Object Oriented Software Engineering (COMP2003)

Lecture 1: Lists, Sets, Maps, and UML

Updated: 23rd February, 2018

Discipline of Computing
School of Electrical Engineering, Computing and Mathematical Sciences (EECMS)

Outline

Programming Languages

- OOSE is about software *design*.
- Design can be a highly theoretical subject.
  - i.e. lots of diagrams and written descriptions and, traditionally, very little actual code.
- To make it more practical, we'll connect it to *implementation*.
- But we still want to keep a broad perspective.
- Therefore, throughout OOSE, we'll look at *three* languages:

  C++ – an old, slightly ugly but widely-used language.
  Java – based on a subset of C++.
  Python – a new, fairly clean and rapidly evolving language.
- We *won't* cover all the features of these languages – only the basics, and whatever we need to understand OO design.
  - Programming Languages (COMP2007) covers the design *of languages themselves*.

Language References

- There isn't enough space in the lecture slides for a comprehensive introduction to C++ and Python.
- There is a language cheat sheet (on Blackboard) which gives a quick comparison between Java, C++ and Python.
- For further Python information, see the official Python documentation:

  Tutorial: docs.python.org/3/tutorial/
  API: docs.python.org/3/library/
- For further C++ information:
  - *Thinking in C++*, by Bruce Eckel (free versions available online).
  - cplusplus.com

Static vs Dynamic Typing

- This can be a matter of heated discussion!
- C, C++, C#, Java and others are *statically typed*.
  - They require you to specify datatypes.
  - You must specify the kinds of values that can be:
    - assigned to variables and fields,
    - passed to methods/functions, and
    - returned from methods/functions.
  - Acts as a safety measure, preventing certain kinds of datatype-related faults.
- Python (and Ruby, PHP, Javascript, etc.) are *dynamically typed*.
  - Datatypes exist, but they are not written explicitly.
  - Variables are not declared at all (in Python).
    - (Some other dynamically-typed languages do provide an *option* to declare variables, but not with a datatype.)
  - This can lead to shorter (and *possibly* more readable) code.

OO in Different Languages

- OO concepts are often very similar across languages.
- In Java, C++ and Python:
  - Classes have constructors, fields, methods, aggregation and inheritance.
  - Objects are constructed, have field *values*, and can be operated on by methods.
- However, some of the details are a bit different.
- For instance, in Python:
  - Everything is technically public.
    - "public", "private" and "protected" keywords do not exist.
    - If you want something to be "private", just don't access it outside the class.
  - Constructors are called "`__init__()`", and they are actually just ordinary methods. There can only be one per class.

Design Patterns

- Reusable solutions to common problems in software design.
- The most-discussed design patterns come from a single book:
  - *Design Patterns: Elements of Reusable Object-Oriented Software* (1994).
  - By the "gang of four": Enrich Gamma, Richard Helm, Ralph Johnson and John Vlissides.
- They analysed several large-scale software projects, and saw "patterns" in the way they were designed.
  - Certain design practices kept popping up in different places.
  - Different developers, solving different problems, had used very similar approaches.
- The gang of four investigated, isolated, and gave names to these practices.

Types of Patterns

- The gang of four identified software design patterns, but patterns exist in all human endeavours.
- There are *organisational* patterns – ways of arranging teams and projects.
- There are *idioms* – language-specific tactics for solving small, specific problems.
- We'll focus on design patterns, introducing a few different ones in most lectures.

Patterns in OOSE

- Why should we teach patterns?
- SE (at uni) often focuses on what you *should not* do.
- Sometimes it neglects to say what you *can* do.
- Often, specific patterns form a crucial part of the design of specific kinds of software.
- *More importantly*, learning patterns is a way of getting ideas.
    - Learning patterns will help develop your mental toolbox.
    - They reveal the possibilities in OO that might not be so obvious.
    - Even if when no pattern meets your exact needs, you can often adapt the principles to the situation at hand.
    - This adaptation of knowledge is the real skill.

