David M. Keil, Framingham State University

CSCI 252 Computer Science II Using Java

1. Java method design

1. Procedural abstraction and Java methods
2. Local variables, parameters, return values
3. Documenting and testing methods
4. Recursive methods
5. Java file I/O

Inquiry

- How may we represent *things*, *people*, *places*, and *events*?
- What is a *procedure*?
- How does a software developer step back from the details of data items and computational steps?
- What is *abstraction*?

Topic objective

Define and test Java static methods with parameters and return values.

1. Procedural abstraction and Java methods

- Did you ever see program code that went on for pages?
- How may the steps of a program be subdivided for modularity?
- What is a *subprogram*?
- How is *println* implemented?

Subtopic objectives

1.0 Recall basic Java method concepts*^{m}

1.1a Explain procedural abstraction**

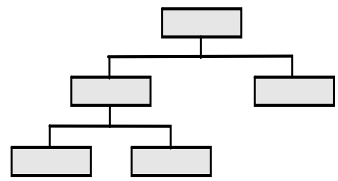
1.1b Define, test a Java method**†

Procedural abstraction

- Definition: the creation of *subprograms*
- Statements and control structures are packaged as named modules
- The statements are replaced by *calls* to the subprograms
- *Procedural abstraction* enables *modularity* and *reuse* of code
- Defining *data types*, is called *data abstraction*, part of *object-oriented design*

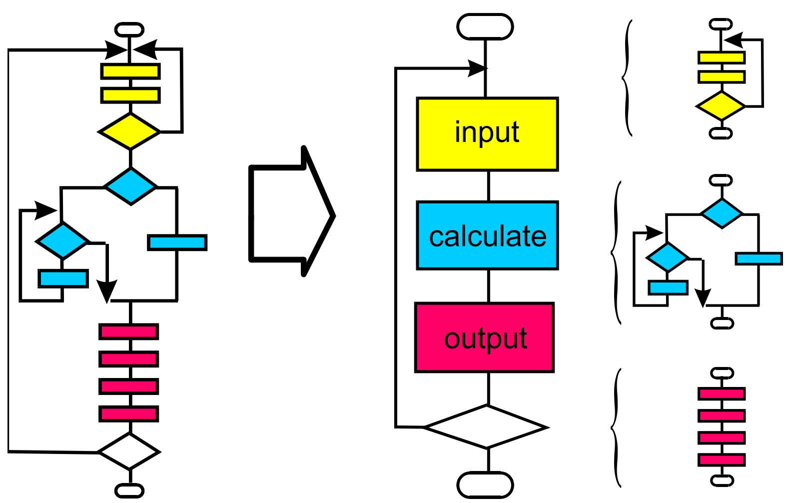
Modular decomposition

- Some solutions are too complex to be easily understood as a single unit
- A structured design can be decomposed into simpler *modules*



- This breaking down is called *modular decomposition*, implemented by *procedural abstraction* (writing of subprograms)
- We may continue the breakdown as needed by *stepwise refinement*

Example of modular decomposition

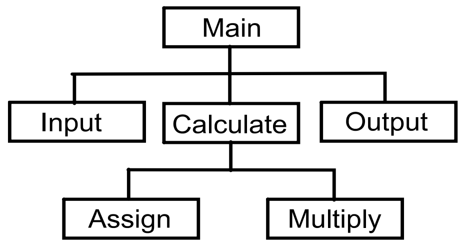


Separate modules are easier to understand.

Module hierarchy diagrams

- *Example: Main* invokes *Input, Calculate,* and *Output; Calculate* calls *Assign* and *Multiply*

Main
A. Input
B. Calculate
  1. Assign
  2. Multiply
C. Output



- A module hierarchy chart shows module dependencies, whereas a flowchart shows order of execution.

