Contents **Topic 04 - Inheritance**

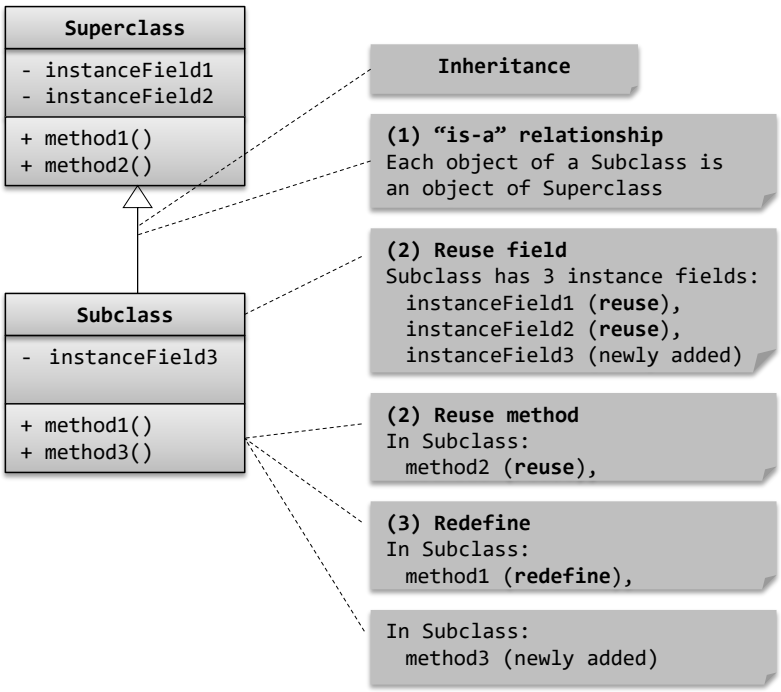
I. Classes, Superclasses, and Subclasses

Inheritance

- **Inheritance**: A fundamental concept of OO Programming
- Idea: create new classes that are built on existing classes
- Terminologies:
  - **subclass / superclass**
    The new class is called a *subclass* (*derived class*, or *child class*).
    The existing class is called a *superclass* (*base class*, or *parent class*).
  - **"is-a" relationship**
    An object of the subclass is also an object of the superclass.
    E.g. Cats are animals. Tom is a cat. Tom is also an animal.
  - **Reuse**
    All attributes/methods of the superclass can be reused (or inherited) in the subclass.
    However, constructors are not inherited.
  - **Redefine** (or called **Override**)
    The methods of the superclass can be redefined in the subclass.

UML notation for Inheritance:

- Illustrating (1) "is-a", (2) Reuse, (3) Redefine



We can add new methods and fields to adapt the sub class to new situations (instanceField3, method3)

Inheritance in JAVA:

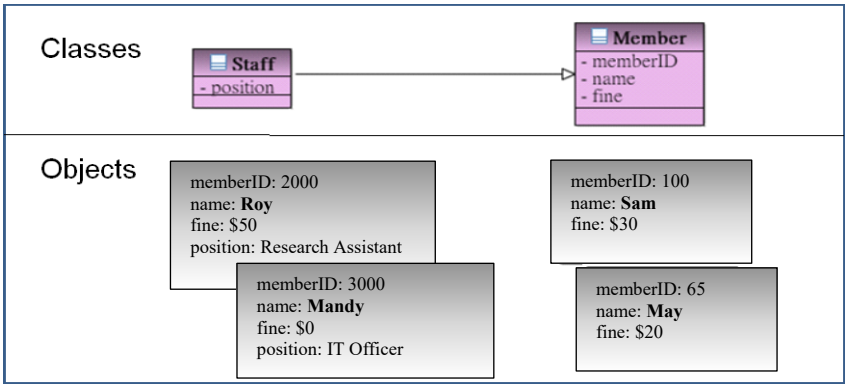
- E.g. In a university, each university staff is also a library member
  ie. an obvious "is-a" relationship.
- Java keyword **extends**:

```
class Staff extends Member
{
    added methods and fields
}
```

```
class Manager extends Employee
{
    added methods and fields
}
```

