

# Inheritance



**Vũ Thị Hồng Nhạn**

([vthnhan@vnu.edu.vn](mailto:vthnhan@vnu.edu.vn))

Dept. of Software Engineering, UET

Vietnam National Univ., Hanoi

# Content

## ❖ Encapsulation

- i.e., a class contains **everything it needs** and **nothing more!**

## ❖ Polymorphism (multiple shapes or forms)

- i.e., **java objects** can have **multiple identifiers**, that way different objects can be grouped as if they are the **same type** under **certain condition**

## ❖ Inheritance

- Passing down traits or characteristics from a parent to their child like skin, hair color

# Example

## Building a bank account manager

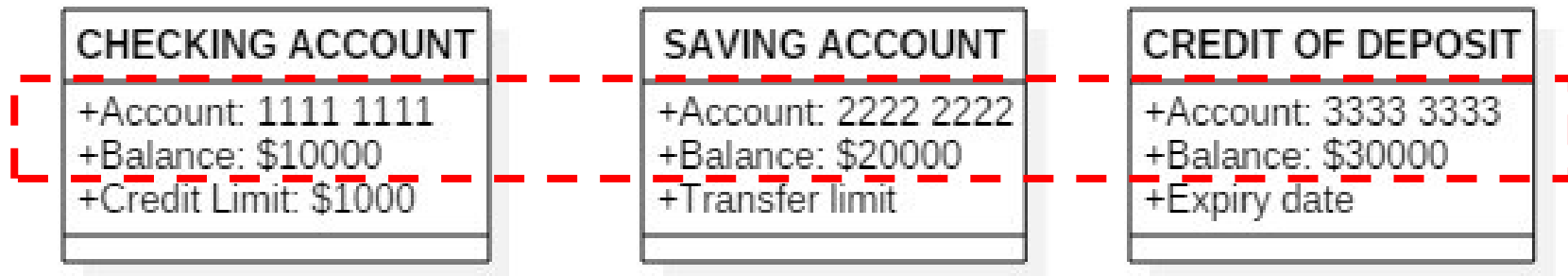
- ❖ Assume you have 3 different types of accounts



- ❖ They **all share similar information** like the **account number** and the **balance** in them
- ❖ But they also have **different attributes**

## Example...

# Building a bank account manager



## Example...

# Building a bank account manager

- ❖ Basically you can implement that in **one single class** in java

```
class BankAccount {  
    int      acctType;  
    String   acctNumber;  
    double   balance;  
    double   limit;  
    int      transfers;  
    Date     expiry;  
}
```

- ❖ ➔ that means, **everything** is included in that class!

## Example...

# Building a bank account manager

- ❖ Another way is to create a class for each account type
  - **each class** would only contain *the fields and methods* that make sense for that class

```
class Checking {
```

```
String acctNumber;  
double balance;  
double limit;
```

```
}
```

```
class Savings {
```

```
String acctNumber;  
double balance;  
int transfers;
```

```
}
```

```
class COD {
```

```
String acctNumber;  
double balance;  
Date expiry;
```

```
}
```

