CSCI 2110 Data Structures and Algorithms

Module 1: Key Concepts in Object-Oriented Programming

CSCI 2110: Module 1 Srini Sampalli

OUR FIRST MODULE:
A Painless Refresher on Object oriented programming

REVIEW +

What we will cover in this module…

- Traditional vs. Object-oriented programming (OOP)
- Basics of OOP in Java
- The Class file → Defines an Object → Encapsulation
- Two useful entities: this parameter and the toString method
- Method Overloading
- Garbage Collection in Java
- Static methods and Static Variables

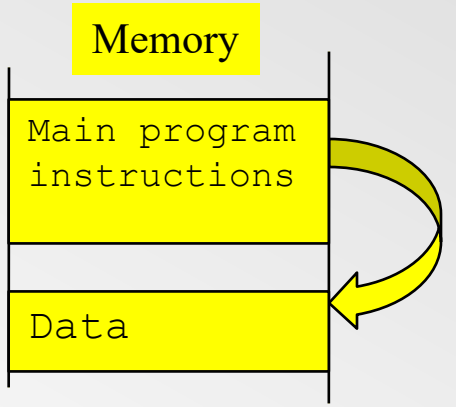
What we will cover in this module…

- Inheritance
- The Object Class
- Abstract Classes and Interfaces
- Polymorphism
- Generics

Traditional Programming Paradigms

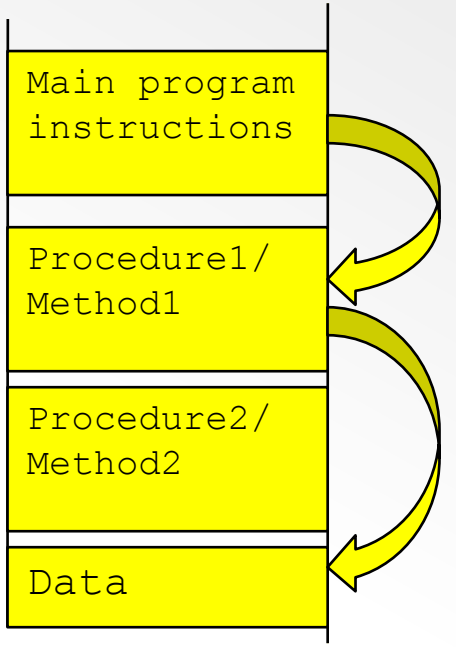
Sequential programming (example - early BASIC)

- Only one main program with a sequence of instructions.
- Instructions directly operate on global data.



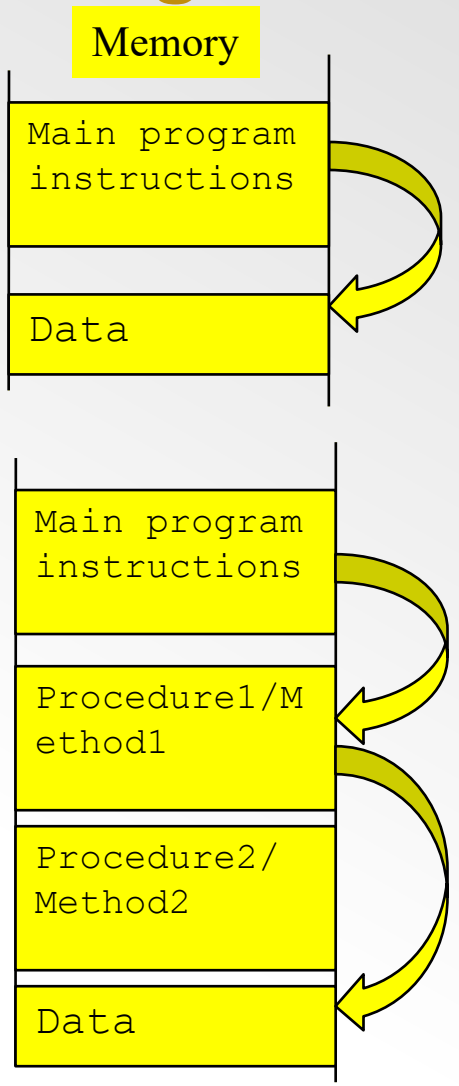
Procedural Programming (examples – C, Pascal, FORTRAN)

- Main program "calls" procedures or methods.
- Control is returned to the main program once the procedure is completed.
- Data is still global.



Some drawbacks of traditional programming

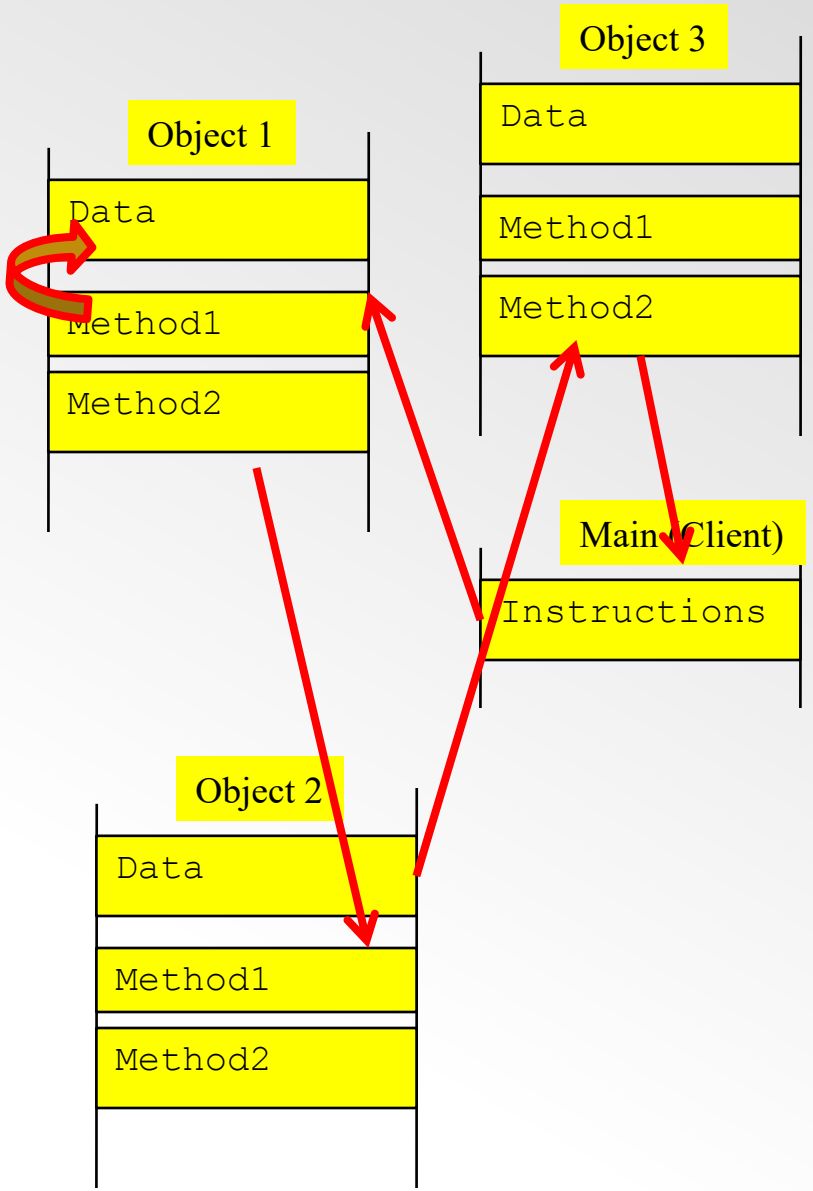
- Does not model the real world.
- Inefficient coding.
- Problem with global data – any method could access any data. This could lead to programming errors.



The Object-Oriented Approach

Key Idea: Encapsulate both data and methods into one software entity called an object.

Each object carries its own data and can be acted upon by its own procedures/methods.



- The program is a web of interacting objects.
- Objects interact with each other by sending messages (invoking operations on methods of other objects).
- There is also a main program (called the client) that initiates the process.

Examples of "objects"

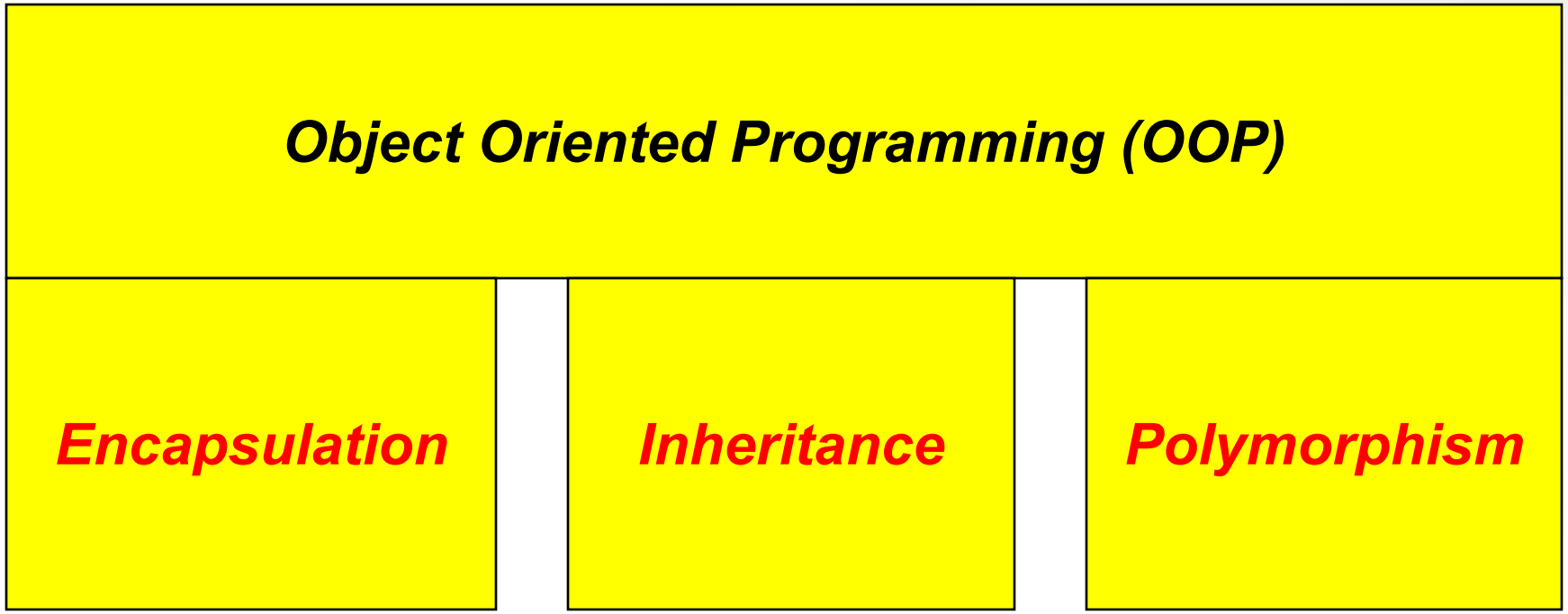
- Geometry program – ***Rectangle object, Circle object, etc.***
- Bank Account program – ***Bank Account object, Loan object***
- Stock Purchase program – ***Stock object***
- Simple car simulator – ***Start button, stop button, gas pedal, brake and steering wheel objects***
- Alarm clock simulator – ***Number display objects, sound alarm object, button objects, etc.***
- Course (database) ***– Student object, Instructor object, Textbook object***

Advantages of object-oriented programming

- Mirrors *real-world* applications.
- Each programming module (object) can be changed independent of the other modules → *flexibility and scalability*.
- Only allowed data (variables) and methods can be accessed by other modules → *less error-prone.*
- Each object can be modeled by an *interface*.
- The interface tells the outside world what the object has to offer and what it is capable of.
- A *client can send a message* to an interface with a guarantee that even if the object's implementation changes, client does not have to rewrite the code.