[Lab04]

- Class vs Object



* Note: in actual java implementation, the name fields store references to string objects, and the position fields store the codes of the positions

- **Subclasses -** subclasses have more data and functionality than their parent classes.

```
class Manager extends Employee
{
  .. added field, eg. bonus
  .. added methods, constructors,
  .. redefine methods in the Employee class,
     e.g. getSalary
}
```

- **Examples of Reuse:**

  [Recall]
  Reuse:
  All attributes/methods of the superclass can be reused (or inherited) in the subclass.
  However, constructors are not inherited.

```
public class Employee
{
    private String id;
    private String name;
    private double salary;

    //Accessor methods
    public String getId()   {return id;}
    public String getName()   {return nam
    ..
}
```

```
public class Manager extends Employee
{
    //A manager has an extra bonus
    private double bonus;
    ..
}
```

```
public static void main()
{
    Manager m;
    m = new Manager(..);
    System.out.println(m.getId()); ✓
    System.out.println(m.getName()); ✓
}
```

- **Examples of Redefine:**

  [Recall]
  Redefine:
  (or called Override)

  The methods of the superclass can be redefined in the subclass.

Example 1:

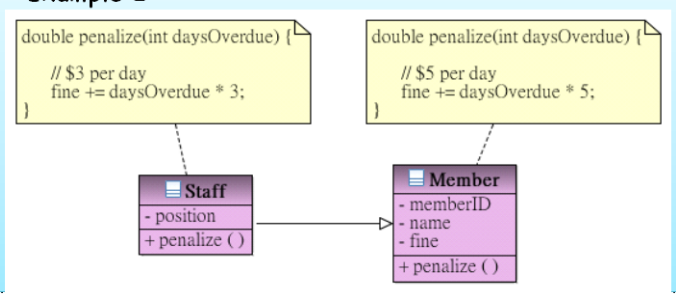
```
class Employee
{
    private double salary;
    ..
    public double getSalary() {return salary;}
}                              ①
```

```
class Manager extends Employee
{
    private double bonus;
    ..
    public double getSalary() { return super.getSalary() + bonus; }
}                             ②
```

Usage of the **super** keyword : Indicate that we are to use the Superclass method ①.
If "super." is removed, then it will call ② ➔ recursion non-stop!

Must not reduce the *visibility*

Example 2:



Access Level Modifiers

[http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html]

- Determine whether other classes can use a particular field or method.
  Or we say, "affect the visibility"

  - ❑ At the class level, the class can be
    1. ***public***: visible to all classes everywhere.
    2. **no modifier**: visible only within its own package (***package-private***)

  - ❑ At the member (field / method) level, it can be
    1. **public**
    2. **protected** – Visible to the package and all subclasses
    3. **no modifier** (package-private)
    4. **private** – Visible to the class only

Access Levels				
Modifier	**Class**	**Package**	**Subclass**	**World**
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

E.g. If a method (or a field) is **`public`**, then it can be used by the code in the same class, the package which contains this class, the subclasses which inherit this class, and the world

E.g. If a method (or a field) is **`protected`**, then it can be used only by the code in the same class, the package which contains this class, the subclasses which inherit this class, but not others from the world.

- Example:

Superclass:

```
class Employee
{
    private double salary;
    ..
    public double getSalary() {return salary;}
}
```

Subclass:

This version is OK

```
class Manager extends Employee
{
    private double bonus;
    ..
    public double getSalary() { return super.getSalary() + bonus; }
}
```

This version has error:
The field Employee.salary is not visible

```
class Manager extends Employee
{
    private double bonus;
    ..
    public double getSalary() { return salary + bonus; }
}
```

- Using "protected" for data fields is considered "against the spirit of OOP".

