Java Programming

CPT111 – Lecture 9

Erick Purwanto

Xi'an Jiaotong-Liverpool University

西交利物浦大学

CPT111 Java Programming

Lecture 9

More Objects and Inheritance

Welcome!

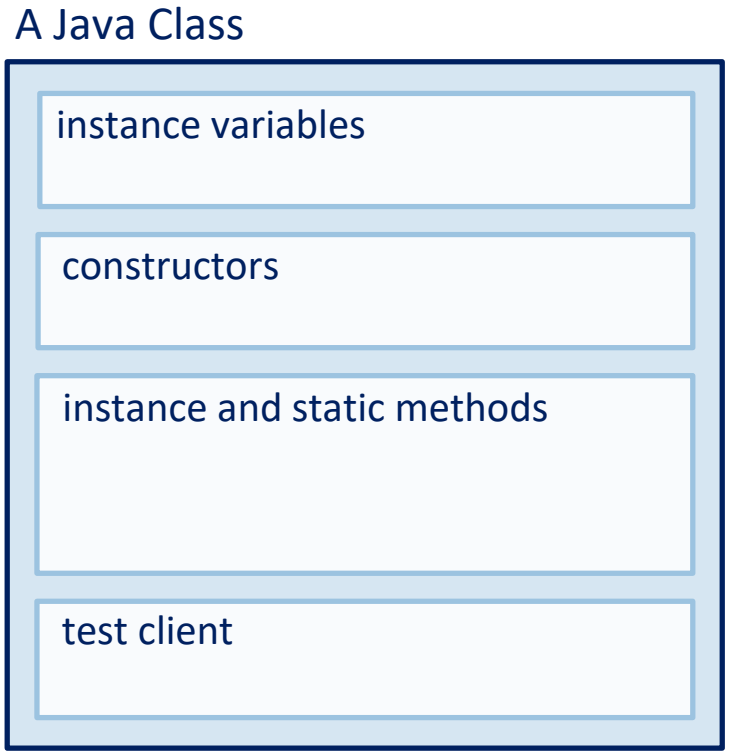
- Welcome to Lecture 9 !
- Last week, we have learned about
  - Class and Objects
    - instance variables, constructor, `this`, instance methods, getter, setter, class/static variables, `toString()`, array of objects
  - Encapsulation
    - `public` vs `private`
- In this lecture we are going to learn about
  - More Objects
    - review Week 7, `final`
  - Inheritance
    - `extends`, `super`, overloading vs overriding, annotation, polymorphism

Part 1: More Objects

- In Week 7, we have learned about objects
  - we will review and explore more about them this week

Review: Java Class

- In Java, we implement a ***type*** (such as `Person`, `StockPrice`) in a class



declare variables names and types

initialize instance variables

access or modify instance variables

Demon Slayer Corps

- The Demon Slayer Corps is a organization that exist to protecting humanity from demons
  - Swordsmen are members of the Demon Slayer Corps that hunts down the demons

- We will use this as our running example in this lecture!

- Classes:
  - `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 create/instantiate *a troop of swordsmen*