The Unified Modelling Language (UML)

- UML is a way of showing software design.
- In its complete form, it is ridiculously complex. The UML 2.5 specification [1] is 794 pages long.
- It was once thought to be the Next Big Thing.
  - People tried to turn it into a diagrammatic programming language.
  - The idea was to auto-generate code from UML, and even write entire compilable programs in UML.
  - This failed, and many developers are now cynical about UML.
- UML is still useful *in moderation*.
  - Helps communicate and document high-level design concepts.
  - Natural language can be too clumsy, and code too detailed.

[1] http://www.omg.org/spec/UML/2.5

Detail in UML

- UML is (or should be) a human-level language.
  - You *can* show a lot of detail.
  - But you don't always *need* to.
- But *how much* detail should you show?
  - Should you show all the classes?
  - Should you show all the fields and methods?
  - Should you show all the method parameters, parameter types and return types?
- Depends on the situation.
  - What's important? What's trivial or irrelevant?
  - What do other people need to understand right now?
  - *When you draw a particular diagram, what are you trying to communicate?*
- In tests/assignments/exams, give enough detail to prove you know what you're doing (err on the side of caution).
- In the real world, don't waste people's time with irrelevant information.

Class Diagrams – Revision

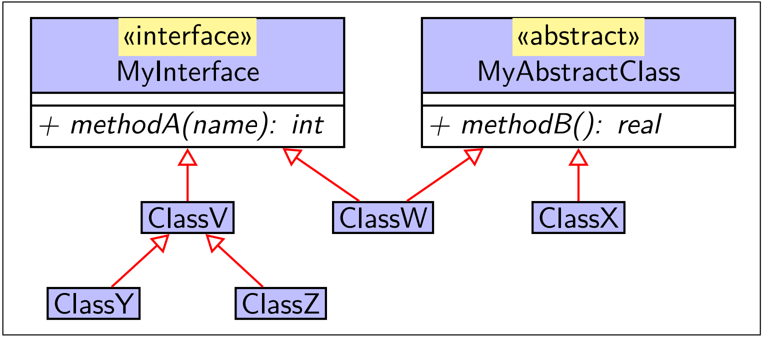
UML represents a class like this:

ClassName
– fieldA: int – fieldB: real – fieldC: string
+ methodA(name: string): int + methodB(): real <u>+ staticMethod()</u> + *abstractMethod()*

- Virtually everything here is optional.
- "–" means private, "+" public and "#" protected.
- Underlining means "static".
- Italics means "abstract" (although see next slide).

Interfaces, Abstract Classes and Inheritance in UML

- Use the *stereotypes* «interface» and «abstract» where needed.
- Use a hollow triangle ("◁——") to show inheritance.



- Technically a dashed line ("◁- - - -") should be used for interface inheritance.
  - But this is splitting hairs a bit. (I don't really care either way!)

Inheritance in Code

- You should know how this works in Java by now:

```
public class MyClass extends SuperClass      // Java
                     implements IntfA, IntfB {...}
```

- Python and C++ have *full multiple inheritance*.
  - No difference between “extending” and “implementing”.
  - Interfaces are just ordinary classes.

```
# Python
class MyClass(SuperClass, IntfA, IntfB):
    ...
```

```
// C++
class MyClass: public SuperClass,
               public IntfA,
               public IntfB {...}
```

Method Overriding in Code

- This should also be revision:

```
@Override                                    // Java
public int myMethod() {...}
```

  - @Override checks that the superclass has the same method.
- Python has no specific overriding syntax. Simply re-define a method in the subclass.
- In C++, you must uses virtual methods:

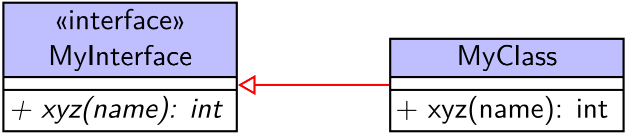
```
// In the superclass                            C++
virtual int myMethod();

// In the subclass
int myMethod();
```

  - virtual means a method *can* be overridden (not that it must be).

