Static Classes

A class is static if all its fields and methods are static.

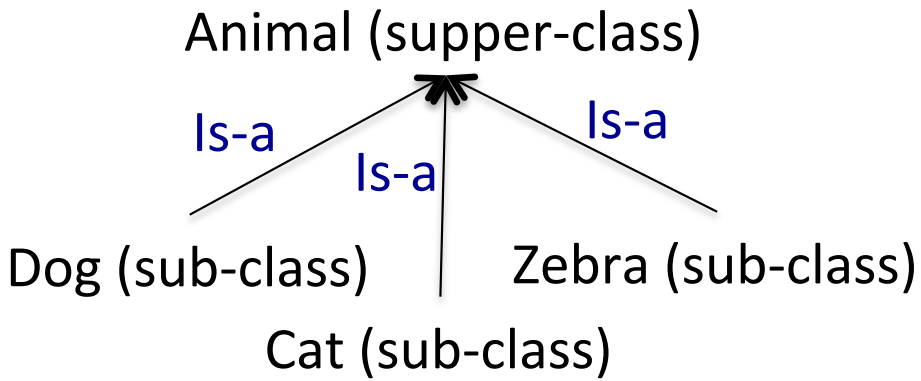
$\Rightarrow$ no point to creating objects of that class.

$\Rightarrow$ useful for grouping functions
(link Math.sin, Math.cos, ...)

Programms (in Java) are static classes!

Inheritance

Inheritance is is-a relation and nothing else.



- final classes cannot be sub classes.
- final methods cannot be over-written.

Inheritance - Cont.

```
class Animal {
 String name;
 public void makeNoise() {System.out.println("Noises...");}}

class Dog extends Animal {
String color;
public void makeNoise() {System.out.println("Roaring:waag!");}}

class Cat extends Animal {
public void makeNoise() {System.out.println("Roaring:miooo!");}}

class Zebra extends Animal {
String type;
}
```

Polymorphism and Interfaces

- *Polymorphism* is the provision of a single interface to entities of different types.
- A *polymorphic type* is a type whose operations can also be applied to values of some other type, or types
- Two kinds:
  - parametric: $\forall$ types
    - without mention of any specific type
    - *generic programming* in OOP; *polymorphism* in functional programming.
  - ad hoc: $\exists$ type which fits.
    - a function denoting different implementations depending on a limited range of individually specified types and combinations
    - 2 kinds: *overloading* & *subtyping*

Polymorphic References

- Which object a reference denotes during program execution cannot always be determined during compilation.
- A *polymorphic reference* is a reference that can refer to objects of different classes (types) at different times.
- A reference to a superclass is polymorphic, since it can refer to objects of all subclasses of the superclass during program execution.

```
Animal e;
Scanner keyboard = new Scanner(System.in);
// 1 = Dog, 2 = Cat, 3 = Zebra
int sel = keyboard.nextInt();
if (sel == 1) e = new Animal("A", "Black");
else if (sel == 2) e = new Animal("B")
else e = new Animal("C", "2");
```

The compiler cannot determine which object the reference e will denote after the if statement above.

Dynamic method lookup

- When a method is called for an object, it is the class of the object (i.e. object type) and the method signature that determines which method definition is executed.
- *Dynamic method lookup* is the process that determines which method definition a method signature denotes during execution, based on the class of the object.
- This process can lead to searching the inheritance hierarchy upwards to find the method definition.

Example- Ad hoc Polymorphism (Overloading)

```
...
public int Add(int x,int y) {
    return x + y;
}

public String Add(String s, String t) {
    return  Concat(s, t); // concatenation
}
...
```

Example- Parametric Polymorphism

defining a separate class for pairs of different types: a stupid design.
We use generics:
Note: T must be Object type
(another good reason for wrapper types)

```
class Pair<T> {
    T first;
    T second;
    Pair<T>(T x, T y) {this.first = x; this.second=y;}
}

Pair<int>, Pair<String>, ...
```

Example- Ad hoc Polymorphism (Subtyping)

```
abstract class Animal {
    abstract String makeNoise();
}
class Cat extends Animal {
    String makeNoise() { return "Meow!"; }
}
class Dog extends Animal {
    String makeNoise() { return "Woof!"; }
}

public class MyClass {
    public static void write(Animal a) {
        System.out.println(a.makeNoise());
    }
 public static void main() {
        write(new Cat());
        write(new Dog());
    }
}
```

