Programming and Algorithms II

Lecture 8: Comparing and Strategy

Nicolas Wu

nicolas.wu@bristol.ac.uk

Department of Computer Science
University of Bristol

Comparables

Comparables

- Objects very often have a *natural ordering*: a way in which it makes sense for them to be compared
- There are some obvious examples, such as numbers:

```
3 < 10
```

Comparables

- Objects in a class can also be compared by some natural ordering: we might consider the weight of an animal

an ant called Dec

<

an elephant called Nellie

Comparables

- We could compare animals by providing a method: *weighsLessThan*

An **abstract class** allows us to have a generic implementation for all animals

```
abstract class Animal {
   public abstract int getWeight();
   public boolean weighsLessThan(Animal that) {
      return (this.getWeight() < that.getWeight());
   };
}
```

Comparables

- Now we can implement Ant and Elephant by simply returning some weight

```
class Ant extends Animal {
  @Override
  public int getWeight() { return 1; }
}

class Elephant extends Animal {
  @Override
  public int getWeight() { return 100000; }
}
```

Comparables

- To use this we can simply create some objects and compare them

```
Animal nellie = new Elephant();
Animal dec    = new Ant();

assert dec.weighsLessThan(nellie);
```

- This roughly corresponds to doing:

`dec < nellie`

Comparables

- In theory, we could use *weighsLessThan* to sort a list of animals, but it's not satisfactory

```
public boolean weighsLessThan(Animal that) { … }
```

Comparables

- In theory, we could use *weighsLessThan* to sort a list of animals, but it's not satisfactory

```
public boolean weighsLessThan(Animal that) { … }
```

Q. What is wrong with **weighsLessThan**?

Comparables

- In theory, we could use *weighsLessThan* to sort a list of animals, but it's not satisfactory

```
public boolean weighsLessThan(Animal that) { … }
```

Q. What is wrong with **weighsLessThan**?

A. It only works for animals: other classes need comparisons too

A. It only returns a boolean: implementing a sort is tedious? (actually, this isn't really problematic)

Comparables

- In theory, we could use *weighsLessThan* to sort a list of animals, but it's not satisfactory

```
public boolean weighsLessThan(Animal that) { … }
```

- A richer method is to return some measure of *difference* between the two objects

```
public int compareWeightTo(Animal that) {
  return (this.getWeight() – that.getWeight());
}
```


.compareWeight(

) = –99999;

Comparables

- But more generally we want to be able to compare other classes of objects too

```
public int compareSpeedTo(Vehicle that) { … }
```

Comparables

- But more generally we want to be able to compare other classes of objects too

```
public int compareSpeedTo(Vehicle that) { … }
```

Q. Why not just implement **compareXTo** for each class of things?

Comparables

- But more generally we want to be able to compare other classes of objects too

```
public int compareSpeedTo(Vehicle that) { … }
```

Q. Why not just implement **compareXTo** for each class of things?

A. A generic sorting algorithm needs to rely on a particular interface

Comparables

- We need a way of unifying all our comparators, so we could add *compareTo* to the *Object* class

```
public Object {
   …
   int compareTo(Object that)
   …
}
```

Comparables

- We need a way of unifying all our comparators, so we could add *compareTo* to the *Object* class

```
public Object {
   …
   int compareTo(Object that)
   …
}
```

Q. What's the problem with this approach?

Comparables

- We need a way of unifying all our comparators, so we could add *compareTo* to the *Object* class

```
public Object {
   …
   int compareTo(Object that)
   …
}
```

Q. What's the problem with this approach?

A. Not all metrics are the same: how would we compare speed and weight?

A. How would we even compare unrelated classes with no access to their implementations?

Comparables

- Instead of imposing a single *compareTo* that works on everything, we use a *generic interface*

```
public interface Comparable<T> {
  int compareTo(T that)
}
```

- This allows us to instantiate different classes as implementations of this interface

```
abstract class Animal implements Comparable<Animal> {
  public abstract int getWeight();
  public int compareTo(Animal that) {
    return (this.getWeight() - that.getWeight());
  };
}
```