  Reason - It breaks data encapsulation:

  Instance fields should be treated as implementation details and encapsulated properly.
  Eg. Any change to the field (say, change the name) should not need outsiders (including subclasses) recompile.

Constructors - Review

We can provide zero or more constructors

- If we do not provide any constructor, then a default constructor is automatically generated.
  (a no-argument constructor: fields are default null or zero or false)

- 3 Examples for class Employee
  - Example 1: We write a 3-arguments constructor
  - Example 2: We do not write any constructor (Java automatically generates a no-argument constructor)
  - Example 3: We write three constructors (no argument, 1 argument, 3 arguments)

Example 1: We write a 3-arguments constructor

```
class Employee
{
    private String id;
    private String name;
    private double salary;

    public String toString() { return id + " " + name + " " + salary;}

    //Constructor with 3 arguments
    public Employee(String i, String n, double s)
    {
        id=i;
        name = n;
        salary = s;
    }
}
----------------------------------------------------
public static void main(String[] args)
{
    //Employee e0 = new Employee();

    Employee e3 = new Employee("001", "Helena", 1000);
    System.out.println(e3); //001 Helena 1000.0
}
```

Error:
The constructor Employee() is undefined

Example 2: We do not write any constructor

```
class Employee
{
    private String id;
    private String name;
    private double salary;

    public String toString() { return id + " " + name + " " + salary;}

    //not provide any constructor

}
----------------------------------------------------
void main(String[] a
{
    Employee e = new Employee();
    System.out.println(e.toString()); //null null 0.0
}
```

OK:
Since we don't write any constructor,
Java automatically generates a no-argument constructor

Example 3: We write three constructors

```
class Employee
{
    private String id;
    private String name;
    private double salary;

    public String toString() { return id + " " + name + " " + salary;}

    public Employee()
    {
        id="[-]";
        name = "[new staff]";
        salary = Math.round(Math.random()*1000);
    }

    public Employee(String i)
    {
        id=i;
    }

    public Employee(String i, String n, double s)
    {
        id=i;
        name = n;
        salary = s;
    }
}
----------------------------------------------------
public static void main(String[] args)
{
    ...
}
```

```
Employee e0 = new Employee();
System.out.println(e0); //  [-] [new staff] 191.0

Employee e1 = new Employee("001");
System.out.println(e1); // 001 null 0.0

Employee e3 = new Employee("001", "Helena", 1000);
System.out.println(e3); // 001 Helena 1000.0
```

Question:
What if we add the 4th constructor as:

```
public Employee(String nm)
{
    name = nm;
}
```

Anwer:
Error : Duplicate method Employee(String)

Compiler:
When I see `new Employee("abc")`, I cannot decide which constructor to run.

Constructors - Calling Constructors of superclass

- Constructors are not inherited
- But inside a subclass constructor

1. We can explicitly call a superclass constructor
   - must be the first statement.
2. Otherwise the no-argument constructor of the superclass is invoked.

Superclass

```
public class Employee
{
   private String id;
   private String name;
   private double salary;
   public String toString() {return id +" " + name +" " + salary;}

   public Employee() (A)
   {
      salary = Math.round(Math.random()*1000);
      //Here other fields get default values (obj fields are null)
   }

   public Employee(String i, String n, double s) (B)
   {
      id=i;
      name = n;
      salary = s;
   }
}
```

Subclass

```
class Manager extends Employee
{
   private double bonus;
   public String toString() {return super.toString() +" " + bonus;}

   public Manager(String i, String n, double s, double b) (C)
   {
      super(i,n,s);
      bonus = b;
   }
}
```

```
public static void main(String[] args)
{
   Employee e = new Manager("001", "Helena", 1000, 10);
   System.out.println(e); //001 Helena 1000.0 10.0
}
```

