoint == 100.0;`

In-Class Quiz 10.3: Overloading

- The initial health point of a Swordsman can also be passed to constructor
- Add that initialization in the Swordsman constructor:

```java
public class Swordsman {

    public Swordsman(String name, double healthPoint) {
        this(name);
        ...
    }
}
```

?

○ `healthPoint == healthPoint;`

○ `healthPoint = healthPoint;`

○ `this.healthPoint == healthPoint;`

○ `this.healthPoint = healthPoint;`

Review: this keyword

- Within a constructor (or an instance method), `this` keyword gives us a way *to refer to the object* whose constructor (or instance method) is being called
  - useful to call the other constructors, and
  - useful to refer to an instance variable with the same name as a local variable

In-Class Quiz 10.4: Instance Method 1

- A Swordsman receives damage point reducing his health point
  - always between 0.0 and 100.0
  - if it ever reaches 0.0, he dies and stays 0.0 forever

```
public double receiveDamage(double damagePoint) {
    healthPoint = healthPoint - damagePoint;
    if ( ... ) {
        healthPoint = 0.0;
        alive = false;
    }
    return healthPoint;
}
```

?

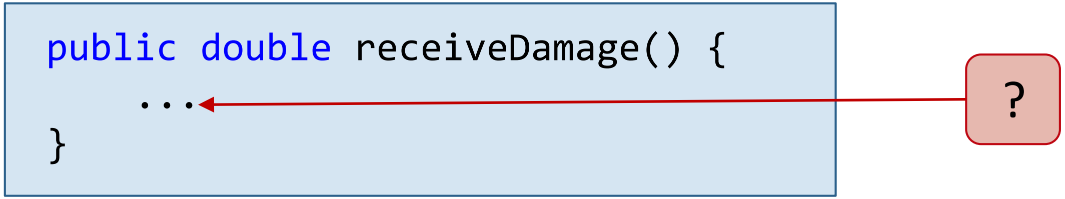
- ○ `healthPoint > 0.0`
- ○ `healthPoint = 0.0`
- ○ `healthPoint == 0.0`
- ○ `healthPoint <= 0.0`

the method also returns the healthPoint after receiving damage

modify constructors, getters and setters as well yourself

In-Class Quiz 10.5: Instance Method 2

- Overload the `receiveDamage` method to the one without parameter
  - the unspecified damage is *ten-percent* of the health point



```
public double receiveDamage() {
    ...
}
```

?

- ○ `0.1 * healthPoint;`
- ○ `return 0.1 * healthPoint;`
- ○ `receiveDamage(0.1 * healthPoint);`
- ○ `return receiveDamage(0.1 * healthPoint);`

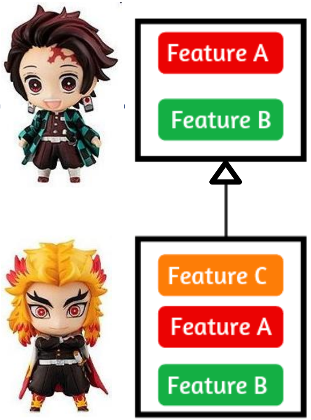
Review: Overloading

- To overload a constructor (method):
  - keep the same name
  - change at least one:
    - number of parameters
    - type of parameters
    - order of parameters

- Try yourself: what if you change (only) the return type instead?

- Overloading **increases readability of the code**

Review: Inheritance



- Another OOP feature of Java is called ***inheritance*** or *subclassing*
- `Pillar` is the *subclass* or *child class* that ***inherits*** variables and methods from `Swordsman`, its *superclass* or *parent class*
  - enabling code reuse
- We will also *overrides* a method of `Swordsman` in `Pillar`
  - same signature (name + parameters)

In-Class Quiz 10.6: Overriding

- A `Pillar` only receives *half* of the damage point reducing their health point

```java
@Override
public double receiveDamage(double damagePoint) {
    damagePoint = 0.5 * damagePoint;
    return ... ;
}
```

?