The three pillars of object-oriented programming

- ***Encapsulation is just one of the important features of OOP.***
- ***The other two are Inheritance and Polymorphism.***



Basics of Object-oriented programming in Java

- Two basic steps:
  - Define objects → **ENCAPSULATION**
  - Create and use the objects
- We define an object in a ***class file***:

Fields/ Attributes to hold data

→ variable declarations (typically private)

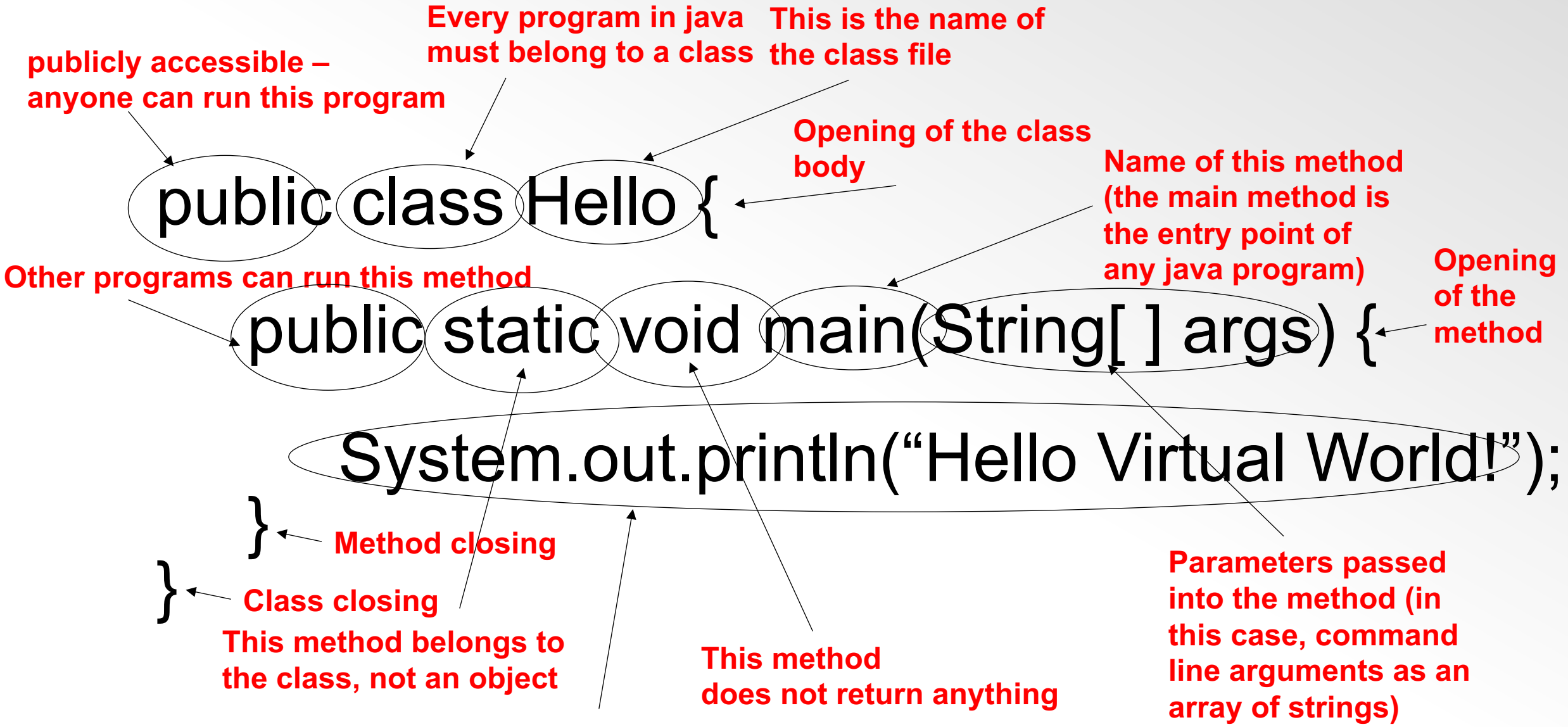
Procedures/Operations on the data

→ methods (typically public)

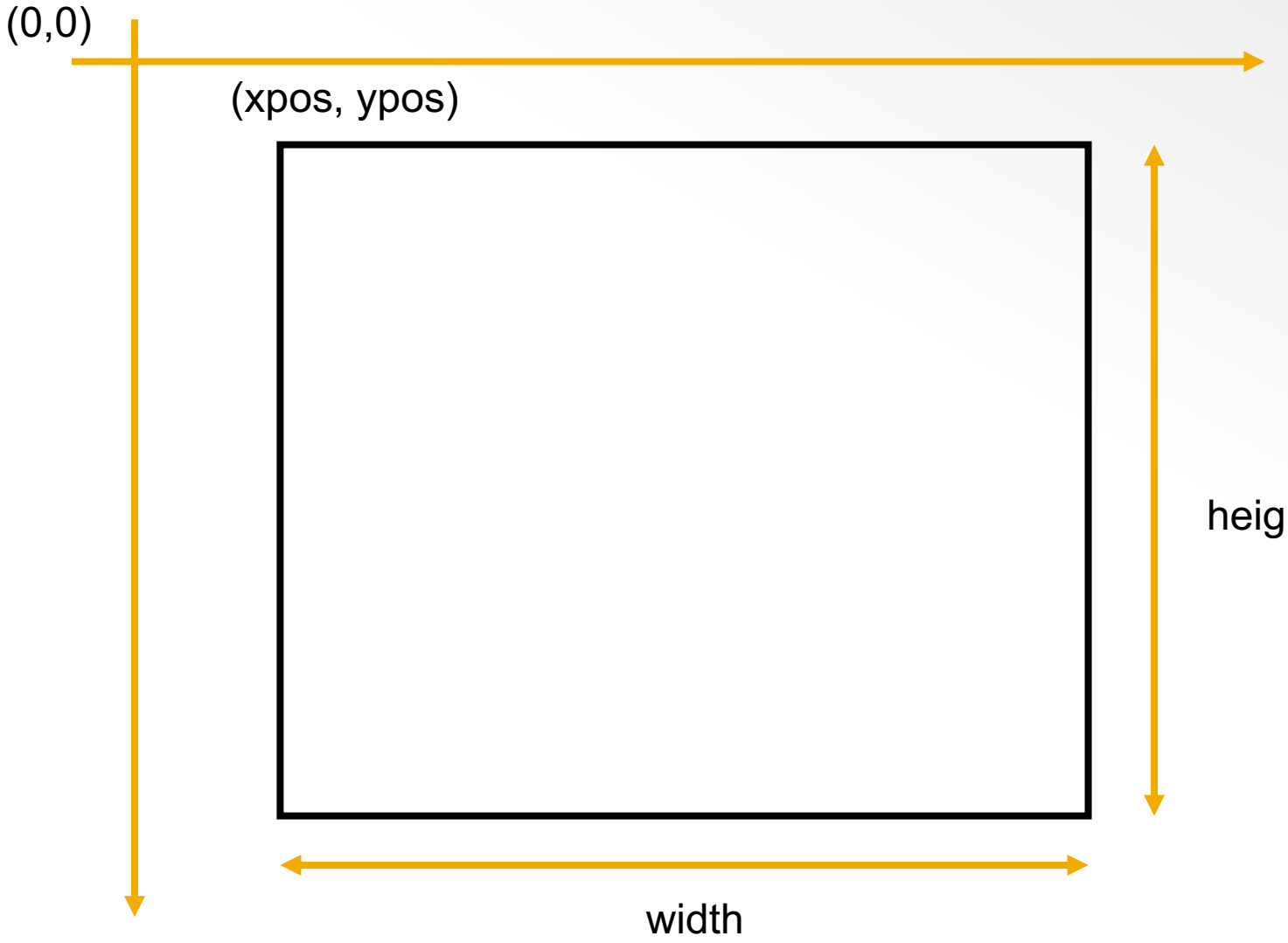
A super-simple example (no objects – just a client program)

```
public class Hello {

    public static void main(String[ ] args) {

        System.out.println("Hello Virtual World!");
    }
}
```

A super-simple example (no objects – just a client program)



Let's look at an object-oriented program with objects



A Rectangle Object

Code Example 1: A Basic Object Oriented Program

```
public class Rectangle1{
      private int xpos, ypos, width, height;

      public Rectangle1(){
      }
      public void setX(int x){
      xpos = x;
      }
      public void setY(int y){
      ypos = y;
      }
      public void setWidth(int w){
      width = w;
      }
      public void setHeight(int h){
      height = h;
      }
      public int getX(){
      return xpos;
      }
      public int getY(){
      return ypos;
      }
      public int getWidth(){
      return width;
      }
      public int getHeight(){
      return height;
      }
      public void moveTo(int x, int y){  xpos = x;
      ypos = y;
      }
      public void resize(int w, int h)
      {
      width = w;
      height = h;
      }
}

public class Rectangle1Demo
      {
      public static void main(String[] args){
            Rectangle1 rect1;
            rect1 = new Rectangle1();
            rect1.setX(100);
            rect1.setY(50);
            rect1.setWidth(20);
            rect1.setHeight(25);
            rect1.moveTo(200,300);
            rect1.resize(10,30);
            System.out.println("[" +
                  rect1.getX()+","+rect1.getY()+"'"+"]" + "\twidth: " +
                  rect1.getWidth() + "\theight: " +
                  rect1.getHeight()+"\n");

      }
}
```

Two useful entities: the keyword "this" and the "toString" method

The keyword this:

In the method

```
public int setX(int x){xpos = x;}
```

suppose that we want to use the same variable name xpos as the argument.

