Advanced Programming CSE 201

Lec 1: Introduction

Dr. Koteswar Rao J
koteswar@iiitd.ac.in

About Me

Education: B.Tech (IIT Roorkee), PhD (NTU, Singapore)

Experience: Lenskart, ADSC, Graphic Era, IIITD

Research Interests: Computer Vision & Multimedia

Teaching Interests: Computer Vision, Image Processing, Programming, Machine Learning

Java Usage

- It’s a purely OOP (Object Oriented Programming) language.
- It deals with object variables.
- It’s made up of just classes (datatypes).
- Even main() needs to be in a class.
- It’s both compiled and interpreted language.
  - Source Code is compiled (translation: all at once)
  - Byte Code is interpreted (translation: line-by-line)

Compilation & Execution (or Running)

For compiling a source code

javac FileName.java

The above step will create as many bytecodes as number of classes present in the Filename.java

A bytecode has same name as the class and will have .class extension

We run the class that has the main() function. If that class is named as Main, we need to do the following:

java Main

Java Usage (Contd..)

```
class{
main(){
All Code
}}
```

```
class{
main(){
Non-repeating code
}
function(){
Repeating code
}....
}
```

```
class{
main(){
Necessary code
}}
class2{
related_functions(){
Unnecessary/Confidential code
}....
}....
```

To avoid Repetition

For Simplicity

Java Usage (Contd..)

- A code also has variables
- Keep related variables together
- Don’t repeat variables
- For example, real and imaginary parts of complex numbers.

For 100 complex numbers, we can easily end up with 200 variables.

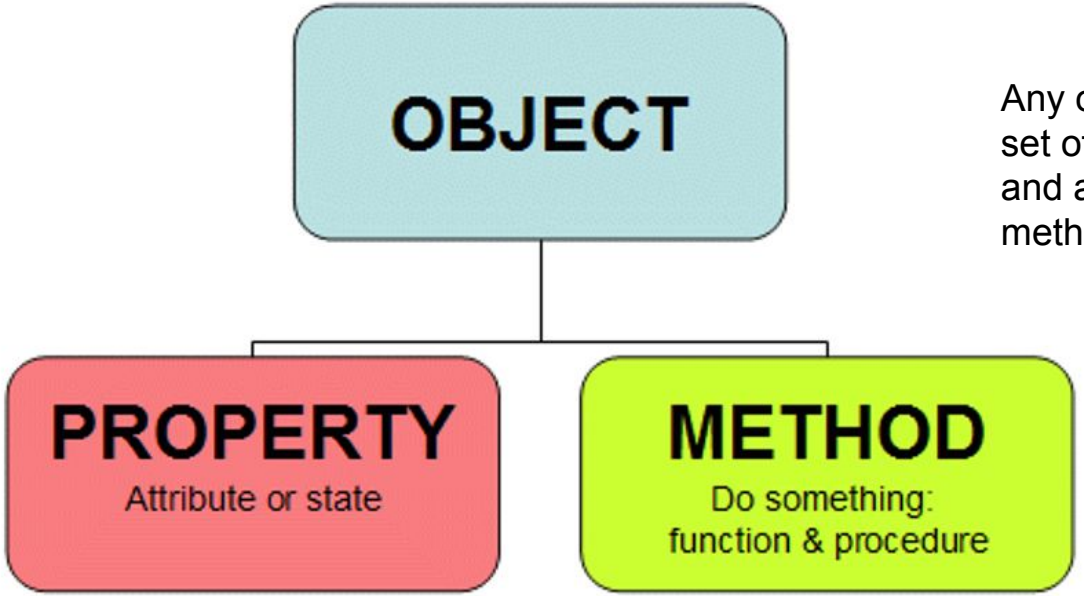
Instead, if we have a class for complex numbers, we will need only 102 variables: 100 to represent complex numbers and 2 for real and imaginary components.

Java Usage (Contd...)

Google Chrome: 4.5 Million lines
Google: 2 Billion lines

It helps you organize the code, which becomes essential to manage large codes

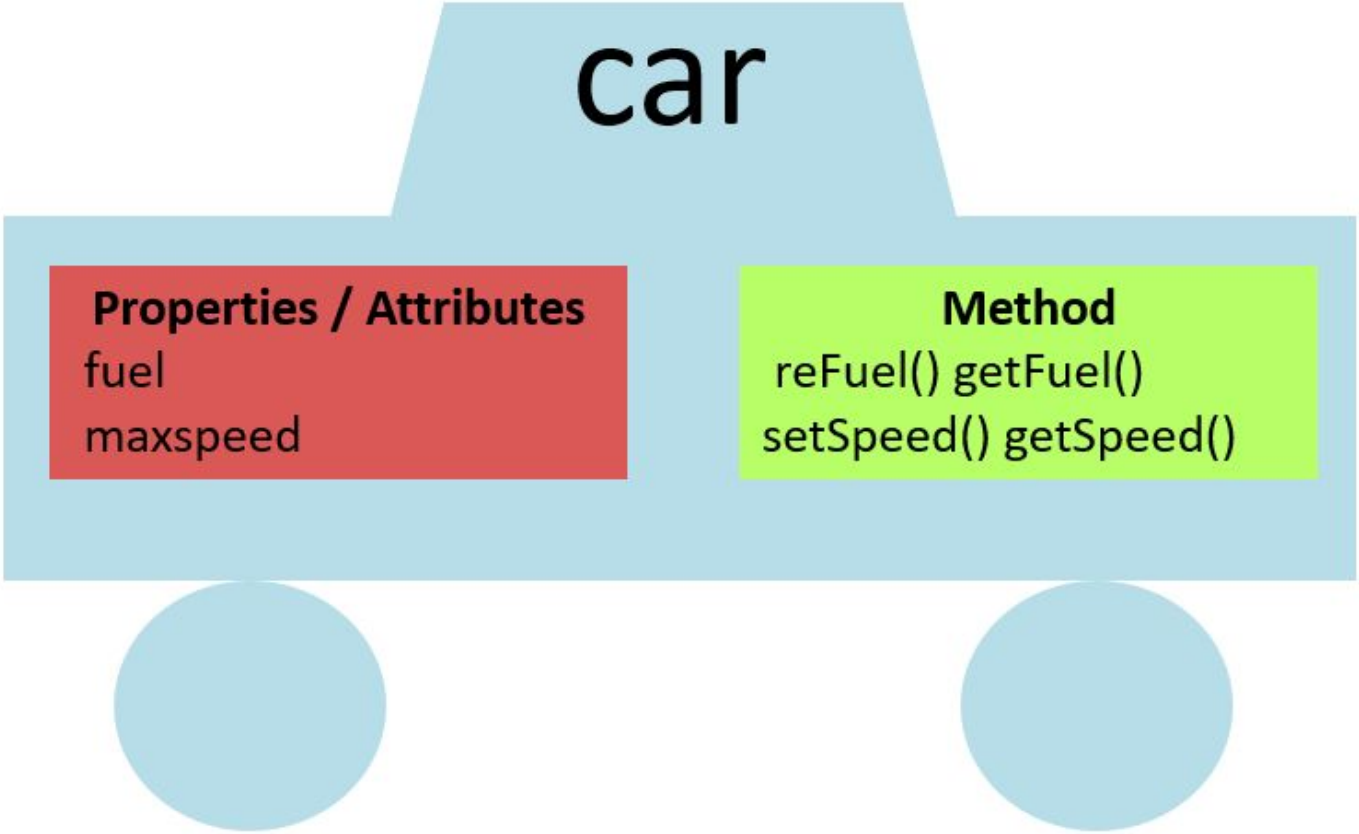
What is OOP?



Any object has a set of attributes and and a set of methods.

It is a programming paradigm based on the concept of "objects", which may contain data in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods
(Wikipedia)

class



A class defines what the attributes and methods of its objects are.

An object is simply an instance of the class.

The process of wrapping code and data together into a single unit this way is called Encapsulation.

For example, a capsule which is mixed of several medicines.

OOP Concepts

- Objects: Variables
- Classes: Datatypes
- Encapsulation: Data-hiding & Code-Data Wrapping
- Inheritance: Re-use
- Polymorphism: Multi-Forms
- Abstraction: Implementation-hiding

Advantages of OOP

Code Reuse

Design Benefits

Software maintenance

Simplicity

More.. : https://www.educba.com/advantages-of-oop/

Advanced Programming CSE 201

Lec 2: Objects, Classes & Encapsulation

Review

Java is a purely OOP language

Two stages: Compilation & Execution (Run)

Variables=> Object Variables

Datatypes=> Classes

An object has attributes and functions. We define them in the class of the object.

Try to bring related variables and methods together.

Course Evaluation

4 Assignments [20 Marks, n-1 policy]

4 Quizzes [10 Marks, n-1 policy]

Project [20 Marks]

Midsem [20 Marks]

Endsem [30 Marks]

Office Hours: Monday (after 3 pm) [on appointment]

How we should think in terms of Real-world?

Objects in real-world can be classified into a class of objects.

For example, all of us can be classified into a class of human being.

The human being class serves as template (or form) where all our attributes and functions are listed.

We are simply an object instance of the human being class

My car

fuel=0

Max speed=50

refuel()

Sachin's car

fuel=5

Max speed=60

refuel()

Your car

fuel=10

Max speed=75

getfuel()

Sehwag's car

fuel=2

Max speed=175

getfuel()

At any point, different cars have different attribute values, and they are performing different activities.

All of them are cars!

Let’s make a complex number calculator that takes two input complex numbers, performs basic algebraic operations of them and outputs the resultant complex number.

How many classes do we need?

Two: one for complex number, and another for calculator

Design Considerations

Complex number comprises of the real and imaginary data.

Calculator performs the algebraic operations.

Main class can take the responsibility of inputting and outputting

Refer to the attached complex_calc.java file

Encapsulation: Data-hiding (protection)

Using *private* keyword: Restricting the accessibility of (or ability to modify) the instance variable (data member) to the class only.

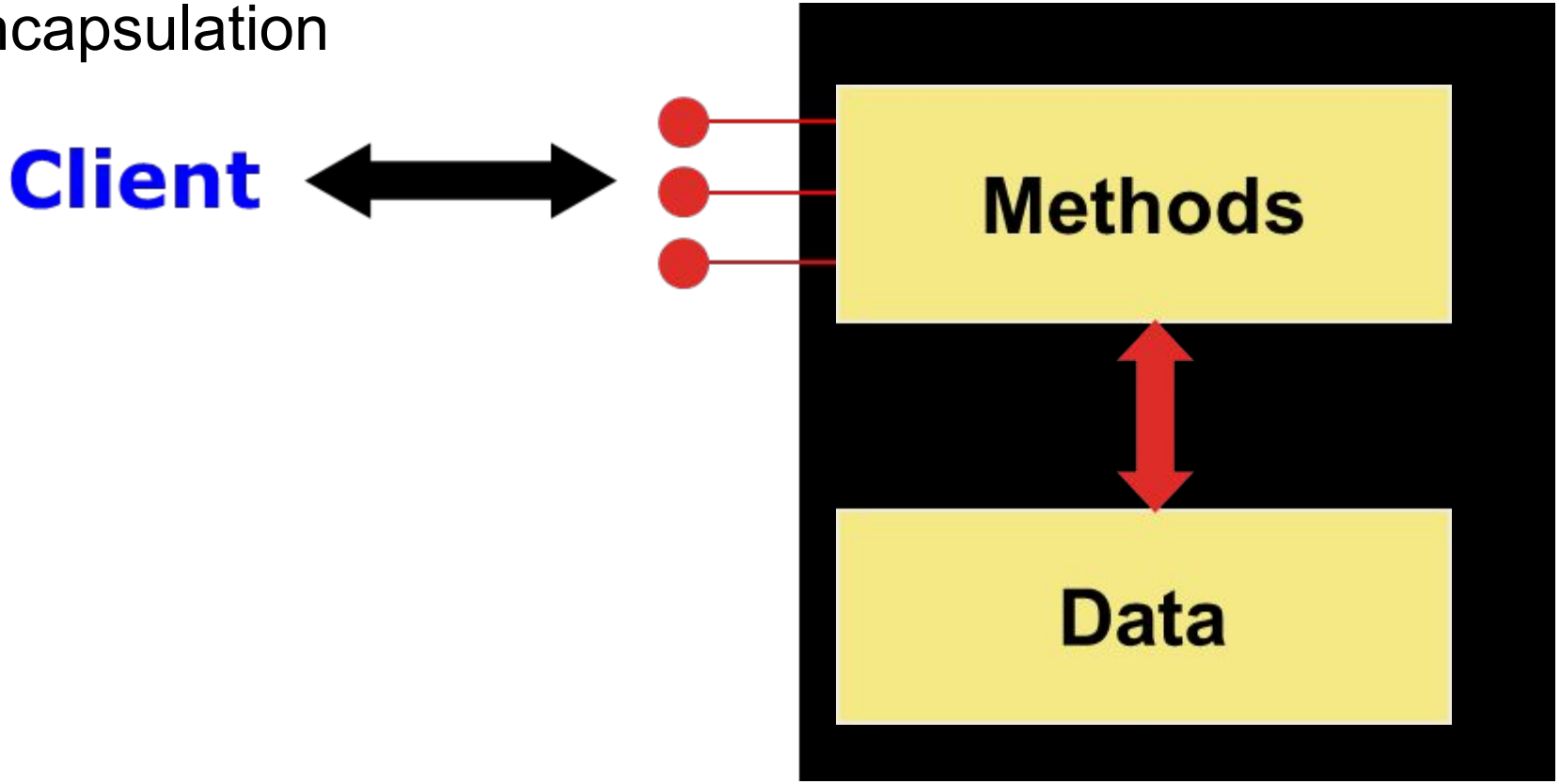
Using *final* keyword: Making the instance variable (data member) constant, i.e., it cannot be changed anymore.

Using Accessors (getters) and Mutators (setters): Allowing the access to (or modification of) private data members from the outside through methods in a controlled manner.

Accessors and Mutators

- Because instance data is private, a class usually provides services to access and modify data values
- An *accessor method* returns the current value of a variable
- A *mutator method* changes the value of a variable
- The names of accessor and mutator methods take the form `getX` and `setX`, respectively, where `X` is the name of the value
- They are sometimes called “getters” and “setters”

Encapsulation



Advanced Programming CSE 201

Lec 3: Class Relationships

Review

Encapsulation:

Using private keyword: Limiting the setting and access of an instance variable to within-class.

Using getters and setters: Allowing setting and access of private instance variables from outside in a controlled manner.

Using final: Not allowing to change an instance variable.

Refer to encapsulation.java file for example usage.

The objects don’t just exist, they also interact with each other.

These interactions lead to different relationships.

Programmatically,

The relationship depends on

1. whether a reference given to the instance variable is from outside or inside (Association vs Composition)
2. whether an outside reference is given to the instance variable or just being used (Association vs Dependance)

Types of relationships we will cover

- Dependence Relationship (using other objects)
- Composition Relationship (consisting of other objects)
- Association Relationship (associating with other objects)

Koteswar (an object of researcher class) uses Computer405 (an object of computer class).

Koteswar (an object of researcher class) consists of Koteswar's heart (an object of heart class).

Koteswar (an object of researcher class) is associated to IIIT-Delhi (an object of university class).

NOTE: Here, other objects means objects of another class

The heart has no independent existence. Its reference is from inside, and it gets assigned to researcher's heart variable
The computer has an independent existence. Its reference is from outside, and it's required just for the task.
The institute has an independent existence. Its reference is from outside, and it gets assigned to researcher's affiliation variable

Koteswar has a heart (Composition)

Koteswar works at IIIT-Delhi (Association)

Koteswar uses Computer405 to do research (Dependency)

These are relationships of different classes with respect to the researcher class.

Class	Object
Researcher	Koteswar
Computer	Computer-405
Heart	Koteswar's heart
Institution	IIIT-Delhi

Dependence Relationship

- The Computer405 is not Koteswar's property. It was given by IIIT-Delhi. But I am dependent on it; I need it for my research.
- So, I can use it in one of my functions, say writing papers.

- Let's say there are 'Koteswar' and 'Computer405' objects in the main function.
- Now, 'Koteswar' needs to write a paper, but 'Koteswar' needs a computer for that. So, *Koteswar.paper_writing(Computer405)* can be called.

Composition Relationship

- Koteswar's heart is Koteswar's property. If Koteswar don't exist, Koteswar's heart also won't exist.
- The Computer405 in previous slide can still exist without Koteswar, because IIIT-Delhi can assign it to some other researcher.

- A heart object needs to be specifically created for 'Koteswar' to call it Koteswar.heart (aka Kotewar's heart)
- When I write ***Koteswar= new Human();*** , the Constructor will create a heart object for me which we call *Koteswar.heart.*
- Once Koteswar doesn't exist, *Koteswar.heart* has no existence.

Association Relationship

- Koteswar is associated to IIIT-Delhi. Similarly, Vivek Sir is also associated. If Koteswar don't exist, IIIT-Delhi will still remain. If at all anything does get destroyed, that's Koteswar's association.

Let's say, In the main function, 'Koteswar' and 'IIIT-Delhi' objects exist.

Now, to form the association, I can call *Koteswar.affiliation=IIIT-Delhi*, where affiliation is an object of university class existing within the Researcher class.

If Koteswar is destroyed, *Koteswar.affiliation* has no meaning, but IIIT-Delhi will still prevail.

Summary

	Dependence	Composition	Association
The other object is an attribute of the object.	No	Yes	Yes
The other object can exist without the object	Yes	No	Yes

Refer to the attached class_rel.java file for more details.

Appreciate the difference between how the external computer's reference is just being used and how the external institution's reference is being passed to the variable 'i'. For Dependence vs Association.

Appreciate the difference between how the variable 'h' received an internal reference and how the variable 'i' received an external reference. For Composition vs Association.

Advanced Programming CSE 201

Lecture 4: Initializer Blocks & Static Members

Review

Dependence Relationship: The external reference is used to perform a task

Association Relationship: The external reference is assigned to an instance variable.

Composition Relationship: The instance variable is given an internal reference.

By external, it means the reference is coming from outside the class.

By internal, it means the reference is created inside the class.

Please see the three attached java files for examples

Ways to create the reference inside the class [For Composition]

- Just initialize where you declare
- Use constructors
- Use Instance Initializer Blocks

Ways to bring an external reference inside the class

- Through constructors: To assign it to the instance variable [For Association]
- Through functions: To perform the task [For Dependence]

Instance Initializer Blocks

- **Instance Initializer block** is used to initialize the instance data members. It runs each time when an object is created, before the first constructor call.
- Although they can be initialized directly (int i=10), we may require to do certain operations along with initialization. For example, your initialization may depend upon a static member data.
- You may say, I can do it in the constructor then.
- What if there are multiple constructors? Will you use the same operations in each constructor?
- Use initializer blocks to execute common part of multiple constructors.

{

}

```
class Bike7{
  int speed;

  Bike7(){System.out.println("speed is "+speed);}

  {speed=100;}

  public static void main(String args[]){
  Bike7 b1=new Bike7();
  Bike7 b2=new Bike7();
  }
}
```

```
speed is 100
speed is 100
```

Static Initializer Blocks (aka Static Blocks)

- Runs even before the main function (if at all present) of the class.
- Runs only once, when the class is loaded into the JVM, i.e., when the first object of the class is created.
- Helps initializing static data members and their default values. For e.g. you may want to initialize a static data member after analyzing system time.

static{

}

```java
class Test {

    // Case 1: Static variable
    static int i;
    // Case 2: non-static variables
    int j;

    // Case 3: Static block
    // Start of static block
    static
    {
        i = 10;
        System.out.println("static block called ");
    }
    // End of static block
}

// Class 2
// Main class
class GFG {

    // Main driver method
    public static void main(String args[])
    {

        // Although we don't have an object of Test, static
        // block is called because i is being accessed in
        // following statement.
        System.out.println(Test.i);
    }
}
```

Static Members

They belong to the class, not the object.

Static functions can access only static members of the class.

Non-static functions can access any member of the class.

Use *static* keyword while declaring these members.

```java
public class MyClass {
    public static void sample(){
        System.out.println("Hello");
    }
    public static void main(String args[]){
        MyClass.sample();
    }
}
```

```java
public class MyClass {
    public static int data = 20;
    public static void main(String args[]){
        System.out.println(MyClass.data);
    }
}
```

Lecture 5

Inheritance

Review

Instance Initializer block:

- to initialize instance variables.
- runs before the constructor
- can add code-based initialization here
- useful for code-reuse: we can place the recurring code in constructors here
- {}

Static Initializer block:

- To initialize static variables
- Runs only once, even before the main function (if there), when a class is loaded in JVM.
- static{}

Static members:

- Belong to the class
- static methods can access only static members
- Accessed through ClassName.

Inheritance

It allows us to create a new class from an existing class.

The new class that is created is known as subclass (child or derived class) and the existing class from where the child class is derived is known as superclass (parent or base class).

The `extends` keyword is used to perform inheritance in Java

```java
class Animal {

  // field and method of the parent class
  String name;
  public void eat() {
    System.out.println("I can eat");
  }
}

// inherit from Animal
class Dog extends Animal {

  // new method in subclass
  public void display() {
    System.out.println("My name is " + name);
  }
}
```

```java
class Main {
  public static void main(String[] args) {

    // create an object of the subclass
    Dog labrador = new Dog();

    // access field of superclass
    labrador.name = "Rohu";
    labrador.display();

    // call method of superclass
    // using object of subclass
    labrador.eat();

  }
}
```

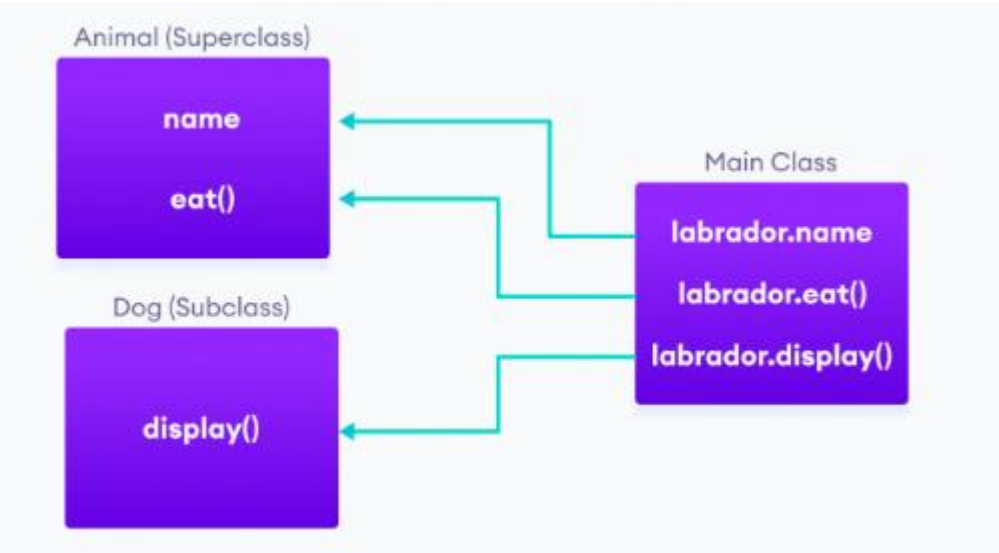
Inheritance relationship

In Java, inheritance is an “is-a” relationship. That is, we use inheritance if there exists an is-a relationship between two classes. For example,

- Car is a Vehicle
- Orange is a Fruit
- Surgeon is a Doctor
- Dog is an Animal

Here, Car can inherit from Vehicle, Orange can inherit from Fruit, and so on.

From where these members are coming?



“protected” keyword

Protected members are accessible to children everywhere, even outside the package.

There are four types of Java access modifiers:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

Path for package

To Compile a package: javac -d . A.java

To Run a package: java pack.A

Source file for package pack

```
package pack;
public class A{

  protected void msg(){System.out.println("Hello");}

  public static void main(String args[]){
    System.out.println("Hi");
  }

}
```

```
import pack.A;

class B{
  public static void main(String args[]){
   C obj = new C();
   obj.msg2();
  }
}

class C extends A{
     void msg2(){msg();}
}
```

1. Single Inheritance

In single inheritance, a single subclass extends from a single superclass. For example,

Java Single Inheritance

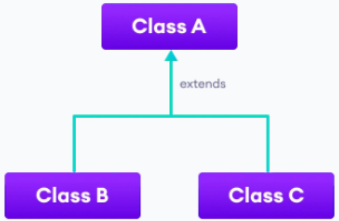
2. Multilevel Inheritance

In multilevel inheritance, a subclass extends from a superclass and then the same subclass acts as a superclass for another class. For example,

Java Multilevel Inheritance

3. Hierarchical Inheritance

In hierarchical inheritance, multiple subclasses extend from a single superclass. For example,



Java Hierarchical Inheritance

There are more..

We will discuss them later.

Why use inheritance?

- The most important use of inheritance in Java is code reusability. The code that is present in the parent class can be directly used by the child class.
- Method overriding is also known as runtime polymorphism. Hence, we can achieve Polymorphism in Java with the help of inheritance.

Method Overloading

```java
class Animal {

  // method in the superclass
  public void eat() {
    System.out.println("I can eat");
  }
}

// Dog inherits Animal
class Dog extends Animal {

  // overriding the eat() method
  @Override
  public void eat() {
    System.out.println("I eat dog food");
  }

  // new method in subclass
  public void bark() {
    System.out.println("I can bark");
  }
}
```

```java
class Main {
  public static void main(String[] args) {

    // create an object of the subclass
    Dog labrador = new Dog();

    // call the eat() method
    labrador.eat();
    labrador.bark();
  }
}
```

You wanted to override funcC, but you made a mistake by typing a 2 extra while overriding. @Override will save you from that mistake

```java
class BaseClass
{
    void funcA(){}
    void funcB(){}
    void funcC(){}
    void funcD(){}
}

class DerivedClass extends BaseClass
        {
        void funcA(){}

        void funcB(){}

        void funcC2(){}
        }
```

w/o @Override annotation

```java
class BaseClass
{
    void funcA(){}
    void funcB(){}
    void funcC(){}
    void funcD(){}
}

class DerivedClass extends BaseClass
        {
        void funcA(){}

        void funcB(){}

        @Override
        void funcC2(){}
        }
```

with @Override annotation

1 sec, 23 ms C:\Users\koteswar\IdeaProjects\AP_Lectures\src\com\IIITD_AP_KR2\Main.java:24:9

java: method does not override or implement a method from a supertype

Activate Windows
Go to Settings to activate Win

super keyword

the `super` keyword is used to call the method of the parent class from the method of the child class.

```java
class Animal {

  // method in the superclass
  public void eat() {
    System.out.println("I can eat");
  }
}

// Dog inherits Animal
class Dog extends Animal {

  // overriding the eat() method
  @Override
  public void eat() {

    // call method of superclass
    super.eat();
    System.out.println("I eat dog food");
  }

  // new method in subclass
  public void bark() {
    System.out.println("I can bark");
  }
}
```

```java
class Main {
  public static void main(String[] args) {

    // create an object of the subclass
    Dog labrador = new Dog();

    // call the eat() method
    labrador.eat();
    labrador.bark();
  }
}
```

It’s helping us distinguish between the eat methods of the two classes in the child class

Advanced Programming CSE 201

Lecture 6: Polymorphism

- Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.
- For example, a man at the same time can be a father, a husband, and an employee. So the same person possesses different behavior in different situations.

Method Overloading

Having multiple functions with a same name but having different number of arguments or their types in a class.

Suppose you have to perform addition of the given numbers but there can be any number of arguments.

If you write methods such as sum2(int,int) for two parameters and sum3(int,int,int) for three parameters, it may be difficult for you as well as other programmers to understand the behavior of the program, as their names are different.

Change is types of arguments can also affect our understanding. Imagine writing sum_int1_float2(int,float,float). It’s difficult to keep track of how we designed the function name earlier.

Instead, What if we have just have one name, sum? It becomes much easier to understand and remember.

Advantage of method overloading

Method overloading *increases the readability of the program*.

Different ways to overload a method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type of arguments

```java
class Adder{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
class TestOverloading1{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
}}
```

```
22
33
```

```java
class Adder{
static int add(int a, int b){return a+b;}
static double add(double a, double b){return a+b;}
}
class TestOverloading2{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
}}
```

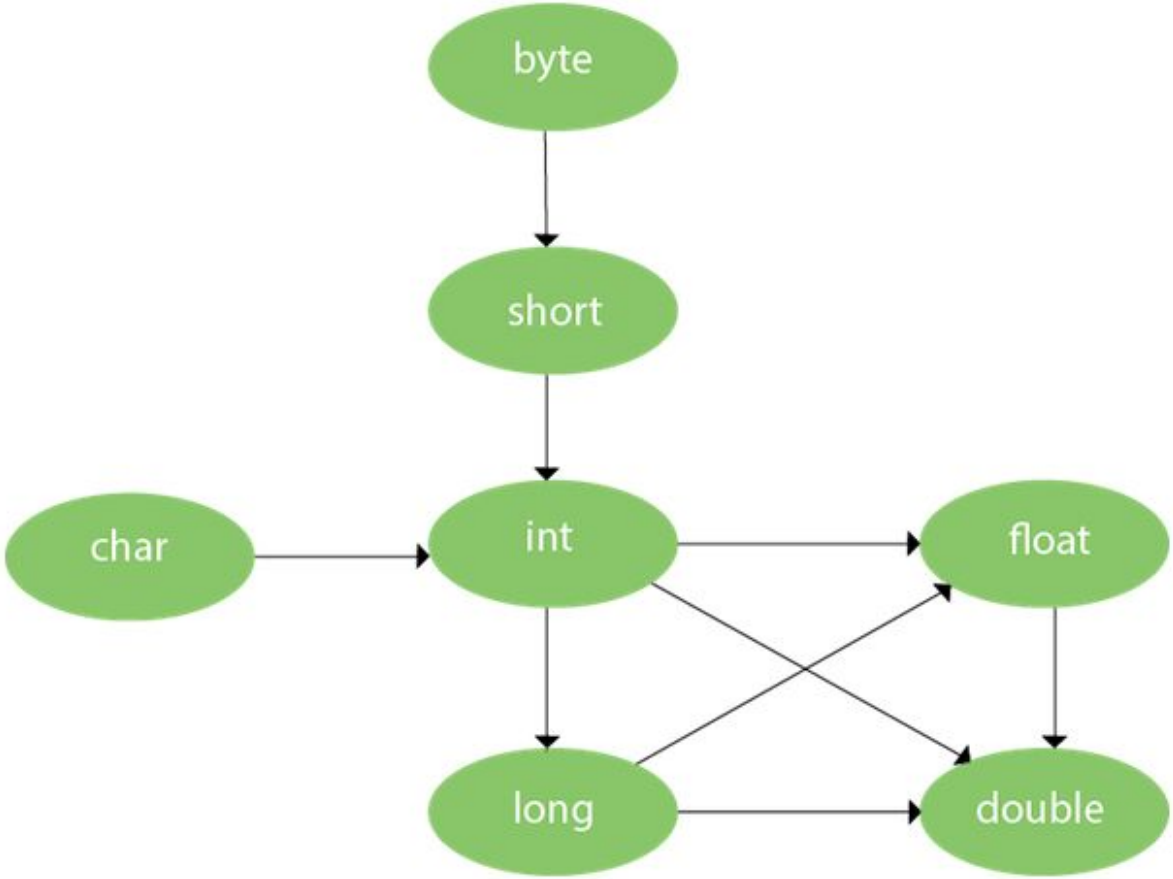
```
22
24.9
```

```java
class Adder{
static int add(int a,int b){return a+b;}
static double add(int a,int b){return a+b;}
}
class TestOverloading3{
public static void main(String[] args){
System.out.println(Adder.add(11,11));//ambiguity
}}
```

```
Compile Time Error
```

```java
class TestOverloading4{
public static void main(String[] args){System.out.println("main with String[]");}
public static void main(String args){System.out.println("main with String");}
public static void main(){System.out.println("main without args");}
}
```

```
main with String[]
```



As displayed in the above diagram, byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int, long, float or double. The char datatype can be promoted to int,long,float or double and so on.

```java
class OverloadingCalculation1{
 void sum(int a,long b){System.out.println(a+b);}
 void sum(int a,int b,int c){System.out.println(a+b+c);}

 public static void main(String args[]){
 OverloadingCalculation1 obj=new OverloadingCalculation1();
 obj.sum(20,20);//now second int literal will be promoted to long
 obj.sum(20,20,20);

 }
}
```

```java
class OverloadingCalculation3{
 void sum(int a,long b){System.out.println("a method invoked");}
 void sum(long a,int b){System.out.println("b method invoked");}

 public static void main(String args[]){
 OverloadingCalculation3 obj=new OverloadingCalculation3();
 obj.sum(20,20);//now ambiguity
 }
}
```

Method Overriding

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding.**

In such cases, subclass provides the specific implementation of the method that has been declared by one of its parent class.