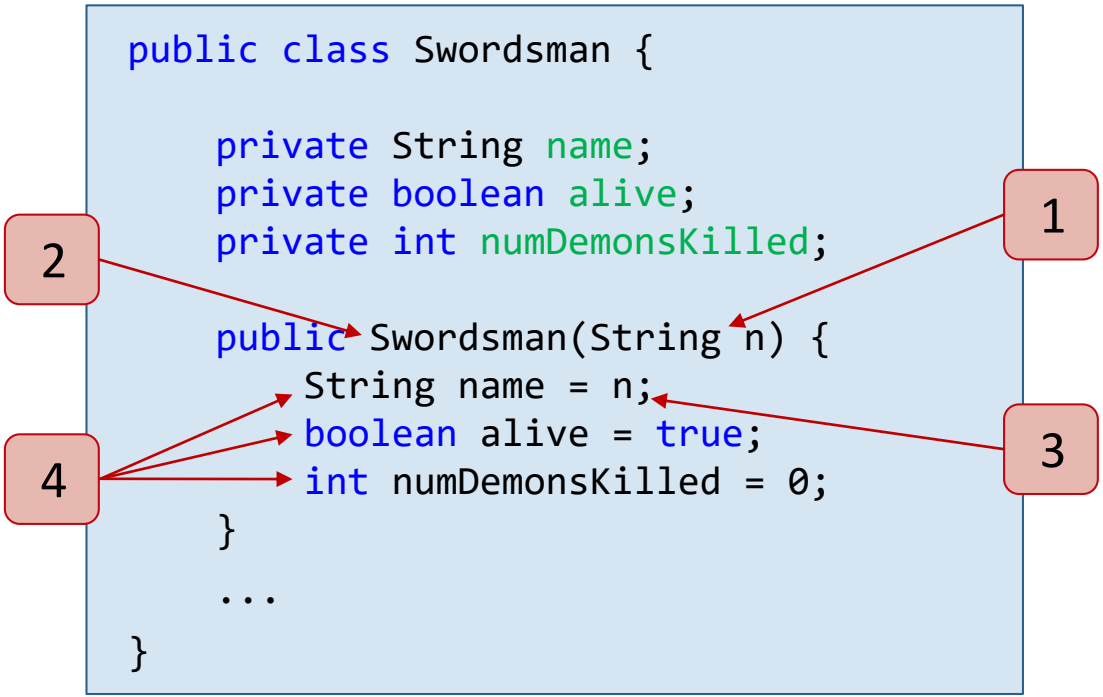
Review: Instance Variable

- A swordsman has a name
  - we also want to keep track whether they are alive, and the number of demons they have killed

```
public class Swordsman {

    private String name;
    private boolean alive;
    private int numDemonsKilled;

    ...

}
```

In-Class Quiz 9.1: Constructor

- The following implementation of constructor has a bug
  - where is the bug?



- 1
- 2
- 3
- 4

Review: Constructor

- Use a constructor to initialize instance variables
  - use `this.` to refer to instance variables having same name as parameters / local variables

```
public class Swordsman {

    private String name;
    private boolean alive;
    private int numDemonsKilled;

    public Swordsman(String name) {
        this.name = name;
        alive = true;
        numDemonsKilled = 0;
    }

    ...
}
```