This can be done by using the keyword **`this`**.

`this` refers to the parameter of the current object and can be used to distinguish the instance variable from the method parameter. Thus:

```
public int setX(int xpos){this.xpos = xpos;}
```

Two useful entities: the keyword "this" and the "toString" method

The toString method:

Just gives the String representation of an object.

We can design the toString method in order to print the object's attributes in whatever convenient form that we wish.

For example, we can use it to print xpos, ypos, width and height of the rectangle object.

The method header is `public String toString()`

`toString()` can be ignored when calling the method.

Code Example 2: Illustration of the 'this' parameter and 'toString' method

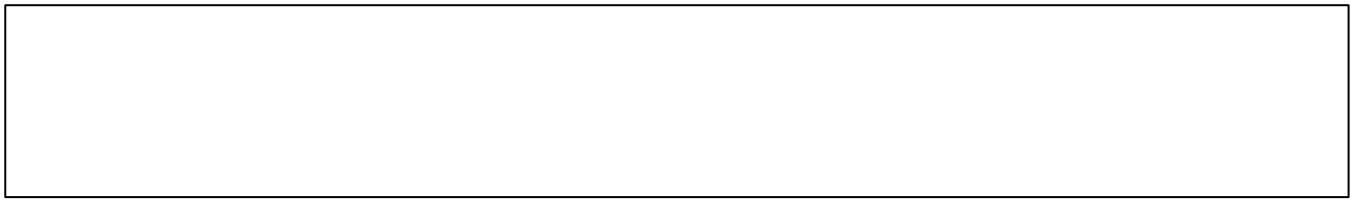
```
public class Rectangle2{
      private int xpos, ypos, width, height;

      public Rectangle2(){ }
      public void setX(int xpos){
            this.xpos = xpos;
      }
      public void setY(int ypos){
            this.ypos = ypos;
      }
      public void setWidth(int width){
            this.width = width;
      }
      public void setHeight(int height){
            this.height = height;
      }
      public int getX(){
            return xpos;
      }
      public int getY(){
            return ypos;
      }
      public int getWidth(){
            return width;
      }
      public int getHeight(){
            return height;
      }
      public void moveTo(int xpos, int ypos){
            this.xpos = xpos;
      this.ypos = ypos;
                  }
      public void resize(int width, int height){
            this.width = width;
            this.height = height;
      }
      public String toString(){
            return "["+ xpos + "," + ypos + "]" +
      "\t" + "width= " + width + "," "height= " + height;
      }

      public static void main(String[] args){
            Rectangle2 rect1;
            rect1 = new Rectangle2();
            rect1.setX(100);
            rect1.setY(50);
            rect1.setWidth(20);
            rect1.setHeight(25);
            System.out.println("Before modification: " +  rect1);
            rect1.moveTo(200,300);
            rect1.resize(10,30);
            System.out.println("After modification: " +  rect1);
            }

}
```

Method overloading

- *Can two methods in a class have the same name?*
- **Yes – provided their signatures are different.**
- *What is the signature of a method?*
- The ***signature of a method*** is the method's name, type and order of its input parameters. Return type is not part of the signature.
- Examples: *resize(int, int)* and *setHeight(int)* are the signatures of the resize and setHeight methods, respectively.
- ***When you write two or more methods with the same name in a class, it is called This method overloading and the methods are called overloaded methods.***
- Method overloading is useful when we need several different ways to perform the same operation.
- ***Constructors can also be overloaded, which means that a class can have more than one constructor and eventually, more than one way of creating objects.***

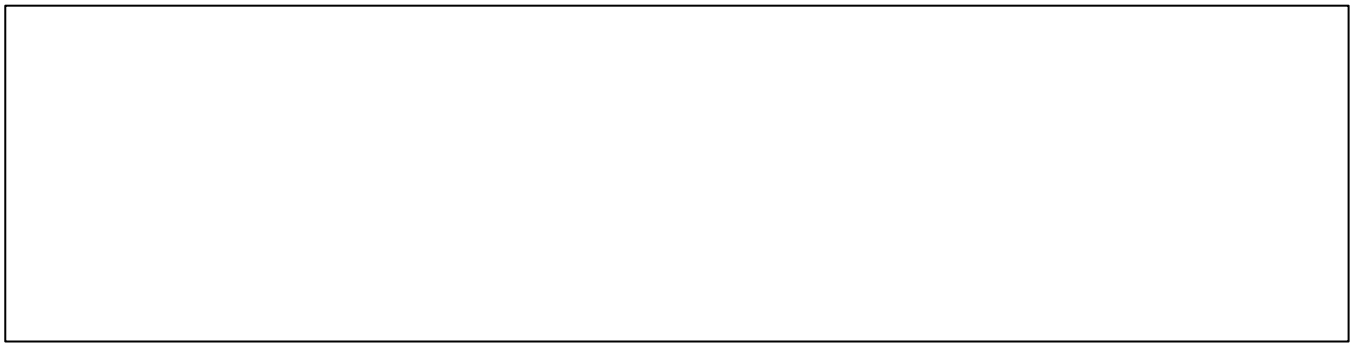
Tryout Code Example 3: Method Overloading

```
public class Rectangle3{
      private int xpos, ypos, width, height;
      public Rectangle3(){
      }
      //overloaded constructor
```



```
      //rest of the code same as Rectangle2.java
      //contains method: true if a point px, py is contained in this rectangle
```

```
      //contains method: true if another rectangle is contained in this  rectangle
```



Garbage Collection in Java

- The memory space occupied by an object is reclaimed by the garbage collector in java when the object is no longer in use.
- For example, suppose that in a program we write***:***

```
Rectangle3 rect1 = new Rectangle3(100,50, 10, 20);
....
rect1 = new Rectangle3(200, 50, 20, 30);
```

- The first rectangle which was created has no reference to it anymore. Thus it is not accessible anymore.
- Nullifying an object reference will also invoke the garbage collector to remove it:

```
rect1 =  null;
```

- Garbage collector always runs in the background, identifies objects without references, and reclaims the space used by the object. You can also request the garbage collector with the following statement in your program: ***System.gc();***

STATIC METHODS AND VARIABLES

Static Methods

- A static method is one that can be used ***without an object.***
- Sometimes we may need methods that have no relation to an object of any kind.
- Examples:
  - Method to determine the larger of two integers.
  - Method to compute the cube root of a number.
  - Method to convert temperature from Celsius to Fahrenheit.

- Static methods are defined by writing the keyword **`static`** in the method header.

Static Methods (cont'd.)

- Static methods are still defined inside a class, but they can be invoked without creating an object.
- For example, you can create a collection of static methods to perform computations that are related and group them within a single class.
- You can also include static methods along with other non-static methods in a class.

Calling a static method

- When you call a static method from outside the class, the class name is used if being invoked without an object:

  className.methodName(….parameters…)

- The class name can be ignored if the static method is called from within the same class.

Examples of static methods

- *"main" is a static method that can be included in any class.*

- Another example of a class of static methods : Math class. (example usage: Math.pow(2.0,3.0), Math.random(5), etc.)

- Note: All the methods that are used in procedural programming are static methods.

```
public class Temp{
 public static double C2F(double c){
     return (c*9.0/5.0 + 32.0);
 }
 public static double F2C(double f){
     return ((f-32.0)*5.0/9.0);
 }
 public static void main(String[] args){
     double ctemp, ftemp;

     ftemp = Temp.C2F(32.0);

     ctemp = Temp.F2C(100.0);
 }
}
```

Static Variables

What is a static variable?

When you write a class file, you can declare two kinds of variables:

- ***Instance variables***
- ***Static variables***

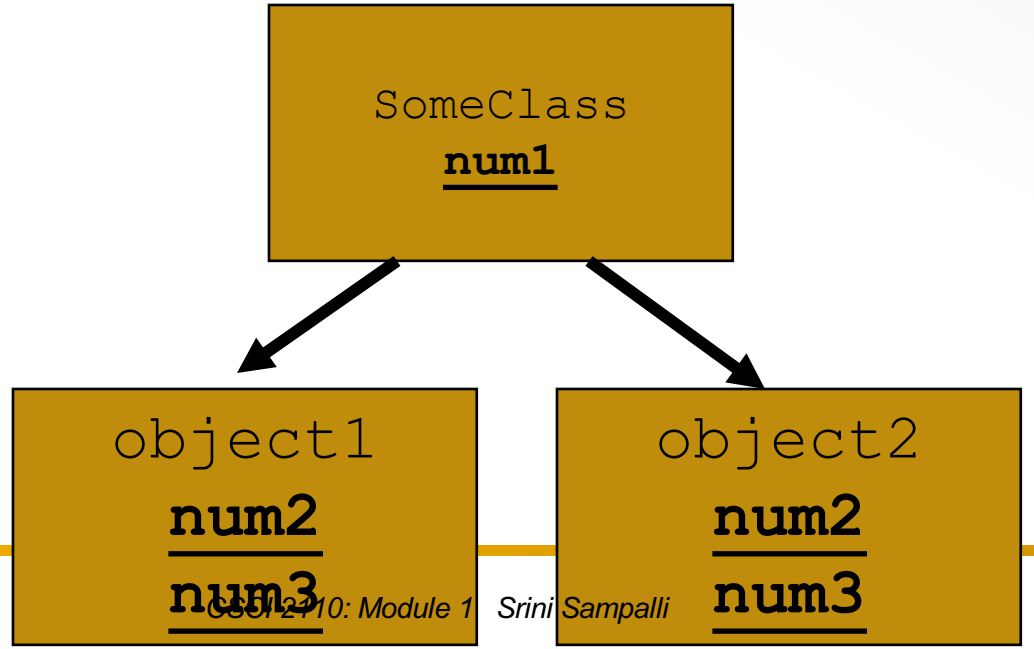
Instance variable: Each object gets a separate version of this variable.