Method Overriding in UML

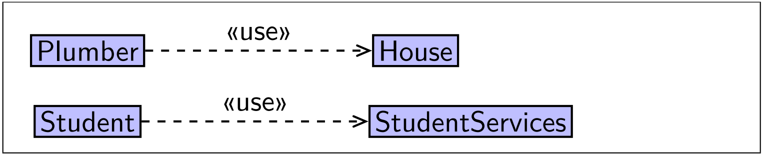
- UML has no specific overriding syntax either.
- However, given the following...



...we assume MyClass.xyz() overrides MyInterface.xyz().

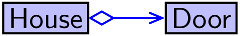
UML Usage Dependencies

- Represents short-term interactions between two classes:



- In code:
  - One class calls another class's method or constructor.
  - *But* there's no field – no lasting relationship between objects.
- There are also sub-types of usage dependencies; e.g.
  - «create» – object creation;
  - «call» – method call.
- Warning: usage dependencies can make your diagrams messy.
  - Only show the *important* ones (if any).
  - Omit them if there's already an association/aggregation.

Aggregation

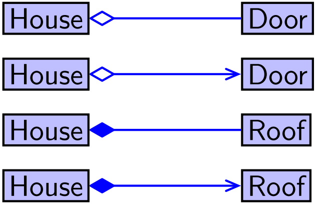



```
public class House
{
    private Door myDoor;
    ...
}
```

- "Aggregation" is a fancy word for a whole-part relationship between objects: one object "owns" another.
- In practice, one class has a field referring to another class.
- In Java and Python, this is a reference (since there are no other options).
- In C++:
  - You can do the same thing with a pointer, or smart pointer.
  - *Alternatively*, you can embed one object directly inside another.

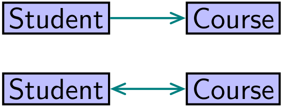
Aggregation in UML

- UML has a few different ways of showing this:



- These all mean that a House object owns a Door or Roof.
  - The diamond goes next to the “whole” object.
  - The arrow doesn’t really add information, but is useful to remind yourself which direction the aggregation goes in.
- The filled-in diamond (“◆⟶”) means “composition”.
  - A stricter form of aggregation.
  - The “part” object (Roof) cannot exist without the whole (House).
  - So, the objects’ creation and destruction is tied together.

Association (or Acquaintance)

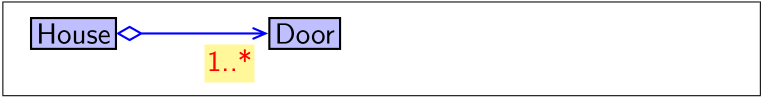



```
public class Student
{
    private Course course;
    ...
} // Same as for aggregation
```

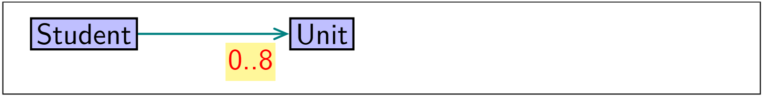
- Like aggregation, except the objects are on "equal terms".
  - Objects *do* have a lasting relationship.
  - But one *does not* "own" the other.
- In UML:
  - Omit the diamond, and keep the arrow.
  - The arrow indicates which object knows about the other (the same with aggregation).
  - Two-way associations are possible but rare.
- The code looks practically identical to aggregation.
  - The difference is *behavioural* – what one object *does* with the other one.

Multiplicity (1)

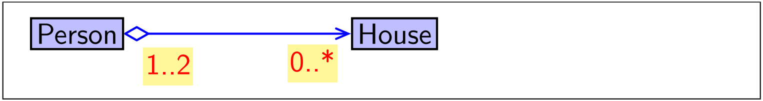
- How many objects are related?
  - Applies to aggregation, composition and association.
  - (*Not* inheritance or usage dependencies.)
- A house has *one or more* doors:



- A student is enrolled in up to 8 units:



- A person can own many houses, and a house can be owned by 1 or 2 people:

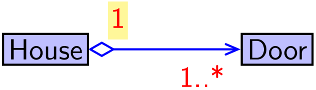


Multiplicity (2)

- In code, multiplicity means arrays, lists, and other "containers" that we'll discuss soon:

```
public class House
{
    private Door[] doorArray;    // OR
    private List<Door> doorList; // etc.
    ...
}
```

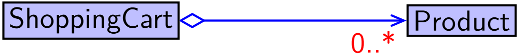
- Also, "reverse" (tail-end) multiplicity is possible:



  - Emphasises that each door is owned by exactly 1 house.
  - Often this is unimportant, and if so can be omitted.

