Today:

- A Brief Side Trip: Enumeration types.
  - Threads
  - Communication between threads
  - Synchronization
  - Mailboxes
Side Trip into Java: Enumeration Types

• Problem: Need a type to represent something that has a few, named, discrete values.

• In the purest form, the only necessary operations are == and !=; the only property of a value of the type is that it differs from all others.

• In older versions of Java, used named integer constants:

```java
interface Pieces {
    int BLACK_PIECE = 0, // Fields in interfaces are static final.
    BLACK_KING = 1,
    WHITE_PIECE = 2,
    WHITE_KING = 3,
    EMPTY = 4;
}
```

• C and C++ provide enumeration types as a shorthand, with syntax like this:

```java
enum Piece { BLACK_PIECE, BLACK_KING, WHITE_PIECE, WHITE_KING, EMPTY }
```

• But since all these values are basically ints, accidents can happen.
Enum Types in Java

- New version of Java allows syntax like that of C or C++, but with more guarantees:

  ```java
  public enum Piece {
    BLACK PIECE, BLACK KING, WHITE PIECE, WHITE KING, EMPTY
  }
  ```

- Defines `Piece` as a new reference type, a special kind of class type.
- The names `BLACK PIECE`, etc., are static, final `enumeration constants` (or `enumerals`) of type `PIECE`.
- They are automatically initialized, and are the only values of the enumeration type that exist (illegal to use `new` to create an enum value.)
- Can safely use `==`, and also `switch` statements:

  ```java
  boolean isKing(Piece p) {
    switch (p) {
      case BLACK KING: case WHITE KING: return true;
      default: return false;
    }
  }
  ```
Making Enumerals Available Elsewhere

- Enumerals like BLACK PIECE are static members of a class, not classes.
- Therefore, unlike C or C++, their declarations are not automatically visible outside the enumeration class definition.
- So, in other classes, must write Piece.BLACK_PIECE, which can get annoying.
- However, with version 1.5, Java has static imports: to import all static definitions of class checkers.Piece (including enumerals), you write

  import static checkers.Piece.*;

  among the import clauses.
- Alas, cannot use this for enum classes in the anonymous package.
Operations on Enum Types

- **Order of declaration of enumeration constants significant**: `.ordinal()` gives the position (numbering from 0) of an enumeration value. Thus, `Piece.BLACK_KING.ordinal()` is 1.

- **The array** `Piece.values()` **gives all the possible values of the type**. Thus, you can write:

  ```java
  for (Piece p : Piece.values())
      System.out.printf("Piece value #\%d is %s\n", p.ordinal(), p);
  ```

- **The static function** `Piece.valueOf` **converts a String into a value of type** `Piece`. So `Piece.valueOf("EMPTY") == EMPTY`. 
Fancy Enum Types

- Enums are classes. You can define all the extra fields, methods, and constructors you want.

- Constructors are used only in creating enumeration constants. The constructor arguments follow the constant name:

```java
enum Piece {
    BLACK_PIECE(BLACK, false, "b"), BLACK_KING(BLACK, true, "B"),
    WHITE_PIECE(WHITE, false, "w"), WHITE_KING(WHITE, true, "W"),
    EMPTY(null, false, " ");

    private final Side color;
    private final boolean isKing;
    private final String textName;

    Piece(Side color, boolean isKing, String textName) {
        this.color = color; this.isKing = isKing; this.textName = textName;
    }

    Side color() { return color; }
    boolean isKing() { return isKing; }
    String textName() { return textName; }
}
```
Threads

- So far, all our programs consist of single sequence of instructions.
- Each such sequence is called a thread (for “thread of control”) in Java.
- Java supports programs containing *multiple* threads, which (conceptually) run concurrently.
- Actually, on a uniprocessor, only one thread at a time actually runs, while others wait, but this is largely invisible.
- To allow program access to threads, Java provides the type `Thread` in `java.lang`. Each `Thread` contains information about, and controls, one thread.
- Simultaneous access to data from two threads can cause chaos, so are also constructs for controlled communication, allowing threads to `lock` objects, to `wait` to be notified of events, and to `interrupt` other threads.
But Why?

• Typical Java programs always have > 1 thread: besides the main program, others clean up garbage objects, receive signals, update the display, other stuff.