Static variable: Only one copy exists. Individual objects do not get separate versions.

```
public class SomeClass
{
        private static int num1;

        private int num2;
        private int num3;
```

num1 is
a static variable

num2 and num3 are
instance variables



Rules for static methods and variables

- You can define a class with static variables and instance variables, static methods and non-static methods.
- A *non-static method in a class* can access both instance variables and static variables of the class.
- A *non-static method in a class* can invoke both static and non-static methods of the class.
- A *static method in a class* can access only static variables and static methods directly.
- It can access non-static variables and non-static methods only through objects.
- Constants are shared by all objects of the class. Thus, constants should be declared final static. For e.g.,

private final static double PI = 3.14159;

Code Example 4: Static variables and methods

The objective is to design a Bank Account class that has a combination of static variables, instance variables, static methods and non-static methods. Implement a BankAccount class with the following specifications:

Instance variables name and balance
Static variables interest rate and number of accounts, both set to 0.
A constructor which sets the name and balance, and increments number of accounts.
A static method to set the interest rate
A static method to get the interest rate
A static method to get the number of accounts
A non-static method to deposit an amount
A non-static method to withdraw an amount
A non-static method to add interest
A non-static method to get the balance.
A non-static method to set the balance.
A non-static method to transfer funds to another account.
A non-static toString method

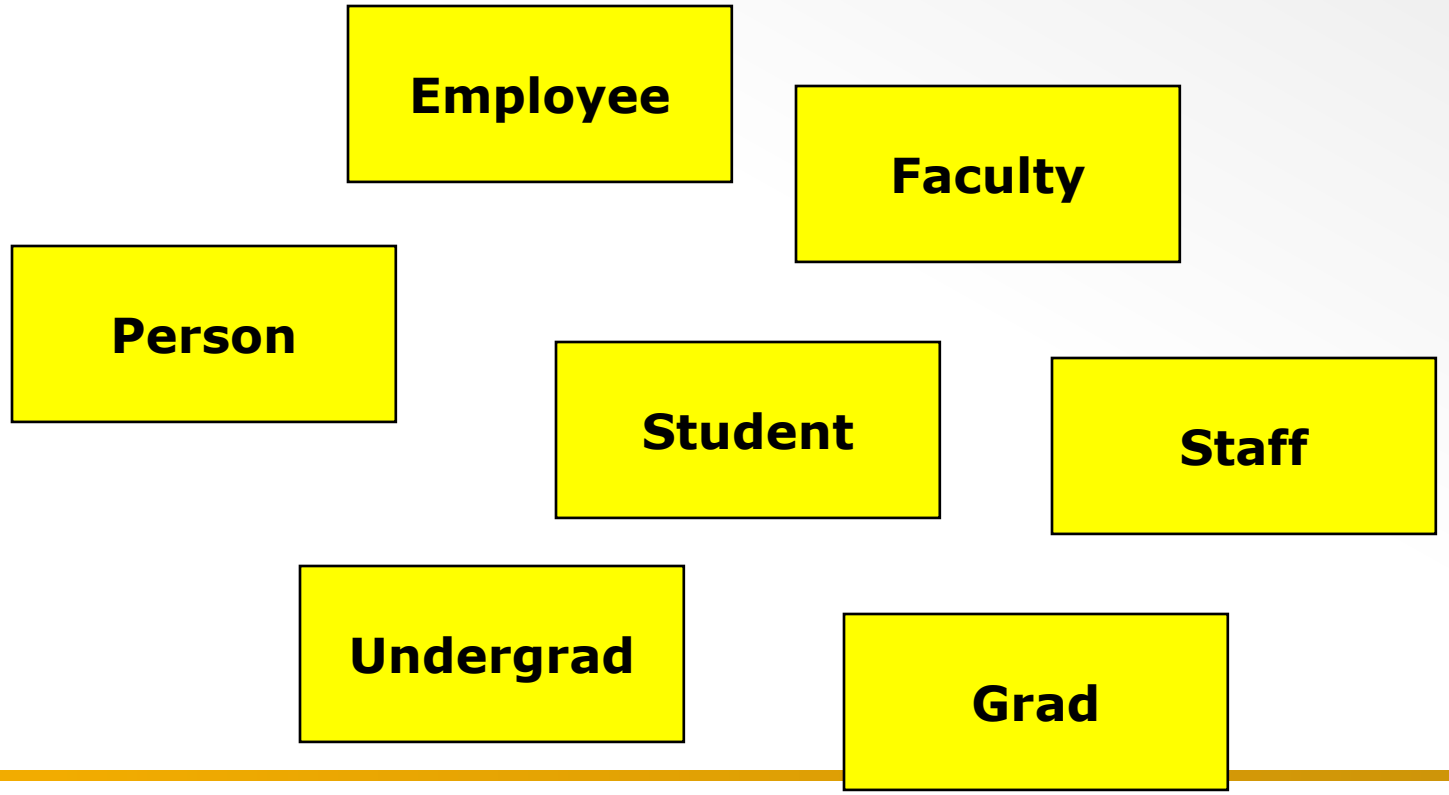
Test the BankAccount class by creating five Bank Accounts, depositing amounts into each, adding interest , withdrawing amounts and transferring funds. Print the individual accounts as well as the static variable that indicates the number of accounts created.

```
//BankAccount.java
public class BankAccount{
	private String name;
	private double balance;
	private static double rate = 0.0;
	private static int numAccounts = 0;

	public BankAccount(String n, double b){
		name = n;
		balance = b;
		numAccounts++;
	}
	public static void setInterestRate(double r){
		rate = r;
	}
	public static double getInterestRate(){
		return rate;
	}
	public static int getNumAccounts(){
		return numAccounts;
	}
	public void deposit(double amount){
		if (amount<0.0)
			System.out.println("Invalid amount");
		else
			balance+=amount;
	}

	public void withdraw(double amount){
		if (amount>balance)
			System.out.println("Insufficient funds");
		else
```

```
                balance-=amount;
        }
        public void addInterest(){
              double interest = balance*rate;
              balance+=interest;
        }
        public double getBalance(){
              return balance;
        }
        public void setBalance(double amount){
              balance +=amount;
        }
        public void transferTo(BankAccount b, double amount){
              double bal = b.getBalance();
              if (amount > this.balance)
                    System.out.println("Insufficient funds. Can't
                    transfer");
              else
              {
                    this.balance = this.balance - amount;
                    b.setBalance(bal + amount);
              }
        }
        public String toString(){
              String str;
              str = "Name: " + name + "\tBalance: " + balance;
              return str;
        }

        public static void main(String[] args){
              BankAccount.setInterestRate(0.01);
              BankAccount A, B, C, D, E;
              A = new BankAccount("Art", 100.00);
              B = new BankAccount("Bob", 200.00);
              C = new BankAccount("Chuck", 300.00);
              D = new BankAccount("Dirk", 400.00);
              E = new BankAccount("Emily", 500.00);
              A.deposit(50.00);
              B.withdraw(25.00);
              C.addInterest();
              D.transferTo(C, 50.00);
              System.out.println("Number of accounts: " +
                          BankAccount.getNumAccounts());
              System.out.println(A + "\n" + B + "\n" + C + "\n" + D +
                    "\n" + E + "\n");

        }


}
```

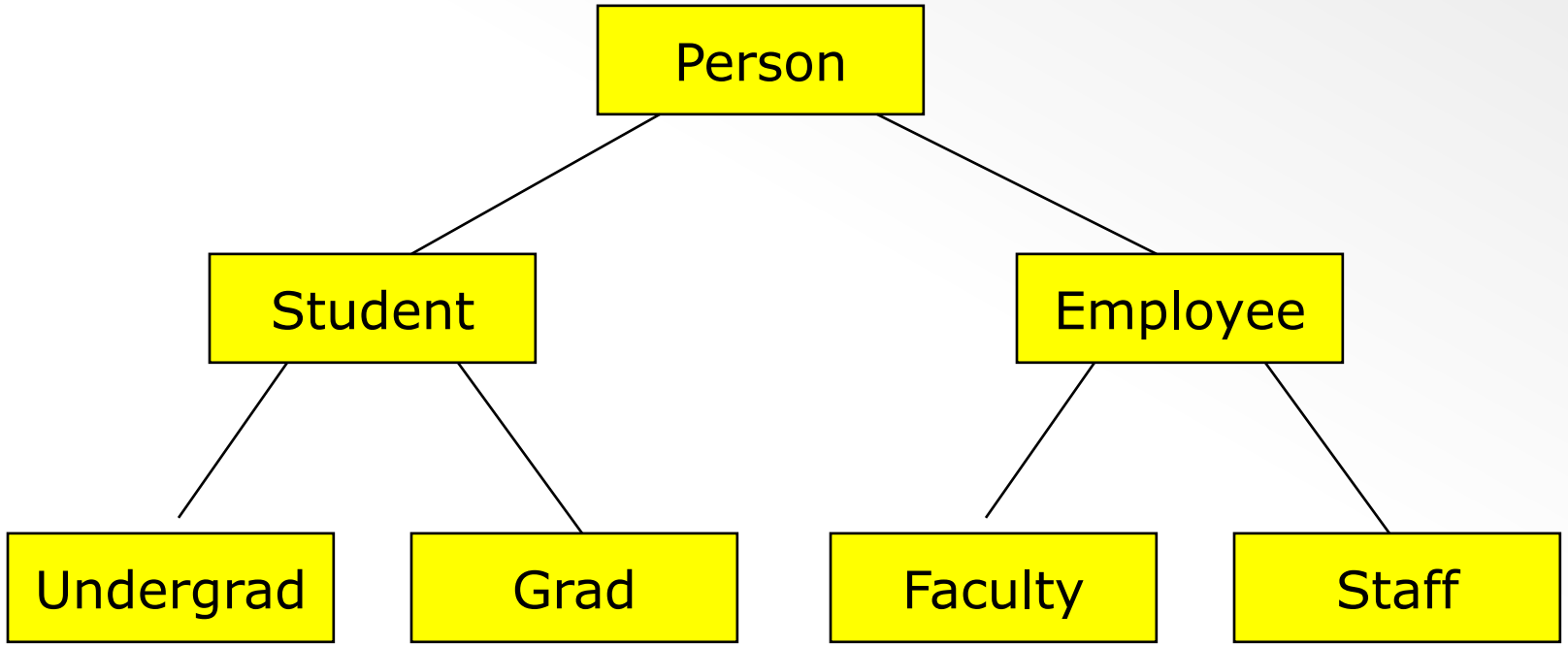
Inheritance: Concept

Suppose that we are developing a database program for record-keeping at a university and we need to design the following classes.