## Example...

### Problem caused by a change!

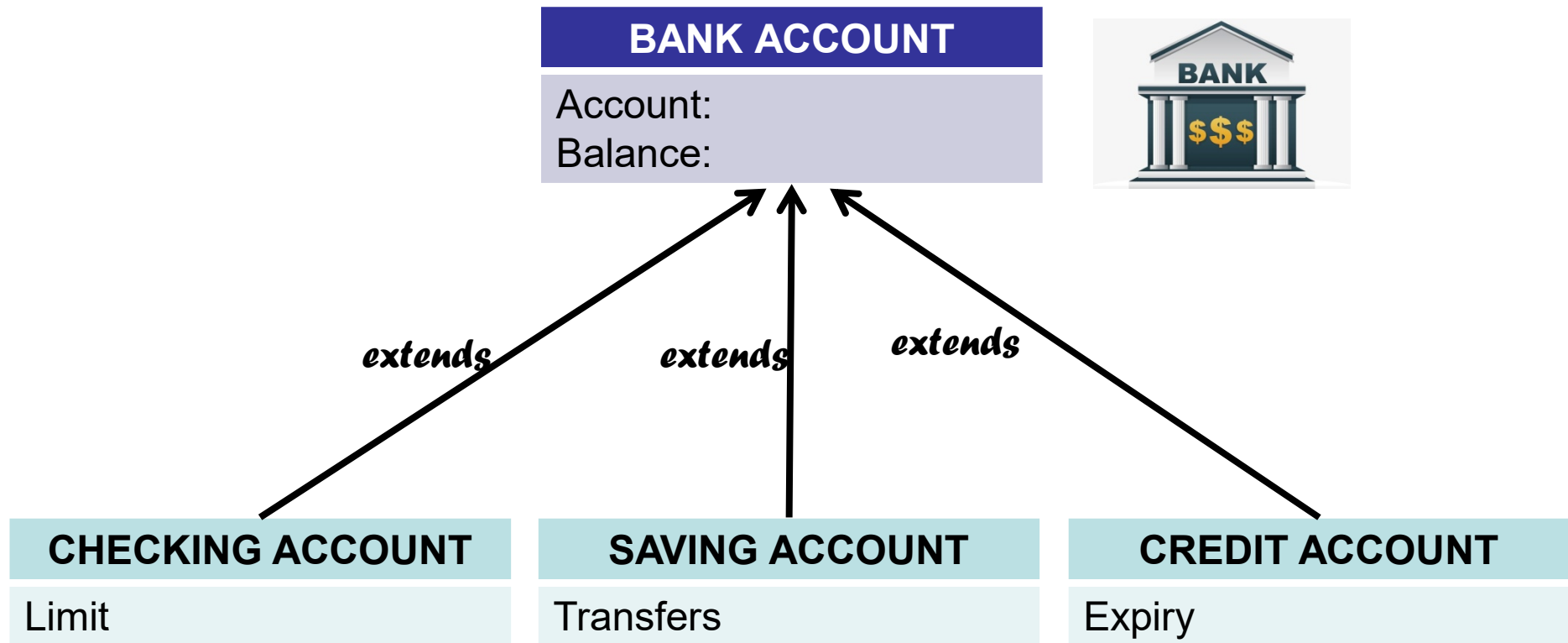
- ❖ If we decide to include **the bank code** → have to change that in **all three classes**

```
class Checking {  
    String acctNumber;  
    double balance;  
    int bankCode;  
    double limit;  
}  
  
class Savings {  
    String acctNumber;  
    double balance;  
    int bankCode;  
    int transfers;  
}  
  
class COD {  
    String acctNumber;  
    double balance;  
    int bankCode;  
    Date expiry;  
}
```

- ❖ This simple change could be **a nightmare** in a **production-sized project**!

## Example...

# Inheritance is the solution





# Inheritance

- ❖ First, start by creating the bank account class

```
class BankAccount{  
    String acctNumber;  
    double balance;  
}
```

- ❖ Next, create another class and points to **extend** that basic class by adding the phrase “**extend**” and then the class name

```
class Checking extends BankAccount{  
    double limit;  
}
```

# Inheritance...

```
class BankAccount{  
    String acctNumber;  
    double balance;  
}
```

*parent*

```
class Checking extends BankAccount{  
    double limit;  
}
```

*child*

# Inheritance...

```
class SavingAccount extends BankAccount{  
    int transfers;  
}
```

*child*

```
class COD extends BankAccount{  
    Date expiry;  
}
```

*child*

# Conclusion

- ❖ From that example, you can see that using **inheritance** has allowed us **to minimize** repeating any code
- ❖ While still having **the flexibility** and **the good design** of **separate account classes**



