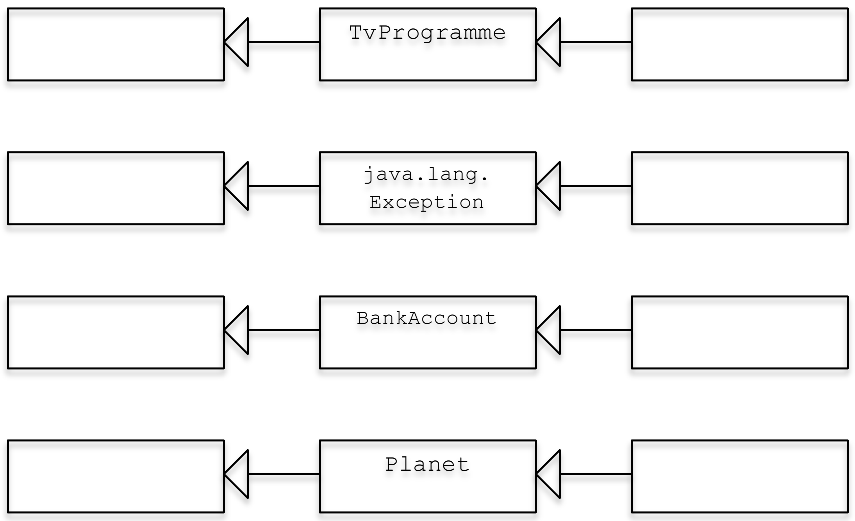
Java Programming 2 – Lecture #12 – Jeremy.Singer@glasgow.ac.uk

Object-Oriented Class Hierarchies

Below are some example class hierarchies, with most general on the left, and most specific on the right.



Motivating the need for Interfaces

A subclass specializes some feature of its superclass, as demonstrated above. However sometimes there are class features which run orthogonal to the inheritance hierarchy. For instance, `Human` and `Parrot` objects can both `speak()`, but in a typical inheritance hierarchy, they would not have a common superclass (other than `Vertebrate`, which does not have the `speak()` method since most other animals with a backbone are unable to talk).

The problem is, we want some classes *to inherit behaviour from multiple parent classes*. `Human` should be a subclass of both `Primate` and `TalkingCreature`. `Parrot` should be a subclass of both `Bird` and `TalkingCreature`.

The solution in Java[1] is to use *interfaces* to encapsulate these relationships that are orthogonal to the main inheritance hierarchy. An interface specifies a number of abstract methods (i.e. method signatures but no bodies). A class that `implements` an interface is obliged to provide an overriding method definition for the abstract methods inherited from the interface (unless the class is declared as abstract). Effectively, an interface is a form of *contract* that implementing classes must honour.

[1] More clunky solutions to this problem (e.g. C++) include *multiple inheritance*. More elegant solutions (e.g. Scala) include *traits* or `mixins`.

```
interface TalkingAnimal {
  void speak(String s);
}
```

```
public class Human extends Primate
                   implements TalkingAnimal {
  public void speak(String s) {
    // vocal chord vibrations…
  }
}
```

```
public class Parrot extends Bird
                    implements TalkingAnimal {
  public void speak(String s) {
    // stretch trachea and whistle…
  }
}
```

Note that a class may only extend one superclass, but it may implement many interfaces. Also, interfaces may extend other interfaces. Interfaces should only contain method signatures, which are implicitly `public` and `abstract`, and constant valued fields, which are explicitly `static` and `final`.

The Comparable Interface

The Java standard library includes an interface `java.lang.Comparable<T>`[2] which requires implementing classes to provide a single method `compareTo()`. This interface enables the correct behaviour of the generic `java.util.Collections.sort()`[3] method.

```
public class Country implements Comparable<Country> {
  String name;
  int population; // in millions
  public int compareTo(Country other) {
    return (this.population-other.population);
  }
}
```

```
ArrayList<Country> cl = new ArrayList<Country<>();
cl.add(new Country("USA", 300);
cl.add(new Country("Scotland", 5);
cl.add(new Country("China", 1300);
Collections.sort(cl);
```

[2] See http://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html
[3] See http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html