cisc3120
design and implementation of software applications I
spring 2015
lecture # I.4

topics:

- introduction to java, part 4
  - 'this'
  - references
  - exception handling
  - comparing objects
  - vectors
  - utility classes

'this' keyword (1).

- *this* is a reference to the current object from within an instance method or a constructor by using this.
- The most common reason for using the this keyword is because a field is shadowed by a method or constructor parameter.

```
public class Coin {
  public Coin() { value = 0; }

  public Coin(int value) { this.value = value; }

  public void setValue(int value) {
   this.value = value;
   this.value += 10;
  }

  private int value;
```



```
  public static void main(String[] args) {
   Coin quarter = new Coin();
   quarter.setValue(15);
  }
}
```

- In the above example, this refers to the quarter object and local variable value in setValue holds 15.

'this' with constructor (2).

- *this* keyword can be used to call another constructor in the same class.
- It is called an *explicit constructor invocation*

```
public class Coin {

  // default constructor
  public Coin() {
    this(25);
  }

  // overload constructor
  public Coin(int v) {
    value = v;
    flip();
  }

}
```

references (1).

- when we declare a variable whose data type is a class, we are declaring an object reference variable
- that variable *refers to* the actual object stored in computer's memory
- *an object reference variable and an object are two separate things*
- declaration of an object reference variable:

```
Coin x;
```

- creation of an object (also called "construction", "instantiation"):

```
x = new Coin();
```


references (2).

- when you declare a variable as a primitive data type, the computer sets aside a fixed amount of memory, based on the size of the data type
- when you declare a variable of any other data type (i.e., a class), you are actually declaring a *reference*
- a reference is typically the size of an *int* or a *long*
- it stores information for the object, which the JVM uses to locate the address in the computer's memory, where the actual data will be kept
- you can think of it like a telephone book
  - the phone book has a bunch of addresses in it
  - but not the actual buildings
  - just the *locations* of buildings

references (3).

- here's how it works inside the computer
- given the following declarations:

```
int    i = 45;
String s = "hello";
```

- the memory looks something like this:

```
 i       s
[45]     • → [hello]
```

- `i` is the label for the location in memory where the actual data is stored — in this case the `int 45`
- `s` is the label for the location in memory where the *address* is stored; the address is the location in memory where the actual data for `s` is stored
- in C, this is called a *pointer*
- we say that `s` *points to* or *references* the location in memory where the actual data for `s` is stored

references (4).

- let's go back to the `Coin` example
- comment out the `toString()` method and re-run the example
- here's the output now:

```
i[0]=Coin@73d6a5
i[1]=Coin@111f71
i[2]=Coin@273d3c
i[3]=Coin@256a7c
i[4]=Coin@720eeb
i[5]=Coin@3179c3
i[6]=Coin@310d42
i[7]=Coin@5d87b2
i[8]=Coin@77d134
i[9]=Coin@47e553
```

- these are the *references* of the array elements
- we can see these reference values because we took out the `toString()` method — calling `System.out.println(pocket[i])` automatically coerces its argument (`pocket[i]`) to a `String` so it can print it; if there is no explicit `toString()` method in the class, then a reference is the closest `String` representation

references (5).

- when an object reference variable has been declared but the object does not refer to an object, then the object reference variable is called a *null* reference
- for example:

```
Coin[] x = new Coin[5];
x[0].flip();
```

- will generate an error called a `NullPointerException` because the object which `x[0]` refers to has not been instantiated
- you can use a constant called `null` to check if an object reference variable is null
- for example:

```
Coin[] x = new Coin[5];
if(x[0] != null) {
  x[0].flip();
}
```


references (6).

- an *alias* is an object reference variable that refers to an object that was previously constructed and is already referred to by another object reference variable
- for example:

```
Coin x = new Coin();
Coin y;
y = x;
y.flip();
```

- `y` is called an "alias" of `x` (and vice versa) because they both refer to the same object in the computer's memory

references (7).

- garbage collection is necessary when all references to an object are gone
- because when there are no object reference variables, then there is no way to know where in memory an object is located
- Java handles this for you automatically
- the JVM periodically invokes *automatic garbage collection* while it is running
- all the memory that is allocated to an application but is not being used is "restored" so that it can be re-allocated to the application later
- if you want to perform some garbage collection on a class that you create yourself, then you would write a method called `finalize()` and whenever the automatic garbage collection was invoked and cleaned up an object of your class type, then your `finalize()` method would be called

references (8).

