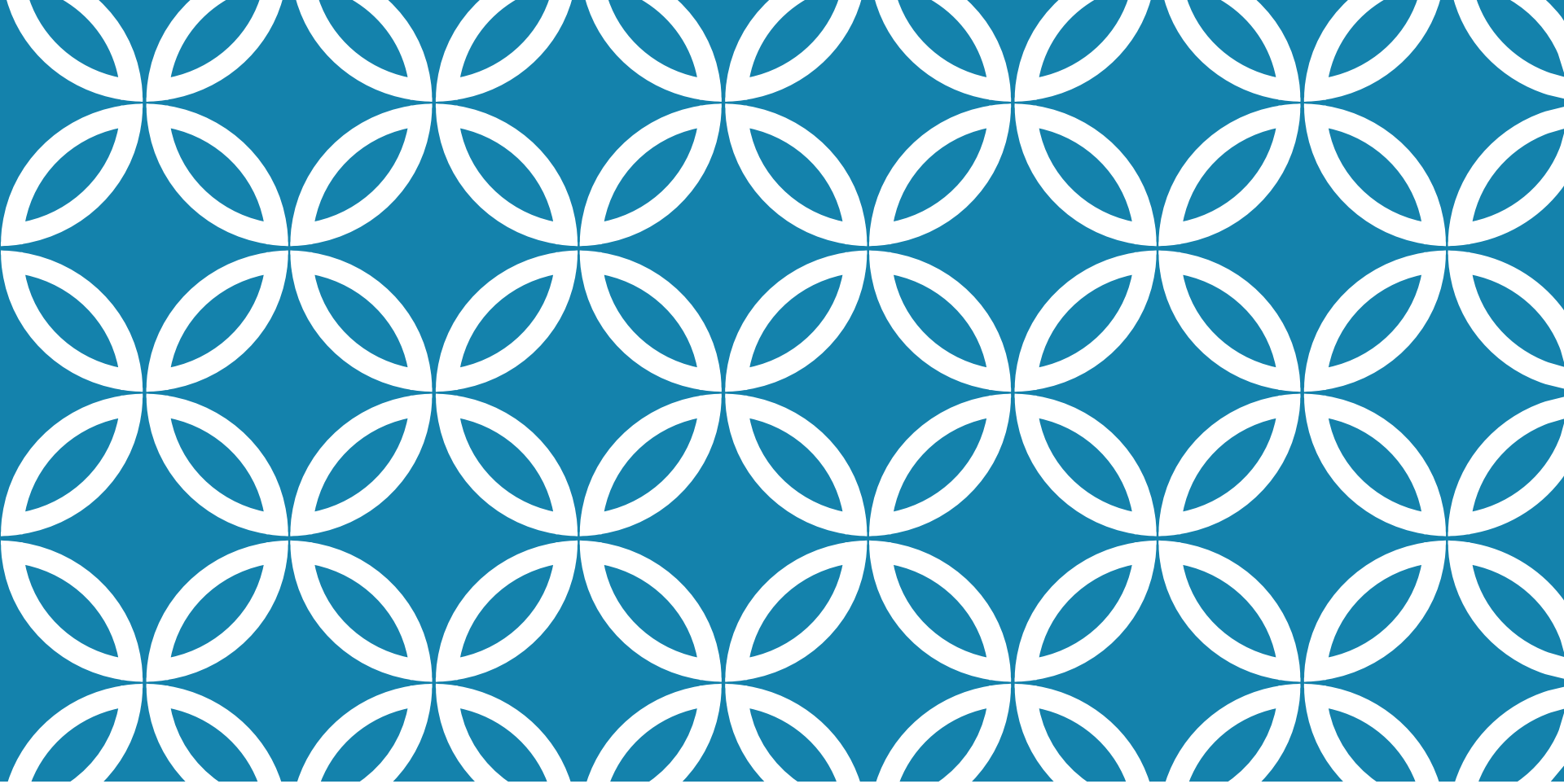


CHAPTER 9
PACKAGES AND INTERFACES

PACKAGES

- Till now all java classes were stored in same name space.
- As the applications grows it becomes difficult to keep all the files in same directory, also we need to ensure that the name we choose should be unique. To address this problem java introduces a mechanism to manage all the classes this is called packages
- packages are containers for classes that are used to keep the class name space compartmentalized.
- It is both a naming and a visibility control mechanism