Consider a scenario where Bank is a class that provides functionality to get the rate of interest. However, the rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7%, and 9% rate of interest.

```java
class Bank{
int getRateOfInterest(){return 0;}
}
//Creating child classes.
class SBI extends Bank{
int getRateOfInterest(){return 8;}
}

class ICICI extends Bank{
int getRateOfInterest(){return 7;}
}
class AXIS extends Bank{
int getRateOfInterest(){return 9;}
}
//Test class to create objects and call the methods
class Test2{
public static void main(String args[]){
SBI s=new SBI();
ICICI i=new ICICI();
AXIS a=new AXIS();
System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
}
}
```

Polymorphic Variables

A variable is called polymorphic if it refers to different values under different conditions.

Object variables represent the behavior of polymorphic variables in Java. It is because object variables of a class can refer to objects of its class as well as objects of its subclasses.

It helps in creating an array of different datatypes. See the attached PolymorphivVariableArray.java for an example.

It helps in calling a method using altered arguments. See the attached PolymorphivVariableMethod.java for an example.

```java
class ProgrammingLanguage {
  public void display() {
    System.out.println("I am Programming Language.");
  }
}

class Java extends ProgrammingLanguage {
  @Override
  public void display() {
    System.out.println("I am Object-Oriented Programming Language.");
  }
}

class Main {
  public static void main(String[] args) {

    // declare an object variable
    ProgrammingLanguage pl;

    // create object of ProgrammingLanguage
    pl = new ProgrammingLanguage();
    pl.display();

    // create object of Java class
    pl = new Java();
    pl.display();
  }
}
```

```
I am Programming Language.
I am Object-Oriented Programming Language.
```

- In statement `pl = new ProgrammingLanguage()`, `pl` refer to the object of the `ProgrammingLanguage` class.
- And, in statement `pl = new Java()`, `pl` refer to the object of the `Java` class.

- Alright, I can get them into an array. What is the advantage?
- You have made a family. You have created a unit that can do what they cannot do alone.
- Let's say you have homemaker and moneymaker classes. You want to create something that can do the both: moneymaking & homemaking.

```
class Homemaker
{
   void homemaking(){System.out.println("I keep the house clean everyday");}
}

class Moneymaker
{
   void moneymaking(){System.out.println("I make 100$s everyday");}
}
```

Activate
Go to Setti

Let’s combine them into an array of family

- For that I need a family class, which will be extended by the two.

```
class Family
{

}

class Homemaker extends Family
{
   void homemaking(){System.out.println("I keep the house clean everyday");}
}

class Moneymaker extends Family
{
   void moneymaking(){System.out.println("I make 100$s everyday");}
}
```

This won’t work!!

- Let’s have a main class also

```
public class Main {

   public static void main(String[] args) {
      Family A=new Homemaker();
      Family B=new Moneymaker();
      Family[] f= {A,B};
      f[0].homemaking();
      f[1].moneymaking();
   }
}
```

Activate Win
Go to Settings to

```java
class Family
{
void homemaking(){};
void moneymaking(){};
}

class Homemaker extends Family
{
@Override
   void homemaking(){System.out.println("I keep the house clean everyday");}
}

class Moneymaker extends Family
{
@Override
   void moneymaking(){System.out.println("I make 100$s everyday");}
}
```

This will work!! Thanks to Overriding!!

```java
public class Main {

   public static void main(String[] args) {
      Family A=new Homemaker();
      Family B=new Moneymaker();
      Family[] f= {A,B};
      f[0].homemaking();
      f[1].moneymaking();
   }
}
```

```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\Je
I keep the house clean everyday
I make 100$s everyday

Process finished with exit code 0
```

Why and how did it work?

- Earlier, when the functions were being called, compiler didn't find any function in the family class. Hence threw an error.
- But when you add overridable functions to the family class, compiler sees the functions but found it overridable. So, it leaves it to JVM.
- JVM finds the object-type and calls respective functions.
- Now if I add any of the three—static, final and private—to these functions, compiler finds the functions it needs, so functions in Family class will be called which print nothing.
- So, in this way, we could create an array that could do tasks of heterogeneous objects.

How do we associate a function to the function call?

a.func()

We will look at the type of the variable a

Now which type to choose?
Reference type or Object type?

Any variable can have two types:
reference-type & object-type

The class of the variable

The class of the object the variable is storing the address of

So, compiler & JVM sign a deal (Overriding)

- Let every function be virtual, which means allowing a function to be overridden.
- Once a class is created, it can always be extended. If the extending class wants to choose a function of same name and signature as the one in extended class, it can always do so. And it's called overriding.
- In that case, we annotate this function with an annotation @Override. It is considered best practice to do so. Why?
- If you make a common mistake of misspelling the overriding method name or not correctly matching the parameters, you will be warned that your method does not actually override as you think it does.

Compiler says, “Let me check in the reference type class, if the function can be overridden”

- It checks for three cases:

❑ If it’s static

❑ If it’s final

❑ If it’s private

cannot override if any of these is found !.

If any of the above satisfies, the Compiler says, “Hurray!! I got it. JVM, better luck next time!”

If that doesn’t happen, the Compile says, ”JVM, I am not sure whether it will get overridden or not. You find it when you run a program.”

JVM says, “Thanks, Man. Let me know whenever I have to do my job.”

- The compiler sends the message through bytecode.
- Now if JVM received the message that it should check. JVM knows there is a reference of an object in the variable. Let me check what is the type of that object, which is called object-type of the variable.
- JVM goes to the object-type class and finds the function to call.

When compiler is not able to resolve the call/binding at compile time, such binding is known as Dynamic or late Binding.

If the compiler is able to resolve, it's called static or early binding.

Steps to know whether it's going to be static or dynamic binding.

1) Go to the reference-type class of the variable ('a' is a.func()) that is calling the function.
2) If there is no such function, there will be a compile time error.
3) If there is such a function and it's a static, final or private, it will static binding.
4) Otherwise, it will be dynamic binding

Method Overloading is compile-time polymorphism.

Method Overloading occurs when multiple methods in the same class have the same name but different parameters

Method Overriding is run-time polymorphism.

Method overriding occurs when a subclass provides a specific implementation of a method that is already defined in its superclass

Advanced Programming CSE 201

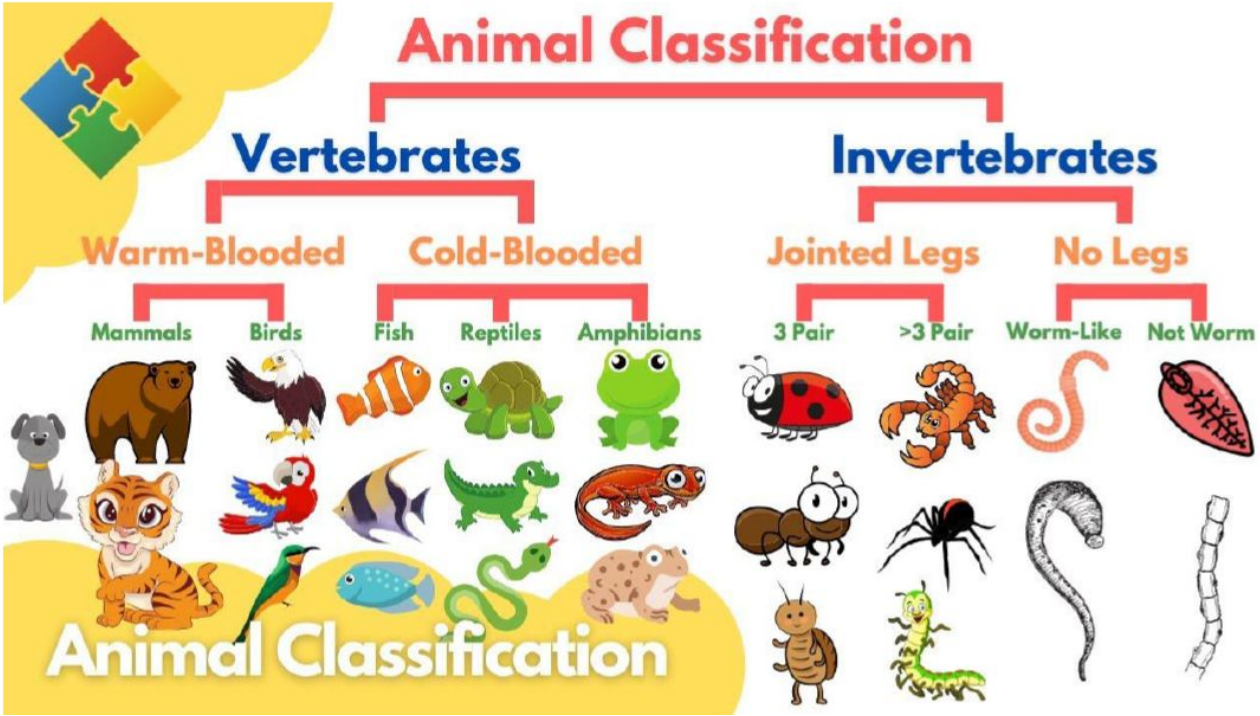
Lecture 7: Abstraction (Abstract Classes & Interfaces)

Abstraction

The main idea is to handle complexity by hiding unnecessary details from the user.

It enables the user to implement more complex logic on top of the provided abstraction without understanding or even thinking about all the hidden complexity.

What do we observe here



A real world object can belong to multiple classes

For e.g. my dog is a dog, a mammal, a warm-blooded animal, a vertebrate, and an animal.

However, only one class covers all the functionalities in minute detail. In this case, that's dog class.

What are other classes then? They are either interfaces or abstract classes

Abstract Method

Just declare the method; don't define (or implement) it.

```
class Main{

public static void main(String args[]){

Parent p;

p=new Child1();
func(p);
p=new Child2();
func(p);
}

static void func(Parent p){p.disp();}

}

class Parent{
void disp(){}
}

class Child1 extends Parent{
void disp(){System.out.println("Child1");}
}

class Child2 extends Parent{
void disp(){System.out.println("Child2");}
}
```

```
class Main{

public static void main(String args[]){

Parent p;

p=new Child1();
func(p);
p=new Child2();
func(p);
}

static void func(Parent p){p.disp();}

}

abstract class Parent{
abstract void disp();
}

class Child1 extends Parent{
void disp(){System.out.println("Child1");}
}

class Child2 extends Parent{
void disp(){System.out.println("Child2");}
}
```

```
class Main{

public static void main(String args[]){

Parent p;

p=new Child1();
func(p);
p=new Child2();
func(p);
}

static void func(Parent p){p.disp();}

}

interface Parent{
 void disp();
}

class Child1 implements Parent{
public void disp(){System.out.println("Child1");}
}

class Child2 implements Parent{
public void disp(){System.out.println("Child2");}
}
```

If there is even a single abstract method, the entire class needs to be made abstract.

In an interface, all the methods are abstract.

Members of an interface are public by default.

Interfaces & Abstract Classes

- They can't create real world objects. Why? They don't cover all the functionalities in minute detail.
- Why don't they cover? Since there are multiple ways to define these functionalities, they leave it to their child classes who can cover them for their own objects.
- For e.g. there are many ways animals can drink. If they were to have real world objects, we need to have multiple functions to just drink. Imagine a class like this:

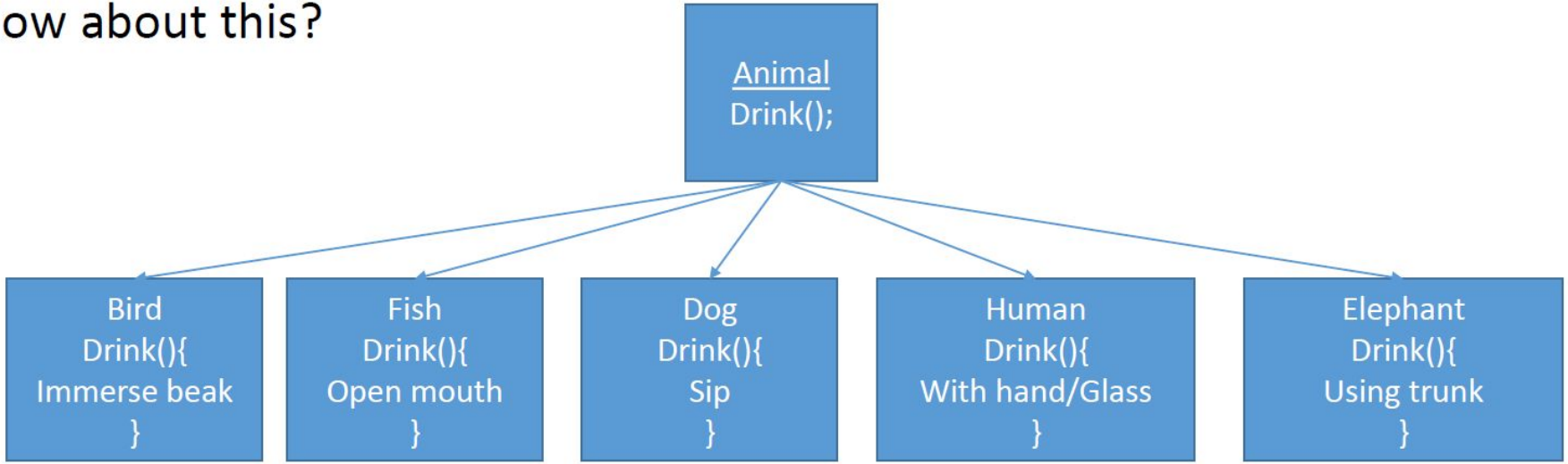
Animal
Drink_immerse_beak()
Drink_open_mouth()
Drink_sip()
Drink_with_hand()
Drink_with_trunk()

Then, there are many things animals can do differently!!

Activate Win
Go to Settings to

Interfaces & Abstract Classes

- How about this?



Interfaces & Abstract Classes

- Why have the animal class then when its children classes do the job?
- Sometimes we need to treat dogs, birds, elephants as animals. Let's say you are creating an array of animals.
- In IIITD, sometimes we need to be treated as IIITD members, irrespective of whether one is a faculty or a student. For example, members of a google classroom. Both you and myself are members.
- So, how does it work?
- Pass on the reference of real world objects of child classes to variables of these types.

Interfaces/abstract classes give sense of equality(=) to child objects. In this way whatever we could do with objects of one type, we can now do with objects of different types.

By equality, I mean we can put them in an array and pass them to a method.

- Interfaces/abstract classes list only common functionalities.
- As IIITD members, our common functionality could be Engage(). You and me are engaged differently.

Reference-type Object-type

- Animal A1=new Dog();
- Animal A2;
- D1=new Dog();
- A2=D1
- Animal A3=new Bird();
- Animal[] arr={A1,A2,A3};

```
class Main{

public static void main(String args[]){

Animal p;

p=new Dog();
func(p);
p=new Fish();
func(p);
}

static void func(Animal p){p.drink();}

}

interface Animal{
 void drink();
}

class Dog implements Animal{
public void drink(){System.out.println("Sips");}
}

class Fish implements Animal{
public void drink(){System.out.println("Opens mouth");}
}
```

How is all this related to Abstraction?

- Implementations are hidden in some other classes. You can see only what is required. You just see function definitions without worrying about how they are implemented.
- Let's take a car interface.
- Every car has a steering, clutch, gear and brake, but the way they are implemented may vary company to company.
- From your perspective, a BMW is car interface. You don't know how it's implemented.
- From a BMW engineer's perspective, a BMW is BMW. He knows how it's implemented.

Interfaces vs Abstract Classes

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves full abstraction (100%).

Another example: Car as Transporter

```
public interface Transporter {

    public void move();

}
 public class Car implements Transporter {
     public Car() {
         //code elided
     }
     public void drive(){
         //code elided
     }
     @Override
     public void move(){
         this.drive();
         this.brake();
         this.drive();
     }
     //more methods elided
 }
```

Complete this program such that a car object is created and is assigned to a transporter variable. When you call the move function, you should be able to alter the speed of the car with the help of a couple of drive (to increase the speed) and brake (to decrease the speed) functions.

- As with signing multiple contracts, classes can implement multiple interfaces
  - "I signed my rent agreement, so I'm a renter, but I also signed my employment contract, so I'm an employee. I'm the same person."
  - what if I wanted `Car` to change color as well?
  - create a `Colorable` interface
  - add that interface to `Car`'s class declaration
- Implementing class must define every single method in each of its interfaces

```
public interface Colorable {

    public void setColor(Color c);
    public Color getColor();

}

public class Car implements Transporter, Colorable{

    public Car(){ //body elided }
    public void drive(){ //body elided }
    public void move(){ //body elided }
    public void setColor(Color c){ //body elided }
    public Color getColor(){ //body elided  }
}
```

- Interfaces are formal contracts and ensure consistency
  - compiler will check to ensure all methods declared in interface are defined
- Can trust that any object from class that implements `Transporter` can `move()`

Question

Given the following interface:

```
public interface Clickable {
    public void click();
}
```

Which of the following would work as an implementation of the Clickable interface? (don't worry about what changeXPosition does)

A.

```
public void click() {
    this.changeXPosition(100.0);
}
```

B.

```
public void click(double xPosition) {
    this.changeXPosition(xPosition);
}
```

C.

```
public void clickIt() {
    this.changeXPosition(100.0);
}
```

D.

```
public double click() {
    return this.changeXPosition(100.0);
}
```

Advanced Programming CSE 201

Lecture 8: Immutable Classes | Introduction to Object Class

Immutable Class

An immutable class is simply a class whose objects cannot be modified.
All of the information contained in each object is fixed for the lifetime of the object, so no changes can be made.

1. **Don’t provide “setter” methods — methods that modify fields or objects referred to by fields.**
   This principle says that for all mutable properties in your class, do not provide setter methods. Setter methods are meant to change the state of an object and this is what we want to prevent here.
2. **Make all fields final and private**
   This is another way to increase *immutability*. Fields declared private will not be accessible outside the class and making them final will ensure the even accidentally you can not change them.
3. **Special attention when having mutable instance variables**
   Always remember that your instance variables will be either mutable or immutable. Identify them and return new objects with copied content for all mutable objects. Immutable variables can be returned safely without extra effort.
4. **Don’t allow subclasses to override methods**
   The simplest way to do this is to declare the class as *final*. Final classes in java can not be extended.

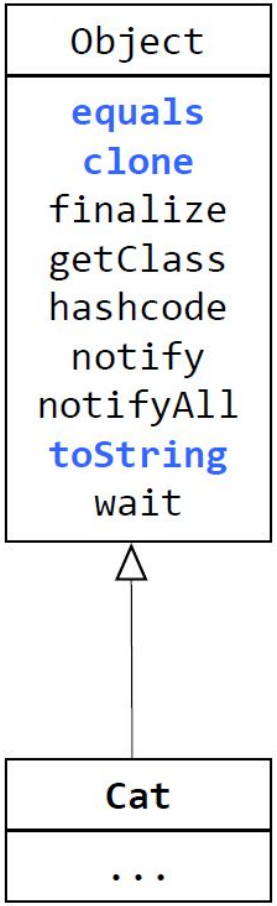
Once you have made an object of such a class, you can freely pass them. You don’t have to worry about someone changing its data.

The Class Object

- The class `Object` forms the root of the overall inheritance tree of all Java classes.
  - Every class is implicitly a subclass of `Object`
  - `No need to explicitly say "extends Object"`

All classes implicitly extend Object class, if they don't extend any other super-class.

- The `Object` class defines several methods that become part of every class you write. For example:
  - `public String toString()`
    Returns a text representation of the object, usually so that it can be printed.



Activate

Given an object variable ‘a’

If there is a toString() function in the class of ‘a’, it overrides the toString() function of the object class. This way, you can have the text representation of the object as as you want.

In System.out.println(a), a.toString() function implicitly called.

Hence, you can simply write System.out.println(a) instead of System.out.println(a.toString()), and it will work the the same.

Object Methods

method	description
`protected Object clone()`	creates a copy of the object
`public boolean equals(Object o)`	returns whether two objects have the same state
`protected void finalize()`	called during garbage collection
`public Class<?> getClass()`	info about the object's type
`public int hashCode()`	a code suitable for putting this object into a hash collection
`public String toString()`	text representation of the object

`public void notify()`
`public void notifyAll()`
`public void wait()`
`public void wait(...)`
	methods related to concurrency and locking (seen later)

Activate Windows
Go to Settings to activate Win

Advanced Programming CSE 201

Lecture 9: Object Class & Comparisons

Using the Object Class

- You can store any object in a variable of type `Object`.

```
Object o1 = new Cat("Meau", "Indian Cat");
Object o2 = "hello there";
```

Question: speak() is a method in Cat class, is this correct?

1) o1.speak()
2) o1.toString()

- You can write methods that accept an `Object` parameter.

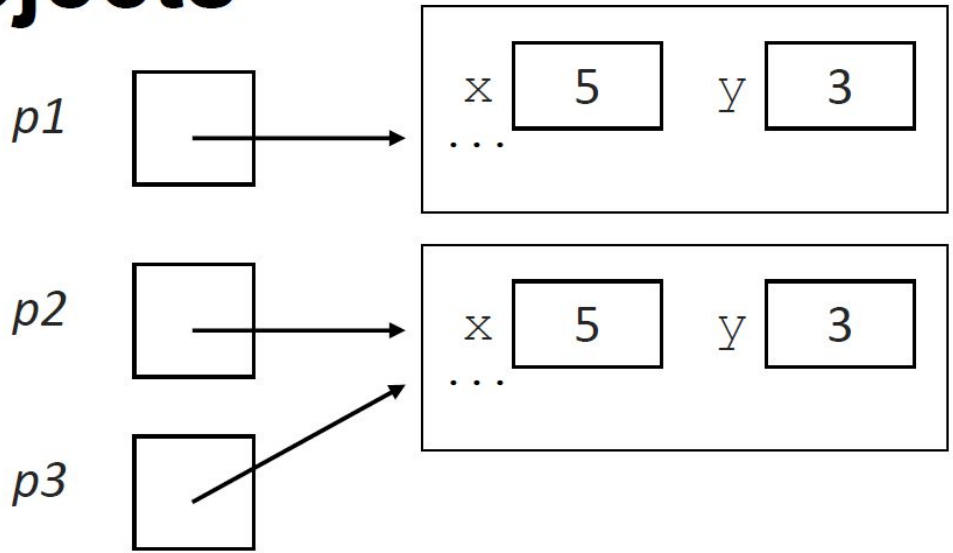
```
public void example(Object o) {
    if (o != null) {
        System.out.println("o is " + o.toString());
    }
```

- You can make arrays or collections of `Objects`.

```
Object[] a = new Object[5];
a[0] = "hello";
a[1] = new Cat();
List<Object> list = new ArrayList<Object>();
```

Equality Test on Objects

```
Point p1 = new Point(5,3);
Point p2 = new Point(5,3);
Point p3 = p2;

// p1 == p2 is false;
// p1 == p3 is false;
// p2 == p3 is true

// p1.equals(p2)?
// p2.equals(p3)?
```



- The == operator does not work well with objects.
  - == tests for **referential equality**, not state-based equality.
  - It produces true only when you compare an object to itself

Default `equals` Method

- The `Object` class's `equals` implementation is very simple:

```
public class Object {
    ...
    public boolean equals(Object o) {
        return this == o;
    }
}
```

- The Object class is designed for inheritance.
  - Subclasses can *override* `equals` to test for equality in their own way

Is this Correctly Implemented

```
1. public class Point {
2.    private int x, y;
3.    public Point(int _x, int _y) { ... }
4.    @Override
5.    public boolean equals(Point o) {
6.       return (x==o.x && y==o.y);
7.    }
8. }
9.
```

- Wrong Implementation !
  - **Flaw-1**
    - Signature of `equals` method doesn't matches with that in class `Object`
    - Compilation error as we are not overriding!
  - The parameter to `equals` method is not of type `Object` but is of type `Point`

Is this Correctly Implemented NOW ?

```
1. public class Point {
2.    private int x, y;
3.    public Point(int _x, int _y) { ... }
4.    @Override
5.    public boolean equals(Object o) {
6.       return (x==o.x && y==o.y);
7.    }
8. }
9.
```

- Still incorrect !
  - **Flaw-2**
    - Compilation error as the parameter to `equals` is of `Object` type but then `x` and `y` is not defined in class `Object`
  - Can we can do the following:

```
Object o1 = new Point(1, 2);
// Type casting below
Point p = (Point) o1;
```

Is this Correctly Implemented NOW ?

```
1. public class Point {
2.     private int x, y;
3.     public Point(int _x, int _y) { ... }
4.     @Override
5.     public boolean equals(Object o1) {
6.         Point o = (Point) o1; //type casting
7.         return (x==o.x && y==o.y);
8.     }
9. }
10.
```

- Still incorrect !
  - **Flaw-3**
    - It compiles and works fine if `Point` type objects are passed but fail to compile if non-`Point` type objects are passed
  - The typecasting will be an issue for following statement

    ```
    Object o1=new Point(1,2);
    Object o2="hello";
    boolean cond=o1.equals(o2);
    ```
  - The flaw is in line 6 as not every Object will be of Point type:

    ```
    Point o = (Point) o1;
    ```

The instanceof Keyword

```
if (variable instanceof type) {
    statement;
}
```

- Tests whether **variable** refers to an object of class **type** (or any subclass of **type**)

```
String s = "hello";
Point p = new Point();
```

expression	result
`s instanceof Point`	`false`
`s instanceof String`	`true`
`p instanceof Point`	`true`
`p instanceof String`	`false`
`p instanceof Object`	`true`
`s instanceof Object`	`true`
`null instanceof String`	`false`
`null instanceof Object`	`false`

(null is a reference and is not an object)

Is this Correctly Implemented NOW ?

```
1. public class Point {
2.    private int x, y;
3.    public Point(int _x, int _y) { ... }
4.    @Override
5.    public boolean equals(Object o1) {
6.       if(o1 instanceof Point) {
7.          Point o = (Point) o1; //type casting
8.          return (x==o.x && y==o.y);
9.       }
10.      else {
11.         return false;
12.      }
13.   }
14. }
15. // subclass of Point
16. class Point3D extends Point {
17.    private int z;
18.    public Point3D(int_x,int _y,int _z) {...}
19.    .....
20. }
```

- Still incorrect !
  - **Flaw-4**
    - The method `equals` will not behave correctly if `Point` class is extended

```
Point3D p1 = new Point3D(1,2,0);
Point3D p2 = new Point3D(1,2,3);
Point p3 = new Point(1,2);
p1.equals(p2); // true
p2.equals(p3); // true
p3.equals(p1); // true
```

Is this Correctly Implemented NOW ?

```
1. public class Point {
2.    private int x, y;
3.    public Point(int _x, int _y) { ... }
4.    @Override
5.     public boolean equals(Object o1) {
6.        if(o1 instanceof Point) {
7.          Point o = (Point) o1; //type casting
8.          return (x==o.x && y==o.y);
9.        }
10.       else {
11.         return false;
12.       }
13.    }
14.  }
15.  // subclass of Point
16.  class Point3D extends Point {
17.     private int z;
18.     public Point3D(int _x, int _y, int _z) { ... }
19.     @Override
20.     public boolean equals(Object o1) {
8.        if(o1 instanceof Point3D) {
9.          Point3D o = (Point3D) o1; //type casting
8.          return (super.equals(o1) && z==o.z);
9.        }
10.       else {
11.         return false;
12.       }
13.     }
14.  }
```

- Still incorrect !
  - **Flaw-5**
    - It produces *asymmetric* results when Point and Point3D are mixed

```
Point p1 = new Point(1,2);
Point3D p2 = new Point3D(1,2,3);
p1.equals(p2); // true
p2.equals(p1); // false
```

Equality should be symmetric !!

Rules of Equality for Any Two Objects

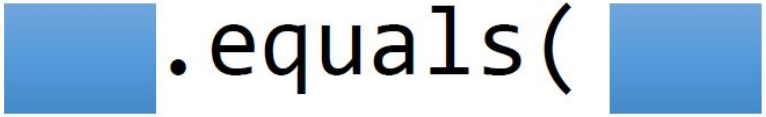
- Equality is reflexive:
  - `a.equals(a)` is `true` for every object a
- Equality is symmetric:
  - `a.equals(b)` ↔ `b.equals(a)`
- Equality is transitive:
  - `(a.equals(b) && b.equals(c))` ↔ `a.equals(c)`
- No non-`null` object is equal to `null`:
  - `a.equals(null)` is `false` for every object a

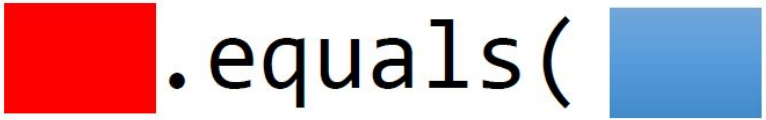
Finally, the Correct Implementation

```
1. public class Point {
2.    private int x, y;
3.    public Point(int _x, int _y) { ... }
4.    @Override
5.     public boolean equals(Object o1) {
6.        if(o1 != null && getClass() == o1.getClass()) {
7.          Point o = (Point) o1; //type casting
8.          return (x==o.x && y==o.y);
9.        }
10.       else {
11.         return false;
12.       }
13.    }
14.  }
15.  // subclass of Point
16.  class Point3D extends Point {
17.     private int z;
18.     public Point3D(int _x, int _y, int _z) { ... }
19.     @Override
20.     public boolean equals(Object o1) {
8.        if(o1 != null && getClass() == o1.getClass()) {
9.          Point3D o = (Point3D) o1; //type casting
8.          return (super.equals(o1) && z==o.z);
9.        }
10.       else {
11.         return false;
12.       }
13.     }
14.  }
```

- `getClass` returns information about the type of an object
  - Stricter than `instanceof`; subclasses return different results
- `getClass` should be used when implementing equals
  - Instead of `instanceof` to check for same type, use `getClass`
  - This will eliminate subclasses from being considered for equality
  - Caution: Must check for `null` before calling `getClass`

Comparing Objects in Java





Can we use `equals` to get the above arrangement?