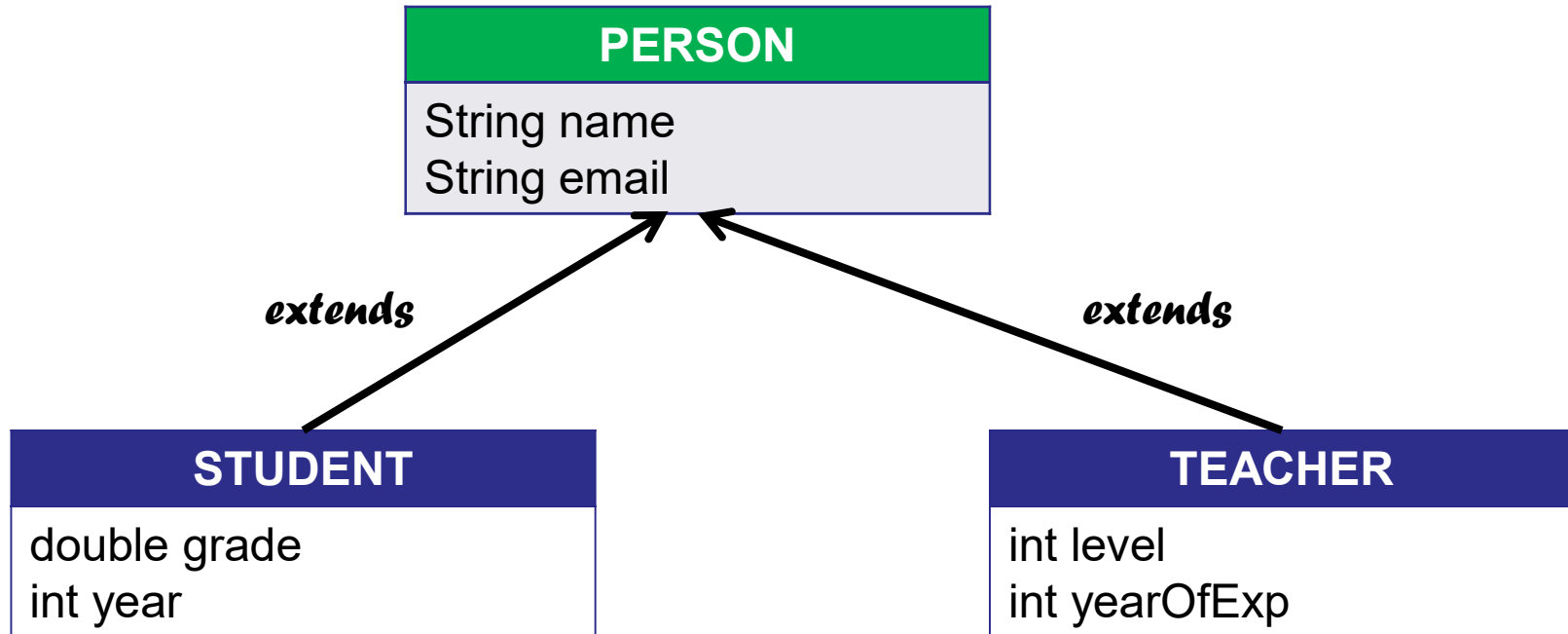
# **Polymorphism**



# what is polymorphism?

- ❖ **Polymorphism** is something that has **multiple** shares or **forms**
- ❖ In object-oriented world, **inheritance** allows **object** to become **polymorphic**
  - Because when an object extends another class, it not only becomes its **own type** but also the **type of its parent**

# Example

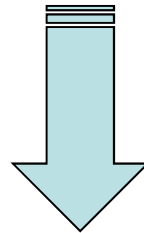
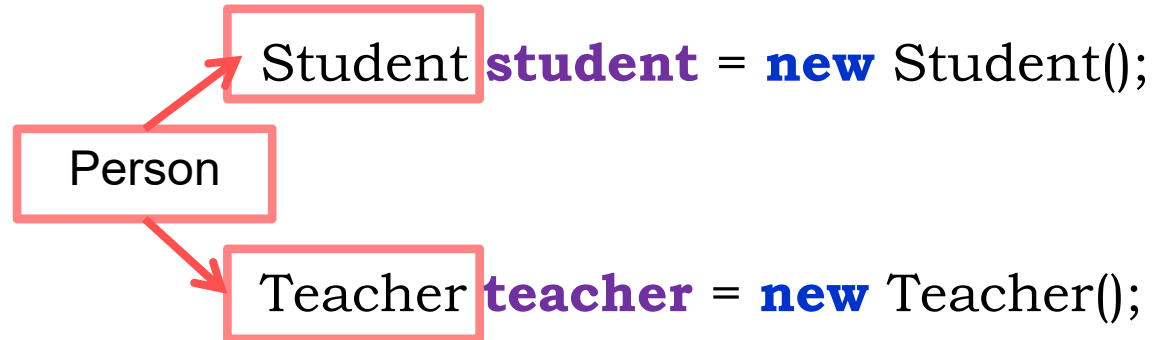


## Example...

- ❖ If you were to create an object of type student, you can start declaring it using the student type and initialize it using the student constructor
  - Student **student** = **new** Student();
- ❖ Do similar if you want to create an object of type teacher
  - Teacher **teacher** = **new** Teacher();



