Intro. To Java Programming Language (II)

- References:
  - Text book : Appendixes & Chapter 30
  - Sun Java Tutorial : http://download.oracle.com/javase/tutorial/index.html

1. Introduction to Classes and Objects

- Quick review :
  - A class is a blueprint or prototype from which objects are created. A class definition is a general description of what the object is and what it can do
  - An object is a program construct that contains data (fields) and can perform certain actions (methods).
  - All objects in the same class have the same kinds of data and the same methods

  - Example

    class name : Rectangle
    data : length, width
    methods : setLength(), setWidth(), computeArea(), etc

    Several objects (instances)

name:	smallBox	largeBox	aSquare
data:			
length:	10	500	50
width:	20	1000	50
Methods:			
	setLength()	setLength()	setLength()
	setWidth()	setWidth ()	setWidth()
	computerArea()	computeArea()	computeArea()

- Sample Java class Name

```
public class Name
{
        // start with data fields
        private String first; // first name
        private String last;  // last name

        // a collection of methods
        public Name()
        {
        } // end default constructor

        public Name(String firstName, String lastName)
        {
                first = firstName;
                last = lastName;
        } // end constructor

        public void setName (String firstName, String lastName)
        {
                setFirst (firstName);
                setLast (lastName);
        } // end setName


        public String getName()
        {
                return toString();
        } // end getName

        public void setLast(String lastName)
        {
                last = lastName;
        } // end setLast

        public String getLast()
        {
                return last;
        } // end getLast


        public String toString()
        {
                return first + " " + last;
        } // end toString

        // more methods: setFirst(), getFirst(), etc
} // end Name
```

- We use the above class Name to explain next few pages.

- Creating objects and accessing/invoking methods :

  A program component that uses a class is called a ***client*** of the class.

  Declare reference variable of type Name:

  ```
  Name james;
  ```

  Using class default constructor methods to create an instant/object:

  ```
  james = new Name();          // OR
  Name james = new Name(); // same as above 2 lines
  ```

  Another way to construct (using another constructor):

  ```
  Name james = new Name("James", "Wong");
  ```

  From class Name: public void setLast(String lastName)

  lastName is ***formal parameter***
  This method does not return value, so it is a ***void method***

  Call/invoke setLast() method by giving an ***argument*** type String "Wong":

  ```
  james.setLast("Wong");
  ```

  From class Name: public String getLast();
  This method returns a single String value.
  In general, a method can only return a single value using ***return*** statement

  Call getLast() method and capture return value to a variable myStr.

  ```
  String myStr = james.getLast();
  ```

Reference and aliases

Recall : Arrays and Classes are reference types

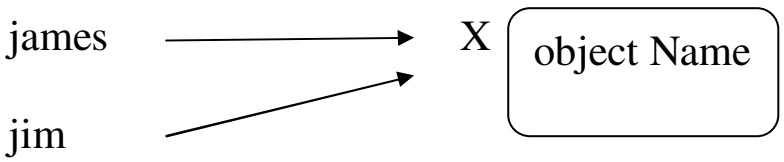
james is a reference variable (of type Name) which contain an address X
note : it is null or unknown before "new" statement

X is an address which is a memory location of an object Name

Consider a new variable:

Name jim = james; // same address as james

So, jim is also a reference variable which contain address X.



james and jim are ***aliases*** – reference the same instance of Name

- Defining a class: A class should describe a single entity, and all class operations should logically fit together. For example : student & faculty are 2 entities

Access/visibility modifier

- Access/visibility modifier: "public", "private" and "protected, specifies where a class, data field or method can be used.

  Default visibility: no visibility is given. Classes, methods and data fields are accessible by any class in the same package

- public class: class is available for use in any other classes (and in any packages). Note: "private or protected" are not available for classes

- public method/field: method & field are available for use in any other classes in the same package (and any package if it is in public class)

- private method/field: only available within its own class

- protected method/field: will cover this in "derived class" section later.

Example : Modify previous class Name to NameNew

```
public class NameNew
{
        private String first; // first name
        private String last;  // last name
        public  int age;
        …
        public void setLast(String lastName)
        {
                last = lastName;
        } // end setLast

        public String getLast()
        {
                return last;
        } // end getLast

        private boolean checkName()
        {       …
        }
    …
}
```

- Class *NameNew* is available in any classes in any packages
- Fields *first* and *last* are available within class Name only, but *age* is available in any class and any package.
- Methods *setLast()* and *getLast()* are available in any classes and any package. checkName() only available within the class.

Defining fields & methods

- Usually data (fields) are defined at the beginning of class. Methods are defined after data

- User-modifier : static, final

  - A static data field (or method) has only one instance variable (method) exists for all instances. It is also called class variable (or method)

  - Note: without "static", variables (or methods) are instance variables (methods), i.e. one copy per instance

  - Example:

    If we add a static variable and a static method to class Name above:

    ```
    public static int counter=1;
    static public int max(int first, int second) { // statements };
    ```

    Then, we may access this static field/method as:

    ```
    james.counter     // same counter, get value 1, should avoid this way
    jim.counter++     // same counter, inc by 1, should avoid this way
    Name.counter      // same counter, may access by class name, get 2

    int val =Name.max(10,20);  // call max method by class Name
    ```

- To define a static constant data, you need to add "final" constant :

  ```
  public static final float PI = 3.14;
  ```

- Some rules

  - Instance methods can access instance variables and instance methods directly.
  - Instance methods can access class variables and class methods directly.

- Class methods can access class variables and class methods directly.

- Class methods ***cannot*** access instance variables or instance methods directly—they must use an object reference.

Common methods

- Special method: *Constructors* - allocate memory for object and initialize data fields. It does not have return type (not even void) and has the same name as class name. May have several constructors.

  Default constructor is the constructor without parameter. If you do not define any constructors for a class, a default constructor will be provided automatically. If you define a constructor, then a default constructor will not be provided automatically.

- General method definition:

  access-modifier user-modifier return-type method-name (parameter-list) { // statements }

- Accessor (query) methods return value of a data field.
  Mutator methods change the value of a data field.

