Example: FIFO

- To show how simple data structures are built without pointers, we'll build a doubly-linked list
  - ListItem class has some user data
  - first refers to that ListItem object at the front of the queue
  - last refers to the object at the end of the queue, i.e. most recently added

```java
public class ListItem {
    // In file ListItem.java
    public Object x; // N.B. a heterogeneous queue
    public ListItem previous;
    public ListItem next;

    // Constructor operation takes initial value
    public ListItem(String val) {
        this.x = val;
        this.previous = null;
        this.next = null;
    }

    public boolean equals(ListItem c) {
        return (x.equals(c.x) && (previous == c.previous) &&
                (next == c.next));
    }

    public void printItem() {
        System.out.println(x);
    }
}
```

```java
import java.applet.*; // overview of fifo.java
public class fifo extends Applet {
    private int count = 0;
    public ListItem first = null; // first is the next item to be removed
    public ListItem last = null; // last is the item most recently added

    public void init() {
        System.out.println("isEmpty returns " + isEmpty());
        putQueue("node 1");
        ...
        getQueue().printItem();
        ...
    }

    public boolean isEmpty() {
        ...
    }

    public void putQueue(String value) {
        ...
    }

    public ListItem getQueue() {
        ...
    }
}
```
// Called to initialize and test the applet.
public void init() {
    System.out.println("isEmpty returns "+isEmpty());
    putQueue("node 1");
    System.out.println("First node is "); first.printItem();
    System.out.println("Last node is "); last.printItem();
    putQueue("node 2");
    System.out.println("First node is "); first.printItem();
    System.out.println("Last node is "); last.printItem();
    getQueue().printItem();
    System.out.println("First node is "); first.printItem();
    System.out.println("Last node is "); last.printItem();
    getQueue().printItem();
    System.out.println("isEmpty returns "+isEmpty());
}

// See if the queue is empty
public boolean isEmpty() {
    return (count == 0);
}

// Add an item to the queue
public void putQueue(String value) {
    ListItem newItem = new ListItem(value);
    if ( isEmpty() ) { // Special case of empty queue
        first = last = newItem;
    } else {
        // next is the next item in the queue
        // previous is the item (if any) that was in the
        // queue right ahead of this (current) item
        last.next = newItem;
        newItem.previous = last;
        last = newItem;
    }
    count++;
}

// Get the first item off the front of the queue
public ListItem getQueue() {
    ListItem firstItem = first;
    // Make sure the queue isn't empty
    if (isEmpty() ) {
        System.out.println("Called getQueue on an empty queue");
    } else {
        this.first = firstItem.next;
        // Did we just remove the only item in the queue?
        if (first == null) {
            last = null;
        } else {
            first.previous = null;
            count--;
        }
    }
    return firstItem;
}

---

Programming by Contract

- A paradigm first introduced by Bertrand Meyer, the creator of the OO programming language Eiffel.
- Eiffel has built-in support for programming by contract, but most of the concepts can be used in any language.
- Idea: create a contract between the software developer (supplier) and software user (consumer)
  - Methods should start with a precondition that must be satisfied by the consumer of the routine.
  - And end with postconditions which the supplier guarantees to be true (if and only if the preconditions were met).
  - Each class has an invariant which must be satisfied after any changes to the object represented by the class, i.e., the invariant guarantees the object is in a valid state.
- Benefits: a good way to document requirements that can also be checked by the program. Saves lots of debugging.
Programming by Contract

- Note that the integer variable `count`, and `first` and `last` (both of type `ListItem`, are redundant in that
  - `first` and `last` are null iff `count == 0`
  - `first == last`, but both not null iff `count == 1`
  - otherwise `first != last` iff `count > 1`
- Java has no assert macro, but we can test and throw an exception.

```java
// See if the queue is empty
// Check consistency of count, first and last
// Note that exceptions are first-class objects
class CorruptFifoException extends Exception;
...
public boolean isEmpty() {
    if (count == 0) {
        if (first == null && last == null) {
            return (true);
        } else {
            throw new CorruptFifoException("first and last should be null");
        }
    } else {    // count != 0
        ...
    }
}
```

Single Inheritance, but

- A class may extend only one class, but it may implement many others
- A subclass inherits the variables and methods of its superclass(es), but may override them
- Overrides the methods defined in the class(es) it implements, as in upcoming thread example

Classes and Interfaces

- The methods of an abstract class are implemented elsewhere
- A final class cannot be extended
- Instances of a synchronizable class can be arguments of a synchronize block
  - Which means that access to “critical sections” is restricted
Interfaces

- Java does not allow “multiple inheritance” because it introduces problems as well as benefits. Fortunately,
- Java allows you to impose requirements on a class from multiple class-like interfaces.
- An interface is like an abstract class in that it can hold abstract method definitions that force other classes to implement ordinary methods.
- But it is also different:
  - An interface does NOT have instance variables (but it can have constants)
  - All methods in an interface are abstract (they each have a name, parameters, and a return type, but no implementation)
  - All methods in an interface are automatically public.