Superclass

```
public class Employee
{
   private String id;
   private String name;
   private double salary;
   public String toString() {return id +" " + name +" " + salary;}

   public Employee() (A)
   {
      salary = Math.round(Math.random()*1000);
      //Here other fields get default values (obj fields are null)
   }

   public Employee(String i, String n, double s) (B)
   {
      id=i;
      name = n;
      salary = s;
   }
}
```

Subclass

```
class Manager extends Employee
{
   private double bonus;
   public String toString() {return super.toString() +" " + bonus;}

   public Manager(String i, String n, double s, double b) (C)
   {
      super(i,n,s); //if this line is not given, the no-argument constructor
      bonus = b;    of the superclass is invoked
   }
}
```

```
public static void main(String[] args)
{
   Employee e = new Manager("001", "Helena", 1000, 10);
   System.out.println(e); //null null 639.0 10.0
}
```

Question:
What if we now
i. Remove B only?
ii. Remove both A and B?
iii. Remove A only?

Anwer:
(i) Remove B only: OK
(ii) Remove both A and B: OK
(A no-argument constructor is given automatically for Employee. The salary will be zero)
(iii) Remove A only: - *Implicit super constructor Employee() is undefined. Must explicitly invoke another constructor*

Polymorphism and Dynamic Binding

- **Polymorphism** – An object variable (Superclass) can refer to different actual types (Superclass and subclass). [Compile time checking]
- **Dynamic Binding** – Automatically select the appropriate non-static method (not field). [Runtime decision]

Example: Suppose we have proper constructors:

```
public Employee(String i, String n, double s)
{
    id=i;
    name = n;
    salary = s;
}
```

```
public Manager(String i, String n, double s, double b)
{
    super(i,n,s);
    bonus = b;
}
```

"Polymorphism" – compile time checking
Using an Employee variable b to refer to a Manager

Failed:
Attempt to assign Manager variable c to refer to the new Employee object.

Compilation error message:
cannot convert from Employee to Manager

"Dynamic binding" – runtime decision
Choose the correct .toString() method for the Manager, although the manager is referred by Employee variable b.

```
public static void main(String[] args)
{
   Employee a = new Employee("001", "Alice", 1000);
   Employee b = new Manager("902", "Brian", 1000, 10);

   //Manager c = new Employee("003", "Candy", 1000);

   Manager d = new Manager("904", "Daisy", 1000, 10);

   System.out.println(a.toString()); // 001 Alice 1000.0
   System.out.println(b.toString()); // 902 Brian 1000.0 10.0
   System.out.println(d.toString()); // 904 Daisy 1000.0 10.0
}
```

Preventing Inheritance: final Classes and Methods

- **Final class and Final method** – Avoid being inherited or redefined

  Example of final class:

```
final class Executive extends Manager
{
    ...
}
```

  Example of final method:

```
class Employee
{
    ...
    public final String getName() {return name;}
}
```

If we redefine .getName() in the Manager class, we get Error:
Cannot override the final method from Employee

- **In Java, the String class is a final class**

  E.g. the following is not allowed.

```
static class StringSubClass extends String
{
    ..
}
```

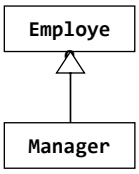
Error: *StringSubClass cannot extend the final class String*

Casting and instanceOf

- **Casting**: **Consider the type of an object as a different type**
  Note: you are not actually changing the object itself.

Two types of casting:

```
Manager m1 = new Manager("902", "Brian", 1000, 10);
Employee e1 = m1; //upcasting
Manager m;
m = (Manager)e1; //downcasting
System.out.println(m.getBonus());
```