Multiplicity: Accessors and Mutators (1)

- How do we set and get Product objects in a ShoppingCart?



- We can't do this anymore (in ShoppingCart):

```
public Product getProduct()         {...}
public void setProduct(Product p) {...}
```

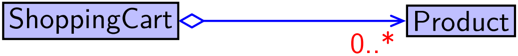
  - That doesn't make sense if there are multiple products.

- We need something more like this:

```
public int getNProducts()         {...}
public Product getProduct(int i) {...}
public void setProduct(int i, Product p) {...}
```

  - One getter for the *number* of products.
  - The other methods require an index parameter.

Multiplicity: Accessors and Mutators (2)



- But what about *adding* and *removing* products?

```
public void appendProduct(Product p) {...}
public void removeProduct(Product p) {...}
```

- Or should we do it by index?

```
public void insertProduct(int i, Product p} {...}
public void removeProduct(int i)            {...}
```

- Maybe someone wants to get or set *all* the products at once?

```
public Product[] getProducts()       {...}
public void setProducts(Product[] p) {...}
```

- Many options, but only write methods you need!

Containers

- But how to implement multiplicity?
- DSA covers arrays, linked lists, trees and hash tables.
- Many languages (Java, C++, Python and others), provide an abstraction layer over the top of these data structures.
- They give you different kinds of "containers":
  - Lists – ordered, linear sequences of values/objects.
  - Sets – non-repeating (possibly sorted) collections of values/objects.
  - Maps – a collection of objects accessible via a "key".
- In OOSE, we only care what they do (not how they work).
- (Note: In OOPD, "container class" means something quite different.)

Lists

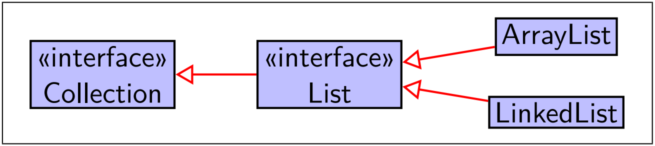
- Allows us to store multiple objects in a specific order.
- Based on arrays or linked lists.
  - Same functionality, different performance characteristics.
- Usual operations: append, insert, retrieve from a specific index, remove, get size, iterate.
- e.g.

```
public class ShoppingCart                                    // Java
{
    private List<Product> products;

    public ShoppingCart() {
        products = new LinkedList<Product>();
    }                  or ArrayList
    ...
}
```

Lists in Different Languages

Java – `ArrayList` and `LinkedList`:



Python – `list` and `tuple`.

- Both are array-based.
- `tuple` is immutable (read-only).

```
myList = ["zebra", "horse", "sheep"]
myList.append("buffalo")

myTuple = ("mint", "parsley")
```

C++ – `std::vector` (array-based) and `std::list` (linked-list-based).

Sets

- A collection of *unique* objects (no duplicates).
- Based on either hash tables or binary trees.
  - Tree-based sets are kept in sorted order.
  - Hash-based sets are unordered but faster.
- There are restrictions on the contents of a set. Classes must:
  - Be *immutable*.
  - Have a working **equals()** method.
  - For hash sets, have a working **hashCode()** method.
  - For tree sets, implement the **Comparable** interface.
  - Discussed later in more detail.