○ `receiveDamage(damagePoint)`

○ `this.receiveDamage(damagePoint)`

○ `super.receiveDamage(damagePoint)`

○ `super.this.receiveDamage(damagePoint)`

Protected Access Modifier

- We can access protected instance variables or methods in a superclass from its subclasses
  - for example:

```
public class Swordsman {

   protected int numDemonsKilled;

```

```
public class Pillar extends Swordsman {

    @Override
    public String toString() {
        return type + " Pillar " + getName() + " has killed " +
               numDemonsKilled + " demons";
    }

```

can *directly* access inherited instance variable from `Swordsman` in `Pillar` subclass

Default Access Modifier

- There are *four* access modifiers in Java, visibility increases in the order:
  `private` → *default (no modifier)* → `protected` → `public`

Access Modifier/From	Same Class	Same Package	Subclass in Different Package	Different Package
`private`	✓	✗	✗	✗
(no modifier)	✓	✓	✗	✗
`protected`	✓	✓	✓	✗
`public`	✓	✓	✓	✓

Polymorphism

- Polymorphism ≈ many forms

- Polymorphism in Java ≈ perform a single action
  - executed in different ways

- There are two types of polymorphism in java:
  - *compile time / static polymorphism* by method overloading
  - *runtime / dynamic polymorphism* by method overriding

In-Class Quiz 10.7: Polymorphism

- A `Swordsman` variable is used to reference a `Pillar` object
  - `receiveDamage(10)` is called on it

```java
public static void main(String[] args) {
    Swordsman kyojuro = new Pillar("Kyojuro", 1000, "Fire");
    System.out.println(kyojuro.receiveDamage(10));
}
```

- What is the output?
  - no output, there is a compile error
  - `90.0` since `receiveDamage` of `Swordsman` is called
  - `95.0` since `receiveDamage` of `Pillar` is called

Dynamic Method Selection

```java
Swordsman kyojuro = new Pillar("Kyojuro", 1000, "Fire");
```

static type

dynamic type

- When the Java Virtual Machine calls an instance method, it locates the method of the implicit class based on the dynamic type
  - this form of dynamic polymorphism is called *dynamic method selection*

- More in CPT204: Advanced OOP course ...

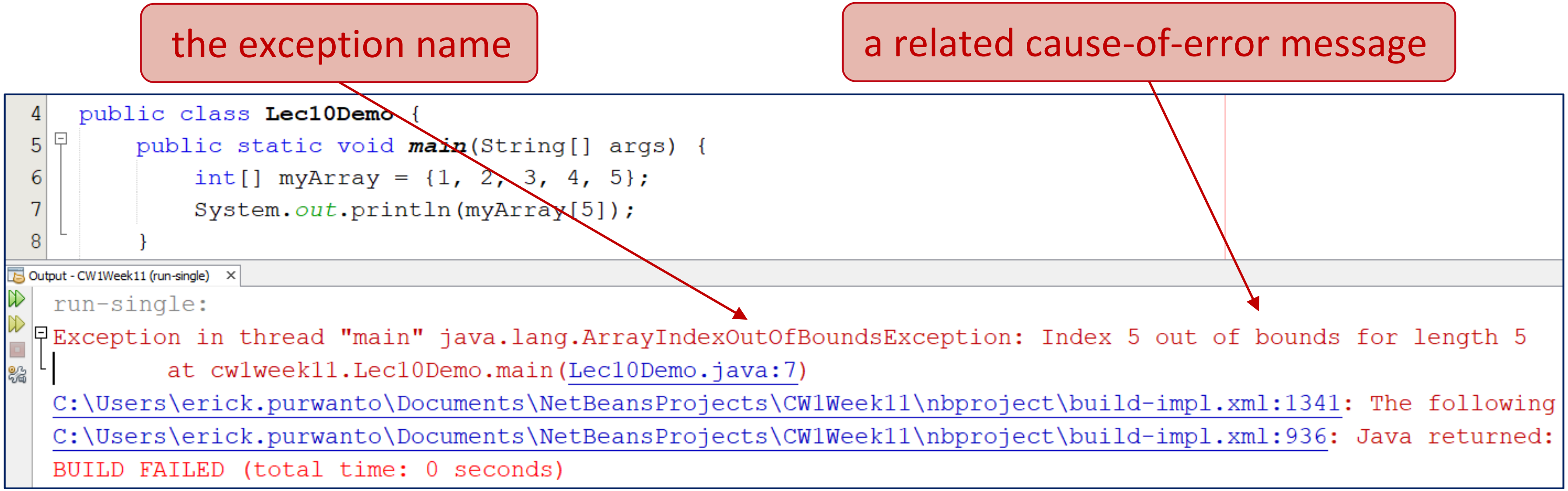
Part 2: Exception

- In this part, we will learn about exception
  - when do we need to throw an exception
  - how do we throw an exception
  - how do we catch an exception

