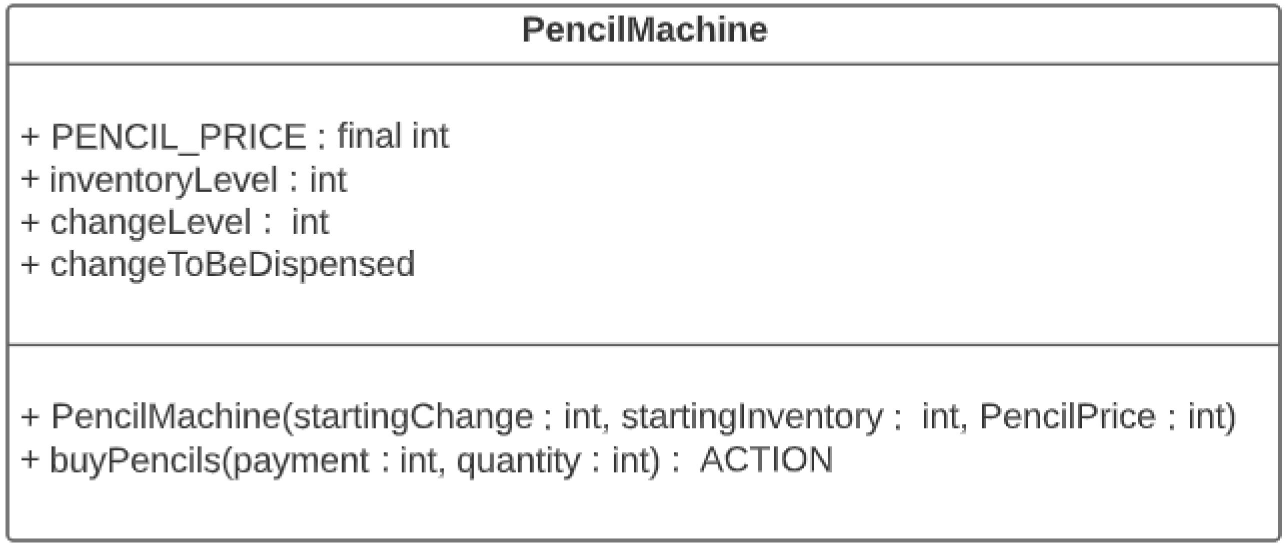
CSE 1325

Week of 10/10/2022

Instructor : Donna French

`toString()`



PencilMachine

+ PENCIL_PRICE : final int
+ inventoryLevel : int
+ changeLevel : int
+ changeToBeDispensed

+ PencilMachine(startingChange : int, startingInventory : int, PencilPrice : int)
+ buyPencils(payment : int, quantity : int) : ACTION

What happens if we try to print our object?

```
System.out.println(PM);
```

```
pencilmachinetest.PencilMachine@5ca881b5
```

What is that?

That is the hashcode value (reference) of the object.

`toString()`

Reference of the object.

Remember `this`?

If we add an instance method to the class

```
public void PrintMe()
{
   System.out.println(this);
}
```

and we

```
PM.PrintMe();
```

```
pencilmachinetest.PencilMachine@5ca881b5
```

toString()

We've seen a reference print somewhere else…

```
int a[] = {1,2,3};
System.out.println(Arrays.toString(a));

[1, 2, 3]
```

```
int a[] = {1,2,3};
System.out.println(a);

[I@7a81197d
```

We printed the array by using `toString` from `Arrays`

`toString()`

Every object you instantiate from a class you have created will have a default version of `toString()` built into it.

Inheritance at work

Similar to how every object has a constructor.

```
System.out.println(PM.toString());
System.out.println(PM);
```

```
pencilmachinetest.PencilMachine@5ca881b5
```

toString()

Remember method overloading??

We can overload the `toString()` instance method to do what WE want.

```
public String toString()
{
    return "Pencil Machine status : " +
           PENCIL_PRICE + " " + changeLevel + " " +
           inventoryLevel;
}
```

toString()

```
public String toString()
{
    return "Pencil Machine status : " +
           PENCIL_PRICE + " " + changeLevel + " " +
           inventoryLevel;
}

System.out.println(PM);

Pencil Machine status : 75 500 100
```

Converting our Pencil Machine to an Object

If we truly want to be as "OOP"ish as possible, then we need to follow the ways of the encapsulation.

We need to make our instance variables private.

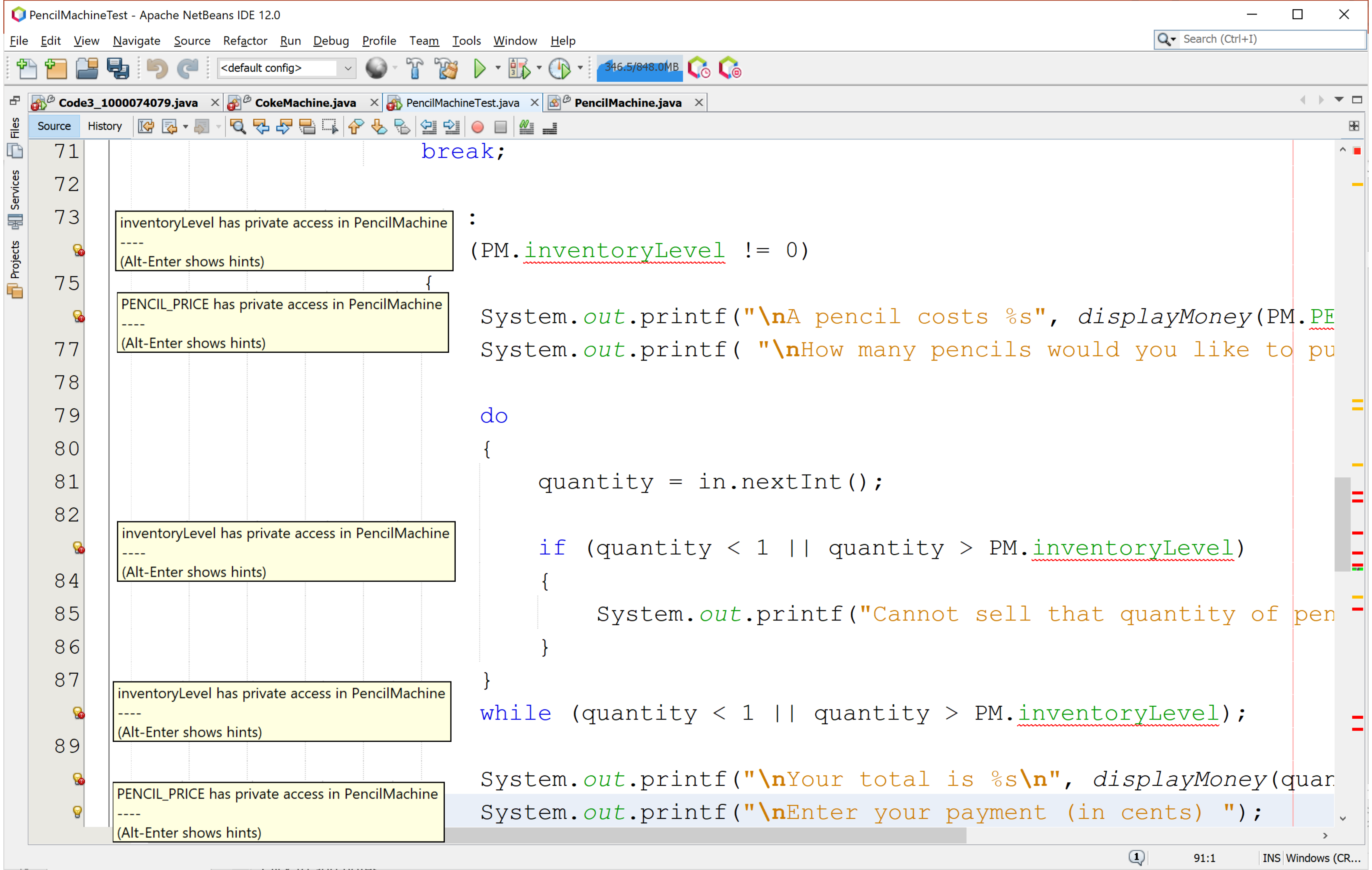
```
private final int PENCIL_PRICE;
private int inventoryLevel;
private int changeLevel;
private int changeToBeDispensed;
```

Converting our Pencil Machine to an Object

What does that do to our code?