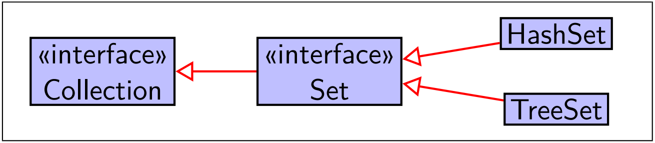
```
private Set<Product> products;                 // Java

public ShoppingCart()
{
    products = new HashSet<Product>();
}
```

or TreeSet

Sets in Different Languages

Java – HashSet and TreeSet:



Python – set (hash-based) and frozenset (hash-based, read-only).

```
mySet = {"zebra", "horse", "sheep"}
mySet.add("buffalo")
mySet.add("buffalo") # Does nothing
```

C++ – std::set (tree-based) and std::unordered_set (hash-based, but only in C++11 onwards).

Maps

- Sometimes called “dictionaries” or “associative arrays”.
- Extremely powerful!
- Contains *pairs* of objects:
  - Each pair consists of a “key” and a “value”.
  - You set and retrieve value objects by supplying the key.
  - Keys and values can be completely different classes.
- Based on either hash tables or binary trees (like sets).
  - The key objects together are effectively a set.
  - Tree-based maps are in sorted order (sorted by the keys).
  - Hash-based maps are unordered but faster.
- The restrictions on sets also apply to *map keys*:
  - The keys must be immutable, have an `equals()` method, etc.
  - The values have no restrictions, however.

Why Use Maps? (1)

Consider this code to search an array of products:

```
private Product[] products; // Array field
...
public Product findProduct(String name)
{
    int i = 0;
    Product foundProduct = null;
    while(i < products.length && foundProduct == null)
    {
        if(products[i].getName().equals(name))
        {
            foundProduct = products[i];
        }
        i++;
    }
    return foundProduct;
}
```

Why Use Maps? (2)

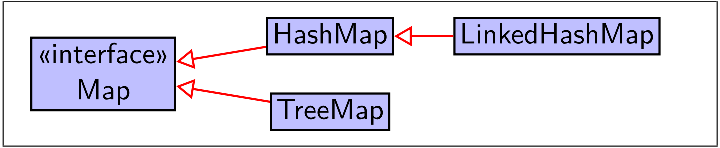
Now using a map to do the same thing:

```
private Map<String,Product> products; // Map field
...
public Product findProduct(String name)
{
    return products.get(name);
}
```

- The keys are the product names (`String`).
- The values are the `Product` objects.
- Obviously much shorter and simpler.
- And also much faster!
  - A hash map finds values in basically $O(1)$.
  - A tree map finds values in $O(\log n)$.
  - Iterating through an array is $O(n)$.

Maps in Different Languages

Java – HashSet and TreeSet (and a few others).



Python – dict ("dictionary") – a hash-based map.

```
myMap = {"Sam": "1987-11-13", "Alex": "1999-04-03"}
myMap["Amanda"] = "1993-01-20"
print(myMap["Sam"])
```

C++ – std::map (tree-based) and std::unordered_map (hash-based, but only in C++11 onwards).

Map Operations (1)

Here's what you can do with a map:

- Set a key-value pair:

```
myMap.put(key, value);  // Java
myMap[key] = value      # Python
myMap[key] = value;     // C++
```

  Overwrites an existing value if the key is already in the map.

- Retrieve a value:

```
value = myMap.get(key); // Java
value = myMap[key]      # Python
value = myMap[key];     // C++
```

  These do different things if the key is *not* in the map:
  - Java returns `null`.
  - Python throws an exception.
  - C++ adds a new entry, with a default-ly initialised value.
  - Python and C++ have other variants too.

Map Operations (2)

- Remove a key-value pair:

```
myMap.remove(key);                          // Java
del myMap[key]                              # Python
myMap.erase(key);                           // C++
```

- Check whether a key exists:

```
if(myMap.containsKey(key)) ...              // Java
if key in myMap: ...                        # Python
if(myMap.find(key) != myMap.end()) ...      // C++
```

- Obtain the number of key-value pairs:

```
size = myMap.size();                        // Java
size = len(myMap)                           # Python
size = myMap.size();                        // C++
```

Map Operations (3)