PACKAGES

- To create a package simply include a package command as the first statement in a Java source file.
- Any classes declared within that file will belong to the specified package .
- The package statement defines a name space in which classes are stored.
- If we omit the ***package*** statement, the class names are put into the default package, which has no name
- General form of the **package** statement:

  package *pkgname*;

- Eg: package mypackage;

Package and CLASSPATH

- How does the Java run-time system know where to look for packages that you create?
- First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found.
- Second, you can specify a directory path or paths by setting the CLASSPATH environmental variable.
- Third, you can use the –classpath option with java and javac to specify the path to your classes
- For eg consider *package MyPackage*
- In order for a program to find MyPackage, one of three things must be true. Either the program can be executed from a directory immediately above MyPackage, or the CLASSPATH must be set to include the path to MyPackage, or the –classpath option must specify the path to MyPackage when the program is run via java.

Package Example

```
// A simple package
package MyPack;

class Balance {
  String name;
  double bal;

  Balance(String n, double b) {
    name = n;
    bal = b;
  }

  void show() {
    if(bal<0)
      System.out.print("--> ");
    System.out.println(name + ": $" + bal);
  }
}
```

Package Example

```
class AccountBalance {
  public static void main(String args[]) {
    Balance current[] = new Balance[3];
    current[0] = new Balance("K. J. Fielding", 123.23);
    current[1] = new Balance("Will Tell", 157.02);
    current[2] = new Balance("Tom Jackson", -12.33);

    for(int i=0; i<3; i++) current[i].show();
  }
}
```

- Java uses file system directories to store packages.
- The **.**class files for any classes you declare to be part of MyPack must be stored in a directory called MyPack/ is significant, and the directory name must match the package name exactly.
- More than one file can include the same **package** statement.
- The package statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package.

- To create package hierarchy separate each package name from the one above it by use of a period.

```
package pkg1[.pkg2[.pkg3]];
package java.awt.image;
```

- A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as

```
package
oop.amp.btech.packages
```