- We have seen how to check equality between two objects:
  - `Obj1 == Obj2`
  - `Obj1.equals(Obj2)`
- But how to check the following:
  - `Obj1 < Obj2`
  - `Obj1 > Obj2`
- Operators like < and > do not work with objects in Java

Comparing Objects in Java

.compareTo( ) < 0

.compareTo( ) > 0

.compareTo( ) = 0

A call of **A**.compareTo(**B**) should return:

// if **A** comes "before" **B** in
// the ordering, a `value < 0`
// if **A** comes ”after" **B** in
// the ordering, a `value > 0`
// or exactly 0 if **A** and **B**
// are “equal” in the ordering

The Comparable Interface

- The standard way for a Java class to define a comparison function for its objects is to implement the Comparable interface.

```
public interface Comparable<T> {
    public int compareTo(T other);
}
```

Helps avoid code duplication

```java
class complex_int {
    int re;
    int im;
    complex_int(int re,int im){
        this.re=re;
        this.im=im;
    }
    @Override
    public String toString(){
        String s=String.valueOf(re)+'+'+String.valueOf(im)+'i';
        return s;
    }
}

class complex_double {
    double re;
    double im;
    complex_double(double re,double im){
        this.re=re;
        this.im=im;
    }
    @Override
    public String toString(){
        String s=String.valueOf(re)+'+'+String.valueOf(im)+'i';
        return s;
    }
}
```

```java
class complex <T> {
    T re;
    T im;

    complex (T re, T im){
        this.re=re;
        this.im=im;
    }

    @Override
    public String toString(){
        String s=String.valueOf(re)+'+'+String.valueOf(im)+'i';
        return s;
    }
}
```

compareTo Example

```
public class Rectangle implements Comparable<Rectangle> {
    private int sideA, sideB, area;
    public Rectangle (int _a, int _b) { ... }

    @Override
    public int compareTo(Rectangle o) {
      if(area == o.area) return 0;
          else if(area < o.area) return -1;
          else return 1;
    }
}
```

- In this Rectangle class, the compareTo method compares the Rectangle objects as per their area
- You can choose your own comparison algorithm!

compareTo v/s equals

```
public class Rectangle implements Comparable<Rectangle> {
   private int sideA, sideB, area;
   public Rectangle (int _a, int _b) { ... }

   @Override
   public int compareTo(Rectangle o) {
         if(area == o.area) return 0;
       else if(area < o.area) return -1;
       else return 1;
   }
   @Override
      public boolean equals(Object o1) {
     if(o1 != null && getClass() == o1.getClass()) {
       Rectangle o = (Rectangle) o1; //type casting
           return (sideA==o.sideA && sideB==o.sideB);
         }
         else {
           return false;
         }
       }
}
```

```
// Area1 = 2 x 32 = 64
Rectangle r1=Rectange(2, 32);
// Area2 = 4 x 16 = 64
Rectangle r2=Rectange(4, 16);
if(r1.compareTo(r2)==0) {
  // is this true??
}
if(r1.equals(r2)) {
  // is this true??
}
```

Recall, that two `Rectangles` with same `area` could still have different values for `sideA` and `sideB`

How to Compare Two Objects in Multiple Ways ?

- Our Rectangle class can only implement one compareTo method and hence only one comparison algorithm (style)
- We may want to compare two Rectangles differently
  - Based on sides
  - Based on area
  - ……

Comparator Interface

```
public interface Comparator<T> {
    public int compare(T first, T second);
}
```

- Interface `Comparator` is an external object that specifies a comparison function over some other type of objects.
  - Allows you to define multiple orderings for the same type.
  - Allows you to define a specific ordering for a type even if there is no obvious "natural" ordering for that type

Comparator Example

```
public class RectangleAreaComparator
                    implements Comparator<Rectangle> {
    @Override
    public int compare(Rectangle r1, Rectangle r2) {
        return r1.getArea() - r2.getArea();
    }
}
```

```
public class RectangleSidesComparator
                    implements Comparator<Rectangle> {
    @Override
    public int compare(Rectangle r1, Rectangle r2) {
        if (r1.getSideA() != r2.getSideA()) {
            return r1.getSideA() - r2.getSideA();
        } else {
            return r1.getSideB() - r2.getSideB();
        }
    }
}
```

class Rectangle {
private int sideA, sideB, Area}

- Using Comparators, two objects could be compared in different possible ways
- For creating different comparison, implement different objects of Comparator type

```
Class Main {
public static void main(String[] args) {
  Rectangle r1=Rectange(2, 32);
  Rectangle r2=Rectange(4, 16);
  RectangleAreaComparator rac = new RectangleAreaComparator();
  RectangleSidesComparator rsc = new RectangleSidesComparator();
  int area_result = rac.compare(r1, r2);
  int sides_result = rsc.compare(r1, r2);
}
}
```

Advanced Programming CSE 201

Lecture 10-11: Generic Programming

- Our generic cup can hold different types of liquid
- In the notation `Cup<T>`:
  - T = Coffee
  - T = Tea
  - T = Milk
  - T = Soup
  - ……

Cup == Generic Container

```java
class complex_int {
    int re;
    int im;
    complex_int(int re,int im){
        this.re=re;
        this.im=im;
    }
    @Override
    public String toString(){
        String s=String.valueOf(re)+'+'+String.valueOf(im)+'i';
        return s;
    }
}

class complex_double {
    double re;
    double im;
    complex_double(double re,double im){
        this.re=re;
        this.im=im;
    }
    @Override
    public String toString(){
        String s=String.valueOf(re)+'+'+String.valueOf(im)+'i';
        return s;
    }
}
```

A way of making it generic that you already know:

```java
class complex{
    Object re;
    Object im;
complex (Object re, Object im){
this.re=re;
this.im=im;
}
@Override
public String toString(){
String s=String.valueOf(re)+"+"+String.valueOf(im)+"i";
return s;
}
}
```

We can also generalize through generic programming.

```java
class complex_int {
    int re;
    int im;
    complex_int(int re,int im){
        this.re=re;
        this.im=im;
    }
    @Override
    public String toString(){
        String s=String.valueOf(re)+'+'+String.valueOf(im)+'i';
        return s;
    }
}

class complex_double {
    double re;
    double im;
    complex_double(double re,double im){
        this.re=re;
        this.im=im;
    }
    @Override
    public String toString(){
        String s=String.valueOf(re)+'+'+String.valueOf(im)+'i';
        return s;
    }
}
```

Generic Type Parameter (GTP)

```java
class complex <T> {
    T re;
    T im;

    complex (T re, T im){
        this.re=re;
        this.im=im;
    }

    @Override
    public String toString(){
        String s=String.valueOf(re)+'+'+String.valueOf(im)+'i';
        return s;
    }
}
```

Generic Type Parameter (GTP)

- Generic meaning: Not specific
- Type meaning: A class or interface
- Parameter meaning: A variable used to represent the type

The idea is to allow type to be a parameter to methods, classes and interfaces.

This way, it becomes possible to create a class that work with different data types.

Other advantages of Generics:
No need for typecasting while retrieving elements from a collection (e.g., ArrayList)

```
public class ArrayList{
private Object[] elementData;
...
public Object get(int i){.....}
public void add(Object o){.....}
}
```

```
ArrayList list = new ArrayList();
list.add("foo");
String foo = (String) list.get(0);
System.out.println(foo);
```

```
ArrayList<String> list = new ArrayList<String>();
list.add("foo");
String foo = list.get(0);
System.out.println(foo);
```

This typecasting is no more necessary

Because of Type-safety provided: We can hold only a single type of objects in generic classes. We are not allowed to store other objects in the collection.

Ensures type safety

```java
ArrayList list = new ArrayList();
list.add("foo");
list.add(5);

if (list.get(0) instanceof Integer){
    int foo0= (int) list.get(0);
    System.out.println(foo0);
}

if (list.get(0) instanceof String){
    String foo0= (String)list.get(0);
    System.out.println(foo0);
}

if (list.get(1) instanceof Integer){
    int foo1= (int) list.get(1);
    System.out.println(foo1);
}

if (list.get(1) instanceof String){
    String foo1= (String)list.get(1);
    System.out.println(foo1);
}
```

```java
ArrayList<String> list = new ArrayList<String>();
list.add("foo");
String foo = list.get(0);
System.out.println(foo);
```

We don’t need to worry about the type of the elements in a list anymore.

Compiler ensures that the correct type enters the array.

Bounded Generic Type Parameter

- You may want to restrict the types that can be used as type arguments.
- Use <T extends **UpperBound**>
- The **UpperBound** can be a class or one or more interfaces.

- <T extends **Class1 & Interface 1 & Interface 2 …**>
- <T extends **Class1**>
- <T extends **Interface 1 & Interface 2 …**>
- <T extends **Interface 1**>

Important

- T can be any class that extends the class **and** implements the interfaces.
- Class must be mentioned first, if at all it's there in the list.

```java
class Bound<T extends A>
{

    private T objRef;

    public Bound(T obj){
        this.objRef = obj;
    }

    public void doRunTest(){
        this.objRef.displayClass();
    }
}

class A
{
    public void displayClass()
    {
        System.out.println("Inside super class A");
    }
}

class B extends A
{
    public void displayClass()
    {
        System.out.println("Inside sub class B");
    }
}
```

```java
class C extends A
{
    public void displayClass()
    {
        System.out.println("Inside sub class C");
    }
}

class Main
{
    public static void main(String a[])
    {

        Bound<C> bec = new Bound<C>(new C());
        bec.doRunTest();

        Bound<B> beb = new Bound<B>(new B());
        beb.doRunTest();

        Bound<A> bea = new Bound<A>(new A());
        bea.doRunTest();

    }
}
```

Won’t be able to do the following:

```java
Bound<String> bes = new Bound<String>(new String());
bea.doRunTest();
```

```java
class Bound<T extends A&B>
{

        private T objRef;

        public Bound(T obj){
                this.objRef = obj;
        }

        public void doRunTest(){
                this.objRef.displayClass();
        }
}

class A implements B
{
        public void displayClass()
        {
                System.out.println("Inside class A");
        }
}

interface B
{
        public void displayClass();
}
```

```java
class C extends A
{
        public void displayClass()
        {
                System.out.println("Inside class C");
        }
}

class D extends A implements B
{
        public void displayClass()
        {
                System.out.println("Inside class D");
        }
}

class Main
{
        public static void main(String a[])
        {

                Bound<A> bea = new Bound<A>(new A());
                bea.doRunTest();
                Bound<C> bea1 = new Bound<C>(new C());
                bea1.doRunTest();
                Bound<D> bea2 = new Bound<D>(new D());
                bea2.doRunTest();

        }
}
```

A= self and child (can say) of B
C= child of A and grandchild of B
D= child of A and child (can say) of B

What’s exactly happening

- GTP is only available to the compiler, not the JVM.
- GTP is only available at compile time, not at run time.
- The byte code generated appears as if a generic type parameter never existed.
- Java compiler applies type erasure to do the following:

1. Replace the generic type parameter with the upper bound (if it’s bounded) or object class (if it’s unbounded).
2. Insert type casts if necessary to preserve type safety.

Behind the Scene: Generics are Implemented using Type Erasures

```
public class MyGenericList <T> {
    private ArrayList <T> myList;
    public MyGenericList() {
       myList = new ArrayList <T>();
    }
    public void add(T o) {
       myList.add(o);
    }
    public T get(int i) {
       return myList.get(i);
    }
}

public class Main {
    public static void main(Sting args[]) {
        MyGenericList<String> strList = new
                              MyGenericList<String>();
        strList.add("hello");
        String str = strList.get(0);
    }
}
```

Compile Time

```
public class MyGenericList {
    private ArrayList myList;
    public MyGenericList() {
       myList = new ArrayList ();
    }
    public void add(Object o) {
       myList.add(o);
    }
    public Object get(int i) {
       return myList.get(i);
    }
}

public class Main {
    public static void main(Sting args[]) {
       MyGenericList strList = new MyGenericList();
       strList.add("hello");
       String str = (String) strList.get(0);
    }
}
```

Generic methods

- Methods that introduce their own generic type parameters.
- They can be both static and non-static.
- In the static function, generic type parameters of class aren't accessible.
- <E> is put just before the return type, where 'E' becomes the local type parameter.
- You can use multiple generic type parameters as well by separating them with comma.

```
public static <E> void check(E r){
    System.out.println(r);

}
```

```
public static <E> E func1(E a){

System.out.println(a);

return a;
}
```

We can have more than one GTPs as well.

```java
public class GenericsTester {
   public static void main(String[] args) {
      Box<Integer, String> box = new Box<Integer, String>();
      box.add(Integer.valueOf(10),"Hello World");
      System.out.printf("Integer Value :%d\n", box.getFirst());
      System.out.printf("String Value :%s\n", box.getSecond());

      Box<String, String> box1 = new Box<String, String>();
      box1.add("Message","Hello World");
      System.out.printf("String Value :%s\n", box1.getFirst());
      System.out.printf("String Value :%s\n", box1.getSecond());
   }
}
```

```java
class Box<T, S> {
   private T t;
   private S s;

   public void add(T t, S s) {
      this.t = t;
      this.s = s;
   }

   public T getFirst() {
      return t;
   }

   public S getSecond() {
      return s;
   }
}
```

Restrictions

- Use proper Classes, not pre-defined datatypes.
- Can’t instantiate a variable of generic type parameter.
- Can’t typecast a variable of generic type parameter and use “instance of” with a variable of generic type parameters.
- Can’t have static variables of a generic type parameters. Generic type parameters of class are inaccessible inside static functions.
- Doesn’t support inheritance: Generic class of class-A doesn’t automatically become the parent of a Generic class of class-A’s child.
- Can’t instantiate an array variable of generic type parameter.

Restrictions (1/6)

- Which of the following is correct?
  1. MyGenericList <double> var = new MyGenericList<Double>();

  2. MyGenericList <Double> var = new MyGenericList<Double>();

Type Parameters Cannot Be Instantiated with Primitive Types !
– No double, only Double

Restrictions (2/6)

```
public class MyGenericClass<T> {
    ......

    public void doSomething() {
        T my_var = new T(); // ERROR
    }

    public static void main(String[] arg){
        ......
    }
}
```

- Instantiating Type variable is not allowed
  - Compile time error
  - Type erasure removes the type information and hence its impossible to figure out the type at runtime

Restrictions (3/6)

```
public class MyGenericClass<T> {
    ......

    static <T> void doSomething(ArrayList<T> list) {
        if(list instanceof ArrayList<Integer>) {

        }
    }

    public static void main(String[] arg){
        .....
    }
}
```

- Cannot use casts (based on relation of types(T)) or instanceof with parameterized types
  - Compile time error
  - Type erasure removes the type information at runtime and hence its impossible to figure out the type at runtime

```
List<Integer> li = new ArrayList<>();
List<Number>  ln = (List<Number>) li;  // compile-time error
```

Can’t do this!!

But casting is correct for the following:

```
List<String> l1 = ...;
ArrayList<String> l2 = (ArrayList<String>)l1;  // OK
```

ArrayList<T> implements List<T>, so compiler allows this typecasting.

Restrictions (4/6)

```
public class MyGenericClass<T> {
    ......
    private static T field;

    public static void main(String[] arg){
        MyGenericClass<Integer> c1 = new .....
        MyGenericClass<String> c2 = new .....
        MyGenericClass<Double> c3 = new .....
        // What is the type of “field” now ?
    }
}
```

- Cannot declare static fields whose types are Type parameter
  - If it was allowed then in the code shown here what will be the Type of “field” as it’s a static object hence shared by c1, c2 and c3, after all they all belong to the same class, `MyGenericClass`,at runtime.

Restrictions (5/6)

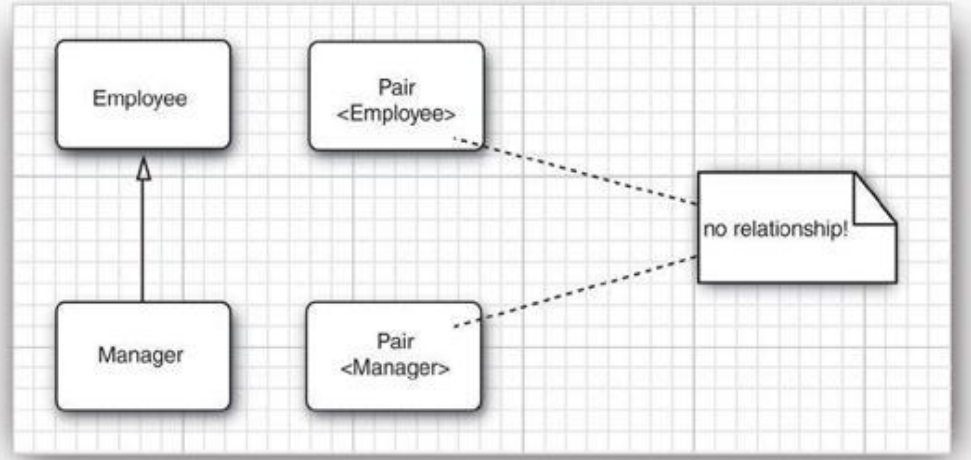


Image Source: Core Java, Volume-1

- Generic does not support sub-typing
  - If a class Employee is the superclass (parent) for a class Manager, then for a generic class Pair<T>, it does not mean Pair<Employee> also becomes the superclass (parent) for Pair<Manager>

Restrictions (6/6)

```
public class MyGenericClass<T> {
    ......

    public void doSomething() {
        T[] my_arr = new T[10]; // ERROR
    }

    public static void main(String[] arg){
        // ERROR
        MyGenericClass<String>[] str_array
                  = new MyGenericClass<String>[10];
    }
}
```

- Generic array creation is not allowed

Generic array creation: not allowed

```
List<Integer> intList[] = new ArrayList<Integer>[5];
//assume the above is allowed

List<String> stringList = new ArrayList<String>();

stringList.add("John");
```

- `Object objArray[] = intList; // This is allowed because arrays are co-variant`
- `objArray[0] = stringList;//we have assigned the list of string (i.e., List<String>) into an array that can only contain the list of integer`

- `// This will generate ClassCastException`
- `int my_int_number = (Integer)objArray[0].get(0);`

Compiler introduced typecast

NOTE: Reference-type & Object-type are different. ArrayList<T> implements List<T>

Covariant means:
(i) A parent array can be assigned the reference of child array
(ii) A parent array can accommodate child objects as its elements.

One of the problem is with arrays where a list of another type has been added as an element. For example,
[0] {hi,bye,gh}
[1] 5
[2] 6

```
public class Main {
    .....
   public static void print(ArrayList<Object> list){
        for(Object o: list)
            System.out.println(o);
    }
    public static void main(String[] arg){
        ArrayList<Integer> I = new
                    ArrayList<Integer>();
        I.add(1);
        I.add(2);
        ArrayList<String> S = new
                    ArrayList<String>();
        S.add("Bob");
        S.add("Paul");
        print(I);
        print(S);
    }
}
```

The code won't compile

Although Object is superclass for Integer and String class, it does not mean that in the `print` method, `ArrayList<Object>` can hold `ArrayList<Integer>` or `ArrayList<String>`

Restriction 5

How to resolve this issue?

The WildCard "?" to our Rescue !

```
public class Main {
    .....

   public static void print(ArrayList<?> list){

        for(Object o: list)
            System.out.println(o);
    }
    public static void main(String[] arg){
        ArrayList<Integer> I = new
                  ArrayList<Integer>();
        I.add(1);
        I.add(2);  ArrayList<String> S
        = new
                  ArrayList<String>();
        S.add("Bob");
        S.add("Paul");
        print(I);
        print(S);
    }
}
```

- We just need **one** change in our code
- Simply use a wildcard character as type variable in the parameter `ArrayList` in print method

This is like Generic<Integer>, Generic <String>, … are all children of Generic<?>

Wild Card

- ? represents an unknown type
- Generic<?> is a supertype of all parameterizations of Generic. Hence objects of any such class can be assigned to it.
- We can assign objects of Generic<A> and Generic<A's children> to Generic<? extends A> . This is called upper bounded wild card
- We can assign objects of Generic<A> and Generic<A's parents> to Generic<? super A>. This is called lower bounded wild card.

An example of using a bounded wildcard

```java
import java.util.ArrayList;
import java.util.List;

class Main {

        public static void main(String[] args) {
                List<Integer> ints = new ArrayList<Integer>();
                ints.add(3); ints.add(5); ints.add(9);

                System.out.println("Sum of ints="+sum(ints));


                List<Double> dubs = new ArrayList<Double>();
                dubs.add(3.3); dubs.add(5.5); dubs.add(9.9);

                System.out.println("Sum of ints="+sum(dubs));
        }

        public static double sum(List<? extends Number> list){
                double sum = 0;
                for(Number n : list){
                        sum += n.doubleValue();
                }
                return sum;
        }
}
```

```
Sum of ints=17.0
Sum of ints=18.7
```

Advanced Programming CSE 201

Lecture 12: Exception Handling-I

Three types of errors

- Compile-time Errors: Errors that occur when you violate the rules of writing syntax.
- Logical Errors: Errors that occur when your program compiles and executes, but does the wrong thing or returns an incorrect result or no output when it should be returning an output.
- Run-time Errors: Errors which occur during program execution(run-time) after successful compilation are called run-time errors.

Compile-time Error Example

```
class Main {

    public static void main(String args[])
    {
        int a = 40, b = 60;
        int c= 89

        String Sum = sum(a,b);
        main2(Sum);


    }

     public static int sum(int a, int b){
     x=a+b;
     if ( x < 100 ) {
        return x;
     }
     else {
     return 100;
     }
System.out.println("done!");

     }


    public static int main2(int sum){
    int s=sum+100;
    system.out.println(
            "Updated sum is "
            + S);
    }
```

Corrected code

```
class Main {

    public static void main(String args[])
    {
        int a = 40, b = 60;
        int c= 89;

        int Sum = sum(a,b);
        main2(Sum);


    }

     public static int sum(int a, int b){
     int x=a+b;
     if ( x < 100 ) {
System.out.println("done!");
        return x;

     }
     else {
 System.out.println("done!");
     return 100;

     }


     }


    public static int main2(int sum){
    int s=sum+100;
    System.out.println(
            "Updated sum is "
            + s);
    return s;
    }
}
```

Run-time Error Example 1

```
import java.util.Scanner;

class DivByZero {

    public static void main(String args[])
    {
        Scanner scn = new Scanner(System.in);

        int var1 = scn.nextInt();
        int var2 = scn.nextInt();

        int ans1 = var1 / var2;


        System.out.println("Division of va1"+ " by var2 is: "+ ans1);

    }
}
```

What if the value entered for var2 is 0?

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

Run-time Error Example 2

Null pointer exception occurs!

```
Exception in thread "main" java.lang.NullPointerException
```

```
import java.util.Scanner;

class NPEDemo {
        public static void main(String args[])
        {

         A a=new A();
         B b=a.b;
         System.out.println(b.c);
         b.disp();
        }
}

class A{
 B b;
}

class B{
 int c;
 void disp(){
  System.out.println(c);
 }
}
```

Gets initialized as Null pointer

Run-time Error Example 3

```
import java.util.Scanner;

class RTErrorDemo {
        public static void main(String args[])
        {
                int arr[] = new int[5];
                Scanner scn = new Scanner(System.in);

                System.out.println("Enter index");
                int ind=scn.nextInt();


                System.out.println("Enter value");
                int val=scn.nextInt();

                arr[ind] = val;

                System.out.println("Value assigned! ");
        }
}
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
```

What if the entered value is equal to 5 or more?

Logical Error Example

```
class LErrorDemo {
        public static void main(String[] args)
        {
                int num = 789;
                int reversednum = 0;
                int remainder;

                while (num != 0) {

                        remainder = num / 10;
                        reversednum = reversednum * 10 + remainder;
                        num /= 10;
                }
                System.out.println("Reversed number is "+ reversednum);
        }
}
```

remainder=num%100.

Neither compiler nor JVM will alert you about this error.

- A good programming practice is to handle all of them, regardless of whose mistake it is.
- The program flow shouldn’t get interrupted.
- In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

Runtime Errors

- ClassNotFoundException
- IOException
- ArthmaticException
- NullPointerException
- ArrayIndexOutOfBoundException
- NumberFormatException

```
String s="abc";
int i=Integer.parseInt(s);//NumberFormatException
```

- ....

A typical program

- Statement 1
- Statement 2//has an exception
- Statement 3
- Statement 4//crucial statement
- Statement 5

If there is a possibly that an exception can occur at statement 2, it must be handled; otherwise we won't be able to execute crucial statements coming next.

Try Block

- The try block contains set of statements where an exception can occur.
- A try block is always followed by catch blocks, a finally block or both.

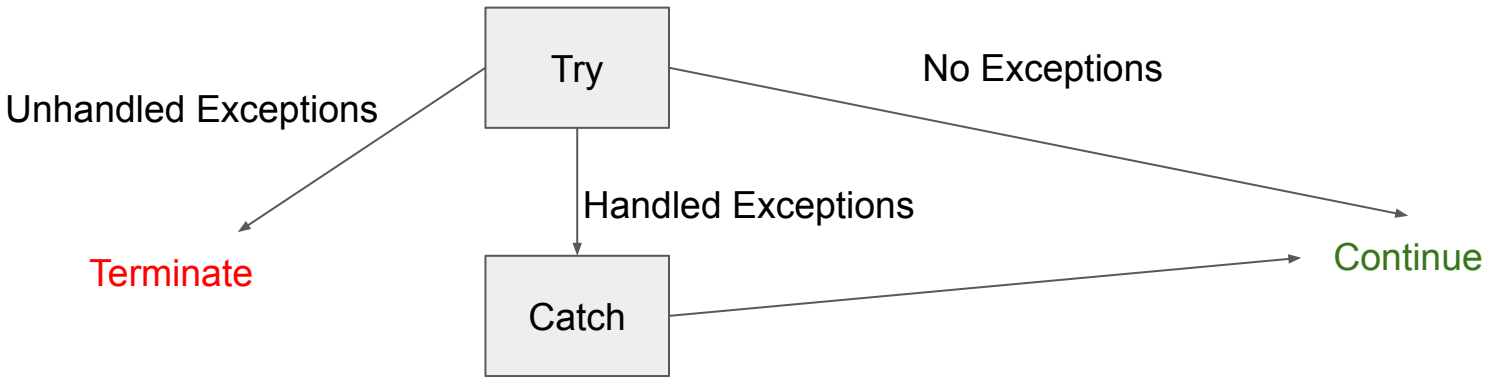
Catch Block

- A catch block is where you handle the exceptions.
- It must follow the try block.
- A single try block can have several catch blocks associated with it. You can catch different exceptions in different catch blocks.
- When an exception occurs in try block, the corresponding catch block that handles that particular exception executes.
- For example if an arithmetic exception occurs in try block then the statements enclosed in catch block for arithmetic exception executes.

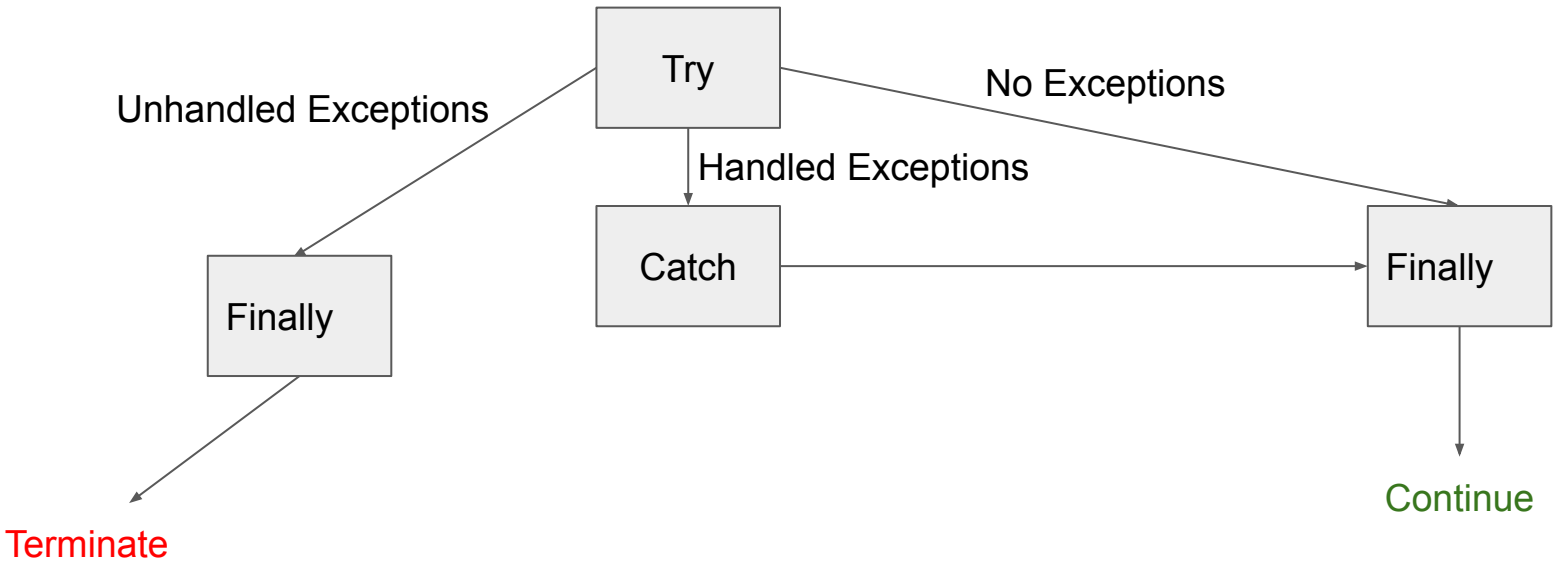
Finally Block

- The finally block in java is used to put important codes such as clean up code e.g. closing the file or closing the connection.
- If it’s present, It executes regardless of whether an exception rises or not and whether exception is handled or not.
- A finally contains all the crucial statements that must get executed regardless of whether exception occurs or not.