- in Java all method calls are *call-by-value*.
- so be careful about what changes!
- here's an example using two classes:
  - `Num`
  - `ParameterTester`

references (9).

```
public class Num {

  private int value;

  public Num(int update) {
    value = update;
  }

  public void setValue(int update) {
    value = update;
  }

  public String toString() {
    return value+"";
  }

}
```


references (10).

```
public class ParameterTester {
    public static void main( String[] args ) {
        int a1 = 111;
        Num a2 = new Num(222);
        Num a3 = new Num(333);

        System.out.println("before call: \ta1=" + a1 + ", \ta2=" + a2 + ", \ta3=" + a3);
        changeValues(a1, a2, a3);
        System.out.println("after call: \ta1=" + a1 + ", \ta2=" + a2 + ", \ta3=" + a3);
    }

    public static void changeValues(int f1, Num n1, Num n2) {
        System.out.println("start call: \tf1=" + f1 + ", \tn1=" + n1 + ", \tn2=" + n2);

        f1 = 999;
        n1.setValue(888);
        n2 = new Num(777);

        System.out.println("end call: \tf1=" + f1 + ", \tn1=" + n1 + ", \tn2=" + n2);
    }
}
```


references (11).

- sample output:

```
before call:    a1=111  a2=222  a3=333
start call:     f1=111  f2=222  f3=333
end call:       f1=999  f2=888  f3=777
after  call:    a1=111  a2=888  a3=333
```


references (12).

- when an object reference variable is declared `final`, its reference cannot be changed.
- however, the object itself can be modified:

```
// binds the reference quarter to the object
final Coin quarter = new Coin(25);

// error! cannot assign a value to final var
quarter = new Coin(5);

// OK!
quarter.setValue(5);

// prints out the value of the coin as 5
System.out.println(quarter);
```

- `final` in Java is not the same as const in C/C++

exception handling (1).

- *exceptions* are 'exceptional' situations which would cause a serious error in your program such as bad input data, file not found etc.
- `try` clause contains code which may generate an exception.
- `catch` clause contains code to execute in case the error happens; i.e., where to go if the exception gets *caught*:

```
try {
  // (1) statement that may cause an exception
  // (2) other statements
}
catch(Exception e) {
  // (3) do something with the exception
}
```

- if the statement at (1) above throws an exception the code will jump to catch statement and 'handle' the exception as stated in (3)
- if no exception is generated at (1), statments at (2) will be executed

exception handling (2).

- an exception is 'thrown' by creating an exception object and placing keyword `throw` before the object:

```
throw new NumberFormatException();
```

- all exceptions in Java derives from `Exception` class and generally divided into two categories:
  - `RuntimeException` which happens due to programming errors (e.g. `ArrayIndexOutOfBoundsException`, `NullPointerException`)
  - `IOException` which refers to situations referring to bad input to an otherwise good program.
- In general, you should handle `IOExceptions` and fix the bugs that causes `RuntimeException`

exception handling (3).

- if a method throws an exceptions its signature contains the `throws` keyword followed by the type of exception thrown. `Integer.parseInt` method signature in API docs:

```
public static int parseInt(String s)
                    throws NumberFormatException
```

- try-catch basic form example:

```
try {
  int i = Integer.parseInt("a");
}
catch(NumberFormatException e) {
  System.out.println("there was an error: " + e.getMessage());
}
```


exception handling (4).

- you can have multiple `catch` blocks for different exception types:

```
Coin c = null;
try {
  int i = Integer.parseInt("a");
  c.flip();
}
catch(NumberFormatException nfe) {
  System.out.println("there was an error: " + nfe.getMessage());
  nfe.printStackTrace();
}
catch(NullPointerException npe) {
  System.out.println("You should probably initialize the Coin" +
                     " object rather than catching this exception!");
}
finally {
  System.out.println("Done!");
}
```

comparing objects (1).

- comparing two Java objects is tricky
- you have to be careful of what you are comparing:
  - is it the *value* of some member(s) of the class?
  - or is it the *reference*? (like a pointer in C/C++)
- using == compares the *references* (if they refer to the same object)
- which is not the same as comparing the values of member(s) of the class
- many classes have a method called `compareTo()` to compare the value of member(s) of the class

comparing objects (2).