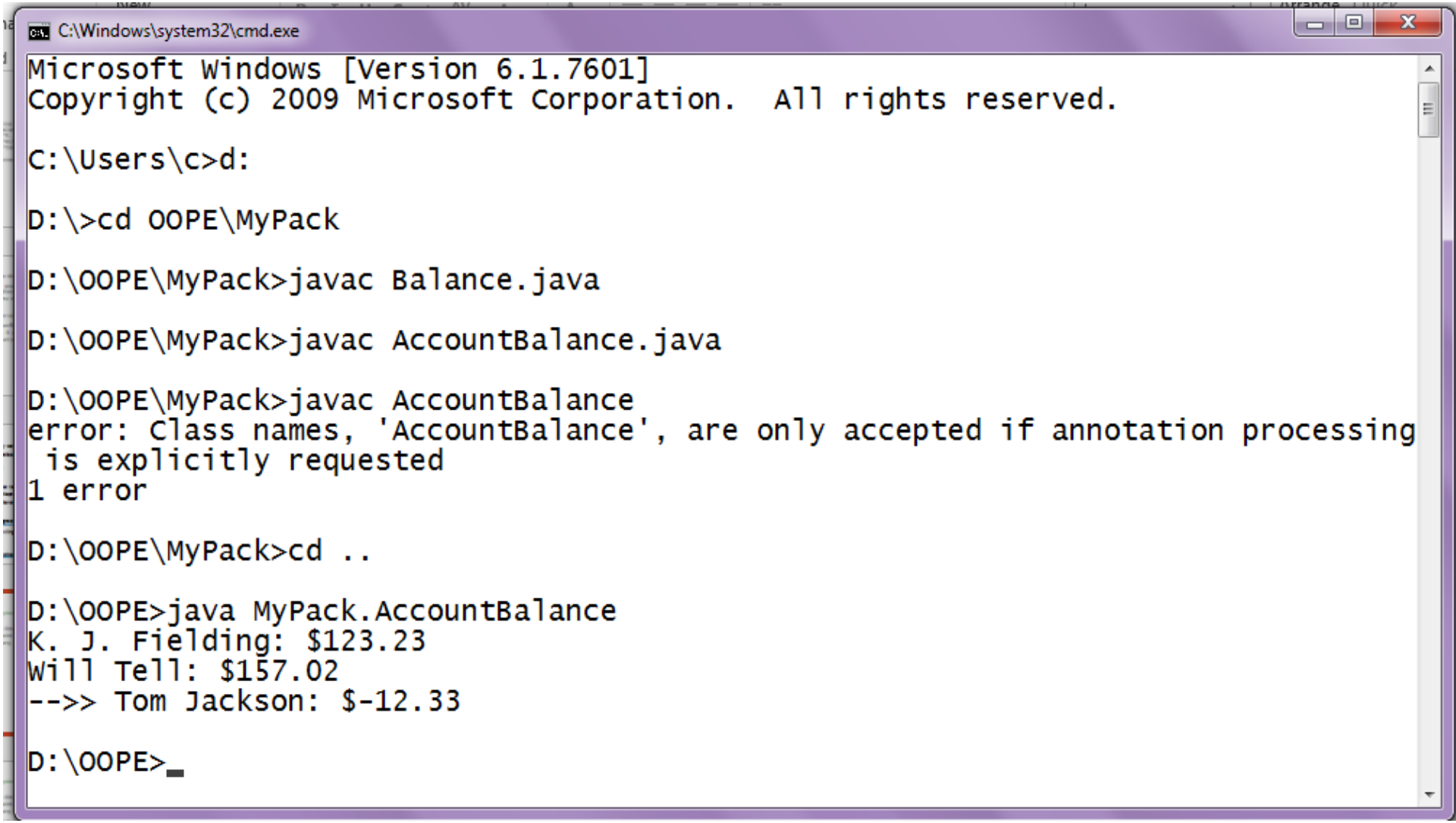
needs to be stored in following directory

oop\amp\btech\packages

Package and CLASSPATH

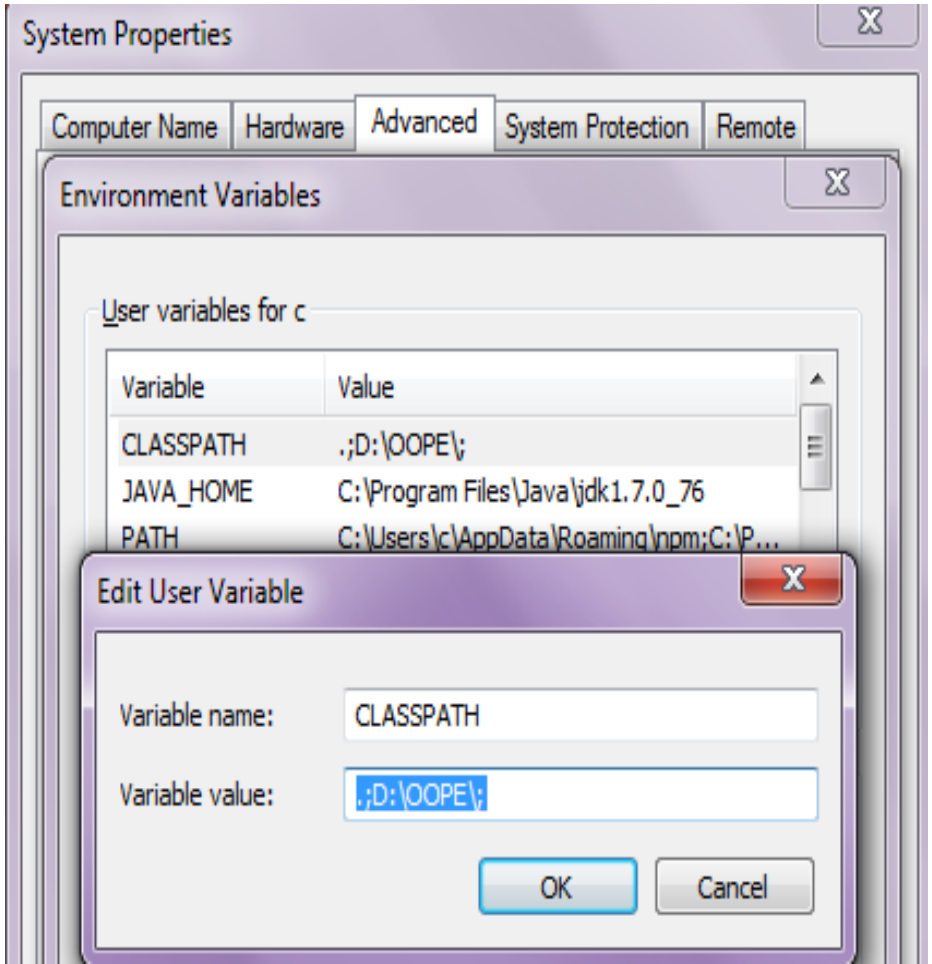
- How does the Java run-time system know where to look for packages that you create?
- First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found.