- We are going to only barely touch the surface
  - you will learn more in *CPT204: Advanced Object Oriented Programming*

ArrayIndexOutOfBoundsException

- We have seen an exception getting thrown before!
  - an `ArrayIndexOutOfBoundsException` object is thrown during runtime
  - since we are trying to access an invalid array index when we run the program



the exception name

a related cause-of-error message

```java
public class Lec10Demo {
    public static void main(String[] args) {
        int[] myArray = {1, 2, 3, 4, 5};
        System.out.println(myArray[5]);
    }
```

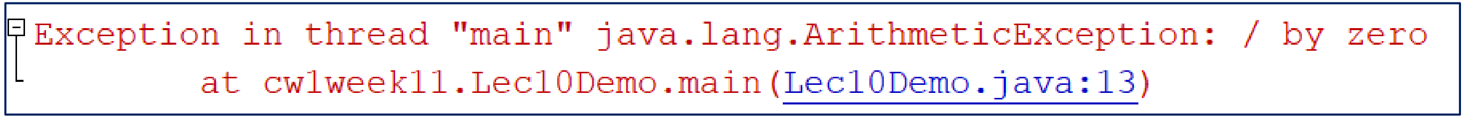
```
run-single:
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5
        at cw1week11.Lec10Demo.main(Lec10Demo.java:7)
C:\Users\erick.purwanto\Documents\NetBeansProjects\CW1Week11\nbproject\build-impl.xml:1341: The following
C:\Users\erick.purwanto\Documents\NetBeansProjects\CW1Week11\nbproject\build-impl.xml:936: Java returned:
BUILD FAILED (total time: 0 seconds)
```

Runtime Error

- When something unexpected happens while the program is running, JVM throws an exception object
  - instead of returning a special value, such as -1 when String method `indexOf` does not find the search value

```java
Scanner sc = new Scanner(System.in);
int n = Integer.parseInt(sc.nextLine());
System.out.println(1/n);
```

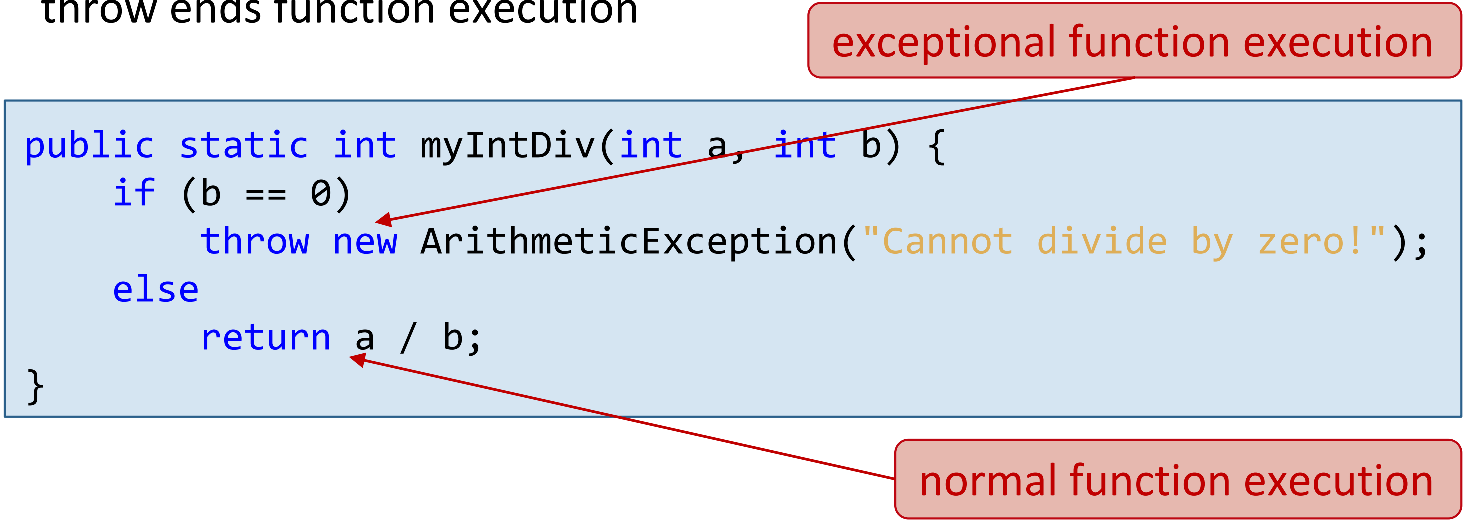
try entering a zero!



```
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at cw1week11.Lec10Demo.main(Lec10Demo.java:13)
```

