cis20.1
design and implementation of software applications I
fall 2007
lecture # I.3

topics:

- introduction to java, part 2
  - branching with `switch`
  - looping
  - java classes
  - more looping
  - inheritance in java
  - more classes
  - writing your own classes
  - static modifier
  - overriding methods
  - overloading methods
  - other terminology

on-line resources:

- java API specification: `http://java.sun.com/javase/6/docs/api/`
- java tutorial "getting started": `http://java.sun.com/docs/books/tutorial/getStarted/`

branching with `switch` (1): recall `if`...

```
public class ex2a {
  public static void main ( String[] args ) {
    int i = (Integer.valueOf( args[0] )).intValue();
    if ( i == 1 ) {
      System.out.println( "one, two, buckle my shoe" );
    }
    else if ( i == 3 ) {
      System.out.println( "three, four, shut the door" );
    }
    else if ( i == 5 ) {
      System.out.println( "five, six, pick up sticks" );
    }
    else if ( i == 7 ) {
      System.out.println( "seven, eight, lay them straight" );
    }
    else if ( i == 9 ) {
      System.out.println( "nine, ten, a big fat hen" );
    } // end if-else
  } // end main()
} // end of class ex2a
```


branching with `switch` (2): simple statements.

```
public class ex2b {
  public static void main ( String[] args ) {
    int i = (Integer.valueOf( args[0] )).intValue();
    switch( i ) {
    case 1:
      System.out.println( "one, two, buckle my shoe" );
      break;
    case 3:
      System.out.println( "three, four, shut the door" );
      break;
    case 5:
      System.out.println( "five, six, pick up sticks" );
      break;
    case 7:
      System.out.println( "seven, eight, lay them straight" );
      break;
    case 9:
      System.out.println( "nine, ten, a big fat hen" );
      break;
    } // end switch
  } // end main()
} // end of class ex2b
```

branching with `switch` (3): compound statements.

```
public class ex2c {
  public static void main ( String[] args ) {
    int i = (Integer.valueOf( args[0] )).intValue();
    switch( i ) {
    case 1:
    case 2:
      System.out.println( "one, two, buckle my shoe" );
      break;
    case 3:
    case 4:
      System.out.println( "three, four, shut the door" );
      break;
    case 5:
    case 6:
      System.out.println( "five, six, pick up sticks" );
      break;
    case 7:
    case 8:
      System.out.println( "seven, eight, lay them straight" );
      break;
    case 9:
    case 10:
      System.out.println( "nine, ten, a big fat hen" );
      break;
    } // end switch
  } // end main()
} // end of class ex2c
```


branching with `switch` (4): using default.

```
public class ex2d {
  public static void main ( String[] args ) {
    int i = (Integer.valueOf( args[0] )).intValue();
    switch( i ) {
    case 1:
    case 2:
      System.out.println( "one, two, buckle my shoe" );
      break;
    case 3:
    case 4:
      System.out.println( "three, four, shut the door" );
      break;
    case 5: case 6:
      System.out.println( "five, six, pick up sticks" );
      break;
    case 7: case 8:
      System.out.println( "seven, eight, lay them straight" );
      break;
    case 9: case 10:
      System.out.println( "nine, ten, a big fat hen" );
      break;
    default:
      System.out.println( "nothing left to say!" );
      break;
    } // end switch
  } // end main()
} // end of class ex2d
```


looping (1).

- if you want to do something many times
- two modes of loops:
  - counter controlled (now)
  - condition controlled (later)
- three loop statements:
  - `for`
  - `while`
  - `do`
- you can actually do both modes with each of the three statements, though some mode/statement pairings are more common than others

looping (2): counter-controlled `for`.

```
public class ex2e {
  public static void main ( String[] args ) {
    int n, i;
    n = (Integer.valueOf( args[0] )).intValue();
    System.out.println( "counting up to " + n + "..." );
    for ( i=0; i<n; i++ ) {
      System.out.print( i+ "  " );
    } // end for
    System.out.println();
  } // end of main
} // end of class ex2e
```

looping (3): counter-controlled `while`.

```
public class ex2f {
  public static void main ( String[] args ) {
    int n, i;
    n = (Integer.valueOf( args[0] )).intValue();
    System.out.println( "counting up to " + n + "..." );
    i = 0;
    while ( i < n ) {
      System.out.print( i+ "  " );
      i++;
    } // end while
    System.out.println();
  } // end of main
} // end of class ex2f
```


looping (4): counter-controlled `do`.

```
public class ex2g {
  public static void main ( String[] args ) {
    int n, i;
    n = (Integer.valueOf( args[0] )).intValue();
    System.out.println( "counting up to " + n + "..." );
    i = 0;
    do {
      System.out.print( i+ "  " );
      i++;
    } while ( i<n );
    System.out.println();
  } // end of main
} // end of class ex2g
```


looping (5): break and continue.