Package and CLASSPATH



Package and CLASSPATH

- How does the Java run-time system know where to look for packages that you create?
- Second, you can specify a directory path or paths by setting the CLASSPATH environmental variable.




```
D:\OOPE>cd ..

D:\>java MyPack.AccountBalance
K. J. Fielding: $123.23
Will Tell: $157.02
-->> Tom Jackson: $-12.33

D:\>
```

Package and CLASSPATH

- How does the Java run-time system know where to look for packages that you create?
- Third, you can use the –classpath option with java and javac to specify the path to your classes.

```
D:\>java MyPack.AccountBalance
Error: Could not find or load main class MyPack.AccountBalance

D:\>java -classpath D:\OOPE\ MyPack.AccountBalance
K. J. Fielding: $123.23
Will Tell: $157.02
-->> Tom Jackson: $-12.33

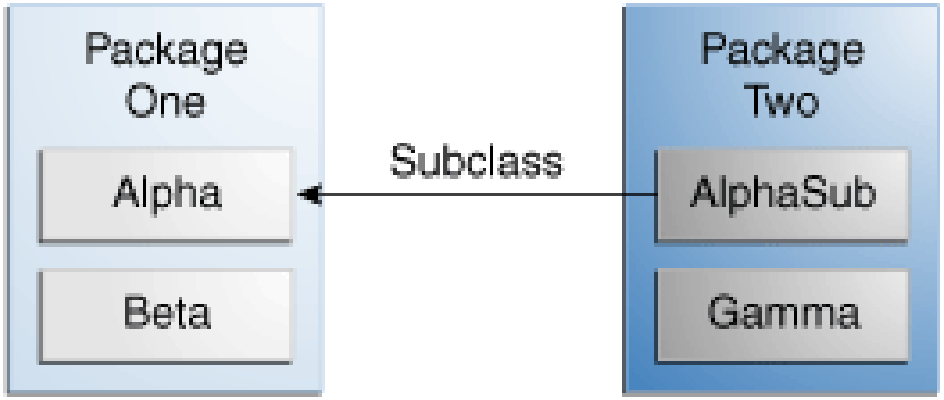
D:\>
```

Access Protection

	Private	No modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Access Protection

- While Java's access control mechanism may seem complicated, we can simplify it as follows.
- Anything declared public can be accessed from anywhere. Anything declared private cannot be seen outside of its class.
- When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. **This is the default access**.
- If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element protected**.**



Visibility

Modifier	Alpha	Beta	Alphasub	Gamma
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

Reference:
https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

Access Protection

This is file **Protection.java**:

```
package p1;

public class Protection {
  int n = 1;
  private int n_pri = 2;
  protected int n_pro = 3;
  public int n_pub = 4;

  public Protection() {
    System.out.println("base constructor");
    System.out.println("n = " + n);
    System.out.println("n_pri = " + n_pri);
    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
```