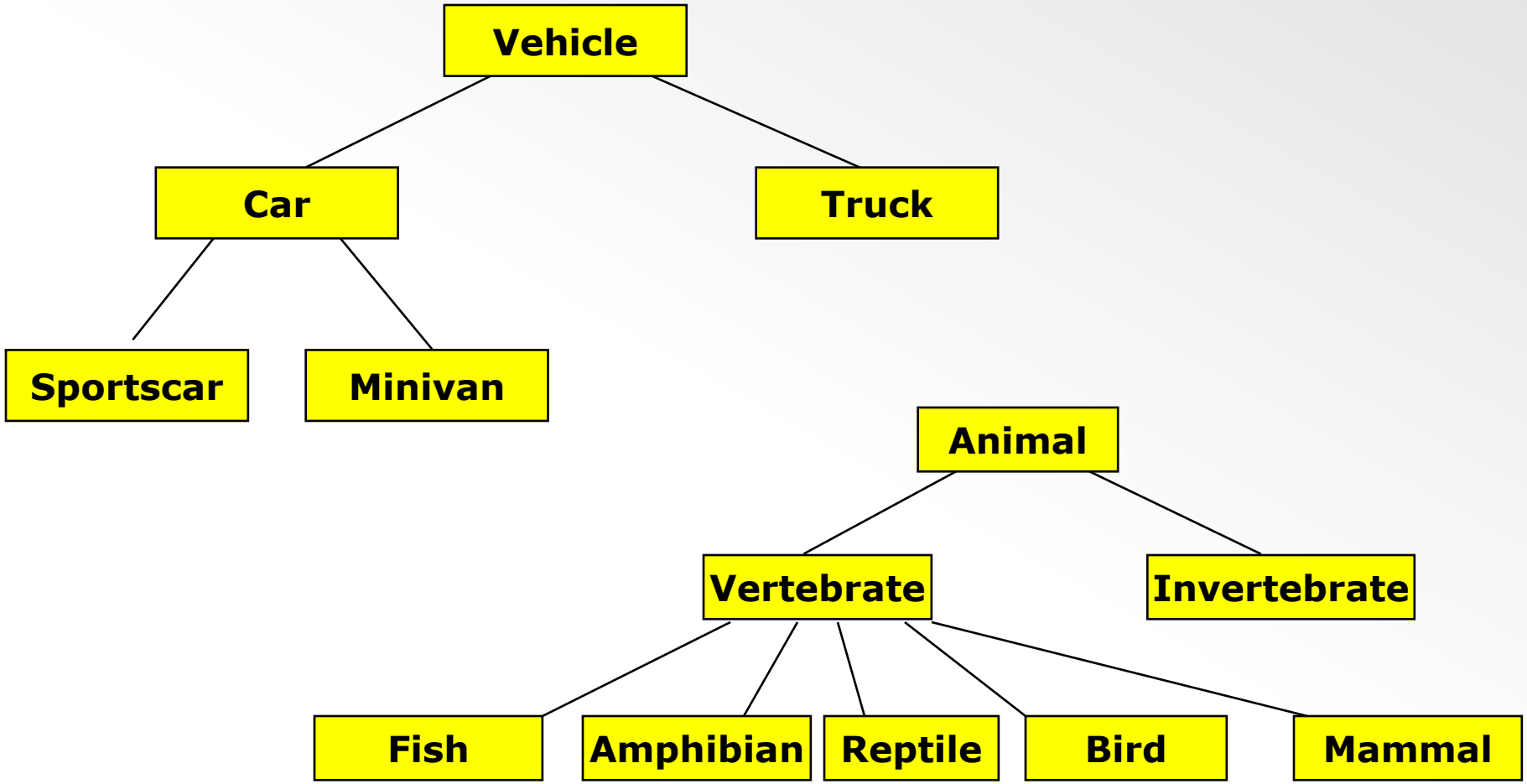


You will notice that many of the classes share common attributes. For example, all Undergrads share some common attributes with the Student class in addition to having their own attributes.

This means that we can arrange the classes in a class hierarchy.



There are many examples in which class hierarchies can be developed.

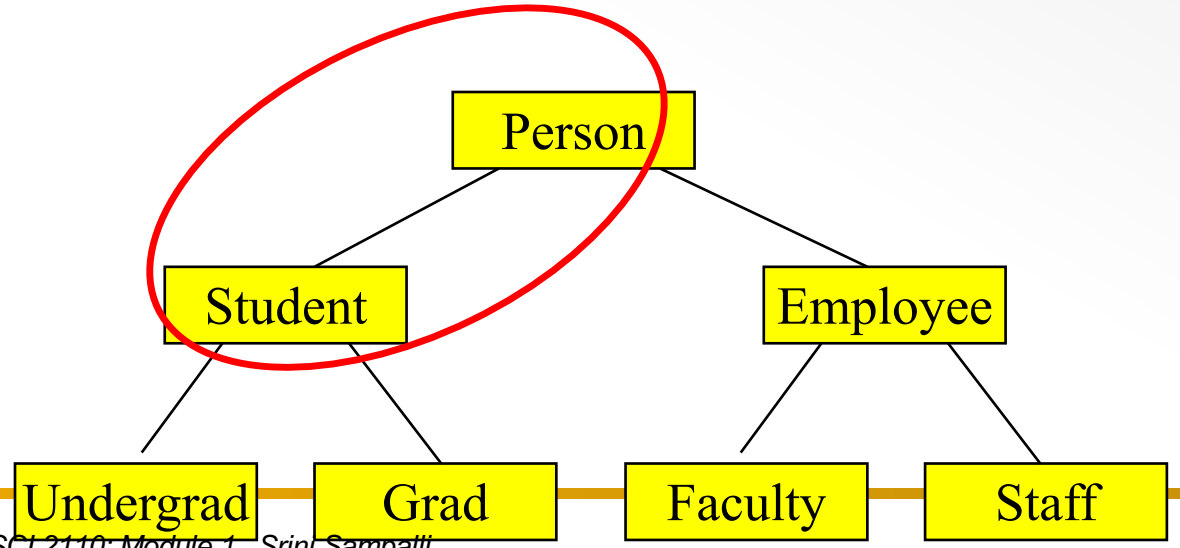


Notice that the class hierarchy can be extended as required.

When you have a class hierarchy where common attributes are shared, inheritance can be used in programming.

What is inheritance (in object-oriented programming) ?

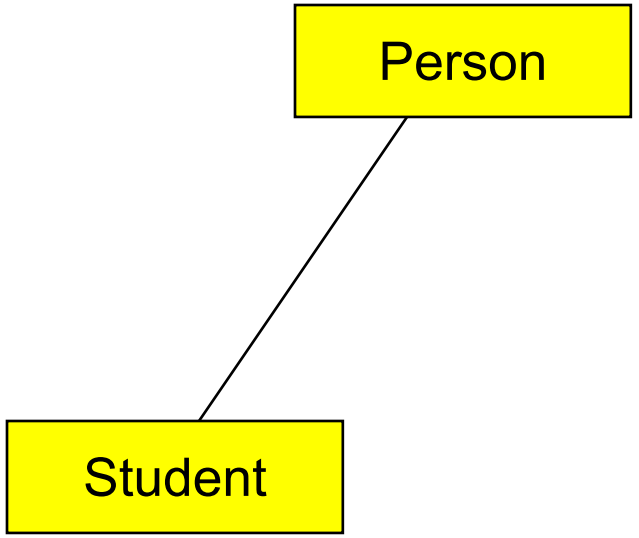
- ***With a class hierarchy, we can start the design of a class at a higher level (e.g., Person) and extend it to the class at the lower level (e.g., Student).***
- ***The Student class is said to inherit the properties of the Person class.***
- ***It will have the behaviour of the Person class plus its own specialized behaviour.***
- ***The process of deriving a new class based on an existing class is called Inheritance.***



How do we program inheritance?

We extend an existing class into a new class.

Notice the keyword "extends" in the class declaration.



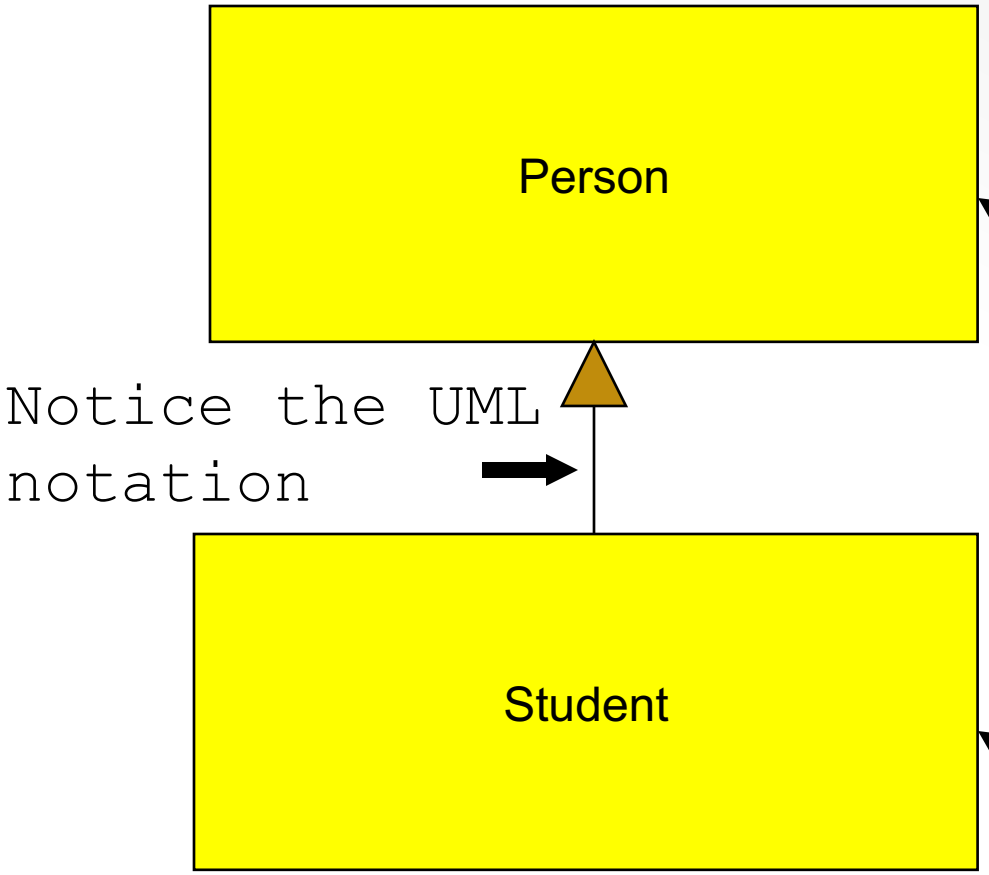
Person.java

```
public class Person
{
…
}
```

```
public class Student extends Person
{
…
}
```

Student.java

Superclass , Subclass and the UML Notation for Inheritance



Notice the UML notation

Superclass
-the generalized class
aka the base class

Subclass
-the specialized class
aka the derived class

Why use inheritance?