Making the instance variables private in the class itself has no effect – everything in the class has access already.

But what happens in our `main()` /outside of our class?



```
                  break;

                      :
                      (PM.inventoryLevel != 0)
                  {
                       System.out.printf("\nA pencil costs %s", displayMoney(PM.PE
                       System.out.printf( "\nHow many pencils would you like to pu

                       do
                       {
                           quantity = in.nextInt();

                           if (quantity < 1 || quantity > PM.inventoryLevel)
                           {
                               System.out.printf("Cannot sell that quantity of pen
                           }
                       }
                       while (quantity < 1 || quantity > PM.inventoryLevel);

                       System.out.printf("\nYour total is %s\n", displayMoney(quar
                       System.out.printf("\nEnter your payment (in cents) ");
```

inventoryLevel has private access in PencilMachine

(Alt-Enter shows hints)

PENCIL_PRICE has private access in PencilMachine

(Alt-Enter shows hints)

inventoryLevel has private access in PencilMachine

(Alt-Enter shows hints)

inventoryLevel has private access in PencilMachine

(Alt-Enter shows hints)

PENCIL_PRICE has private access in PencilMachine

(Alt-Enter shows hints)

```
public int getPencilPrice()
{
    return PENCIL_PRICE;
}

public int getInventoryLevel()
{
    return inventoryLevel;
}

public int getChangeLevel()
{
    return changeLevel;
}

public int getChangeToBeDispensed()
{
    return changeToBeDispensed;
}
```

Converting our Pencil Machine to an Object

```
if (PM.inventoryLevel != 0)

if (PM.getInventoryLevel() != 0)

if (quantity < 1 || quantity > PM.inventoryLevel)

if (quantity < 1 || quantity > PM.getInventoryLevel())

System.out.printf("\nA pencil costs %s", displayMoney(PM.PENCIL_PRICE));

System.out.printf("\nA pencil costs %s", displayMoney(PM.getPencilPrice()));
```

Converting our Pencil Machine to an Object

```
while (quantity < 1 || quantity > PM.inventoryLevel);

while (quantity < 1 || quantity > PM.getInventoryLevel());

System.out.printf("\nYour total is %s\n",
     displayMoney(quantity * PM.PENCIL_PRICE));

System.out.printf("\nYour total is %s\n",
     displayMoney(quantity * PM.getPencilPrice()));
```

Converting our Pencil Machine to an Object

```
System.out.printf("Here's your pencils and your change of %s\n",
      displayMoney(PM.changeToBeDispensed));

System.out.printf("Here's your pencils and your change of %s\n",
      displayMoney(PM.getChangeToBeDispensed()));

System.out.printf("\nThe current inventory level is %d\n",
      PM.inventoryLevel);

System.out.printf("\nThe current inventory level is %d\n",
      PM.getInventoryLevel());

System.out.printf( "\nThe current change level is %s\n",
      displayMoney(PM.changeLevel));

System.out.printf( "\nThe current change level is %s\n",
      displayMoney(PM.getChangeLevel()));
```

enhanced for loop and 2D arrays

```
int TwoD[][] = new int [9][9];

for (int i = 0; i < 9; i++)
{
    for (int j = 0; j < 9; j++)
    {
        TwoD[i][j] = (i+1)*(j+1);
    }
}
```

enhanced for loop and 2D arrays

```
1    2    3    4    5    6    7    8    9
2    4    6    8   10   12   14   16   18
3    6    9   12   15   18   21   24   27
4    8   12   16   20   24   28   32   36
5   10   15   20   25   30   35   40   45
6   12   18   24   30   36   42   48   54
7   14   21   28   35   42   49   56   63
8   16   24   32   40   48   56   64   72
9   18   27   36   45   54   63   72   81
```

```
for (int i = 0; i < 9; i++)
{
    for (int j = 0; j < 9; j++)
    {
        System.out.printf("%5d", TwoD[i][j]);
    }
    System.out.println();
}
```

enhanced for loop and 2D arrays

```
for (int i = 0; i < 9; i++)
{
   System.out.printf("%s", Arrays.toString(TwoD[i]));
   System.out.printf("\n");
}
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[2, 4, 6, 8, 10, 12, 14, 16, 18]
[3, 6, 9, 12, 15, 18, 21, 24, 27]
[4, 8, 12, 16, 20, 24, 28, 32, 36]
[5, 10, 15, 20, 25, 30, 35, 40, 45]
[6, 12, 18, 24, 30, 36, 42, 48, 54]
[7, 14, 21, 28, 35, 42, 49, 56, 63]
[8, 16, 24, 32, 40, 48, 56, 64, 72]
[9, 18, 27, 36, 45, 54, 63, 72, 81]
```

enhanced for loop and 2D arrays

```
for (int[] row : TwoD)
{
    System.out.printf("%s\n", Arrays.toString(row));
}
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[2, 4, 6, 8, 10, 12, 14, 16, 18]
[3, 6, 9, 12, 15, 18, 21, 24, 27]
[4, 8, 12, 16, 20, 24, 28, 32, 36]
[5, 10, 15, 20, 25, 30, 35, 40, 45]
[6, 12, 18, 24, 30, 36, 42, 48, 54]
[7, 14, 21, 28, 35, 42, 49, 56, 63]
[8, 16, 24, 32, 40, 48, 56, 64, 72]
[9, 18, 27, 36, 45, 54, 63, 72, 81]
```

enhanced for loop and 2D arrays

```
for (int[] row : TwoD)
{
     for (int column : row)
     {
          System.out.printf("%5d", column);
     }
     System.out.println();
}
```

```
1    2    3    4    5    6    7    8    9
2    4    6    8   10   12   14   16   18
3    6    9   12   15   18   21   24   27
4    8   12   16   20   24   28   32   36
5   10   15   20   25   30   35   40   45
6   12   18   24   30   36   42   48   54
7   14   21   28   35   42   49   56   63
8   16   24   32   40   48   56   64   72
9   18   27   36   45   54   63   72   81
```

Using setters to validate input

```
public class Time1
{
    private int hour;
    private int minute;
    private int second;

    public void setTime(int hour, int minute, int second)
    {
        if (hour < 0 || hour >= 24 ||
            minute < 0 || minute >= 60 ||
            second < 0 || second >= 60)
        {
            throw new IllegalArgumentException("hour, minute and/or second
was out of range");
        }
        this.hour = hour;
        this.minute = minute;
        this.second = second;
    }
```

Validate inputs before applying them

This instantiates a new exception object of class `IllegalArgumentException`.

We pass the exception object's constructor the string `"hour, minute and/or second was out of range"`

```
public static void main(String[] args)
{
    Time1 time = new Time1();

    try
    {
        time.setTime(99,99,99);
    }
    catch (IllegalArgumentException e)
    {
        System.out.printf("Exception: %s\n\n", e.getMessage());
    }
}
```

Class `Time1` did not declare a constructor so the default version is used

Method `setTime()` is able to throw an exception so we put the call to `setTime()` in a `try` block.

`setTime()` throws an instantiation of the `IllegalArgumentException` so that is what we need to catch.