scope of instance variable name

scope of local variable name

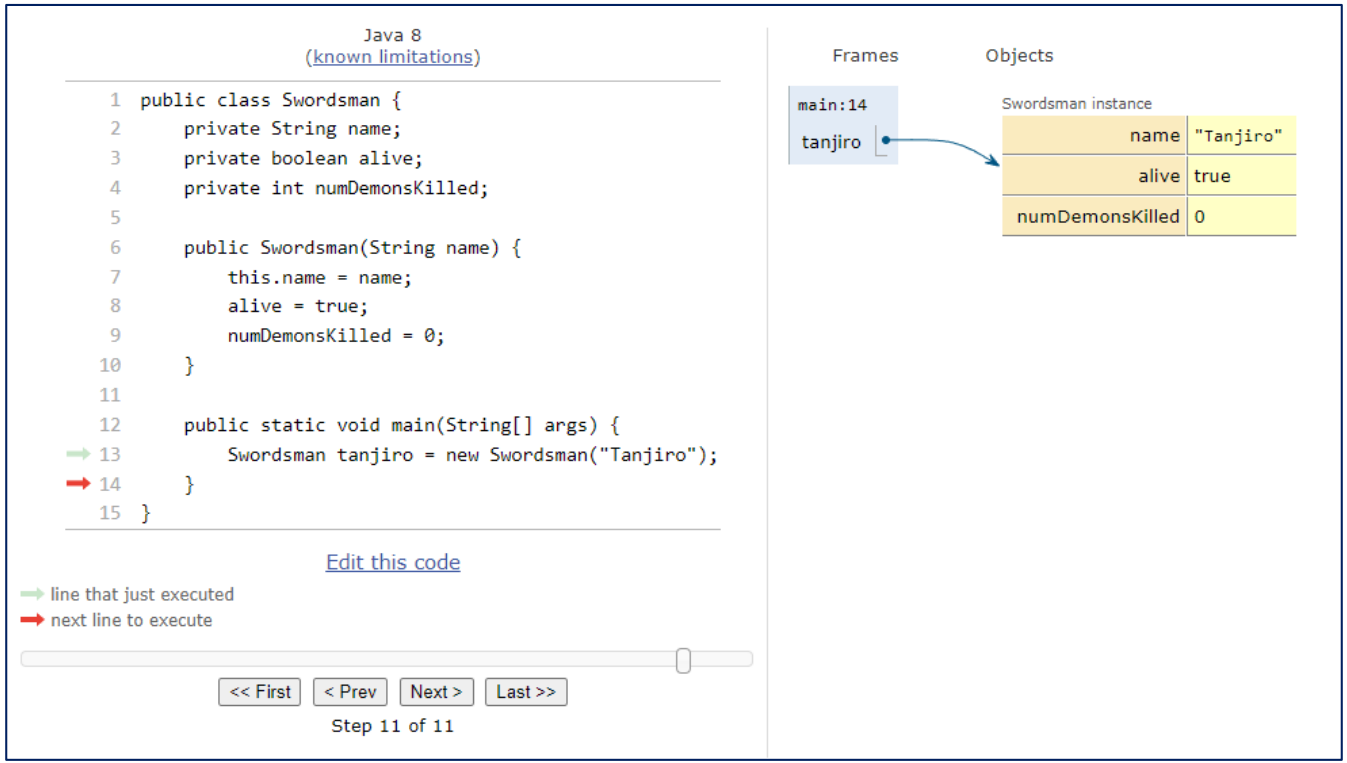
Review: Test Client, Object Instantiation

- Use main method in class to implement a simple test client
  - call constructors to create (instantiate) and initialize a new object (instance) with operator `new`, to return an object of the type, a `Swordsman` object

```
public class Swordsman {

    ...

    public static void main(String[] args) {
        Swordsman tanjiro = new Swordsman("Tanjiro");
    }
}
```

Review: Reference Type

- Variable tanjiro references/points to a `Swordsman` object/instance



Review: Private

- By setting the instance variables to be `private`, client classes such as `DemonSlayerCorps` *cannot* directly access or change their values

```
public class Swordsman {

    private String name;
    ...
}
```

```
public class DemonSlayerCorps {

    ...

    public static void main(String[] args) {
        Swordsman tanjiro = new Swordsman("Tanjiro");
        tanjiro.name = "Inosuke";  // compile error
    }
}
```

Review: ToString() Method

- `toString()` is a method that returns string representation of this object

```
public class Swordsman {

    ...
    public String toString() {
        String alive;
        if (this.alive) alive = "alive";
        else            alive = "dead";
        return "Swordsman " + name + " is " + alive +
               " and has killed " + numDemonsKilled + " demons";
    }
```

Review: ToString() Method

- `toString()` is a method that returns string representation of this object
  - it enables an object to be printed

```
public class Swordsman {

    ...
    public String toString() {
        String alive;
        if (this.alive) alive = "alive";
        else            alive = "dead";
        return "Swordsman " + name + " is " + alive +
               " and has killed " + numDemonsKilled + " demons";
    }
    public static void main(String[] args) {
        Swordsman tanjiro = new Swordsman("Tanjiro");
        System.out.println(tanjiro);
    }
}
```

automatically calls `tanjiro.toString()`

Review: Getter and Setter

- Getter / Accessor : return the value of an instance variable
- Setter / Mutator : set the instance variable to a new value

```
public class Swordsman {
    private String name;
    ...
    public String getName() {
        return name;
    }

    public void setName(String newName) {
       name = newName;
    }
    ...
}
```

add getter for other instance variables!

