Manchester Metropolitan University

Programming

Week 24 – Wrapping up

Learning Objectives

- This Week
  - Access control modifiers and data hiding (**public, protected, private**)
  - Packages and namespaces (**package, import**)
  - Next steps in programming
    - Suggestions for things you can do to take if from here…

Puzzle Solution

```
StringBuffer text = new StringBuffer('x');
System.out.println("length: "+text.length()+" contents: "+text);
```

Output:

```
length: 0 contents:
```

The key to this is we passed a single **char** ('x') to the **StringBuffer** constructor (not a **String**, e.g."x"). **StringBuffer** has no constructor which takes a **char** argument, but it *does* have one which takes an **int** (the **int** argument sets the initial size to reserve for the buffer). Java silently casts the **char** to an **int**, reserves **(int)'x'** characters in the initial buffer, and creates an empty string!

Data Hiding

- Data hiding in object oriented programming essentially means not giving the outside world access to the internals of a class.
- Classes communicate through the interfaces they expose through their public methods.
- The 'inner workings' (data and methods) should be hidden.

Some reasons why this is a good idea:

- You can completely change how a class works internally, but as long as the interface remains the same, you won't break any other code that uses the class.
- Giving *direct* access to class data means external code can put an object into an internally inconsistent state – access through methods allows you to ensure internal consistency at all times.

(See Week 16 Lecture)

Access Modifiers

We have used (and described) these already, on **methods** and **fields**:

public – anyone can read/write the data, or call the method.

protected – access is limited to within the class, package and subclasses.

private – access is strictly within the current class, the field/method is hidden to subclasses (the data is there – it's inherited - and may be accessible through getters/setters, but can't be accessed directly).

default (no specifier) – access is within the same **package**. Also known as **package-private.**

Access Modifiers

Access to fields and methods

Modifier	Class	Package	Subclass	World
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	✗
`default`	✓	✓	✗	✗
`private`	✓	✗	✗	✗

Notice that subclasses **can't** access fields in the superclass with default access.

Question: so how have we been getting away with this so far?

(we'll return to this).

Class Access Modifiers

Modifiers can also be applied to class types (and interfaces, enumerations).

public – class visible to the world. A source file can only contain one public class.

default (no modifier) – class is visible only within its package.

protected and **private** can only be applied to 'inner classes' (classes defined within a class – not covered in this unit).

Packages

- Packages are Java's mechanism for defining **namespaces.**
- A **namespace** is a similar to the scope of a variable but refers to types (classes, interfaces, enumerations etc.) – the namespace is where the name is valid and the type can be used.
- Idea is to prevent conflicts between names – two libraries may define a class with the same name, but this is ok if the names are in different namespaces or packages.
- Membership of a package is declared using the **package** statement.
  - Must be the first line of the file (so only one package per file)
  - A package can be spread across multiple files – all have the same package statement in the first line.

Example

Animal.java

```java
package animals;

interface NoiseMaker
{
    void MakeNoise();
}

public class Animal implements NoiseMaker
{
    int legs = 0; // default value

    public void MakeNoise()
    {
        // do nothing
    }
    public String Description()
    {
        return "Generic Animal";
    }
}
```

Dog.java

```java
package animals;

public class Dog extends Animal
{
    public Dog()
    {
        legs = 4;
    }

    public void MakeNoise()
    {
        System.out.println("woof");
    }

    @Override
    public String Description()
    {
        return "Dog, a "+legs+"-legged animal";
    }
}
```

1. Package **animals** contains three types – what are they? Which can be used outside the package?
2. How is the class **Dog** able to access the field **legs**?

Example

Animal.java

```java
package animals;

interface NoiseMaker
{
    void MakeNoise();
}

public class Animal implements NoiseMaker
{
    int legs = 0; // default value

    public void MakeNoise()
    {
        // do nothing
    }
    public String Description()
    {
        return "Generic Animal";
    }
}
```

Dog.java

```java
package animals;

public class Dog extends Animal
{
    public Dog()
    {
        legs = 4;
    }

    public void MakeNoise()
    {
        System.out.println("woof");
    }

    @Override
    public String Description()
    {
        return "Dog, a "+legs+"-legged animal";
    }
}
```

1. Types: **NoiseMaker, Animal** and **Dog** – only **Animal** and **Dog** can be used outside the package.

2. **Dog** can access **legs** because even though this is hidden to subclasses, it is visible within the package.