Access Protection

This is file **Derived.java**:

```
package p1;

class Derived extends Protection {
  Derived() {
    System.out.println("derived constructor");
    System.out.println("n = " + n);

//  class only
//  System.out.println("n_pri = "4 + n_pri);

    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
```

Access Protection

This is file **SamePackage.java**:

```
package p1;

class SamePackage {
  SamePackage() {

    Protection p = new Protection();
    System.out.println("same package constructor");
    System.out.println("n = " + p.n);

//  class only
//  System.out.println("n_pri = " + p.n_pri);
    System.out.println("n_pro = " + p.n_pro);
    System.out.println("n_pub = " + p.n_pub);
  }
}
```

Access Protection

This is file **Protection2.java**:

```
package p2;

class Protection2 extends p1.Protection {
  Protection2() {
    System.out.println("derived other package constructor");

//  class or package only
//  System.out.println("n = " + n);

//  class only
//  System.out.println("n_pri = " + n_pri);

    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
```

Access Protection

This is file **OtherPackage.java**:

```
package p2;

class OtherPackage {
  OtherPackage() {
    p1.Protection p = new p1.Protection();
    System.out.println("other package constructor");

//  class or package only
//  System.out.println("n = " + p.n);

//  class only
//  System.out.println("n_pri = " + p.n_pri);

//  class, subclass or package only
//  System.out.println("n_pro = " + p.n_pro);

    System.out.println("n_pub = " + p.n_pub);
  }
}
```

Importing Packages

- In java all of the standard classes are stored in some named package.
- If we need use some classes from different package we should import that package using import statement
- Once imported, a class can be referred to directly, using only its name.
- To specify either an explicit classname or a star (*), which indicates that the Java compiler should import the entire package.

```
import pkg1[.pkg2].(classname|*);

import java.util.Date;

import java.io.*;
```

CAUTION *The star form may increase compilation time—especially if you import several large packages. For this reason it is a good idea to explicitly name the classes that you want to use rather than importing whole packages. However, the star form has absolutely no effect on the run-time performance or size of your classes.*

Importing Packages

```
package MyPack;

/* Now, the Balance class, its constructor, and its
   show() method are public.  This means that they can
   be used by non-subclass code outside their package.
*/
public class Balance {
  String name;
  double bal;

  public Balance(String n, double b) {
    name = n;
    bal = b;
  }

  public void show() {
    if(bal<0)
      System.out.print("--> ");
    System.out.println(name + ": $" + bal);
  }
}
```

Importing Packages

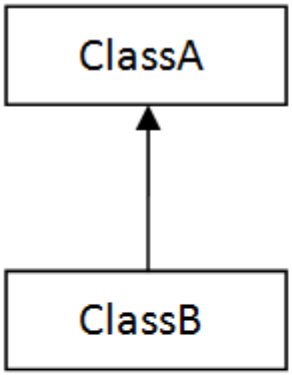
```
import MyPack.*;

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

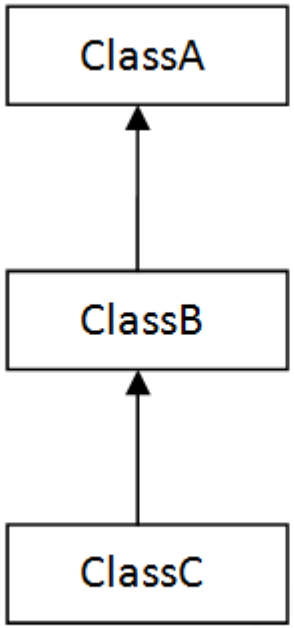
    /* Because Balance is public, you may use Balance
       class and call its constructor. */
    Balance test = new Balance("J. J. Jaspers", 99.88);

    test.show(); // you may also call show()
  }
}
```