upcasting: label a subclass object reference as a superclass.
- It is done automatically (implicitly): You DO NOT need to add (*Superclass*) for explicit casting.
- Always successful at run time.
- Example of use: a subclass object (like Manager) is added as an element in a collection of the superclass (Employee[])

```
class Employee    {
    private String id;
    private String name;
    private double salary;
    public String toString() {return id + " " + name + " " + salary;}
    .. //constructor and other methods
}

class Manager extends Employee {
    private double bonus;
    public String toString() {return super.toString() + " " + bonus;}
    .. //constructor and other methods
}

public static void main(String[] args) {
    Employee[] allEmployees;
    allEmployees = new Employee[3];
    allEmployees[0] = new Employee("001", "Alice", 1000);
    allEmployees[1] = new Manager("902", "Brian", 1000, 10); // upcasting
    allEmployees[2] = new Manager("904", "Daisy", 1000, 15); // upcasting

    for (Employee e: allEmployees)
        System.out.println(e);
}
```

downcasting: label a superclass object reference as a subclass.
- It requires explicit casting: You need to add (*subclass*) for explicit casting.
- Example of use: To use an object 's actual features after its actual type has been temporarily forgotten.

Given an array: **Employee[] allEmployees**, each **allEmployees[i]** belongs to the **Employee** Type
Suppose we know that **allEmployees[2]** is actually a **Manager**. **We want to run .getBonus()**.
However **allEmployees[2].getBonus()** won't work because the type of **allEmployees[2]** is not **Manager**

```
public static void main(String[] args) {
    Employee[] allEmployees;
    allEmployees = new Employee[3];
    allEmployees[0] = new Employee("001", "Alice", 1000);
    allEmployees[1] = new Manager("902", "Brian", 1000, 10); // upcasting
    allEmployees[2] = new Manager("904", "Daisy", 1000, 15); // upcasting

    Manager m;
    m = (Manager) allEmployees[2]; // downcasting
    System.out.println(m.getBonus());
}
```

Be careful of casting problem during run time!!

```
class Manager extends Employee
{
      private double bonus;
      public double getBonus() {return bonus;}
      ..
}
--------------------------------------------------------------------------------
public static void main(String[] args)
{
   Employee[] allEmployees;
   allEmployees = new Employee[3];
   allEmployees[0] = new Employee("001", "Alice", 1000);
   allEmployees[1] = new Manager("902", "Brian", 1000, 10); //upcasting
   allEmployees[2] = new Manager("904", "Daisy", 1000, 15); //upcasting

   for (Employee e: allEmployees)
   {
         Manager m;
         m = (Manager)e;  //Runtime error!!!  Alice is not a manager.  (Program stops running
         System.out.println(m.getBonus());
   }
}
```

Runtime Error:
Employee cannot be cast to Manager at Main.main(Main.java:99)

Solution:
May use the instanceof operator to check the class first:

```
for (Employee e: allEmployees)
{
   if (e instanceof Manager)
   {
      Manager m;
      m = (Manager)e;
      System.out.println(m.getBonus());
   }
}
```

Output:
10.0
15.0

It works now. ☺

- Use of **`instanceOf`**

  - An object variable is declared with a type, eg. **`Employee e;`**
    Then the object variable can refer to an object of the type, or its subclass.

  - The **`instanceof`** operator compares an object to a class.
    Syntax: **`x instanceof C`**
    where **x** is an object and **C** is a class

```
for (Employee e: allEmployees)
{
   if (e instanceof Manager)
   {
      Manager m;
      m = (Manager)e;
      System.out.println(m.getBonus());
   }
}
```

Try:
```
System.out.println(e instanceof Manager);
System.out.println(e instanceof Employee);
```

Output:
true
true

- Where are `instanceof` and `cast` among other operators?