Accessing Types

- Types from a package can be accessed using their **qualified name** – this is of the form **package.type.**

```
public class Main
{
    public static void main( String args[] )
    {
        animals.Dog dog = new animals.Dog();
        System.out.println(dog.Description());
    }
}
```

`animals.Dog` is **the qualified** class name.

- Use the **import** keyword to import a package, and access the types in the namespace without the package name.

```
import animals.*;

public class Main
{
    public static void main( String args[] )
    {
        Dog dog = new Dog();
        System.out.println(dog.Description());
    }
}
```

By importing the package, we can use the short name: **Dog**

Hierarchical Package Names

- Finally, note that packages can be named using a hierarchical naming scheme.
- Examples – **processing.core**, **java.awt.event**.
- Although these appear as though they are packages within packages, they are separate entities – for example importing **java.awt.*** will **not** import the types in **java.awt.event**.
- Can import an individual class from a package, e.g.

```
import java.util.ArrayList;
```

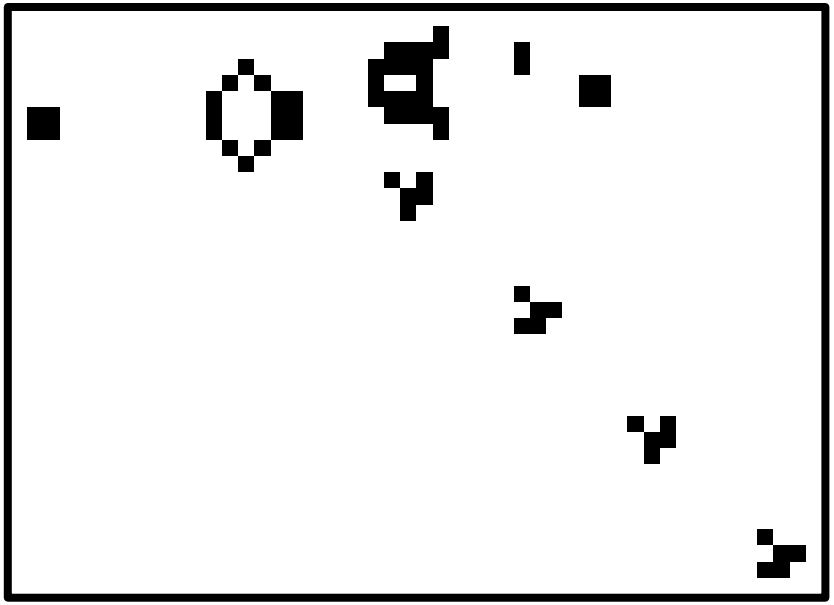
Next Steps

- You will continue programming next year (how much depends on your pathway and option choices).
- Also depending on your pathway, you may continue to more advanced Java or may use other languages (e.g. C#, C++).
- Skills are transferrable between languages – the key skill is problem-solving by ‘thinking like a programmer’.
- Object-oriented thinking and design also transfers between Java, C#, C++, and many other modern OO languages.
- The most important thing is – keep programming!
- Here are some things you could do over the summer…

Java Practice - 1

Try the Game of Life project on Moodle

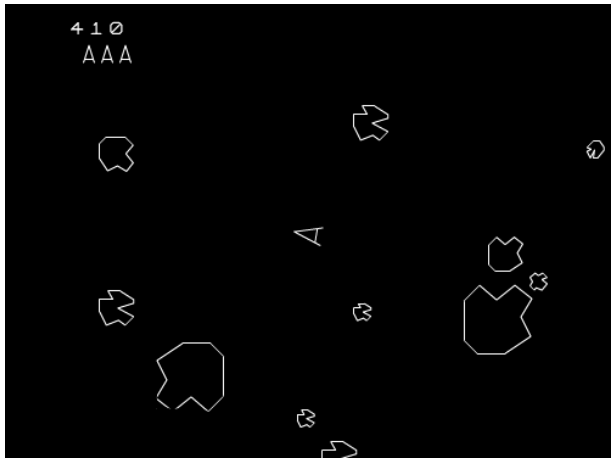
- Using the Processing classes to implement Conway's **Game of Life**.
- Life is a 2D **cellular automaton**, which produces complex behaviour from simple rules.
- This is a **Gosper's Glider Gun** – the Moodle project walks you through implementing this.



