Inheritance

- Can create new classes by extending others
  - Subclass inherits all members of superclass
    - But cannot directly access `private` members
  - Can add new fields and new methods
  - Can override existing methods
  - Cannot remove fields or methods
- Can only extend *one* other class in Java
  - Makes for clear hierarchies (less complication)
  - But indirectly extend superclass's parent, …
    - *All* Java classes are descendants of `Object`
- Note: composition another way to reuse code

Polymorphism

- Literally: the ability to assume *many forms*
- OOP idea: a superclass reference can refer to many types of subclass objects
  - Each object may behave differently – if subclasses override methods
- Imagine a `Shape` class with a `draw()`method
  - Subclasses `Circle`, `Triangle`, … override `draw()`
  - Then say `void picture(Shape s) { s.draw(); }`
    - Object `s` is a `Shape` or a subclass of `Shape`
- Relies on "dynamic method binding"

Abstract classes and interfaces

- Abstract class has one or more `abstract` methods
  - Subclasses *must* implement these methods
  - Cannot instantiate – objects must be subclass objects
  - Subclasses inherit implementation *and* interface
- A Java `interface` has no implementation at all
  - e.g., "… `implements Comparable`" means the class responds to `compareTo(Object other)`
  - A class may implement multiple interfaces
    - No implementation to inherit – so no complications

More about interfaces

- All methods are `public abstract` – omit explicit modifiers by convention
- Constants okay too
  - All `public static final` – omitted by convention
  - Must be initialized when declared
- Can extend, just like classes
  - But okay to extend more than one:

```
public interface SerializableRunnable
        extends java.io.Serializable, Runnable
```

- Tend to be much more flexible than classes as a way to unite objects in system designs
  - Hence the basis of many “design patterns”

What is abstraction?

- Workable answer – a *blurring* of details
- Idea: agree to ignore certain details (*for now*)
  - Convert original problem to a simpler problem
  - Procedural abstraction is one way to simplify – main algorithm calls methods to handle detailed steps
- Works for data types too
  - Think (*and write code*) in terms of abstract data types like Lists, Stacks, Trees, …
    - What should matter – what you can do with a List
    - What should not matter – what goes on inside the List
  - *Assume* the ADT works – just use it!

Example: A Priority Queue ADT

- ADT is defined by its interface – *what* it does
- Imagine a `PriorityQueue` with these methods:

```
void insert(Comparable item);
```

/ * add the item to the queue */

```
Comparable remove();
```

/ * always returns item with highest priority */

```
boolean isEmpty();
```

/* true if queue has no items */

- Never mind *how* it works – think about that later

Interface is enough to use ADT

- Easy way to sort – let a priority queue do it

```
void easySort(Comparable a[]) {
   PriorityQueue pq = new PriorityQueue();
   int i, n = a.length;
   for (i=0; i<n; i++)  // put all items in queue
      pq.insert(a[i]);
   for (i=n-1; i>=0; i--)  // items come out sorted
      a[i] = pq.remove();
}  // There are more efficient ways to sort, but that's not the point.
```

- The point is that we can use it without knowing how it works.
- Abstraction is good!

Linked data structures

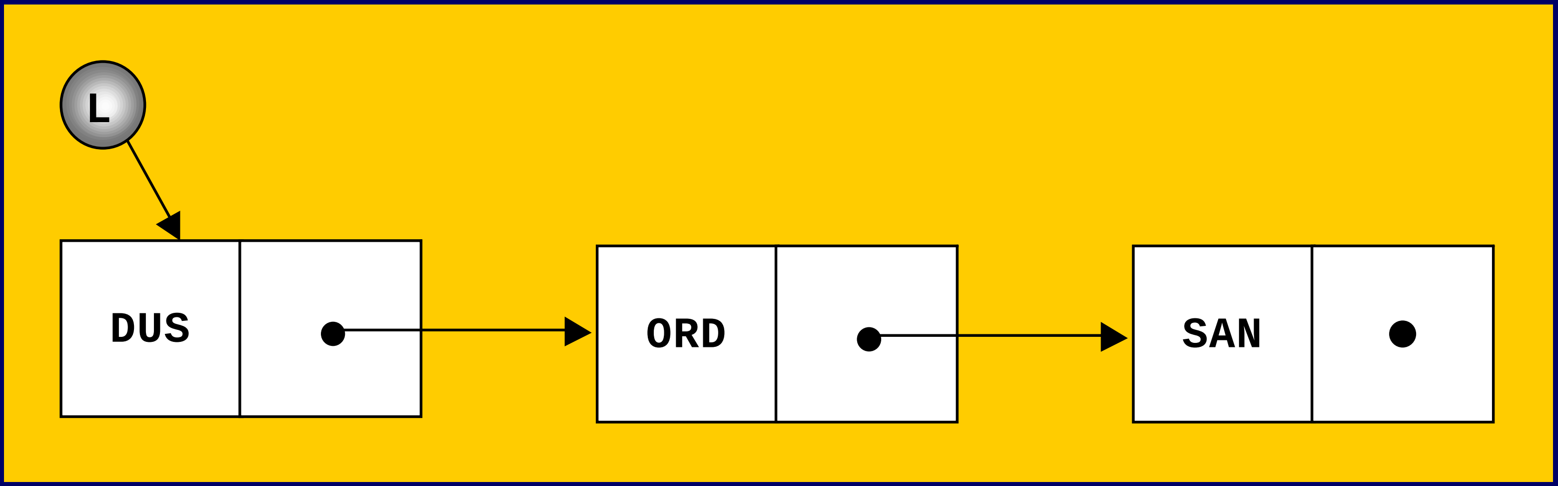
- Made up of nodes and links between nodes
  - As purpose is data storage/retrieval, also contains information field(s) inside nodes
- Simplest is a linear linked list with single links:
  - Key is to define a node class to hold info and a link:

```
class ListNode {        // note: class Entry<E> in Collins text
   Object data;
   ListNode next;
... /* maybe set and get methods for fields if not nested class */ }
```