  Example:

```
public String getLast();    // usually, start with "get"
public void setLast (String lastName); // usually, start with "set"
```

- toString() method: return a string related to object.

  For example : toString() method in Name returns a string *first* + " " + *last*

  Note: a statement *System.out.println (someObjectName);* invoke toString() method automatically

Quick Review on modifiers

package one

```
public class A {
        // data/fields
        int a;
        public int b;
        private int c;
        static public int d;
        static private int e;

        // methods
        public void f1() {….}
        private void f2() {…}
        void f3() {…}
        static public f4() {….}

        void f5() {    //Q1 : list all fields/methods in A that can be accessed here }

        static void f6() {// Q2: list all fields/methods in A that can be accessed here }
}
// in same package as A
class B {
        void g() {
                A objA = new A();
                System.out.println(objA.a);
                objA.f1();

                // Q3: : list all fields/methods in objA that can be accessed here
        }
}
```

package two

```
import one.*;
public class C {
        void g() {
                A objA = new A();
                System.out.println(objA.b);
                objA.f1();

                // Q4: list all fields/methods in objA that can be accessed here
                // Q5: can you access (or create) object of class B here?
        }
}
```

Some General rules

- Use *this* to avoid confusion. *this* refers to current object

  Example: Name with field *first* and method with parameter *first*

```
// parameter first has the same name as data field first
public void setFirst(String first)
{
        this.first = first;  // use this.first to refer to data field
} // end setFirst
```

  May use ***this()*** to call a constructor from another constructor in the same class. It must be the first statement. Example:

```
this();  //refer to default constructor in Name
this("hello","world"); // refer to 2nd constructor in Name
```

  Class methods cannot use *this* keyword as there is no instance for this to refer to.

- Arguments/parameters (in call) must match in number, order and type to formal parameters (in definition)

- Pass by value : When a formal parameter is primitive type
  formal parameter in method initialized by value
  argument can be constant or variable

- Pass by reference : When a formal parameter is a class/array type, it is initialized with memory address of object passed to it

  Example: Assume the sampleMethod() is invoked as follows :

```
        sampleMethod(james, 10, arrayU);   // arguments

void sampleMethod(Name jimmy, int x, int[] arrayV) {
```

```
        // jimmy refers to same object james, x has a value 10
        // arrayV refers to same array arrayU
        jimmy.setLast("zzzz"); // this change data in object james

        // this create new Name object,
        // but will not change james object
        jimmy = new Name("abc", "def");
    }
```

Example: if an argument is a constant string, a string object is constructed and it is a passed to the corresponding formal parameter

- A method may call another method within a class without preceding the method name with object name. Note : you may use "this.methodName"

- To return a reference to the current object, use *return this;*

- Overloading methods:

  Multiple methods within the same class can have the same name

  Java distinguishes them by noting the parameters
  - Different numbers of parameters
  - Different types of parameters

  This is called the signature of the method

2. More on Classes and Objects

- A major advantage of OOP is the ability to use existing classes to define new classes

- There are two major ways to accomplish this task: composition and inheritance

- Composition ("has a" relationship)

  - When a class *has a* data field that is an instance of another class.

    For example: We can define a class Student where one of its data field is object Name:

```
public class Student
{
      private  Name fullName;
      private  String id; // identification number

      public Student ()
      {
                  fullName = new Name();
                  id = "";
      } // end default constructor


      public Student (Name studentName, String studentId)
      {
                  fullName = studentName;
                  id = studentId;
      } // end constructor

     //… more methods
}
```

- Generic Types: for a class with fields that can be of any class type

  Syntax to define a generic type in a class: public class MyClass <T>

  Example:

```
public class OrderedPair <T>
{
    private T first, second;

    public OrderedPair ()
    {
    } // end default constructor


    public void setPair (T firstItem, T secondItem)
    {
        first = firstItem;
        second = secondItem;
    } // end setPair

    public void changeOrder ()
    {
        T temp = first;
        first = second;
        second = temp;
    } // end changeOrder


    public String toString ()
    {
        return "(" + first + ", " + second + ")";
    } // end toString
} // end OrderedPair
```

To use the class OrderedPair:

```
// T is replaced by type String
OrderedPair < String > fruit = new OrderedPair < String > ();
fruit.setPair ("apples", "oranges");
System.out.println (fruit);
fruit.changeOrder ();
System.out.println (fruit);

// T is replaced by type Name
Name tweedleDee = new Name ("Tweedle", "Dee");
Name tweedleDum = new Name ("Tweedle", "Dum");
OrderedPair < Name > couple = new OrderedPair < Name > ();
couple.setPair (tweedleDee, tweedleDum);
System.out.println (couple);
couple.changeOrder ();
System.out.println (couple)
```

Note: primitive types are not allowed!! Should use wrapper classes.

```
OrderedPair < int >  myObj = new OrderedPair < int> ();  // Error!!

OrderedPair < Integer> myObj = new OrderedPair <Integer > ();  // OK
```

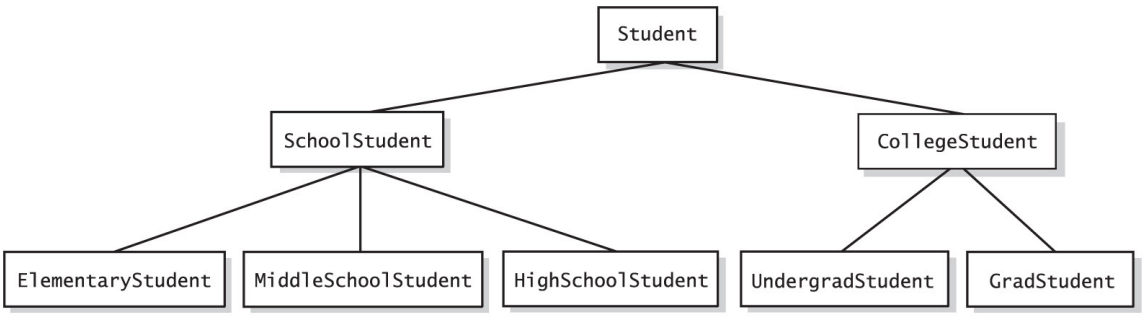
- Inheritance ("is a" relationship)
  - A general class (or base class or super class or parent class) is first defined. Then a derived class (subclass or child class) is defined:

    Adding to details of the base class
    Revising details of the base class

  - Advantages

    Saves work
    Common properties and behaviors are define only once for all classes involved

  - "is a" : An instant of a derived class is also an instance of the base class
  - A ***final*** class such as Math and String class cannot be a parent class.
  - Example: A hierarchy of classes



Student class:

private data fields: fullName, Id
public methods: setName(), setId(), toString()

These are common to all descendent classes

CollegeStudent class:

// use "*extends* Students" to indicate a derived class of Students

```
public class CollegeStudent extends Student
{
   private int year; // year of graduation
   private String degree; // degree sought

   public CollegeStudent ()
   {
      super(); // call constructor in Student, must be first statement
      year = 0;
      degree = "";
   } // end default constructor

   public CollegeStudent (Name studentName, String studentId,
         int graduationYear, String degreeSought)
   {
      super(sudentName, studentId); // must be first
      year = graduationYear;
      degree = degreeSought;
   } // end constructor

   public void setStudent (Name studentName, String studentId,
         int graduationYear, String degreeSought)
   {
      setName (studentName); // NOT fullName = studentName;
      setId (studentId); // NOT id = studentId;
      year = graduationYear;
      degree = degreeSought;
   } // end setStudent

   < The methods setYear, getYear, setDegree, and getDegree go here. >
      . . .
    public String toString ()  // override base class method
   {
      return super.toString () + ", " + degree + ", " + year;  // note:  super usage
   } // end toString

} // end CollegeStudent
```

- Derived class inherits all properties and behaviors from base class (see "modifier" rules for accessing constraints)

- Constructors are not inherited since it does not make sense to have base class constructor names as constructor for derived class.

- Constructor in a derived class

  - must call the base class constructor
  - **super** is the name for the constructor of the base class
  - When **super** is used, it must be the first action in the derived constructor definition
  - Must not use the (base class) name of the constructor
  - If you do not invoke base class constructor, default constructor will be invoked automatically

- Derived class may access base class fields and methods.
  This is subject to "modifier" rules as follows:

  Data field or method that is "private" in base class cannot be accessed by derived class. However, it may be accessed indirectly.

  Data field or method that is "***protected***" or "public" in base class can be accessed by derived class.

  Example:

  CollegeStudent class cannot access to FullName and Id (from Student)

  CollegeStudent class can access to setName(), setID(), toString() etc

- General access control (modifiers) table

Modifier: Data/methods	same package		different package	
	same class	other class	sub class	other class
public	OK	OK	OK	OK
protected	OK	OK	OK	X
default	OK	OK	X	X
private	OK	X	X	X

Example:

```
Package 1:

public class A {
    public int x;
    protected int y;
    int z;
    private int u;
    protected void m() {}
}
```

```
Package 1:

public class B {
        A a = new A();
        // can access/invoke
        a.x; a.y; a.z; a.m();
        // cannot access a.u
}
```

```
Package 1:

public class C
    extends A {
      // can access
      x; y; z; m()
      // cannot access
      // u
}
```

```
Package 2:

public class D
   extends A {
      // can access
      x; y; m();
      // cannot access
      // z; u;
}
```

```
Package 2:

public class E {
   A a = new A();
   // can access
   a.x
   // cannot access
   // a.y; a.z; a.u; a.m();
}
```

- Overriding a method

  When a derived class defines an instant method with the same signature and same return type as in base class, i.e. same name, same return type, same number & types of parameters

  Objects of the derived class that invoke the method will use the definition from the derived class

  May use super in the derived class to call an overridden method of the base class. Example: from CollegeStudent class

```
public String toString ()  // override base class method
{
        // call toString() from Student class using "super"
        return super.toString () + ", " + degree + ", " + year;
} // end toString
```

  Use modifier "final" to specify that a method cannot be overridden
  Note: "final" is also used in constants

  Example:

```
public final void mySpecialMethod()
{ …}
```

- You may change the modifier of overridden method in derived class:
  // more access in overriding method is OK

  public→ public; protected → protected or public
  default → default, public or protected

- Note: *super* refers to baseclass (only one level of base class).
  i.e. *super.super* is not valid

- Multiple inheritance
  Java does not allow this. A derived class can have only one base class.
  However, a derived class can have multiple interfaces

- Object Type Compatibility

  Recall: If a class B is derived from class A. Then class B object is also a class A object.

  Example:

  a CollegeStudent object is also a Student object
  Student amy = new CollegeStudent(); // this is OK
  CollegeStudent james = new Student(); // this is ILLEGAL

  Note: amy (reference variable of Student type) may only access data/methods that are defined in class Student only

- Special Java class : Object

  Every class is a descendant of the class Object
  Object is the class that is the beginning of every chain of derived classes
  It is the ancestor of every other class

  Methods include:

```
toString()  // return a String ; public method
equals();  // public method
getClass()  // return type of object, final public method

public boolean equals(Object other) // override equal for Name
{
        // check for special cases : short-circuit evaluation
        if ((other == null) || (getClass() != other.getClass())
                return false;
        else {
                Name otherName = (Name) other; // need to cast it
                return ( this.first.equals(otherName.first) &&
                        this.last.equals(otherName.last));
        }
}
```

Quick Review on Object Type Compatibility & Casting

When we cast a reference/object along the class hierarchy in a direction from the root class towards the children or subclasses, it is a ***downcast***. When we cast a reference/object along the class hierarchy in a direction from the sub classes towards the root, it is an ***upcast***. We need not use a cast operator in this case. Note: Compile errors vs runtime error