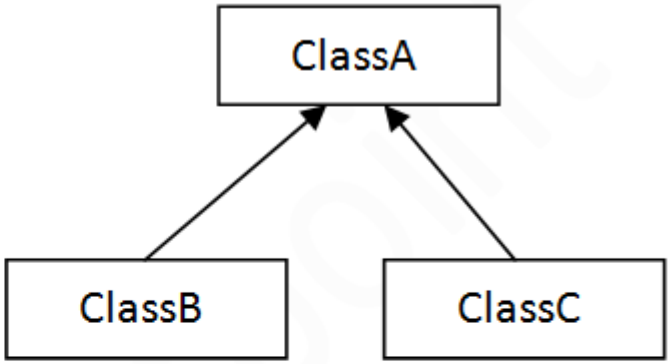
TYPES OF INHERITANCE



1) Single



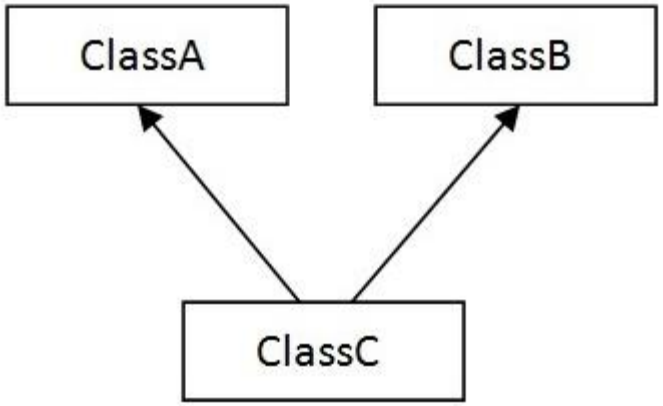
2) Multilevel



3) Hierarchical

TYPES OF INHERITANCE

Multiple inheritance is not supported in java through class.



4) Multiple

Interfaces

- Using interface, we can fully abstract a class' interface from its implementation.
- Using **interface**, we can specify what a class must do, but not how it does it.
- Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.
- Once it is defined, any number of classes can implement an **interface.**
- One class can implement any number of interfaces.
- To implement an interface, a class must create the complete set of methods defined by the interface.
- Each class is free to determine the details of its own implementation

Why interface?

- Interfaces are designed to support dynamic method resolution at run time.
- Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible. This requirement by itself makes for a static and non extensible classing environment.

Defining interface

```
access interface name {
        return-type method-name1(parameter-list);
        return-type method-name2(parameter-list);
        type final-varname1 = value;
        type final-varname2 = value;
        // ...
        return-type method-nameN(parameter-list);
        type final-varnameN = value;
}
```

- The methods which are declared have no bodies(abstract). They end with a semicolon after the parameter list.
- Each class that includes an interface must implement all of the methods.
- Variables can be declared inside of interface declarations. They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class.
- Variables must also be initialized with a constant value. All methods and variables are implicitly **public** if the interface, itself, is declared as **public**

interface Example

```
interface Callback {
      void callback(int param);
}
```

Once an **interface** has been defined, one or more classes can implement that interface

Implementing interface

- To implement an interface, include the implements clause in a class definition, and then create the methods defined by the interface

```
class classname [extends superclass] [implements interface [,interface...]] {
    // class-body
}
```

- If a class implements more than one interface, the interfaces are separated with a comma
- If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface. Methods implementing interface has to be declared as public
- The type signature of the implementing method must match exactly the type signature specified in the interface definition

```
interface left{
        public void m1();
}
interface right{
        public void m1();
}
class Test implements left,right
{
public void m1(){
        System.out.println("Hi");
}
public static void main(String args[]){
        Test t = new Test();
        t.m1();
}
}
```

Output ?

```
interface InterfaceX
{      public int geek();
}
interface InterfaceY
{      public String geek();
}
public class Testing implements InterfaceX, InterfaceY
{
public String geek()
    {          return "hello";
    }
public static void main(String args[]){
          Testing t = new Testing();
          System.out.println(t.geek());
}
}
```

Output ?