Review: Instance Method

- Instance method performs operation on instance variables of the instance
  - Instance method is called on an object

```
public class Swordsman {
    private int numDemonsKilled;
    ...
    public void killsDemon() {
        numDemonsKilled++;
    }

    public static void main(String[] args) {
        Swordsman tanjiro = new Swordsman("Tanjiro");
        tanjiro.killsDemon();
        System.out.println(tanjiro);
    }
}
```

there is one input to `killsDemon` instance method: `this` object

Review: Method Overloading

- Method overloading: same name, different signature
  - same method name, different parameter numbers or types

```
public class Swordsman {
    private int numDemonsKilled;
    ...
    public void killsDemon() {
        numDemonsKilled++;
    }

    public void killsDemon(int demonsKilled) {
        numDemonsKilled += demonsKilled;
    }

    ...
}
```

there are two inputs to `killsDemon` instance method: `this` object and `int demonsKilled`

Review: Constructors Overloading

- Constructor overloading: same name, different signature
  - we can call *other* constructors by using `this()`

```
public class Swordsman {
    public Swordsman(String name) {
        this.name = name;
        alive = true;
        numDemonsKilled = 0;
    }
    public Swordsman(String name, int numDemonsKilled) {
        this(name);
        this.numDemonsKilled = numDemonsKilled;
    }
    public Swordsman() {
        this("Nameless");
    }
}
```

empty constructor with no parameters

Review: Static Method / Function (Week 5, 6)

- Just like instance methods, it can access and modify instance variables
  - however, a static method is *not necessarily* called on an object

```
public static boolean sameNumDemonsKilled(Swordsman s1, Swordsman s2) {
    return s1.numDemonsKilled == s2.numDemonsKilled;
}

public static void main(String[] args) {
    Swordsman tanjiro = new Swordsman("Tanjiro");
    tanjiro.killsDemon();

    Swordsman zenitsu = new Swordsman("Zenitsu", 1);

    boolean sameKills = sameNumDemonsKilled(tanjiro, zenitsu);
    System.out.println(sameKills);
}
```

Review: Class Variable / Static Variable

- Unlike instance variables where there are separate variables for each instance, there is *only one* class variable, shared among all instances

```
public class Swordsman {

    private String name;
    private boolean alive;
    private int numDemonsKilled;

    private static int numSwordsman = 0;

    public Swordsman(String name) {
        this.name = name;
        alive = true;
        numDemonsKilled = 0;
        numSwordsman++;
    }

}
```

initialized in the class:
initially, there are 0 swordman

add one to `numSwordsman` each time a new `Swordman` object is created

Review: Accessing Class Variable

- To access a class variable, we can use *both* static or instance methods

```
public class Swordsman {

    public static int getNumSwordsman() {
        return numSwordsman;
    }

    public int getNumSwordsmanInst() {
        return numSwordsman;
    }

    public static void main(String[] args) {
        ...
        System.out.println(getNumSwordsman());
        System.out.println(tanjiro.getNumSwordsmanInst());
    }
}
```

Review: Array of Objects

- Just like `int` or `String`, we can create an array of `Swordsman` objects

```
public class DemonSlayerCorps {

    public static void main(String[] args) {

        Swordsman[] swordsmanTroop = new Swordsman[2];
        swordsmanTroop[0] = new Swordsman("Tanjiro");
        swordsmanTroop[1] = new Swordsman("Zenitsu", 1);

        for (int i = 0; i < swordsmanTroop.length; i++) {
            System.out.println(swordsmanTroop[i]);
        }
    }
}
```

Part 2: Inheritance

- Another OOP feature of Java is called ***inheritance*** or *subclassing*
- The idea is to define a new class (*subclass* or *child class*) that ***inherits*** instance variables (attribute, state) and instance methods (operation, behavior) from another class (*superclass* or *parent class*)
    - enabling code reuse
- Furthermore, we will learn to make the subclass ***redefines*** or *overrides* some of the methods in the superclass

