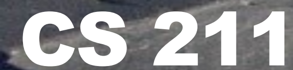


CS 211

File I/O (PrintWriter)
static
References
Scope, Encapsulation
Autoboxing/Wrappers

File Input/Output Basics

PrintWriter

Working with Files

- We can get input from files just as easily as from the keyboard, using a Scanner.
- We can write to a file as easily as to the terminal, using a PrintWriter.
- The file extension is arbitrary (.txt, .csv, .etc). The file just contains a sequence of characters that we can use however we choose.
- The One Hitch: Java requires us to deal with FileNotFoundException.
→ see examples next slides.

Reading Files

- We can get input from files just as easily as from the keyboard.

```
import java.util.Scanner;  //outside the class
...
try {
      Scanner sc = new Scanner (new File("outs.txt"));
      String s = "";
      while (sc.hasNextLine()){
        s += sc.nextLine()+"\n";
      }
      System.out.print("contents: \n"+s);
  } catch (FileNotFoundException e){
      System.out.println("file not present... :( ");
  }
```

Writing Files

We can write strings to files just as easily as the terminal.

- Make a PrintWriter, call print/println/printf. close it.

```
import java.io.PrintWriter;  // outside the class
...
try{
      PrintWriter pw = new PrintWriter(new File("outs.txt"));
      pw.print("writing a file from a program! :) \na\nb\nc");
      pw.close();
   } catch (FileNotFoundException e){
      System.out.println("file not found…   >:|");
   }
```

lots of method overloading! Look up the PrintWriter class.

Practice Problems

- Use a **PrintWriter** to write the numbers 1-100 to a file.
- Use a **Scanner** attached to that file to read in the numbers into an array; find the sum of them.
- Write a program that asks for a number, then calculates all the primes less than that number, writing them to **`primes_under_n.txt`**
  (where n is the number they gave you)

static

static keyword

static variable: one copy, always. It's part of the *class,* not part of objects.
→ no object is required/used to access it
→ sort of like a class-scoped global
→ called a *class variable*.

static method: callable without any object of its class. Again, it's part of the *class,* not part of objects.
→ Accessible without an object
→ thus *cannot use any non-static things in its class*.
→ these feel like what we called functions in Python.
→ called a *class method*.

static example

```
public class Trumpet {

    private static int nextSerialNum = 1;
    int serialNum, numValves;

    public Trumpet(int numValves){
        serialNum = nextSerialNum++;
        this.numValves = numValves;
    }

    public static int numBuilt(){
        return nextSerialNum;
    }
}
```

using static things

- **via the class**: use the class name to get to the correct scope.
  `classname . staticthing`

- **inside the class**: just directly use the member's name
  `staticthing`

- **unnecessary use of an object**: an object of the same class can be used to access it, though it's misleading.
  `objectExpr . staticthing`

static example – Math class

```
// idealized portion of the Math class

public class Math {
    public static final double PI = 3.14159;

    public static double sqrt(double a) { …}

        …
}
```

Quick Distinction

- **final**: definition can't change
- **static**: can use without instance of the class

→ **Math.PI** is both of these! Know both terms.

Pytania Poll

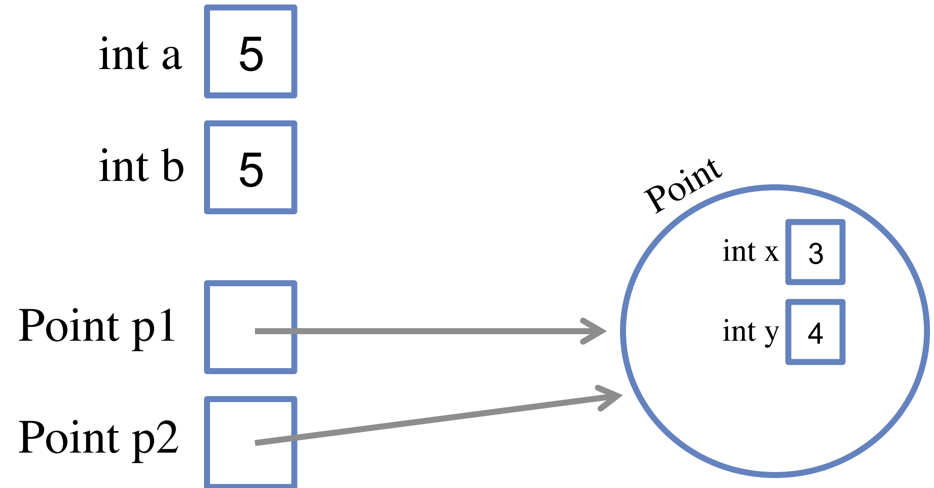
- static

References

Primitive vs Reference types

- each variable is a location that can store one value of its type
- assignment ***always*** just copies some value into that location
  - primitives: a copy of the primitive value
  - reference types: a copy of the arrow(reference). ***causes aliasing.***

```
int a = 5;
int b = a;
Point p1 = new Point(3,4);
Point p2 = p1;
```



new keyword

- The **new** keyword is the only thing that creates objects!
  - arrays: `new int[4]`
  - classes: `new Point(3,4)`

- mentally visualize memory: variables, references, objects.
  - aliasing: multiple references to the same object
  - variables can't point to each other! only to objects
  - reference-variable expression: simplifies to (a copy of) the reference

Scope

local variables

- parameters & variables declared in method are **local variables**.
- only exist while executing that method's code
- method's locals are all discarded upon `return`
- calling method: feeds *copies* of each argument.

