Building Java Programs

Inner classes, generics, abstract classes

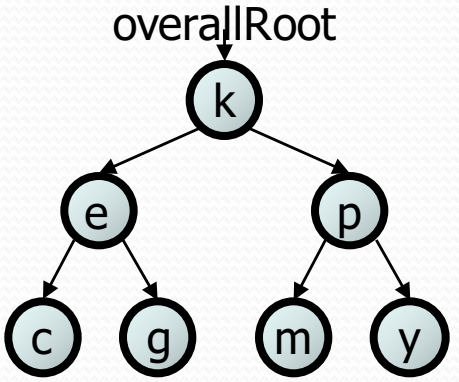
reading: 9.6, 15.4, 16.4-16.5

A tree set

- Our `SearchTree` class is essentially a set.
  - operations: `add`, `remove`, `contains`, `size`, `isEmpty`
  - similar to the `TreeSet` class in `java.util`

- Let's actually turn it into a full set implementation.
  - *step 1:* create ADT interface; implement it
  - *step 2:* get rid of separate node class file
  - *step 3:* make tree capable of storing any type of data (not just `int`)

  - We won't rebalance the tree, take a data structures class to learn how!



Recall: ADTs (11.1)

- **abstract data type (ADT)**: A specification of a collection of data and the operations that can be performed on it.
  - Describes *what* a collection does, not *how* it does it.

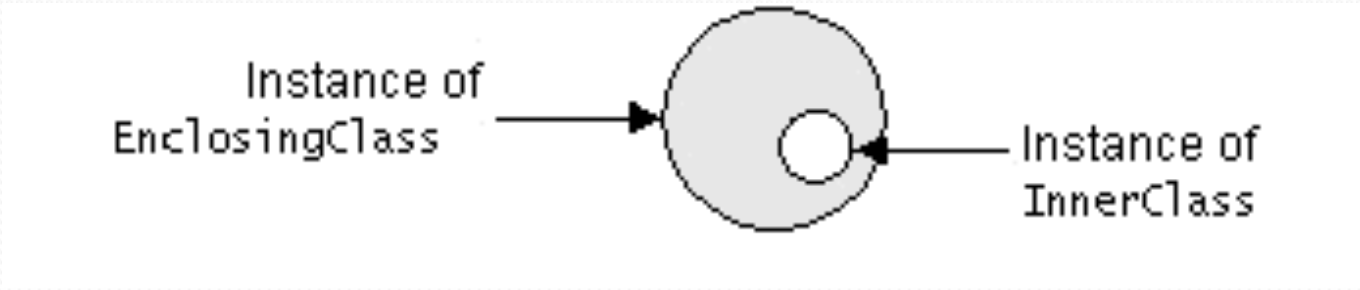
- Java's collection framework describes ADTs with interfaces:
  - `Collection`, `Deque`, `List`, `Map`, `Queue`, `Set`, `SortedMap`

- An ADT can be implemented in multiple ways by classes:
  - `ArrayList` and `LinkedList` implement `List`
  - `HashSet` and `TreeSet` implement `Set`
  - `LinkedList`, `ArrayDeque`, etc. implement `Queue`

Inner classes

To get rid of our separate node file, we use an *inner class*.

- **inner class**: A class defined inside of another class.
  - inner classes are hidden from other classes (encapsulated)
  - inner objects can access/modify the fields of the outer object



Inner class syntax

```
// outer (enclosing) class
public class name {
    ...

    // inner (nested) class
    private class name {
        ...
    }
}
```

- Only this file can see the inner class or make objects of it.
- Each inner object is associated with the outer object that created it, so it can access/modify that outer object's methods/fields.
  - If necessary, can refer to outer object as **OuterClassName**`.this`

Recall: Type Parameters

```
ArrayList<Type> name = new ArrayList<Type>();
```

- When constructing a `java.util.ArrayList`, you specify the type of elements it will contain in < and >.
  - `ArrayList` accepts a **type parameter**; it is a **generic** class.

```
ArrayList<String> names = new ArrayList<String>();
names.add("Marty Stepp");
names.add("Helene Martin");
names.add(42);  // compiler error
```

Implementing generics

```
// a parameterized (generic) class
public class name<Type> {
    ...
}
```

- Forces any client that constructs your object to supply a type.
  - Don't write an actual type such as String; the client does that.
  - Instead, write a type variable name such as `E` (for "element") or `T` (for "type").
  - You can require multiple type parameters separated by commas.
- The rest of your class's code can refer to that type by name.

Generics and inner classes

```
public class Foo<E> {
    private class Inner<E> {...}    // incorrect
    private class Inner {...}       // correct
}
```

- If an outer class declares a type parameter, inner classes can also use that type parameter.
- The inner class should NOT redeclare the type parameter.
  - (If you do, it will create a second type param with the same name.)

Issues with generic objects

```
public class TreeSet<E> {
    ...
    public void example(E value1, E value2) {

        // BAD:  value1 == value2      (they are objects)
        // GOOD: value1.equals(value2)

        // BAD:  value1 < value2
        // GOOD: value1.compareTo(value2) < 0

    }
}
```

- When testing objects of type `E` for equality, must use `equals`
- When testing objects of type `E` for `<` or `>`, must use `compareTo`
  - Problem: By default, `compareTo` doesn't compile! What's wrong!

Type constraints

```
// a parameterized (generic) class
public class name<Type extends Class/Interface> {
    ...
}
```

- A **type constraint** forces the client to supply a type that is a subclass of a given superclass or implements a given interface.
  - Then the rest of your code can assume that the type has all of the methods in that superclass / interface and can call them.