  - By convention, `next == null` if last node in list
    - Otherwise it refers to next node in the list

So what is a linked list, really?

- Answer: a sequence of zero or more nodes, with each node pointing to the next one
- Need: a reference to the first node – `first`
  - Often this reference is considered “the list”
  - Might be null – just means it is an empty list



List class can hide details

- Interface says nothing about list nodes
- Best to prevent clients from direct node access
  - Clients don't have to know nodes even exist!
  - Clients cannot set links inappropriately
- Easiest way (with Java) – private nested class:

```
public class LinkedList {
      ListNode first;
      …
      private static class ListNode { }
}
```

Nested classes/interfaces

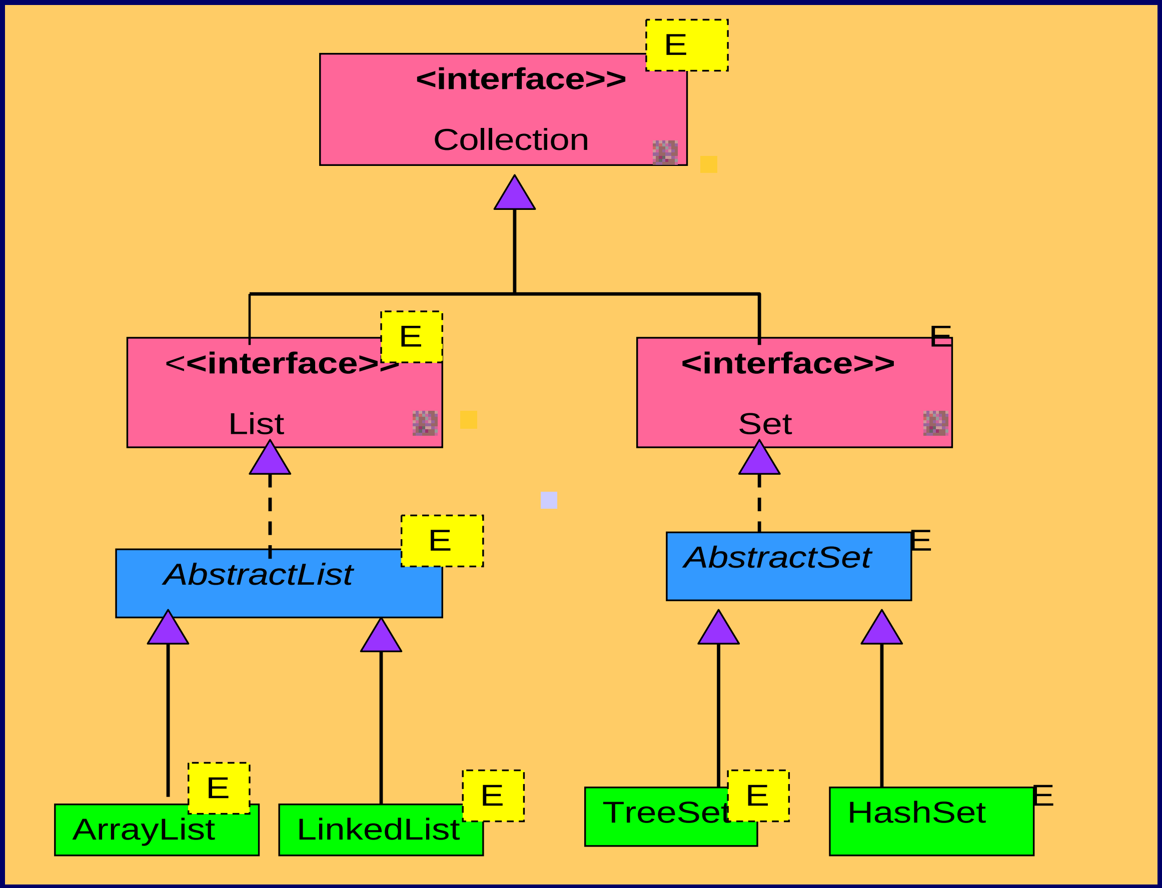
- Okay to define a class (or interface) inside another class (or interface)
  - Good for grouping logically related types
  - Nested and outer class share data – even `private`
- If declared `static` – works just like non-nested
  - Can extend, or be extended like any other class
  - Can only access static fields/methods of outer class
- If not declared static – called an inner class
  - Instances of the inner class are *associated with an instance* of outer class – the "enclosing object"