Example :

```
//X is a supper class of Y and Z which are sibblings.
class X{ }
class Y extends X{ }
class Z extends X{ }

public class RunTimeCastDemo{
        public static void main(String args[]){
                X x = new X();
                Y y = new Y();
                Z z = new Z();

                X xy = new Y();    // compiles ok
                X xz = new Z();    // compiles ok
                Y yz = new Z();    // incompatible type (sibblings), error

                Y y1 = new X();    // X is not a Y, error
                Z z1 = new X();    // X is not a Z, error

                X x1 =  y;         // compiles ok (y is subclass of X), upcast
                X x2 =  z;         // compiles ok (z is subclass of X), upcast

                Y y1 = (Y) x;      // compiles ok but produces runtime error
                Z z1 = (Z) x;      // compiles ok but produces runtime error
                Y y2 = (Y) x1;     // compiles and runs ok (x1 is type Y), downcast
                Z z2 = (Z) x2;     // compiles and runs ok (x2 is type Z), downcast
                Y y3 = (Y) z;      // inconvertible types (sibblings), error
                Z z3 = (Z) y;      // inconvertible types (sibblings), error

                Object o  = z;     // OK, upcast
                Y o1 = (Y) o;      // compiles ok but produces runtime error

        }
}
```

- **Polymorphism**

Polymorphism : One instant method name in an instruction can cause different actions according to the kinds of objects that invoke the method

This process is called *dynamic binding or late binding*, i.e. method invocation is not bound until run time.

Example: Each class has a display() instant method

Class Student → CollegeStudent → UndergraduateStudent

display() //1 display() //2 display()//3

displayAt() //4, only in Student, it calls display() in it

```
Student s;

UndergraduateStudent us = new UndergraduateStudent(…);
s=us;   // s and us refer to same UndergraduateStudent object
s.displayAt();     // call 4 → call 3, since it refers to us object
us.displayAt();   // same as above line

CollegeStudent cs = new CollegeStudent(…);
s=cs;   // s refers to CollegeStudent object
s.displayAt();   // call 4 → call 2, since it refers to us object

s = new UndergraduateStudent(…);  // OK, refer to US object
```

Note: the object, not the variable, determines which definition of a method should be used!!

Example: (continue from previous example..)

```
s=cs;
System.out.println(s); // do not need to specify .toString()
                       // should call method in CollegeStudent
```

- Hiding class methods/data: If a subclass defines a class method/data with the same signature as a class method in the superclass, the method/data in the subclass *hides* the one in the superclass. Example:

```
public class Animal {
   public int i=10;
   public static void testClassMethod() {
      System.out.println("The class method in Animal."); }

   public void testInstanceMethod() {
      System.out.println("The instance method in Animal.");}
}

public class Cat extends Animal {
   public int i=20;
   public static void testClassMethod() {
      System.out.println("The class method in Cat.");}

   public void testInstanceMethod() {
      System.out.println("The instance method in Cat.");}

   public static void main(String[] args) {
      Cat myCat = new Cat();
      Animal myAnimal = myCat;
      Animal.testClassMethod();
      myAnimal.testInstanceMethod();
      myAnimal.testClassMethod();
      myCat.testClassMethod();
      //myCat.i → 20
      //myAnimal.i → 10
   }
}
```

The output from this program is as follows:

```
The class method in Animal.
The instance method in Cat.
The class method in Animal.
The class method in Cat.
```

Abstract Classes and Methods

- Some base classes are not intended to have objects of that type (no instances). The objects will be created from derived classes.

  Use "abstract" to declare that base class.

  Note: Sometime, it is called abstract base class.

- Abstract method: The designer often specifies methods of the abstract methods without a body,

```
// signature of the method without body
public abstract void doSomething();
```

- A class with at least one abstract method must be declared an abstract class.

- Constructors cannot be abstract: derived class cannot override baseclass constructor.

  Constructors in abstract class should be protected since it is only used by subclasses.

- Example: Abstract class for Line, Circle, Rectangles classes

```
abstract class GraphicObject {
   int x, y;
        …
   void moveTo(int newX, int newY) {
       …
   }
   abstract void draw(); // abstract method
   abstract void resize(); // abstract method
}
```

- A non-abstract class (→ no abstract methods) is called concrete class

  Each concrete subclass of GraphicObject, such as Circle and Rectangle, must provide implementations for the draw and resize methods:

```
class Circle extends GraphicObject {
   void draw() { // implement draw() }
   void resize(){ // implement resize() }
}
```

- It is OK to have no abstract methods in an abstract class. It this case, you cannot use "new" to create an instance of this class

- If a subclass of an abstract class does not implement all abstract methods, the subclass must be defined abstract.

  Example:

```
public abstract class AbstractSequentialList<E>
      extends AbstractList<E>
{…}
```

- Abstract class can be used as data type (but cannot be instantiated)

  Example:

```
// it is OK to create an array with GraphicObject type
GraphicObject[] objs = new GraphicObject[5];

// now, may create derived class instances
objs[0] = new Circle();   // it is OK
objs[1] = new GraphicObject (); // ILLEGAL
```

- superclass of abstract class may be concreate class

- Interfaces
  - In many ways, it is similar to abstract class, contains only public static constants and abstract methods

    Example:

```
interface shape
{
    public static final String MyName="shape";   // constant
    abstract public void Draw();
}

interface shape2
{
    // Same as shape. It is OK to omit modifiers in interface
    // and “abstract” keyword
    String MyName="shape";
    void Draw();
}
```

  - A class that implements an interface must state so at start of definition with *implements* clause. A concrete class must implement every method declared in the interface. An abstract class does not need to implement all methods in the interface.

    Example:

```
class circle implements shape
{
    public void Draw() {
        System.out.println("Drawing Circle here");
    }
}
```

  - Multiple classes can implement the same interface

    Example: circle, square, etc classes

- A class can implement more than one interface

  Example:

```
// Assume baseclass A, interfaces B, C and D
// define new class E
// must define each methods in interfaces
// duplicated methods in interfaces only need to implement once

public class E extends A implements B, C, D
```

- Interface can be used as data type (and cannot be instantiated)

  Example: usage of objects v and w

```
// must pass an object of a class that implement the interface B
// Can only call methods/data in B
public void aMethod (B v);

C v = new E();  // it is OK, but can only call methods/data in C

E w = new E(); // can use all methods in E
```

- May use generic type within interface

  Example :

```
// interface with a generic type S
public interface Pair <S>
{       public void setPair (S firstItem, S secondItem);      }

// implement interface Pair
public class OrderedPair <S> implements Pair <S>
```