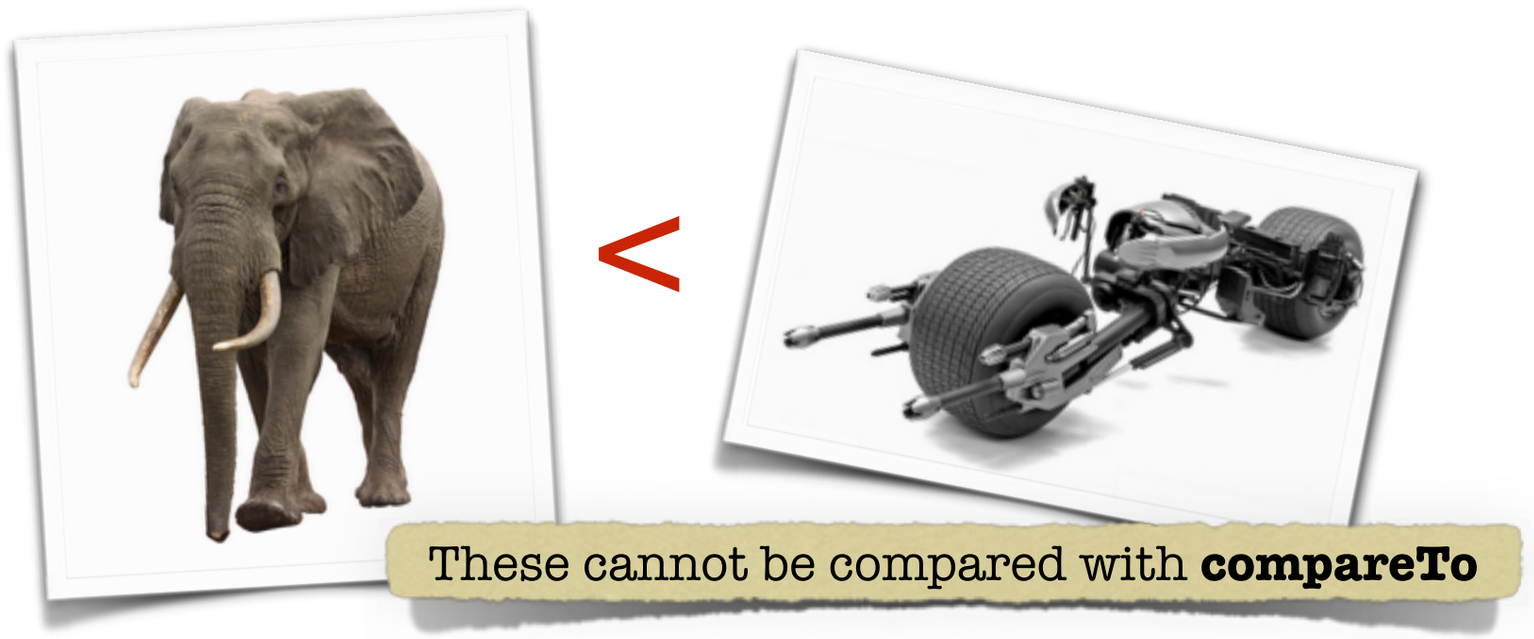
NB. It's important that T = Animal here!

Comparables

- Note that we can't compare things with different unrelated generic instances:

```
Vehicle batpod = new Motorbike();
Animal  nellie = new Elephant();

Animal nellie.compareTo(batpod);  //BOGUS!
```



Comparables

- While unrelated objects cannot be compared, inheritance does let us compare things in the same hierarchy

```
Animal dec    = new Ant();
Animal nellie = new Elephant();
assert dec.compareTo(nellie) == -99999;
```

Comparables

- While unrelated objects cannot be compared, inheritance does let us compare things in the same hierarchy

```
Animal dec    = new Ant();
Animal nellie = new Elephant();
assert dec.compareTo(nellie) == -99999;
```

Q. Is Dec treated as an Ant or an Animal?


.compareTo(

) = -99999;

Comparables

- While unrelated objects cannot be compared, inheritance does let us compare things in the same hierarchy

```
Animal dec    = new Ant();
Animal nellie = new Elephant();
assert dec.compareTo(nellie) == -99999;
```

Q. Is Dec treated as an Ant or an Animal?

```
.compareTo(
```

```
) = -99999;
```

A.Dynamic dispatch dictates that the receiver is dynamically resolved: dec is an ant

Comparables

- While unrelated objects cannot be compared, inheritance does let us compare things in the same hierarchy

```
Animal dec    = new Ant();
Animal nellie = new Elephant();
assert dec.compareTo(nellie) == -99999;
```

Q. Is Dec treated as an Ant or an Animal?

Q. How is nellie passed: as an Elephant or Animal?

```
.compareTo(
```

```
) = -99999;
```

A.Dynamic dispatch dictates that the receiver is dynamically resolved: dec is an ant

Comparables

- While unrelated objects cannot be compared, inheritance does let us compare things in the same hierarchy

```
Animal dec    = new Ant();
Animal nellie = new Elephant();
assert dec.compareTo(nellie) == -99999;
```

Q. Is Dec treated as an Ant or an Animal?

Q. How is nellie passed: as an Elephant or Animal?

.compareTo(

) = -99999;

A.Dynamic dispatch dictates that the receiver is dynamically resolved: dec is an ant

A. Parameters are always passed statically: nellie is treated like an Animal

Comparables

- The previous *compareTo* worked because the parameter was dispatched statically
- We can also consider the following situation

```
Animal   dec    = new Ant();
Elephant nellie = new Elephant();
assert dec.compareTo(nellie) == -99999;
```