Iterate over all key-value pairs:

```
for(KeyClass key : myMap.keySet())                        // Java
{
    ValueClass value = myMap.get(key);
    ...
} // Replace 'KeyClass' and 'ValueClass' as appropriate
```

```
for key, value in myMap.items():                         # Python
    ...
```

```
for(auto& kv : myMap)          // C++ (version 11 and above)
{
    // 'kv.first' is the key, 'kv.second' is the value.
    ...
}
```

Immutable and Comparable Objects

Requirements for objects in sets, and used as map keys:

- They must be *immutable*, or at least not changed.
  - No mutators, OR you must not use them.
  - Maps & sets assume these objects never change.
- They must be comparable (**`equals()`** in Java).
- They must be *sortable* (for tree maps/sets).
  - If two objects are not equal, one must be "smaller" than the other. Which one?
- They must be *hashable* (for *hash* maps/set).
  - Each object must report its "hash code" (an integer).
  - Objects with different hash codes are definitely *not equal*.
  - Objects with the same hash code are *probably* equal.
    - Coincidence is possible, but rare.
    - Hash maps/sets then use **`equals()`** to confirm equality.
    - Actually much faster than *always* calling **`equals()`**.

Standard Classes

- Strings and integers satisfy all these requirements:

```
Map<String,Anything> myMap;
Set<Integer> mySet;
```

  Real numbers too, technically, but you may have problems with rounding errors.

- Containers can contain other containers:

```
List<List<List<Anything>>> myList;
Set<Map<String,String>> mySet;
Map<Set<String>,Set<String>> myMap;
Map<String,Map<String,Anything>> twoKeyMap;
```

  - However, containers have no natural ordering.
    - You can sort numbers, but how do you sort lists?
  - So, using tree-based containers may require some extra work.

Designing Classes for Hash-Sets/Maps in Java

```
public class Product                                       // Java
{
    private String name;
    private float price;
    ...
    @Override public boolean equals(Object other) {...}

    @Override             // If you need equals(), write
    public int hashCode() // hashCode() too (good practice).
    {
        return Objects.hash(name, price);
    }
}
```

The same fields that equals() looks at.

- Contract: equal objects must have the same hash code.
  - The same hash may also be shared by different objects.
  - But different hashes *guarantee* different objects.

...and for Tree-Sets/Maps in Java

```
public class Product implements Comparable<Product>  // Java
{
    private String name;
    private float price;
    ...
    @Override
    public int compareTo(Product other)
    {
        int result = name.compareTo(other.name);
        if(result == 0) // If names are equal, compare prices
        {
            result = (int)((price - other.price) * 1000.0);
        }
        return result;
    }   // Returns: 0 if equal, <0 if 'this' is smaller,
}       //                      >0 if 'this' is bigger.
```

- compareTo() returns 0 if and only if equals() returns true.

Hashing and Equality in Python

```
class Product:                                        # Python
    def __init__(self, name, price): # Constructor
        self.name = name
        self.price = price

    def __eq__(self, other):
        return self.name == other.name and \
               self.price == other.price  # Should really use
                                          # a tolerance here.
    def __hash__(self):
        return hash([name, price])
```

- Conceptually exactly the same as Java – just different syntax.
- In Python, "`x == y`" is the same as "`x.__eq__(y)`".
  - (Write "`x is y`" to check for reference equality.)

Object Ordering in Python 3

Not critical, as Python has no tree-based containers. But FYI...

```
import functools                                        # Python

@functools.total_ordering
class Product:
    def __init__(self, name, price): # Constructor
        self.name = name
        self.price = price

    def __eq__(self, o): ... # As before

    def __lt__(self, o):
        return (self.name, self.price) < (o.name, o.price)
        # 'tuple < tuple' will compare corresponding elements
```

- `__lt__()` returns true if 'self' is less than 'other'.
    - "`x < y`" is equivalent to "`x.__lt__(y)`".
- We could instead define `__le__()`, `__gt__()`, or `__ge__()`.