• When programs deal with asynchronous events, is sometimes convenient to organize into subprograms, one for each independent, related sequence of events.

• Threads allow us to insulate one such subprogram from another.

• GUIs often organized like this: application is doing some computation or I/O, another thread waits for mouse clicks (like ‘Stop’), another pays attention to updating the screen as needed.

• Large servers like search engines may be organized this way, with one thread per request.

• And, of course, sometimes we do have a real multiprocessor.
Java Mechanics

• To specify the actions “walking” and “chewing gum”:

```java
class Chewer1 implements Runnable {
    public void run()
    {
      while (true) ChewGum();
    }
}
class Walker1 implements Runnable {
    public void run()
    {
      while (true) Walk();
    }
}
```

```java
// Walk and chew gum
Thread chomp = new Thread(new Chewer1());
Thread clomp = new Thread(new Walker1());
chomp.start(); clomp.start();
```

• Concise Alternative (uses fact that Thread implements Runnable):

```java
class Chewer2 extends Thread {
    public void run()
    {
      while (true) ChewGum();
    }
}
class Walker2 extends Thread {
    public void run()
    {
      while (true) Walk();
    }
}
```

```java
Thread chomp = new Chewer2(),
"            clomp = new Walker2();
chomp.start();
"            clomp.start();
```
Avoiding Interference

• When one thread has data for another, one must wait for the other to be ready.

• Likewise, if two threads use the same data structure, generally only one should modify it at a time; other must wait.

• E.g., what would happen if two threads simultaneously inserted an item into a linked list at the same point in the list?

• A: Both could conceivably execute

```java
p.next = new ListCell(x, p.next);
```

with the same values of \( p \) and \( p.next \); one insertion is lost.

• Can arrange for only one thread at a time to execute a method on a particular object with either of the following equivalent definitions:

```java
void f(...) {
    synchronized (this) {
        body of f
    }
}
```

```java
synchronized void f(...) {
    body of f
}
```
Communicating the Hard Way

• **Communicating data is tricky**: the faster party must wait for the slower.

• **Obvious approaches for sending data from thread to thread don't work**:

```java
class DataExchanger {
    Object value = null;
    Object receive() {
        Object r; r = null;
        while (r == null) {
            r = value;
        }
        value = null;
        return r;
    }
    void deposit(Object data) {
        while (value != null) {
        }
        value = data;
    }
}
```

DataExchanger exchanger = new DataExchanger();

```java
// thread1 sends to thread2 with
exchanger.deposit("Hello!");
```

```java
// thread2 receives from thread1 with
msg = (String) exchanger.receive();
```

• **BAD**: One thread can monopolize machine while waiting; two threads executing `deposit` or `receive` simultaneously cause chaos.
Primitive Java Facilities

- The `wait` method on `Object` makes thread wait (not using processor) until notified by `notifyAll`, unlocking the `Object` while it waits.

- Example, `ucb.util.mailbox` has something like this (simplified):

```java
interface Mailbox {
  void deposit(Object msg) throws InterruptedException;
  Object receive() throws InterruptedException;
}

class QueuedMailbox implements Mailbox {
  private List<Object> queue = new LinkedList<Object>();

  public synchronized void deposit(Object msg) {
    queue.add(msg);
    this.notifyAll(); // Wake any waiting receivers
  }

  public synchronized Object receive() throws InterruptedException {
    while (queue.isEmpty()) wait();
    return queue.remove(0);
  }
}
```
Message-Passing Style

- Use of Java primitives very error-prone. Wait until CS162.
- Mailboxes are higher-level, and allow the following program structure:

```
while (!gameOver()) {
  if (myMove())
    outBox.deposit(computeMyMove(lastMove));
  else
    lastMove = inBox.receive();
}
```

- Where each Player is a thread that looks like this:
More Concurrency

- Previous example can be done other ways, but mechanism is very flexible.
- E.g., suppose you want to think during opponent’s move:

```java
while (!gameOver()) {
    if (myMove())
        outBox.deposit(computeMyMove(lastMove));
    else {
        do {
            thinkAheadALittle();
            lastMove = inBox.receiveIfPossible();
        } while (lastMove == null);
    }
}
```

- `receiveIfPossible` (written `receive(0) in our actual package`) doesn’t wait; returns null if no message yet, perhaps like this:

```java
public synchronized Object receiveIfPossible() throws InterruptedException {
    if (queue.isEmpty())
        return null;
    return queue.remove(0);
}
```
Coroutines

- A coroutine is a kind of synchronous thread that explicitly hands off control to other coroutines so that only one executes at a time, like Python generators. Can get similar effect with threads and mailboxes.