Comparables

- The previous *compareTo* worked because the parameter was dispatched statically
- We can also consider the following situation

```
Animal   dec    = new Ant();
Elephant nellie = new Elephant();
assert dec.compareTo(nellie) == -99999;
```

Q. The parameter is now an Elephant, not an Animal: should this work?

Comparables

- The previous *compareTo* worked because the parameter was dispatched statically
- We can also consider the following situation

```
Animal   dec    = new Ant();
Elephant nellie = new Elephant();
assert dec.compareTo(nellie) == -99999;
```

Q. The parameter is now an Elephant, not an Animal: should this work?

A. No: Without fancy machinery, this is a type error!

Comparables

- The previous *compareTo* worked because the parameter was dispatched statically
- We can also consider the following situation

```
Animal   dec    = new Ant();
Elephant nellie = new Elephant();
assert dec.compareTo(nellie) == -99999;
```

Q. The parameter is now an Elephant, not an Animal: should this work?

A. No: Without fancy machinery, this is a type error!

A. But it works! Java allows **upcasting** of parameters!

Comparables

- Once you have implemented the *Comparable* interface, your elements have a *natural ordering*
- The natural ordering is used by default when you use the *Collections.sort* method

```
List<Animal> animals = new ArrayList<Animal>();
animals.add(dec);
animals.add(nellie);
Collections.sort(animals);
```

Q. Why doesn't **animals.sort** exist?

A. Sorting in Java is a List centric operation

Comparables

- Some collections, such as *SortedSet*, store all the elements in their natural ordering
- The corresponding iterator is thus sorted!

```
SortedSet<Animal> animals = new TreeSet<Animal>();
animals.add(nellie);
animals.add(scooby);
…
for (Animal animal : animals) {
  // animals come out in sorted order
}
```

NB. This works because a **SortedSet** is **Iterable**

Comparators

Comparators

- Sometimes the *natural ordering* is not what you're after:
  - The nodes in a graph might be ordered by their name, but you want them ordered by the number of neighbours
    - Animals might be compared by their height instead of their weight
    - There might be *no* natural ordering!
- Sometimes the code might unmodifiable, and without a defined ordering

Comparators

- Instead of forcing a class to be *Comparable*, you can instead provide a *Comparator*

```
interface Comparator<X> {
  int compare(X x1, X x2);
}
```

- Different implementations of this interface will be different ways of comparing

Comparators

- For instance, you might compare on *age*

```
class DogAgeComparator implements Comparator<Dog> {
  public int compare(X x1, X x2) {
    return (x1.age - x2.age);
  }
```

- Alternatively, maybe you compare *names*

```
class DogNameComparator implements Comparator<Dog> {
  public int compare(X x1, X x2) {
    return (x1.name - x2.name);
  }
}
```

Comparators

- For instance, you might compare on *age*

```
class DogAgeComparator implements Comparator<Dog> {
  public int compare(X x1, X x2) {
    return (x1.age - x2.age);
  }
```

- Alternatively, maybe you compare *names*

```
class DogNameComparator implements Comparator<Dog> {
  public int compare(X x1, X x2) {
    return (x1.name.length() - x2.name.length());
  }
}
```

NB. In both cases we return an **int** that measures how far away the two objects are from one another

Comparators

- As with *Comparable* classes, you can use a *Comparator* to sort using two techniques

- If your method of comparison will not change over time, then you can use a *Comparator* when you initialise the collection:

```
SortedSet<Dog> dogs =
  new TreeSet<Dog>(new DogAgeComparator());
```

- This only makes sense if you will often require the elements to be sorted in one specific way

Comparators

- If you only sort as a one-off, or the method of comparing changes, you should use an intermediate *List*

```
Collection dogs = … ;
List<Dog> listDogs = new ArrayList<Dog>(dogs);
Collections.sort(listDogs, new DogAgeComparator());
for (dog : listDogs) {
  // dogs are in age order
}

Collections.sort(listDogs, new DogNameComparator());
for (dog : listDogs) {
  // dogs are in name order
}
```

Comparators

- Lists also have a *sort* method that uses a comparator

```
Collection dogs = … ;

List<Dog> listDogs = new ArrayList<Dog>(dogs);
listDogs.sort(new DogAgeComparator());
for (dog : listDogs) {
  // dogs are in age order
}

listDogs.sort(new DogNameComparator());
for (dog : listDogs) {
  // dogs are in name order
}
```

Design Patterns

Design Patterns

- Engineers encounter the *same* problems many times in different forms
- Since the *problem* is the same, the ideal *solution* ends up being the same too
- We call those solutions *design patterns*
- You have *already* encountered several patterns!