Pillars

- Pillars are the most powerful swordsmen of the Demon Slayer Corps

- There are nine types of Pillars:
  - Stone Pillar
  - Flame Pillar
  - Sound Pillar
  - Mist Pillar
  - Water Pillar
  - Serpent Pillar
  - Love Pillar
  - Insect Pillar
  - Wind Pillar

- We will implement a `Pillar` object using inheritance in this lecture

Subclass and Superclass

- Since a Pillar ***is a*** Swordsman, a Pillar has all attributes that a Swordsman has, and a Pillar can do everything that a Swordsman can (and potentially more)
  - instead of copying all the codes from `Swordsman` class to `Pillar` class, we set `Pillar` class to *extend* the `Swordsman` class

```
public class Pillar extends Swordsman {

}
```

- We say that `Pillar` is a *subclass* of `Swordsman`, and `Swordsman` is the *superclass* of `Pillar`
  - or, `Pillar` is a *child class* of `Swordsman`, and `Swordsman` is the *parent class* of `Pillar`
  - in Java, a class can only be a subclass of one superclass, but a superclass can be extended by many subclasses

Subclass Constructor and super()

- A subclass inherits all of the superclass' instance variables
  - however, the subclass *cannot* directly initialize the inherited instance variables that are declared in the superclass as private
  - use `super()` to call the superclass' constructor and pass the initial values

```
public class Pillar extends Swordsman {

    public Pillar(String name) {
        super(name);
    }

    public static void main(String[] args) {
        Pillar kyojuro = new Pillar("Kyojuro");
    }
}
```

Inherited Methods

- We can call the inherited public methods on the superclass' instance

```
public class Pillar extends Swordsman {

    public Pillar(String name) {
        super(name);
    }

    public static void main(String[] args) {
        Pillar kyojuro = new Pillar("Kyojuro");
        System.out.println(kyojuro.getName());
        System.out.println(kyojuro);
    }
}
```

In-Class Quiz 9.2: Shadowing Instance Variables

- What will happen if we try to solve the problem by declaring an instance variable with the same name?

```
public class Pillar extends Swordsman {

    private String name;

    public Pillar(String name) {
        this.name = name;
    }

    public static void main(String[] args) {
        Pillar kyojuro = new Pillar("Kyojuro");
        System.out.println(kyojuro.getName());
    }
}
```

- compile error
- runtime error
- works well, prints "Kyojuro"
- prints something else

Overloading Constructor

- We can overload the constructor
  - match the new constructor with the overloaded superclass constructor

```
public class Pillar extends Swordsman {

    public Pillar(String name) {
        super(name);
    }

    public Pillar(String name, int numDemonsKilled) {
        super(name, numDemonsKilled);
    }

    public static void main(String[] args) {
         Pillar kyojuro = new Pillar("Kyojuro", 1000);
    }
}
```

Add instance variables

- We can introduce new instance variables

```
public class Pillar extends Swordsman {

    private String type;

    public Pillar(String name, String type) {
        super(name);
        this.type = type;
    }

    public Pillar(String name, int numDemonsKilled, String type) {
        super(name, numDemonsKilled);
        this.type = type;
    }

    public static void main(String[] args) {
         Pillar kyojuro = new Pillar("Kyojuro", 1000, "Fire");
    }
}
```

Add instance methods

- We can introduce new instance methods

```
public class Pillar extends Swordsman {

    private String type;

    ...

    public String getType() {
        return type;
    }

    ...
}
```

Final Instance Variable

- If the instance variable will *never* be changed after it is initialized, we can declare it to be *final*:

```
public class Pillar extends Swordsman {

    private final String type;

    public Pillar(String name, String type) {
        super(name);
        this.type = type;
    }

    ...

}
```

any attempt to modify `type` will generate compile error!

for instance variables that will possibly be changed in the future, such as `numDemonsKilled`, do *not* declare it to be `final`