- The subclass will inherit the properties of the superclass.

- ***No need to repeat the instance variables of the super class (Person) in the sub class (Student).***
- ***All the methods of the parent class can be used by the sub class object.***

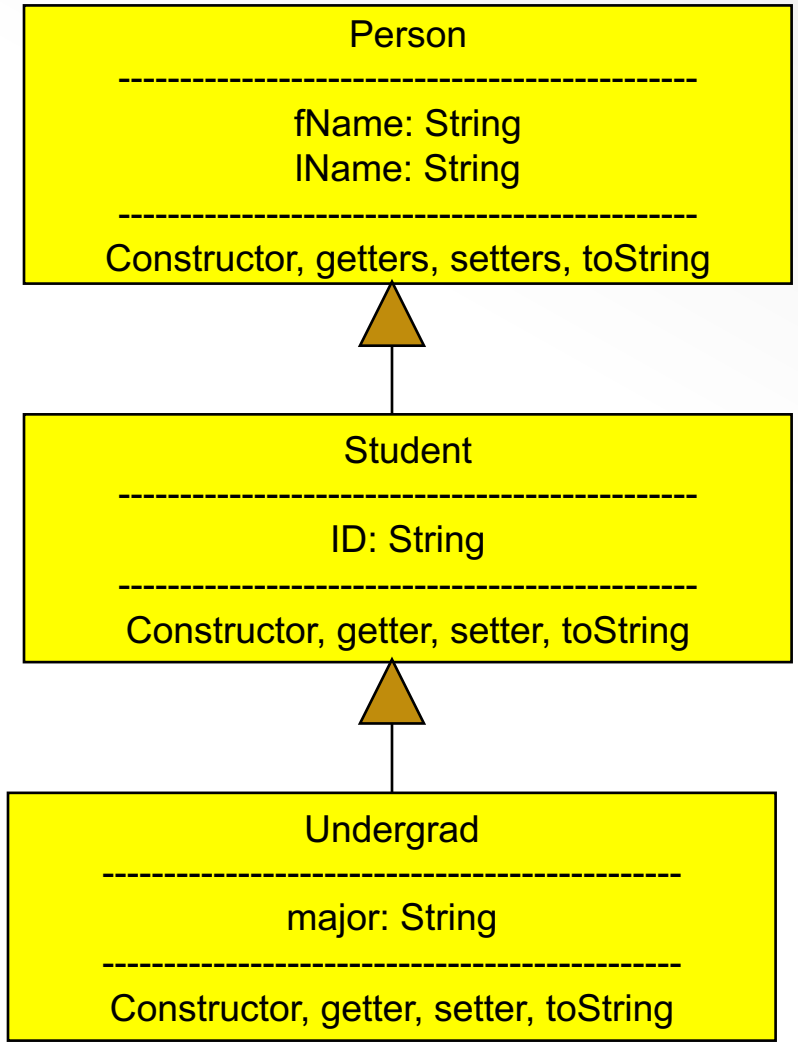
- ***Advantages:***
  - ***Code modularization.***
  - ***Code reusability.***
  - ***Accessibility of methods by other classes.***
  - ***Code extension by creating class hierarchies.***

How can you decide if two classes have an inheritance relationship?

Answer: If there is an "is-a" relationship between the classes, then it is inheritance.

- ***Car is a Vehicle → Inheritance***
- ***Rectangle is a Shape → Inheritance***
- ***Monkey is an Animal → Inheritance***
- ***Student is a Person → Inheritance***

An example program: Undergrad extends Student, Student extends Person



Person class has attributes first name and last name

Student class has attribute ID only. It inherits first name and last name. Methods in the superclass can also be used on the Student object.

Undergrad class has attribute major. It inherits first name, last name and ID, and methods from its super classes.

Person class

```
//class Person
public class Person{
   private String fName;
   private String lName;

   public Person(String fName, String lName){
         this.fName=fName;
         this.lName=lName;
   }
   public String getfName(){return fName;}
   public String getlName(){return lName;}
   public void setfName(String fName) {this.fName=fName;}
   public void setlName(String lName) {this.lName=lName;}
   public String toString(){return "Name: " + fName + " " + lName;}
}
```

Student class

```
public class Student extends Person{
        private String ID;
        public Student(String fName, String lName, String ID)
        {
            super(fName, lName);

            this.ID = ID;
        }


        public String getID(){return ID;}
        public void setID(String ID){this.ID=ID;}
        public String toString(){ return super.toString() + "\nID: " + ID;}
}
```

Calls the super class constructor

Calls the super class toString method

Undergrad class

```
//class Undergrad
public class Undergrad extends Student{
        private String major;

        public Undergrad(String fName, String lName, String ID, String major)
        {
                super(fName, lName, ID);
                this.major = major;
        }


        public String getMajor(){return major;}
        public void setID(String major){this.major=major;}
        public String toString(){ return super.toString() + "\nMajor: " + major;}
}
```

Undergrad Demo

```
//Client program that creates and uses an Undergrad student object
import java.util.Scanner;
public class UndergradDemo{
        public static void main(String[] args){
                Scanner kbd = new Scanner(System.in);
                System.out.println("Enter first name: ");
                String f = kbd.nextLine();
                System.out.println("Enter last name: ");
                String l = kbd.nextLine();
                System.out.println("Enter ID: ");
                String i = kbd.nextLine();
                System.out.println("Enter major: ");
                String m = kbd.nextLine();


                Undergrad s1 = new Undergrad(f, l, i, m);
                System.out.println("\n\n" + s1 + "\n");
        }
}
```

Method Overriding

Question: ***Can a method in the subclass have the same name and same signature as a method in the superclass?***

Answer: ***Yes!***

The method in the subclass is said to override the method in the superclass.

Abstract Class

An abstract class in Java is one from which objects cannot be instantiated.

It just defines the variables and the methods.

Methods can also be abstract.

An abstract method just has a header – no body.

Sub-classes must implement the abstract methods.

```
public abstract class Animal{
        private String type;
        public Animal(String t){type = t;}
        public void display(){System.out.println("I am an animal");}
        public abstract void makeSound();
}
```

```
public class Lion extends Animal{
        private String subtype;
        public Lion(String t, String s){
                super(t);
                subtype = s;
        }
        public void makeSound(){
                System.out.println("Roar");
        }
}
```

```
  public class LionDemo{
          public static void main(String[] args){
                  Lion l = new Lion("Mammal", "Wild");
                  l.display();
                  l.makeSound();
          }
  }
```

```
Output:
I am an animal
Roar
```

Interface

An interface is just a collection of method declarations with no data and no bodies.

That is, the methods are just method signatures.

Any class that "implements" an interface guarantees to implement all of the methods. Thus an interface enforces an API in software design.

```
public interface Sellable{
    public double listPrice(); //price of the object
    public double lowestPrice(); //lowest acceptable price
}
```

```
public class Book implements Sellable{
   private String title;
   private double price;
   public Book(String t, String p){
         title = t; price = p;
   }
   public double listPrice(){
          return price*1.5;
   }

   public double lowestPrice(){
          return price/2;
   }
}
```

The Object Class

- In Java, every class inherits implicitly from a class called Object.
- Thus, any class that extends a class A also extends the class Object by transitivity.

```
public class Rectangle {…}
```

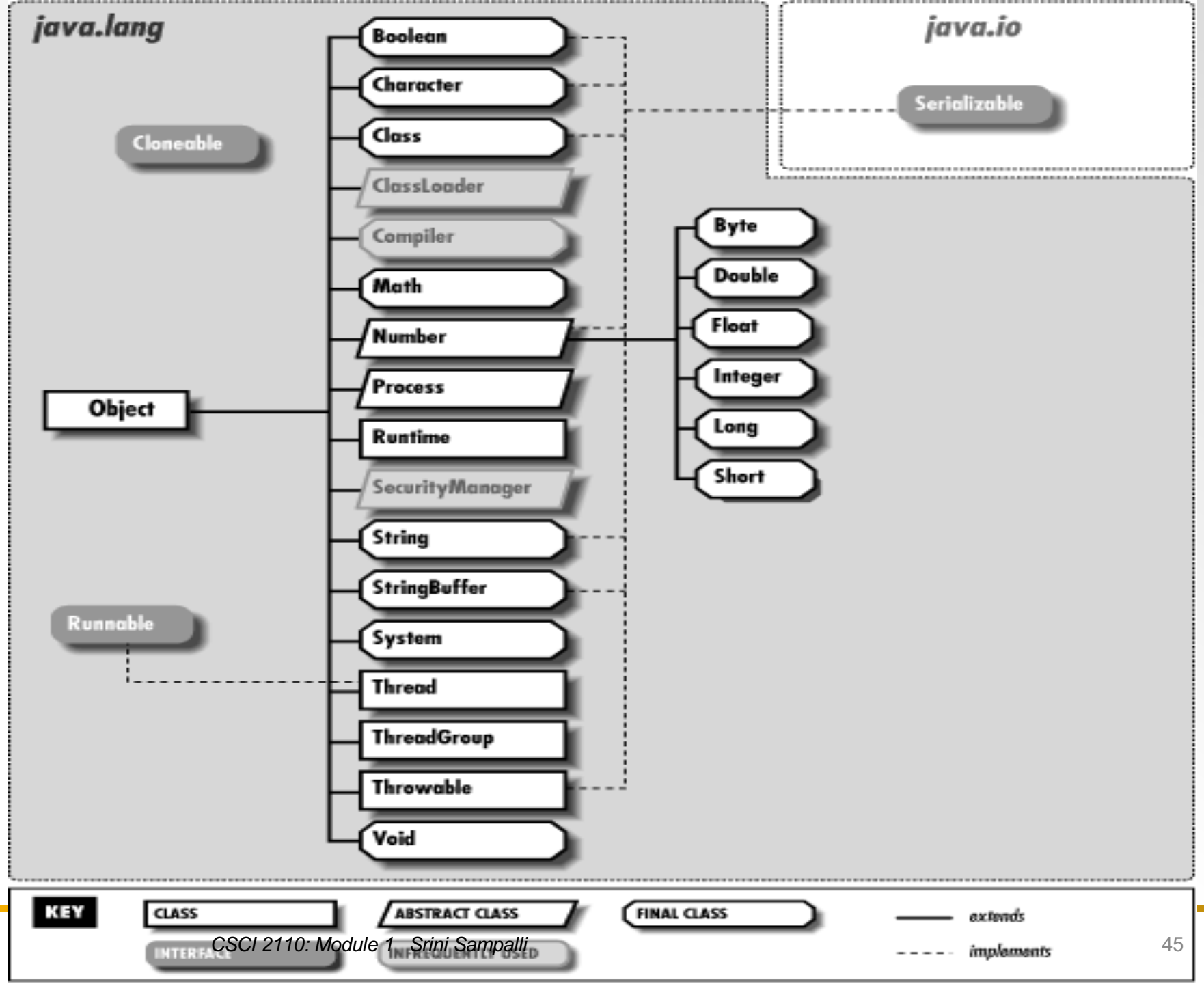
implicitly means

```
public class Rectangle extends Object {...}
```