Design Patterns

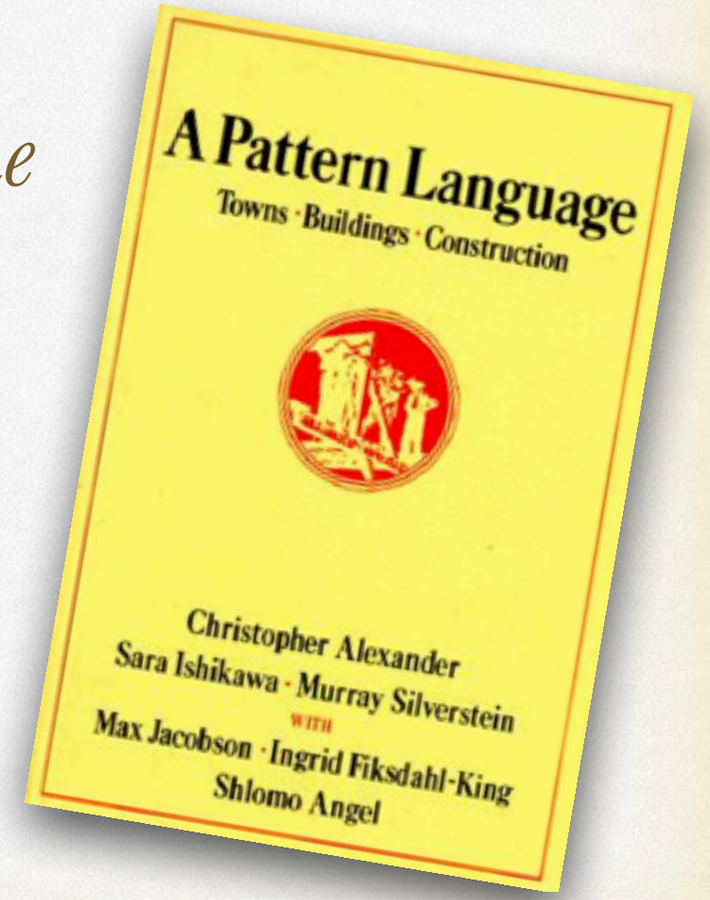
- Design patterns were introduced by architect Christopher Alexander in around 1977

- He introduced patterns to describe solutions in *architecture*, but the ideas were picked up by *Software Engineers* and applied to programming

Design Patterns

… each pattern represents our current best guess as to what arrangement of the physical environment will work to solve the problem presented …

— Christopher Alexander et al., A Pattern Language, p. xv



Design Patterns

- The introduction of patterns to software was in the book: “Design Patterns: Elements of Reusable Object-Oriented Software” by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides in 1994
- Often called “The Gang of Four” or “GoF”
- The book is based on techniques the developers believed made good object-oriented software design

Design Patterns

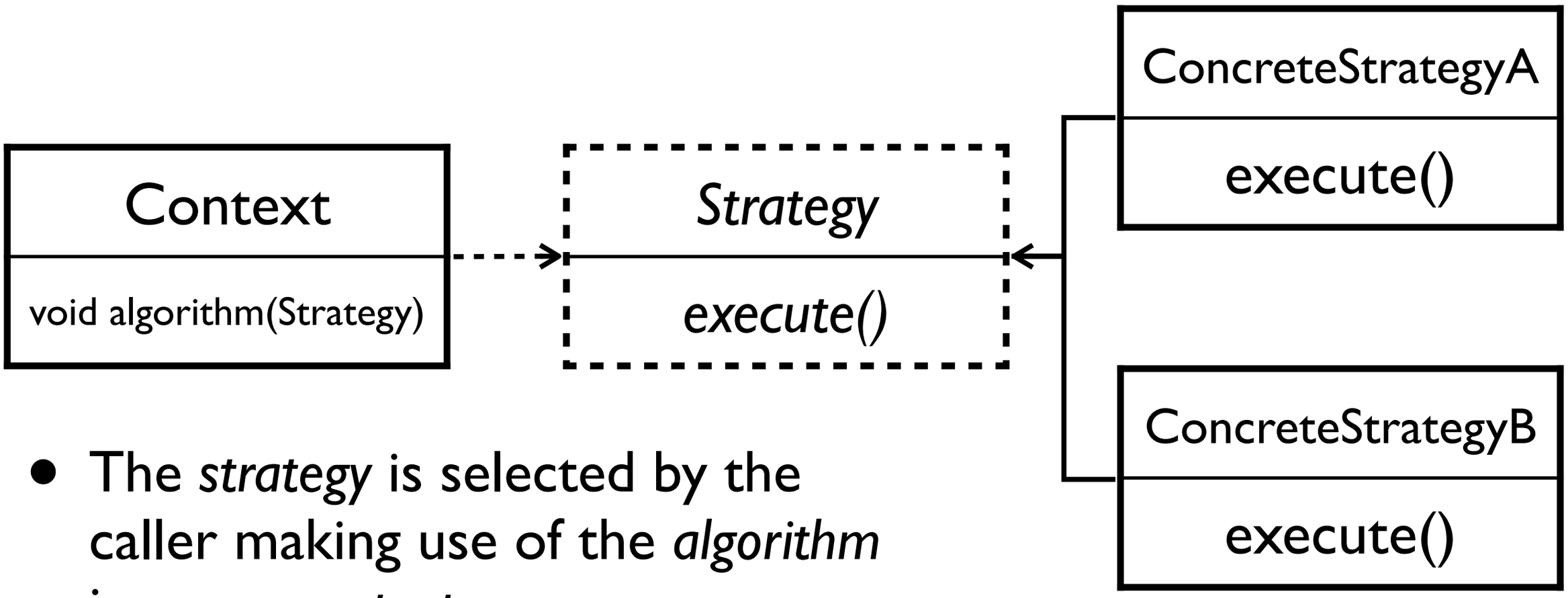
- Design patterns are useful because they give us a *vocabulary* for describing problems and their solutions
- They also make us aware of challenges we might face in the wild—along with sensible solutions
- Design patterns are *guides* and are expressed loosely and informally: there's no established formal expression of each pattern