- parameters of reference types:
  - given reference is often an alias!
  - reassigning parameter to other (new?) object breaks aliasing.
    Can't change external variable's *reference*!

scope example #1

What is printed?

```
public class Test {
    public static void main(String[] args) {
        int x=3;
        changeVal(x,5);
        System.out.println(x);

    }

    public static void changeVal(int p, int v) {
        p = v;
    }

}
```

scope example #2

What is printed?

```
public class Test {
    public static void main(String[] args) {

        Person p = new Person("Mason",21);
        changeName(p,"Thomas");
        System.out.println(p.name);

    }

    public static void changeName(Person p, String n) {
        p.name = n;
    }

}
```

scope example #3

What is printed?

```
public class Test {
    public static void main(String[] args) {
        Person p = new Person("Mason",21);
        changeName(p,"Thomas");
        System.out.println(p.name);
    }
    public static void changeName(Person p, String n) {
        p = new Person(n,30);
    }
}
```

Terminology

Field : a variable declared directly inside a class.

- static: one copy for all
- non-static (instance variable): one copy per object(instance)

Method: a method declared inside a class.

- static: callable without object; only accesses other static members
- non-static (instance methods): object required to call (because it may use instance variables).

Member: **any field or method**. Similar issues of visibility make it convenient to group them together under one term.

visibility

- fully accessible fields/methods in a class: easy to abuse/mess up!

```
bankAccount.balance = 10000000;
```

- **visibility modifiers**: restrict access to fields based on usage site.
  - **public**: always accessible from anywhere
  - **private**: only accessible by code inside this object's class
  - ***<package default>***: accessible in the package, not outside.
  - **protected**: accessible in the package and in child classes

Visibility Modifiers in Java

Modifier	Class	Package	Subclass	World
public	yes	yes	yes	yes
protected	yes	yes	yes	no
<package>	yes	yes	no	no
private	yes	no	no	no

Class Scope versus Local Scope

- members have **class scope**. They may be used anywhere in the class, or outside (visibility permitting)

- variables declared inside a method (including parameters!) have **local scope**. They only exist during the method call, and thus have no choices in visibility.

private visibility example

```
public class Trumpet {
    private static int nextSerialNum = 1;
    private int serialNum, numValves;
    public Trumpet(int numValves){
        serialNum = nextSerialNum++;
        this.numValves = numValves;
    }
    public static int numBuilt(){ // restore reading privileges
        return nextSerialNum;
    }
    public int getNumValves(){     // restore reading privileges
        return numValves;
    }
}
```

using private

- private fields: for internal use only
- public fields: read/write available everywhere

- public methods: can use private things in class!

  → outsiders' only ways of using private things is
  by provided public methods.

encapsulation/abstraction

- We can take one of two views of an object:
  - ***internal*** - members know details of each other
  - ***external*** - visible members are the only way outsiders may use this object.

- outsiders see an encapsulated entity that only exposes what interface it wants the outside world to see

empowered objects!

- an object should be ***self-governing***

- outsiders ("clients") request actions/modifications by calling methods. Implementation details are hidden inside the methods' code.

- We should make it difficult, if not impossible, for a client to access an object's variables directly

- Java enforces this with visibility modifiers

Visibility Modifiers

- public variables violate encapsulation

→ others can change values without object's permission

→ instance variables shouldn't be public

- It is acceptable to give a `CONSTANT` public visibility, which allows it to be used outside of the class

→ Public constants okay: client can access it, can't change it. (more convenient than 'getter' method).

Method Visibilities

- public methods are intended for clients.
- helper methods should be private.

- Methods to restore reading or writing privileges to restricted fields are called **getters** and **setters**, or more formally, **accessors** and **mutators**.

- restricts the client's ability to modify object's state: only way to change a variable's value is to run code the object already chose to make accessible

Pytania Poll

- scope and visibility

Packages

packages

- group related java files together
  → use `package` statements in each file, too
  → packages can also be placed in other packages.
- provide visibility boundary (dis/allow access from outside)
- must import any code that's defined in other packages
  → or, give fully-qualified name *every single time…*
- file name matches class name to help Java find definitions

Example Packages

package	purpose	examples
java	Java's top-level package	<many sub-packages>
java.util	useful data structures and classes	Scanner, ArrayList, …
java.io	file/resource interactions	File, IOException, …
java.util.stream	recent additions like	Stream, Collector, …
java.lang	core functionality	String, Math, …

java.lang.*   is always implicitly imported!

The import Declaration

- use something's *fully qualified name*, always:

```
java.util.Scanner myScanner = new
java.util.Scanner(System.in);
```

- **import** the class, and use just the class name

```
import java.util.Scanner;                    //outside of class
...
Scanner myScanner = new Scanner(System.in); // inside the class
```

- To import all classes in a particular package:

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

Importing classes

Example:

- you have `Assert.class` in the jar-file, junit-cs211.jar
- further, it's inside the org/ folder, and in the junit/ folder
- The Assert class is in the package **`org.junit`**
- You set your path to include the jar file, e.g.

```
-cp .:junit-4.12.jar
```

- You can then import the Assert class

```
import org.junit.Assert;
```

Packages Example

→ Look at the packages code example.

Pytania Poll

- **packages**

Wrapper classes autoboxing

Wrapper Classes

Each primitive type has a corresponding class. Examples:

- int vs Integer
- double vs Double
- char vs Character

provides (immutable) object version. related definitions go here. Examples:

- Integer.MAX_VALUE
- Integer.parseInt(String input)

- Java freely converts between them (almost) whenever you need.
- (later): sometimes we need reference types; these help us out.

wrapper classes example

```
int count = 5;
Integer quantity = count;          // conversion performed
count = quantity;                  // conversion performed
quantity = null;
count = quantity;                  // the only thing that can go wrong
```