Example: implement interface java.lang.Comparable.

```
// define compareTo abstract method
public interface Comparable <T>
{       public int compareTo(T other);     }

// assume Measurable is an interface with
//  methods such as getArea() etc

public class Circle implements Comparable < Circle > , Measurable
{
   //…
   // couple of ways to implement compareTo
   private double radius;

    public int compareTo (Circle other)
    {
       int result;
       if (this.equals (other))
          result = 0;
       else if (radius < other.radius)
          result = -1;
       else
          result = 1;
       return result;
    } // compareTo

      // Version 2 : only if radius is integer
     // if radius is an integer, we can do simplify this:
     //         return radius - other.radius;
     // Note: value returned not necessarily -1 or +1
     //          but satisfies the definition
}
```

- May derive an interface from (by combining) several interfaces :

  Example: Assume interface A, B, define new interface C as follows

```
public interface C extends A,B {
        public void newMethod();  // may add new methods
}
```

- Abstract Classes versus Interfaces : Few important points
  - Unlike interfaces, abstract classes can contain fields that are not static and final, and they can contain implemented methods.
  - If an abstract class contains *only* abstract method declarations, it should be declared as an interface instead.
  - Interfaces do not have constructors
  - Use abstract classes when some general methods should be implemented (and may be override by derived classes)
  - An interface can extend other interfaces but not classes. A class can extend its superclass and implement multiple interface.

3. Exceptions

- Exceptions in java are any abnormal, unexpected events or special conditions that may occur at runtime (i.e. runtime errors).

  Example: file not found exception, unable to connect, divide by 0

- Exception overview :

  - Java Exceptions are basically Java objects:

    Object → Throwable → Exception → exception classes (see Example)
    → Error (internal system error from JVM)

    Example exception classes:

    IOException → EOFException, FileNotFoundException, etc

    RuntimeException → NullPointerException, ArithmeticException, etc.

    Throwable class provides methods such as

    Constructors to input an error msg (or no error msg)

    String getMessage(); // retrieve error msg

    void printStackTrace(); // print stack trace

    Error class (same level as Exception class): usually, not much you can do with these internal system (JVM) errors. Example : LinkageError

- Throwing an exception: An exception is created when an unusual event has occurred during execution time (use *throw* explicitly)

```
// create an exception object from class RuntimeException
throw new RuntimeException("hello, my execption!!");
```

Note: You may provide a string msg to constructor of exception object
Exception can be created implicitly such as 'int divided by 0'

- Handling exception: The code that detects and deals with the exception, i.e. using *try, catch, finally* blocks.

```
try
{
        // program statements
}
catch ( ExceptionClass1 identifier )
{
        // statements
}
catch ( ExceptionClass2 identifier )
{
        // statements
}
…
catch ( ExceptionClassN identifier )
{
        // statements
}
finally
{
        // statements
}
```

- try Block - statements that you think may produce an exception is placed within a try block for a suitable catch block to handle the error.

  If an exception is generated within the try block, the remaining statements in the try block are not executed.

  Must be followed by either at least one catch block or one finally block

- catch Block - Exceptions thrown during execution of the try block can be caught and handled in a catch block.

  Order of catch blocks: A superclass type (e.g. Exception) catch block must appear after subclass type (e.g. IOException) catch block.

  An exception will look for the matching catch block to handle the error. On exit from a catch block, normal execution continues and the finally block is executed.

  If no exception occurs (or no matching catch block) the execution proceeds with the finally block.

- finally Block - A finally block is always executed, regardless of the cause of exit from the try block, or whether any catch block was executed (except System.exit() is called).

  Generally finally block is used for freeing resources, cleaning up, closing connections etc.

  If the finally block executes a control transfer statement such as a return or a break statement, then this control statement determines how the execution will proceed regardless of any return or control statement present in the try or catch.

- An exception propagates to a try block with the proper catch block. i.e. if an exception is not caught in the current method (or there is no matching catch block), it is passed to its caller. The process is repeated until the exception is caught. If there is no applicable catch block, the program terminates abnormally.

- Example1 :

```
public class DivideException {

   public static void main(String[] args) {
       division(100,4);       // Line 1
       division(100,0);       // Line 2
       System.out.println("Exit main().");
   }

   public static void division(int totalSum, int totalNumber) {
       System.out.println("Computing Division.");
       int average  = totalSum/totalNumber;
       System.out.println("Average : "+ average);
   }
}
```

Output: // note the stack trace on line number

```
Computing Division.
Average : 25
Computing Division.
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DivideException.division(DivideException.java:11)
        at DivideException.main(DivideException.java:5)
```

note: default output - exception name, getMessage and printStackTrace

- Example2 :

```
public class ExceptionExample2 {

        public static void main(String[] args) {

                int x = 10, y = 2;
                int counter = 0;
                boolean flag = true;

                while (flag) {
                  start:   // label
                        try {
                                if ( y > 1 )
                                        break start;
                                if ( y < 0 )
                                        return;
                                x = x / y;
                                System.out.println ( "x : " + x + " y : "+y );
                        }
                        catch ( Exception e ) {
                                System.out.println ( e.getMessage() );
                        }
                        finally {
                                ++counter;
                                System.out.println ( "finally:  Counter : " + counter );
                        }
                        --y;  // after exception, continue here…
                }
        }

}
```

Output:

```
finally : Counter : 1
x : 10 y : 1
finally : Counter : 2
/ by zero
finally : Counter : 3
finally : Counter : 4
```

- Example3 :

```
public class ExceptionExample3 {

        public static void main(String[] args) {
                MyExceptionExample3 obj = new MyExceptionExample3();
                try
                {       obj.m2();   }
                catch (Exception e)
                {       System.out.println("main :" + e.getMessage()); }

                System.out.println("main : resume here");
                Obj.m2();  // do not catch, terminate abnormally!
                System.out.println("main : do not print this line ….");
        }
}

class MyExceptionExample3 {
        public void m1( )
        {
                System.out.println("m1 : print this line....");
                // int x=5/0;
                throw new RuntimeException("hello exception from m1()!");
                // do not reach here!
        }

        public void m2()
        {
                try
                {       m1();         }
                catch (Exception e)
                {       System.out.println("m2 :" + e.getMessage());  }

                System.out.println("m2 : resume here");
                m1();          // catch in main function
        }
}
```