Throwing our own exception object

- We can throw an exception object in our function (static method) or method (instance method) whenever something unexpected happens
  - for example, we create our own integer division function that throws an `ArithmeticException` object whenever the divisor is zero
  - throw ends function execution



```
public static int myIntDiv(int a, int b) {
    if (b == 0)
        throw new ArithmeticException("Cannot divide by zero!");
    else
        return a / b;
}
```

Catching an exception object 1

- After that, the method that calls the function catch the exception
  - try block : put the risky method call here, and what to do next if execution turns out to be normal

the risky function that might throw an exception

```java
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int n = Integer.parseInt(sc.nextLine());
    try {
        System.out.println(myIntDiv(1, n));
    }
    catch (ArithmeticException e) {
        System.out.println("You entered a zero!");
    }
}
```

Catching an exception object 2

- We can also display the error message
    - try block : put the risky method call here, and what to do next if execution turns out to be normal
    - catch block : what to do instead if an exception is caught

```java
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int n = Integer.parseInt(sc.nextLine());
    try {
        System.out.println(myIntDiv(1, n));
    }
    catch (ArithmeticException e) {
        System.out.println("You entered a zero!");
    }
}
```

if an exception is thrown anywhere in try block, the program execution jumps to catch block

after that, the execution continues

Displaying the error message

- After that, the method that calls the function catch the exception
  - check few slides before, the error message is passed to the exception constructor thrown by the function
  - use getMessage() method called on the caught exception object

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int n = Integer.parseInt(sc.nextLine());
    try {
        System.out.println(myIntDiv(1, n));
    }
    catch (ArithmeticException e) {
        System.out.println(e.getMessage());
    }
}
```

the exception object is caught and named this parameter e

In-Class Quiz 10.8: Throwing IllegalArgumentException

- A Swordsman's damage point received **must be positive**
  - otherwise, argument is invalid and throw an Illegal Argument Exception

```
public double receiveDamage(double damagePoint) {
    if ( ... )
        throw ...

    healthPoint = healthPoint - damagePoint;
    // code omitted
    return healthPoint;
}
```

?

?

- `damagePoint > 0.0` `new IllegalArgumentException();`
- `damagePoint > 0.0` `IllegalArgumentException();`
- `damagePoint <= 0.0` `new IllegalArgumentException();`
- `damagePoint <= 0.0` `IllegalArgumentException();`

In-Class Quiz 10.9: Catching IllegalArgumentException

- In a main method that calls the `receiveDamage` method:

```java
try {
    double newHP = kyojuro.receiveDamage(-10);
    System.out.println(newHP);
}
catch ( ... ) {
     System.out.println("Illegal non-positive damage point detected");
}
```

?

○ `IllegalArgumentException`

○ `IllegalArgumentException iae`

○ `IllegalArgumentException()`

○ `new IllegalArgumentException()`

Frequently-used Exception Classes

- For this introductory course, we can just use these popular exception classes:
  - `ArrayIndexOutOfBoundsException`
  - `ArithmeticException`
  - `IllegalArgumentException`
  - `NumberFormatException` – thrown by `parseInt`/`parseDouble` if the `String` given cannot be parsed into `int`/`double`
  - and a few more related to I/O or data structures in future lectures ...

What are the advantages of OOPs concepts? (1)

- Simplicity:
  - OOP objects model real world objects,
    so the complexity is reduced and the program structure is clear
- Modularity:
  - each object forms a separate entity whose internal workings are decoupled from other parts of the system
- Modifiability:
  - changes inside a class do not affect any other part of a program,
    since the only public interface that the external world has to a class is through the use of methods

What are the advantages of OOPs concepts? (2)

- Extensibility:
  - adding new features or responding to changing operating environments can be solved by introducing a few new objects and modifying some existing ones
- Maintainability:
  - objects can be maintained separately,
    making locating and fixing problems easier
- Reusability:
  - objects can be reused in different programs

Thank you for your attention !

- In this lecture, you have learned:
  - how to use encapsulation to limit access
  - how to reuse code by inheritance
  - how to use overloading and overriding to apply polymorphism
  - how to throw and catch an exception indicating exceptional cases

- Please continue to Lab 10 to complete Lab Tasks, and then solve
  - Exercise #10.1 - #10.4 and
  - CW1 #10.1 - #10.3