Interfaces in Java

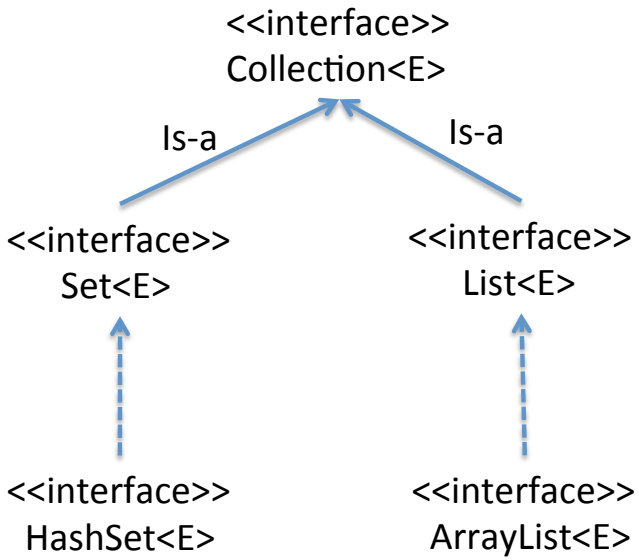
- An *interface* in Java defines a set of services but without providing an implementation.
- The interface contains a set of *abstract methods*.
- An abstract method is comprised of the method name, parameter list and return type - but no implementation.
- Example:

```
interface ISportsClubMember {
    double calculateFee();    // abstract method
}
```

- Classes that want to implement an interface have to specify this using the keyword *implements*, for instance:

```
class Student implements ISportsClubMember {
    double calculateFee() { return 100.0; }
    // ... other methods and fields
}
```

Collections



- A collection is a data structure that can keep track of references to other objects
- Java API defines several other types of collections in the java.util package.
- Central to the java.util package are a few important generic interfaces that collections implement.

Interface Collection<E>

Selected basic operations from the interface Collection <E>:	
int size()	Returns the number of items in the collection.
boolean isEmpty()	Find out if the collection is empty.
boolean contains(Object element)	Find out if element is included in the collection.
boolean add(E element)	Insertion; returns true if successful.
boolean remove(Object element)	Deletion; returns true if successful.
Iterator<E> iterator()	Returns an iterator

Bulk operations are performed on the entire collection.

Selected bulk operations from the interface Collection <E>	
boolean containsAll(Collection<?> s)	Subset.
boolean addAll(Collection<? extends E> s)	Union.
boolean retainAll(Collection<?> s)	Intersection.
boolean removeAll(Collection<?> s)	Difference.
void clear()	Deletes all items.

Traversing over a Collection

- Iterator - low level, use:

```
Collection<String> coll = new ArrayList<String>();
    coll.add("foo");
      . . .
Iterator<String> iter = coll.iterator();
    while(iter.hasNext()) {
       system.out.println(iter.next()); }
```

- Iterable - much better:

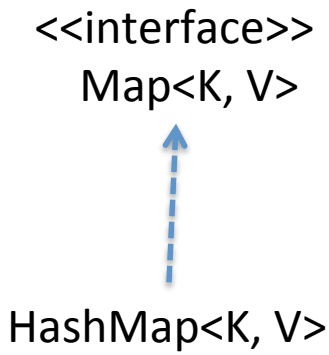
```
interface Iterable<E> {
  Iterator<E> iterator();
}
```

The for(:)-loop can be used to traverse a collection:

```
for (String str : collection) System.out.print(str);
```

Hash tables (Maps)

- A *hash table* is used to store entries.
- An *entry* is a pair of objects, called *keys* and *values*.
- The functionality of the hash tables is specified in the Map interface in the java.util package.
- The HashMap<K,V> class is a concrete implementation of the Map interface.



Basic Operations in the interface Map<K,V>

Selected basic operations from the interface Map<K,V>:	
int size()	Returns the number of entries in the hash table.
boolean isEmpty()	Find out if the hash table is empty.
V put(K key, V value)	Binds key to value and stores the entry.
V get(Object key)	Returns the value of the entry that is key.
V remove(Object key)	Delete the entry of the key and returns the value.
boolean containsKey(Object key)	Returns true if the key has an entry.
boolean containsValue(Object value)	Returns true if the value is included in some entry.

Map Views

Note: no iterator! To do this, we need to create a view.

- A *map view* is a collection that is associated with an underlying hash table.
- By means of such a view we can, for example, traverse the underlying hash table.
- The changes can be made through a view, and are reflected in the underlying hash table.

Selected view operations from the interface Map:	
public Set<K> keySet()	(Key View) Returns the Set-view of all the keys.
public Collection<V> values()	(Value View) Returns the Collection-view of all the values

Switch Statement

- The switch statement can be used to select one of several actions
- Syntax:

```
switch (expr) {
case e1: (s1;)?
break;
case e2: (s2;)?
break;
...
case en: (sn;)?
break;
(default: s;) ?
}
```

- Execution of the *break statement* in the loop body transfers the program control out of the loop.

Model-View-Controller (MVC)

- a *software pattern* for implementing user interfaces.
- divides a given software application into three interconnected parts:
  - *model* consists of application data, business rules, logic, and functions.
  - *view* can be any output representation of information.
  - *controller* accepts input and converts it to commands for the model or view.
- and defines interactions between them.
- An example: an MVC architecture from a web site (like amazon's)
  - Model (M): db schema
  - View (V): webpage (mobile, ...)
  - Controller (C): server-side code