CSC 142

Instance methods

[Reading: chapter 4]

Motivation

- Divide and conquer
  - Break up a complex task into a set of tasks.
  - Each simpler task is coded in a separate method.
- Simplify the development of code (faster, less bugs).

An illustration

- Drawing a mountain landscape

```
public void drawLandscape()
{
 drawMountainRange();
 drawChalet();
 drawTrees();
}
```

- What the method does is obvious
- How it does it is relegated in the methods drawMountainRange(),…

Defining a method

- e.g.,

```
public boolean isAHoliday(Date d){/*code*/}
```

- Inside of a class block {}
- Start with the access modifier (public or private)
- Follow with the type of the method
  - e.g. if the method returns an int, write int for the type of the method
- Follow with the method name
  - first word: lowercase, others: capitalize the 1st letter.
- Follow with the list of formal parameters inside ()
- The method code is written between { and }

Access modifier: public

- The method is part of the class interface
- Anyone using the class can access the method

```
// file MyClass.java
public class MyClass{
  public void sayHello()
  { System.out.println("Hello");}
  // constructor and other methods...
}
// for any user of MyClass
MyClass mc = new MyClass();
mc.sayHello(); //OK, sayHello is public
```


Access modifier: private

- The method is part of the class implementation
- The method can only be accessed within the class

```
// file MyClass.java
public class MyClass{
  private void sayHello(){
   System.out.println("Hello");
   }
  public void saySomething(){
   sayHello(); // OK
   }
}
// file OtherClass.java
public class OtherClass{
  public void doSomething(){
   MyClass mc = new MyClass();
   mc.sayHello(); //Oups! Doesn't compile
  }
}
```

void return value

- If the method doesn't send back a result, use void as the type of the method

```
public void printWelcome()
{
  System.out.println( "Welcome to Java!");
   return; // optional statement
}
```


Returning a value

- The method sends 'something' back to whoever called it. That 'something' (=a result in some form) has a type. The type must appear next to the method name, e.g.

```
public boolean isZero(int val)
{
  // is val 0?
  if (val==0)
   return true;
 else
   return false;
}
```

The method is of type boolean since it returns a boolean (true or false).

Formal and actual parameters

- Store in the formal parameters the input values sent to the method, e.g. in class Output:

```
public void printInfo(String name, int age)
{
  System.out.println("My name is " + name +
    ". I am "+age+" years old.");
}
```

formal parameters

- actual parameters: what is sent to the method

```
Output output = new Output();
output.printInfo("Sandra",26);
//prints My name is Sandra. I am 26 years old.
```

actual parameters

- If no input, write *methodName*()

Rule for calling a method (1)

- Actual and formal parameters must match
  - in number
  - in type
  - in order

```
//OK
output.printInfo("Sandra",26);
//Oups! wrong order
 output.printInfo(26,"Sandra");
// Boom! wrong number
 output.printInfo(26);
// No way! 26.5 is not an int
 output.printInfo("Sandra",26.5);
```


Rule for calling a method (2)

- Actual and formal parameters don't have to match by name, e.g.

```
// In the Output class
public void printInfo(String name,int age)
{
  System.out.println("My name is " + name +
    ". I am "+age+" years old.");
}
// in some other method in some other class
String someName = "David";
int someAge = 11;
Output output = new Output();
output.printInfo(someName,someAge); // OK
```


Control flow (1)

```
public class CalendarUtil {
  public boolean isLeapYear(int year)
  { /*code*/ }
  /* constructor and other methods... */}

public class MyClass{
 private CalendarUtil cu = new CalendarUtil();
  public void checkBirthYear(int birthYear){
   if (cu.isLeapYear(birthYear))
     System.out.println("Born on a leap year");
  }
}
```

- What happens when checkBirthYear(1988) is executed?