# Example...



```
Person student = new Student();
```

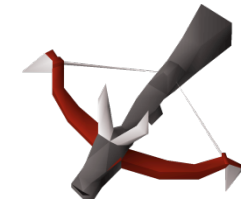
```
Person teacher = new Teacher();
```

# Bags & items example



**Coins**

int amount  
int weight



**Crossbow**

int power  
int weight



**RareItem**

int value  
int weight

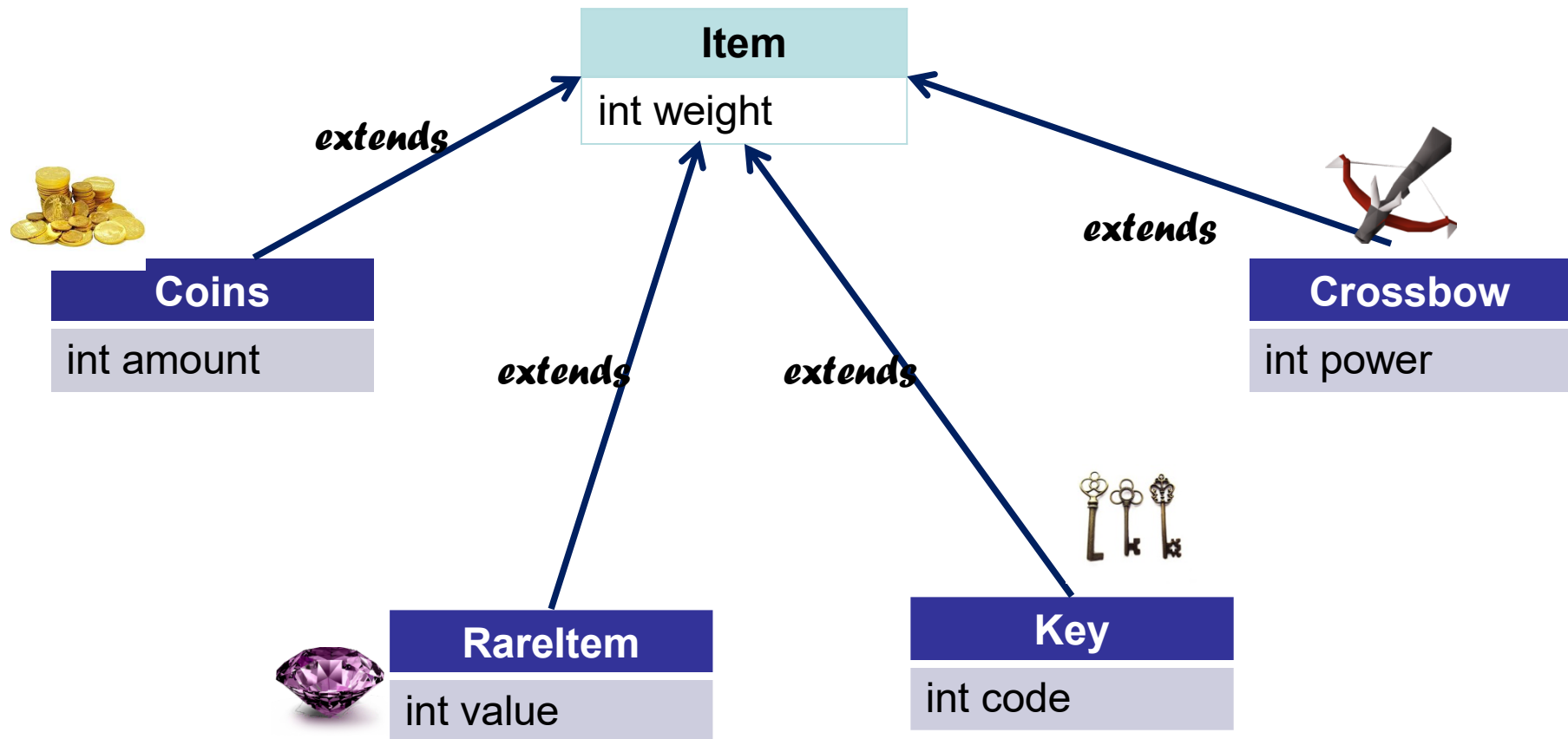


**Key**

int code  
int weight

# Bags & items example

## items



# Bags & items example

## Bags

```
public class Bag{  
    int currentWeight;  
    boolean canAddItem(Item item);  
}
```



# Bags & items example

## Bags...

```
boolean canAddItem(Item item){  
    if(currentWeight + item.weight >20)  
        return false;  
    else  
        return true;  
}
```