Strategy Pattern

Strategy Pattern

- GoF: Defines a set of encapsulated algorithms that can be swapped to carry out a specific behaviour
- Example: You might want to program to a minimum spanning tree interface, allowing you to decide to use Prim's, Dijkstra's or something else.

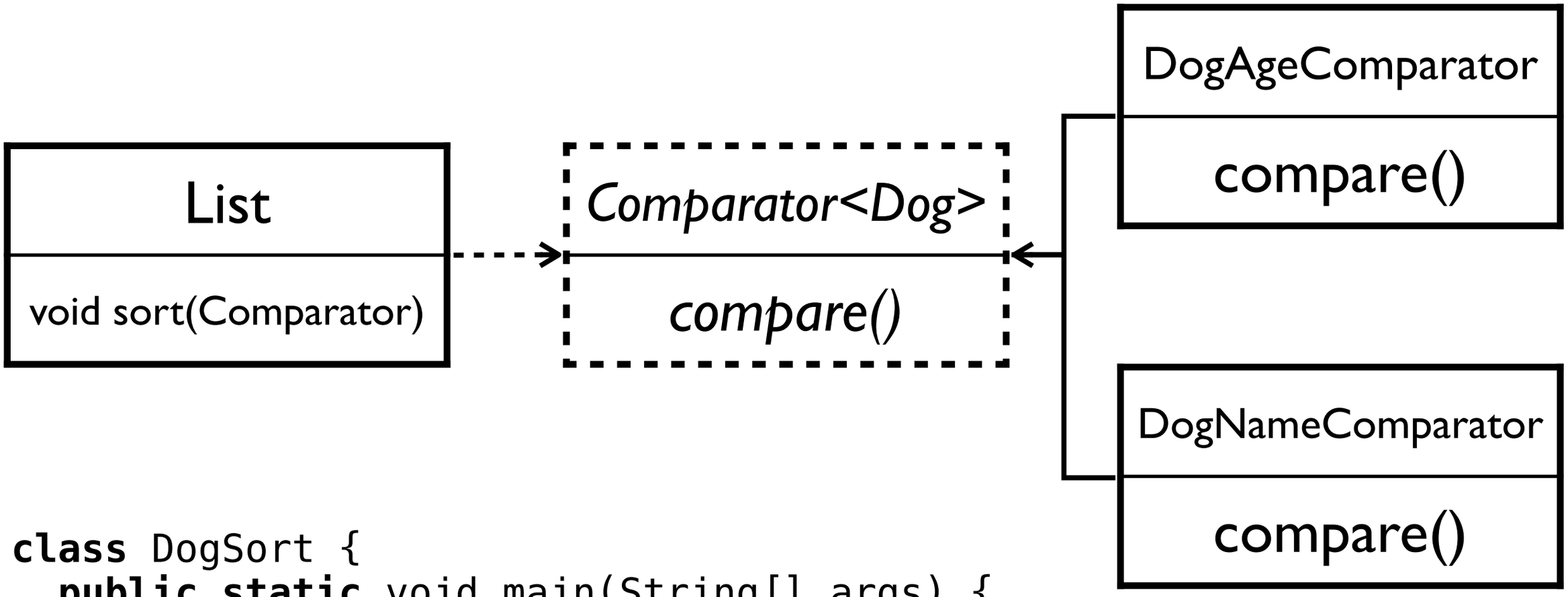
Strategy Pattern



- The *strategy* is selected by the caller making use of the *algorithm* in some *context*
- The *algorithm* uses *execute* but does not rely on its implementation

Variations bake the Strategy into the context on initialisation, rather than making it a parameter to the algorithm