FYI: more Java nested classes

- Local classes
  - Defined *inside* methods or other blocks
  - Not members of the class – local to the block
- Anonymous classes
  - When just want an object; no need for type
  - Must extend existing class or implement interface
    - Purpose is to override one or more methods
  - Used frequently for event-handling:

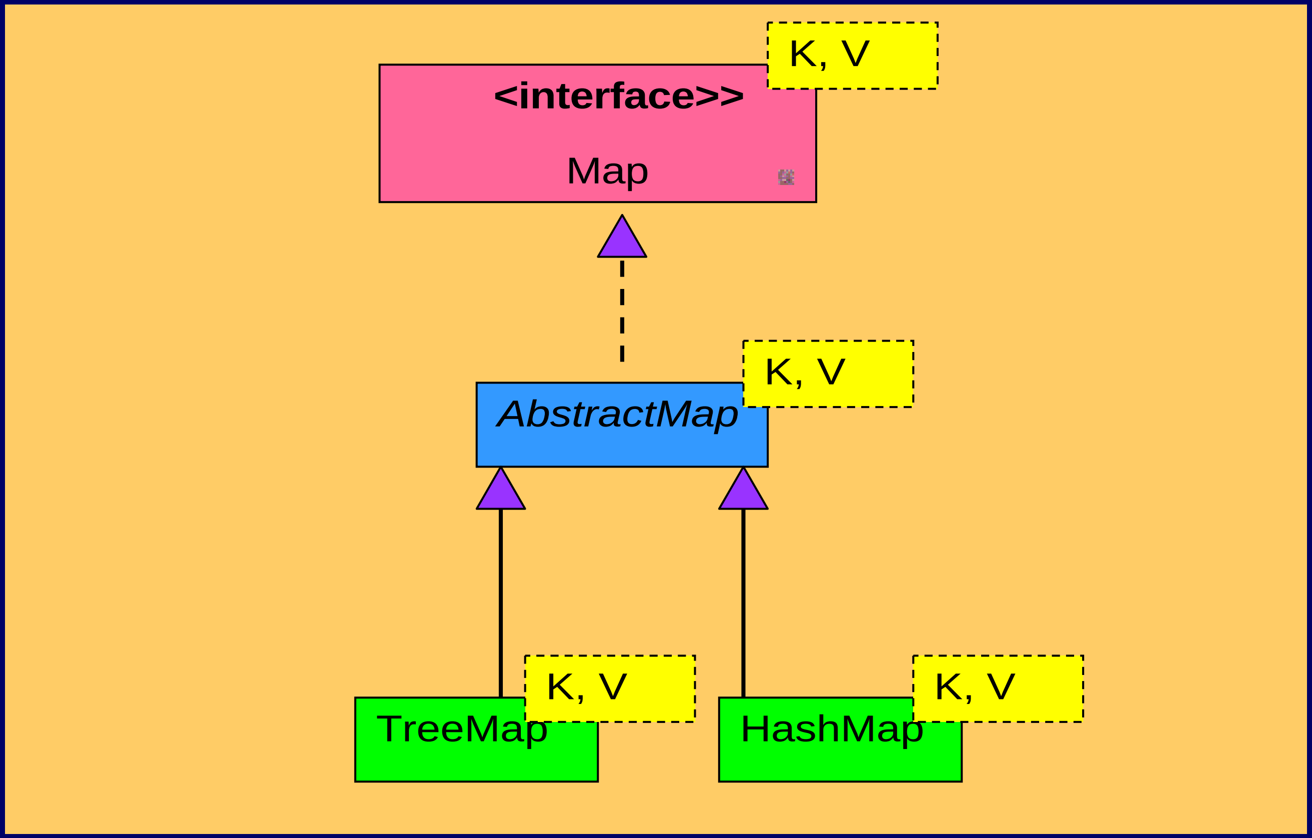
```
new ActionListener (  // define anonymous class right here:
{ public void actionPerformed(ActionEvent e) {…} }
);
```

Collection hierarchy (simplified)