`getMessage` is a method in the `IllegalArgumentException` object that gets the message that was written to the object when it was thrown

```
public static void main(String[] args)
{
    BagOfCandy Bag1 = new BagOfCandy();

    Bag1.addCandy("Snickers");
    Bag1.addCandy("M&Ms");
    Bag1.addCandy("Candy Corn");

    Bag1.printCandy();

}
```

Instantiate a `BagOfCandy` using the default constructor

```
[Snickers, M&Ms, Candy Corn]
```

What prints if we call `printCandy()` BEFORE adding any candy?

```
[]
```

```
public class BagOfCandy
{
    private ArrayList<String>CandyList = new ArrayList<>();

    public void addCandy(String candy)
    {
        CandyList.add(candy);
    }
```

```
public void printCandy()
{
    System.out.println(CandyList);
}
```

What if we don't want certain types of candy in our Candy Bag?

We could add validation in `addCandy()` to not allow certain candies to be added to our bag

```
public void addCandy(String candy)
{
    if (candy == "Candy Corn")
    {
        throw new IllegalArgumentException("allergy alert");
    }
    CandyList.add(candy);
}
```

Now, when "Candy Corn" is added, that exception will be thrown and end the program if not caught.

```
Exception in thread "main" java.lang.IllegalArgumentException: allergy alert
	at slide.BagOfCandy.addCandy(BagOfCandy.java:19)
	at slide.Slide.main(Slide.java:18)
C:\Users\frenc\Documents\NetBeansProjects\Slide\nbproject\build-im         55:
The following error occurred while executing this line:
C:\Users\frenc\Documents\NetBeansProjects\Slide\nbproject\build-impl      61:
Java returned: 1
BUILD FAILED (total time: 2 seconds)
```

Notice that our text of "allergy alert" shows up in the exception output

```
public static void main(String[] args)
{
    BagOfCandy Bag1 = new BagOfCandy();
    try
    {
        Bag1.addCandy("Snickers");
        Bag1.addCandy("M&Ms");
        Bag1.addCandy("Candy Corn");
    }
    catch (IllegalArgumentException reject)
    {
        System.out.printf("Candy not added : %s\n", reject.getMessage());
    }

    Bag1.printCandy();

}
```

Object throws the exception and `main()` catches it.

```
public static void main(String[] args)
{
    BagOfCandy Bag1 = new BagOfCandy();
    try
    {
        Bag1.addCandy("Snickers");
        Bag1.addCandy("M&Ms");
        Bag1.addCandy("Candy Corn");
    }
    catch (Exception reject)
    {
        System.out.printf("Candy not added : %s\n", reject.getMessage());
    }

    Bag1.printCandy();

}
```

We can catch a more generic `Exception`

```
private ArrayList<String>CandyList = new ArrayList<>();
private ArrayList<String>AllergyAlert = new ArrayList<>();

public BagOfCandy()
{
    AllergyAlert.add("Candy Corn");
    AllergyAlert.add("Smarties");
    AllergyAlert.add("Tootsie Roll");
    AllergyAlert.add("Butterfinger");
}

public void addCandy(String candy)
{
    if (AllergyAlert.contains(candy))
    {
        String message = candy + " - allergy alert";
        throw new IllegalArgumentException(message);
    }
    CandyList.add(candy);
}
```

Created a constructor that takes no parameters so replaces the default one but still acts like/looks like the default.

more information in the message

```
public static void main(String[] args)
{
    BagOfCandy Bag1 = new BagOfCandy();
    try
    {
        Bag1.addCandy("Snickers");
        Bag1.addCandy("M&Ms");
        Bag1.addCandy("Candy Corn");
        Bag1.addCandy("Butterfinger");
    }
    catch (Exception reject)
    {
        System.out.printf("Candy not added : %s\n", reject.getMessage());
    }

    Bag1.printCandy();
}
```

```
Candy not added : Candy Corn - allergy alert
[Snickers, M&Ms]
```

```
public static void main(String[] args)
{
    BagOfCandy Bag1 = new BagOfCandy();
    try
    {
        Bag1.addCandy("Butterfinger");
        Bag1.addCandy("Snickers");
        Bag1.addCandy("M&Ms");
        Bag1.addCandy("Candy Corn");
    }
    catch (IllegalArgumentException reject)
    {
        System.out.printf("Candy not added : %s\n", reject.getMessage());
    }

    Bag1.printCandy();
}
```

```
Candy not added : Butterfinger - allergy alert
[]
```

Composition

We talked about composition – the "has a" relationship

In code, this can be represented by a class having references to objects of other classes as members.

Our `BagofCandy` is filled with Candy. Candy could be its own object instead of just a string containing the candy's name.

Composition

We can make a class `Candy`.

```
public class Candy
{
    private String name;
    private int calories;
    private String size;
```

Composition

```
public class Candy
{
    private String name;
    private int calories;
    private String size;

    Candy(String name, int calories, String size)
    {
        this.name = name;
        this.calories = calories;
        this.size = size;
    }
```

We want to create our own constructor so that we can set these values when the objects are instantiated.

Composition

Now our `BagOfCandy` has an `ArrayList` of `Candy` objects in it rather than `String`s containing the names of the candy.

```
private ArrayList<String>CandyList = new ArrayList<>();
```

changes to

```
private ArrayList<Candy>CandyList = new ArrayList<>();
```

Composition

```
private ArrayList<String>Candy = new ArrayList<>();

private ArrayList<Candy>CandyList = new ArrayList<>();

public void addCandy(String candy)
{
    Candy.add(candy);
}

public void addCandy(Candy candy)
{
    CandyList.add(candy);
}
```

Composition

Now we have to change our Allergy alert code

```
if (AllergyAlert.contains(candy))
{
   String message = candy + " - allergy alert";
   throw new IllegalArgumentException(message);
}
```

Before, `candy`, was just a `String` of the candy's name and we could compare that to the list of candy names in `AllergyAlert`.

Now `Candy` is an object. How do we get the name of the object?

We need to add a getter to our `Candy` class to retrieve the value stored in the private instance variable `name`.

```
public String getName()
{
    return name;
}
```

Now we can change our Allergy Alert code to retrieve the name to check against the list of candies.

```
public void addCandy(Candy candy)
{
    if (AllergyAlert.contains(candy.getName()))
    {
        String message = candy.getName() + " - allergy alert";
        throw new IllegalArgumentException(message);
    }
    CandyList.add(candy);
}
```

Is there a way to form our "message" without creating a new String variable?

```
public void addCandy(Candy candy)
{
    if (AllergyAlert.contains(candy.getName()))
    {
        String message = candy.getName() + " - allergy alert";
        throw new IllegalArgumentException(message);
    }
    CandyList.add(candy);
}
```

Remember `sprint()` in C?

```
sprintf(message, "%s - allergy alert", candyname);
```

Java has a similar method in the `String` class called `format`

```
String message = candy.getName() + " - allergy alert";
throw new IllegalArgumentException(message);
```

can be changed to

```
throw new IllegalArgumentException(String.format("%s - allergy alert",
                                                   candy.getName()));
```

`String.format` returns the `String` `"Butterfinger - allergy alert"`

How do we print the candy in our bag?