Java methods

- In Java, subprograms are called "methods"
- These are not general ways of doing things, but specific sequences of commands
- All Java methods are *members of classes*
- Every program defines at least one class with a special method, *main*, that executes when the program executes
- Other methods may be defined and may be called from *main*

Writing and calling Java methods

Example: Drawing a rectangle

```
public static void
  main(String[] args)
{
  horizontal();
  vertical();
  vertical();
  horizontal();
}
```

method calls

```
*******
*     *
*     *
*******
```

```
private static void horizontal()
{ out.println("*******"); }

private static void vertical()
{ out.println("*     *"); }
```

method defini-tions

Methods in the Java language

- A *method call* statement invokes the method and may pass parameters to it in parentheses
- Many method calls follow an object or class name, and a dot, as a message to the object
- A *method definition* spells out the method's executable code and declares local variables
- A method definition has
  - a *header* (access specifier; type; method ID; parameters in parentheses) and
  - a *body* (block or compound statement)

Java static methods

- *Definition:* a method that is not called by a message to an object
- *Examples: sum*, above
- Static methods don't access fields of their classes
- The *Math* class is a utility that has static methods that implement functions

Static method example

```
public class HelloApp
{
  static void hello()
  {
    System.out.println("Hello");
  }

  public static void
      main(String[] args)
  {
    hello();
  }
```

Class name, part of class definition

method definitions

method call

Some Java code in these slides not yet tested

Method design

- *Goal:* cohesion; give any method a single clear responsibility
- *Goal:* weak coupling – make methods independent of each other
- *main* should briefly outline the entire program
- Let a method handle data at the lowest possible level or scope

2. Local variables, parameters, and return values

- How do methods communicate?

Subtopic objectives

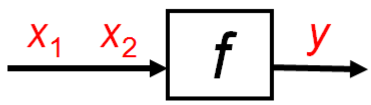
1.2a Explain method signatures and scope*

1.2b Write a method with parameters and return values**

1.2c Debug a method†

A method's ways to communicate

- A *parameter* is a value passed to a method by the method's call
- A *return value* is passed from a method to the statement that calls it



- A method signature specifies the names and types of parameters, and type of return value

Local variables

```
int quantity = 2;

public static
  void add()
{
  int sum = 2 * quantity;
  out.println("sum = " + sum);
}
```

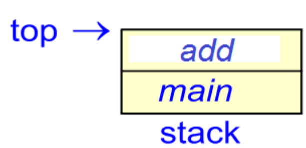
Class member (instance variable): accessible to all methods in class

Local variable: accessible only in this method

A *local* variable declared inside a method is inaccessible from outside

Local variables and scope of access

- A variable declared within a compound statement is usable only there.
- When a method is called, an *activation record* for the call, containing local variables, is placed on top of the stack.
- When the method terminates, the stack is popped and local variables are deallocated.
- Two activation records:



Parameters: example

```
public static void main()
{
  out.print("2 + 5 = ");
  display_sum(2,5);
}

private static
  void display_sum(int a, int b)
{
  out.print(a + b);
}
```

Actual parameters

Formal parameters

Output:
`2 + 5 = 7`

A *return* statement passes a value back to the calling method

- The *return* keyword precedes the returned value in the called method
- The return value's type must be type compatible with the method's type, declared in header
- The *return* statement terminates the method call
- The returned value goes on the stack for retrieval by the calling method
- The method call is an expression whose value is the value returned

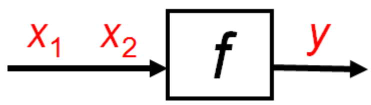
Passing objects as parameters

- Primitive type items are passed by value
- Objects are passed by reference; i.e., the address of the object is passed
- Hence a method may change the state of an object
- *Example:*

```
void capitalize(String s)
{
   s = s.charAt(0) + s.substring(1);
} // changes s
```


A method's ways to communicate

- A *parameter* is a value passed to a method by the method's call
- A *return value* is passed from a method to the statement that calls it
- A *local* variable declared inside a method is inaccessible from outside



