Classes and Types
Interfaces and Inheritance

and collections too ...

that’s a lot
Week 7 assignment

An Agenda is a list of Appt
They could represent free or busy times

A MeetingScheduler merges free time agendas
Start with one Appt giving range of times
For each attendee
Restrict available times
to the free times of attendee
Report possible times for all to meet
Why Types?

Interpret the meaning of an operation
- Does that + mean integer addition, floating point addition, or string concatenation?
- Is x.toString() the method defined in Foo.java or in Bar.java?

Prevent meaningless (dangerous) operations
- Applying integer addition to strings would create nonsense. Calling a method that doesn’t exist might be a security back door.

Static or dynamic checking – same purpose
Classes are Types

Every class in Java is also a type
Some types are not classes
   The primitive types int, boolean, float, ...
   Interfaces are types but not classes

Type checking includes both classes and other types
Subclasses and Subtypes

class AvailTime extends Agenda { ... }

Every AvailTime object “is-a” Agenda object
(AvailTime is a subtype of Agenda)

Any operation that is allowed on Agenda is also allowed on AvailTime
(even though it might do something different)
A hierarchy of types

class Creature { ... }
class Monster extends Creature { ... }
class Harmless extends Creature { ... }
Class Zombie extends Monster { ... }

Object
  Creature
      Monster
         Zombie
            Harmless
A hierarchy of types

Interface Repellable { void repel(); }

Class Zombie extends Monster
  implements Repellable { ... }

Class Vampire extends Immortal
  implements Repellable { ... }
Variables have two types
(at least)

Creature b = new Zombie();

static type of b is Creature
dynamic type of b is Zombie

OK because Zombie is a subtype of Creature.
Covariance and Contravariance
(forget the words, don’t forget the concept)

public Monster myMethod(Monster x) { ... }
...
Creature crigarth; Monstor mermazon;
Zombie zorgath;
\[\text{zorgath} = \text{mermazon}.\text{myMethod}(\text{crigarth}); \quad \text{// WRONG}\]
\[\text{crigarth} = \text{zorgath}.\text{myMethod}(\text{mermazon}); \quad \text{// OK}\]
Interfaces allow multiple supertypes
(but still only one superclass)

class Bar extends Foo implements Iterable { ... }

```
class Bar extends Foo implements Iterable {
  ...
}
```

```
interface Iterable

Object

Foo

Bar

Zot

BarNone

<<interface>> Iterable
```
Iterators ...

Java has a lot of “Collection” classes
Each has its own way of looping through the elements ... but that’s too hard to learn and remember

“Iterable” and “Iterator” (and “ListIterator”) allow many different implementations to provide a common, easy to remember interface
Interfaces vs Inheritance

We saw:

```java
class foo implements bar {
}
```

bar is an *interface*; foo is an *implementing class*

We can also extend or “inherit from” classes

```java
class foo extends zot {
}
```

zot is a *superclass*; foo is a *subclass*

We inherit to modify and extend an *implementation*
Like that one, but ...

class Shape {
    public abstract int area();
}

class Rect extends Shape {
    public int area() {
        return height * width;
    }
    ...
}

class Square extends Rect {
    public Square(int width) {
        return width * width;
    }
}
Like that one, but ...

class Shape {
    ...
}

class Rect extends Shape {
    public int area() { return height * width; }
    ...
}

class Square extends Rect {
    public int area() { return width * width; }
}
Like that one, but ...

```java
abstract class Shape {
    public abstract int area();
}

class Rect extends Shape {
    public int area() {
        return height * width;
    }
    ...
}
class Square extends Rect {
    public int area() {
        return width * width;
    }
}
```
```
shape

Shape[]

class height width

class height width

class height base

Square
super area

Rect
super area

Triangle
super area
```
Iterators
Any “iterable” collection type provides a way of obtaining an Iterator (or ListIterator) to look at elements in some order.

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

class TreeSet<E>
    implements Iterable<E>
    ...
```

```java
interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); // Optional operation
}
```
An iterator provides a way of looking at the elements in order, even if the original collection is not a simple list or array.
You’ve seen an iterator-like object

```java
static void indent( Scanner pgm,
                  PrintStream out ) {
    int blockLevel = 0;
    while ( pgm.hasNext() ) {
        String line = pgm.nextLine();
        ...
    }
}
```

Scanner moves “finger” through the file
TreeSet<Stuff> t;

ListIterator<Stuff> i = t.listIterator();

The iterator doesn’t really make a new list with the elements in order. It just knows how to keep a finger in the collection.
TreeSet<Stuff> t;

The “next()” method of an iterator moves the finger, and returns the item it moved over.

ListIterator<Stuff> i = t.listIterator();
Stuff s = t.next();

A
The iterator finger is actually between elements.

```java
ListIterator<Stuff> i = t.listIterator();
```
next() moves the finger forward and returns an element

ListIterator<Stuff> i = t.listIterator();

A   B   C   D   E   F   G

Stuff s = t.next();

A

A   B   C   D   E   F   G

s = t.next();

B

A   B   C   D   E   F   G
The whole purpose of iterators is to make it easier to write loops, like this:

```java
for (ListIterator<Appt> blocks = this.appts.listIterator(); blocks.hasNext();)
{
    Appt block = blocks.next();
    ...
}
```

or even better, like this (if we don’t need to modify the collection):

```java
for (Appt block: this.appts)
{
    ...
}
```

(which is really creating an iterator and calling the “`next()`” and “`hasNext()`” methods for us)