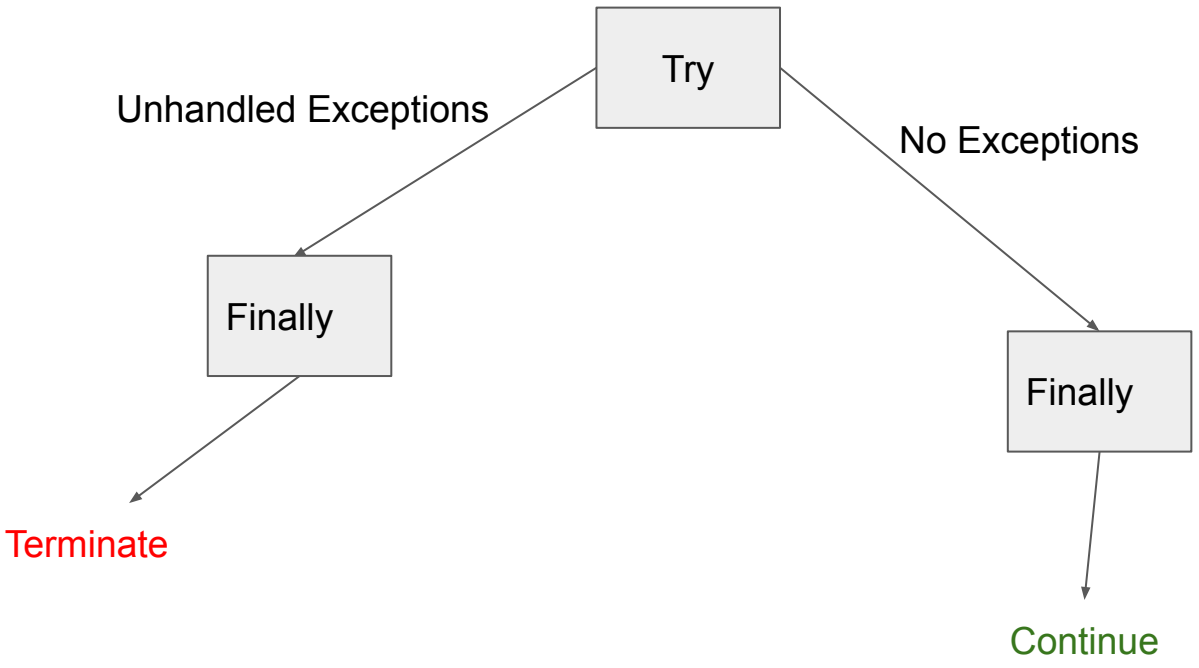
Try-Catch


Try
Unhandled Exceptions
No Exceptions
Handled Exceptions
Terminate
Catch
Continue

Try-Catch-Finally



Try-Finally


Try
Unhandled Exceptions
No Exceptions
Finally
Finally
Terminate
Continue

Try-catch example

```java
class Example1 {
   public static void main(String args[]) {
      int num1, num2;
      try {
         /* We suspect that this block of statement can throw
          * exception so we handled it by placing these statements
          * inside try and handled the exception in catch block
          */
         num1 = 0;
         num2 = 62 / num1;
         System.out.println(num2);
         System.out.println("Hey I'm at the end of try block");
      }
      catch (ArithmeticException e) {
         /* This block will only execute if any Arithmetic exception
          * occurs in try block
          */
         System.out.println("You should not divide a number by zero");
      }
      catch (Exception e) {
         /* This is a generic Exception handler which means it can handle
          * all the exceptions. This will execute if the exception is not
          * handled by previous catch blocks.
          */
         System.out.println("Exception occurred");
      }
      System.out.println("I'm out of try-catch block in Java.");
   }
}
```

You can have catch blocks for as many exceptions you want

This is to handle any exception. Should be at last. Because you may want to be specific what went wrong

Try-catch-finally example

```java
class GFG
{
    public static void main (String[] args)
    {

        // array of size 4.
        int[] arr = new int[4];

        try
        {
            int i = arr[4];

            // this statement will never execute
            // as exception is raised by above statement
            System.out.println("Inside try block");
        }

        catch(ArrayIndexOutOfBoundsException ex)
        {
            System.out.println("Exception caught in catch block");
        }

        finally
        {
            System.out.println("finally block executed");
        }

        // rest program will be executed
        System.out.println("Outside try-catch-finally clause");
    }
}
```

List of Exceptions

- CloneNotSupportedException ✓
- InterruptedException ✓
- ReflectiveOperationException ✓
  - ClassNotFoundException ✓
  - IllegalAccessException ✓
  - InstantiationException ✓
  - NoSuchFieldException ✓
  - NoSuchMethodException ✓

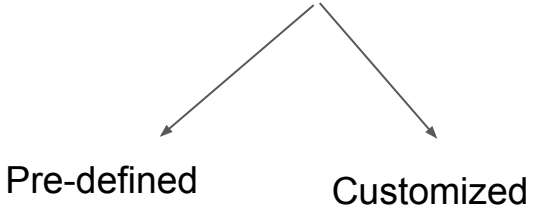
- RuntimeException
  - ArithmeticException
  - ArrayStoreException
  - ClassCastException
  - EnumConstantNotPresentException
  - IllegalArgumentException
    - IllegalThreadStateException
    - NumberFormatException
  - IllegalCallerException
  - IllegalMonitorStateException
  - IllegalStateException
  - IndexOutOfBoundsException
    - ArrayIndexOutOfBoundsException
    - StringIndexOutOfBoundsException
  - LayerInstantiationException
  - NegativeArraySizeException
  - NullPointerException
  - SecurityException
  - TypeNotPresentException
  - UnsupportedOperationException

Advanced Programming CSE 201

Lecture 13: Exception Handling - II

“throw” keyword

- It’s used to throw an exception explicitly.
- We specify the **exception** object which is to be thrown, along with an error message.
- **throw new** exception_class("error message");



In the case of customized exception, we need to create a class that extends Exception class.

The constructor of Exception class is required to be called in the constructor of this class to pass the error message, which is done using **super**.

Exception Class methods

getMessage() function outputs the error message string. You may require this in the catch block.

toString() function outputs the class name and the error message.

printStackTrace() function prints contents of toString()'s output and prints the stack trace to help the programmer locate where the exception occurred. The stack trace has locations (package name, Class Name, function name and line number) where the exception raised.

toString() output

```
java.lang.ArithmeticException Create breakpoint : The percentage value must be between 0 to 100
    at com.IIITD_AP_KR22.Main6.percentage2grades(Main6.java:20)
    at com.IIITD_AP_KR22.Main6.main(Main6.java:60)
```

Stack Trace

It starts with the function where the exception originated. Being a stack, it follows LIFO

Throwing a predefined exception

```java
import java.util.Scanner;

public class Main4 {
    static Scanner scn = new Scanner(System.in);
    public static void main(String[] args) {


        System.out.println("Input your 10th class percentage");
        try{
            double val= scn.nextDouble();

           if (val>100 | val<0){
               throw new ArithmeticException("The percentage value must be between 0 to 100");
           }
        }
        catch(Exception e){
            System.out.println(e.getMessage());
            System.out.println(e);
            e.printStackTrace();
        }
    }
}
```

Throwing a Customized Exception

```java
import java.util.Scanner;

class Percentage_Exception2 extends Exception{
    Percentage_Exception2(String msg){
        super(msg);
    }
    @Override
    public String toString() {
        return super.getMessage();
    }
}

public class Main5 {
    static Scanner scn = new Scanner(System.in);
    public static void main(String[] args) {
        System.out.println("Input your 10th class percentage");
        try{
            double val= scn.nextDouble();
           if (val>100 | val<0){
               throw new Percentage_Exception2( msg: "The percentage value must be between 0 to 100");
           }
        }
        catch(Exception e){
            System.out.println(e.getMessage());
            System.out.println(e);
            e.printStackTrace();
        }
    }
}
```

“throws” keyword

If a method is declared as function that throws an exception, any function calling it must either handle that exception or declare itself as a function that throws that exception.

```
return_type func1 () throws Exception_name {
…….
}
```

```java
class Percentage_Exception3 extends Exception{
    Percentage_Exception3(String msg){
        super(msg);
    }
    @Override
    public String toString() {
        return "Ex:"+super.getMessage();
    }
}
```

```java
public class Main6 {
    static Scanner scn = new Scanner(System.in);
    public static String percentage2grades(double p) throws Percentage_Exception3{
        if (p>100 | p<0){
            throw new Percentage_Exception3( msg: "The percentage value must be between 0 to 100");
        }
        String grade = switch ((int) (p / 10)) {
            case 10 -> "A+";
            case 9 -> "A";
            case 8 -> "B";
            case 7 -> "B-";
            case 6 -> "C";
            case 5 -> "C-";
            case 4 -> "D";
            default -> "F";
        };
        return grade;
    }
    public static void main(String[] args) {
        System.out.println("Input your 10th class percentage");
        try{
            double val= scn.nextDouble();
            System.out.println(percentage2grades(val));
        }
        catch(Exception e){
            System.out.println(e.getMessage());
            System.out.println(e);
            e.printStackTrace();
        }
    }
}
```

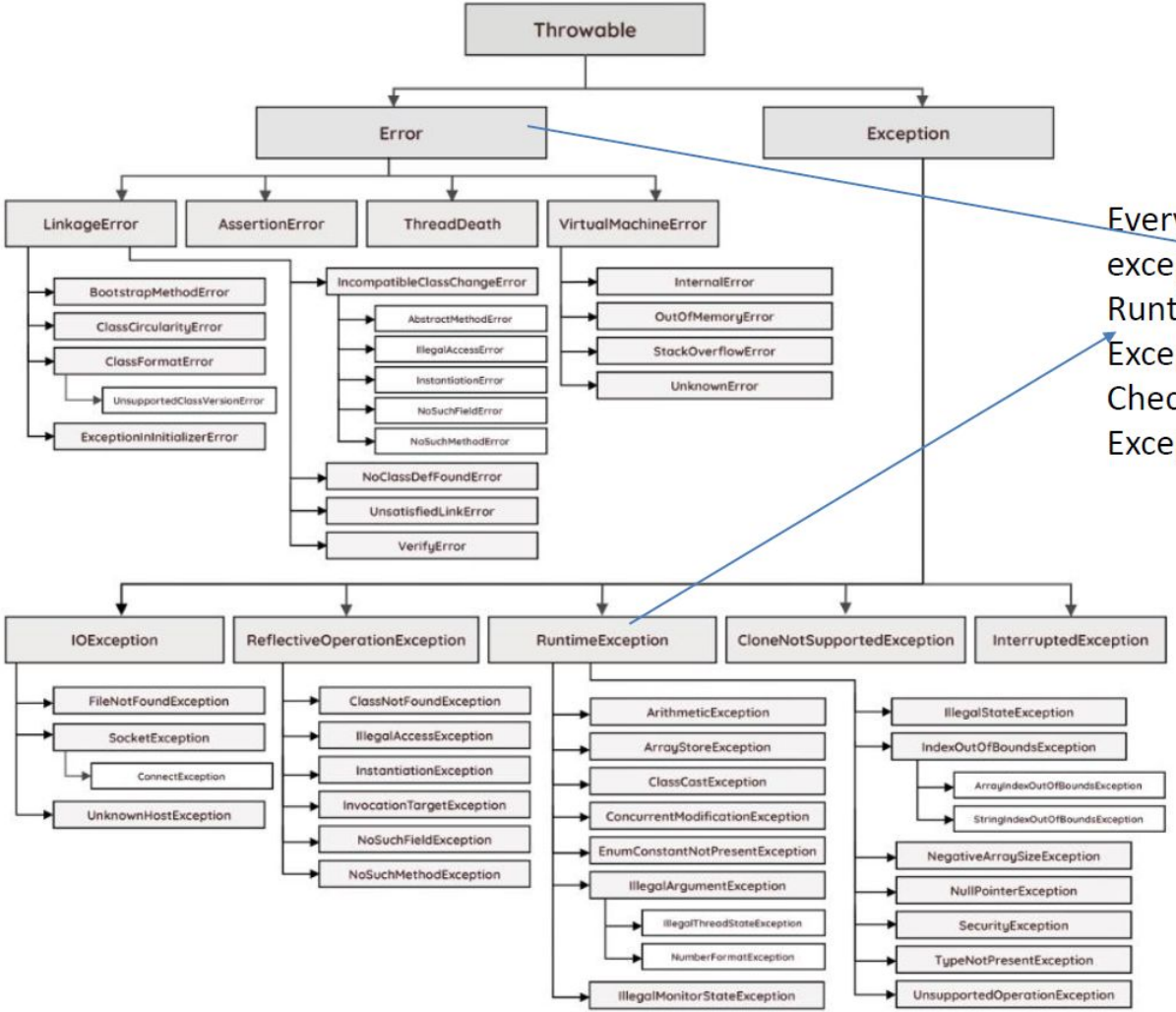
Here, percentage2grades() is the function that throws the exception

Checked vs Unchecked Exception

Although exceptions occur at run-time, some of them can be checked at compile-time itself.

These functions are called checked exceptions. The remaining exceptions are called unchecked exceptions.

If a function has a checked exception, the compiler won’t compile unless the exception is either handled or the function is declared as a function that throws the exception.



Everything except Error and Runtime Exception is a Checked Exception

Checked Exception Example

Must use throws

```java
import java.io.*;
// Main class
class Main3 {

    // Main driver method
    public static void main(String[] args) throws IOException
    {

        // Reading file from path in local directory
        FileReader file = new FileReader( fileName: "E:\\JavaExp\\hello.txt");

        // Creating object as one of ways of taking input
        BufferedReader fileInput = new BufferedReader(file);

        // Printing first 3 lines of file "C:\test\a.txt"
        for (int counter = 0; counter < 2; counter++)
            System.out.println(fileInput.readLine());

        // Closing file connections
        // using close() method
        fileInput.close();
    }
}
```

Unchecked Exception Example

Using throws is optional

```java
public class Main6 {
    static Scanner scn = new Scanner(System.in);
    public static String percentage2grades(double p){
        if (p>100 | p<0){
            throw new ArithmeticException("The percentage value must be between 0 to 100");
        }
        String grade = switch ((int) (p / 10)) {
            case 10 -> "A+";
            case 9 -> "A";
            case 8 -> "B";
            case 7 -> "B-";
            case 6 -> "C";
            case 5 -> "C-";
            case 4 -> "D";
            default -> "F";
        };
        return grade;
    }
    public static void main(String[] args) {
        System.out.println("Input your 10th class percentage");
        try{
            double val= scn.nextDouble();
            System.out.println(percentage2grades(val));
        }
        catch(Exception e){
            System.out.println(e.getMessage());
            System.out.println(e);
            e.printStackTrace();
        }
    }
}
```

Advanced Programming CSE 201

Lecture 14: Object Cloning

The **object cloning** is a way to create exact copy of an object.

The clone() method of Object class is used to clone an object.

Definition

protected Object clone() **throws** CloneNotSupportedException{

…….

…….

}

It helps us make a shallow copy (field-to-field) of the object.

```
class Student implements Cloneable{

int rollno;
String name;

Student(int rollno,String name){
this.rollno=rollno;
this.name=name;
}

Object getCopy() throws CloneNotSupportedException{
return clone();
}

}

class Main {

public static void main(String args[]) throws CloneNotSupportedException{

Student s1=new Student(101,"amit");
System.out.println(s1.rollno+" "+s1.name);

Student s2=(Student)s1.getCopy();
System.out.println(s2.rollno+" "+s2.name);

}

}
```

```
101 amit
101 amit
```

Cloneable Interface

- It’s a marker interface that allows using the Object.clone() method for making copies.
- If a class doesn’t implement it, JVM throws CloneNotSupportedException and we won’t be able to make copies of it’s objects using the Object.clone() method, as in that case.
- It acts as an indicator to the JVM that we can call the *Object.clone()* method.
- It doesn’t contain any functions.

@Override to have a clone method of itself

We have just overridden the function so that we can use the same function name, i.e. clone().

```
class Student implements Cloneable{

int rollno;
String name;

Student(int rollno,String name){
this.rollno=rollno;
this.name=name;
}

@Override
public Object clone() throws CloneNotSupportedException{
return super.clone();
}

}

class Main {

public static void main(String args[]) throws CloneNotSupportedException{

Student s1=new Student(101,"amit");
System.out.println(s1.rollno+" "+s1.name);

Student s2=(Student)s1.clone();
System.out.println(s2.rollno+" "+s2.name);

}

}
```

```
101 amit
101 amit
```

Overriding can also help us make a shift from shallow copy to deep copy

In the shallow copy, only the references of mutable members get copied.

Hence, after copying, mutable members of original object and its clone will be referring to the same object. In that case, if we make any change in one, it will reflect it other too.

In deep copy, we want the clones to be totally independent of the original objects. Thus, in addition to calling super.clone(), the developer takes the responsibility of copying mutable members properly.

Shallow Copy

```
class Student implements Cloneable{

int rollno;
String name;
Branch b;

Student(int rollno,String name,String bname){
this.rollno=rollno;
this.name=name;
this.b=new Branch(bname);
}

@Override
public Object clone() throws CloneNotSupportedException{
return super.clone();
}

}

class Branch{
String name;
Branch(String bname){
name=bname;

}
}
```

```
101 amit CSE
101 amit CSE
after changes
101 amit ECE
102 Amit Rai ECE
```

Mutable members of s1 also got changed

```
class Main {

public static void main(String args[]) throws CloneNotSupportedException{

Student s1=new Student(101,"amit","CSE");
System.out.println(s1.rollno+" "+s1.name+" "+s1.b.name);

Student s2=(Student)s1.clone();
System.out.println(s2.rollno+" "+s2.name+" "+s2.b.name);

s2.b.name="ECE";
s2.rollno=102;
s2.name="Amit Rai";

System.out.println("after changes");

System.out.println(s1.rollno+" "+s1.name+" "+s1.b.name);
System.out.println(s2.rollno+" "+s2.name+" "+s2.b.name);

}

}
```

Deep Copy

```
class Student implements Cloneable{

int rollno;
String name;
Branch b;

Student(int rollno,String name,String bname){
this.rollno=rollno;
this.name=name;
this.b=new Branch(bname);
}

@Override
public Object clone() throws CloneNotSupportedException{

Object o=super.clone();

((Student)o).b= new Branch(b.name);

return o;

}

}

class Branch{
String name;
Branch(String bname){
name=bname;

}
}
```

```
101 amit CSE
101 amit CSE
after changes
101 amit CSE
102 Amit Rai ECE
```

s1 is no more getting changed by changes in s2

```
class Main {

public static void main(String args[]) throws CloneNotSupportedException{

Student s1=new Student(101,"amit","CSE");
System.out.println(s1.rollno+" "+s1.name+" "+s1.b.name);

Student s2=(Student)s1.clone();
System.out.println(s2.rollno+" "+s2.name+" "+s2.b.name);

s2.b.name="ECE";
s2.rollno=102;
s2.name="Amit Rai";

System.out.println("after changes");

System.out.println(s1.rollno+" "+s1.name+" "+s1.b.name);
System.out.println(s2.rollno+" "+s2.name+" "+s2.b.name);

}

}
```

Using Try-catch

Removed the throws declaration on main function and caught the exception.

```
class Student implements Cloneable{

int rollno;
String name;
Branch b;

Student(int rollno,String name,String bname){
this.rollno=rollno;
this.name=name;
this.b=new Branch(bname);
}

@Override
public Object clone() throws CloneNotSupportedException{

Object o=super.clone();

((Student)o).b= new Branch(b.name);

return o;

}

}

class Branch{
String name;
Branch(String bname){
name=bname;

}
}
```

```
class Main {

public static void main(String args[]){

Student s1=new Student(101,"amit","CSE");
System.out.println(s1.rollno+" "+s1.name+" "+s1.b.name);

try{
Student s2=(Student)s1.clone();
System.out.println(s2.rollno+" "+s2.name+" "+s2.b.name);

s2.b.name="ECE";
s2.rollno=102;
s2.name="Amit Rai";

System.out.println("after changes");

System.out.println(s1.rollno+" "+s1.name+" "+s1.b.name);
System.out.println(s2.rollno+" "+s2.name+" "+s2.b.name);
}catch(CloneNotSupportedException c){

System.out.println("Exception Occurred");

}

}

}
```

If we don’t handle the exception in either way, we end up with the following error at compile time

```
L14_3.java:42: error: unreported exception CloneNotSupportedException; must be c
aught or declared to be thrown
Student s2=(Student)s1.clone();
                             ^

1 error
```

This error has occurred because of not reporting that exception can happen, not because of the occurrence of exception.

If we have handled the exception but didn’t use the cloneable marker interface for Student class, we get a run-time error with the following output

```
101 amit CSE
Exception in thread "main" java.lang.CloneNotSupportedException: Student
        at java.base/java.lang.Object.clone(Native Method)
        at Student.clone(L14_3.java:16)
        at Main.main(L14_3.java:42)
```

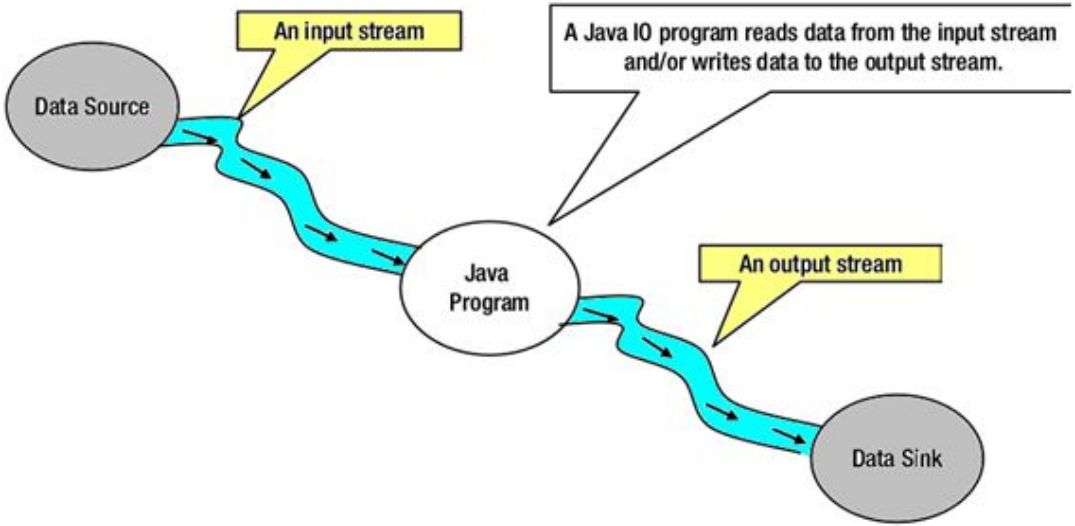
The exception has occurred at run-time because JVM didn’t find Student class is cloneable to create a copy.

Advanced Programming CSE 201

Lecture 15: I/O Streams | Serialization & Deserialization

I/O Streams

- Stream is a sequence of data
  - Flows in/out the program to/from an external source such as file, network, console, etc.
- Similar to a stream of flowing water…
- Program uses **input stream** to read data from a source, one at a time
- Program uses **output stream** to write data to a destination, one at a time



Types of Streams

- Two types of streams
  - Byte stream
  - Character stream
- Byte stream
  - Operates upon stream of “byte” ASCII code (8-bit)
- Character stream
  - Operates upon stream of “character” Unicode (16-bit)

`java.io` Package

- Reading

  open a stream

  while more information

  read information

  close the stream

- Writing

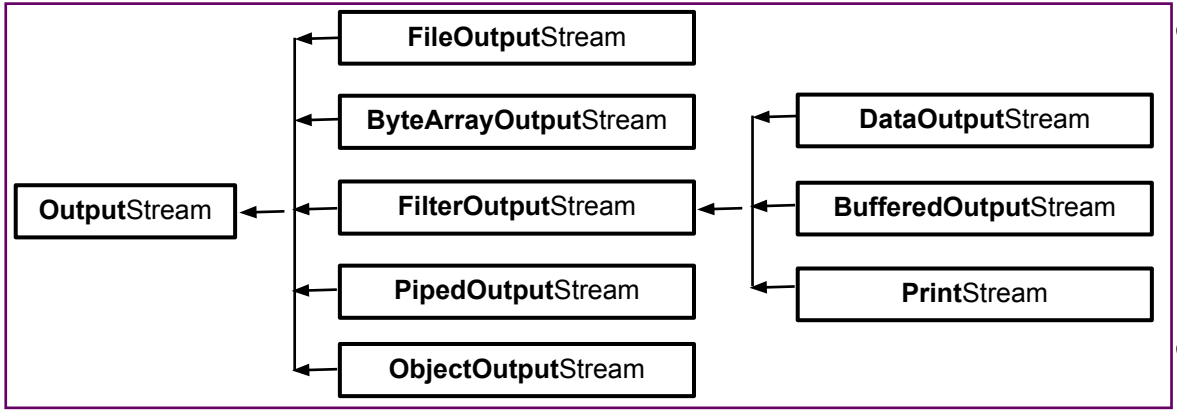
  open a stream

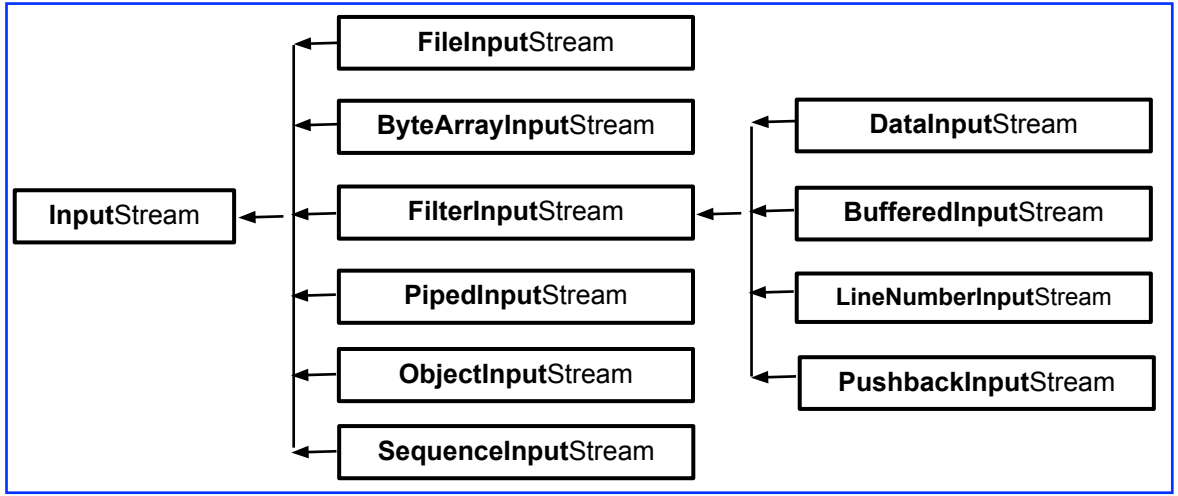
  while more information

  write information

  close the stream

Byte Stream Hierarchy





- OutputStream
  - This is the abstract class
  - Parent class of all classes representing an output stream of bytes
- InputStream
  - This is the abstract class
  - Parent class of all classes representing an input stream of bytes

Byte Streams in System Class

```
public final class System {

    public static final InputStream in;

    public static final PrintStream out;

    public static final PrintStream err;

    ...........

}
```

- In java, 3 streams are created for us automatically. All these streams are attached with console
  - **System.out:** standard output stream
  - **System.in:** standard input stream
  - **System.err:** standard error stream

```
import java.io.*;

class TakeByte {

   public static void main(String args[]) throws IOException{

         int c=System.in.read();
         System.out.print((char)c);

   }
}
```

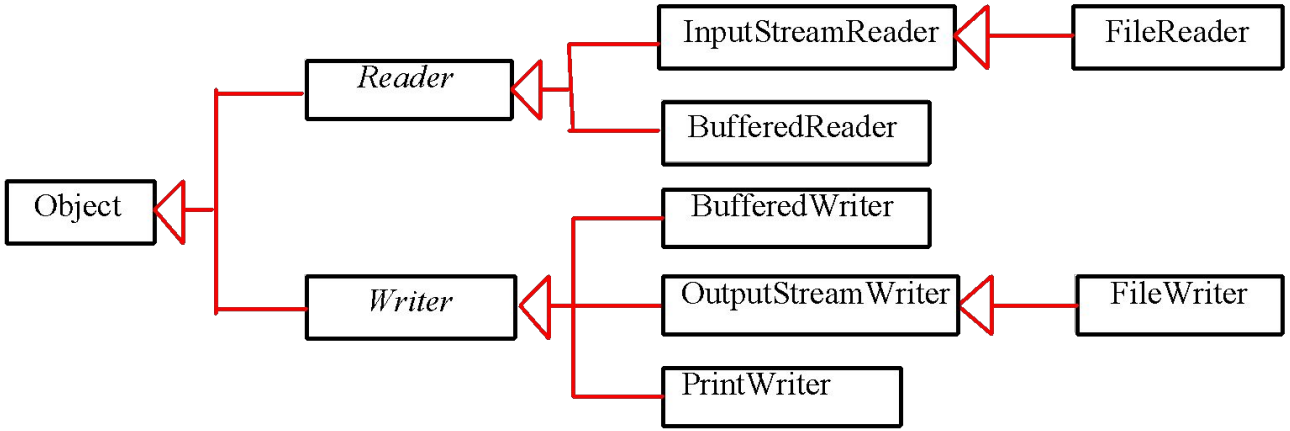
Typecasting is necessary because c contains the ASCII value

Byte Stream Example

```
import java.io.*;

class Main{
public static void main (String args[]) throws IOException
{
    FileInputStream in = null;
    FileOutputStream out = null;
    try {

        in = new FileInputStream("input.txt");
        out = new FileOutputStream("output.txt");
        int c;
        c = in.read();//reads next byte, and returns -1 if the end of stream is reached
        while (c!=-1) {

            out.write(c);
            c = in.read();

        }
    } finally {
        if (in != null)
            in.close();
        if (out != null)
            out.close();
        }
}
}
```

- InputStream
  - **`read()`** – read the next byte of data from the input stream
  - **`close()`** – close input stream
- OutputStream
  - **`write(int)`** – write a byte to current output stream
  - **`close()`** – close output stream
- Byte stream is used for low-level I/O, e.g., processing binary files

Character Stream Hierarchy



- All character stream classes are subclasses of `Reader` and `Writer` class
- Used for processing text files (character by character)

Character Stream Example

```
public static void main(String args[])
            throws IOException
{
    FileReader in = null;
    FileWriter out = null;
    try {
        // both constr. throws FileNotFoundException
        in = new FileReader("input.txt");
        // throws IOException
        out = new FileWriter("output.txt");
        int c;
        c = in.read();
        while (c != -1) {
            out.write(c);
            c = in.read();
        }
    }finally {
        if (in != null)
            in.close();
        if (out != null)
            out.close();
        }
}
```

- This example is very similar to the byte stream I/O
- In terms of coding, the difference is in using FileReader and FileWriter for input and output
- Note that “int” type variable is used in both these examples to read and write. Although internally they are working differently:
  - In byte stream example, the “int” variable holds a 8 bit value (ASCII Code)
  - In this example, the “int” variable holds a 16 bit value (UniCode)

Buffered Streams (1/2)

```
public static void main(String args[])
            throws IOException
{
    BufferedReader in = null;
    PrintWriter out = null;
    try {
        in = new BufferedReader( new
                FileReader("input.txt"));
        out = new PrintWriter( new
                FileWriter("output.txt"));
        String l;
        while ((l = in.readLine()) != null){
            out.println(l);
        }
    }finally {
        if (in != null)
            in.close();
        if (out != null)
            out.close();
        }
}
```

- Combine streams into chains to achieve more advanced input and output operations
- Reading/Writing character by character is slow
- Faster to read/write a larger block of data, such as a line.
- Helps read the entire line.
- readline function returns the next line. It returns null if there are no more lines.