# Bags & items example...

```
public static void main(String[] args){  
    CrossBow crossbow = new Crossbow();  
    if(bag.canAddItem(crossbow)  
        bag.addItem(crossbow);// can add crossbow without being casted  
}
```



# **Overriding methods**



# Chess game example

- ❖ A good design in java will have a class for **each piece** of type



- They should inherit from the base class **Piece**
- ❖ **Why?**
    - Because according to the concept of polymorphism, you could represent the **chess-board** as a **2D array** of **Piece objects**,
    - and then each cell in the 2D array can **contain any of child** classes that **extend** the **Piece class**



# Chess game example...

## Other classes

- ❖ To store **2D array**, we need a class that represents the **Game** itself

```
class Game{
    Piece [][] board;
    Game(){
        board = new Piece[]8[8];
    }
}
```

- ❖ Finally, a simple class called **Position** that has nothing more than **a row value** and **a column value** to represent a specific slot on the board

```
class Position{
    int row; int column;
    Position(int row, int column){
        this.row=row; this.column=column; }
}
```

# Chess game example...

## Other classes...

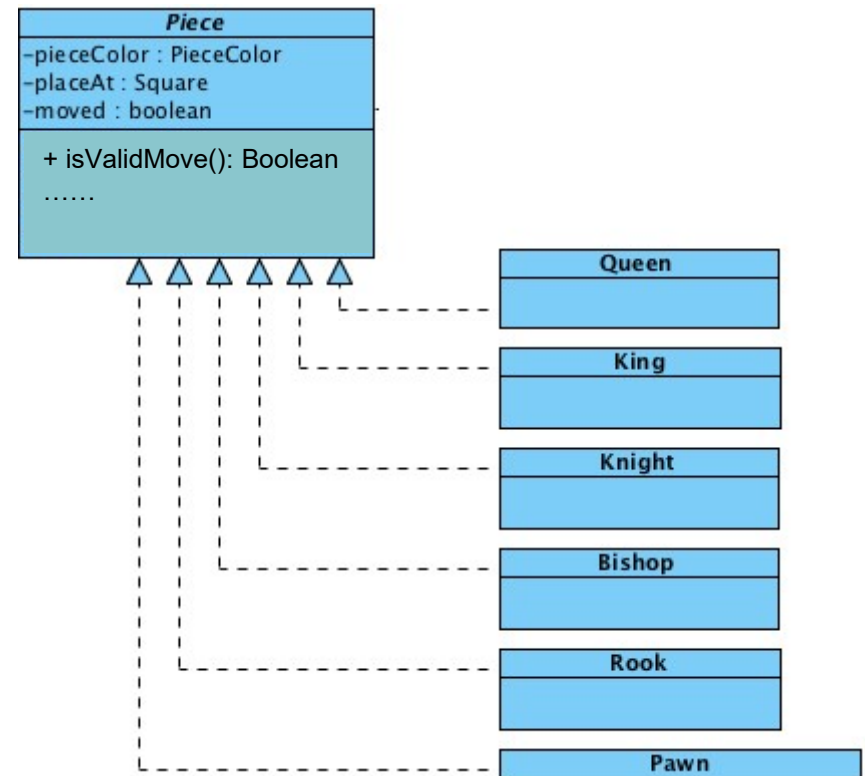
- ❖ That way, the **Piece** class can include a field variable of type **Position** that stores the current position of that piece on the board

```
class Piece{  
    Position position;  
  
    ....  
}
```

# Chess game example...

## Other classes...

- ❖ Now since **all piece types** inherit from **the same parent class Piece**, they will all **share** any methods declared in that class
  - e.g., it will be useful to have **a method** that checks if a **potential movement** of a piece is valid one
  - A good idea would be to include a method called **isValidMove()** inside the **Piece** class that take **a potential new position** and decides if that position is within the bounds of the chess board



# Chess game example...

## Other classes...

```
class Piece{
    Position position;

    boolean isValidMove(Position newPosition){
        if(newPosition.row > 0 && newPosition.column >0 && newPosition<8
        && newPosition.column <8)

            return true;

        else

            return false;
    }
}
```

# Chess game example...

## Other classes...

- ❖ Since each of **the child class** inherits from **that Piece class**, **each** will **automatically include this method**, which means you can call it from any of those classes directly