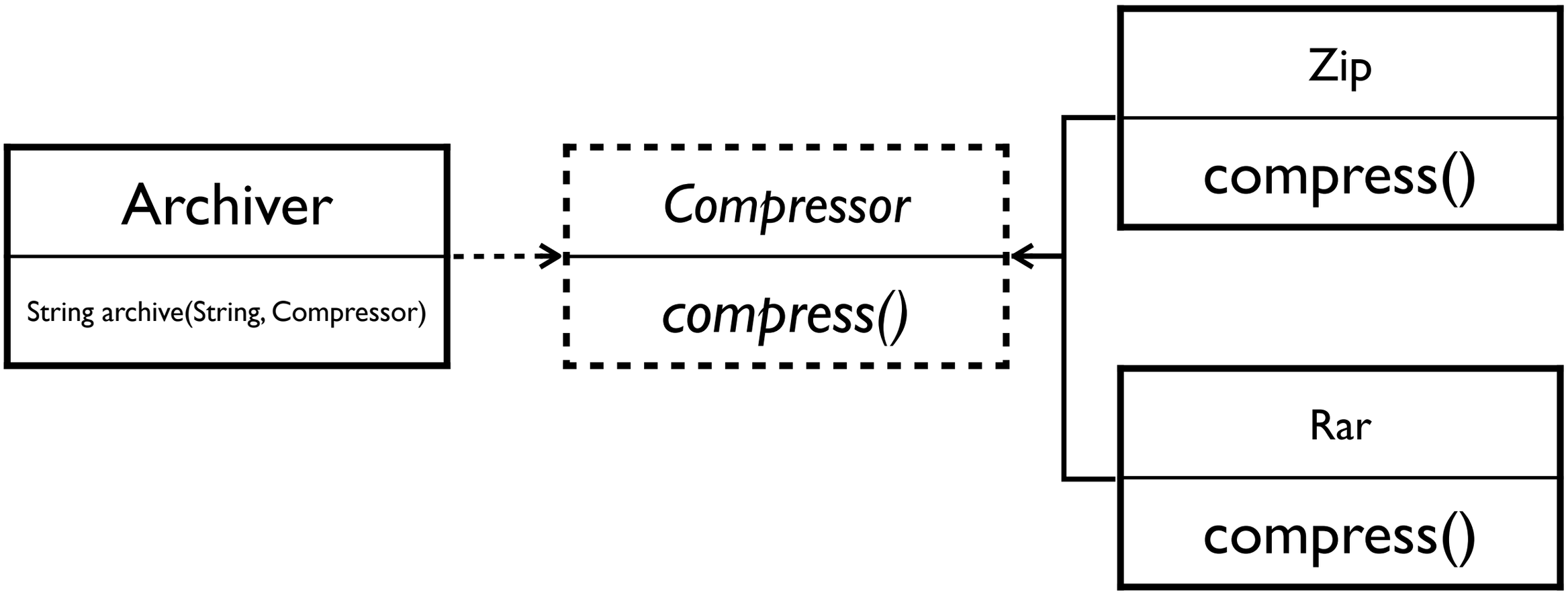
Strategy Pattern




```
class DogSort {
  public static void main(String[] args) {
    List<Dog> listDogs = new ArrayList<Dog>(dogs);
    Collections.sort(listDogs, new DogAgeComparator);
    for (dog : listDogs) {
      // dogs are in age order
    }
  }
}
```

NB. The specifics can vary from the “template” pattern

Strategy Pattern




```
class Archiver {
  public void archive(String fileName, Compressor compressor) {
    String file   = readFile(fileName);
    return (compressor.compress(file));
  }
}
```

