CIT 5940

Lists

Agenda

1. Abstract Data Types & Interfaces
2. The `Leaderboard` Problem
3. The List ADT
4. Implementing `Leaderboard` using `List.java`

CIT 5940

ADTs & Interfaces

> *[W]e consider not only the data structure but also the class of operations to be done on the data;* the design of computer representations depends on the desired function of the data as well as on its intrinsic properties. Indeed, an emphasis on function as well as form is basic to design problems in general.
> — Don Knuth, *The Art of Computer Programming, Volume 1*

Function vs. Form

The **function** of an type in Java is what you expect it to *do*. This is the set of methods that it can execute.

The **form** of a type in Java, or the set of "intrinsic properties", refers to the instance variables and method bodies for an implementation of that data type.

Communicating in Terms of Function

When describing [software] systems, we need to be able to work at different levels of abstractions.

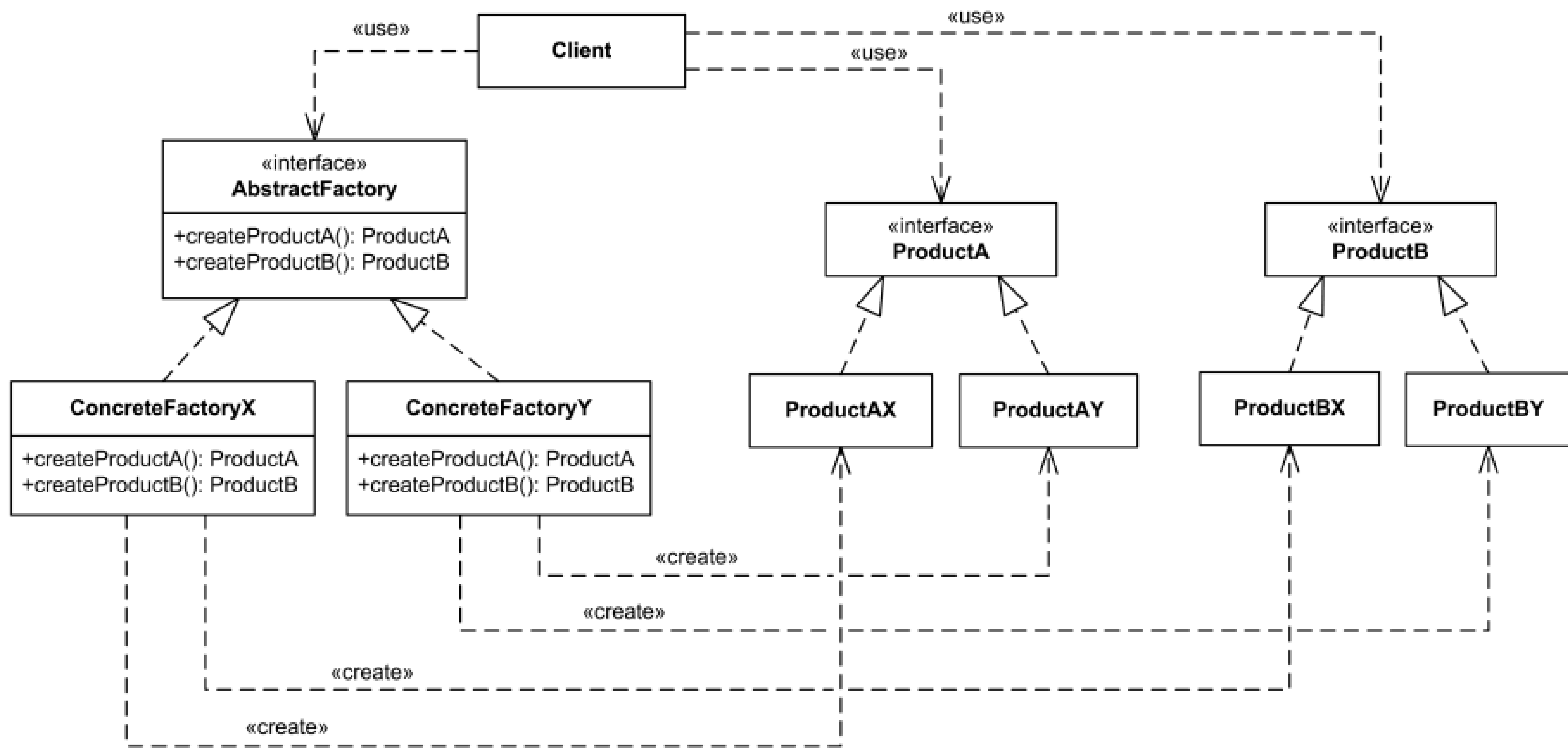
An **abstraction** is an intentionally incomplete representation of some entity that includes only the information that is relevant in a particular context with all other properties omitted.