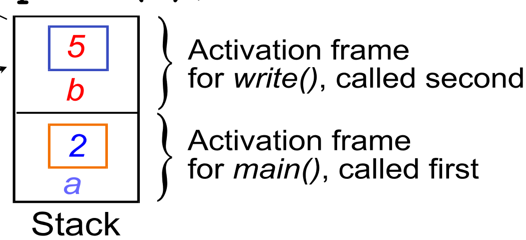
Memory allocation for local variables

When method terminates, activation frame is popped from stack

```
public static
  void main()
{
  int a = 2;
  write(5);
}
public static
void
 write(int b)
{
  out.print(b);
}
```

This method call pushes an activation frame onto stack

Stack pointer points here, to top of stack



How local data is stored

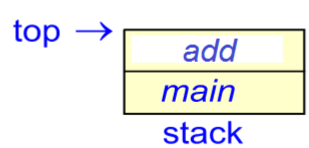
- The stack stores parameters and local variables in *activation records*
- Each call to a method is recorded as an activation record
- When the method terminates, the activation record is popped from the stack; all local data disappears from memory

Activation records in memory

- Each function call, at run time, causes an activation record to be pushed on top of the stack
- When the function terminates, the activation record is popped and its memory is released
- Activation records contain local variables and function parameters
- A *return* statement pushes the return value on the stack after activation record is popped

Scope of access for variables

- A variable declared within a compound statement is usable only there
- When a method is called, an *activation record* for the call, containing local variables, is placed on top of the stack
- When the method terminates, the stack is popped and local variables are deallocated
- Two activation records:



Parameters: example

```
public static void main()
{
  out.print("2 + 5 = ");
  display_sum(2,5);
}
```

Actual parameters

```
public static
  void display_sum(int a, int b)
{
  out.print(a + b);
}
```

Formal parameters

Output:
`2 + 5 = 7`

Parameters

- A way for the calling method to pass data of any types and quantity to the called method
- Value of *actual parameter* in method call is copied to the *formal parameter* declared in called method's definition
- Parameter is local; is deallocated when method terminates
- Method definition must specify parameter names and types

Parameters act like local variables

```
public static void display_sum(int a, int b)
// Displays (a + b)
{
  out.print(a + "+" + b + "=");
  while (b-- > 0)
     a++;
  out.print(a);
}
public static void main()
{
  int a= in.nextInt(), b= in.nextInt();
  out.print("Enter two numbers");
  display_sum(a, b);
  display_sum(6, 3);
}
```


Return values: example

```
public static void
  main(String[] args)
{
  int age = input_age();
  out.print("You are " +
     age + " years old");
}

private static int
  input_age()
{
  out.print("Your age? ");
  return in.nextInt();
}
```

The value returned by a method is the value used in the **return** *statement in the method definition.*

A *return* statement passes a value back to the calling method

- The *return* keyword precedes the returned value in the called method
- The return value's type must be type compatible with the method's type, declared in header
- The *return* statement terminates the method call
- The returned value goes on the stack for retrieval by the calling method
- The method call is an expression whose value is the value returned

sum with return value

```
public static void main()
{
  out.print("Enter past and current: ");
  int past_due=in.nextInt(),
    current=in.nextInt();
  out.print("You owe "
       + sum(past_due, current));
}

public static int sum(int a, int b)
{
  return a + b;
}
```

Overloading

- More than one Java method may have the same name, if number or types of parameters differ
- *Examples:*

```
public static int sum(int a, int b)
  {
   return (a + b);
  }
public static int
  sum(int a, int b, int c)
  {
   return (a + b + c);
  }
```


Parameter and return-value types

```
public static double sum
   (double a, double b)
{
  return (a + b);
}
public static boolean is_even(int n)
{
  return (n % 2 == 0);
}
public static char nth_ch(String s, int n)
{
  return s.charAt(n);
}
```

Objects are passed by reference

- The name of an object is actually a *reference*, which is the RAM address of the object
- Whereas the state of a primitive-type parameter is not changed by a method, a field of an *object* parameter may be changed
- To return objects, return a copy using *clone*
- If a null reference is passed as a parameter, a *NullPointerException* is thrown