- The Object class library provides two sets of methods:
  - Utility methods (e.g., equals, toString, clone)
  - Methods for multi-threading

The entire Java standard library is just a hierarchy of classes.

The superclass for all classes is the Object class.



POLYMORPHISM

Polymorphic:
Many forms

Polymorphism in Java

- The third pillar of object-oriented programming.
- What is polymorphism? "Polymorphic" → many forms.
- In OOP, it means the ability of one object to be treated, or used, like another.
- It works for objects in the **inheritance** hierarchy.
- In other words, we can use a reference of a superclass type to refer to an object of its subclass type.

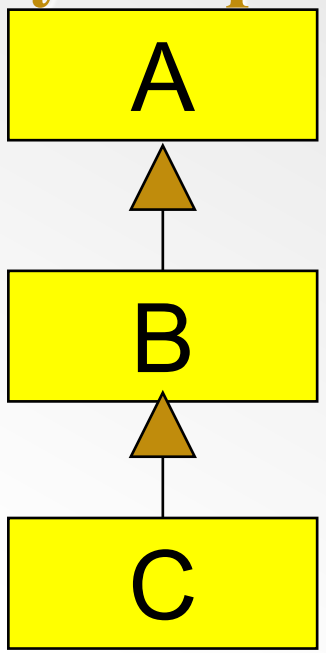
A simple example to illustrate Polymorphism

Suppose we have:

public class A{ //code … }

public class B extends A { //code …}

public class C extends B {//code …}



A simple example to illustrate Polymorphism

Suppose we have:

public class A{ //code … }

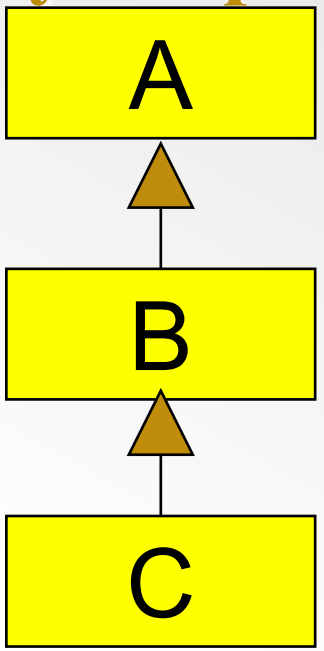
public class B extends A { //code …}

public class C extends B {//code …}

Then in the client program(main):

A obj; //obj is an object of type A

obj = new A(); → OK



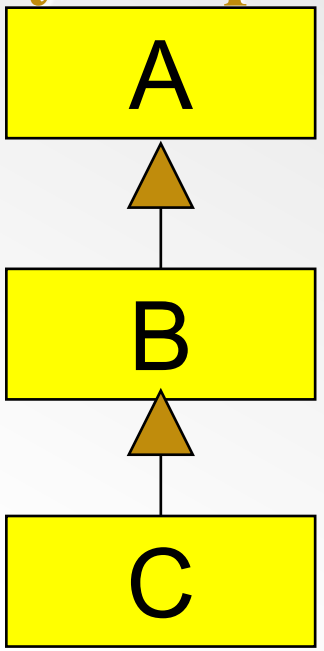
A simple example to illustrate Polymorphism

Suppose we have:

public class A{ //code … }

public class B extends A { //code …}

public class C extends B {//code …}



Then in the client program(main):

A obj; //obj is an object of type A

obj = new A(); → OK

obj = new B(); → also OK!

obj = new C(); → also OK!

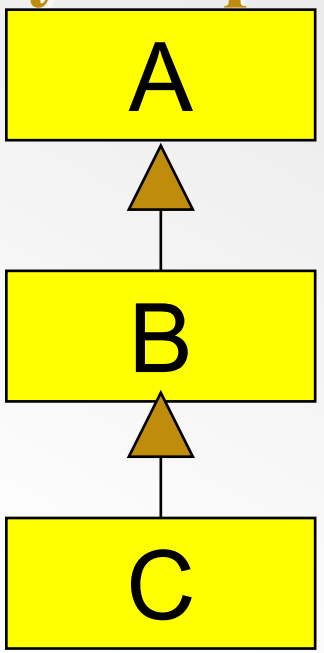
A simple example to illustrate Polymorphism

Suppose we have:

public class A{ //code … }

public class B extends A { //code …}

public class C extends B {//code …}



Then in the client program(main):

A obj; //obj is an object of type A

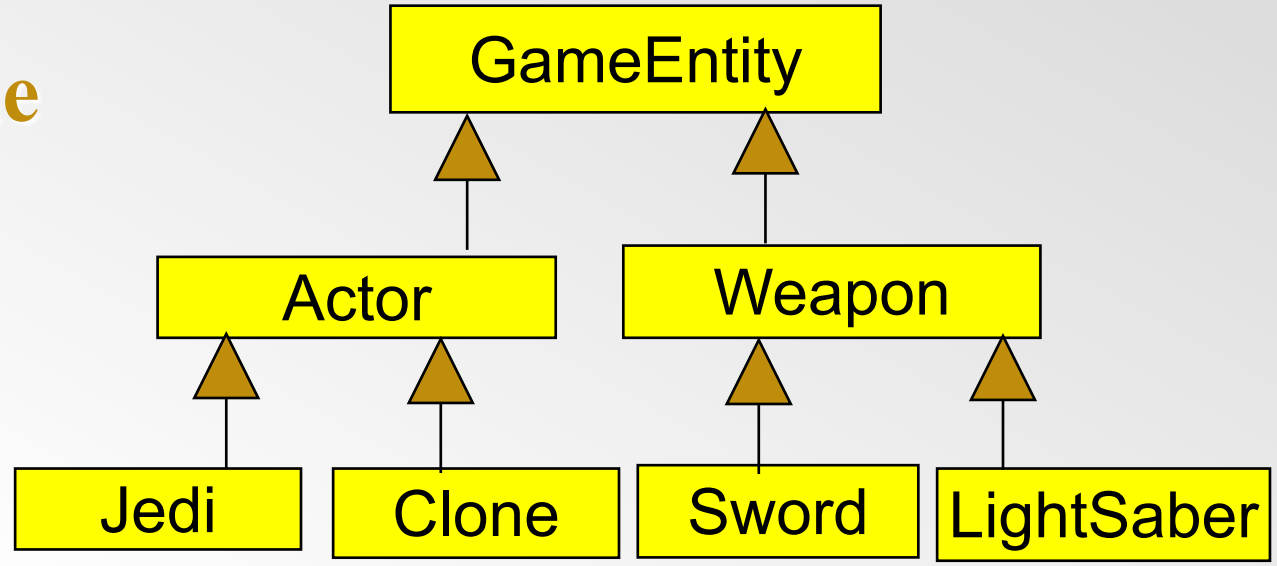
obj = new A(); → OK

obj = new B(); → also OK!

obj = new C(); → also OK!

Polymorphism allows you to declare an object of a superclass type and instantiate an object of a subclass using the same reference.

Another example: Building the Star Wars game



In a client program, we can declare:

GameEntity obj1, obj2, obj3, obj4;

and then use obj1, obj2, etc.

to create any type of object:

obj1 = new GameEntity();

obj2 = new Weapon();

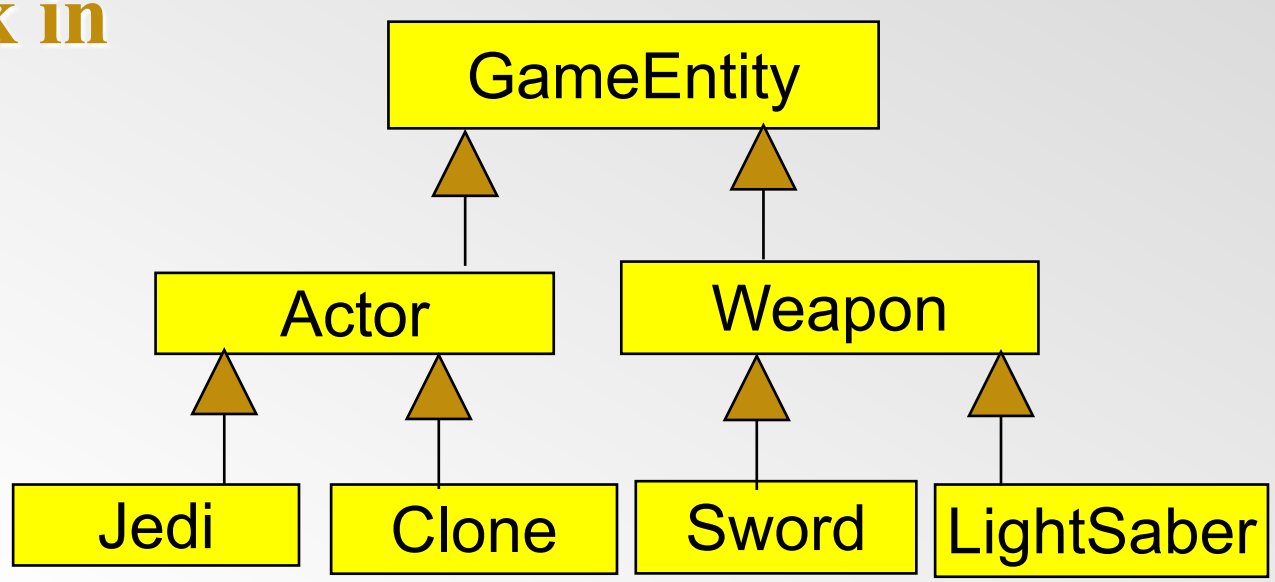
obj3 = new Jedi();

obj4 = new LightSaber();

For instance, we can declare an array of GameEntity objects and then assign them to different types of objects within the program at runtime.