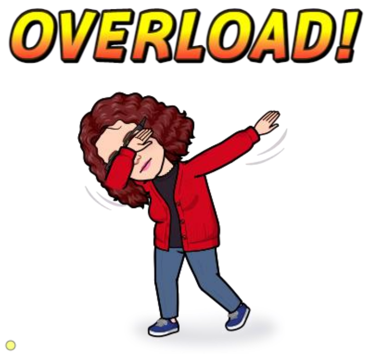
Before, we just printed the ArrayList

```
public void printCandy()
{
    System.out.println(CandyList);
}
```

Now we need to print the object

```
public void WhatsInMyBag()
{
    for (Candy it : CandyList)
        System.out.println(it.getName());
}
```

Is there another way to do this that is more flexible and useful?



toString()

```
public String toString()
{
    return "Name:  " + name +
           "\tCalories:  " + calories +
           "\tSize:  " + size;
}

public void WhatsInMyBag()
{
    for (Candy it : CandyList)
        System.out.println(it);
}
```

No overhead of calling getter methods to retrieve private data. `toString()` is an instance method in `Candy`; therefore, has access.

```java
Candy C1 = new Candy("Butterfinger", 100, "fun size");
Candy C2 = new Candy("Snickers", 100, "mini");
Candy C3 = new Candy("M&Ms", 100, "snack size");
Candy C4 = new Candy("Candy Corn", 123, "handful");

BagOfCandy Bag1 = new BagOfCandy();
try
{
    Bag1.addCandy(C2);
    Bag1.addCandy(C3);
    Bag1.addCandy(C1);
    Bag1.addCandy(C4);
}
catch (IllegalArgumentException reject)
{
    System.out.printf("Candy not added : %s\n", reject.getMessage());
}
```

When we encountered a "bad" candy, our try-catch set up did not allow the rest of the candies to be added.

```
Candy C1 = new Candy("Butterfinger", 100, "fun size");
Candy C2 = new Candy("Snickers", 100, "mini");
Candy C3 = new Candy("M&Ms", 100, "snack size");
Candy C4 = new Candy("Candy Corn", 123, "handful");

BagOfCandy Bag1 = new BagOfCandy();
try
{
    Bag1.addCandy(C2);
    Bag1.addCandy(C3);
    Bag1.addCandy(C1);
    Bag1.addCandy(C4);
}
catch (IllegalArgumentException reject)
{
    System.out.printf("Candy not added : %s\n", reject.getMessage());
}
```

We could add multiple try-catch blocks, but that would mean repeating a lot of code and is not sustainable as we add more candy.

```
ArrayList<Candy>AllCandy =  new ArrayList<>();
AllCandy.add(new Candy("Butterfinger", 100, "fun size"));
AllCandy.add(new Candy("Snickers", 100, "mini"));
AllCandy.add(new Candy("M&Ms", 100, "snack size"));
AllCandy.add(new Candy("Candy Corn", 123, "handful"));

BagOfCandy Bag1 = new BagOfCandy();

for (Candy it : AllCandy)
{
    try
    {
        Bag1.addCandy(it);
    }
    catch (IllegalArgumentException reject)
    {
        System.out.printf("Candy not added : %s\n", reject.getMessage());
    }
}
```

```
Candy not added : Butterfinger - allergy alert
Candy not added : Candy Corn - allergy alert
Name: Snickers    Calories: 100     Size: mini
Name: M&Ms  Calories: 100     Size: snack size
```

Garbage Collection

Memory leaks are common in other languages like C and C++

memory is not automatically reclaimed in those languages

Memory leaks are less likely in Java

some can still happen in subtle ways

Resource leaks other than memory leaks can also occur.

An app may open a file on disk to modify its contents.

If the app does not close the file, it must terminate before any other app can use the file.

Garbage Collection

Java's Garbage Collector is always running in the background.

Once it determines that an object is unreachable, it will destroy the object and return the memory to heap.

This is why Java does not need `free` like C does or `delete` like C++

Garbage Collection

```
ArrayList<Integer> Hello =  new ArrayList<>();
System.out.println(Hello);
Hello = null;
System.out.println(Hello);
Hello.add(1);
```

```
[]
null
Exception in thread "main" java.lang.NullPointerException
	at garbagecollector.GarbageCollector.main(GarbageCollector.java:16)
C:\Users\frenc\Documents\NetBeansProjects\GarbageCollector\nbproject\build-impl.xml:1355: The following error occurred while executing this line:
C:\Users\frenc\Documents\NetBeansProjects\GarbageCollector\nbproject\build-impl.xml:961: Java returned: 1
BUILD FAILED (total time: 1 second)
```

`static` Class members

Sometimes we need only one copy of a particular variable to be *shared* by all objects of a class.

A `static` field – called a class variable – is used in these cases.

A `static` variable represents classwide information—all objects of the class share the same piece of data.

The declaration of a `static` variable begins with the keyword `static`.

`static` Class members

Use a `static` variable when all objects of a class must use the same copy of the variable.

For example, you can have the class itself keep track of how many objects have been instantiated from it.

```
public Time1()
{
     counter++;
}
```

`Time1` was using the default constructor so we just created our own constructor with no parameters and add the `counter++`

`static` Class members

```
public static void main(String[] args)
{
    Time1 timeA = new Time1();
    Time1 timeB = new Time1();
    Time1 timeC = new Time1();

    System.out.printf("We have created %d objects\n", timeA.howMany());
    System.out.printf("We have created %d objects\n", timeB.howMany());
    System.out.printf("We have created %d objects\n", timeC.howMany());

    Time1 timeD = new Time1();
    System.out.printf("We have created %d objects\n", timeD.howMany());
}
```

```
We have created 3 objects
We have created 3 objects
We have created 3 objects
We have created 4 objects
```

Rubber Duck Debugging

https://en.wikipedia.org/wiki/Rubber_duck_debugging

In software engineering, rubber duck debugging is a method of debugging code.

The name is a reference to a story in the book "The Pragmatic Programmer" in which a programmer would carry around a rubber duck and debug their code by forcing themselves to explain it, line-by-line, to the duck.

Many other terms exist for this technique, often involving different (usually) inanimate objects (teddy bear) or pets such as a dog or a cat.

Rubber Duck Debugging

Many programmers have had the experience of explaining a problem to someone else, possibly even to someone who knows nothing about programming, and then hitting upon the solution in the process of explaining the problem.

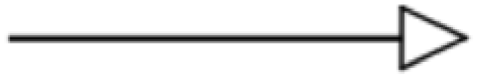
In describing what the code is supposed to do and observing what it actually does, any incongruity between these two becomes apparent.

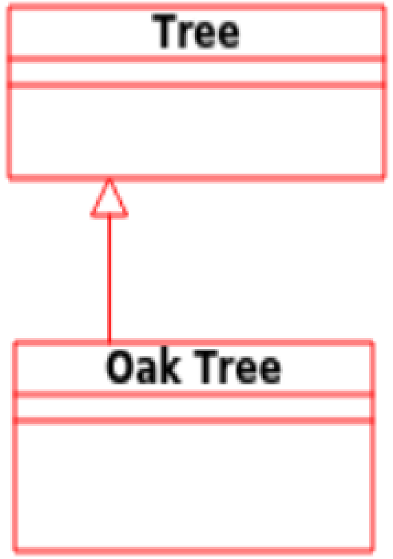
More generally, teaching a subject forces its evaluation from different perspectives and can provide a deeper understanding.

By using an inanimate object, the programmer can try to accomplish this without having to interrupt anyone else.

Object Relationships

Inheritance





- Represents the "is a" relationship
- Shows the relationship between a superclass/base class/parent and a derived/subclass/child.
- Arrow is on the side of the superclass/parent

Inheritance

Inheritance involves creating new objects by directly acquiring the attributes and behaviors of other objects and then extending or specializing them.

Inheritance is everywhere in real life.

Most living things (including you) inherited traits from parents.

Even non living things can inherit traits from their predecessors.