Implementing interface

```
class Client implements Callback {
  // Implement Callback's interface
  public void callback(int p) {

    System.out.println("callback called with " + p);
  }
}
```

- It is both permissible and common for classes that implement interfaces to define additional members of their own

```
class Client implements Callback {
  // Implement Callback's interface
  public void callback(int p) {
    System.out.println("callback called with " + p);
  }

  void nonIfaceMeth() {
    System.out.println("Classes that implement interfaces " +
                       "may also define other members, too.");
  }
}
```

Accessing implementation through interface references

- It is possible to declare variables as object references that use an interface rather than a class type.
- Any instance of any class that implements the declared interface can be stored in such a variable. When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interfaces.
- ```
  class TestIface {
    public static void main(String args[]) {
      Callback c = new Client();
      c.callback(42);
    }
  }
  ```
- c.callback(42) will call implementation in Client class
  ```

Accessing implementation through interface references

```
// Another implementation of Callback.
class AnotherClient implements Callback {
  // Implement Callback's interface
  public void callback(int p) {
    System.out.println("Another version of callback");
    System.out.println("p squared is " + (p*p));
  }
}
class TestIface2 {
  public static void main(String args[]) {
    Callback c = new Client();
    AnotherClient ob = new AnotherClient();

    c.callback(42);

    c = ob; // c now refers to AnotherClient object
    c.callback(42);
  }
}
```

```
callback called with 42
Another version of callback
p squared is 1764
```

- The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code which calls methods on them.
- The calling code can dispatch through an interface without having to know anything about the "callee." This process is similar to using a superclass reference
- access a subclass object, as described in Dynamic Method Dispatch

Partial Implementations

- If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared as `abstract`.

```
abstract class Incomplete implements Callback {
   int a, b;
   void show() {
     System.out.println(a + " " + b);
   }
   // ...
}
```

- Here, the class Incomplete does not implement callback**()** and must be declared as abstract.
- Any class that inherits Incomplete must implement callback() or be declared abstract itself.

Nested Interfaces

- An interface can be declared a member of a class or another interface.
- A nested interface can be declared as public, private, or protected. This differs from a top-level interface, which must either be declared as public or use the default access level
- When a nested interface is used outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is a member.

Nested Interfaces

```
// A nested interface example.

// This class contains a member interface.
class A {
  // this is a nested interface
  public interface NestedIF {
    boolean isNotNegative(int x);
  }
}

// B implements the nested interface.
class B implements A.NestedIF {
  public boolean isNotNegative(int x) {
    return x < 0 ? false : true;
  }
}

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

    // use a nested interface reference
    A.NestedIF nif = new B();

    if(nif.isNotNegative(10))
      System.out.println("10 is not negative");
    if(nif.isNotNegative(-12))
      System.out.println("this won't be displayed");
  }
}
```

Applying Interfaces

- The Stack class that implemented a simple fixed-size stack
- It can be growable.
- It was implemented with array. It can be implemented with linked list, vector, trees etc.
- No matter how the stack is implemented, the interface to the stack should remains the same like push and pop functions
- The interface to a stack is separate from its implementation, it is easy to define a stack interface, leaving it to each implementation to define the specifics.

```
// Define an integer stack interface.
interface IntStack {
  void push(int item); // store an item
  int pop(); // retrieve an item
}
```

Applying Interfaces

```
// An implementation of IntStack that uses fixed storage.
class FixedStack implements IntStack {
  private int stck[];
  private int tos;

  // allocate and initialize stack
  FixedStack(int size) {
    stck = new int[size];
    tos = -1;
  }

  // Push an item onto the stack
  public void push(int item) {
    if(tos==stck.length-1) // use length member
      System.out.println("Stack is full.");
    else
      stck[++tos] = item;
  }

  // Pop an item from the stack
  public int pop() {
    if(tos < 0) {
      System.out.println("Stack underflow.");
      return 0;
    }
    else
      return stck[tos--];
  }
}
```

Applying Interfaces

```
class IFTest {
  public static void main(String args[]) {
    FixedStack mystack1 = new FixedStack(5);
    FixedStack mystack2 = new FixedStack(8);

    // push some numbers onto the stack
    for(int i=0; i<5; i++) mystack1.push(i);
    for(int i=0; i<8; i++) mystack2.push(i);

    // pop those numbers off the stack
    System.out.println("Stack in mystack1:");
    for(int i=0; i<5; i++)
      System.out.println(mystack1.pop());