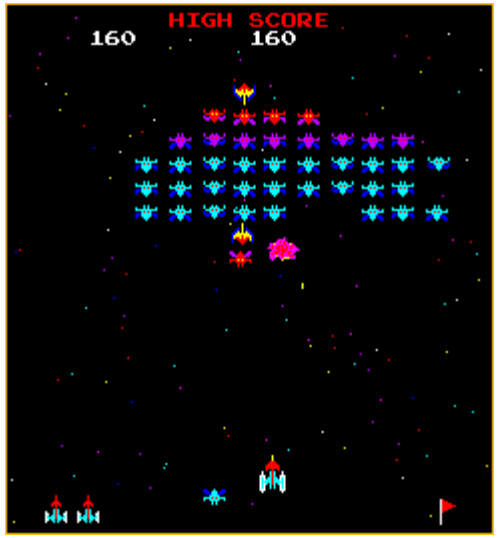
- See **Week 23** on Moodle for the PDF.

Java Practice - 2

Write another retro game – you've done one, now do another.

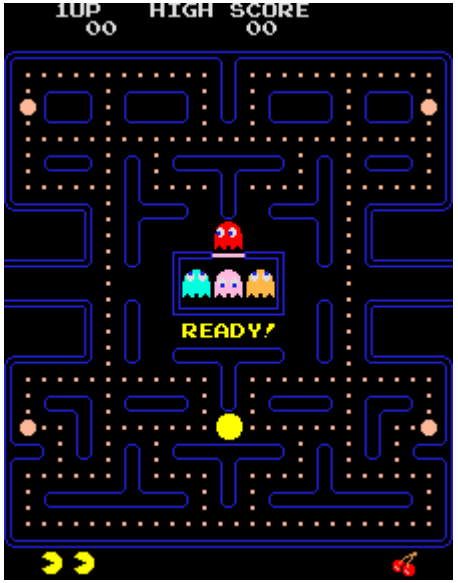


Asteroids (easy)

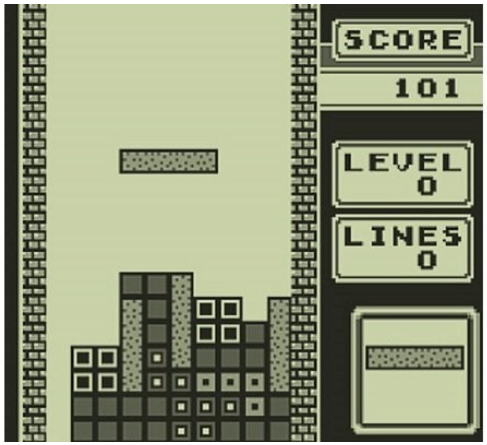


Galaxian (medium)

Scramble (medium)



Pacman (hard)