- these statements interrupt the normal flow of control of a program
- `break` is used in the `switch` statement to jump out of a case clause, without dropping down into the next one
- `break` can also be used from within a loop to interrupt the loop and jump to the end of the loop
- if loops are nested, it only jumps out of the loop where the `break` is imbedded
- `continue` is used from within a loop to interrupt the loop and jump to the next iteration of the loop
- in general, these statements are bad to use because they allow you to write code that jumps around and may be more prone to errors

looping (6): other facts about loops.

- you don't always have to count up
- you can count down too
- you don't always have to count by ones
- you can increment or decrement by any integer
- `do` loops always execute at least once
- `for` and `while` loops can be defined so that they don't execute (sometimes you might want to do this)

java classes (1).

- *classes* are the block around which Java is organized
- classes are composed of
  - data elements
    - *variables* — i.e., their values can change during the execution of a program
    - *constants* — i.e., their values CANNOT change during the execution of a program
      - like variables, they have a type, a name and a value
  - *methods*
    - modules that perform actions on the data elements
      - like variables, they have a type, a name and a value
      - unlike variables, the type can be *void*
    - *constructors* — special types of methods used to set up an object before it is used for the first time
- classes are *hierarchical*
- groups of related classes are organized into *packages*
- we'll start looking at *native* packages

java classes (2): the `java.lang` package.

- the superclass for all Java classes, at the top of the hierarchy
  - `java.lang.Object`
- *wrapper* classes that wrap around primitive data types; classes that define numeric limits and contain conversion methods
  - `java.lang.Boolean`
  - `java.lang.Character`
  - `java.lang.Byte`, `java.lang.Short`, `java.lang.Integer`, `java.lang.Long`, `java.lang.Float`, `java.lang.Double`
- string handling functions
  - `java.lang.String`
- math functions
  - `java.lang.Math`

java classes (3): `java.lang.Integer` class.

- a *constructor:*
```
public Integer( int value );
```
- some *constants:*
```
public static final int MIN_VALUE
public static final int MAX_VALUE
```
- some *methods:*
```
public int intValue();
public static String toString( int i );
public static Integer valueOf( String s );
public static int parseInt( String s );
```
- there is one for each primitive data type
- exercise:
  use the on-line Java documentation to look up the name of the wrapper classes for each of the primitive data types

java classes (4): `java.lang.String` class.

- some *constructors:*
```
public String();
public String( String value );
```
- some *methods:*
```
public static String valueOf( int i );
public int charAt( int index );
public int compareTo( String anotherString );
public int length();
```

java classes (5): `java.lang.Math` class.

- some *constants:*

```
public static final double E
public static final double PI
```

- some *methods:*

```
public static int abs( int a );

public static native double sin( double a );
public static native double cos( double a );
public static native double tan( double a );

public static native double pow( double a, double b );
public static native double sqrt( double a );

public static double random();
```


java classes (6): `java.util.Random` class (1).

- there is another way to generate random numbers besides using the `Math.random()` from the `java.lang.Math` class
- there are two methods defined in the `Random` class:

```
public Random();
public Random( long seed );
// constructor -- can be called with or without a seed

public void setSeed( long seed );
// sets the seed for the random number generator
```

- this class implements a *pseudo random number generator*
- which is really a sequence of numbers
- the *seed* tells the random number generator where to start the sequence

java classes (7): `java.util.Random` class (2).

- more methods defined in the `Random` class, used to get the random numbers:

```
public float nextFloat();
// returns a random number between 0.0 (inclusive) and
// 1.0 (exclusive)

public int nextInt();
// returns a random number that ranges over all possible
// int values (positive and negative)
```


java classes (8): `java.util.Date` class (1).