Inheritance

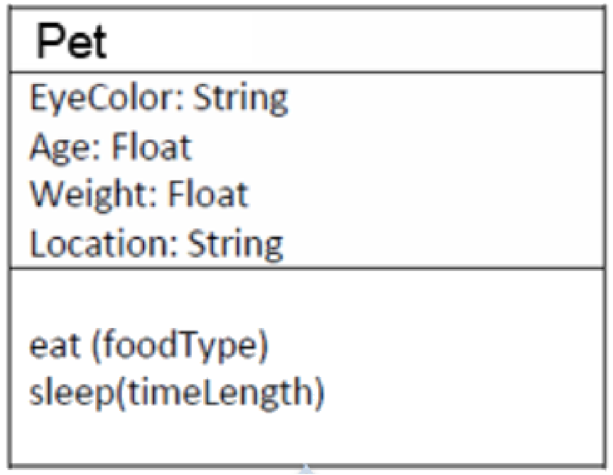
When Apple decides to create the next generation of iPhone, it does not start from scratch when creating a new phone.

They start with what they already know about the current version of the iPhone and build upon that.

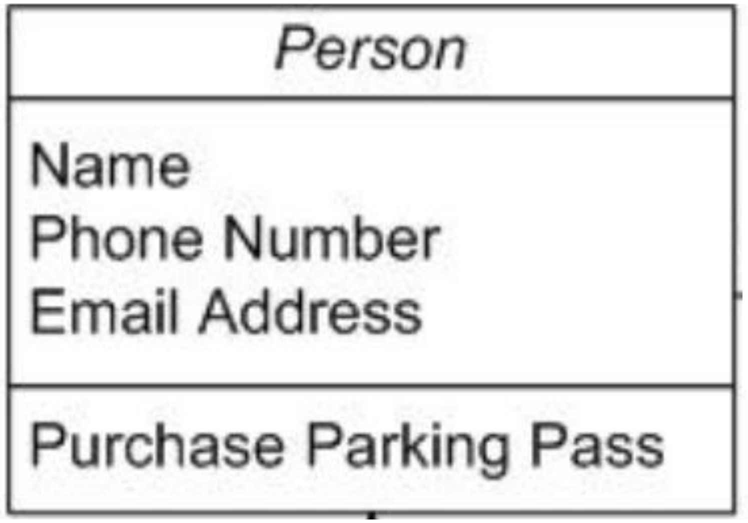
Most new version of electronics build upon the previous version. Not only does this lead to less work to create a new version but it also allows for backward compatibility.

Inheritance can save time during program development by basing new classes on existing proven and debugged high quality software.

Inheritance increases the likelihood that a system will be implemented and maintained effectively.


Pet
EyeColor: String
Age: Float
Weight: Float
Location: String
eat (foodType)
sleep(timeLength)

has a


Person
Name
Phone Number
Email Address
Purchase Parking Pass

is a

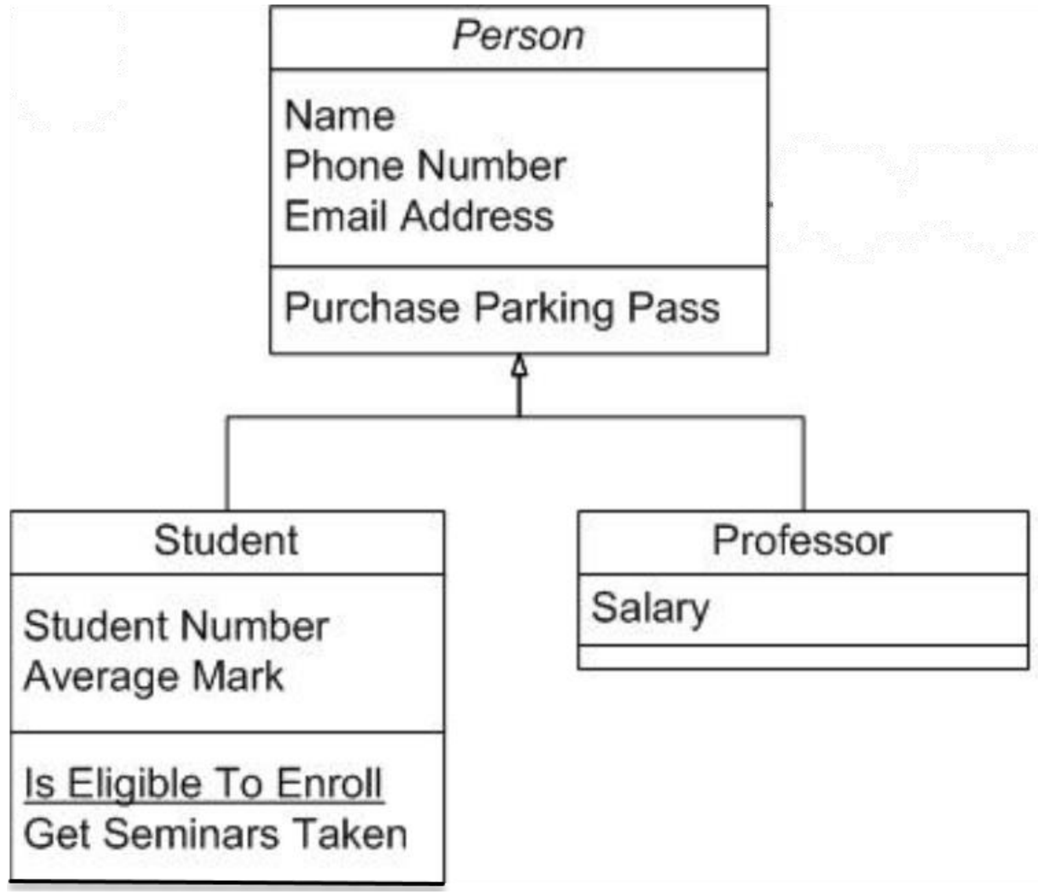
Inheritance

So does a student have a name, phone number and email address? Can a student purchase a parking pass?

Does a professor have a name, phone number and email address? Can a professor purchase a parking pass?

A professor has a salary – does every person have a salary?

A student has a student number – does every person have a student number?