    System.out.println("Stack in mystack2:");
    for(int i=0; i<8; i++)
      System.out.println(mystack2.pop());
  }
}
```

```
// Implement a "growable" stack.
class DynStack implements IntStack {
  private int stck[];
  private int tos;

  // allocate and initialize stack
  DynStack(int size) {
    stck = new int[size];
    tos = -1;
  }

  // Push an item onto the stack
  public void push(int item) {
    // if stack is full, allocate a larger stack
    if(tos==stck.length-1) {
      int temp[] = new int[stck.length * 2]; // double size
      for(int i=0; i<stck.length; i++) temp[i] = stck[i];
      stck = temp;
      stck[++tos] = item;
    }
    else
      stck[++tos] = item;
  }

  // Pop an item from the stack
  public int pop() {
    if(tos < 0) {
      System.out.println("Stack underflow.");
      return 0;
    }
    else
      return stck[tos--];
  }
}
```

```
/* Create an interface variable and
   access stacks through it.
*/
class IFTest3 {
  public static void main(String args[]) {
    IntStack mystack; // create an interface reference variable
    DynStack ds = new DynStack(5);
    FixedStack fs = new FixedStack(8);

    mystack = ds; // load dynamic stack
    // push some numbers onto the stack
    for(int i=0; i<12; i++) mystack.push(i);

    mystack = fs; // load fixed stack
    for(int i=0; i<8; i++) mystack.push(i);


    mystack = ds;
    System.out.println("Values in dynamic stack:");
    for(int i=0; i<12; i++)
       System.out.println(mystack.pop());

    mystack = fs;
    System.out.println("Values in fixed stack:");
    for(int i=0; i<8; i++)
       System.out.println(mystack.pop());
  }
}
```

Variables in Interfaces

- You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables which are initialized to the desired values.
- If an interface contains no methods, then any class that includes such an interface doesn't actually implement anything. It is as if that class were importing the constant variables into the class name space as `final` variables.

Variables in Interfaces

```
import java.util.Random;

interface SharedConstants {
  int NO = 0;
  int YES = 1;
  int MAYBE = 2;
  int LATER = 3;
  int SOON = 4;
  int NEVER = 5;
}

class Question implements SharedConstants {
  Random rand = new Random();
  int ask() {
    int prob = (int) (100 * rand.nextDouble());
    if (prob < 30)
      return NO;            // 30%
    else if (prob < 60)
      return YES;           // 30%
    else if (prob < 75)
      return LATER;         // 15%
    else if (prob < 98)
      return SOON;          // 13%

    else
      return NEVER;         // 2%
  }
}
```

Variables in Interfaces

```
class AskMe implements SharedConstants {
  static void answer(int result) {
    switch(result) {
      case NO:
        System.out.println("No");
        break;
      case YES:
        System.out.println("Yes");
        break;
      case MAYBE:
        System.out.println("Maybe");
        break;
      case LATER:
        System.out.println("Later");
        break;
      case SOON:
        System.out.println("Soon");
        break;
      case NEVER:
        System.out.println("Never");
        break;
    }
  }
```

Variables in Interfaces

```
    public static void main(String args[]) {
      Question q = new Question();
      answer(q.ask());
      answer(q.ask());
      answer(q.ask());
      answer(q.ask());
    }
}
```

Output:

```
Later
Soon
No
Yes
```

INTERFACES CAN BE EXTENDED

One interface can inherit another by use of the keyword `extends`. The syntax is the same as for inheriting classes.

When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain

```
// One interface an extend another.
interface A {
  void meth1();
  void meth2();
}
// B now includes meth1() and meth2()-
// it adds meth3().
interface B extends A {
  void meth3();
}

// This class must implement all of A and B
class MyClass implements B {
  public void meth1() {
    System.out.println("Implement
meth1().");
  }
  public void meth2() {
    System.out.println("Implement
meth2().");
  }
public void meth3() {
    System.out.println("Implement
meth3().");
  }
}

class IFExtend {
  public static void main(String arg[]) {
    MyClass ob = new MyClass();

    ob.meth1();
    ob.meth2();
    ob.meth3();
  }
}
```