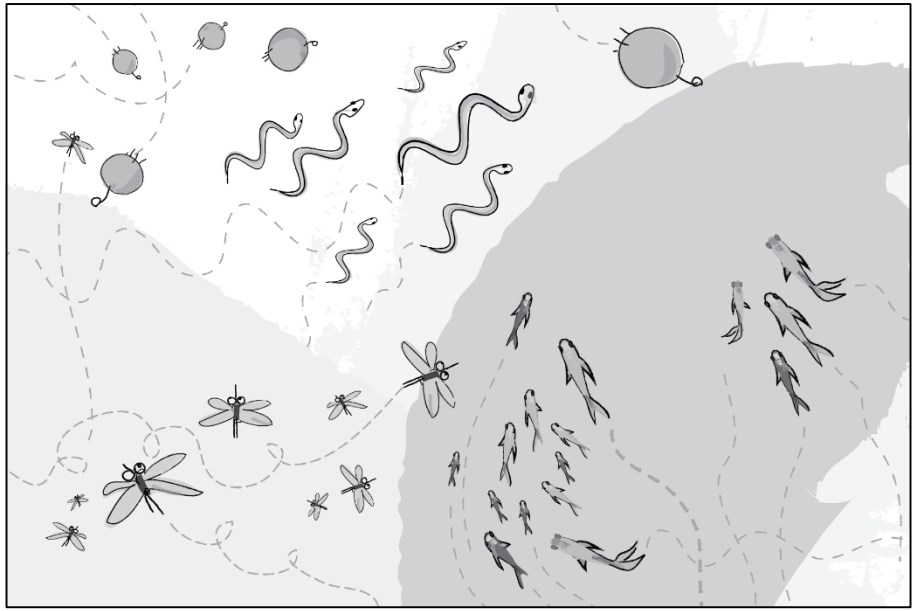
Tetris (hard)

Programming

Java Practice - 3

Work through "The Nature of Code".

- Uses processing.
- Projects on:
  - Physics Simulation
  - Particle systems
  - Autonomous Agents (AI)
  - Cellular Automata
  - Fractals
  - Genetic Algorithms
  - Neural Networks



Ecosystem Project (natureofcode.com)

See links on Moodle and www.natureofcode.com

Programming

Java Practice - 4

Spend some time on problem-solving sites:

1. Revisit codingbat.com
2. Join one of these sites (problem-solving, competitive coding, community, achievements, league tables)

www.hackerrank.com – good OO language-specific problems (not just algorithm/maths type problems).

www.spoj.com – "Sphere Online Judge" – huge community, thousands of problems.

www.codechef.com

All these sites will automatically check your code and mark it right or wrong. All support multiple languages (including Java).

Related Skills - 1

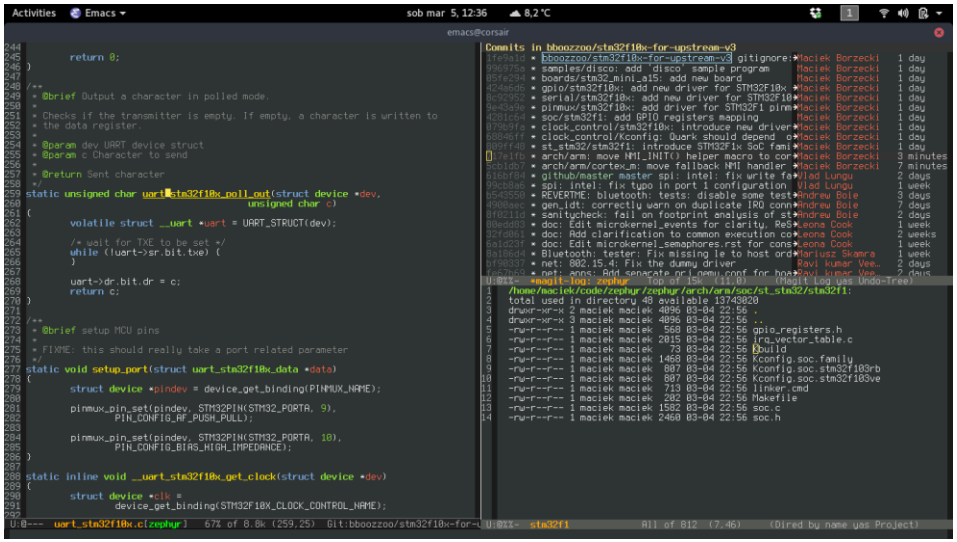
Learn a text editor

Vim – descended from vi (Unix editor)

Famously steep learning curve, huge productivity gains once mastered.

vim-adventures.com

(Zelda meets text editing)



Emacs – Written by Richard Stallman (founder of GNU and the Free Software Foundation).

Highly programmable (uses a dialect of Lisp). Open emacs and type “CTRL-h t” (tutorial).

