

# A Third Look At Java

# A Little Demo

```
public class Test {
    public static void main(String[] args) {
        int i = Integer.parseInt(args[0]);
        int j = Integer.parseInt(args[1]);
        System.out.println(i/j);
    }
}
```

```
> javac Test.java
> java Test 6 3
2
>
```

# Exceptions

```
> java Test
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 0
    at Test.main(Test.java:3)
> java Test 6 0
Exception in thread "main"
java.lang.ArithmeticException: / by zero
    at Test.main(Test.java:4)
```

In early languages, that's all that happened: error message, core dump, terminate.

Modern languages like Java support *exception handling*.

# Outline

- 17.2 Throwable classes
- 17.3 Catching exceptions
- 17.4 Throwing exceptions
- 17.5 Checked exceptions
- 17.6 Error handling
- 17.7 Finally
- 17.8 Farewell to Java

# Some Predefined Exceptions

Java Exception	Code to Cause It
<b>NullPointerException</b>	<pre>String s = null; s.length();</pre>
<b>ArithmeticException</b>	<pre>int a = 3; int b = 0; int q = a/b;</pre>
<b>ArrayIndexOutOfBoundsException</b>	<pre>int[] a = new int[10]; a[10];</pre>
<b>ClassCastException</b>	<pre>Object x =     new Integer(1); String s = (String) x;</pre>
<b>StringIndexOutOfBoundsException</b>	<pre>String s = "Hello"; s.charAt(5);</pre>

# An Exception Is An Object

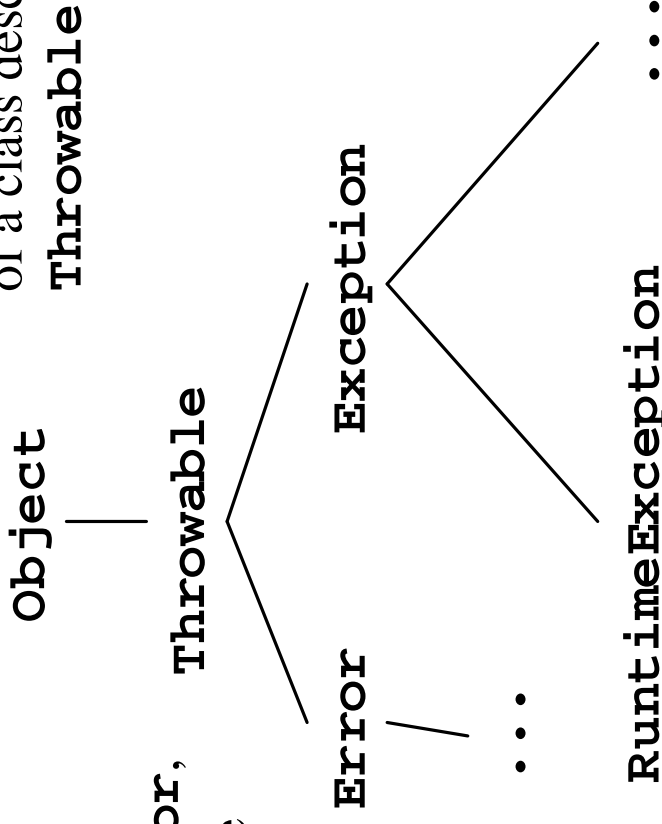
- The names of exceptions are class names, like `NullPointerException`
- Exceptions are objects of those classes
- In the previous examples, the Java language system automatically creates an object of an exception class and *throws* it
- If the program does not *catch* it, it terminates with an error message

# Throwable Classes

- To be thrown as an exception, an object must be of a class that inherits from the predefined class **Throwable**
- There are four important predefined classes in that part of the class hierarchy:
  - **Throwable**
  - **Error**
  - **Exception**
  - **RuntimeException**

Classes derived from **Error** are used for serious, system-generated errors, like **OutOfMemoryError**, that usually cannot be recovered from