Generic set interface

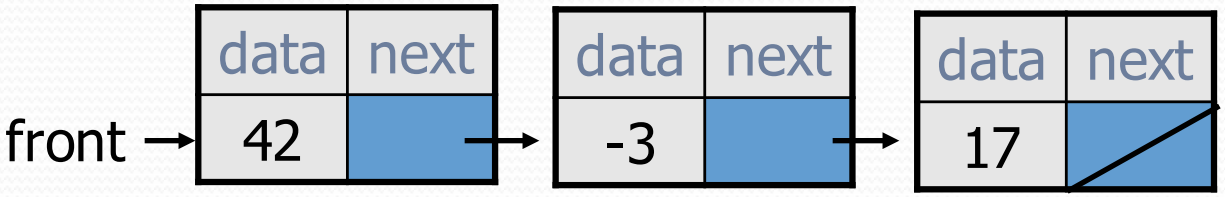
```
// Represents a list of values.
public interface Set<E> {
    public void add(E value);
    public boolean isEmpty();
    public boolean contains(E value);
    public void remove(E value);
    public int size();
}

public class TreeSet<E extends Comparable<E>>
                     implements Set<E> {
    ...
```

Our list classes

- We have implemented the following two list collection classes:
  - `ArrayIntList`

index	0	1	2
value	42	-3	17

  - `LinkedIntList`



  - Problems:
    - We should be able to treat them the same way in client code.
    - Linked list carries around a clunky extra node class.
    - They can store only `int` elements, not any type of value.
    - **Some methods are implemented the same way (redundancy).**
    - It is inefficient to get or remove each element of a linked list.

Generics and arrays (15.4)

```
public class Foo<T> {
    private T myField;                          // ok

    public void method1(T param) {
        myField = new T();                  // error
        T[] a = new T[10];                  // error

        myField = param;                        // ok
        T[] a2 = (T[]) (new Object[10]);    // ok
    }
}
```

- You cannot create objects or arrays of a parameterized type.
- You can create variables of that type, accept them as parameters, return them, or create arrays by casting from `Object[]`.

Common code

- Notice that some of the methods are implemented the same way in both the array and linked list classes.
  - `add(`**`value`**`)`
  - `contains`
  - `isEmpty`
- Should we change our interface to a class? Why / why not?
  - How can we capture this common behavior?

Abstract classes (9.6)

- **abstract class**: A hybrid between an interface and a class.
  - defines a superclass type that can contain method declarations (like an interface) and/or method bodies (like a class)
  - like interfaces, abstract classes that cannot be instantiated (cannot use `new` to create any objects of their type)

- What goes in an abstract class?
  - implementation of common state and behavior that will be inherited by subclasses (parent class role)
  - declare generic behaviors that subclasses must implement (interface role)

Abstract class syntax

```
// declaring an abstract class
public abstract class name {
    ...
    // declaring an abstract method
    // (any subclass must implement it)
    public abstract type name(parameters);
}
```

- A class can be `abstract` even if it has no abstract methods
- You can create variables (but not objects) of the abstract type

- Exercise: Introduce an abstract class into the list hierarchy.

Abstract and interfaces

- Normal classes that claim to implement an interface must implement all methods of that interface:

```
public class Empty implements IntList {}  // error
```

- Abstract classes can claim to implement an interface without writing its methods; subclasses must implement the methods.

```
public abstract class Empty implements IntList {} //
  ok

public class Child extends Empty {}        // error
```

An abstract list class

```
// Superclass with common code for a list of integers.
public abstract class AbstractIntList implements IntList {
    public void add(int value) {
        add(size(), value);
    }

    public boolean contains(int value) {
        return indexOf(value) >= 0;
    }

    public boolean isEmpty() {
        return size() == 0;
    }
}


public class ArrayIntList extends AbstractIntList { ...
public class LinkedIntList extends AbstractIntList { ...
```

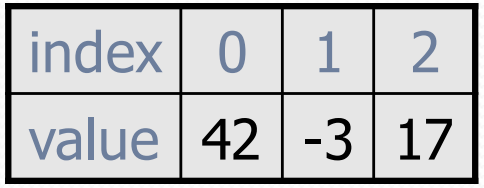
Abstract class vs. interface

- Why do both interfaces and abstract classes exist in Java?
  - An abstract class can do everything an interface can do and more.
  - So why would someone ever use an interface?

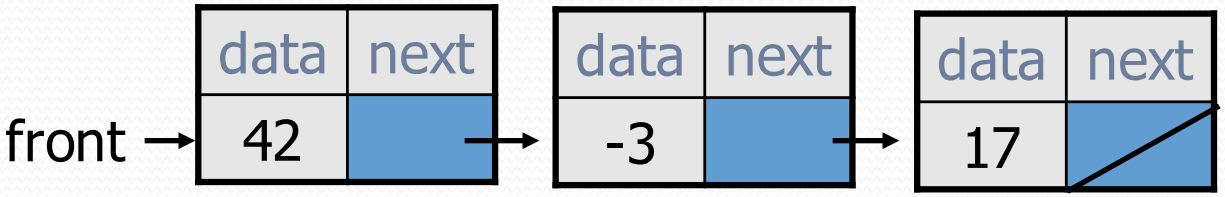
- Answer: Java has single inheritance.
  - can extend only one superclass
  - can implement many interfaces
  - Having interfaces allows a class to be part of a hierarchy (polymorphism) without using up its inheritance relationship.

Our list classes

- We have implemented the following two list collection classes:
  - `ArrayIntList`



  - `LinkedIntList`



  - Problems:
    - We should be able to treat them the same way in client code.
    - Linked list carries around a clunky extra node class.
    - They can store only `int` elements, not any type of value.
    - Some of their methods are implemented the same way (redundancy).
    - **It is inefficient to get or remove elements of a linked list.**