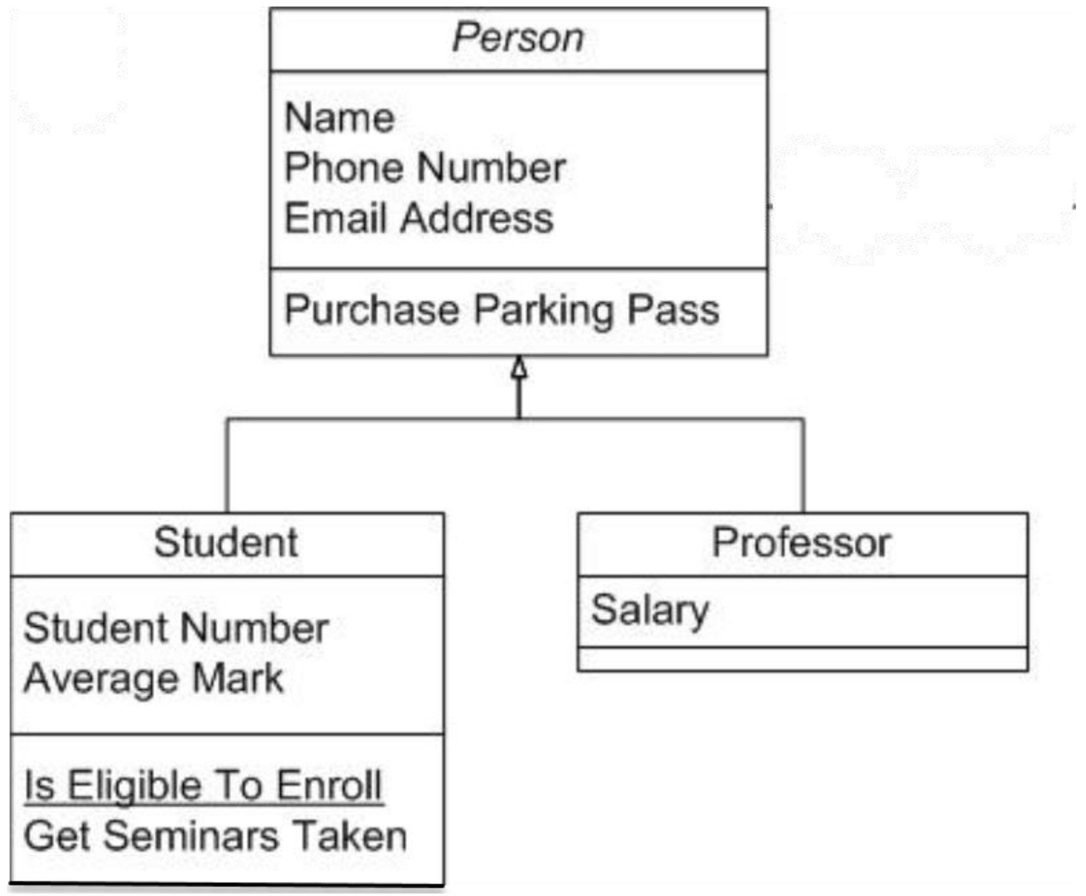


Inheritance

Rather than include name, phone number, email address and the ability to purchase a parking pass in our Student and Professor classes, we allow Student and Professor to inherit those attributes/abilities from Person.

This reduces the complexity of the Student and Professor class by making them contain less.

This also allows us to make changes to the Person class without directly changing Student and Professor.



Inheritance

A form of software reuse in which you create a class that absorbs an existing class's data and behaviors and enhances them with new capabilities.

- You can designate that a new class should inherit the members of an existing class.
- This existing class is called the superclass, and the new class is referred to as the subclass.

Inheritance

Existing class is the **superclass** and new class is the **subclass**.

A subclass can be a superclass of future subclasses.

A subclass can add its own fields and methods.

A subclass is more specific than its superclass and represents a more specialized group of objects.

The subclass exhibits the behaviors of its superclass and can add behaviors that are specific to the subclass.

This is why inheritance is sometimes referred to as specialization.

Inheritance

The **direct superclass** is the superclass from which the subclass explicitly inherits.

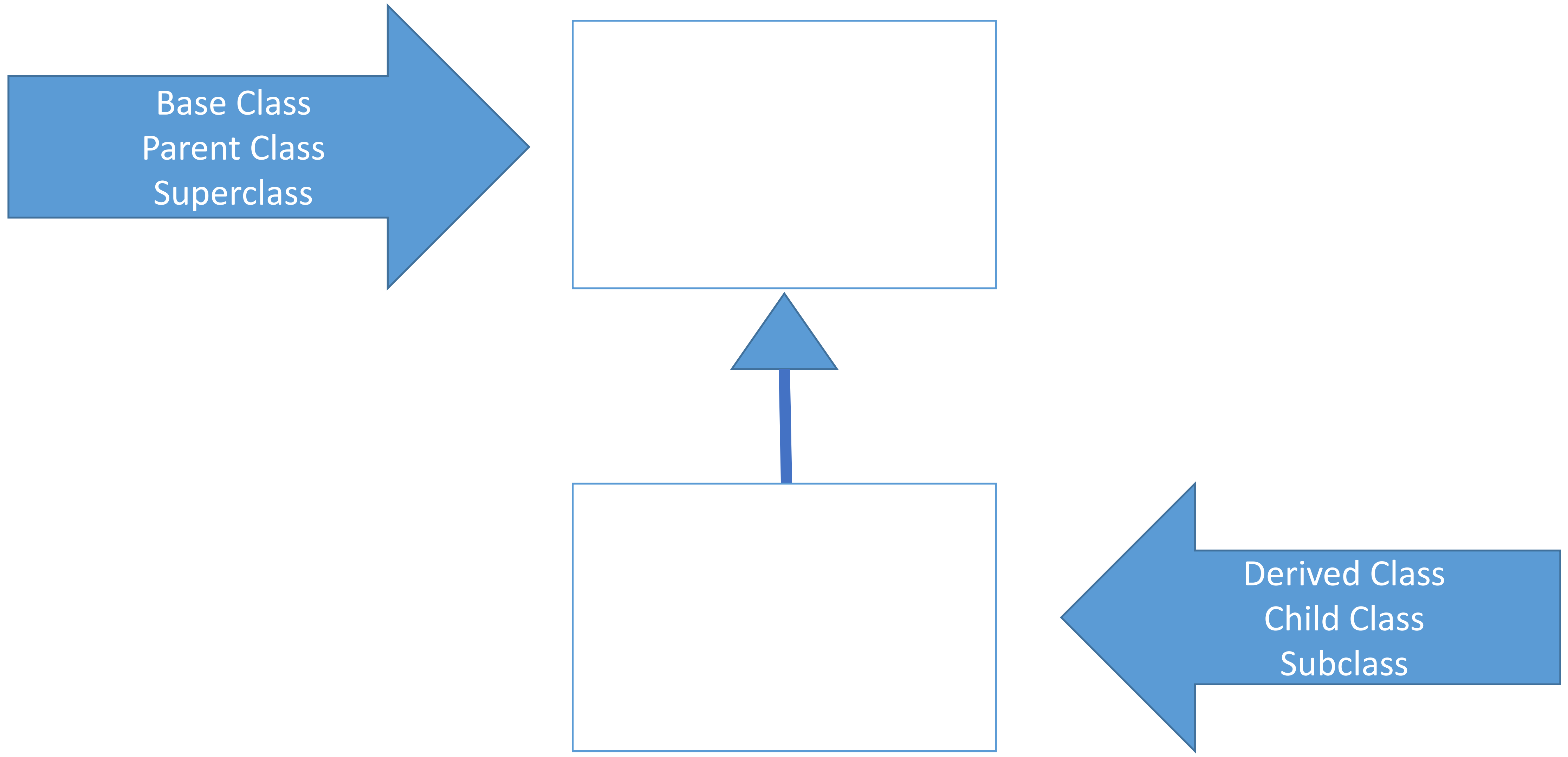
An **indirect superclass** is any class above the direct superclass in the class hierarchy.

The Java class hierarchy begins with class `Object` (in package `java.lang`)

Every class in Java directly or indirectly extends (or “inherits from”) `Object`.

Java supports only single inheritance, in which each class is derived from exactly one direct superclass.