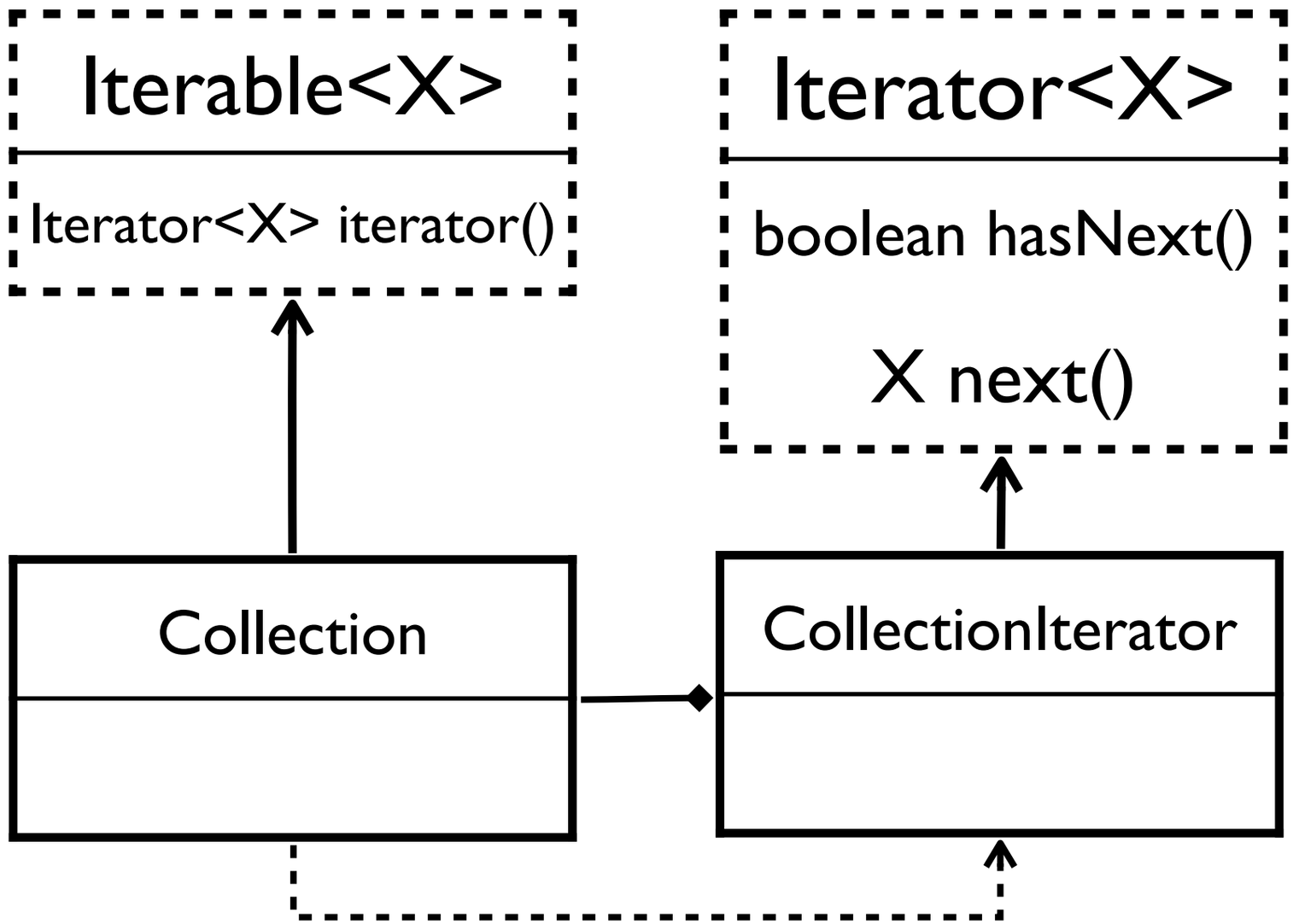
Warning: Java-esque pseudocode!

Iterator Pattern

Iterator Pattern

- GoF: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Example: You have a Set to store nodes in a graph. The structure of the Set is unknown, but you need to perform a computation on each node.