[Core Java Chp 3.5.7)

precedence level

Operators	Associativity
[] . () (method call)	Left to right
! ~ ++ -- + (unary) - (unary) () (cast) new	Right to left
* / %	Left to right
+ -	Left to right
<< >> >>>	Left to right
< <= > >= instanceof	Left to right
== !=	Left to right
&	Left to right
^	Left to right
|	Left to right
&&	Left to right
||	Left to right
?:	Right to left
= += -= *= /= %= &= |= ^= <<= >>= >>>=	Right to left

- Note: `instanceof` is **often not needed**

```
for (Employee e: allEmployees)
    System.out.println(e.getSalary());
```

Beginners often use `instanceof` in an unnecessary way:
e.g. make decision about using which version of `.getSalary()`

As a proper practice, we should often let JAVA select the appropriate (redefined) method at run-time (Dynamic Binding).
You will know that this follows important OO principle
(e.g. OCP: Open-Close Principle / Lab06)

Abstract Classes

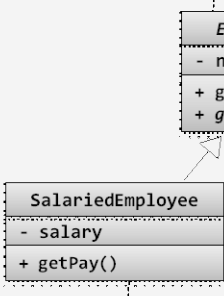
- **Abstract method**
  - a method with the `abstract` keyword;
  - no implementation
  - It acts as placeholders for *concrete* (i.e. nonabstract) methods that are implemented in the subclasses.

```
public abstract class Employee
{
    private String name;
    public abstract double getPay( );
    public Employee(String aName)  {name = aName;}
    public String getName( ) {return name; }
}
```

- **Abstract classes**
  - Abstract classes cannot be instantiated. i.e., we cannot create new object using an abstract class.
  - We can declare an object variable whose type is an abstract class.
  - Then the object variable can refer to an object of its *concrete* (i.e. nonabstract) subclass.
  - A class which has one or more abstract methods must be declared abstract.

An abstract superclass class
i.e. it contains abstract method(s)

```
public abstract class Employee
{
  private String name;
  public Employee(String n) {name = n;}
  public String getName( ) {return name;}
  public abstract double getPay( );
}
```




```
public class SalariedEmployee extends Employee
{
   private double salary; //annual
   public SalariedEmployee(String n, double s)
   {
      super(n); salary = s;
   }
   public double getPay( ) {return salary/12;}
}
```

A concrete subclass:
i.e. all methods (getPay(), getName() are concrete)

```
public class HourlyEmployee extends Employee
{
   private double wageRate;
   private double hours; //for the month
   public HourlyEmployee(String n, double w, double h)
   {
      super(n); wageRate = w;  hours = h;
   }
   public double getPay( ) {return wageRate*hours;}
}
```

A concrete subclass:
i.e. all methods (getPay(), getName() are concrete)

- **Abstract object variable and concrete object:**

Declare object variables whose types are abstract (here `arr[0..2],e`)

Then `arr[0..2]` and `e` can refer to objects of the *concrete* subclasses (i.e. nonabstract ones).

```
public static void main(String[] args)
{
   Employee[] arr = new Employee[3];
   arr[0] = new HourlyEmployee("Helena",52.5,30);
   arr[1] = new SalariedEmployee("Kit", 15000);
   arr[2] = new HourlyEmployee("Jason", 100,60);

   for (Employee e: arr)
      System.out.println(e.getName()+
         " ($" + e.getPay()+")");
}
```

Output:
```
Helena ($1575.0)
Kit ($1250.0)
Jason ($6000.0)
```

II. Object: The Cosmic Superclass

- **The Java's `Object` class: `java.lang.object`**
  - ❑ Every class automatically "is-a" subclass of the **`Object`** class. (Inheritance)
  - ❑ Every object of every class is of type **`Object`**.

```
Object o1 = new Employee("002", "Jim", 10000);
System.out.println(o1); //002 Jim 10000.0
System.out.println(o1.getClass().toString()); //class Employee
System.out.println(o1 instanceof Object); //true
System.out.println(o1 instanceof Employee); //true
Object o2 = "Hello";
System.out.println(o2); //Hello
System.out.println(o2.getClass().toString()); //class java.lang.String
System.out.println(o2 instanceof Object); //true
System.out.println(o2 instanceof Employee); //false
```

  - ❑ Object methods: `equals, toString, getClass, clone` etc..
    - automatically inherited in every class
    - `equals, toString` : we usually need to override appropriately
    - `getClass` : returns a `Class` object which represents the class itself
    - `clone` : returns a copy of an object

- **The `equals` method of the `Object` class:**

```
//java.lang.object.equals:
public boolean equals(Object obj)
```

- The Right way to **override** `equals` for a class - note the explicit parameter

```
class ClassName
{
   ..
   public boolean equals(Object obj){..}
}
```

Note: To override a method in a subclass, we must give exactly the same signature (method name + parameter list) and return type. To avoid mistake, use the **override annotation**.