Inheritance

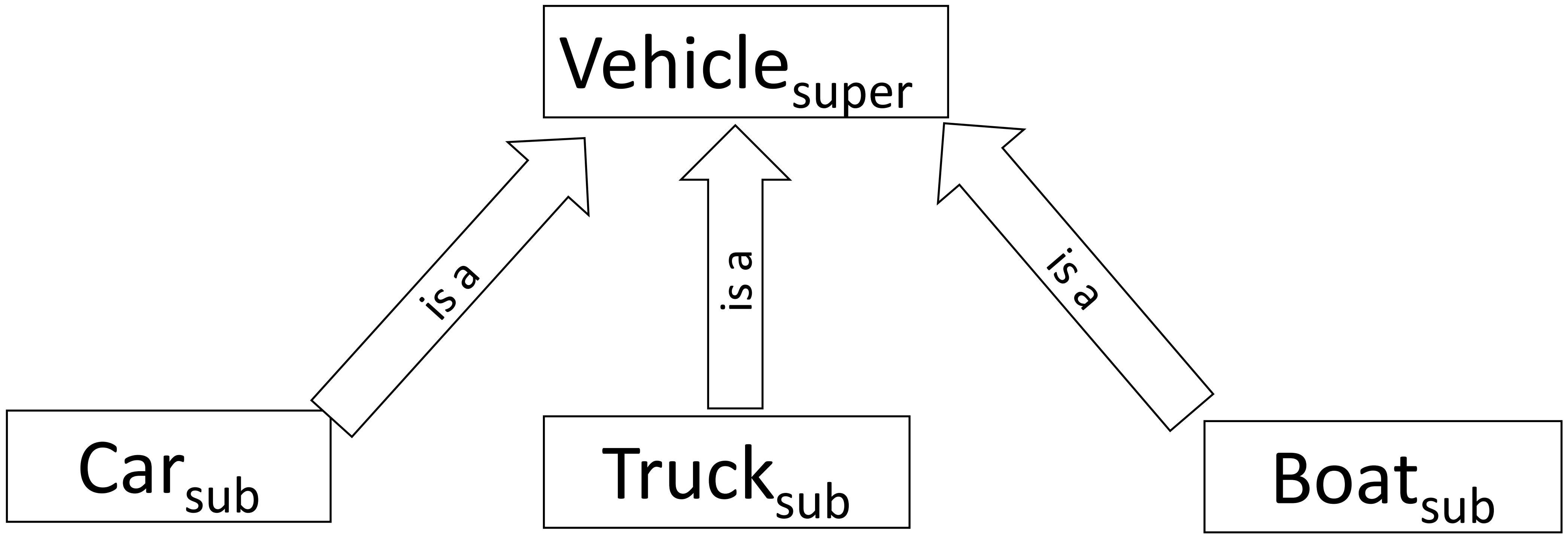


Inheritance

Superclasses tend to be *more general* and subclasses tend to be *more specific*.

Superclass	**Subclass**
Student	GraduateStudent, UnderGraduateStudent
Shape	Circle, Triangle, Rectangle, Sphere, Cube
Loan	CarLoan, HomeImprovementLoan, StudentLoan
Employee	Faculty, Staff
Account	CheckingAccount, SavingsAccount

superclass objects are not objects of their subclasses



every object of a subclass is also an object of that subclass’s superclass.

Inheritance

Because every subclass object *is an* object of its superclass and one superclass can have *many* subclasses, the set of objects represented by a superclass typically is *larger* than the set of objects represented by any of its subclasses.

Superclass `Vehicle` represents all vehicles including cars, boats, trucks, airplanes and bicycles.

Subclass `Car` represents a smaller, more specific subset of all vehicles.

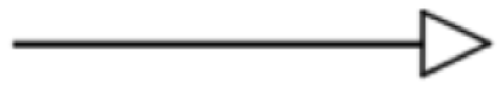
Superclass `Pet` represents all animals kept as pets; whereas, subclass `Cat` is a specific subset of `Pet`.

Inheritance

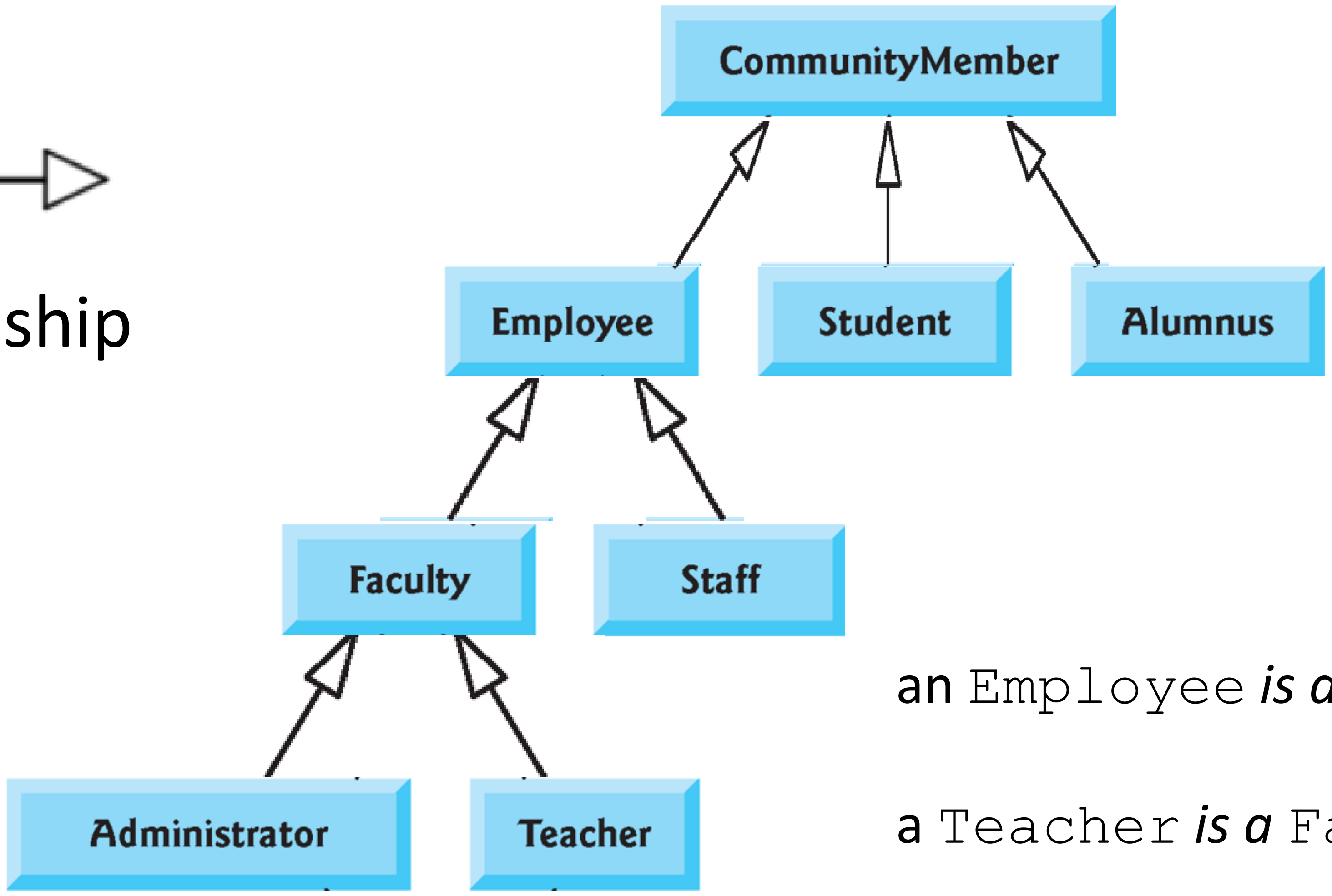
Inheritance relationships form class hierarchies.

- A superclass exists in a hierarchical relationship with its subclasses.
- Although classes can exist independently, once they are associated with an inheritance relationships, they become related to other classes.
- A class becomes either a superclass—supplying instance variables and methods to other classes, or a subclass—inheriting its instance variables and methods from other classes, or *both*.