Constant

- To define **a constant**, we use keywords *static final*

```
public class Swordsman {

    private int healthPoint;

    public static final int MAX_HP = 100;

    public Swordsman(String name) {
        this.name = name;
        alive = true;
        numDemonsKilled = 0;
        numSwordsman++;
        healthPoint = MAX_HP;
    }

}
```

Redefining Superclass Method in Subclass

- Let's say a Swordsman has an attack damage according to formula

```
attackDamage = 10 + 5 * numDemonsKilled
```

- We can add an instance method to `Swordsman` class:

```
public class Swordsman {
    public int attackDamage() {
        return 10 + 5 * numDemonsKilled;
    }
}
```

- However, a Pillar is much more powerful than a regular Swordsman
  - a Pillar's attack damage is

```
attackDamage = 1000 + 100 * numDemonsKilled
```

need to redefine `attackDamage()` method in `Pillar` class!

Overriding Method

- We redefine the inherited `attackDamage()` method of `Swordsman` class:

```
public class Swordsman {
    public int attackDamage() {
        return 10 + 5 * numDemonsKilled;
    }
}
```

to a method in `Pillar` class with the *same signature*:

```
public class Pillar extends Swordsman {
    public int attackDamage() {
        return 1000 + 100 * getNumDemonsKilled();
    }
}
```

- This is called *overriding* the superclass method

Overloading vs Overriding

- Overloading: on same class, same name, different signature
- Overriding: on subclass, same signature

```
public class Swordsman {
    public int attackDamage() {
        return 10 + 5 * numDemonsKilled;
    }
}
```

```
public class Pillar extends Swordsman {
    public int attackDamage() {
        return 1000 + 100 * getNumDemonsKilled();
    }
}
```

In-Class Quiz 9.3: Overriding Method

- After writing the method below, what is the output ?

```
public class Pillar extends Swordsman {

    public int attackDammage() {
        return 1000 + 100 * getNumDemonsKilled();
    }

    public static void main(String[] args) {
        Pillar kyojuro = new Pillar("Kyojuro", 1, "Fire");
        System.out.println(kyojuro.attackDamage());
    }
}
```

- 1
- 15
- 1000
- 1100

@Override Annotation

- We can put an annotation before the overriding method:

```
public class Pillar extends Swordsman {

    @Override
    public int attackDamage() {
        return 1000 + 100 * getNumDemonsKilled();
    }
}
```

- The purpose is
  - if we made a mistake such as typo on the method name or parameter
    - then we will get a compile error
  - without annotation, instead of overriding, we may overload the superclass method or introduce a new method *by mistake*

Calling Overriden Method

- Suppose that a Pillar's attack damage is

  `attackDamage = 1000 + 100 * attackDamage` as a `Swordsman`

- We can *still* call the overridden `attackDamage()` method of the superclass `Swordsman` by using `super.`

```
public class Pillar extends Swordsman {

    @Override
    public int attackDamage() {
        return 1000 + 100 * super.attackDamage();
    }
}
```

without using `super.` the method will call itself recursively, stay tuned, we will discuss Recursion in Week 13

Overriding toString()

- We actually have been doing overriding when we write the `toString()` method
  - we overrode the `toString()` method of the `Object` class

```
public class Swordsman {

    @Override
    public String toString() {
        String alive;
        if (this.alive) alive = "alive";
        else            alive = "dead";
        return "Swordsman " + name + " is " + alive +
               " and has killed " + numDemonsKilled + " demons";
    }
}
```

local variable `alive`

- In Java, every class is a subclass of `Object` class

Final Instance Method

- If you declare **an instance method** to be *final*, it cannot be overridden

```
public class Swordsman {
    public final int attackDamage() {
        return 10 + 5 * numDemonsKilled;
    }
}
```

```
public class Pillar extends Swordsman {
    public int attackDamage() {
        return 1000 + 100 * getNumDemonsKilled();
    }
}
```

attackDamage() in Pillar cannot override attackDamage() in Swordsman
overridden method is final