- this class is handy for getting the current date
- or creating a `Date` object set to a certain date
- some methods defined in the `Date` class:

```
public Date();
public Date( long date );
// constructor -- called without an argument, uses the
// current time; otherwise uses the time argument

public boolean after( Date arg );
public boolean before( Date arg );
public boolean equals( Object arg );
public long getTime();
public String toString();
```

- computer time is measured in milliseconds since midnight, January 1, 1970 GMT
- a `Date` object is handy to use as a seed for a random number generator

java classes(9): instantiating objects.

- in order to use a class, you *instantiate* it by creating an *object* of that type
- this is kind of like declaring a variable

```
import java.util.*;
public class ex2h {
  public static void main( String[] args ) {
    Date   now = new Date();
    Random rnd = new Random( now.getTime() );
    System.out.println( "here's the first random number: "+
                        rnd.nextInt() );
  } // end of main()
} // end of class ex2h
```


more looping (1).

- back to loops
- condition-controlled loops

more looping (2): condition-controlled `while`.

```
public class ex2i {
  public static void main ( String[] args ) {
    int card1=(int)(Math.random()*52);
    int card2=(int)(Math.random()*52);
    int count=1;
    while ( card1 != card2 ) {
      System.out.println( "count="+count+" card1="+card1+
                          " card2="+card2 );
      card1=(int)(Math.random()*52);
      card2=(int)(Math.random()*52);
      count++;
    } // end while
    System.out.println( "MATCH! count="+count+" card1="+card1+
                        " card2="+card2 );
    System.exit( 0 );
  } // end of main
} // end of class ex2i
```


more looping (3): condition-controlled `do`.

```
public class ex2j {
  public static void main ( String[] args ) {
    int card1=(int)( Math.random()*52 );
    int card2=(int)( Math.random()*52 );
    int count=1;
    do {
      System.out.println( "count="+count+" card1="+card1+
                          " card2="+card2 );
      card1=(int)( Math.random()*52 );
      card2=(int)( Math.random()*52 );
      count++;
    } while ( card1 != card2 );
    System.out.println( "MATCH! count="+count+" card1="+card1+
                        " card2="+card2 );
    System.exit( 0 );
  } // end of main
} // end of class ex2j
```

more looping (3): condition-controlled `for`.

```
public class ex2k {
  public static void main ( String[] args ) {
    int card1=(int)( Math.random()*52 );
    int card2=(int)( Math.random()*52 );
    int count=1;
    for ( ; card1 != card2; ) {
      System.out.println( "count="+count+" card1="+card1+
                          " card2="+card2 );
      card1=(int)( Math.random()*52 );
      card2=(int)( Math.random()*52 );
      count++;
    } // end for
    System.out.println( "MATCH! count="+count+" card1="+card1+
                        " card2="+card2 );
    System.exit( 0 );
  } // end of main
} // end of class ex2k
```

OR you can include all updates in the update section of the for loop:

```
for ( ; card1 != card2; card1=(int)(Math.random()*52),
                        card2=(int)(Math.random()*52),
                        count++ ) {
  System.out.println( "count="+count+" card1="+card1+
                      " card2="+card2 );
} // end for
```


inheritance in java (1).

- *inheritance* is the means by which classes are created out of other classes
- it is a cornerstone of object-oriented programming
- the idea is to create classes that can be re-used from one application to another
- classes contain *data objects* and *methods*
- you want to be able to change the *data type* of the data objects and still be able to use the same methods
- you also want to be able to change the flavor of what the methods do

inheritance in java (2).

- think of the most primitive Java class, `Object` as being at the root of the inheritance tree
- all other classes are "children" or *subclasses* of that class
- here is an example of the inheritance tree for `Integer`:

```
java.lang.Object
  |
  +--java.lang.Number
        |
        +--java.lang.Integer
```

- `Integer` is a subclass of `Number` and `Number` is a subclass of `Object`
- `Integer` is also a subclass of `Object`
- conversely a parent is also called a *superclass*
- `Object` is a superclass of `Number` and `Number` is a superclass of `Integer`
- `Object` is also a superclass of `Integer`
- `Object` is also called the *base class* of `Integer`

inheritance in java (3).