➡️ Often the correct level of abstraction is to communicate in terms of *function only.*

Interfaces

Interfaces allow us to specify data types entirely based on their function—the list of abstract methods any implementing class must implement—without paying attention to the type's intrinsic properties.

The data type defined by an interface is an **abstract data type.**


«use»
Client
«use»
«use»
«interface»
AbstractFactory
+createProductA(): ProductA
+createProductB(): ProductB
«interface»
ProductA
«interface»
ProductB
ConcreteFactoryX
+createProductA(): ProductA
+createProductB(): ProductB
ConcreteFactoryY
+createProductA(): ProductA
+createProductB(): ProductB
ProductAX
ProductAY
ProductBX
ProductBY
«create»
«create»
«create»
«create»

Function & Form in Java: `Path` Example

The `Path` interface in Java specifies an abstract data type for **file paths**.

- A file path can be expressed as a `String`, but it abstractly represents a path through a filetree encoding ancestor/descendent relationships.
- The details of how a file path is specified differs among operating systems.

💡: Describe a `Path` based on what it does using an interface, and create different implementing classes to sort out how they will work on different operating systems.

The `Path` Interface

Tells you what you can do with something that is a `Path`. Tells you absolutely nothing about how the `Path` is implemented. And that's good!

```
public interface Path {
  Path getParent();
  int getNameCount();
  Path getName(int index);
  boolean endsWith(Path other);
  boolean endsWith(String other);
  boolean startsWith(Path other);
  boolean startsWith(String other);
  ...
}
```

```java
public interface Path {
  Path getParent(); // returns parent path or null if none exists
  int getNameCount();
  Path getName(int i); // returns the name of the stop on the path at index i
  boolean endsWith(Path other);
  boolean endsWith(String other);
  boolean startsWith(Path other);
  boolean startsWith(String other);
  ...
}
```

Given a directory, determine whether or not that directory is contained inside of the `"Documents"` directory.

```java
public static boolean isInDocuments(Path current) {}
```

Given a directory, determine whether or not that directory is contained inside of the `"Documents"` directory.

```
public static boolean isInDocuments(Path current) {
  Path parent = current.getParent();
  if (parent == null) {
    return false;
  }
  return parent.endsWith("Documents") || isInDocuments(parent);
}
```

You did not need to see inside a class implementing `Path` in order to do this!

Implementing Interfaces

To actually construct an object that has the type of `Path`, we need to define one or more classes that implement the `Path` interface.

```
public class WindowsPath implements Path {
  char separator = '\';
  public WindowsPath(String descriptor) { ... }
}
```

```
public class UnixPath implements Path {
  char separator = '/';
  public UnixPath(String descriptor) { ... }
}
```

Then, in the Application:

```java
public static void main(String[] args) {
  String targetPathSpec = args[0];
  Path targetPath;
  if (System.getProperty("os.name").equals("Windows")) {
    targetPath = new WindowsPath(targetPathSpec);
  } else {
    targetPath = new UnixPath(targetPathSpec);
  }
  boolean isPersonal = isInDocuments(targetPath);
  ...
}
```

Polls!

CIT 5940

The Leaderboard Problem

A Leaderboard

You're working on a Pennsylvania State Government that tracks quarterly CO2 emissions from many business receiving conditional subsidies. In a given quarter, a firm's rank on this emissions **leaderboard** determines the amount of a subsidy they receive.

Your job on the software team is to implement a class that can store an **ordered sequence** of business names where their position in the sequence encodes their relative ranks.

Polls!