Output:

```
m1 : print this line....
m2 :hello exception from m1()!
m2 : resume here
m1 : print this line....
main :hello exception from m1()!
main : resume here
m1 : print this line....
m2 :hello exception from m1()!
m2 : resume here
m1 : print this line....
Exception in thread "main" java.lang.RuntimeException: hello exception
from m1()
!
        at MyExceptionExample3.m1(ExceptionExample3.java:26)
        at MyExceptionExample3.m2(ExceptionExample3.java:38)
        at ExceptionExample3.main(ExceptionExample3.java:15)
```

- Checked and Unchecked Exceptions

  - Checked exceptions are subclass of Exception excluding class RuntimeException and its subclasses. Checked Exceptions forces programmers to deal with the exception that may be thrown.

    Example: FileNotFoundException, IOException

  - Unchecked exceptions are RuntimeException and any of its subclasses. The compiler doesn't force the programmers to either catch the exception or declare it in a throws clause.

    Example: ArithmeticException, ArrayIndexOutOfBoundException

  - Note: In Example3 above, m1() throws unchecked exceptions

- Example: "new FileInputStream" must be checked; otherwise, will get compilation error. See //* or //** comments below:

```
import java.io.*;
class ExceptionExample4 {

    public static FileInputStream m1(String fileName)
        throws FileNotFoundException    // *  handle by "throws"
    {
        FileInputStream fis = new FileInputStream(fileName);
        return fis;
    }


    public static void main(String args[])
    {
        FileInputStream fis1 = null;
        String fileName = "foo.bar";

        // ** handle by try, catch
        try {
                fis1 = new FileInputStream(fileName);
        }
        catch (FileNotFoundException ex)
        {
                System.out.println("Oops! Exception caught")
        }
    }

   // more code… only display partial code

}
```

- You may define your own exception classes and use it in your own code.

```
// only need constructor, store "reason" string into super()
// may also add other methods to customize it

class BadTemperature extends Exception{
  BadTemperature( String reason ){
          super ( reason );
    }
}


// … sample usage in a method of a class
public void setTemperature( int newTemp)
   throw BadTemperature   //  exceptions from this method
{
       if ( newTemp < 0)
             throw new BadTemperature ("Invalid value");
       //… more statetements
}
```

Intro. to Mutable and Immutable objects (Reference: Chap 30)

- Mutable objects

  A mutable object belongs to a class that has public mutator or set methods for its data fields. Note: data fields are usually private.

  The client uses set methods to change values of the object's data fields

  Example: may change last or first name in a Name object

- Immutable objects (i.e. read only class)

  An object whose content cannot be changed.

  Its class is called immutable class and must satisfy requirements:

  all data fields are private,
  no mutator methods and
  no accessor method that returns a reference to a data that is mutable
  (or data fields that are mutatble objects should be final)

  Example: String objects, primitive data type wrappers such as Integer, Float

- Companion classes

  If it is necessary to alter an immutable object, it can be accomplished by having a companion class of corresponding mutable objects

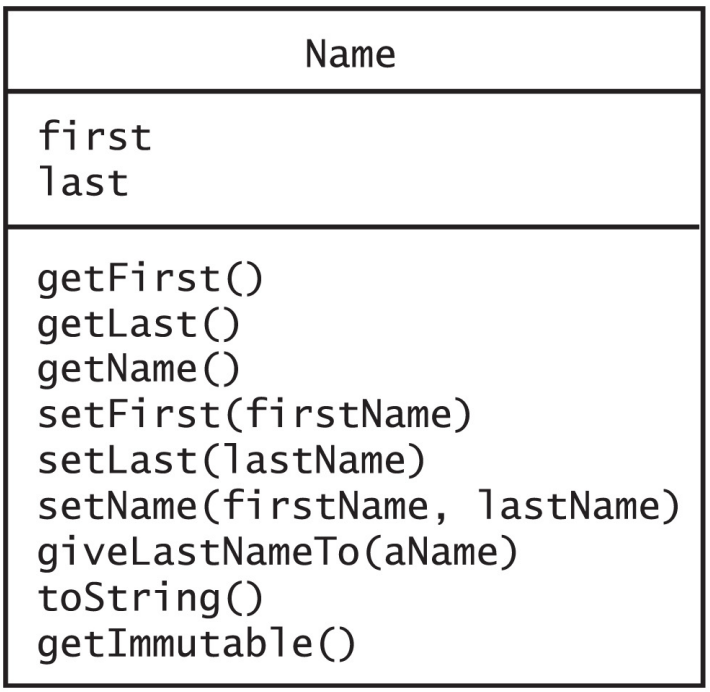
  It is helpful to have methods that convert an object from one type to another

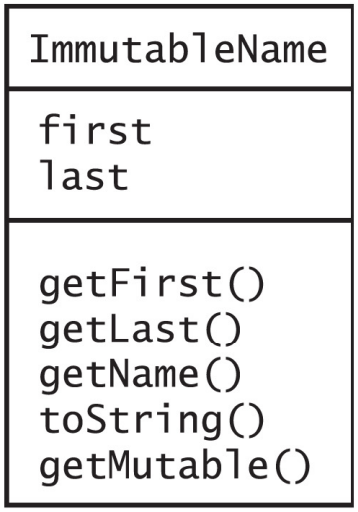
- Example: String class (immutable) vs Class StringBuilder (mutable)

- Example: Name class vs ImmutableName class

  Note: methods setFirst(), setLast(), setName(), giveLastNameTo()

  Also getImmutable() vs getMutable()






```
// return a Name object from current ImmutableName object
public Name getMutable()
{
  return new Name(first, last); // call Name's constructor
} // end getMutable

// return a ImmutableName object from Name object
public ImmutableName getImmutable()
{
  return new ImmutableName(first, last); //call ImmutableName constructor
} // end getMutable
```

- **Cloneable objects (Reference: Chap 30)**

  - A clone is a copy of an object
  - Cloning is a potentially dangerous action, because it can cause unintended side effects.