- as you move DOWN the inheritance tree from the root to the leaf, you are *extending* subclasses from parent classes
  - parent classes are also called *superclasses*
  - or *base classes*
  - children classes are *derived* from their parents
- as you move UP the inheritance tree from the leaf to the root, you can say that each subclass is a *more specific* version of its parent
- this is known as the *is-a* relationship between a subclass and the parent class that the child extends
- the keyword `this` is used to specify a member of the current or immediate class

more classes (1): define objects.

- are "blueprints" for creating *instances* of objects
- example: a house
  - class = architect's blueprint
  - instance = a house built following that blueprint
- *instantiate* = to build the house
- you can build MANY houses using the same blueprint, so you can instantiate many objects using the same class

more classes (2): contain members.

- **data declarations** *(e.g., the people and the stuff inside the house)*
  - constants
  - variables
- **methods** *(e.g., the things people do with the stuff)*
  - actions that are performed on the object and/or with its data
  - a *constructor* is a special method used to *instantiate* an object of that class
  - some methods may change the values of the variables
  - some methods may *return* the values of the variables
- **scope** *(e.g., where can people do things with the stuff?)*
  - *local* vs *global*
  - *instance data*
  - *method data*

more classes (3): instantiating objects.

- in order to use a class, you *instantiate* it by creating an *object* of that type
- this is kind of like declaring a variable

```
import java.util.*;
public class ex3a {
  public static void main( String[] args ) {
    Date   now = new Date();
    Random rnd = new Random( now.getTime() );
    System.out.println( "here are ten positive integers:" );
    for ( int i=0; i<10; i++ ) {
      System.out.println( Math.abs( rnd.nextInt() ));
  } // end of main()
} // end of class ex3a
```


writing your own classes (1).

- you can create your own classes in two ways:
  - by writing a completely new class
  - by *extending* an existing class

writing your own classes (2).

- when you write your own class, you can define
  - "global" data elements
    - variables
    - constants
  - methods
  - constructor

writing your own classes (3): variables.

- have a name, type and value
- value is initialized, to 0 for numbers (unlike C)
- have "global" scope if they are declared outside of any method

writing your own classes (4): constants.

- their values CANNOT change during the execution of a program
- i.e., their values remain *constant*
- like variables, they have a type, a name and a value
- the keyword `final` indicates that the variable is a *constant* and its value will not change during the execution of the program
- example:

```
public class java.lang.Math {
  static final double PI=3.1415927...;
  .
  .
  .
} // end of Math class
```


writing your own classes (5): method declaration.

- like a variable, has:
  - data type:
    - primitive data type, or
    - class
  - name (i.e., identifier)
- also has:
  - arguments (optional)
    - also called *parameters*
    - *formal parameters* are in the blueprint, i.e., the method declaration
    - *actual parameters* are in the object, i.e., the run time instance of the class
  - throws clause (optional)
    (we'll defer discussion of this until later in the term)
  - body
  - return value (optional)

writing your own classes (6): method use.

- program control jumps inside the body of the method when the method is *called* (or *invoked*)
- arguments are treated like local variables and are initialized to the values of the calling arguments
- method body (i.e., statements) are executed
- method *returns* to calling location
- if method is not of type *void*, then it also *returns* a value
  - return type must be the same as the method's type
  - calling sequence (typically) sets method's return value to a (local) variable; or uses the method's return value in some way (e.g., a print statement)

writing your own classes (7): constructor.

- a constructor is a special method that is invoked when an object is *instantiated*
- a constructor can have arguments, like any other method
- a constructor does not return a value
- a constructor's name is the same as the name of the class to which it belongs
- a constructor is invoked by using the *new* keyword
- example:

```
Date now = new Date();
Random r1 = new Random();
Random r2 = new Random( now.getTime() );
```


writing your own classes (8): encapsulation and visibility.