GameEntity[] obj = new GameEntity[100];

Polymorphism doesn't work in reverse

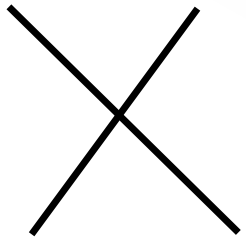


For example, suppose we have:

Weapon obj5;

then we cannot create:

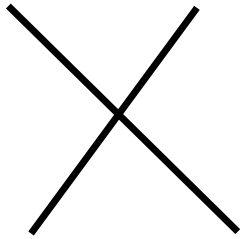
obj5 = new GameEntity();



Similarly:

Sword obj6;

obj6 = new Weapon();

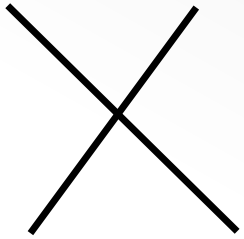


Also:

Clone obj7, obj8;

obj7 = new Weapon();

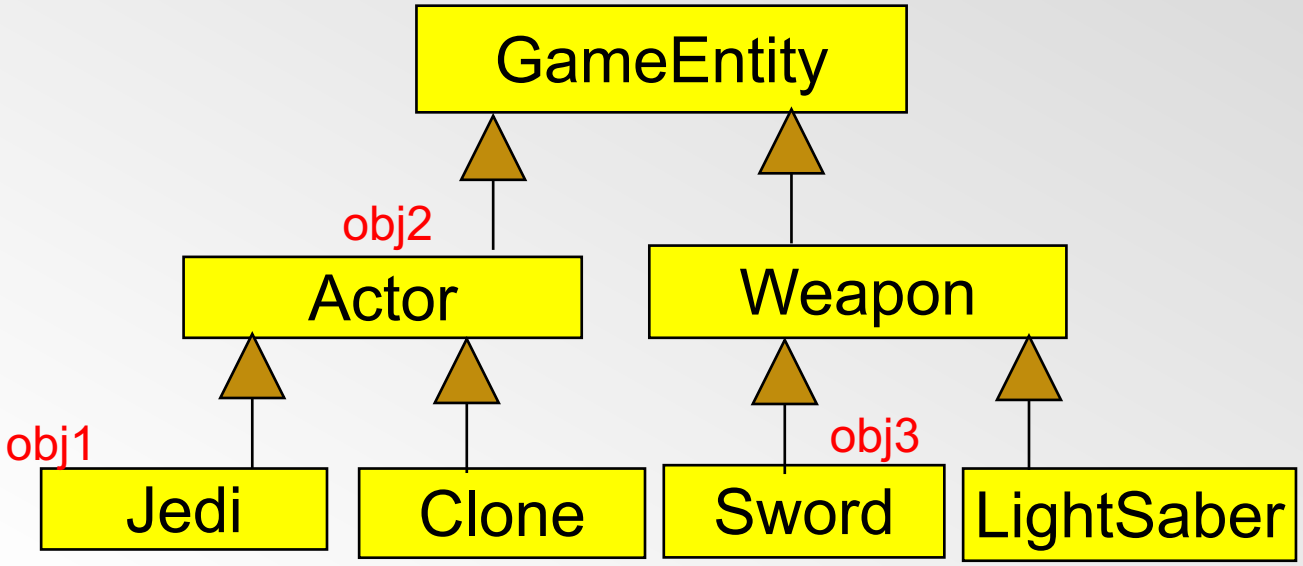
obj8 = new Sword();



The instanceof operator

- In a large program, it becomes necessary to keep track of (and test) what type of object is referred to by the variable.
- A useful operator to do this is called instanceof
- *X instanceof Y returns true if X refers to an object of type Y or a subclass (any number of levels down) of Y.*

Example to illustrate `instanceof` operator



For example, suppose we have:

```
GameEntity obj1, obj2, obj3;
obj1 = new Jedi();
obj2 = new Actor();
obj3 = new Sword();
```

Then:

System.out.println(obj1 instanceof Jedi); → will display true

System.out.println(obj2 instanceof Actor); → True

System.out.println(obj3 instanceof Sword); → True

System.out.println(obj2 instanceof GameEntity); → True

System.out.println(obj2 instanceof Clone); → False

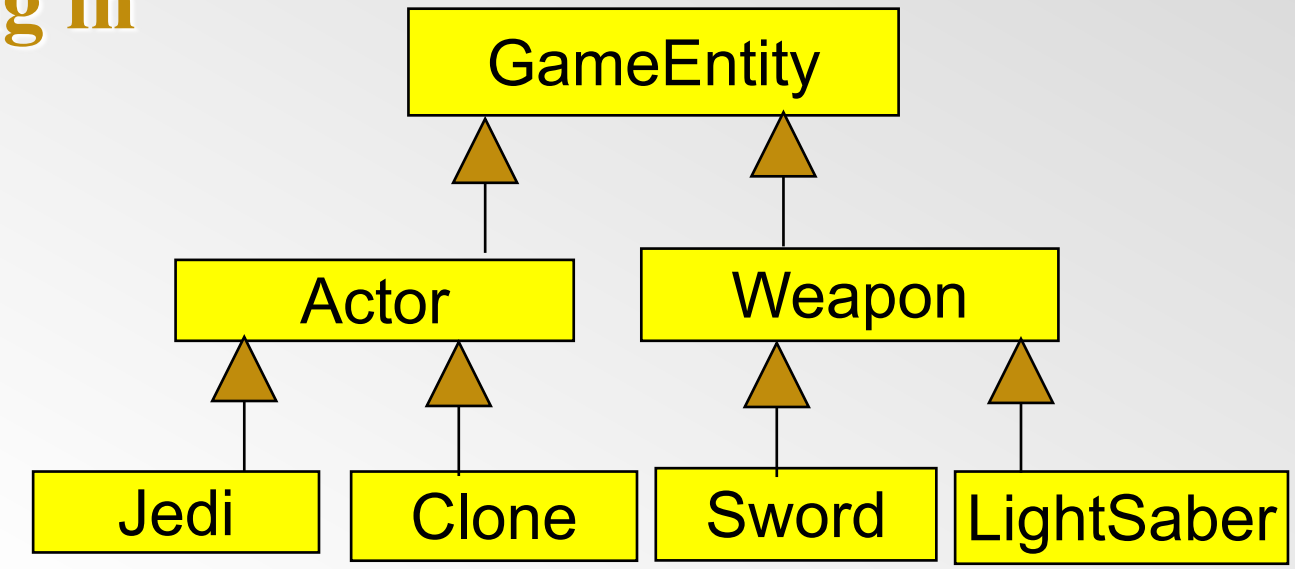
System.out.println(obj3 instanceof LightSaber); → False

System.out.println(obj3 instanceof Weapon); → True

Casting

- Polymorphism works downwards in a class hierarchy (superclass to subclass).
- Can we do the reverse (i.e., can we use the reference of a subclass type to refer to an object of a superclass type)?
- Yes – by casting (similar to casting a double as an int)

Example to illustrate casting in Polymorphism



For example, suppose we have:

```
Jedi j1, j2;
GameEntity g1;
```

Then:

```
j1 = new Jedi(); //OK
g1 = j1; //OK

g1 = new GameEntity(); //OK
j2 = g1; //not OK
j2 = (Jedi) g1; // OK – g1 has been cast to Jedi.
```

Generics in Java

A simple example to illustrate the concept and power of generics… Suppose you write a class that has a bunch of methods for a Rectangle object:

```
public class SomeClass{
   private double num;
   private char ch;
   private Rectangle rect;

   //Bunch of methods that operate on the Rectangle object

   //method to add a rectangle object
   //method to remove a rectangle object
   //method to retrieve a rectangle object
   //method to change a rectangle object
   ….etc, etc.
}
```

After you develop the class, you are told that you need a similar class to handle Circle objects, Triangle objects, Quadrilateral objects, etc….

A silly way to write the class would be…

```
public class SomeClass{
   private double num;
   private char ch;
   private Rectangle rect;
   private Circle circ;
   private Triangle tri;
   private Quadrilateral quad;

   //bunch of methods for rectangle objects
   //bunch of methods for circle objects
   //bunch of methods for triangle objects
   //bunch of methods for Quadrilateral objects
    ….etc, etc.
}
```

With Generics, you can write a class with just one set of methods that can be applied to any kind of object.

Here's the same class written using Generics ….

```
public class MyCollection<T>{
   private double num;
   private char ch;
   private T item;

   //method to add an item of type T
   public void add(T item, …)

   //method to remove an item of type T
   public void remove(…)

   //method to retrieve an item of type T
   public T get(…)

   ….etc, etc.
}
```

From a client program, you can create and manipulate different types of objects using the same class….

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

   MyCollection<Rectangle> r1 = new MyCollection<Rectangle>();

   MyCollection<Circle> c1 = new MyCollection<Circle>();

   MyCollection<Triangle> t1 = new MyCollection<Triangle>();

   ….etc, etc.
}
```

Generics Program Example:

Create a Grade class that has one instance variable called value, constructor to set the value, get and set methods and a toString method.

The Grade class must be generic such that it must work for either a String object or a Integer object.

```
//generic class Grade
public class Grade<T>{
        private T value;
        public Grade(T entry){value = entry;}
        public void setValue(T entry){value = entry;}
        public T getValue(){return value;}
        public String toString(){return "" + value;}

}

//demo client class
public class GradeDemo{
        public static void main(String[] args){
                Grade<String> m1 = new Grade<String>("A");
                Grade<Integer> m2 = new Grade<Integer>("86");
                System.out.println(m1);
                System.out.println(m2);
                m1.setValue("A+");
                m2.setValue(91);
                System.out.println(m1);
                System.out.println(m2);
        }
}
```