© Vivek Kumar

Using Scanner Class

```
public static void main(String args[]) {
    Scanner in = new Scanner(System.in); //java.lang
    // Scanner class implements iterator
    while (in.hasNext()) {
        System.out.println(in.next());
    }
    in.close();
}
```

hasNext() returns true if this scanner has another token in its input.

next() finds and returns the next complete token from this scanner and returns it as a string; a token is usually ended by whitespace such as a blank or line break. If no token exists, NoSuchElementException is thrown.

Buffered Streams (2/2)

```
public static void main(String args[])
            throws IOException
{
    Scanner in = null;
    PrintWriter out = null;
    try {
        in = new Scanner( new BufferedReader( new
                FileReader("input.txt")));
        out = new PrintWriter( new
                FileWriter("output.txt"));
        while (in.hasNext()) {
            out.println(in.next());
        }
    }finally {
        if (in != null)
            in.close();
        if (out != null)
            out.close();
        }
}
```

- Here we are combining three classes for breaking input into tokens:
  - Scanner
  - BufferedReader
  - FileReader
- BufferedReader will read one line at a time
- Scanner will be able to parse this line by white space separated tokens

Serialization and Deserialization

- Serialization in Java is a mechanism of writing the state of an **object** into a **byte stream**
- The reverse operation is called deserialization
- Some usage
  - Storing live objects in a file
  - Hibernating applications
  - Moving object state over the network

`java.io.Serializable` Interface

```
package java.io;
public interface Serializable { /*empty*/ }
```

- Must be implemented by the class to be serialized
- This is a tag/marker interface similar to Cloneable interface

© Vivek Kumar

Example: Serializing and Deserializing

```
1. import java.io.*;
2.
3. class Manager implements Serializable {
4.     private String name;
5.     public Manager(String n) { ..... }
6. }

7. public class Main {
8.    public static void serialize()
9.                          throws IOException {
10.        Manager s1 = new Manager("Amy");
11.        ObjectOutputStream out = null;
12.        try {
13.           out = new ObjectOutputStream (
14.                 new FileOutputStream("out.txt"));
15.           out.writeObject(s1);
16.        } finally {
17.           out.close();
18.        }
19.    }
20.
/* Continued on RHS window */
```

```
    /* Continued from LHS window */
21.    public static void deserialize()
22.       throws IOException, ClassNotFoundException {
23.        ObjectInputStream in = null;
24.        try {
25.            in =  new ObjectInputStream (
26.                   new FileInputStream("out.txt"));
27.            Manager s1 = (Manager) in.readObject();
29.        } finally {
30.            in.close();
31.        }
32.    }
33.
34.    public static void main(String[] args)
35.        throws IOException,ClassNotFoundException {
36.
37.       serialize();
39.       deserialize();
40.    }
41. } /* End of Main class */
```

```java
import java.io.*;
class Address {
    private String city;
    public Address(String c) { ...... }
}
class Manager implements Serializable {
     private String name;
     private Address addr;
     public Manager(String n, String city) { ..... }
}
public class Main {
  public static void serialize() throws IOException {
        Manager s1 = new Manager("Amy”, “Delhi”);
        ObjectOutputStream out = null;
        try {
           out = new ObjectOutputStream (
                 new FileOutputStream("out.txt"));
           out.writeObject(s1);
        } finally {
           out.close();
        }
    }
    ......
}
```

- This program compiles fine but will generate `NotSerializableException`
- All fields of Manager class should either be primitive type or serializable objects
  - Address class should also implement Serializable interface

- There is no point in serializing static field members in a class
  - Static fields do not represents object state but they represent class state
  - There will not be any compilation/runtime issue, although the value serialized will not make any sense as it can always be updated later in the class.
- `transient` keyword in Java
  - If you don't want any field to be serialized then mark that as "`transient`".
  - When JVM comes across **transient** keyword, it ignores original value of the variable and save default value of that variable data type.

```java
import java.io.*;
class Test implements Serializable
{
    // Normal variables
    int i = 10, j = 20;

    // Transient variables
    transient int k = 30;

}

class Main{

    public static void main(String[] args) throws Exception
    {
        Test input = new Test();

        // serialization
        FileOutputStream fos = new FileOutputStream("output.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(input);

        // de-serialization
        FileInputStream fis = new FileInputStream("output.txt");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Test output = (Test)ois.readObject();
        System.out.println("i = " + output.i);
        System.out.println("j = " + output.j);
        System.out.println("k = " + output.k);
    }

}
```

```
i = 10
j = 20
k = 0
```

Advanced Programming CSE 201

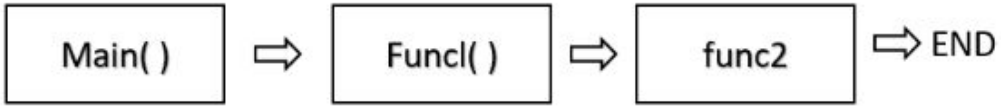
Lecture 16: Multithreading

Multithreading

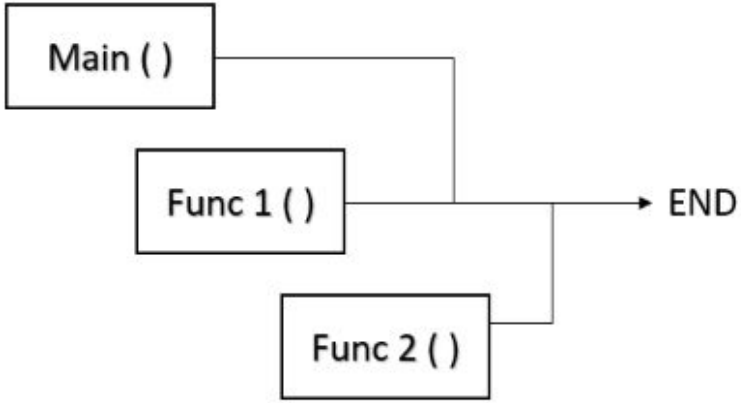
- Used to maximize the CPU utilization.
- We don't want our CPU to be in a free state; for example, Func1() is running and demands any input/output process. The CPU unnecessarily wait until Func1() completes its input/output operation.
- While Func1() completes its I/O operation, the CPU is free and not executing any thread. The efficiency of the CPU is decreased in the absence of multithreading.
- In the case of multithreading, if a thread demands any I/O operation, then the CPU will let the thread perform its I/O operation and will start the execution of a new thread. In this case, two threads are executing at the same time. If there is a waiting time in the new thread, the original thread can be reactivated.

Flow of control in java

- Without threading :



- With threading :



Creating a Threading

There are two ways to create a thread in java

1. By extending **Thread** class
2. By implementing **Runnable** interface

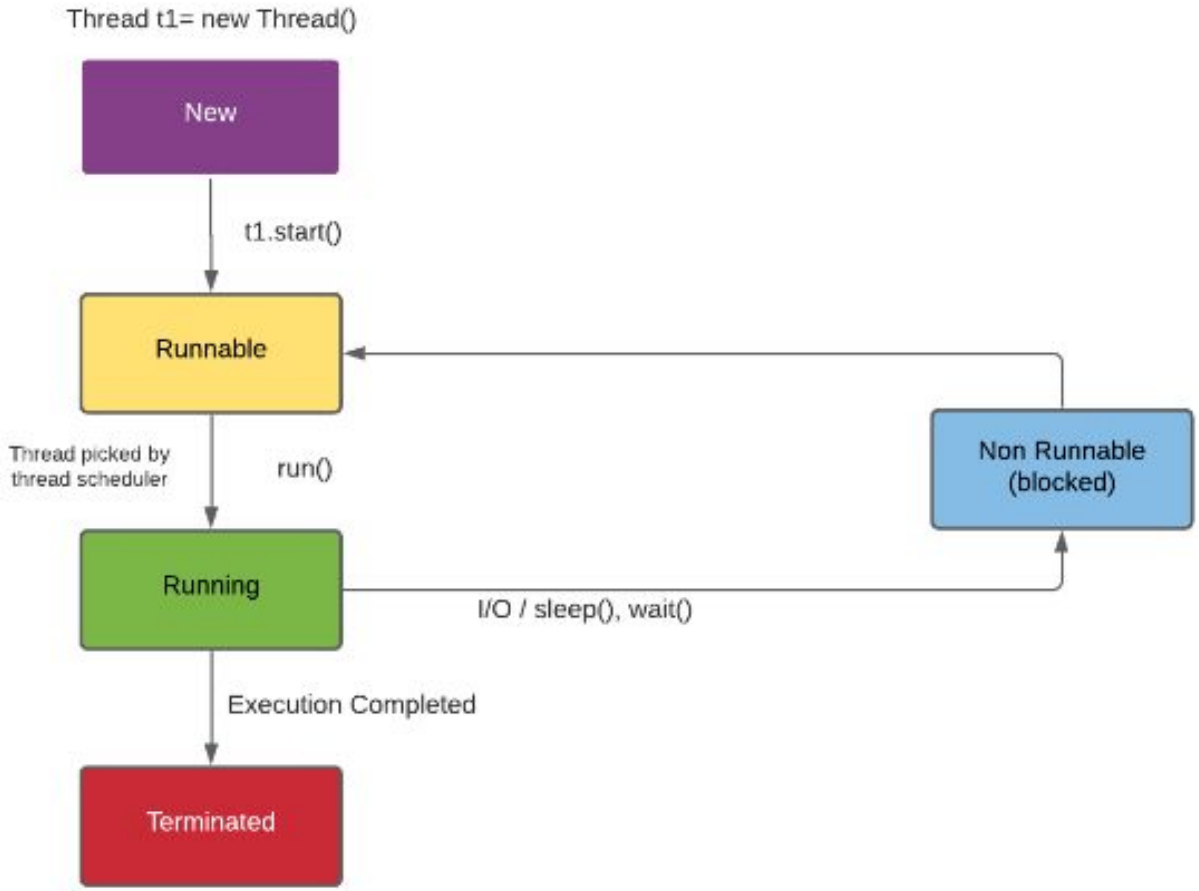
By Extending Thread Class

```
class MyThread1 extends Thread{
    @Override
    public void run(){
        int i =0;
        while(i<40000){
            System.out.println("111111111111");
            i++;
        }
    }
}

class MyThread2 extends Thread{
    @Override
    public void run(){
        int i =0;
        while(i<40000){
            System.out.println("2222222222222222");
            i++;
        }
    }
}

class thclass {
    public static void main(String[] args) {
    MyThread1 t1 = new MyThread1();
    MyThread2 t2 = new MyThread2();
    t1.start();
    t2.start();
    }
}
```

```
class Runnable1 implements Runnable{
    public void run(){


        int i =0;
        while(i<40000){
        System.out.println("1111111111111111");
        i++;
        }

    }
}
class Runnable2 implements Runnable{
    public void run(){
        int i =0;
        while(i<40000){
        System.out.println("2222222222222222222222222222");
        i++;
        }

    }
}

class rin {
    public static void main(String[] args) {
        Runnable1 r1 = new Runnable1();
        Thread t1 = new Thread(r1);

        Runnable2 r2 = new Runnable2();
        Thread t2 = new Thread(r2);

        t1.start();
        t2.start();
    }
}
```

By Implementing Runnable Interface

Lifecycle of a thread



1. New - Instance of a thread is created, which is not yet started by invoking `start()`. In this state, the thread is also known as the born thread.
2. Runnable - After invocation of `start()` & before it is selected to be run by the scheduler.
3. Running - After the thread scheduler has selected it.
4. Non-runnable - thread alive, not eligible to run.
5. Terminated - `run()` method has exited.

Types of constructors

1. Thread ( )
2. Thread (string)
3. Thread (Runnable r)
4. Thread (Runnable r, String name)

```
class MyThr extends Thread{
    public MyThr(String name){
        super(name);
    }
    public void run(){

        System.out.println("The name of the thread t is " + this.getName());

    }
}



class thcons {
    public static void main(String[] args) {
    MyThr t1 = new MyThr("Name1");
    MyThr t2 = new MyThr("Name2");
    t1.start();
    t2.start();


    }
}
```

In a Multithreading environment, all the threads which are ready and waiting to be executed are present in the Ready queue. The thread scheduler is responsible for assigning the processor to a thread. But the question is on what basis the thread scheduler decides that a particular thread will run before other threads. Here comes the concept of priority in the picture.

- Every single thread created in Java has some priority associated with it.
- The programmer can explicitly assign the priority to the thread. Otherwise, JVM automatically assigns priority while creating the thread.
- In Java, we can specify the priority of each thread relative to other threads. Those threads having higher priorities get greater access to the available resources than lower priorities threads.
- Thread scheduler will use priorities while allocating processor.

- The valid range of thread priorities is 1 to 10 (but not 0 to 10), where 1 is the least priority, and 10 is the higher priority.
- If there is more than one thread of the same priority in the queue, then the thread scheduler picks any one of them to execute.
- The following static final integer constants are defined in the Thread class:
  - MIN_PRIORITY: Minimum priority that a thread can have. Value is 1.
  - NORM_PRIORITY: This is the default priority automatically assigned by JVM to a thread if a programmer does not explicitly set the priority of that thread. Value is 5.
  - MAX_PRIORITY: Maximum priority that a thread can have. Value is 10.

You can set priority using any number between 1 and 10, not just these constants.

```
class MyThr1 extends Thread{
    public MyThr1(String name){
        super(name);
    }
    public void run(){


        while(true){

            System.out.println("Thank you: " + this.getName());

        }

    }
}

class thprio {
    public static void main(String[] args) {

        MyThr1 t1 = new MyThr1("H1");
        MyThr1 t2 = new MyThr1("H2");
        MyThr1 t3 = new MyThr1("H3");
        MyThr1 t4 = new MyThr1("H4");
        MyThr1 t5 = new MyThr1("H5 (most Important)");
        t5.setPriority(Thread.MAX_PRIORITY);
        t1.setPriority(Thread.MIN_PRIORITY);
        t2.setPriority(Thread.MIN_PRIORITY);
        t3.setPriority(Thread.MIN_PRIORITY);
        t4.setPriority(Thread.MIN_PRIORITY);
        t5.setPriority(Thread.MIN_PRIORITY);

        t1.start();
        t2.start();
        t3.start();
        t4.start();
        t5.start();

    }
}
```

Introducing wait time though Thread.sleep().

T2 started right in the first iteration in t1 because of the wait time introduced. Here, 1000 means 1000 ms, i.e. 1 sec.

```
th11
th22
th22
th22
th22
th22
th22
th22
th22
th22
th22
th11
th11
th11
th11
th11
th11
th11
th11
th11
th11
```

```
class runnable1 implements Runnable{

public void run(){
int i=0;
while (i<10){
System.out.println(Thread.currentThread().getName());
try{
Thread.sleep(1000);
}catch(Exception e){
        System.out.println(e);
}
i++;
}


}

}

class runnable2 implements Runnable{

public void run(){
int i=0;
while (i<10){
System.out.println(Thread.currentThread().getName());
i++;
}
}

}


class Main{
public static void main(String args[]) {
runnable1 r1=new runnable1();
runnable2 r2=new runnable2();

Thread t1=new Thread(r1,"th11");

Thread t2=new Thread(r2,"th22");


t1.start();

t2.start();

}
}
```

join()

Forcing current thread to wait until a particular thread completes its job using join function.

In this case the main thread is the current thread and t1 is the particular thread.

Therefore, despite long waits in t1, t2 couldn't start, as main thread was waiting for t1 to finish before t2.start() statement can be reached.

```
class runnable1 implements Runnable{

public void run(){
int i=0;
while (i<10){
System.out.println(Thread.currentThread().getName());
try{
Thread.sleep(1000);
}catch(Exception e){
        System.out.println(e);
}
i++;
}


}

}

class runnable2 implements Runnable{

public void run(){
int i=0;
while (i<10){
System.out.println(Thread.currentThread().getName());
i++;
}
}

}


class Main{
public static void main(String args[]) throws InterruptedException
{
runnable1 r1=new runnable1();
runnable2 r2=new runnable2();

Thread t1=new Thread(r1,"th11");

Thread t2=new Thread(r2,"th22");


t1.start();

t1.join();

t2.start();

}
}
```

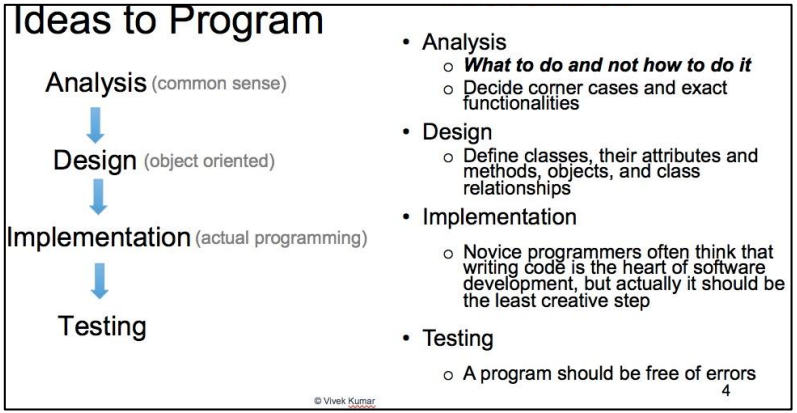
```
th11
th11
th11
th11
th11
th11
th11
th11
th11
th11
th11
th22
th22
th22
th22
th22
th22
th22
th22
th22
th22
```

Advanced Programming CSE 201

Lecture 17: UML

What is UML?

- UML stands for Unified Modeling Language
- It's a widely used modeling language in the field of software engineering
- It's used to analyze, design, and implement software-based systems
- Pretty pictures (diagrams)



© Vivek Kumar

Motivations for UML

- We need a modeling language to:
  - help develop efficient, effective and correct designs, particularly Object Oriented designs
  - communicate clearly with project stakeholders (concerned parties: developers, customer, etc)
  - give us the “big picture” view of the project

© Vivek Kumar

UML Diagrams

Three types of UML diagrams:

1. **Class diagrams:** show the classes in a system, attributes, and operations of each class and the relationship between each class.
2. **Use case diagrams:** give a graphic overview of the actors involved in a system, different functions needed by those actors and how these different functions interact.
3. **Sequence diagrams:** show how objects interact with each other and the order of those interactions.

© Vivek Kumar

UML Diagrams: Class Diagrams

A class diagram in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's:

- classes,
- their attributes,
- operations (or methods),
- class relationships.

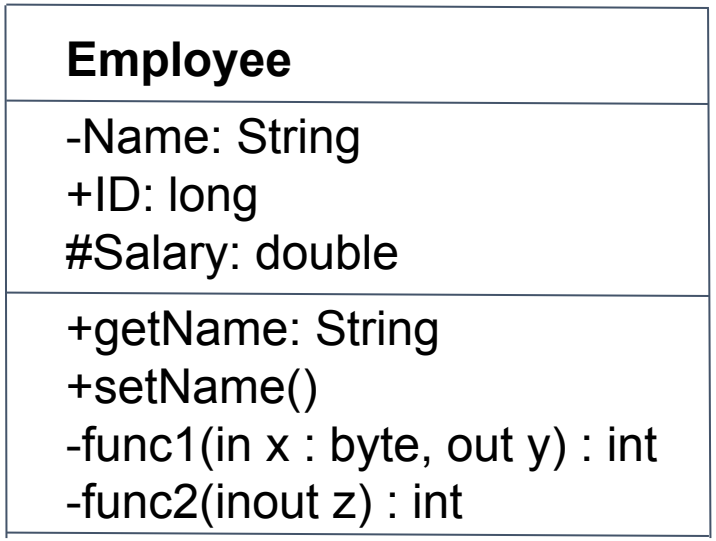
© Vivek Kumar

UML Class Notation

- A class is a rectangle divided into three parts
  - Class name
  - Class attributes
  - Class operations

- Modifiers
  - Private: -
  - Public: +
  - Protected: #
  - Static: Underlined

- Abstract class/methods
  - Name in italics

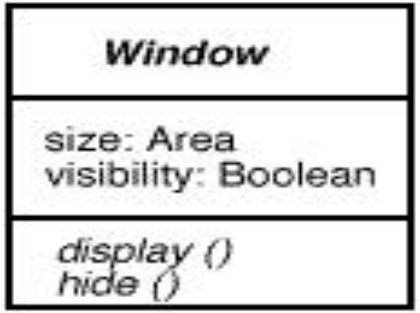


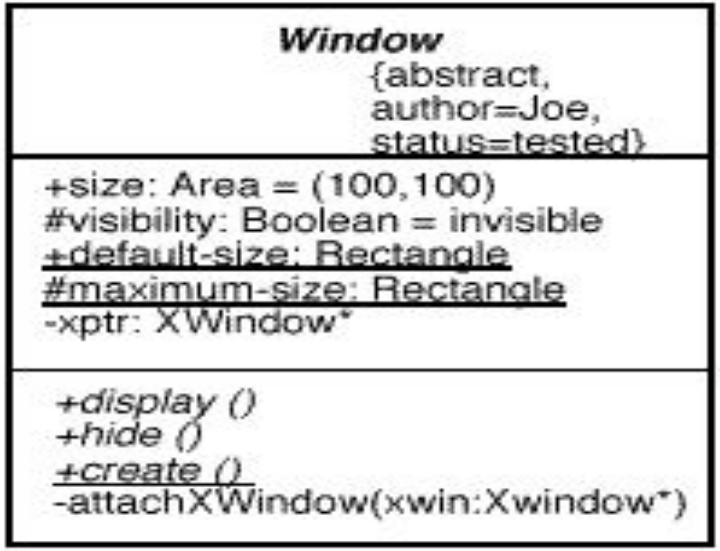
© Vivek Kumar

Different Levels of Specifying Classes

Conceptual

Specification





Use this for your project

Implementation

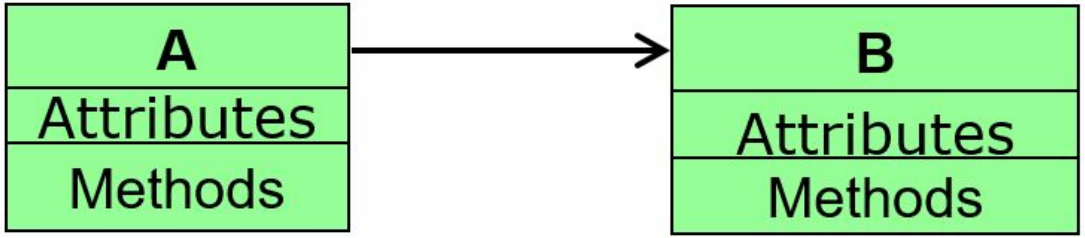
© Vivek Kumar

Composition in UML

- Represented by a solid arrow with diamond head
- In below UML diagram, A is composed of B

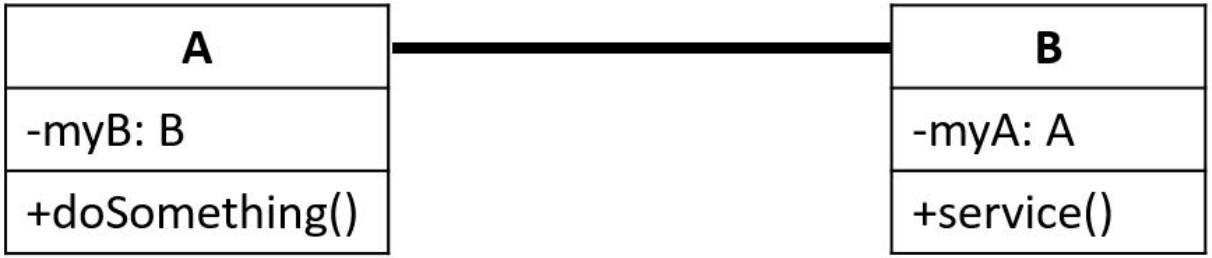
Association in UML

- Represented by a solid arrow
- In below UML diagram, A holds a reference of B



Class Relationship: Binary Association

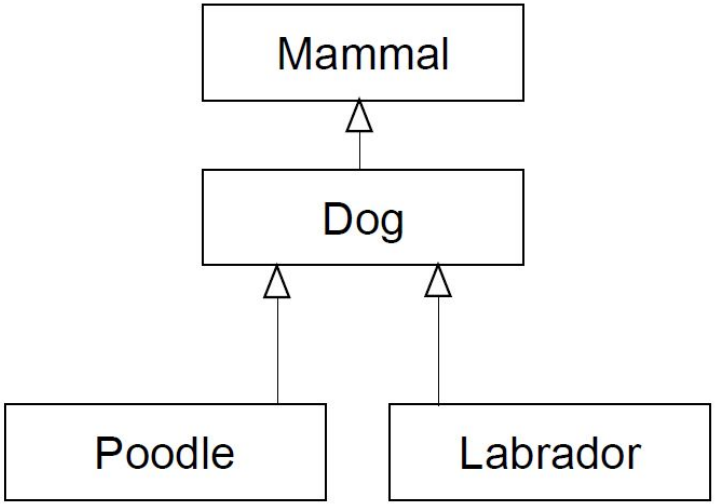
Both entities “Knows About” each other (two-way association)



Dependency in UML

- Represented by a dashed arrow starting from the dependent class to its dependency
  - A is dependent on B
  - A is requesting service from B

Inheritance in UML



UML Multiplicities

Links on associations to specify more details about the relationship

Multiplicities	**Meaning**
0..1	zero or one instance. The notation "***n . . M***" indicates ***n*** to ***m*** instances.
0..* *or* *	no limit on the number of instances (including none).
1	exactly one instance
1..*	at least one instance

Multiplicity example

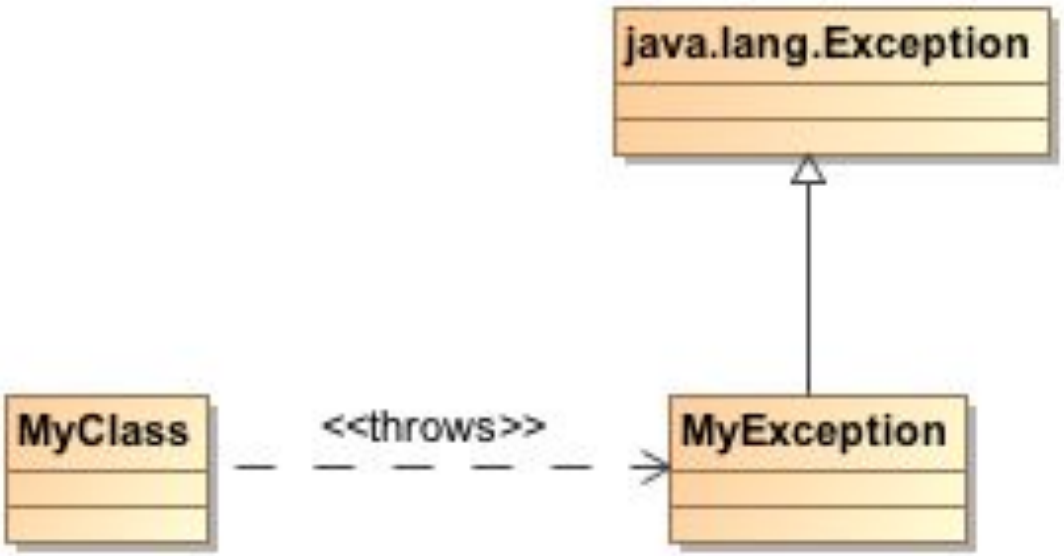
A customer can have 1 to 5 bank accounts.