Classes vs. Interfaces

- A class definition that implements an interface must define all the methods specified in that interface. In this respect, an interface is like an abstract class.
- An interface differs from an abstract class, however, in several respects:
  - An interface only imposes definition requirements; interfaces do not supply definitions.
  - A class extends exactly one superclass; a class can implement an unlimited number of interfaces.
  - Thus, the purpose of the interface is strictly to impose requirements via its abstract methods; there are no method implementations:

Interfaces

- Interfaces provide no mechanism for enforcing method specifications, other than method signatures
  - you are free to deposit descriptive comments in an interface, however.
- Interfaces are excellent places for descriptive comments for two reasons:
  - Interfaces, unlike class definitions, are free of clutter from implementing code.
  - Programmers look to interfaces for method and class documentation.

Interfaces

- The interface mechanism is an enormously important aid to good programming practice.
- Interfaces allow you to shift to the Java compiler a requirement-managing responsibility
  - that otherwise would engage your own, human attention.
  - Interfaces encourage you to document your classes by acting, by convention, as documentation centers.
Interfaces Example

- java.lang defines a Comparable interface as:
  ```java
  public interface Comparable {
      int compareTo(Object other); // no implementation
  }
  ```
- If you want an interface to impose requirements on a particular class, don’t extend it; instead implement it:
  ```java
  public class someClassName implements I1, I2 {
      …
  }
  ```
- ```java
  public class Movie3 extends Attraction implements Comparable {
      public int compareTo(Object otherMovie) {
          Movie3 other = (Movie3) otherMovie;
          if (rating() < other.rating()) return -1;
          else if (rating() > other.rating())
              return 1;
          else return 0;
      }
  }
  ```

Exceptions

- If an error does occur, that error is said to be exceptional behavior that throws an exception.
- Whenever an expression has the potential to throw an exception, you can embed that expression in a try–catch statement, in which you specify explicitly what Java is to do when an exception actually is thrown.
- Exceptions are objects in their own right
  - They can be generated, caught and handled under program control
  - Examples: IOException, ArithmeticException, etc.

try/catch/finally

- Associates a set of statements with one or more exceptions and some handling code
  ```java
  try {
      Thread.sleep(200);
  }
  catch (InterruptedException e) {
      System.out.println(e);
  }
  finally {
      System.out.println("Wakeup");
  }
  ```

Exceptions

- Java will “throw an exception” when unusual conditions arise during execution of programs, e.g.,
  - E.g., Attempt to divide an integer by zero
- To handle the exception, use the following:
  ```java
  try [statement with potential to throw exception]
  catch [exception-class-name parameter] {
      exception-handling-code
  }
  ```
- To catch I/O exceptions, use:
  - FileNotFoundException or IOException class.
Exceptions

• Suppose, for example, that you want to open a file for reading using a FileInputStream instance.
• You can acknowledge that the attempt may throw an exception by embedding the reading expressions in a block following the try keyword.
• Java stops executing statements in the try block as soon as an exception is thrown:

```java
try {
  ...  <-- An attempt to attach a stream to a file occurs here
}
```

• You specify what to do in the event that the exception is an instance of the IOException class by writing the keyword catch, followed by a parameter typed by IOException, surrounded by parentheses, followed by another block:

```java
catch (IOException e) {
  ...
}
```

• To shut a program down, use System.exit(0);
• To have a block of statements executed after a try (whether or not an exception was thrown) use:

```java
finally {
  clean-up statements
}
```

• You can create (and throw) your own exceptions, e.g.,

```java
public class StrangeNewException extends Exception {
  throw (new StrangeNewException());
}
catch (StrangeNewException e) {
  ...
}
```

• Alternative method to handle exceptions:

```java
public static void f(params) throws Exception-class {
  ...
}
```