What is it important that we be able to do with a `Leaderboard` *object?*

Understanding the Problem: Leaderboard Capabilities

- Gain access to the kth record of the `Leaderboard` to examine and/or to change the contents of its fields.
- Insert a new record just before or after the kth node.
- Delete the kth record.
- Determine the number of records in a `Leaderboard`.

c.f. Knuth, AOCP Vol. 1

Understanding the Problem: Other Issues

- Should the `Leaderboard` allow duplicates?
- Should the most polluting companies be at the "top" or the "bottom" of the `Leaderboard`?
- ...

These are important questions, but we won't really focus on them for this lecture.

Formalize the Interface

1. What **information properties** (instance variables) should a `Leaderboard` class store?
2. What **methods** should a `Leaderboard` support?

Polls!

*What **information properties** (instance variables) should a `Leaderboard` class store?*

Polls!

*What **methods** should a `Leaderboard` support? (Write a signature.)*

For Reference

This is one reasonable interface, but we'll go with whatever folks volunteer in lecture.

```
public interface ILeaderboard {
    public String get(int i);
    public void set(int i, String name);
    public int size();
    public void insert(int i, String name);
    public void addToEnd(String name);
    public void addToStart(String name);
    public void remove(int i);
    public void remove(String name);
}
```

Writing Tests

A unit test always consists of some:

1. input/start configuration
2. **expected behavior** that some operation should exhibit
3. **actual behavior** that gets exhibited
4. some assertion that compares the actual behavior to the expected behavior

Writing Tests: Example

Given the following representation of a `Leaderboard`...

Leaderboard
Harry's Coal Burning Bonanza
Travis' Professional Oil Spillers
Arvind's Green Gallery

...and I insert "Jérémie's Vegan Jerky" at position 2...

Writing Tests: Example

I would expect to see the following `Leaderboard`:

Leaderboard
Harry's Coal Burning Bonanza
Travis' Professional Oil Spillers
Jérémie's Vegan Jerky
Arvind's Green Gallery

Writing Tests: Think/Pair/Share

Pick at least one of the methods that we chose as part of the interface. Come up with a unit test or two for each method.

Doesn't have to be JUnit, but should have all components of a unit test.

CIT 5940

List ADT

A (Linear) List

A linear list is a sequence of $n \geq 0$ records $X_1, X_2, \ldots, X_n$ whose essential structural properties involve only the relative positions between items as they appear in a line.

- if $n > 0$, X_1 is the first record and X_n is the last
- if $1 < k < n$, the kth record X_k is preceded by X_{k-1} and followed by X_{k+1}.

c.f. Knuth, AOCP Vol. 1

A `List` is an **ordered** sequence of records. Given these relationships among the records, one could expect to be able to perform the following operations:

- *Gain access to the kth record of the list to examine/change the contents of its fields.*
- *Insert a new record just before or after the kth record.*
- *Delete the kth record.*
- Combine two or more linear lists into a single list.
- Split a linear list into two or more lists.
- Make a copy of a linear list.
- *Determine the number of records in a list.*
- Sort the records of the list
- Search the list for the occurrence of a record

Java's `List` ADT

Cartoon version:

```
public interface List<E> {
  public E get(int i);
  public E set(int i, E elem);
  public int size();
  public void add(int i, E e);
  public boolean add(E e);
  public E remove(int i);
  public boolean remove(Object o);
  // plus lots of other stuff...
}
```

Connecting the Dots

Looks like a lot of what we want a `Leaderboard` to be able to do is really just what a `List` is supposed to do.

CIT 5940

Leaderboard With List

(See code.)

CIT 5940

Leaderboard As List

Backing Up...

We're playing this game where sometimes we pretend we know what an `ArrayList` or `LinkedList` is and sometimes we don't.

In the previous section, we used an `ArrayList` to solve the problem. This shows us that there isn't really a problem to solve assuming we have access to the modern Java toolkit.

Going Forward...

In this section, we will refrain from using an `ArrayList` in order to get an appreciation for how they work. In this case, we'll think of a `Leaderboard` differently:

- **not** as an ADT
- as a specific instance (implementation) of the `List` ADT.

A `Leaderboard` With A Fixed Size

If we assume that we know the maximum number of elements included in a `Leaderboard`, then it would be reasonable to use an array to store the elements.

This is called **sequential allocation**.

Now: Implement

Next: Analyze