A bank account can belong to one or two customers (in case of joint account)

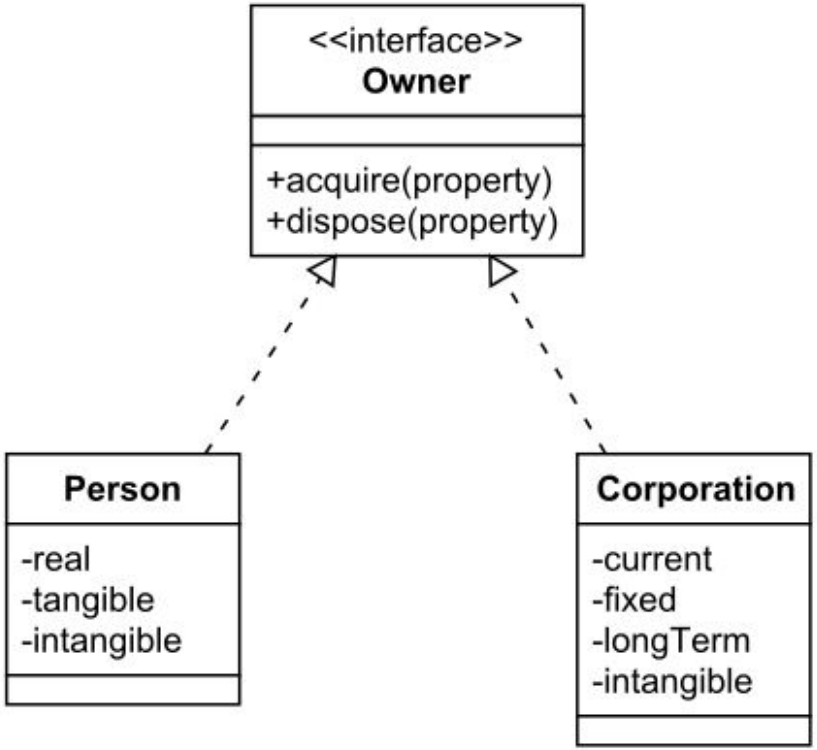
Final Multiplicity

Both multiplicities combined in a single diagram

Exceptions

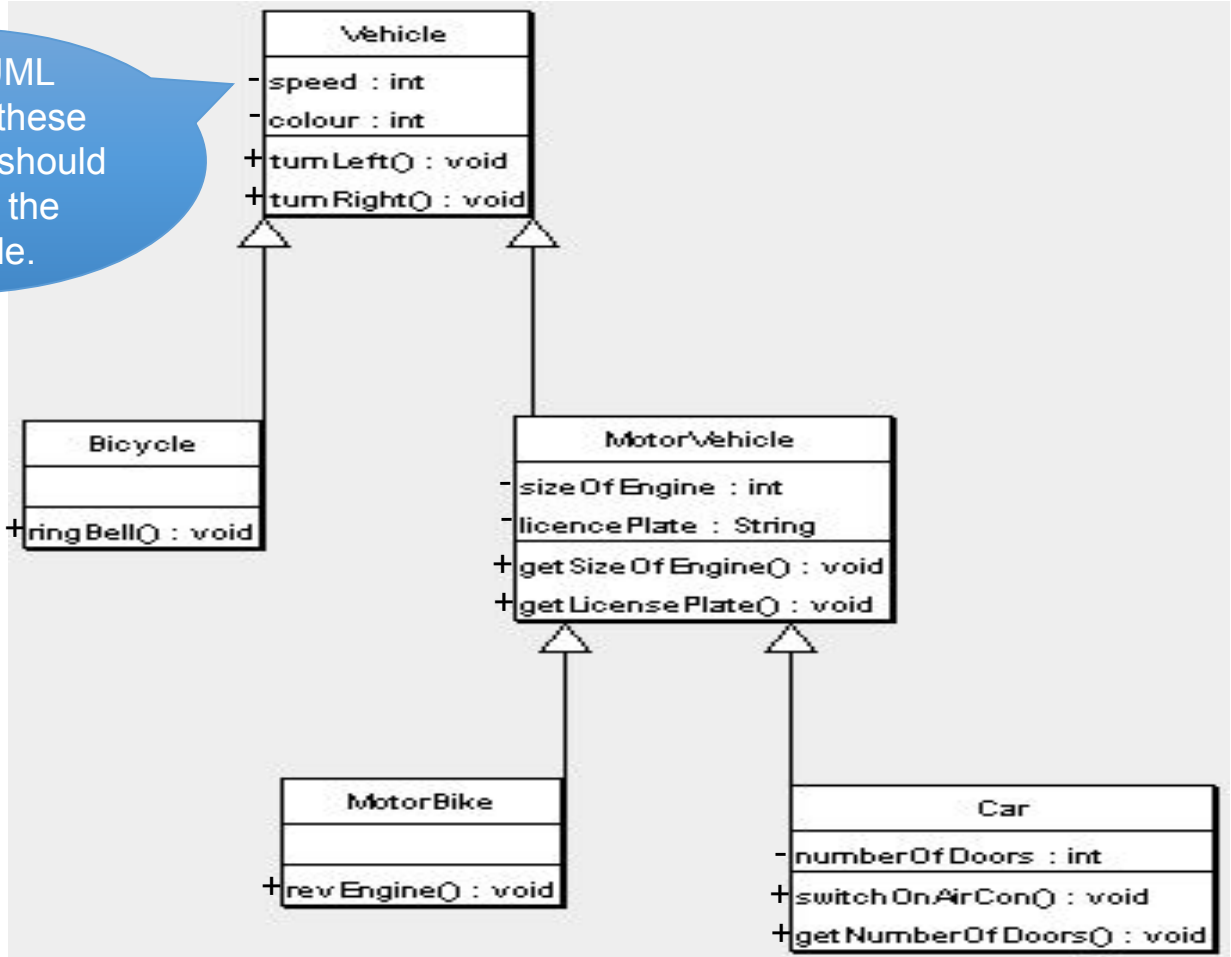


Interfaces

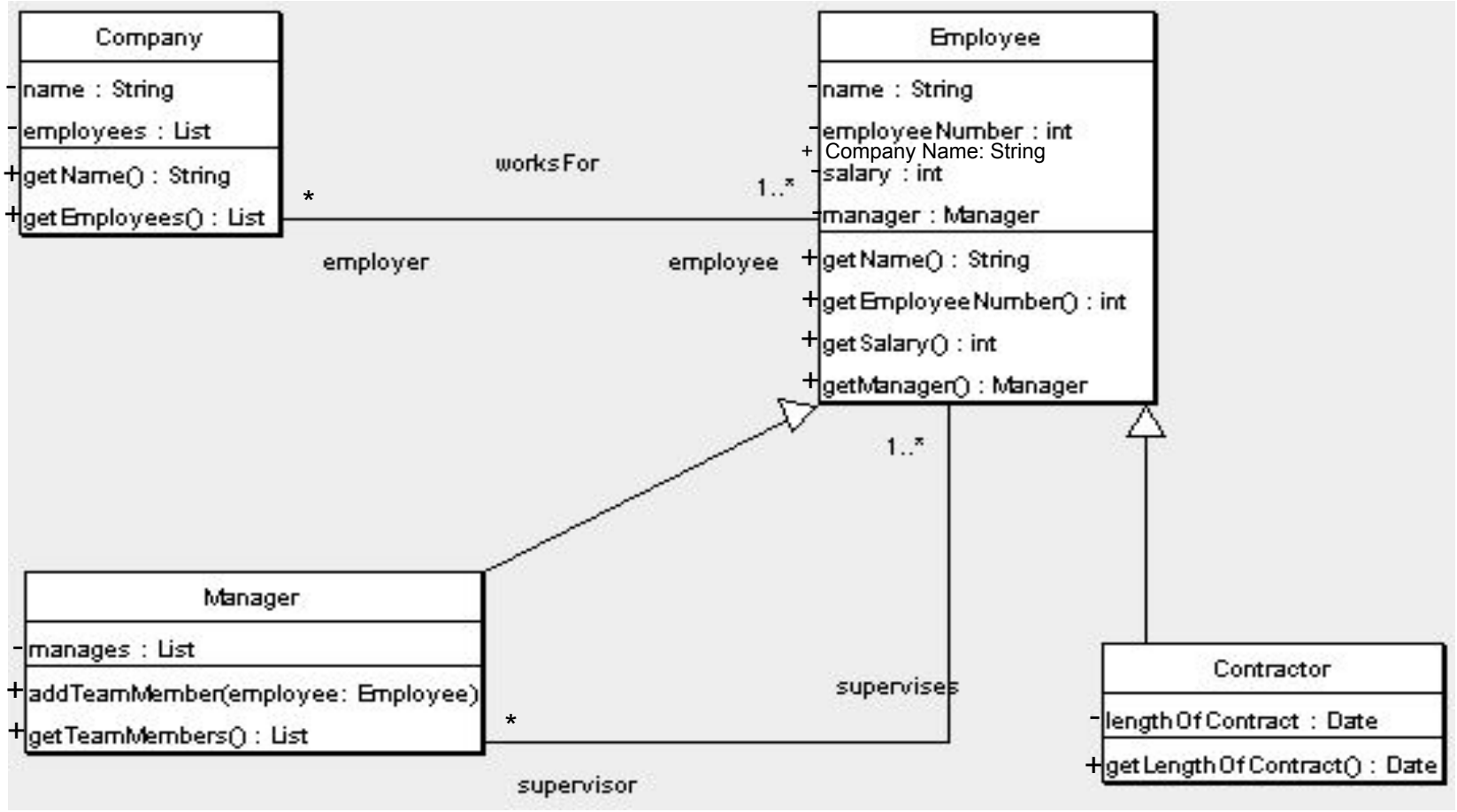

<<interface>>
Owner
+acquire(property)
+dispose(property)
Person
-real
-tangible
-intangible
Corporation
-current
-fixed
-longTerm
-intangible

Sample Class Diagram (1/2)

In your UML diagrams, these "+", "-", etc, should be inside the rectangle.



Sample Class Diagram (2/2)



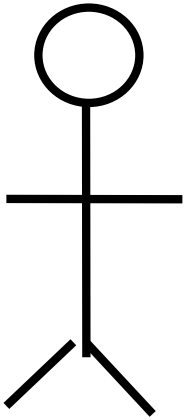
UML Diagrams: Use Cases

- Means of capturing requirements
  - Used at a very early phase of software development for requirement gathering (analysis phase)
  - Provides a high level overview of the system
  - Class diagrams are created after generating use case diagrams
- Document interactions between user(s) and the system
  - User (actor) is not part of the system itself
  - But an actor can be *another* system
- **Use case diagrams** describe what a system does from the standpoint of an external observer. The emphasis is on *what* a system does rather than *how*

© Vivek Kumar

Actors in Use Case

- What is an Actor?
  - A user or outside system that interacts with the system being designed in order to obtain some value from that interaction
  - It can be a:
    - Human
    - Peripheral device (hardware)
    - External system or subsystem
    - Time or time-based event
  - Labelled using a descriptive noun or phrase
  - Represented by stick figure



Use Case Analysis (1/4)

- Sample scenario
  - *“A patient calls the clinic to make an appointment for a yearly checkup. The receptionist finds the nearest empty time slot in the appointment book and schedules the appointment for that time slot"*

- We want to write a use case for this scenario

Source: http://www.cs.fsu.edu/~baker/swe1/restricted/notes/ppt/UseCaseDiagrams.ppt

Use Case Analysis (2/4)

- Sample scenario
  - *“A patient calls the clinic to make an appointment for a yearly checkup. The receptionist finds the nearest empty time slot in the appointment book and schedules the appointment for that time slot”*

- Who is the actor?
  - The actor is a “Patient” here

Source: http://www.cs.fsu.edu/~baker/swe1/restricted/notes/ppt/UseCaseDiagrams.ppt

Use Case Analysis (3/4)

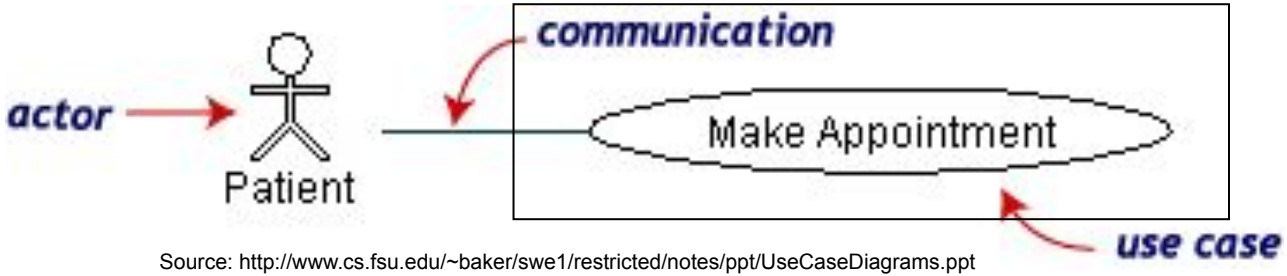
- Sample scenario
  - "A *patient calls the clinic to make an appointment for a yearly checkup. The receptionist finds the nearest empty time slot in the appointment book and schedules the appointment for that time slot"*

- A use case describes a function that a system performs to achieve the user's goal.
- A use case must yield an observable result that is of value to the user of the system.
  - So, what is the use case here?
  - The use case is "Make Appointment"

Source: http://www.cs.fsu.edu/~baker/swe1/restricted/notes/ppt/UseCaseDiagrams.ppt

Use Case Analysis (4/4)

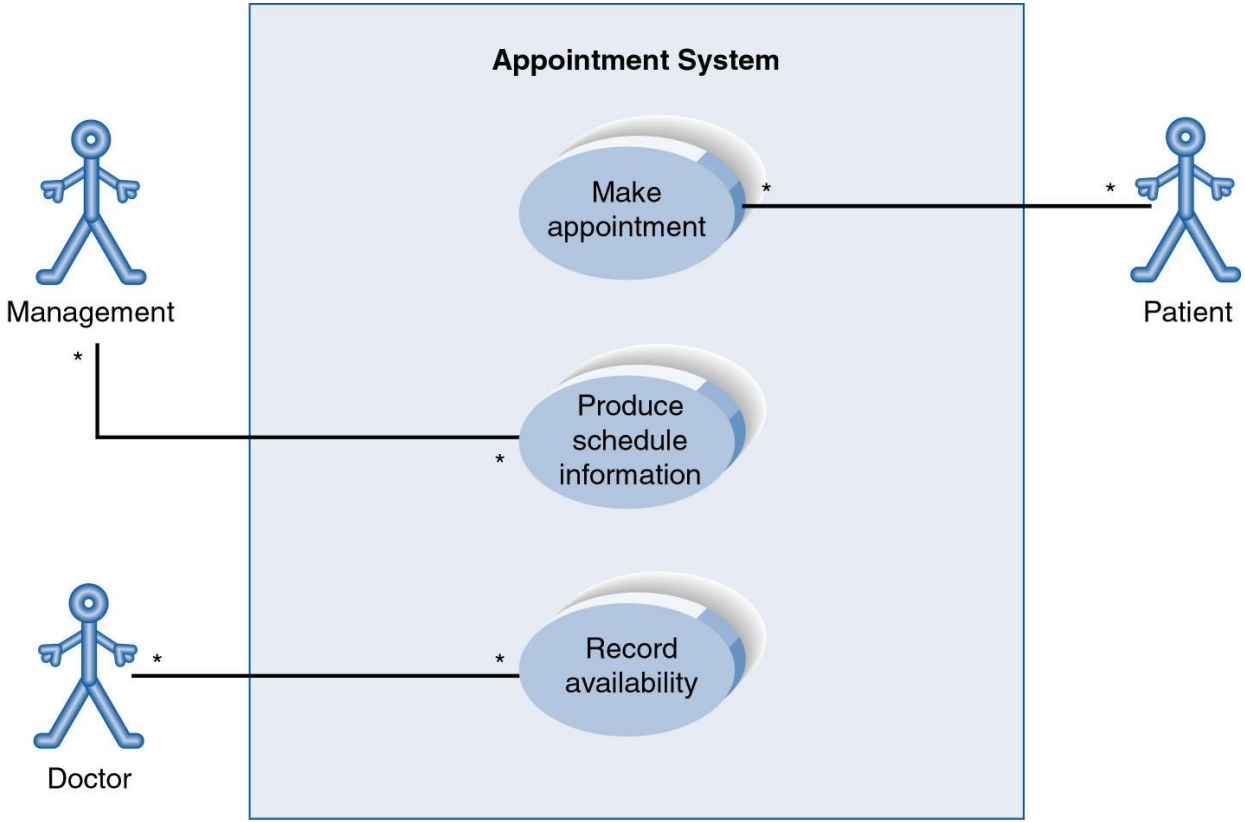
- The picture below is a **Make Appointment** use case for the medical clinic.
- The actor is a **Patient**. The connection between actor and use case is a **communication**
- Actors are stick figures
- Use cases are ovals
  - Labelled using a descriptive verb-noun phrase
- Communications are lines that link actors to use cases
- Boundary rectangle is placed around the perimeter of the system to show how the actors communicate with the system



Source: http://www.cs.fsu.edu/~baker/swe1/restricted/notes/ppt/UseCaseDiagrams.ppt

Use Case Diagram

- A use case diagram is a collection of actors, use cases, and their communications



Source: http://www.cs.fsu.edu/~baker/swe1/restricted/notes/ppt/UseCaseDiagrams.ppt

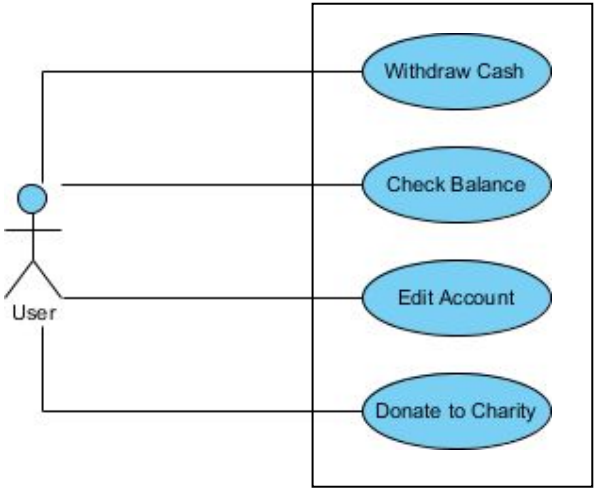
Relationships for Use Cases

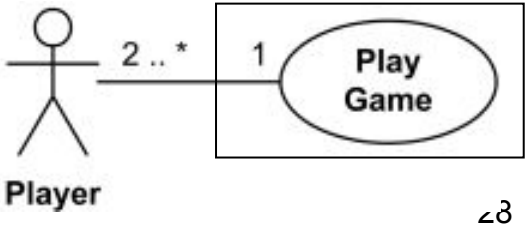
- Association
- Generalization
- Extend
- Include

© Vivek Kumar

Association Relationship

- Exists only between an actor and a use case
  - Indicates that an actor can use certain functionality of the system
- Represented by a solid line without arrowhead
  - Most commonly used representation
  - Uncommon to show one-way association
- The association between an actor and a use case can also show multiplicity at each end

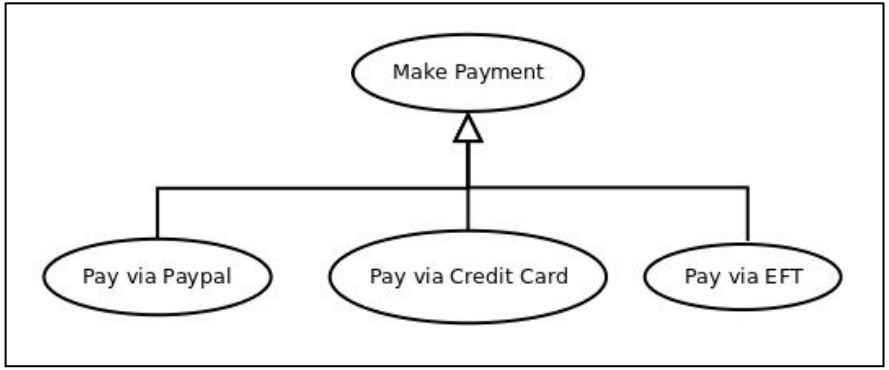


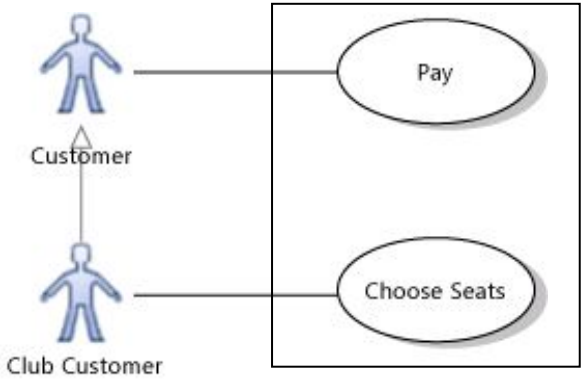


© Vivek Kumar

Generalization Relationship

- Could exist between two actors or between two use cases
  - Indicates parent/child relationship
- Represented by a solid line with a triangular and hollow arrowhead
  - From child to parent

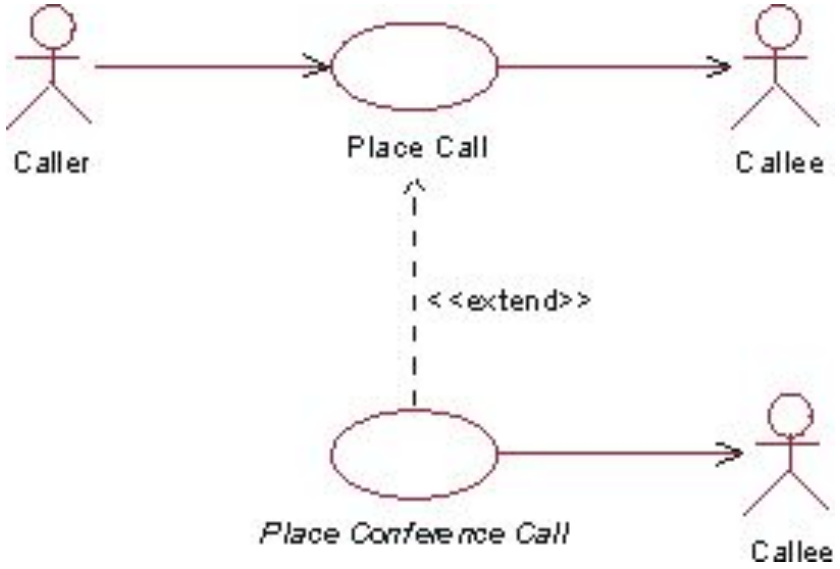




© Vivek Kumar

Extend Relationship “<<extend>”

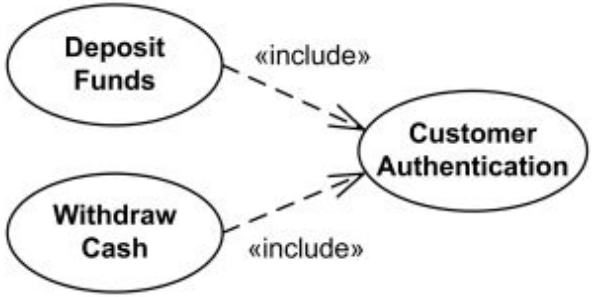
- Exists only between use cases
  - This relationships represent optional or seldom invoked cases
  - Indicates that although one use case is a variation of another but it is invoked rarely
    - Lot of shared code between these use cases **(not to be confused with inheritance)**
- Represented using a dashed arrow with an arrowhead. The notation “<< extend >>” is also mentioned above the arrow
  - The direction of the arrow is toward the extended use cases

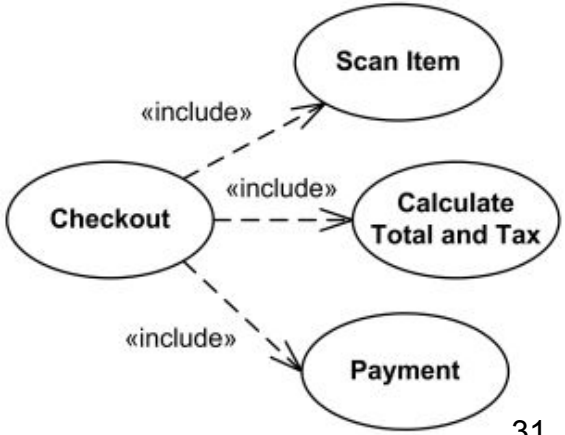


© Vivek Kumar

Include Relationship “<<include>”

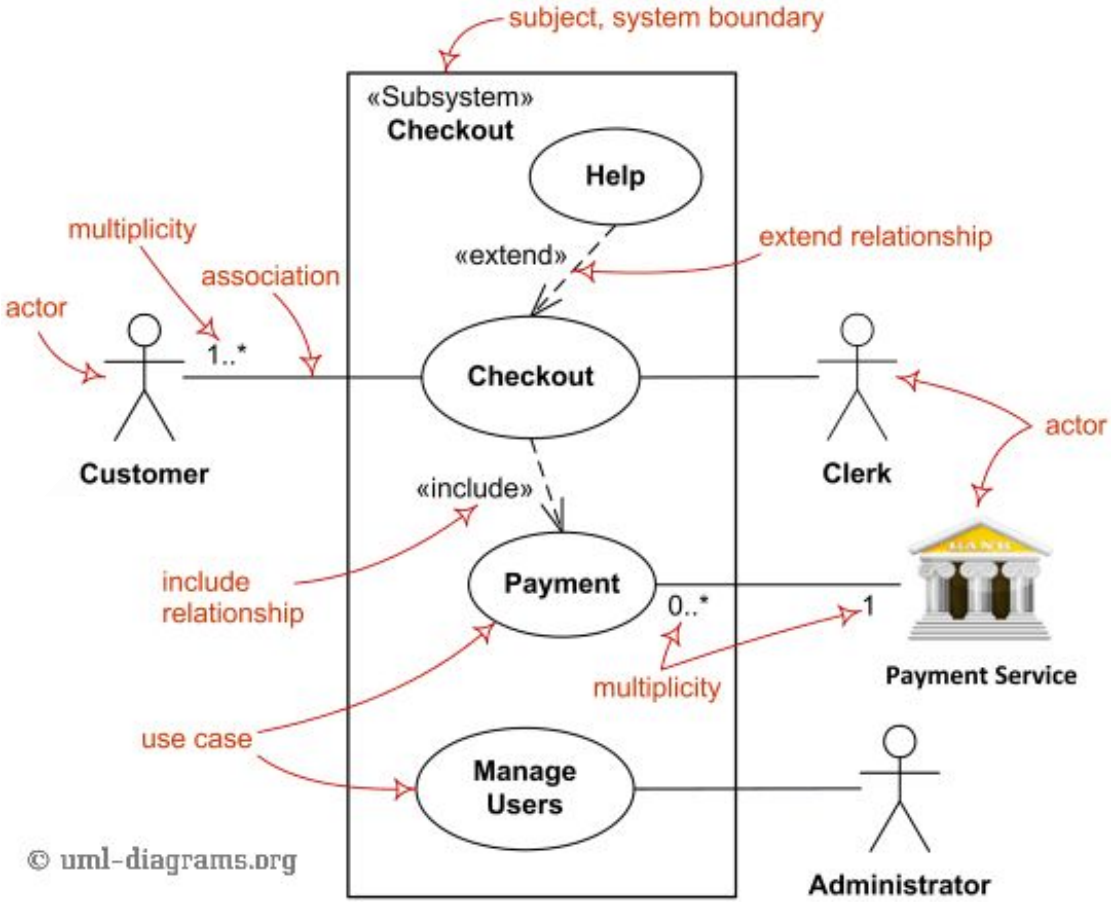
- Exists only between use cases
  - Represents behavior that is factored out of the use case
  - Doesn’t mean that the factored out use case is an optional or seldom invoked cases
- Represented using a dashed arrow with an arrowhead. The notation “<< include>>” is also mentioned above the arrow
  - The direction of the arrow is toward the included use case

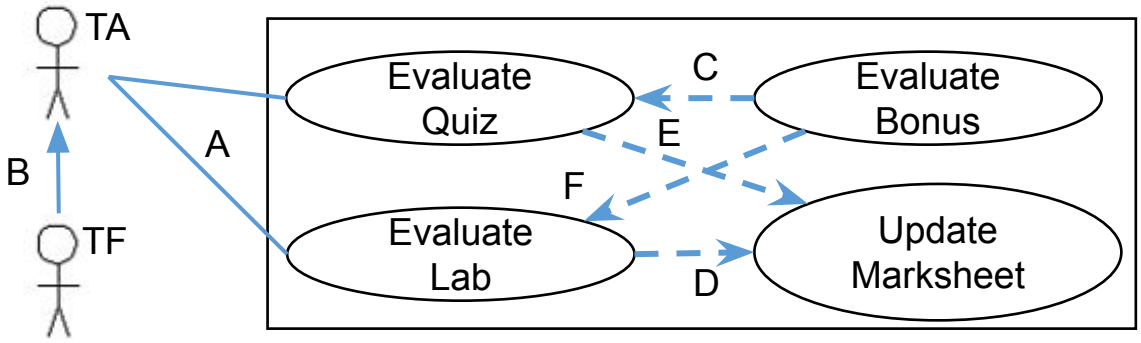




© Vivek Kumar

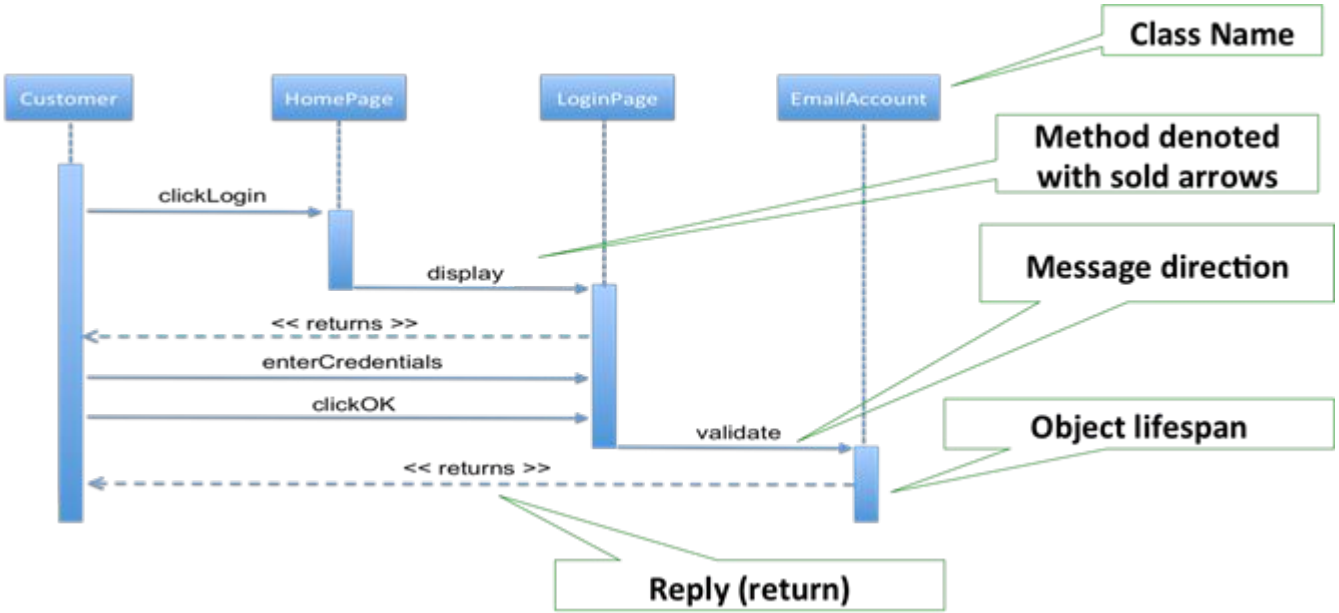
Sample Use Case




TA
TF
A
B
C
D
E
F
Evaluate Quiz
Evaluate Bonus
Evaluate Lab
Update Marksheet

Sequence Diagrams

A sequence diagram simply depicts interaction between objects in a sequential order i.e. the order in which these interactions take place.