- here's an example from the `Coin` class:
  - comparing the value of the `face` member of two coins:

```
Coin coin0 = new Coin(10);
Coin coin1 = new Coin(10);
if(coin0.getValue() == coin1.getValue()) {
  System.out.println("coins 0 and 1 have the same value");
}
```

  - versus comparing the references:

```
if(coin0 == coin1) {
  System.out.println("coins 0 and 1 are the same");
}
```


comparing objects (3).

- in order to compare the value of two `Strings`, we need to use the method
  `public int compareTo(String str)`
  from the `java.lang.String` class
- this method does a *lexical comparison* of its `String` argument with the current object (i.e., its instantiated value)
- it returns an `int` as follows:

if the current object...	*then the method returns*
is the same text as `str`	0
comes lexically before `str`	an `int` < 0 (e.g., -1)
comes lexically after `str`	an `int` > 0 (e.g., $+1$)

- using == to compare two `Strings` compares the references, NOT the values of the text they store
- this is the same for comparing any two objects in Java
- most classes define a `compareTo()` method, just as most classes define a `toString()` method

comparing objects (4).

- for example:

```
public class CompareString {
  public static void main(String[] args) {
    String s1 = new String("hello");
    String s2 = new String("hello");
    System.out.println("s1=[" + s1 + "]");
    System.out.println("s2=[" + s2 + "]");
    System.out.println("(s1 == s2) = " + (s1 == s2));
    System.out.println("s1.compareTo(s2)=" + s1.compareTo(s2));
    System.out.println("s2.compareTo(s1)=" + s2.compareTo(s1));
  }
}
```

- sample output:

```
s1=[hello]
s2=[hello]
(s1 == s2) = false
s1.compareTo(s2)=0
s2.compareTo(s1)=0
```

comparing objects (5).

- so we could add to our `Coin` class:

```
public int compareTo(Coin coin) {
  if(value == coin.getValue()) {
    return 0;
  }
  else if(value < coin.getValue()) {
    return -1;
  }
  else {
    return 1;
  }
}
```


vectors (1).

- Java has a nice class which handles arrays dynamically: `java.util.Vector`
- the elements of a `Vector` can be any type of Java `Object`
- note that when you fetch an element from a vector, you have to cast it from a generic object to the specific class type the object should be (see example below)
- some methods:
  - constructor: `Vector();`
  - `public void addElement(Object obj);`
  - `public void insertElementAt(Object obj, int index);`
  - `public void removeElementAt(int index);`
  - `public void removeAllElements();`
  - `public void setElementAt(Object obj, int index);`
  - `public Object elementAt(int index);`
  - `public int size();`

vectors – example.

```
import java.util.*;
import java.io.*;

public class Ex4c {

  public static void main(String[] args) {
    Vector pocket;
    int npocket = Integer.parseInt(args[0]);

    pocket = new Vector(npocket);
    for(int i = 0; i < npocket; i++) {
      pocket.addElement(new Coin());
    }

    for(int i = 0; i < npocket; i++) {
      Coin tmp = (Coin)pocket.elementAt(i);
      System.out.print(tmp + " ");
    }
    System.out.println();
  }

}
```


vectors – things to notice.

- notice that we instantiate twice...
- notice that we instantiate in the call to `pocket.addElement()`:

```
pocket.addElement(new Coin());
```

- notice that we *cast* the return from `pocket.elementAt()`:

```
Coin tmp = (Coin)pocket.elementAt(i);
```

utility classes: `java.util.StringTokenizer`.

- used to break up a string into "tokens", i.e. components
- each token is separated by a "delimiter"
- default delimiter is whitespace
- but you can set another value for delimiter
- primary method used: `public String nextToken();`
- example:

```
line = infile.readLine();
tokenizer = new StringTokenizer(line);
name = tokenizer.nextToken();
try {
  units = Integer.parseInt(tokenizer.nextToken());
  price = Float.parseFloat(tokenizer.nextToken());
}
catch(NumberFormatException nfx) {
  System.out.println("error in input; line ignored: " + line);
}
```


utility classes: `java.text.DecimalFormat`.

- used to format decimal numbers
- construct an object that handles a format
- use that format to output decimal numbers
- formatting patterns include:
  - 0 used to indicate that a digit should be printed, or 0 if there is no digit in the number (i.e., leading and trailing zeros)
  - # used to indicate that if there is a digit in the number, then it should be printed; indicates rounding if used to the right of the decimal point
- example:

```
DecimalFormat fmt = new DecimalFormat("#.00");
double price;
System.out.println("price = $" + fmt.format(price));
```