Reference parameter example

```
public static void main(String[] args)
{
    FileReader reader =
      new FileReader(”x.txt”);
    Scanner in = new Scanner(reader);
    int x1 = readFile(in);
}
public static int readFile(Scanner sc)
// This method advances the file
// scanner object to next file position
{
    return sc.nextInt();
}
```

Objects used as parameters are mutable

```
• Point p = new Point(1, 2);
  …
  public static void zero(Point p)
  {
     p.set(0,0); // changes state of p
  }
```

- This principle applies to arrays as well as single objects
- Exception: *String*

Methods return objects as references

- A string that is created by a method is deallocated when the method terminates
- Therefore if such a string is to be returned, we use the *clone* method to return a *copy* of it:

```
String s = "foo";
return s.clone();
```

- The same applies to all objects created by methods

Variable-length parameter lists (implicit array parameters)

- The parameter declaration *int ... x* creates an array at runtime
-
```
int sum(int . . . x)
{
   int y = 0;
   for (int i = 0; i < x.length; i++)
   {
       y += x[i];
   }
   return y;
}
```


Parameters to *main*

- May be passed from command line
- *args:* an array of strings
- the first command-line argument
- Applications: file names, switches

```
public static void main(String[] args)
{
  for (int i=0; i < args.length; ++i)
    out.print(args[i] + " ");
}
```

3. Documenting and testing methods

- What was said in Intro to Programming about documenting and testing?

Subtopic objectives

1.3 Explain method documentation and testing*

Suggestions for writing methods

- A method has a single purpose
- Its purpose is documented in a comment at the top
- Code longer than a page is usually broken down into method definitions
- Experienced programmers avoid *side effects* on variables declared outside the method

Using *Javadoc*

- This JDK tool reads a Java source file and creates an HTML documentation file using comments from the Java file
- It uses comments such as **`/** @param … */`** or **`/** @return… */`** to include parameter or return value documentation in the HTML file, where "…" is the programmer's description of the parameter or return value
- See the *Javadoc* tutorial online

Stubs test a top-down design

A stub method simply reports that it has been called

```
public static void main(String[] args)
{
 char option;
 do {
    out.print("1 Add\n 2 Sub\n 3 Quit");
    option=in.nextChar();
    switch (option) {
       case '1': add(); break;
       case '2': subtract(); break;
    }
 } while (option != '3');
} // [see stub.cpp]
```

Calls to stub methods

Stub method definitions

```
public static void add()
{
 out.print("Calling 'add'");
}
public static void subtract()
{
 out.print("Calling 'subtract'");
}
```

- Stubs are called by *driver* programs while being tested

Driver programs test methods

The driver below tests *sum:*

```
public static void main()
{
  int x1 = in.nextInt(),
      x2 = in.nextInt();
  out.print("x1 + x2 = " + sum(x1, x2));
}

public static int sum(int a, int b)
{
  return a + b;
}
```


Checking validity of data

- *Input validation* may enforce reasonable types or ranges of values
- *Examples:* Whole-number quantities; *age* between 0 and 120
- User-friendly input handling should allow user to re-enter input in case of error
- Similar checks should enforce method preconditions
- Methods may throw exceptions in cases like illegal argument or null pointer

Assertions and preconditions

- *Precondition:* a logical assertion about a value that must hold if code is to be able to do its job
- The *assert* statement throws an exception if its parameter is false; if a parameter is invalid, throw an *IllegalArgumentException*
- when testing program *x* use *java –ea x* to enable assertion testing
- *Example:*

```
public double tax(int income)
  throws IllegalArgumentException
{
  assert (income >= 0);
```


Debuggers and debugging

- Enable programmer to understand bugs by comparing variable values with what correct values are known to be
- Tools provided: breakpoints, single stepping, inspecting variables
- Available with BlueJ, JDK, Eclipse
- Standalone program JZSwat: *code.google.com/p/jswat*

Group exercise

Suppose you are to write an application around a menu with items *delete, replace, find, insert*

1. Give a module-hierarchy diagram of three levels (you will have to invent at least one reasonable module)
2. Document using Javaddocs tags
3. Write stubs and a driver for this program or module; test

4. Recursive methods

- What's induction?
- What's recursion?
- Have you discussed these in Precalc or CS I?
- Can a method call itself ?