- The **override annotation** :
  - ❑ Denotes that the annotated method is required to override a method in the superclass.
  - ❑ Helps us check for misspelling (eg. `equals`), or wrong parameter list etc..

```
@Override
public boolean equal(Object otherObject)
{
```

The method equal(Object) must override or implement a supertype method

But the superclasses do not have .equal(Object)

"Oh! Probably typing mistake!"

- The Right way to **override** `equals` for a class:

Use the `@Override` annotation

Parameter: `Object`

Check against `null`

Compare the classes

Cast to our class type

Check the fields one by one
- use `.equals` for object fields
- use == for primitive fields

Call `.equals` of superclass:
`super.equals(otherObject);`

```
class SubjectResult
{
   private String name; //e.g. "Chemistry", "Geography"
   private char grade; //e.g. 'A', 'B', 'C'

   SubjectResult(String n, char g) {name=n; grade=g;}

   @Override
   public boolean equals(Object otherObject)
   {
      if (otherObject == null)
         return false;

      if (this.getClass() != otherObject.getClass())
         return false;

      SubjectResult otherSR = (SubjectResult) otherObject;

      if (!this.name.equals(otherSR.name))
         return false;
      if (this.grade!=otherSR.grade)
         return false;

      return true;
   }
}
```

```
SubjectResult s1 = new SubjectResult("Chemistry",'A');
SubjectResult s2 = new SubjectResult("Physics",'A');
System.out.println(s1.equals(s2));//false
```

III Generic Array Lists

- **The Java's Generic `ArrayList`: `java.util.ArrayList`**

```
Array Sample

Integer[] arr;
arr = new Integer[3];

arr[0]= 100;
arr[1]= 101;
arr[2]= 109;

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

```
ArrayList Sample

ArrayList<Integer> arrlist;
arrlist = new ArrayList<Integer>();

arrlist.add(100);
arrlist.add(101);
arrlist.add(109);

for (int i=0;i<arrlist.size();i++)
   System.out.println(arrlist.get(i));
```

  - ❑ `java.util.ArrayList` is a very useful class which is:
    - Similar to an array, for storing a collection of object elements
    - Automatically adjusts its capacity
    - Need a type parameter to specify the type of elements.
      Syntax: `ArrayList<Element_type>`
    - Add / retrieve / remove elements: `.add(obj), .get(index), .remove(index)`
    - The count of elements: `.size()`

  - ❑ With polymorphism, an ArrayList of super-type can refer to the same type as well as sub-type objects. (e.g. Employee / Manager)

  - ❑ Starting from Java 7, we can omit the type argument when new is used.
    ie. `arrlist = new ArrayList<Element_Type>();` ← OK
    `arrlist = new ArrayList<>();` ← Also OK. The compiler itself will check what type is needed.

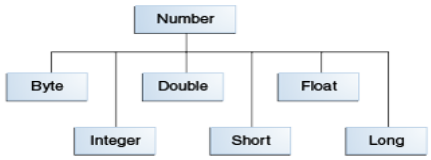
`<>` is called the diamond syntax

IV Object Wrappers and Autoboxing

- **The Java primitive types and Wrappers**
  - ❑ Recall: Java has 8 primitive types:
    `int, long, float, double, short, byte, char, boolean`
  - ❑ Each primitive type has a class counterparts which is called a **Wrapper Class**:
    `Integer, Long, Float, Double, Short, Byte, Character, Boolean`
  - ❑ Example:
```
Integer x = 3; //Autoboxing
```

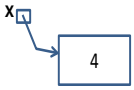
- **Java collections (like `ArrayList`) – require Wrappers**
  - ❑ Java collections (like `ArrayList`) can only store object references, not primitive types (`Integer`✓, `int`✗)
    OK: `ArrayList<Integer>`
    Invalid: `ArrayList<int>`
  - ❑ **Autoboxing** is done upon getting and setting:
```
ArrayList<Integer> arrlist = new ArrayList<>();
arrlist.add(123); // automatically translated to arrlist.add(Integer.valueOf(123));
```

- **More about Wrapper classes:**
  - ❑ The **`Number`** class: the superclass of Integer etc..
  - ❑ Wrapper classes are **final**, so we cannot subclass them.
  - ❑ Wrapper classes are **immutable** – we cannot change a wrapped value after the wrapper has been constructed.



http://docs.oracle.com/javase/tutorial/java/data/numberclasses.html