Java's Object superclass

- As every class is a subclass of the java.lang.Object class, every class inherits

Modifier and Type	Method and Description
protected Object	clone() Creates and returns a copy of this object.
boolean	equals(Object obj) Indicates whether some other object is "equal to" this one.
protected void	finalize() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?>	getClass() Returns the runtime class of this Object.
int	hashCode() Returns a hash code value for the object.
void	notify() Wakes up a single thread that is waiting on this object's monitor.
void	notifyAll() Wakes up all threads that are waiting on this object's monitor.
String	toString() Returns a string representation of the object.
void	wait() Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
void	wait(long timeout) Causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
void	wait(long timeout, int nanos) Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

we have learned about `toString()` method

we will learn about `equals()` method in the Advanced OOP course

Override toString() in Pillar

- We can override the `toString()` of `Swordsman` in `Pillar` to get a better description of a `Pillar` object/instance

```
public class Pillar extends Swordsman {

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

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

Converting between Subclass and Superclass (1)

- Sometimes we want to convert a subclass type to a superclass type
- We have that `Pillar` class extends `Swordsman` class
  - a `Pillar` object ***is a*** (special case) of a `Swordsman` object
  - therefore, a reference to a `Pillar` object (superclass) can be converted to a `Swordsman` reference (subclass)

```
public class Pillar extends Swordsman {

    public static void main(String[] args) {
        Swordsman kyojuro = new Pillar("Kyojuro", 1000, "Fire");
    }
}
```

Converting between Subclass and Superclass (2)

- However, only methods of `Swordsman` (superclass) can be called
  - call to `getType()`, method of `Pillar` (subclass) will cause compile error

```
public class Pillar extends Swordsman {

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

compile error

OK

  - but, we can call the `toString()` method,
    which actually has been overridden in `Pillar`

Polymorphism (1)

- Moreover, `toString()` of `Pillar` (that overrides one in `Swordsman`) will be called
  - useful when we have many subclasses of `Swordsman` that have overridden the same method like `toString()`
  - each overridden `toString()` will be called with the *same call*, which we will do next

```
public class Pillar extends Swordsman {

    public static void main(String[] args) {
        Swordsman kyojuro = new Pillar("Kyojuro", 10000, "Fire");
        System.out.println(kyojuro);
    }
}
```

  - people call this OOP feature *polymorphism*
    - superclass variable calls the overridden method of each specific subclass

Polymorphism (2)

- A Successor is an exceptionally talented swordsman who is designated as the apprentice of one of the Pillars

```
public class Successor extends Swordsman {

    private String apprenticedTo;

    public Successor(String name, String apprenticedTo) {
        super(name);
        this.apprenticedTo = apprenticedTo;
    }

    @Override
    public String toString() {
        return "Successor " + getName() + " apprenticed to " + apprenticedTo +
               " has killed " + getNumDemonsKilled() + " demons";
    }
}
```

Polymorphism (3)

- Now, we can have an array of Swordsman objects
  - and each individual object can be printed accordingly

```
public class DemonSlayerCorps {

    public static void main(String[] args) {

        Swordsman[] swordsmanTroop = new Swordsman[3];
        swordsmanTroop[0] = new Swordsman("Tanjiro");
        swordsmanTroop[1] = new Pillar("Kyojuro", 10000, "Fire");
        swordsmanTroop[2] = new Successor("Kanao", "Shinobu");

        for (int i = 0; i < swordsmanTroop.length; i++) {
            System.out.println(swordsmanTroop[i]);
        }
    }
}
```

Thank you for your attention !

- In this lecture, you have learned:
  - to leverage inheritance and create subclasses
  - to learn relationship between a class and its superclass
  - to distinguish between overloading and overriding method
  - to use polymorphism to call overridden subclass method on an array of superclass objects

- Please continue to Lab 9 to complete Lab Tasks, and then solve
  - Exercise #9.1 - #9.4 and
  - CW1 #9.1, #9.2, #9.3