- Example: recursive inorder tree iterator:

```java
class TreeIterator extends Thread {
    Tree root; Mailbox r;
    TreeIterator(Tree T, Mailbox r) {
        this.root = T; this.dest = r;
    }
    public void run() {
        traverse(root);
        r.deposit(End marker);
    }
    void traverse(Tree t) {
        if (t == null) return;
        traverse(t.left);
        r.deposit(t.label);
        traverse(t.right);
    }
}
void treeProcessor(Tree T) {
    Mailbox m = new QueuedMailbox();
    new TreeIterator(T, m).start();
    while (true) {
        Object x = m.receive();
        if (x is end marker)
            break;
        do something with x;
    }
```

Last modified: Fri Nov 17 01:28:46 2017
Use In GUIs

- Jave runtime library uses a special thread that does nothing but wait for events like mouse clicks, pressed keys, mouse movement, etc.

- You can designate an object of your choice as a listener; which means that Java’s event thread calls a method of that object whenever an event occurs.

- As a result, your program can do work while the GUI continues to respond to buttons, menus, etc.

- Another special thread does all the drawing. You don’t have to be aware when this takes place; just ask that the thread wake up whenever you change something.
Highlights of a GUI Component

/** A widget that draws multi-colored lines indicated by mouse. */
class Lines extends JComponent implements MouseListener {
    private List<Point> lines = new ArrayList<Point>();

    Lines() { // Main thread calls this to create one
        setPreferredSize(new Dimension(400, 400));
        addMouseListener(this);
    }

    public synchronized void paintComponent(Graphics g) { // Paint thread
        g.setColor(Color.white); g.fillRect(0, 0, 400, 400);
        int x, y; x = y = 200;
        Color c = Color.black;
        for (Point p : lines)
            g.setColor(c); c = chooseNextColor(c);
            g.drawLine(x, y, p.x, p.y); x = p.x; y = p.y;
    }

    public synchronized void mouseClicked(MouseEvent e) { // Event thread
        lines.add(new Point(e.getX(), e.getY())); repaint();
    }
    ...
}
Interrupts

- An interrupt is an event that disrupts the normal flow of control of a program.

- In many systems, interrupts can be totally asynchronous, occurring at arbitrary points in a program, the Java developers considered this unwise; arranged that interrupts would occur only at controlled points.

- In Java programs, one thread can interrupt another to inform it that something unusual needs attention:

  ```java
  otherThread.interrupt();
  ```

- But `otherThread` does not receive the interrupt until it waits: methods `wait`, `sleep` (wait for a period of time), `join` (wait for thread to terminate), and mailbox deposit and receive.

- Interrupt causes these methods to throw `InterruptedException`, so typical use is like this:

  ```java
  try {
    msg = inBox.receive();
  } catch (InterruptedException e) { HandleEmergency(); }
  ```
Remote Mailboxes (A Side Excursion)

• RMI: Remote Method Interface allows one program to refer to objects in another program.

• We use it to allow mailboxes in one program be received from or deposited into in another.

• To use this, you define an interface to the remote object:

```java
import java.rmi.*;
interface Mailbox extends Remote {
    void deposit(Object msg)
        throws InterruptedException, RemoteException;
    Object receive()
        throws InterruptedException, RemoteException;
    ...
}
```

• On machine that actually will contain the object, you define

```java
class QueuedMailbox ... implements Mailbox {
    Same implementation as before, roughly
}
```
Remote Objects Under the Hood

// On machine #1:  // On Machine #2:
Mailbox outBox    = new QueuedMailbox();    Mailbox inBox
                   = get outBox from machine #1

- Because Mailbox is an interface, hides fact that on Machine #2 doesn't actually have direct access to it.
- Requests for method calls are relayed by I/O to machine that has real object.
- Any argument or return type OK if it also implements Remote or can be serialized—turned into stream of bytes and back, as can primitive types and String.
- Because I/O involved, expect failures, hence every method can throw RemoteException (subtype of IOException).