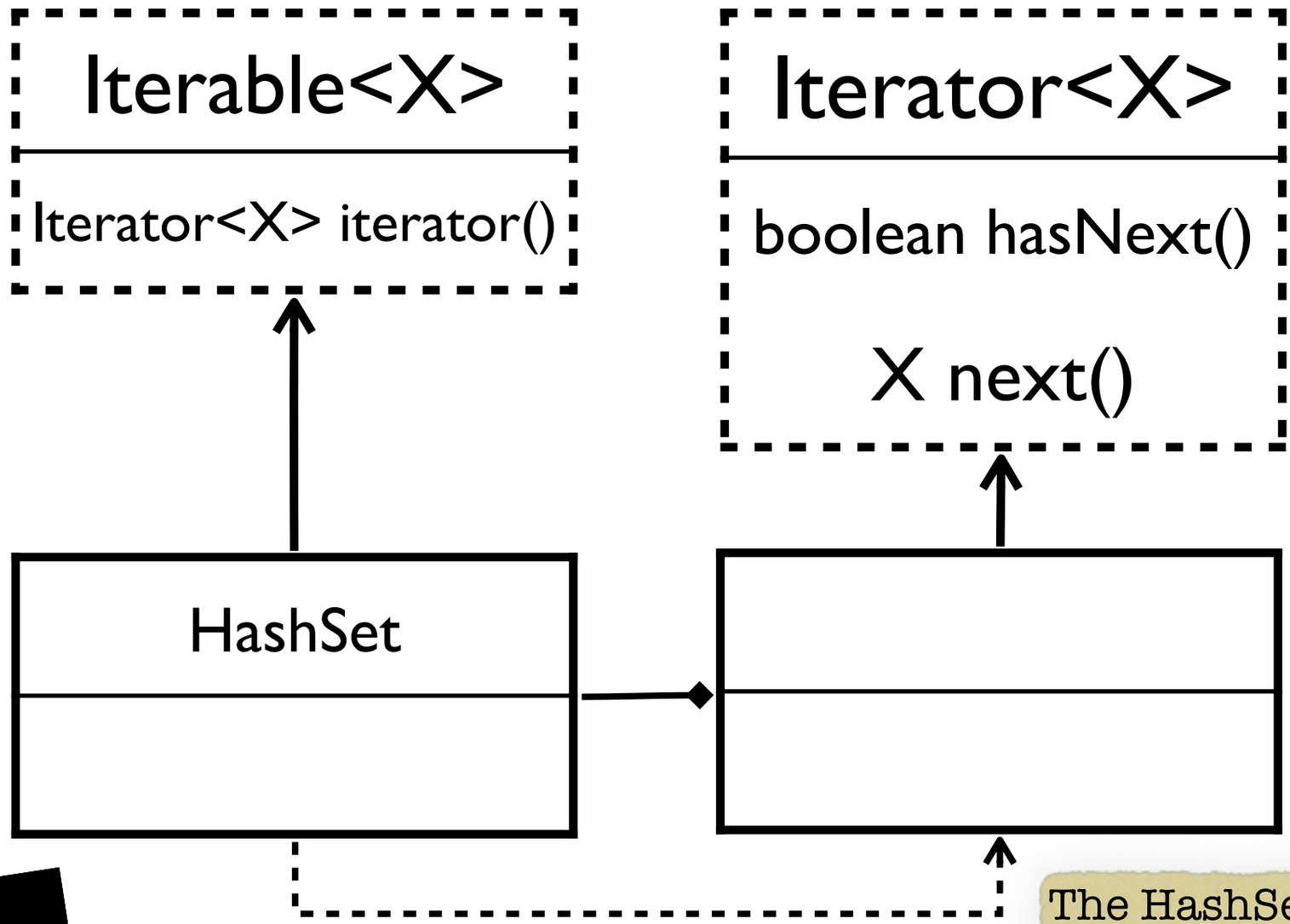
Iterator Pattern

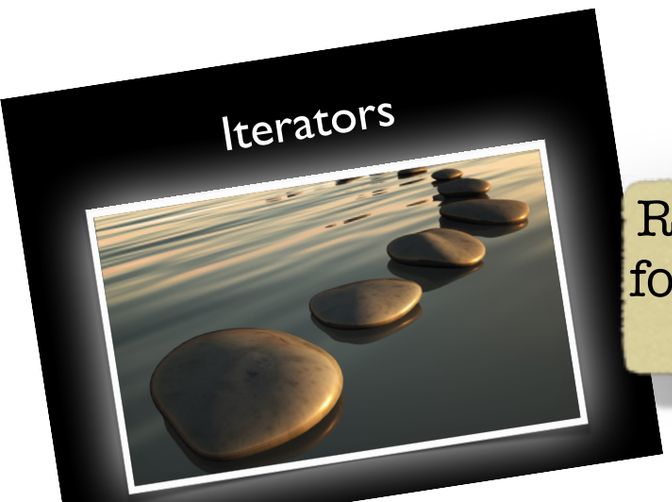


- A *Collection* is an *Iterable* object: asking for an *iterator* returns an appropriate *CollectionIterator* with details of the *Collection*

- An *iterator* traverses through a collection: as long as *hasNext* is true, *next* will return a value from the collection

Iterator Pattern





Remember: Java gives special for-loop support for collections that are Iterable

The HashSet Iterator might be an anonymous inner class defined inside HashSet

Wrapped Primitives

Wrapped Primitives

- PODs (plain old datatypes) are simple values
- Sadly, they can't be used in the type instantiation of a generic parameter
- Java provides simple *wrapper classes* that gives us a classy version

POD	Wrapped POD
`boolean`	`Boolean`
`char`	`Character`
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`

Java uses autoboxing to magically translate between the wrapped and plain versions

Wrapped Primitives

- The wrapped versions have a constructor that turns the POD into an object

```
Integer i = new Integer(3);
```

- Generally speaking, you'll never really need the wrapped versions directly, unless you're storing values in a collection type
- There are a few caveats:
    - Remember == and != are not int comparisons
    - Integer references expose you to the wrath of nulls
    - Integer is slower than int

Wrapped Primitives

- The wrapped versions have a constructor that turns the POD into an object

```
Integer i = new Integer(3);
```

- Generally speaking, you'll never really need the wrapped versions directly, unless you're storing values in a collection type
- There are a few caveats:
  - Remember == and != are not int comparisons
  - Integer references expose you to the wrath of nulls
  - Integer is slower than int

NB! In 99% of cases, favour clarity, correctness and defensive programming over micro-optimisations. Focus instead on good algorithms to keep asymptotic complexity down

Wrapped Primitives

- Java will do *autoboxing* for you, to automatically wrap and unwrap primitives where appropriate
- Without autoboxing, you need to wrap things yourself:

```
List<Integer> xs = new ArrayList<Integer>();
xs.add(new Integer(42));
xs.add(new Integer(35));
int sum = 0;
for (Integer x : xs) {
  sum = sum + x.getValue();
}
```

- Java can take care of the boxing for you:

```
List<Integer> xs = new ArrayList<Integer>();
xs.add(42);
xs.add(35);
int sum = 0;
for (int x : xs) {
  sum = sum + x;
}
```