- objects should be self-contained and *self-governing*
- only methods that are part of an object should be able to change that object's data
- some data elements should not even be seen (or visible) outside the object
- *public* data elements can be seen (i.e., read) and modified (i.e., written) from outside the object
- *private* data elements can be seen (i.e., read) and modified (i.e., written) ONLY from inside the object
- typically, **variables** are **private** and **methods** that provide access to them (both read and write) are **public**
- typically, **constants** are **public**
- example: house
  - walls provide privacy for the inside
  - windows provide public viewing of some of the inside

writing your own classes (9): example.

```
public class Coin {

  // declare constants
  public static final int HEADS = 0;
  public static final int TAILS = 1;

  // declare variables
  private int face;
  private int value;

  // constructor
  public Coin( int value ) {
    this.value = value;
    flip();
  } // end of Coin()
```

```
// flip the coin by randomly choosing a value for the face
public void flip() {
  face = (int)(Math.random()*2);
} // end of flip()

// return the face value
public int getFace() {
  return face;
} // end of getFace()

// return the coin's value
public int getValue() {
  return value;
} // end of getValue()
```



```
  // return the coin's face value as a String
  public String toString() {
    String faceName;
    if ( face == HEADS ) {
      faceName = "heads";
    }
    else {
      faceName = "tails";
    }
    return faceName;
  } // end of toString()

} // end of class Coin
```


static modifier (1).

- when we *instantiate* an object in order to use it, we are creating an *instance variable*
  e.g., `Random r = new Random();`
- some members in some classes are *static* which means that they don't have to be instantiated to be used
- for example, all the methods in the `java.lang.Math` class are `static`
  - you don't need to create an object reference variable whose type is `Math` in order to use the methods in the `Math` class
  - e.g., `Math.abs()`, `Math.random()`
- you use the name of the class preceding the dot operator, instead of the name of the instance variable, in order to access the static members of the class
- e.g., `Math.random()` vs `r.nextFloat()` (where `r` is the instance variable of type `Random` that we created above)
- that is why we can use `main()` without instantiating anything
  i.e., `public static void main()`

static modifier (2).

- constants, variables and methods can all be static
- except constructors
  (since they are only used to instantiate, it doesn't make sense to have a static constructor)
- typically, *constants* are static
- example:

```
public class Coin {
  public static final int HEADS=0;
  public static final int TAILS=1;
  .
  .
  .
} // end of Coin class
```

- we can now access `Coin.HEADS` and `Coin.TAILS` without instantiating and/or without referring to a specific instance variable

overriding methods.

- when you *extend* a class, you can *override* methods defined in the parent class by defining them again in the child (and giving the child version different behavior)
- the rule is: *the version of any method that is invoked is the definition closest to the leaf of the tree*
- if you want to refer to the version of the method in a class's superclass, you use the `super` reference

overloading methods (1).

- in addition to changing precisely what a method does, you can also change the arguments to that method
- this is very useful if you are changing the data type of data objects defined in the class
- you can create a new version of a method which has different arguments from the version of the method defined in the class's superclass
- this is what happens when we use different versions of the `println()` method:

```
int i = 5;
String s = "hello";
System.out.println( i );
System.out.println( s );
```


overloading methods (2).

- in other words, you are using the same method name with formal parameters of different types
- example:
  - `java.lang.System` *has-a* variable called `out`, which *is-a* `java.io.PrintStream`
  - whose declarations include:

```
public void println();
public void println( boolean x );
public void println( char x );
public void println( double x );
public void println( float x );
public void println( int x );
public void println( Object x );
public void println( String x );
```

- these are all different ways of *printing* data, but the difference is the type of *object* being printed

other terminology...

- *polymorphism*
  - "having many forms"
  - lets us use different implementations of a single class
  - we will talked about this later in relation to *interfaces*
  - a polymorphic reference can refer to different types of objects at different times
- *abstract* class
  - represents a generic concept in a class hierarchy
  - cannot be instantiated — can only be extended

example.

```
public class Quarter extends Coin {

  // overload constructor
  public Quarter() {
    value = 25;
    flip();
  } // end of Quarter()

  OR

  public Quarter() {
    super( 25 );
  } // end of Quarter()


} // end of class Quarter
```


exercise.

- create a class called `Card` which is a playing card
- the card has a face (hearts, diamonds, clubs or spades)
- the card has a value (2..10, J, Q, K, A), all face cards have value 10
- define a constructor that randomly sets the card's face and value
- define methods to return the card's face and value
- define another method called `pick` that will change the card's face and value, as if you picked another card from the deck
- create a second class that contains a `main()` method
- define variable(s) in the second class of type `Card`
- loop inside the `main()`, randomly picking cards until the total is greater than or equal to 21
- assume that you replace each card in the deck immediately after it has been picked (so you don't have to keep track of which cards you have picked)
- *extension:* modify the exercise so that you do keep track of which cards have been picked