Inheritance Hierarchy



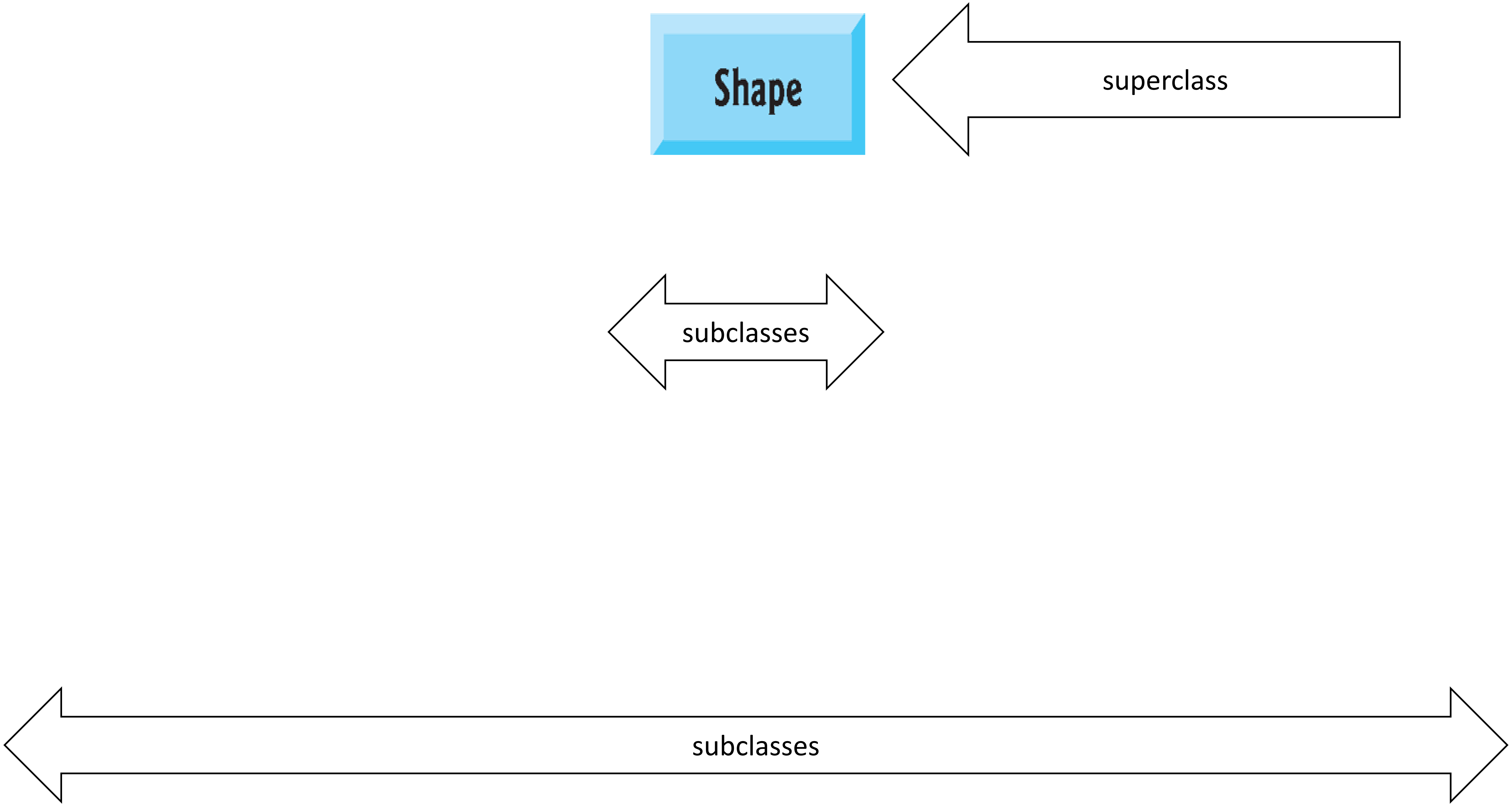
is-a relationship



an `Employee` *is a* `CommunityMember`

a `Teacher` *is a* `Faculty` member

`CommunityMember` is the direct superclass of `Employee`, `Student` and `Alumnus` and is an indirect superclass of all the other classes in the diagram.


Shape
superclass
subclasses
subclasses

Inheritance

Not every class relationship is an inheritance relationship.

Has-a relationship

Create classes by composition of existing classes

Our `BagOfCandy` contained `Candy`.

`BagOfCandy` is a `Candy`? ✗

`BagOfCandy` has a `Candy`. ✓

Inheritance

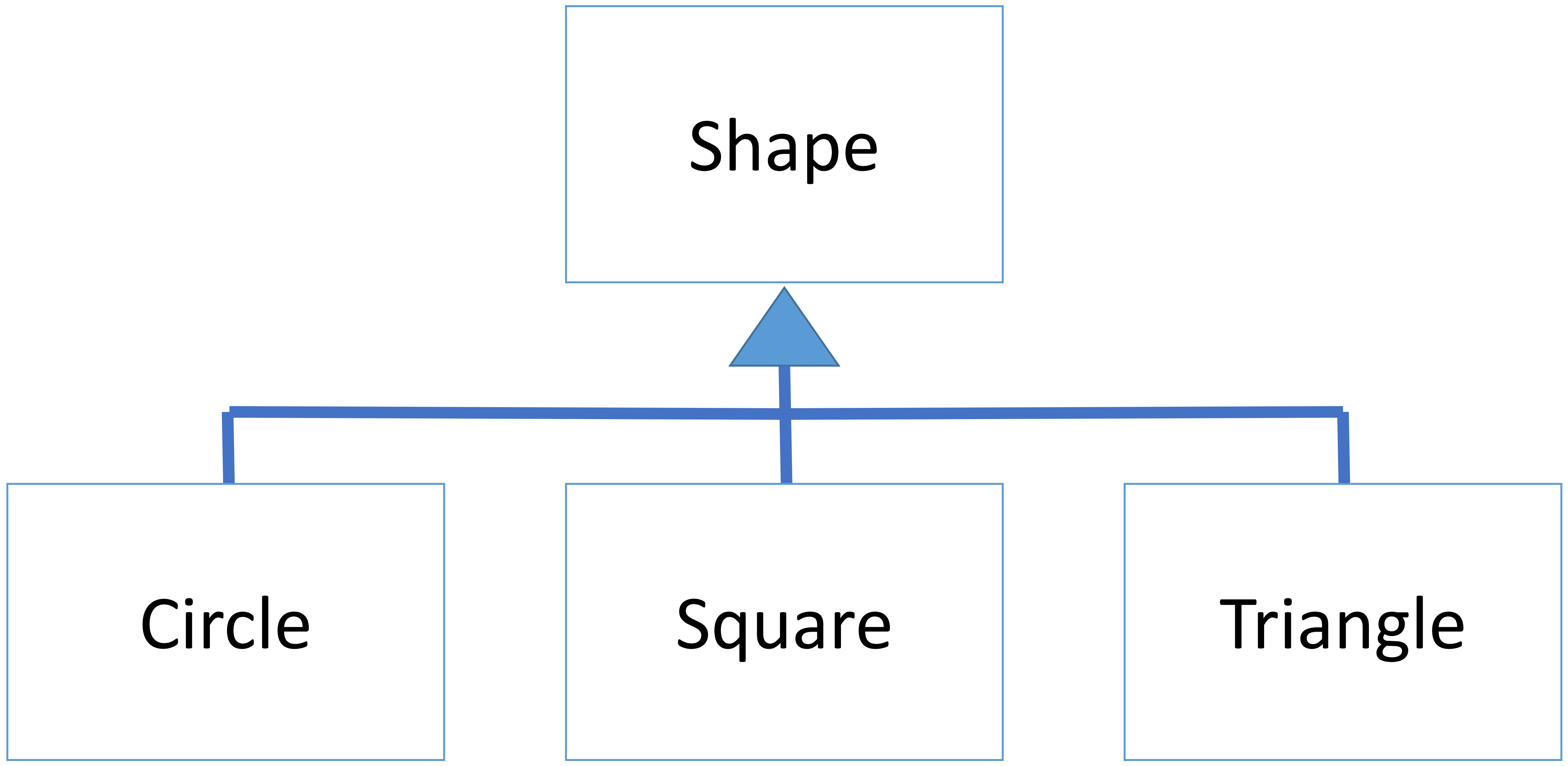
Inheritance Issue

A subclass can inherit methods that it does not need or should not have.

Even when a superclass method is appropriate for a subclass, that subclass often needs a customized version of the method.

The subclass can override (redefine) the superclass method with an appropriate implementation.

Inheritance


Shape
Circle
Square
Triangle