Advanced Programming CSE 201

Lecture 18: JUnit-2

Sample JUnit Test

```java
/* The class method to be tested */
public class Sum {
    private int var1, var2;
    public Sum(int v1, int v2) {var1=v1; var2=v2;}
    public int sum () {
        return var1 + var2;
    }
}
```

```java
/* Junit test class */

import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class MyTest {

    @Test
    public void testSum() {
        Sum mySum = new Sum(1, 1);
        int sum = mySum.sum();
        assertEquals(2, sum);
    }
}
```

```java
/* Junit test runner class */

import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
   public static void main(String[] args) {
      Result result= JUnitCore.runClasses(MyTest.class);
      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }
      System.out.println(result.wasSuccessful());
   }
}
```


© Vivek Kumar

JUnit Assertion Methods

`assertTrue(`**test**`)`	fails if the boolean test is `false`
`assertFalse(`**test**`)`	fails if the boolean test is `true`
`assertEquals(`**expected**`,` **actual**`)`	fails if the values are not equal
`assertSame(`**expected**`,` **actual**`)`	fails if the values are not the same (by ==)
`assertNotSame(`**expected**`,` **actual**`)`	fails if the values *are* the same (by ==)
`assertNull(`**value**`)`	fails if the given value is *not* `null`
`assertNotNull(`**value**`)`	fails if the given value is `null`
`fail()`	causes current test to immediately fail

- Each method can also be passed a string to display if it fails:
  - e.g. assertEquals("**message**", **expected**, **actual**)
- Detailed description: https://junit.org/junit4/javadoc/4.8/org/junit/Assert.html

What is Wrong?

```java
/* The class method to be tested */
public class Sum {
    private int var1, var2;
    public Sum(int v1, int v2) {var1=v1; var2=v2;}
    public void incr () {
        var1++; var2++;
    }
}
```

```java
/* Junit test class */
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class MyTest {

    @Test
    public void testIncr() {
        Sum mySum = new Sum(1, 1);
        mySum.incr();
        Sum expected = new Sum(2, 2);
        assertEquals(expected, mySum);
    }
}
```

- We are passing two objects of `Sum` type into the `testIncr()` method where the `assertEquals` checks for equality
  - Missing `equals()` method in Sum !
  - No compilation/runtime error but test will fail

© Vivek Kumar

What's Still Wrong?

```java
/* The class method to be tested */
public class Sum {
    private int var1, var2;
    public Sum(int v1, int v2) {var1=v1; var2=v2;}
    public void incr () {
        var1++; var2++;
    }

    @Override
    public boolean equals(Object o) {
        if(o!=null && getClass()==o.getClass()) {
            Sum s = (Sum) o;
            return ((var1==s.var1)&&(var2==s.var2));
        }
        return false;
    }
}
```

```java
/* Junit test class */
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class MyTest {

    @Test
    public void testIncr() {
        Sum mySum = new Sum(1, 1);
        mySum.incr();
        Sum expected = new Sum(3, 3);
        assertEquals(expected, mySum); //should fail
    }
}
```

testIncr(MyTest): expected:<Sum@2e817b38> but was:<Sum@c4437c4>

- Missing `toString()` method!!

© Vivek Kumar

The Correct Version

```
/* The class method to be tested */
public class Sum {
    private int var1, var2;
    public Sum(int v1, int v2) {var1=v1; var2=v2;}
    public void incr () {
        var1++; var2++;
    }

    @Override
    public boolean equals(Object o) {
        if(o!=null && getClass()==o.getClass()) {
            Sum s = (Sum) o;
            return ((var1==s.var1)&&(var2==s.var2));
        }
        return false;
    }

    @Override
    public String toString() {
       return "("+Integer.toString(var1)+",”
                      +Integer.toString(var2)+")";
    }
}
```

```
/* Junit test class */
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class MyTest {

    @Test
    public void testIncr() {
        Sum mySum = new Sum(1, 1);
        mySum.incr();
        Sum expected = new Sum(3, 3);
        assertEquals(expected, mySum); //should fail
    }
}
```

testIncr(MyTest):
expected:<(3,3)> but was:<(2,2)>

© Vivek Kumar

Tests With a Timeout

```
@Test(timeout = 5000)
public void name() { ... }
```

- The above method will be considered a failure if it doesn't finish running within 5000 ms

```
private static final int TIMEOUT = 2000;
...
@Test(timeout = TIMEOUT)
public void name() { ... }
```

- Times out / fails after 2000 ms

Testing for Exceptions

```
@Test(expected = ExceptionType.class)
 public void name() {
     ...
 }
```

- Will pass if it *does* throw the given exception.
  - If the exception is *not* thrown, the test fails
  - Use this to test for expected errors

```
@Test(expected = ArrayIndexOutOfBoundsException.class)
public void testBadIndex() {
    ArrayIntList list = new ArrayIntList();
    list.get(4);   // should fail
}
```

Setup and Teardown

```
@Before
public void name() { ... }
@After
public void name() { ... }
```

- Methods to run before/after **each test case** method is called [NOTE: not any test case].
- You need to define it only once, but they will be called along with every test case method.

```
@BeforeClass
public static void name() { ... }
@AfterClass
public static void name() { ... }
```

- Methods to **run once** before/after the entire test class runs

Tips for Testing

- You cannot test every possible input, parameter value, etc.
  - So you must think of a limited set of tests likely to expose bugs.
- Think about boundary cases
  - positive; zero; negative numbers
  - right at the edge of an array or collection's size
- Think about empty cases and error cases
  - 0, -1, null; an empty list or array
- test behavior in combination
  - maybe add() usually works, but fails after you call remove()
  - make multiple calls; maybe size() fails the second time only

Trustworthy Tests

- Test one thing at a time per test method
  - 10 small tests are much better than 1 test 10x as large

- Each test method should have few (likely 1) assert statements
  - If you assert many things, the first that fails stops the test
  - You won't know whether a later assertion would have failed

- Tests should avoid logic.
  - minimize if/else, loops, switch, etc
  - avoid try/catch
    - If it's supposed to throw, use expected= ... if not, let JUnit catch it

Advanced Programming CSE 201

Lecture 18: Inner Class & JUnit-1

Phone Batteries are Becoming Non-Removable

Non Removable Battery Vs. Removable Battery

- Nowadays most of the phones are coming with non removable batteries
- Earlier, when there were removable batteries, we could easily keep a spare battery and replace when the primary one drained!
  - *Yes, I know we can use a power bank today!*

© Vivek Kumar

Inner Class

```
public class SamsungGalaxy {

    private FixedBattery myBattery;

    public SamsungGalaxy() {
        myBattery = new FixedBattery();
    }


    private class FixedBattery {
        ....
    }
}
```


© Vivek Kumar

Inner Class

- Description
  - Class defined in scope of another class
- Property
  - Can directly access all variables & methods of enclosing class (including private fields & methods)
- Why inner class?
  - Logical grouping of functionality
  - Increases encapsulation
    - Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world
  - More readability and maintainable code

© Vivek Kumar

Inner Class Instance Inside a Method of Outer Class

```
public class SamsungGalaxy {

    private FixedBattery myBattery;

    public SamsungGalaxy() {
        myBattery = new FixedBattery();
    }


    private class FixedBattery {
        private boolean runDiagnosis() { ..... }
        ....
    }

    public static void main(String[] args) {
        SamsungGalaxy sg = new SamsungGalaxy();
        SamsungGalaxy.FixedBattery sgb
           = sg.new FixedBattery();
        boolean test = sgb.runDiagnosis();
    }
}
```

- To instantiate the inner class in some other class, first we have to instantiate the outer class
- Thereafter, using the object of the outer class, we can instantiate the inner class
  - Note the usage of “new” keyword

© Vivek Kumar

Inner Class Instance Outside Outer Class

```
public class SamsungGalaxy {

    private FixedBattery myBattery;

    public SamsungGalaxy() {
        myBattery = new FixedBattery();
    }


    private class FixedBattery {
        private boolean runDiagnosis() { ..... }
        ....
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        SamsungGalaxy sg = new SamsungGalaxy();
        SamsungGalaxy.FixedBattery sgb
           = sg.new FixedBattery();
        boolean test = sgb.runDiagnosis();
    }
}
```

- Is this code correct?
  - NO, compilation error!
- Inner class FixedBattery is private and hence cannot be accessed in another class
  - Rules of “private” modifier
  - Making the inner class and its method as public will correct this code

© Vivek Kumar

```
class SamsungGalaxy {
    private int version=2;
    private FixedBattery myBattery;

    public SamsungGalaxy() {
        myBattery = new FixedBattery();
    }

    public class FixedBattery {
        private int version=3;
        public void print() {
            int version = 4;
            System.out.println(version);
            System.out.println(this.version);
            System.out.println(SamsungGalaxy.this.version);
        }
    }
}

class Main {
    public static void main(String[] args) {
SamsungGalaxy sg = new SamsungGalaxy();
SamsungGalaxy.FixedBattery sgb = sg.new FixedBattery();
        sgb.print();
    }
}
```

Output = 4, 3, 2

Bugs and Testing

- **Software reliability**: Probability that a software system will not cause failure under specified conditions.

- **Bugs** are inevitable in any complex software system.
  - Industry estimates: 10-50 bugs per 1000 lines of code.
  - A bug can be visible or can hide in your code until much later.

- **Testing**: A systematic attempt to reveal errors.
  - Failed test: an error was demonstrated.
  - Passed test: no error was found (for this particular situation)
  - It plays a critical role in helping a software company deliver quality products to its customers.

© Vivek Kumar

Manual Testing v/s Automated Testing

Manual Testing	Automated Testing
Executing a test cases manually without any tool support is known as manual testing	Taking tool support and executing the test cases by using an automation tool is known as automation testing
Time-consuming and tedious – Since test cases are executed by human resources, it is very slow and tedious	**Fast** – Automation runs test cases significantly faster than human resources
Huge investment in human resources – As test cases need to be executed manually, more testers are required in manual testing	**Less investment in human resources** – Test cases are executed using automation tools, so less number of testers are required in automation testing
Less reliable – Manual testing is less reliable, as it has to account for human errors	**More reliable** – Automation tests are precise and reliable.
Non-programmable – No programming can be done to write sophisticated tests to fetch hidden information	**Programmable** – Testers can program sophisticated tests to bring out hidden information

Source: https://www.tutorialspoint.com/junit/junit_overview.htm

JUnit: Java Unit Testing Framework

- The Java library **JUnit** helps us to easily perform automated unit testing
- The basic idea:
  - For a given class `Foo`, create another class `FooTest` to test it, containing various "test case" methods to run.
  - Each method looks for particular results and passes / fails
- JUnit provides "**assert**" commands to help us write tests.
  - The idea: Put assertion calls in your test methods to check things you expect to be true. If they aren't, the test will fail

© Vivek Kumar

Sample JUnit Test

```java
/* The class method to be tested */
public class Sum {
    private int var1, var2;
    public Sum(int v1, int v2) {var1=v1; var2=v2;}
    public int sum () {
        return var1 + var2;
    }
}
```

```java
/* Junit test class */

import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class MyTest {

    @Test
    public void testSum() {
        Sum mySum = new Sum(1, 1);
        int sum = mySum.sum();
        assertEquals(2, sum);
    }
}
```

```java
/* Junit test runner class */

import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
   public static void main(String[] args) {
      Result result= JUnitCore.runClasses(MyTest.class);
      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }
      System.out.println(result.wasSuccessful());
   }
}
```


© Vivek Kumar

CSE201: Monsoon 2022
Advanced Programming

Lecture 19: Introduction to Design Patterns

Raghava Mutharaju

CSE, IIIT-Delhi

raghava.mutharaju@iiitd.ac.in

© Vivek Kumar

What is Design Pattern

- It is a solution for a repeatable problem in the software design
- This is not a complete design for a software system that can be directly transformed into code
- It is a description or template for how to solve the problem that can be used in many different situations

Why Study Patterns

- Reuse tried, proven solutions
  - Provides a head start
  - Avoids gotchas later (unanticipated things)
  - No need to reinvent the wheel
- Establish common terminology
  - Design patterns provide a common point of reference
  - Easier to say, “We could use Strategy here.”
- Provide a higher level prerspective
  - Frees us from dealing with the details too early

“GoF” (Gang of Four) patterns

- **Creational Patterns** *(abstracting the object-instantiation process)*
  - Factory Method Abstract Factory Singleton
  - Builder Prototype
- **Structural Patterns** *(how objects/classes can be combined)*
  - Adapter Bridge Composite
  - Decorator Facade Flyweight
  - Proxy
- **Behavioral Patterns** *(communication between objects)*
  - Command Interpreter Iterator
  - Mediator Observer State
  - Strategy Chain of Responsibility Visitor
  - Template Method

In 1990 a group called the Gang of Four or "GoF" (Gamma, Helm, Johnson, Vlissides) compile a catalog of design patterns in the book “Design Patterns: Elements of Reusable Object-Oriented Software”

Pattern: Iterator

objects that traverse collections

Pattern: Iterator

- Recurring Problem
  - How can you loop over all objects in any collection. You don’t want to change client code when the collection changes. Want the same methods
- Solution
  1. Provide a standard *iterator* object supplied by all data structures
  2. The implementation performs traversals, does bookkeeping
  3. The implementation has knowledge about the representation
  4. Results are communicated to clients via a standard interface
- Consequences
  - Can change collection class details without changing code to traverse the collection

Java Iterator Recap

Iterable Interface Source Code

```
package java.lang;
public interface Iterable<E> {
    Iterator<E> iterator();
}
```

- Just one method in this interface
- Objects of all classes that implements this interface can be the target of foreach statement
- Iterators allow iterating over the entire collection. It also allows element removal from collection during iteration

© Vivek Kumar

Iterator Interface

- Defines three fundamental methods
  - `Object next()`
  - `boolean hasNext()`
  - `void remove()`
- These three methods provide access to the contents of the collection
- An Iterator knows position within collection
- Each call to next() “reads” an element from the collection
  - Then you can use it or remove it

© Vivek Kumar

Iterator Position

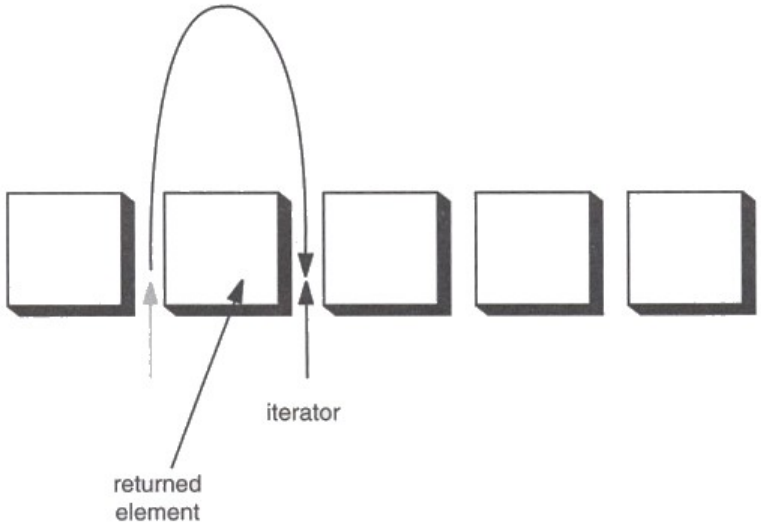


Figure 2–3: Advancing an iterator

© Vivek Kumar

Collection Interface Source Code

```
package java.util;
public interface Collection<E> extends
Iterable<E> {
    int size();
    boolean isEmpty();
    contains(Object o);
    boolean add(E e);
    boolean remove(Object o);
    equals(Object o);
    ........
    ........
}
```

- Defines fundamental methods that are enough to define the basic behavior of a collection
- Inherit the method from Iterable interface

© Vivek Kumar

Example - SimpleCollection

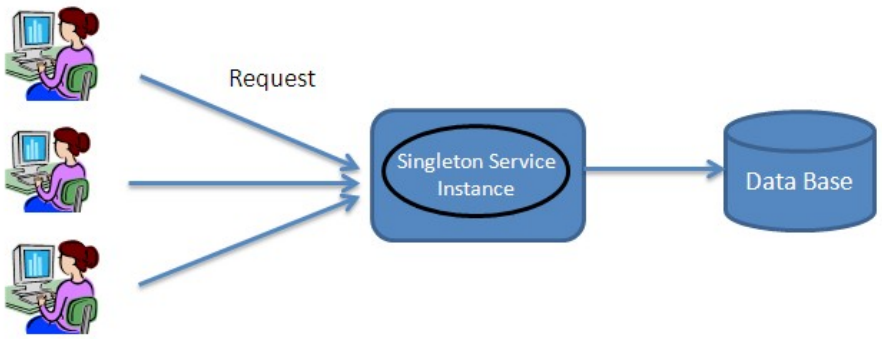
```
public class SimpleCollection  {
  public static void main(String[] args) {
      Collection c = new ArrayList();
      for (int i=0; i < 10; i++) {
          c.add(i);
      }
      Iterator iter = c.iterator();
      while (iter.hasNext())
          System.out.println(iter.next());
  }
}
```


© Vivek Kumar

Pattern: Singleton

A class that has only a single instance

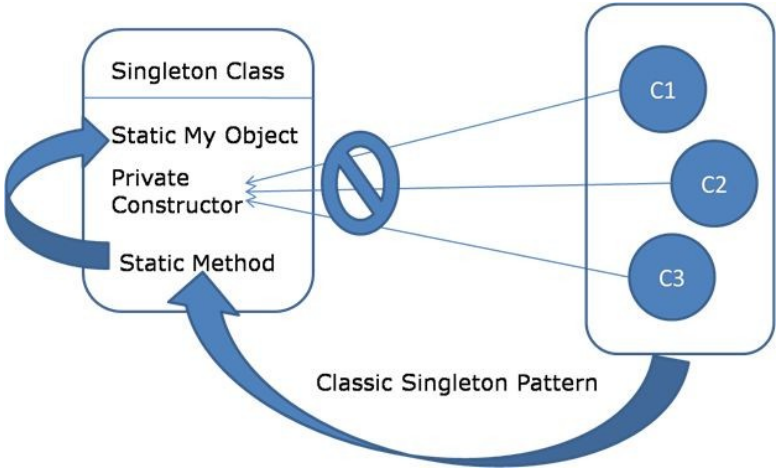
Pattern: Singleton



- Recurring problem
  - Sometimes we only ever need one instance of a particular class
  - It should be illegal to have another instance of the same class
- Solution
  - Singleton pattern – ensuring that a class has at most one instance
  - Providing global access to that instance

Fig. source: https://rajneekanth.wordpress.com/2014/04/11/what-are-design-patterns/

Implementing Singleton



1. Make constructor private so that no client is able to call it from outside
2. Declare a single private static instance of the class
3. Write a getInstance() method (or similar) that allows access to the single instance
   - Ensure thread safety in case multiple threads can access this method

Singleton Example

```
public class RandomGenerator {
    private static RandomGenerator gen = null;
    public static RandomGenerator getInstance()
    {
        if (gen == null) {
            gen = new RandomGenerator();
        }
        return gen;
    }
    private RandomGenerator() {}
    ...
}
```

- Creates a new random generator
- Clients will not use the constructor directly but will instead call getInstance to obtain a RandomGenerator obect that is shared by all classes in the application
- Lazy initialization
  - Can wait until client asks for the instance to create it
  - How to ensure thread safety?

© Vivek Kumar

Singleton Comparator

```
public class LengthComparator
                implements Comparator<String> {

    private static LengthComparator comp = null;

    public static LengthComparator getInstance()
    {
        if (comp == null) {
            comp = new LengthComparator();
        }
        return comp;
    }

    private LengthComparator() {}

    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
}
```

- Comparators make great singletons because they have no state
- Saves memory by not allowing the creation of more than one object

© Vivek Kumar

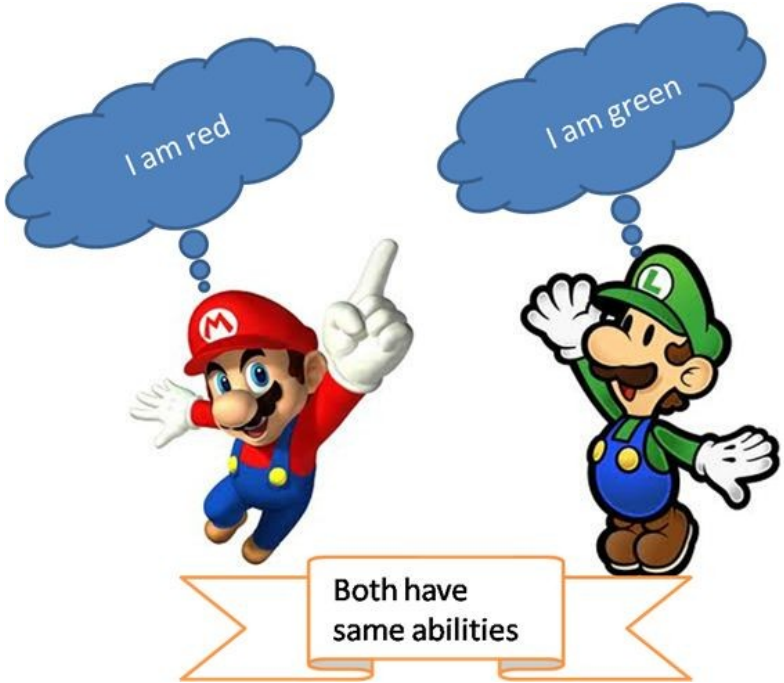
Pattern: Flyweight

a class that has only one instance for each unique state

Pattern: Flyweight

- Problem
  - **Redundant objects** can bog down the system
    - Many objects have the same state
  - Example: `File` objects that represent the same file on disk
    - `new File("chatlog.txt")`
    - `new File("chatlog.txt")`
    - `new File("chatlog.txt")`
      `...`
    - `new File("notes.txt")`
  - Example: Date objects that represent the same date of the year
    - `new Date(4, 18)`
    - `new Date(4, 18)`

Pattern: Flyweight



- An assurance that no more than one instance of a class will have identical state
  - Achieved by caching identical instances of objects.
  - Similar to singleton, but one instance for each unique object state
  - Useful when there are many instances, but many are equivalent

Fig source: http://www.c-sharpcorner.com/UploadFile/SukeshMarla/learn-design-pattern-flyweight-pattern/

Implementing a Flyweight (1/2)

```
public class Flyweighted {
    private static Map<KeyType, Flyweighted> instances
            = new HashMap<KeyType, Flyweighted>();

    private Flyweighted(...) { ... }

    public static Flyweighted getInstance(KeyType key) {
        if (!instances.contains(key)) {
            instances.put(key, new Flyweighted(key));
        }
        return instances.get(key);
    }
}
```

Implementing a Flyweight (2/2)

```java
public class Point {
    private int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() { return x; }
    public int getY() { return y; }

    public String toString() {
        return "(" + x + ", " + y + ")";
    }
}
```

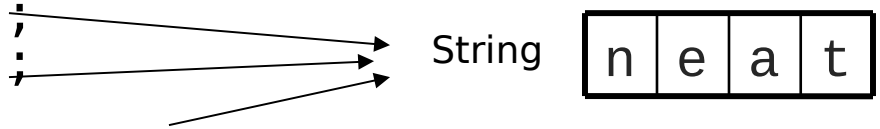
```java
public class Point {

    private static Map<String, Point> instances =
        new HashMap<String, Point>();

    public static Point getInstance(int x, int y)
    {
        String key = x + ", " + y;
        if (!instances.containsKey(key)) {
            instances.put(key, new Point(x, y));
        }
        return instances.get(key);
    }

    private final int x, y;  // immutable

    private Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() { return x; }
    public int getY() { return y; }
    public String toString() {
        return "(" + x + ", " + y + ")";
    }
}
```


© Vivek Kumar

Flyweighting in String by JVM

- The possible combinations for Strings is close to infinite, hence JVM maintains a cache for strings, called the **string constant pool**
  - It is empty at startup and is filled constantly during the lifecycle of the JVM
- Java String objects are automatically flyweighted by the JVM **whenever** possible
  - If you declare two string variables that point to the same literal.
  - If you concatenate two string literals to match another literal

```
String a = "neat";
String b = "neat";
String c = "n" +
"eat";
```



Next Lecture

- More design patterns

CSE201: Monsoon 2022
Advanced Programming

Lecture 20: Adapter and Strategy Design Pattern

Raghava Mutharaju

CSE, IIIT-Delhi

raghava.mutharaju@iiitd.ac.in

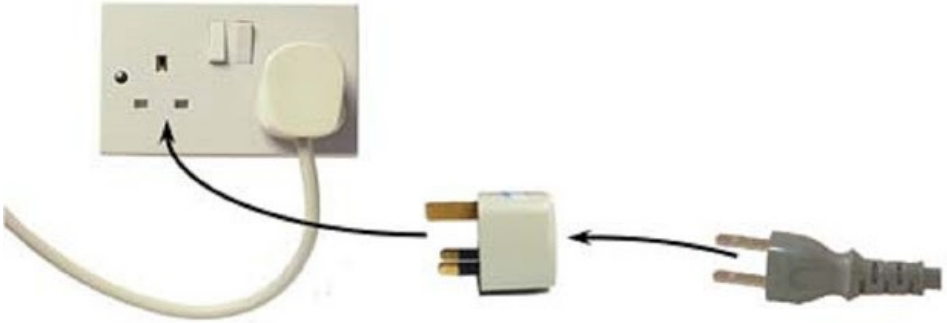
Today’s Lecture

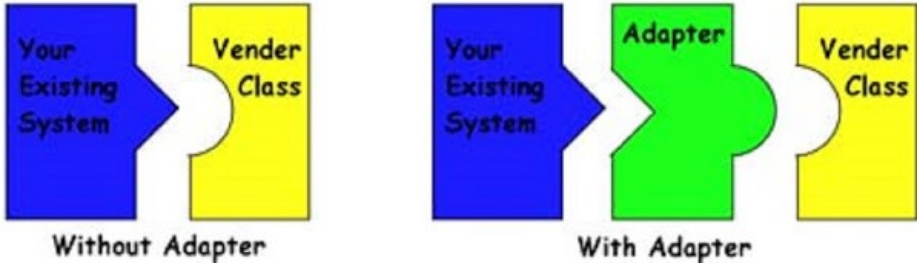
- Adapter design pattern (DP # 4)
- Strategy design pattern (DP # 5)

Pattern: Adapter

an object that fits another object into a given interface

Pattern: Adapter





- Recurring problem
  - We have an object that contains the functionality we need, but not in the way we want to use it
- Solution
  - Create an **adapter object** that bridges the provided and desired functionality

Fig source: http://javarevisited.blogspot.in/2016/08/adapter-design-pattern-in-java-example.html

Adapter Pattern Example (1/2)

```
public interface Movable {
    public void move();
}

public class Car implements Movable {
    public void move() {
        System.out.println("Car is moving");
    }
}

public class Bike implements Movable {
    public void move() {
        System.out.println("Bike is moving");
    }
}
```

```
public class Vehicle {
  public static void main(String[] args) {
    List<Movable> mylist = new ArrayList<Movable>();

    mylist.add(new Car());
    mylist.add(new Bike());


    for(Movable obj: mylist) {
        obj.move();
    }
  }

}
```

```
public interface Flyable {
   public void fly();
}

public class Airplane implements Flyable {
   public void fly() {
      System.out.println("Airplane is flying");
   }
}

public class Drone implements Flyable {
   public void fly() {
      System.out.println("Drone is flying");
   }
}
```

- The **adaptee** interface "Flyable" only implements fly() method, although it is similar to move() in Movable inteface
- Client class, Vehicle, doesn't understand Flyable and only use Movable
  - How to add Flyable type objects inside Movable type list in Vehicle?
  - We will code an adaptor that can serve this client by using this adaptee without any modifications

© Vivek Kumar

Adapter Pattern Example (2/2)

```
public interface Movable {
    public void move();
}

public class Car implements Movable {
    public void move() {
        System.out.println("Car is moving");
    }
}

public class Bike implements Movable {
    public void move() {
        System.out.println("Bike is moving");
    }
}
```

```
public interface Flyable {
   public void fly();
}

public class Airplane implements Flyable {
   public void fly() {
      System.out.println("Airplane is flying");
   }
}

public class Drone implements Flyable {
   public void fly() {
      System.out.println("Drone is flying");
   }
}
```

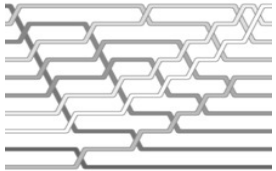
```
public class Vehicle {
  public static void main(String[] args) {
    List<Movable> mylist = new ArrayList<Movable>();

    mylist.add(new Car());
    mylist.add(new Bike());

    mylist.add(new FlyableAdapter(new Airplane()));
    mylist.add(new FlyableAdapter(new Drone()));

    for(Movable obj: mylist) {
        obj.move();
    }
  }

}
```

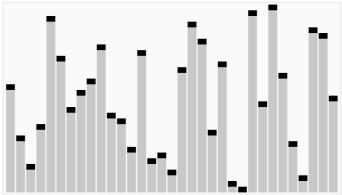
```
public class FlyableAdapter implements Movable {
    Flyable type;
    public FlyableAdapter(Flyable type) {
        this.type = type;
    }
    public void move() {
        type.fly();
    }
}
```

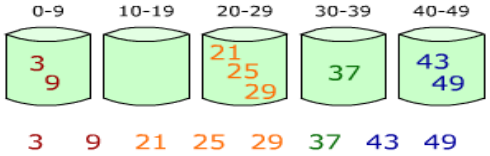

© Vivek Kumar

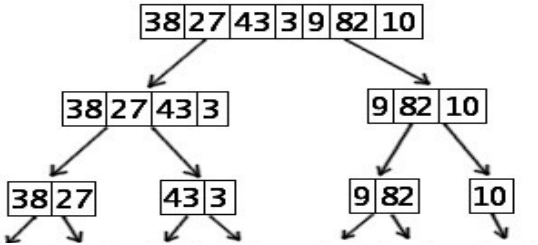
Pattern: Strategy

objects that hold different algorithms to solve a problem









The Ducks File Structure for Redux – S ...
medium.com

Wood Duck Identification, All Abo
allaboutbirds.org

Duck test - Wikipedia
en.wikipedia.org

Ever Wanted to Know About Ducks
thoughtco.com

Duck

Bufflehead

(*Bucephala albeola*)

Scientific classification

Kingdom:	Animalia
Phylum:	Chordata
Class:	Aves
Order:	Anseriformes
Superfamily:	Anatoidea
Family:	**Anatidae**

Subfamilies

see text

Amazon.com: Giant Duck F...
amazon.com

Different Kind of Ducks | D'Artagnan
dartagnan.com

Caring for Ducks in Winter | Modern ...
oysfarm.com

Different Types of Ducks With Examples
thespruce.com

Let’s Build a Duck Simulator!

- Concepts we will revisit
  - Inheritance
  - Interfaces
  - Polymorphism

© Vivek Kumar

What are their Characteristics?

- I’m Dabbler duck
- I can quack
- I can swim
- I can fly
- My home is on ground

- I’m Wood duck
- I can quack
- I can swim
- I can fly
- My home is on trees

© Vivek Kumar

How to Code a Duck Simulator?

- I'm Dabbler duck
- I can quack
- I can swim
- I can fly
- My home is on ground

- I'm Wood duck
- I can quack
- I can swim
- I can fly
- My home is on trees

Inheritance?