Subtopic objectives

1.4 Derive a recursive method from a loop

A recursive factorial method

```
public static int factorial(int n)
{
   if (n <= 1)                          Base case
      return 1;
   else
      return n * factorial(n - 1);
}                                       Recursive call
```

A recursive method
(a) provides a direct solution for a simple *base case*, or
(b) calls itself to solve a simpler version of the problem it solves

Factorial with *while*

```
public static int factorial(int n)
{
   int i = 1, y = 1;
   while (i <= n)
      y = y * i;
   return y;
}
```

- The *iterative* loop here is equivalent to the *recursive* one on the previous slide

Recursion implements a loop

```
public static int input_age()
// Prompts for, returns age,
// repeats until gets valid input.
{
  out.print(“Age? “);
  int age = in.nextInt();
  if (age >= 0) return age;
  else return input_age();
}
```

Recursive method call

- A method that calls itself is recursive
- A *base case* (as, *age >= 0*) triggers a simple result; a recursive case triggers a recursive call
- Base case enables eventual termination

Integer to string conversion

Problem: design a method that converts an integer to a string

```
public static String toString(int n)
{
   if (n < 10)
      return "" + char((int)'0' + n);
   else
      return toString(n/10) +
         char('0' + (n % 10));
}
```

- [To be tested]

How recursion uses the stack

```
public static void main()
{
  backwards();
}

public static void backwards()
{
  char ch = in.nextChar();
  if (ch != '\n')
    backwards();
  out.print(ch);
}
```

Sample I/O:
Hello
olleH

Question: How can one *char* variable store the whole string?

Recursive definition: natural numbers

1. 0 is a natural number
2. Every natural number n has a unique successor, n', which is a natural number
3. All natural numbers follow (1) or (2)

- *Significance:* These assumptions give a logical basis to work with counting numbers.
- Computation is a formal way to manipulate numbers and objects represented by them.

---

*1. $0 \in N$; 2. $(\forall n \in \mathbf{N})\ n' \in N$; 3. $(\forall n \in \mathbf{N})\ n = 0 \vee (\exists m \in \mathbf{N})\ n = m'$

Recursive definitions of math operations

- 1 is shorthand for $0'$ (*successor* of 0), 2 for $0''$, etc.; n is *predecessor* of n'
- $(a + b)$ is shorthand for $sum(a, b) =$

$$\begin{cases} a & \text{if } b = 0 \\ sum(a', pred(b)) & \text{otherwise} \end{cases}$$

- Given the successor function, the addition function is computable using a loop
- *Significance:* Any finite repetitive process can be specified by inductive methods.

A recursive method to add

$$sum(a,b) = \begin{cases} b & \text{if } a = 0 \\ sum(a-1,\ b+1) & \text{otherwise} \end{cases}$$

```
public static int sum(int a, int b)
// Returns a + b. Recursive.
{
  if (a == 0)
     return b;
  else
     return sum(a-1, b+1);
}
```


Example: Σ (Sigma, summation)

- The summation operator Σ lets us add a series of numbers
- *Case:* $\sum_{k=1}^{n} k = \begin{cases} 1 & \text{if } n = 1 \\ n + \sum_{k=1}^{n-1} k & \text{otherwise} \end{cases}$
- *E.g.:* $\sum_{k=1}^{3} k = 1 + 2 + 3 = 6$
- *Generalizing to any function f:*

$$\sum_{k=1}^{n} f(k) = \begin{cases} f(1) & \text{if } n = 1 \\ f(n) + \sum_{k=1}^{n-1} f(k) & \text{otherwise} \end{cases}$$

- *Examples:* if $f(x) = 2$; if $f(x) = x$, etc.