    For example, if the object being cloned contains a reference variable called *obRef,* then when the clone is made, *obRef* in the clone will refer to the same object as does *obRef* in the original.

    If the clone makes a change to the contents of the object referred to by *obRef,* then it will be changed for the original object, too

  - The Object class contains a protected method clone that returns a copy of an object

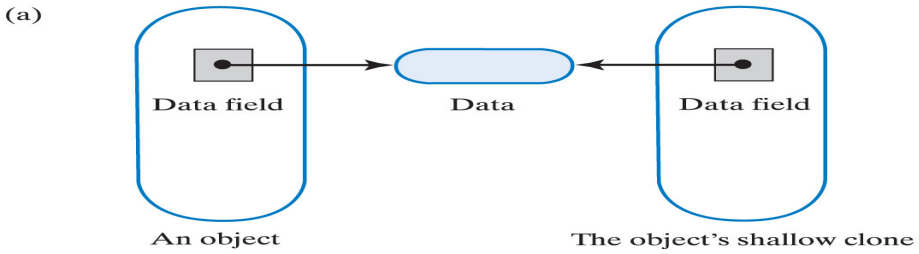
    protected object clone() throws CloneNotSupportException;

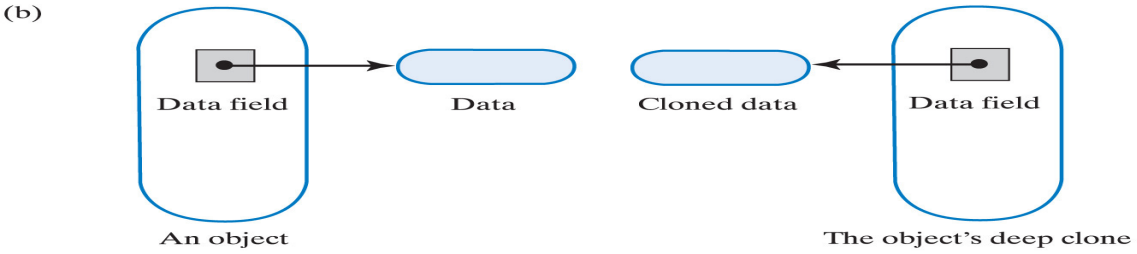
  - Most classes do not need to use this method. Read-only classes do not have this method. Since objects are immutable, it is safe for several variables to reference a same object.

    String st1 = str2; // reference to same object

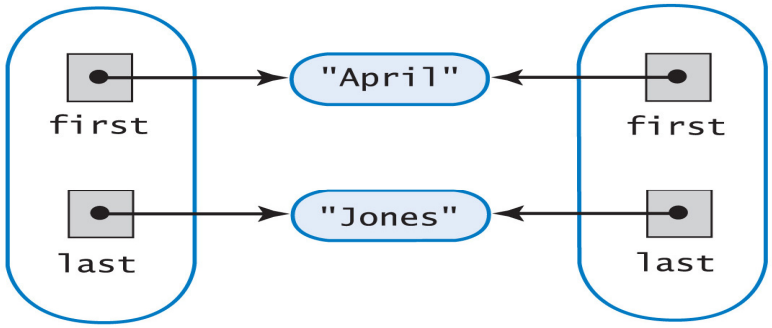
  - A class want to override clone() method must implement a marker interface Cloneable
  - We will not cover clone() method
  - Next page shows some examples in diagrams

Shallow clone vs. deep clone :

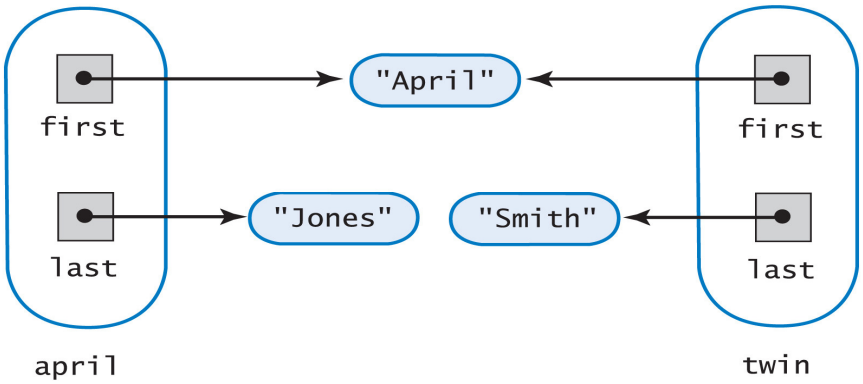




Name april = new Name("April", "Jones");
Name twin = (Name) april.clone();



twin.setLast("Smith");



4. Intro. to File Input and Output

- All I/O is handled by stream. A stream is an object that either:

  Delivers (or write) data to destination such as screen or file OR
  Takes (or read) data from a source such as keyboard or file.

- Advantages of files:

  File data are stored in disk after program ends
  May read the same file over and over again

- Files are stored as binary bits, i.e. a sequence of 0 and 1

- Text files – may think of it as a sequence of characters; can be created or read using a text editor. Recall ASCII and Unicode character sets

- Binary files – not a sequence of characters; unable to easily read a file using a text editor; may be able to read and written by specific programs

- May use Java programs to create both text files and binary files

- Java classes to handles files are: java.io.*

- Standard steps using file I/O: open, read/write. close

- Writing to text file using PrintWriter and FileWriter

  http://download.oracle.com/javase/6/docs/api/java/io/PrintWriter.html

```
// code segment
String fileName = "data.txt";  // file name
PrintWriter toFile = null;

try
{
        // open a file, can throw FileNotFoundException
        toFile = new PrintWriter(fileName);
}
catch (FileNotFoundException e)
{
        System.out.println("PrintWriter error opening the file " +
                fileName);
         System.exit(0);
} // end try/catch

// recall usage of scanner
System.out.println("Enter four lines of text:");
Scanner keyboard = new Scanner(System.in);

for (int count = 1; count <= 4; count++)
{
        String line = keyboard.nextLine();  // read line from keyboard
         toFile.println(count + " " + line);  // write line to file
} // end for

System.out.println("Four lines were written to " + fileName);

toFile.close();   // close file
```