See RandomList.java and RandomSet.java (Collins pp. 111, 114)

Map hierarchy (simplified)



See StudentMap.java (Collins p. 117)

Testing

- Goal is to find faults
- Faults (a.k.a. bugs) cause systems to fail
  - e.g., a system crashes – the most obvious type of fault
  - e.g., a security system that allows unauthorized entry
  - e.g., a shot-down video game plane continues on path
- Can verify the presence of bugs, not their absence
  - Testing fails if no bugs are found! (a good thing really)
- Testing and debugging are separate processes
  - Testing identifies; debugging corrects/removes faults

Testing steps

- Unit testing – insure each part is correct
  - Independently test each function in each file
- Integration testing – insure parts work together
  - Test functions working together; not whole system yet
- System testing – insure system does what it is supposed to do
  - Lots of testing left to do – especially for large systems
  - Includes functional tests, performance tests, acceptance tests, and installation tests

Testing approaches

- Black box testing – best by independent tester
  - Plan good test cases, and conduct *automated* tests
- Open box testing – a separate, preliminary activity
  - "Coverage testing" is the goal
    - i.e., test every line of code at least once
  - Includes unit testing and integration testing
- Regression testing – repeat tests frequently
  - Because fixing a new bug may re-introduce old ones
  - Easy to do with automated testing framework

Test plans (i.e., test data contents)

- Test a representative sample of normal cases
  - Usually no way to test all possibilities
    - But don't really need to – random sample of cases okay
  - At least be sure to test all normal operations
- Test boundary cases
  - Test the extremes – includes empty cases, lone cases, last case, first case, …, any other "edge" cases
- Test error cases too
  - e.g., test how bad input is handled – should not crash!

Program Correctness

- A correct program (1) always produces the right answer, and (2) terminates
- Predicate logic – used to verify *partial* correctness of program segments: `p{S}q`
  - If predicate `p` is true, after program segment `{S}` executes (and terminates), predicate `q` is true
  - e.g., `x > 0 {z = x + y} z > y`
- Basic idea: trace the algorithm (step by step) – verify correctness of intermediate results
  - And/or test such assertions in the code itself

Programming with assertions

- Assertions are conditions that should *all* be true for a program to be considered correct
- Most important types of assertions:
  - Method "contract" clauses
    - Pre-conditions – must be true on function entry
    - Post-conditions – must be true on function exit, *if the pre-conditions were true beforehand*
  - Loop invariants – must be true on each iteration

Javadoc

- Cheap external documentation – get to know it
  - `/**` Comment each public declaration.
    - `*` Including classes, variables, methods.
    - `*` Use `@param`, `@return`, `@throws`, other tags. `*/`
- Let clients "program to the interface, not the implementation" – all they see is the interface
  - But must be complete – even if redundant sometime
  - Most critical – pre-conditions and post-conditions
- Remember to update to reflect any changes!

Executable assertions

- Historical origin – a C macro called assert
  - e.g., pre-condition of `inverse(x)` is that x is not zero

```
double inverse(int x)
{  assert(x != 0); /* halts with message if x == 0 */
   return 1. / x;   /* better than crashing here */ }
```

- Java counterpart available since version 1.4
  - New keyword `assert`, and related class `AssertionError`:

```
assert x!= 0; // throws AssertionError if false
```

More executable assertions

- New keyword, `assert`, required special handling for compilation before version 1.5
  - e.g., `javac -source 1.4 MyProgram.java`
    - Otherwise got syntax errors wherever assert keyword used
  - Of course, cannot use `assert` as an identifier either
  - Likewise cannot compile at all with 1.3 or earlier
- Also must *enable* assertions when run
  - e.g., `java -ea MyProgram`
  - Idea is to speed up run-time if code is already tested

More using assertions

- Also use assert to check post-conditions
  - In this case, errors are the fault of this method
- And assert loop invariants – useful for debugging
- Q. Why assert to check your own code?
  - Answer: catch bugs early and effectively
    - Bugs appear as soon as testing begins
    - Also know where bug occurred, and maybe where to fix it
- Note: use assert as a development tool ONLY
  - Just do not use `-ea` parameter for execution
- Also note: use other exceptions to enforce public method contracts – as specified in javadocs

Exceptions

- Object-oriented way to signal *exceptional* conditions
  - When a method does not know what else to do, it should `throw` an `Exception` object (or `Error` object in extreme cases)
- If invoked in a `try` block, the calling method can `catch` an exception if it knows how to *handle* it – otherwise exception passes through.
- If not handled by any method, execution stops with error message

Exception types

- Checked exceptions – must be caught, or the method must declare that it `throws` that exception type
  - Includes `IOException` and all of its subclasses
- Unchecked exceptions – subclasses of `RuntimeException`
  - e.g., `ArithmeticException`, `NumberFormatException`, `IllegalArgumentException`