© Vivek Kumar

Lets See the Code

```
public abstract class Duck {
    private String name;
    public Duck(String n) { this.name = n; }

    public void type() {
        System.out.println("I am "+ name+" Duck");
    }
    public void speak() {
        System.out.println("I can quack");
    }
    public void swim() {
        System.out.println("I can swim");
    }
    public void fly() {
        System.out.println("I can fly");
    }
    public abstract void home();
    public void display() {
        this.type();
        this.speak();
        this.swim();
        this.fly();
        this.home();
    }
}
```

```
public class Dabbler extends Duck {
    public Dabbler() { super("Dabbler"); }

    public void home() {
        System.out.println("My home is on ground");
    }
}
```

```
public class Wood extends Duck {
    public Wood() { super("Wood"); }

    public void home() {
        System.out.println("My home is on trees");
    }
}
```

```
// Calling display on above two Duck type
objects
I am Wood Duck
I can quack
I can swim
I can fly
My home is on trees
I am Dabbler Duck
I can quack
I can swim
I can fly
My home is on ground
```

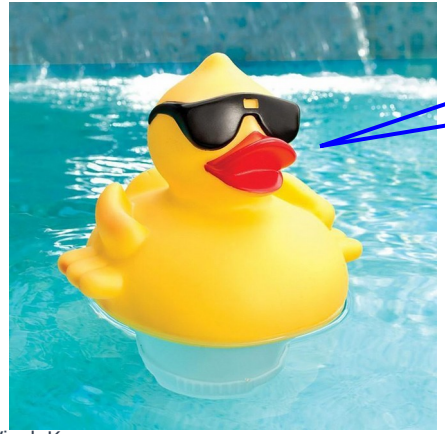

© Vivek Kumar

Any Problems?

```
public abstract class Duck {
    private String name;
    public Duck(String n) { this.name = n; }

    public void type() {
        System.out.println("I am "+ name+"
Duck");
    }
    public void speak() {
        System.out.println("I can quack");
    }
    public void swim() {
        System.out.println("I can swim");
    }
    public void fly() {
        System.out.println("I can fly");
    }
    public abstract void home();
    public void display() {
        this.type();
        this.speak();
        this.swim();
        this.fly();
        this.home();
    }
}
```

```
public class Dabbler extends Duck {
    public Dabbler() { super("Dabbler"); }

    public void home() {
        System.out.println("My home is on
ground");
    }
}
```

```
public class Wood extends Duck {
    public Wood() { super("Wood"); }

    public void home() {
        System.out.println("My home is on
trees");
    }
}
```



Please code me too

- I'm Rubber duck
- **I can squeak**
- I can swim
- **I don't fly**
- Your home is my home

© Vivek Kumar

What are the Issues?

- Applying inheritance for code reuse sometimes backfires
- Poor solution for maintenance
  - Our assumption that all Ducks can Fly is incorrect
  - Our assumption that all Ducks make quack-quack sound is incorrect
- How to fix this issue?
  - Overriding both the methods fly() and speak() in subclass Rubber Duck

© Vivek Kumar

Let’s Implement the Fix

```
public abstract class Duck {
    private String name;
    public Duck(String n) { this.name = n; }

    public void type() {
        System.out.println(“I am “+ name+”
Duck”);
    }
    public void speak() {
        System.out.println(“I can quack”);
    }
    public void swim() {
        System.out.println(“I can swim”);
    }
    public void fly() {
        System.out.println(“I can fly”);
    }
    public abstract void home();
    public void display() {
        this.type();
        this.speak();
        this.swim();
        this.fly();
        this.home();
    }
}
```

```
public class Rubber extends Duck {
    public Rubber() { super(“Rubber”); }

    @Override
    public void speak() {
        System.out.println(“I can Squeak”);
    }
    @Override
    public void fly() {
        System.out.println(“I don’t Fly”);
    }
    public void home() {
        System.out.println(“Your home is my
home”);
    }
}
```

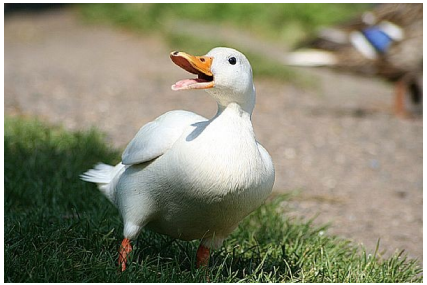
```
// Calling display on Rubber Duck type object
I am Rubber Duck
I can Squeak
I can swim
I don’t Fly
Your home is my home
```


© Vivek Kumar

Wait.. What if we get other non-flyable Duck?

```
public abstract class Duck {
    private String name;
    public Duck(String n) { this.name = n; }

    public void type() {
        System.out.println("I am "+ name+"
Duck");
    }
    public void speak() {
        System.out.println("I can quack");
    }
    public void swim() {
        System.out.println("I can swim");
    }
    public void fly() {
        System.out.println("I can fly");
    }
    public abstract void home();
    public void display() {
        this.type();
        this.speak();
        this.swim();
        this.fly();
        this.home();
    }
}
```

```
public class Rubber extends Duck {
    public Rubber() { super("Rubber"); }

    @Override
    public void speak() {
        System.out.println("I can Squeak");
    }
    @Override
    public void fly() {
        System.out.println("I don't Fly");
    }
    public void home() {
        System.out.println("Your home is my
home");
    }
}
```



- If we have to code a **Domestic Duck** then they too don't fly
  - This means we need to Override the fly() method even inside Domestic Duck class

© Vivek Kumar

What are the Issues?

- Another Duck type could speak in a language other than “Quack” and “Squeak”
  - Examples:
    - **Decoy** Duck can’t speak
    - **Whistling** Duck make whistles
  - As there are **several possible** ways to speak, we don’t have any choice other than **Overriding** the speak() method
- However, the flying capability could be either true or false only. As the options for flying capability is limited, can we write a better code?
  - How about using an interface called Flyable that has fly() method?
    - Again there will be lot of duplicate code as each Duck type will have to implement this interface to show their flying capability

© Vivek Kumar

Recap: Design Principals

- Program to a supertype and not for an implementation
  - We used Duck as superclass in past
- Identify the aspects of the implementation that differs and separate them out from what stays the same
  - We took out similar functionality inside the superclass Duck and left the specialized implementation inside subclass

© Vivek Kumar

Using Strategy Pattern for Final Fix

1. We will still use **Flyable** interface BUT will limit its implementation in only **two** classes
2. Create a field of **Flyable** type in supertype (Duck)
3. Each subclass will simply instantiate this field inside their constructor with correct flying ability. The flying capability are defined inside the two classes mentioned in Step-1
4. display() method in Duck will use polymorphism to show the correct flying capability

© Vivek Kumar

Applying Strategy Pattern: The Final Fix!

```
public interface Flyable {
    public void fly();
}
```

```
public abstract class Duck {
    private String name;
    private Flyable flyStatus;
    public Duck(String n, Flyable f) {
        this.name = n;
        this.flyStatus = f;
    }
    ........
    ........
    public void tryFlying() {
        flyStatus.fly();
    }
    public void display() {
        this.type();
        this.speak();
        this.swim();
        this.tryFlying();
        this.home();
    }
}
```

```
public class CannotFly implements Flyable {
    public void fly() {
        System.out.println("I don't Fly");
    }
}
```

```
public class CanFly implements Flyable {
    public void fly() {
        System.out.println("I can Fly");
    }
}
```

```
public class Dabbler extends Duck {
    public Dabbler() {
        super("Dabbler", new CanFly());
    }
    ........
}
```

```
public class Rubber extends Duck {
    public Rubber() {
        super("Rubber", new CannotFly());
    }
    @Override
    public void speak() {
        System.out.println("I can Squeak");
    }
    public void home() {
        System.out.println("Your home is my home");
    }
}
```

© Vivek Kumar

Summary: Strategy Pattern

- In Strategy pattern, a class behavior (or its algorithm) can be changed at run time
- In Strategy pattern, we create objects which represent various strategies and a context object whose behavior varies as per its strategy object
- The strategy object changes the executing algorithm of the context object
- This type of design pattern comes under behavior pattern

© Vivek Kumar

Next Lecture

- More design patterns

CSE201: Monsoon 2022
Advanced Programming

Lecture 21: Template, Prototype, Factory & Façade Design Patterns

Raghava Mutharaju
CSE, IIIT-Delhi
raghava.mutharaju@iiitd.ac.in

Last Lecture

Adaptor Design Pattern

- Recurring problem -- We have an object that contains the functionality we need, but not in the way we want to use it
- Solution – Create an **adapter object** that bridges the provided and desired functionality

Strategy pattern

- Here, a class behavior (or its algorithm) can be changed at run time
- In Strategy pattern, we create objects which represent various strategies and a context object whose behavior varies as per its strategy object
- The strategy object changes the executing algorithm of the context object

```
public interface Movable {
    public void move();
}

public class Car implements Movable {
    public void move() {
        System.out.println("Car is moving");
    }
}

public class Bike implements Movable {
    public void move() {
        System.out.println("Bike is moving");
    }
}
```

```
public interface Flyable {
   public void fly();
}

public class Airplane implements Flyable {
   public void fly() {
      System.out.println("Airplane is flying");
   }
}

public class Drone implements Flyable {
   public void fly() {
      System.out.println("Drone is flying");
   }
}
```

```
public class Dabbler extends Duck {
    public Dabbler() {
        super("Dabbler", new CanFly());
    }
    ........
}
```

```
public class Rubber extends Duck {
    public Rubber() {
        super("Rubber", new CannotFly());
    }
    @Override
    public void speak() {
        System.out.println("I can Squeak");
    }
    public void home() {
        System.out.println("Your home is my home");
    }
}
```

```
public class Vehicle {
  public static void main(String[] args) {
    List<Movable> mylist = new ArrayList<Movable>();

    mylist.add(new Car());
    mylist.add(new Bike());

    mylist.add(new FlyableAdapter(new Airplane()));
    mylist.add(new FlyableAdapter(new Drone()));

    for(Movable obj: mylist) {
        obj.move();
    }
  }
}
```

```
public class FlyableAdapter implements Movable {
    Flyable type;
    public FlyableAdapter(Flyable type) {
        this.type = type;
    }
    public void move() {
        type.fly();
    }
}
```

```
public interface Flyable {
    public void fly();
}
```

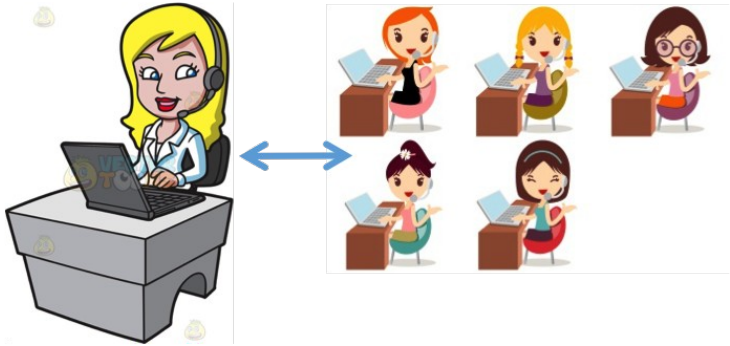
```
public abstract class Duck {
    private String name;
    private Flyable flyStatus;
    public Duck(String n, Flyable f) {
        this.name = n;
        this.flyStatus = f;
    }
    ........
    ........
    public void tryFlying() {
        flyStatus.fly();
    }
    public void display() {
        this.type();
        this.speak();
        this.swim();
        this.tryFlying();
        this.home();
    }
}
```

```
public class CannotFly implements Flyable {
    public void fly() {
        System.out.println("I don't Fly");
    }
}
```

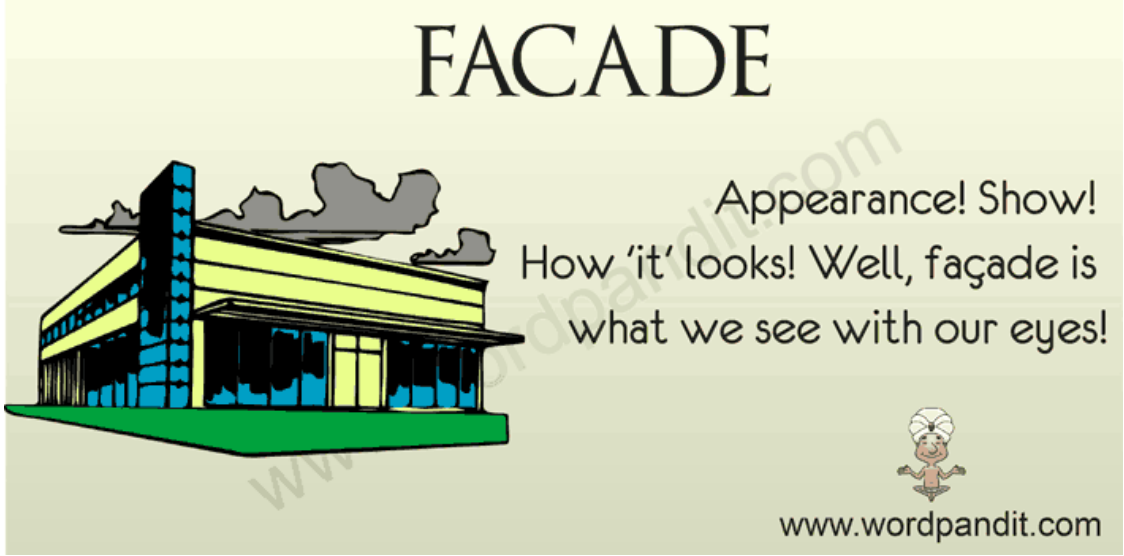
```
public class CanFly implements Flyable {
    public void fly() {
        System.out.println("I can Fly");
    }
}
```

Today’s Lecture

- Some more design patterns
  - Template (DP # 6)
  - Prototype (DP # 7)
  - Factory (DP # 8)
    - Abstract Factory (DP # 9)
  - Façade (DP # 10)



Pattern: Facade



Facade Pattern

- **Facade:** a structural design pattern used to identify a simple way to realize relationships between entities
- Provide a unified “interface” to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use

© Vivek Kumar

The Tale of a Call Center

```
class CallCenter {
    public void handleNetwork() { / *Some code */ }
    public void handleBilling() { /* Some code */ }
    public void handleRoaming() { /* Some code */ }
    public void handleAccount() { /* Some code */ }
     ......
}
```

```
public class Client {
    public static void main(String[] args) {
        CallCenter c = new CallCenter();
        c.handleNetwork();
        c.handleBilling();
        c.handleRoaming();
        c.handleAccount();
    }
}
```

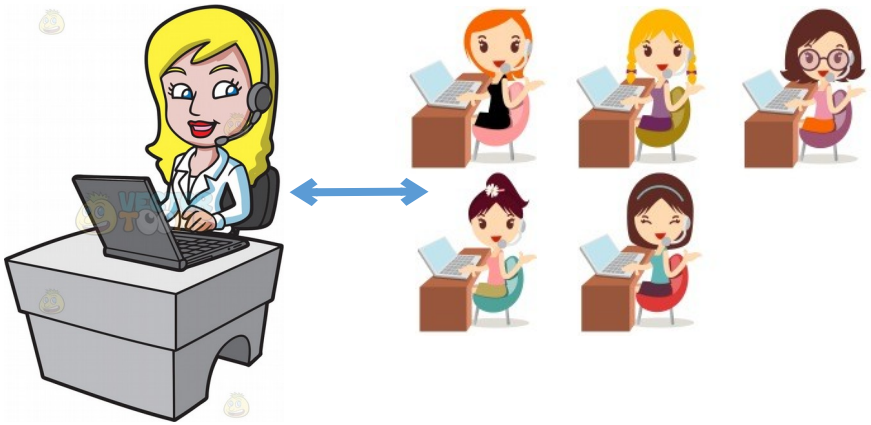
- Call center wants cost cutting and employees only one agent for handling all customer issues
  - Result?
    - Overloaded employee and bad customer satisfaction!

© Vivek Kumar

A Better Call Center Using Facade

```
class CallCenter {
    NetworkTeam net;
    BillingTeam bill;
    RoamingTeam roam;
    AccountTeam account;
    public CallCenter() { /* initializations */ }
    public void handleCalls(int option) {
        switch(option) {
            case 1:
                net.handleNetwork();
                break;
            case 2:
                bill.handleBilling();
                break;
            .......
        }
    }
}
```

```
public class Client {
    public static void main(String[] args) {
        CallCenter c = new CallCenter();
        c.handleCalls(1);
        ......
    }
}
```



- Facade design to the rescue
  - Hiding the complexities of a large body of code by providing a simplified interface

© Vivek Kumar

Pattern: Template

Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses

Acknowledgements for slides on template pattern: Informatics 122 Software Design II, Bren School (UCI)

Let’s Build a Café Simulator

- Coffee
  - Boil Water
  - Brew Coffee in boiling water
  - Pour in cup
  - Add sugar and milk

Inheritance?

- Tea
  - Boil Water
  - Steep tea in boiling water
  - Pour in cup
  - Add sugar and lemon

© Vivek Kumar

Let’s See the Code

```
public abstract class NonAlcoholicDrink {
    public void boilWater() {
        System.out.println(“Boil Water”);
    }
    public void pourInCup() {
        System.out.println(“Pour in Cup”);
    }
    public abstract void prepare();
}
```

```
public class Coffee extends NonAlcoholicDrink {
    public void prepare() {
        boilWater();
        brewCoffee();
        pourInCup();
        addSugarAndMilk();
    }
    private void brewCoffee() {
        System.out.println(“Brew Coffee”);
    }
    private void addSugarAndMilk() {
        System.out.println(“Add Sugar and Milk”);
    }
}
```

```
public class Tea extends NonAlcoholicDrink {
    public void prepare() {
        boilWater();
        steepTeaBag();
        pourInCup();
        addSugarAndLemon();
    }
    private void steepTeaBag() {
        System.out.println(“Steep Tea Bag”);
    }
    private void addSugarAndLemon() {
        System.out.println(“Add Sugar and Lemon”);
    }
}
```

- Do you see any issues here?
  - Similar algorithms in `prepare` !!
    - How about doing the following?
      - Replace brewCoffee() and steepTeaBag() with brew()
      - Replace addSugarAndMilk() and addSugarAndLemon() with addCondiments()

© Vivek Kumar

Template Pattern

- The Template Method pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses
- Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure
- Usage
  1. Define the algorithm in superclass and ensure that subclasses cannot change the structure of this algorithm
  2. Each step of the algorithm is represented by a method
  3. Steps (methods) handled by subclasses are declared abstract
  4. Shared steps (concrete methods) are placed in the superclass

The Fixed Code

```java
public abstract class NonAlcoholicDrink {
    public void boilWater() {
        System.out.println("Boil Water");
    }
    public void pourInCup() {
        System.out.println("Pour in Cup");
    }
    // "final" ensures that the person preparing
    // the beverage sticks to the recipe of this
    // Café instead of generating their own
    public final void prepare() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }
    public abstract void brew();
    public abstract void addCondiments();
}
```

```java
public class Coffee extends NonAlcoholicDrink {
    private void brew() {
        System.out.println("Brew Coffee");
    }
    private void addCondiments() {
        System.out.println("Add Sugar and Milk");
    }
}
```

```java
public class Tea extends NonAlcoholicDrink {
    private void brew() {
        System.out.println("Steep Tea Bag");
    }
    private void addCondiments() {
        System.out.println("Add Sugar and Lemon");
    }
}
```


© Vivek Kumar

Pattern: Prototype

An object that serves as a basis for creation of others

Source of this slide: CSE331, Washington University

Let’s Build a Cloning Laboratory Simulator

- We are going to **clone** following Animals in our lab
  - Sheep
    - **“Is an”** Animal but has wool
  - Chicken
    - **“Is an”** Animal but lay eggs
- Which concepts we will be using?
  - Inheritance
  - Object cloning

© Vivek Kumar

Cloning Lab Simulator

```
public class Animal {
    private String name;
    public Animal(String n) { name=n; }
    public void sayHello() {
        System.out.println("I am a " + name);
    }
}
```

```
public class Sheep extends Animal implements Cloneable {
    private String wool;
    public Sheep() { super("Sheep"); wool ="10KG"; }
    public void sayHello() {
        super.sayHello();
        System.out.println("I have "+wool+" wool");
    }
    public Sheep clone() throws CloneNotSupportedException {
        return (Sheep) super.clone();
    }
}
```

```
public class Chicken extends Animal implements Cloneable {
    private int eggs;
    public Chicken() { super("Chicken"); eggs=3; }
    public void sayHello() {
        super.sayHello();
        System.out.println("I have "+eggs+" eggs");
    }
   public Chicken clone() throws CloneNotSupportedException{
        return (Chicken) super.clone();
    }
}
```

```
public class Lab1 {
    public static Sheep getClone(Sheep s)
           throws CloneNotSupportedException {
        return s.clone();
    }
}
```

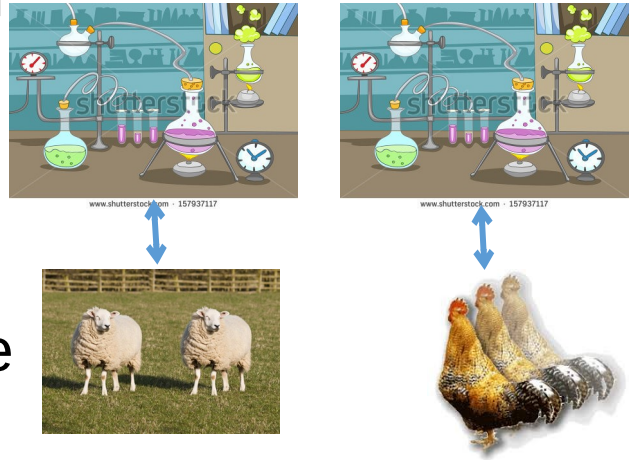
```
public class Lab2 {
    public static Chicken getClone(Chicken s)
           throws CloneNotSupportedException {
        return s.clone();
    }
}
```

```
public class Client {
  public static void main(String[] args) throws CloneNotSupportedException{
        Sheep s1 = new Sheep(); Chicken c1 = new Chicken();
        Sheep s2 = Lab1.getClone(s1);
        Chicken c2 = Lab2.getClone(c2);
    }
}
```


© Vivek Kumar

What are the Issues?

- Instead of having just one laboratory for all Animal types, we ended up creating individual Animal specific laboratory
  - No use of polymorphism!
- Client has to ensure that they request the laboratory suited for the Animal type
- Code duplication!
  - More serious when we need to code some more Animal types (Cow, Dog, etc.)



© Vivek Kumar

Prototype Pattern

- **Problem:** Client wants another object similar to an existing one, but doesn't care about the details of the state of that object
  - Creating an instance of a class is time-consuming or complex in some way

- **Solution**
  - Decouple product creation from system behavior
  - Avoid subclasses of an object creator in the client application

© Vivek Kumar

The Fixed Version

```
public class Animal implements Cloneable {
    private String name;
    public Animal(String n) { name=n; }
    public void sayHello() {
        System.out.println("I am a " + name);
    }
    public Animal clone() throws CloneNotSupportedException {
        return (Animal) super.clone();
    }
}
```

```
public class Lab {
    public static Animal getClone(Animal s) {
        return s.clone();
    }
}
```

```
public class Sheep extends Animal {
    private String wool;
    public Sheep() { super("Sheep"); wool ="10KG"; }
    public void sayHello() {
        super.sayHello();
        System.out.println("I have "+wool+" wool");
    }
    public Sheep clone() throws CloneNotSupportedException {
        return (Sheep) super.clone();
    }
}
```

```
public class Chicken extends Animal {
    private int eggs;
    public Chicken() { super("Chicken"); eggs=3; }
    public void sayHello() {
        super.sayHello();
        System.out.println("I have "+eggs+" eggs");
    }
   public Chicken clone() throws CloneNotSupportedException{
        return (Chicken) super.clone();
    }
}
```

Sheep and Chicken also requires clone() implementation to enable deep copy (if any such fields are there in class)

```
public class Client {
  public static void main(String[] args) throws CloneNotSupportedException{
        Animal s1 = new Sheep(); Animal c1 = new Chicken();
        Animal s2 = Lab.getClone(s1);
        Animal c2 = Lab.getClone(c2);
    }
}
```


© Vivek Kumar

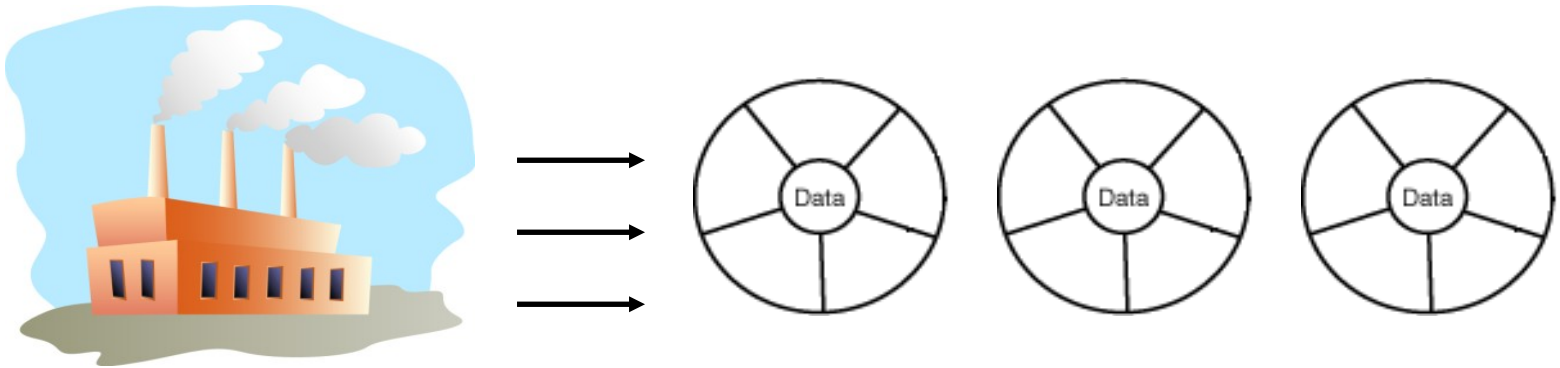
Drawback of Prototype Pattern

- It is built on the method `clone()`, which could be complicated sometimes in terms of shallow copy and deep copy

© Vivek Kumar

Pattern: Factory

A method or object that creates other objects



Source of this slide: CSE331, Washington University

Let’s Revisit our Client from Cloning Laboratory

```
public class Client {
  public static void main(String[] args) throws
CloneNotSupportedException{
        String need = args[0];
        Animal animal;
        if(need.equals(“wool”) {
            animal = new Sheep();
        }
        else if(need.equals(“eggs”) {
            animal = new Chicken();
        }
        else if(need.equals(“milk”) {
            animal = new Cow();
        }
        else System.exit(-1);
        // Our client is too greedy
        Animal[] cloned = new Animal[100];
        for(int i=0; i<cloned.length; i++) {
            cloned[i] = Lab.getClone(animal);
        }
    }
}
```

- We have got more funding and our lab now support some more Animals!
  - Our client now has options to choose Animals based on their requirements
- What is the issue here?
  - Mixing two events in same place (or method)
    - Animal creation
    - Cloning of Animal

© Vivek Kumar

The Issue with “new”

```
public class Client {
  public static void main(String[] args) throws
CloneNotSupportedException{
        String need = args[0];
        Animal animal;
        if(need.equals(“wool”) {
            animal = new Sheep();
        }
        else if(need.equals(“eggs”) {
            animal = new Chicken();
        }
        else if(need.equals(“milk”) {
            animal = new Cow();
        }
        else if(need.equals(“protection”) {
            animal = new Dog();
        }
        else if(need.equals(“riding”) {
            animal = new Horse();
        }
        ........
    }
}
```

- When we have several related classes, that’s an indication that they might change in future
  - We might expand our Lab to support cloning of several other Animals…
- What is the issue?
  - Client code needs to be recompiled:
    - Every time we **add** the support for a new Animal in our Lab
    - Every time if we **remove** the support for an existing Animal in our Lab

© Vivek Kumar

Factory Pattern

- **Factory**: A method or object whose primary purpose is to manage the creation of other objects (usually of a different type)

- **Problem:** Object creation is cumbersome or heavily coupled for a given client. Client needs to create but doesn't want the details.

- **Solution:** A helper method that creates and returns the object(s)

The Fix: Encapsulate Creation Code

```
public class AnimalFactory {
    public Animal createAnimal(String need) {
        if(need.equals("wool") {
            return new Sheep();
        }
        else if(need.equals("eggs") {
            return new Chicken();
        }
        .......
    }
}
```

```
public class Client {
  public static void main(String[] args) throws
CloneNotSupportedException{
        String need = args[0];
        AnimalFactory factory = new AnimalFactory();
        Animal animal = factory.createAnimal(need);
        // Our client is too greedy
        Animal[] cloned = new Animal[100];
        for(int i=0; i<cloned.length; i++) {
            cloned[i] = Lab.getClone(animal);
        }
    }
}
```

- What are the benefits?
  - Client need not recompile if support for Animals are added or removed in our Lab
  - Easy to serve some other Client class
  - Ensure consistent object initialization

© Vivek Kumar

We Have Another Problem Now…

```
public class AnimalFactory {
    public Animal createAnimal(String need) {
        if(need.equals("wool") {
            return new Sheep();
        }
        else if(need.equals("eggs") {
            return new Chicken();
        }
        .......
    }
}
```

```
public class Client {
  public static void main(String[] args) throws
CloneNotSupportedException{
        String need = args[0];
        AnimalFactory factory = new AnimalFactory();
        Animal animal = factory.createAnimal(need);
        // Our client is too greedy
        Animal[] cloned = new Animal[100];
        for(int i=0; i<cloned.length; i++) {
            cloned[i] = Lab.getClone(animal);
        }
    }
}
```

- Our cloning Lab is in very high demand and we have started cloning almost every Animal (*except ourselves…*)
- Supporting creation of so many Animals in just `AnimalFactory` class is becoming a bottleneck

© Vivek Kumar

Abstract Factory Pattern

- A superclass factory that can be extended to provide different sub-factories, each with different features
- Used when we have **multiple families of object** components

Cat Family

Dog Family

© Vivek Kumar

The Fix: Abstract Factory Pattern

```
public abstract class AnimalFactory {
    public abstract Animal createAnimal(String need);
}
```

```
public class CatFactory extends AnimalFactory {
    public Animal createAnimal(String need) {
        if(need.equals("pet") {
            return new HouseCat();
        }
        else if(need.equals("zoo") {
            return new Lion();
        }
    }
}
```

```
public class DogFactory extends AnimalFactory {
    public Animal createAnimal(String need) {
        if(need.equals("kids") {
            return new Poodle();
        }
        else if(need.equals("hunting") {
            return new Greyhound();
        }
    }
}
```

```
public class ClientForCats {
  public static void main(String[] args) throws
CloneNotSupportedException{
        String need = args[0];
        AnimalFactory factory = new CatFactory();
        Animal animal = factory.createAnimal(need);
        // Our client is too greedy
        Animal[] cloned = new Animal[100];
        for(int i=0; i<cloned.length; i++) {
            cloned[i] = Lab.getClone(animal);
        }
    }
}
```

```
public class ClientForDogs {
  public static void main(String[] args) throws
CloneNotSupportedException{
        String need = args[0];
        AnimalFactory factory = new DogFactory();
        Animal animal = factory.createAnimal(need);
        // Our client is too greedy
        Animal[] cloned = new Animal[100];
        for(int i=0; i<cloned.length; i++) {
            cloned[i] = Lab.getClone(animal);
        }
    }
}
```


© Vivek Kumar

Next Lecture

- **Quiz-4 on Friday**