Programming

Related Skills - 1

Learn a text editor

Modern editors:

Sublime Text – one of the best modern editors

- ‘Pesterware’ version is free.
- Great features: multiple cursors, helicopter view.

www.sublimetext.com

Atom –

- Free
- Github’s own editor

atom.io

Challenge:

Get to the stage where you never need to reach for the mouse…

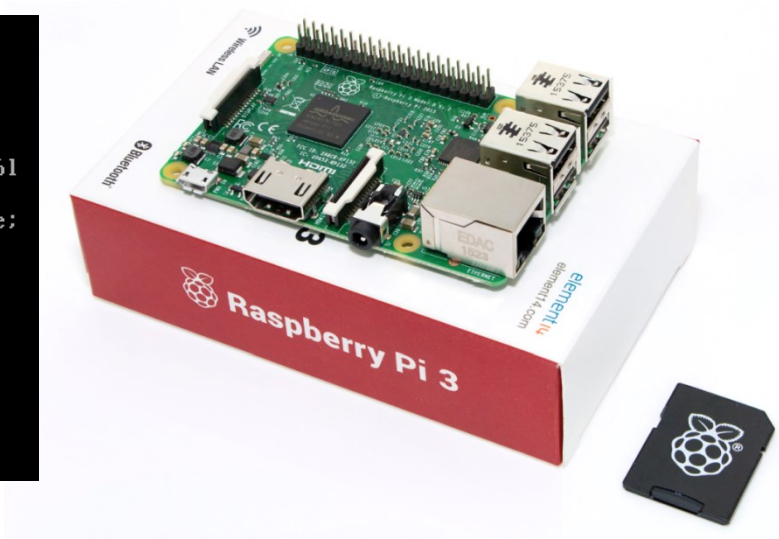
Related Skills - 2

- Learn to use the Unix command line (Mac or Linux).
- A great way in is to get a Raspberry Pi

https://www.raspberrypi.org/

- Lots of learning resources, projects to follow, large and active community.

```
Debian GNU/Linux wheezy/sid raspberrypi tty1

raspberrypi login: pi
Password:
Last login: Tue Aug 21 21:24:50 EDT 2012 on tty1
Linux raspberrypi 3.1.9+ #168 PREEMPT Sat Jul 14 18:56:31 BST 2012 armv6l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.

Type 'startx' to launch a graphical session

pi@raspberrypi ~ $
```



Programming

Related Skills - 3

- Learn to use a **version control** system.
- Open an account on www.bitbucket.org or github.com
- Both offer free private repositories for student accounts.
- Learn **git** – GUIs available but the command line is the way to go to understand the concepts.
  - Learning resources on bitbucket and github
  - Lynda.com videos:
    - "Up and running with git"
    - "Git essential training"
- Stop carrying code around on pen drives! You can't leave your bitbucket repo on a bus.

Branching out

Try another language

- High-level, object-oriented language.
- Huge library support – there's a library for everything.
- Used by sysadmins, security/forensics, data science, scripting language in Autodesk Maya, scientific computing.

Links:

"A Byte of Python" - http://python.swaroopch.com/

Lynda.com – "Up and running with Python"

Branching out

Try another language

Ruby

A Programmer's Best Friend

- Designed for "programmer productivity and fun".
- Lots of "syntactic sugar" – downside is lots of syntax.
- Used in web development (Ruby on Rails) and as a general scripting language.
- Pure OO language, but with good support for **functional programming** (the next big thing).

Try the quick tutorial:

https://www.ruby-lang.org/en/documentation/quickstart/

Branching out

Try another language

- Tiny language (implementation fits in about 1 MB).
- Small syntax, easy to learn.
- Uses:
  - Games – used as a scripting language in many game engines (many in-house engines, CryEngine, World of Warcraft).
  - Embedded systems – tiny footprint means it can be embedded into low-power devices e.g. **NodeMCU** is a Lua-based **Internet of Things** platform.