Equality in C++

Like Python, C++ uses == to check equality between objects.

```
class Product                          // Header file (product.h)
{
    private:
        std::string name;
        float price;
    public:
        ...
        bool operator==(const Product& other) const;
};
```

Like '__eq__()' in Python

```
#include "product.h"    // Implementation file (product.cpp)
...
bool Product::operator==(const Product& other) const
{
    return name == other.name && price == other.price;
}
```

Hashing in C++

- To make your own C++ class hashable, *brace yourself!*
  - I won't show you exactly what to do, because it's too bizarre and complex to properly get to grips with right now.
  - Also, only C++11 onwards implement hashing.
- C++ classes don't do hashing *themselves*. Instead, you:
  - Create *another* class...
  - Which defines the call operator "`operator()(...)`", making the object behave like a function...
  - Which calculates a hash for instances of your first class.
- And then you either:
  - Supply this function-like object to the set/map constructor, OR
  - For maximum C++ nerd points, you define it to be an "explicit specialisation" of the **`std::hash`** "template class".
    - Allows sets and maps to find it automatically.
- But *why?* I assume someone needed job security.
- Note: this isn't needed if you're just storing strings, ints, etc.

Object Ordering in C++

```
class Product                          // Header file (product.h)
{
    private:
        std::string name;
        float price;
    public:         // C++'s maps and sets use the < operator
        ...         // specifically to order objects.
        bool operator<(const Product& other) const;
};
```

Like '__lt__()' in Python

```
#include "product.h"    // Implementation file (product.cpp)
...
bool Product::operator<(const Product& other) const
{
    return (name < other.name) ||
        (name == other.name && price < other.price);
}
```

The Iterator Pattern

- In most lectures, we'll investigate a new pattern or three.
- There is one that you should already know (from DSA): Iterator.
- Iterators abstract away the container type. This means you can:
  - Get elements from a container without knowing how it works.
    - There are different kinds of iterators for `LinkedList` vs `TreeSet`, but all obey the same contract.
  - Change container types with minimal impact:
    - e.g. replace `HashSet` with `ArrayList`.
    - Much of your code will probably be unaffected.
- Being so successful, iterators are now permanently embedded in Java, C++, Python and other languages.
  - Even a language construct built around them – for-each loops.

Iterators in Java

- You can use iterators directly like this:

```
Iterator<Product> it = productContainer.iterator();
while(it.hasNext())
{
    Product p = it.next();
    ... // Do something with each product.
}                                          // Java
```

- However, the for-each loop handles the details for you:

```
for(Product p : productContainer)          // Java
{
    ...
}
```

- Where `productContainer` can be an `Iterable` (e.g. list or set) or an array.

Iterators in Python

Python is very similar in concept:

```
it = iter(productContainer)                    # Python
try:
    while True:
        product = it.__next__()
        ... # Do something with 'product'
except StopIteration:
    pass    # Finished
```

```
for product in productContainer:               # Python
    ...
```

- Iterators use an exception (`StopIteration`) to signal the end.
  - Not really an exceptional situation. Philosophically debatable?
- Python's for-each loop works on any class with an `__iter__()` method.

Iterators in C++11

- C++ code often co-opts the standard for loop:

```
auto end = products.end();                                  // C++11
for(auto it = products.begin(); it != end; it++)
{
    Product& p = *it;
    ...
} // In earlier C++ versions, replace 'auto' with
  // 'std::vector<Product>::iterator', etc.
```

  - **begin**() and **end**() each retrieve an iterator. We mostly use **end**() to check whether we're finished.
  - "auto" lets us omit the full messy data type.
  - "**it**++" moves forward, while "***it**" gets the current element.
    - Operator overloading (*not* pointer arithmetic).
- Since C++11, there is now a for-each loop:

```
for(Product& p : products) { ... }                          // C++11
```

Polymorphism

- Containers and iterators rely a great deal on polymorphism:
  - The ability to treat different kinds of objects in the same way.
  - In order words, you can swap one kind of object for another, and the rest of your code doesn't need to change.
  - Usually relies on inheritance, but there's more to it than *just* inheritance.
- We like it because it helps reduce coupling.
- Polymorphism will come up numerous times in OOSE.
  - It features in a lot of other design patterns.
  - We'll get started on it next lecture.
- That's all for now!