Note:

- Before closing file, the data may be stored in temporary buffer, to force it to write to destination force, use toFile.flush() method

- In constructor, if the file exists then it will be truncated to zero size; otherwise, a new file will be created

- How to append to an existing text file? Use FileWriter object as an input to PrintWriter.

  new FileWriter(fileName, **true**); // true – do not truncate existing file

  Modify the previous example as:

```
String fileName = "data.txt";

FileWriter fw = null;    // new File writer object
PrintWriter toFile = null;

try
{
   fw = new FileWriter(fileName, true); // can throw IOException
   toFile = new PrintWriter(fw);
}
catch (FileNotFoundException e)
{
   // same as before
}
catch (IOException e)
{
   // catch IO exception from FileWriter
} // end try/catch

//… same as before
```

- Reading from text file using Scanner

  http://download.oracle.com/javase/6/docs/api/java/util/Scanner.html

```
// Use scanner to read lines from files

    String fileName = "data.txt";
    Scanner fromFile = null;   // declare a scanner

    System.out.println("The file " + fileName +
                       " contains the following lines:");
    try
    {
          // Scanner constructor does not take a string filename
          // need to provide a File object
          fromFile = new Scanner(new File(fileName));
    }
    catch (FileNotFoundException e)
    {
          System.out.println("Error opening the file " + fileName);
          System.exit(0);
    } // end try/catch

    while (fromFile.hasNextLine())   // check if there are more lines
    {
          // read next line;
          // if there is no next line, an exception will be throw
          String line = fromFile.nextLine();
          System.out.println(line);
    } // end while
    fromFile.close();
```

Note:

- May also use scanner to read next int, double, long etc
- May also use BufferedReader & FileReader to read file, see textbook

- Data in program are stored in main memory. You may store these data in binary file. For example: int data is 4 bytes, if you store in a binary file, it occupies 4 bytes in file.

- You may store all kinds of program data into binary files, including primitive data type and object data

- Writing to a binary file using DataOutputStream

```
String fileName = "integers.bin";
Scanner keyboard = new Scanner(System.in);
System.out.println("Creating a file of integers that you enter.");

try
{
      // can throw FileNotFoundException
      // open file
     FileOutputStream fos = new FileOutputStream(fileName);
     DataOutputStream toFile = new DataOutputStream(fos);

     System.out.println("Enter nonnegative integers, one per line.");
     System.out.println("Enter a negative number at the end.");

   // read 4 integer data from keyboard, write to binary file using toFile
     int number = keyboard.nextInt();
     while (number >= 0)
     {     // can throw IOException, fo r example:  number is invalid
           toFile.writeInt(number);
           number = keyboard.nextInt();
     } // end while
}
catch (FileNotFoundException e)
{     // if file does not exist
     System.out.println("Error opening the file " + fileName);
     System.exit(0);
}
```

```
catch (IOException e)
{    // if cannot write to file
     System.out.println("Error writing the file " + fileName);
     System.exit(0);
} // end try/catch
finally
{
     try
     {
              // write file
             if (toFile != null)
                      toFile.close();          // can throw IOException
     }
     catch (IOException e)
     {
              System.out.println("Error closing the file " + fileName);
              System.exit(0);
     } // end try/catch
} // end finally
```

Note:

- Binary file "integer.bin" is not readable by text editor. You need to use Java program to read those data.
- May set "true" flag in "FileOutputStream(filename, true)" to append data (instead of truncating data in binary file)
- Other methods in DataOutputStream: writeChar(), writeBoolean() etc
- http://download.oracle.com/javase/6/docs/api/java/io/DataOutputStream.html

- Reading from a binary file using DataInputStream
  For each method in DataOutpurStream, DataInputStream provide a correspoing method for reading.

  Ref:
  http://download.oracle.com/javase/6/docs/api/java/io/DataInputStream.html

```
String fileName = "integers.bin";
DataInputStream fromFile = null;
try
{
    // can throw FileNotFoundException
    FileInputStream fis = new FileInputStream(fileName);
    fromFile = new DataInputStream(fis);
    System.out.println("Reading all the integers in the file " + fileName);

    while (true)
    {       // can throw EOFException, IOException
            int nextNumber = fromFile.readInt();    // will throw exception
            System.out.println(nextNumber);
    } // end while
}
catch (FileNotFoundException e)
{
    System.out.println("Cannot open file " + fileName);
    System.exit(0);
}
catch (EOFException e)
{
    System.out.println("End of the file " + fileName + " reached.");
}
catch (IOException e)
{
    System.out.println("Error reading the file " + fileName);
    System.exit(0);
} // end try/catch
finally
{
    try
    {
            if (toFile != null)
                    fromFile.close();               // can throw IOException
    }
    catch (IOException e)
    {
            System.out.println("Error closing the file " + fileName);
            System.exit(0);
    } // end try/catch
} // end finally
```

- Object serialization using ObjectInputStream and ObjectOutputStream

  - You can write/read object into/ from a binary file using ObjectInputStream and ObjectOutputStream – this is called object serialization

    http://download.oracle.com/javase/6/docs/api/java/io/ObjectOutputStream.html

    http://download.oracle.com/javase/6/docs/api/java/io/ObjectInputStream.html

  - To serialize an object, its class (or its superclass) must implement the marker interface Serialization.

    For example: to implement Serializable interface in class Student

```
import java.io.Serializable
public class Student implements Serializable
{   //….
```

- To write a student object into a binary file.

```
FileOutputStream fos = new FileOutputStream(fileName);
ObjectOutputStream toFile = new ObjectOutputStream(fos);
// . . .
toFile.writeObject(aStudent);  // write object to binaryfile
```

- To read a student object into a binary file.

```
FileInputStream fis = new FileInputStream(fileName);
ObjectInputStream fromFile = new ObjectInputStream(fis);
// . . .
Student joe = (Student)fromFile.readObject();  // cast to Student
```

- Array is serializable. Example:
  Student[] studentArray = (Student []) fromFile.readObject()