```
Integer x = 3;    // Autoboxing
//x.setValue(4);  ✗ there is no method like this (i.e. Integer is immutable)
x = 4;            // This new value, 4, is wrapped to create another Integer object.  x is now set to refer to this new object.
```



New Integer object to wrap the int value of 4

The `Number` Class

- **All subclasses of** Number **provide the following methods:**
  - ❑ Conversion from this object to primitive data
  - ❑ Compare this object with another explicit object
  - ❑ Equals

Method
byte byteValue()
short shortValue()
int intValue()
long longValue()
float floatValue()
double doubleValue()
int compareTo(Byte anotherByte)
int compareTo(Double anotherDouble)
int compareTo(Float anotherFloat)
int compareTo(Integer anotherInteger)
int compareTo(Long anotherLong)
int compareTo(Short anotherShort)
boolean equals(Object obj)

- **Each also provide additional methods for conversion.**
  E.g. The Integer class provide these methods:

Method
static int parseInt(String s) // Returns an integer (decimal only).
String toString() // Returns a String object representing the value of this Integer
static Integer valueOf(int i) // Returns an Integer object holding the value of the specified primitive
static Integer valueOf(String s) // Returns an Integer object holding the value of the given string

V Design Hints for Inheritance

Inheritance - Design Hints:

1. Place common operations and fields in the superclass.

2. Don't use protected fields
   - Using "protected" for data fields is considered "against the spirit of OOP".
   - Reason: It breaks data encapsulation:

     Instance fields should be treated as implementation details and encapsulated properly.
     Eg. Any change to the field (say, change the name) should not need outsiders (including subclasses) recompile.

[Topic03]

Recall: **Encapsulation (sometimes called information hiding)**

- Simply combining data and behavior in one package, which are considered as implementation details
- Implementation details are hidden away from the users of the objects
- E.g., We can use `Scanner` objects, but their implementation details are encapsulated in the `Scanner` class.

It is a good practice to encapsulate data as **private** instance fields.

1. **Protect the data from corruption by mistake.**
   Outsiders must access the data through the provided public methods
2. **Easier to find the cause of bug**
   Only methods of the class may cause the trouble
3. **Easier to change the implementation**
   - e.g. change from using 3 integers for `year,month,day` to using 1 integer `yyyymmdd`
   - We only need to change the code in the class for that data type
   - The change is invisible to outsiders, hence not affect users.

   - However, protected methods can be useful to indicate methods that are not ready for general use and should be redefined in subclasses (eg. `.clone()`)

3. Use inheritance to model the "is-a" relationship. Don't use inheritance unless all inherited fields and methods make sense.
   - Do not extend a class if your intension is just to reuse a portion (<100%) of the superclass.

4. Don't change the expected behavior when you override a method.

5. We should use polymorphism and rely on dynamic binding.
   - Do not explicitly check the type (Ref. Lab06 page 1 Approach 1)

--- end ---