Links:

lua.org

https://www.lua.org/pil/contents.html - free book online

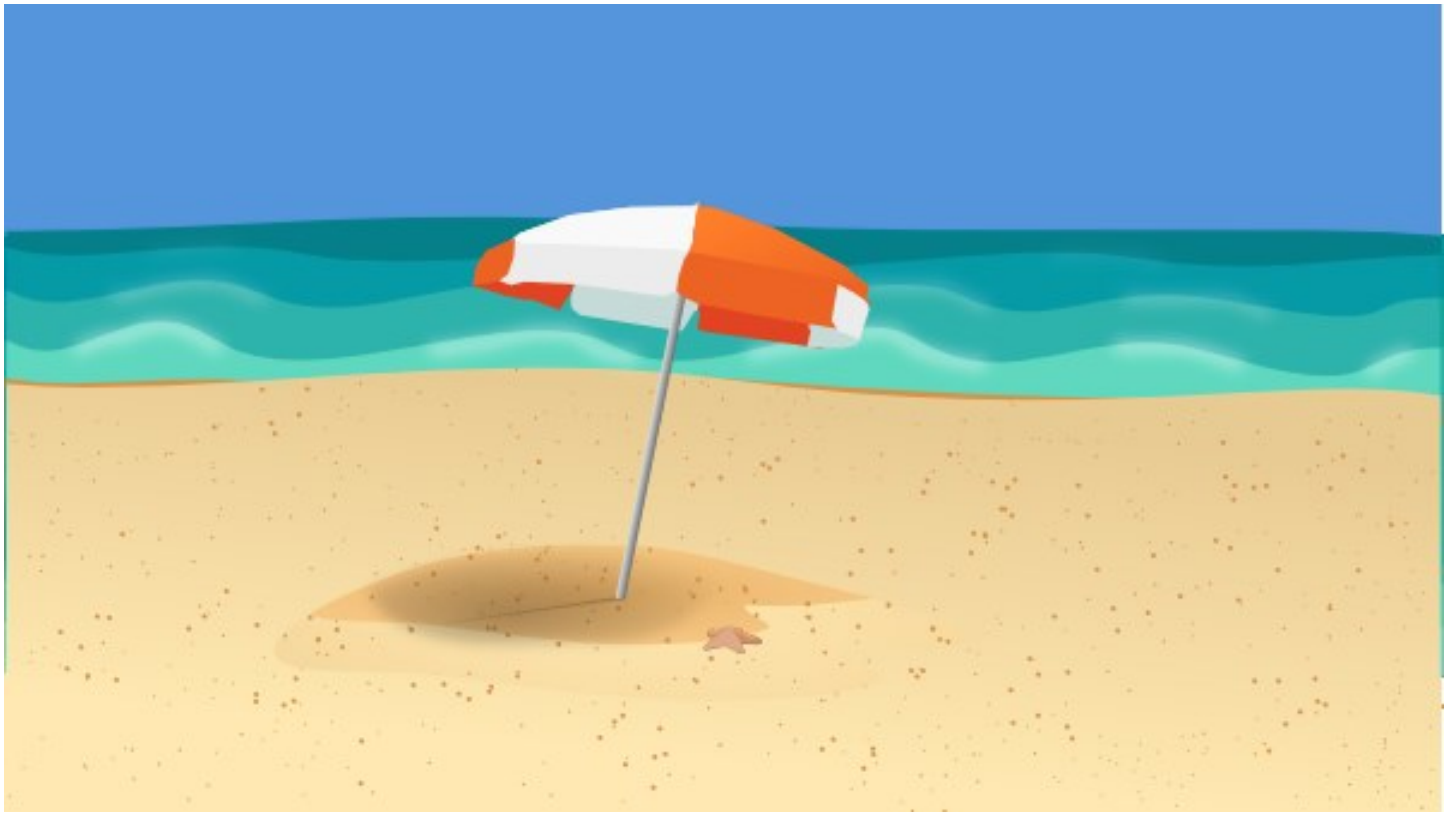
Branching out

- Try the languages out in the browser at www.repl.it
- Practice your new language:
  - If it's Python, try **codingbat.com** in Python.
  - Use your new language on spoj, hackerrank, codechef, especially to solve problems you've already cracked in Java.
  - Try some of the Java lab problems – this will give you a feel for the relative strengths of the languages in different areas.
  - Find a graphics API and write a retro game!

Manchester Metropolitan University

Programming

The End



Have a good summer, keep programming, have fun!