Control flow (2)

- Execute checkBirthYear(1988)
  - initialize birthYear to 1988
  - evaluate the condition in if statement: need to call the instance method isLeapYear of cu, with an actual parameter equal to 1988
- Leave checkBirthYear, start executing isLeapYear
  - initialize year to 1988
  - execute the code in isLeapYear: should return true
- Leave isLeapYear with the value true. Resume execution in checkBirthYear.
  - The if condition is true: print "Born on a leap year"

Control flow (3)

- Inside a method, code is executed top down, statement by statement, unless
  - another method is called: move the control flow to that method
  - a return statement is encountered: move the control flow to the caller of the method (if non void return, send back a value as well).
  - an if statement is encountered: skip statements in if block (false condition) or, if there is an else, in else block (true condition)
  - a loop is encountered: repeat some statements several times (see later).

pre and post conditions

- Document the methods. One approach: use
  - preconditions: what must be true to call the method
  - postconditions: what is true when done executing the method
- e.g. to describe boolean isLeapYear(int year)

```
/* precondition: year > =1752 (when the
   gregorian calendar was adopted in England) */
/* postcondition:
   result==(year is a leap year) */
```

- Could have used javadoc comments /** and */

Defining variables (1)

- As an instance field: within a class outside any method. The field is visible throughout the class and beyond if declared public.
- As a local variable: within a method.

```
public int foo(double x)
 {
   int i;
    // some more code
 }
```

- x and i can be used only between { and } defining foo. x and i are local to foo.

Defining variables (2)

- What is the difference between x and i?
  - x is a formal parameter. It is automatically initialized when foo is called. It receives the value of the actual parameter, e.g.

    int someInt = foo(3.2); // in foo, for this call x is 3.2
  - i is not automatically initialized. In Java, a local variable used without having been initialized generates a compilation error.

```
public int foo(double x) {
   int i;
   if (x > 10.)
    return i;  /* error: What is i ? */
   else
    return -i; /* error: What is i ? */
}
```


Local variable scope

- Where the variable is visible (in other words where it can be used)
  - A local variable is only visible within the block where it is declared.

```
public void foo()
{
  int i=2;
   { // inner bloc
      i=3; //OK
      String s = "abc"; //OK
   } // end inner block
   s="def";
   //error! s visible only in inner block
}
```

Variable lifetime

- Local variable (within a method)
  - Created when the declaration line is executed.
  - Destroyed when exiting the block where the variable was declared (the memory used by the variable is reclaimed).
    If it is a reference, is the object garbage collected? No, if there is another reference to the object. Yes, if there is no other reference of the object.
- Instance variable
  - Created when the object is constructed
  - Destroyed when the object is destroyed

this keyword (1)

- An instance method of a class is always called using an instance of the class, e.g.

```
window.add(circle);
```

- The instance object is not part of the actual parameter list. However, it is still passed to the method implicitly.
- It can't appear in the formal parameter list
- To refer to the object used to make the call, use the keyword this.

Using this: an example (1)

- In an instance method, access the object implicitly passed to that method with the this keyword.

```
public class SomeClass{
 private int var;
 public void changeVar(int n){
  this.var = n;/*OK, but superfluous*/}
}
```

- Could also do

```
public void changeVar(int var)
{this.var=var;}
```

var is local to changeVar. Since the name is the same as the instance variable var, we need to use this to refer to the instance variable var (potentially confusing). The local variable shadows the instance variable.

Using this: an example (2)

- In an instance method, any reference to an instance field or any call to another instance method done without an explicit reference uses the implicit object.

```
public class Bar{
 public void foo1(){/*code*/}
 public void foo2()
 {
   foo1(); // call the instance method foo1
           // of the implicit object.
          // same as this.foo1();
 }
}
```


Using this: another example

- Consider a Clock class to model a clock. Clock has a method tick to add 1 second to the time.

```
Clock clock = new Clock(); //0h0mn0s
clock.tick(); // 0h0mn1s
```

- What if we want to write

```
clock.tick().tick(); // add 2 seconds
```

- Define tick as

```
public Clock tick()
{
  /* code to add one second to the current
  time... */
 return this;
}
```


Program organization

- Define the classes: one java file per class
- For each class, define the instance fields and methods.
- Inside of each method, define local variables as needed.
- What about import?
  - used to access libraries organized in packages.
  - can also define our own packages when defining new classes. Not needed for simple programs.
  - In practice, classes that are defined within the same folder can access one another without using any import.