- ❖ E.g.,

```
Queen queen = new Queen();  
Position testPosition = new Position(3,3);  
if(queen.isValidMove(testPosition))  
    System.out.println("Yes, you can move there");  
else  
    System.out.println("Nope, cannot move there");
```

# Chess game example...

- ❖ Till now, we're just able to check the **validity** of the **movement of a piece** on the board
- ❖ **Each piece** type has a **different pattern** of allowed movements, which means that **each of those child classes** need to implement the **isValidMove()** method **differently**
- ❖ Luckily, java not only offers a way **to inherit** a method from a parent class but also **modify** it to build your own custom version of it

# Overriding

- ❖ When a class **extends** another class, all **public** methods declared in that parent class are **automatically included** in the child class without doing anything
- ❖ However, you are allowed to **override any** of those methods
  - Overriding basically means **re-declaring** them in **the child class** and then **re-defining** what they should do

# the chess example

- ❖ Assume we're implementing the `isValidMove()` method in the **Rook** class
  - the **Rook** class extends the **Piece** class that already includes a definition of the `isValidMove()` method
- ❖ The **Rook** class extends the **Piece** class that **already includes** a definition of the `isValidMove()` method
- ❖ Now, let's implement a **custom version** of that method inside the Rook class



# The chess example...

```
class Rook extends Piece{  
    boolean isValidMove(Position newPosition){  
        if(newPosition.column==this.position.column ||  
           newPosition.row==this.position.row)  
            return true;  
        else  
            return false;  
    }  
}
```

- ❖ Notice how **both** method declarations are **identical**, **except** that the implementation in the **child** class has **different code** customizing the validity check for the Rook piece

# The chess example...

- ❖ Basically it's checking that the **new position** of the **rook**
  - has **the same column** value as the current position (which means **it's trying to move up or down** )
  - or has **the same row** position which means it has moved sideways,
  - ← **both** valid movements for a Rook piece
- ❖ Remember that **this.position.column** and **this.position.row** are the local fields of the **Rook** class holding *the current position of that piece*

# The chess example...

- ❖ Can do the same for all the other piece types, e.g., the **Bishop** class would have **slightly different** implementation
- ❖ Since Bishop moves diagonally, we want to check that **#vertical steps** is equal to **#horizontal steps**.

```
class Bishop extends Piece{
    boolean isValidMove(Position newPosition){
        if(Math.abs(newPosition.column - this.position.column) ==
           Math.abs(newPosition.row - this.position.row))
            return true;
        else
            return false;
    }
}
```

# The chess example...

- ❖ Now let's try to override that method for the Queen class
  - **Queen** can move **diagonally** or in **straight lines**
  
- ❖ What can we do?
  1. In the Queen class, **override** the **isValidMove()** method
  2. First ,call the parents' **isValidMove()** to check for the boundaries
  3. Add **more code** to check the queen's specific move validity



**Super**



# Super?

- ❖ Note that once you **override** a method,
  - you basically **ignore everything** that was in the **parent class**
- ❖ It's possible **not to throw away** the **parent implementation**
  - **ADD** the **extra checks** for each child class individually
- ❖ To **re-use** the parent method in the child class by using the **“super”** keyword, followed by a dot and then the method name
  - **super.isValidMove(position);**

# Super...

- ❖ That means, in **each** of the child classes,
  - before you get to check the custom movement, you can check if `super.isValidMove(position)` has returned false
  - If so, then no need to do any more checks and immediately return false
  - otherwise, continue checking
- ❖ The **new** implementation for the **Rook class** will look like....

# example

```
class Rook extends Piece{  
    boolean isValidMove(Position newPosition)  
        if(!super.isValidMove(position)) // check for the board bounds  
            return false;  
        // check for the specific rock movement  
        if(newPosition.column==this.position.column &&  
            newPosition.row==this.position.row)  
            return true;  
        else  
            return false;  
}
```



# example...

```
class Rook extends Piece{  
    //default constructor  
    public Rook(){  
        super(); //this will call the parent's constructor  
        this.name="rook";  
    }  
}
```

**Note:** if a child's constructor **doesn't explicitly** call the parent's constructor using `super`, the java compiler **automatically inserts** a call to the *default constructor of the parent* class

If the parent class *doesn't have a default* constructor, you'll get a **compiler-time error**