5. Java file I/O

- Who has done file I/O?

Subtopic objectives

1.5a Explain streams and sequential file I/O*

1.5b Read a file using a loop**†

File streams

- A *file stream* is a sequence of characters moving to or from a storage device
- Java standard file manipulation classes: *Scanner, FileReader, PrintWriter*
- If a file cannot be opened, a *FileNotFoundException* is thrown at runtime, possibly generating an error message
- *Exceptions* are covered later in this course

File stream input

- To open file for input:
```
FileReader reader = new FileReader("x.txt");
Scanner fin = new Scanner(reader);
```
- The *FileReader* class defines a sequential text file stream and its constructor opens the named file
- To read integer from input file (see keyboard input):
```
x = fin.nextInt();
```
- To close input file: `fin.close();`
- Any *Scanner* method usable for keyboard input is also valid for file input
- *Example file:* `update.java`

File stream output

- To open file for output:
  `PrintWriter out = new PrintWriter("x.txt");`
- *Warning:* Opening a text file for output in this way erases any data previously stored under this name
- To write to file:
  `out.println("Hello");`
- To close output file:
  `out.close();`
- Any *System.out* method usable for screen output is also valid for file output
- *Example program:* **`update.java`**

File errors

- Attempting to open a file to read generates an exception (with possible *runtime error*) if file is not found
- The error is represented as an *exception object*
- Any method, such as *main*, that opens a file to read should be defined with "throws FileNotFoundException" in header
- Other file errors include attempting to read past end of file, attempting to read an item of the wrong data type

File-reading example

```
public class add // update.java
{
  public static void main(String[] args)
  {
   FileReader reader =
     new FileReader("x.txt");
   Scanner in = new Scanner(reader);
     int x1 = in.nextInt();
     int x2 = in.nextInt();
     int sum = x1 + x2;
     System.out.println("Sum is " + sum);
  }
}
```


Check file stream before reading

```
FileReader freadr = new
 FileReader("x.txt");
Scanner fin = new Scanner(freadr);
String line;
if (fin.hasNextLine())
 line = infile.getNextLine();
```

- Here, the *FileReader* object *fin* can detect the state of the stream
- Possible errors: file not found; file empty

A file-reading loop

```
public static void main(String[] args)
    throws FileNotFoundException
{
  System.out.println("Reading file");
  FileReader reader = new
    FileReader("Readfile.txt");
  Scanner fin = new Scanner(reader);
  while (fin.hasNextInt())
  {
    int x = fin.nextInt();
    System.out.print(x + "  ");
  }
  fin.close();
}
```


Passing a file object to a method

- Passing a *Scanner* object to a method enables error checking and reusability
- *Example:*

```
public static int readInt(Scanner sc)
{
   if (sc.hasNextInt())
     return sc.nextInt();
   else return (-1);
}
```

Opening files with path names

- A file may be opened even if in a different directory
- File name is specified using the entire path name
- Example: **`“c://cs1//myprog.java”`**

Scanning for patterns

- *Scanner method: useDelimiter*
- *Parameter:* regular expression, e.g., [a-zA-Z’] scans for any word, possibly with apostrophe
- *Caret* scans all characters *up to* specified delimiter, e.g., [^)]
- Token vs. line based file processing: Reges, pp. 407-409

Scanning a string

- A *Scanner* object may be associated with a string
- *Example:*
```
Scanner sc = new Scanner("1 2 3 4 5");
```
  enables use of *nextInt*() five times

The *File* class

- Alternative to *FileReader*, *PrintWriter*
- *Note:* any method that opens any file must be defined with
```
throws FileNotFoundException
```

References

Cay Horstmann. *Big Java*, 3rd ed. Wiley, 2008, Ch. 3.

D. Keil. Defining and using methods. Classroom handout.

D. Keil. Defining a class. Classroom handout.

S. Reges and M. Stepp. *Building Java Programs*. Pearson, 2014.