Java will only throw objects of a class descended from **Throwable**



Classes derived from **RuntimeException** are used for ordinary system-generated errors, like **ArithmeticException**

Classes derived from **Exception** are used for ordinary errors that a program might want to catch and recover from

# Outline

- 17.2 Throwable classes
- 17.3 Catching exceptions
- 17.4 Throwing exceptions
- 17.5 Checked exceptions
- 17.6 Error handling
- 17.7 Finally
- 17.8 Farewell to Java

# The **try** Statement

```
<try-statement> ::= <try-part> <catch-part>  
<try-part> ::= try <compound-statement>  
<catch-part> ::= catch ( <type> <variable-name> )  
                  <compound-statement>
```

- Simplified... full syntax later
- The *<type>* is a throwable class name
- Does the **try** part
- Does the **catch** part only if the **try** part throws an exception of the given *<type>*

# Example

```
public class Test {
    public static void main(String[] args) {
        try {
            int i = Integer.parseInt(args[0]);
            int j = Integer.parseInt(args[1]);
            System.out.println(i/j);
        }
        catch (ArithmeticException a) {
            System.out.println("You're dividing by zero!");
        }
    }
}
```

This will catch and handle any **ArithmeticException**. Other exceptions will still get the language system's default behavior.

# Example

```
> java Test 6 3
2
> java Test 6 0
You're dividing by zero!
> java Test
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 0
    at Test.main(Test.java:3)
```

- Catch type chooses exceptions to catch:
  - **ArithmeticException** got zero division
  - **RuntimeException** would get both examples above
  - **Throwable** would get all possible exceptions

## After The **try** Statement

- A **try** statement can be just another in a sequence of statements
- If no exception occurs in the **try** part, the **catch** part is not executed
- If no exception occurs in the **try** part, or if there is an exception which is caught in the **catch** part, execution continues with the statement following the **try** statement

# Exception Handled

```
System.out.print("1, ");  
try {  
    String s = null;  
    s.length();  
}  
catch (NullPointerException e) {  
    System.out.print("2, ");  
}  
System.out.println("3");
```

This just prints the line

**1, 2, 3**

# Throw From Called Method

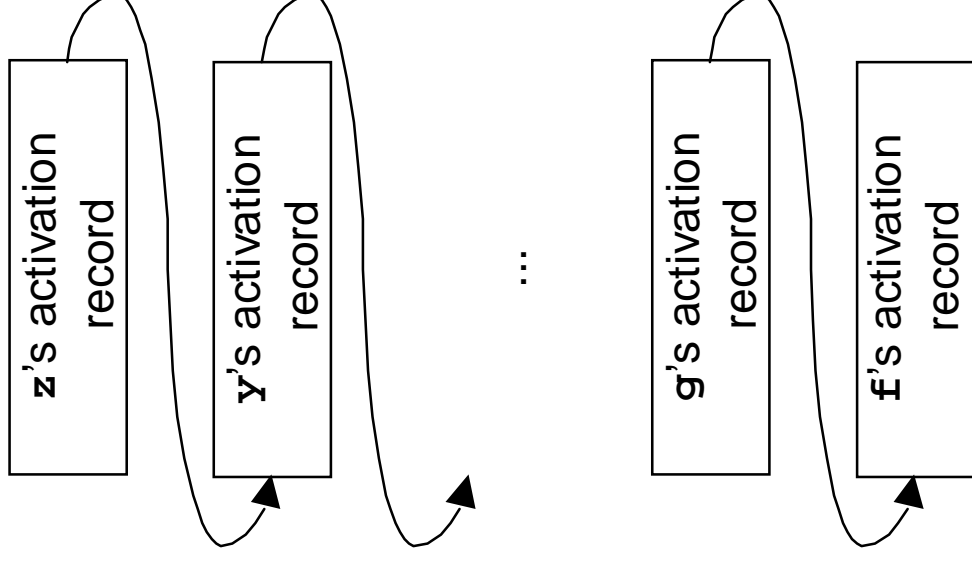
- The **try** statement gets a chance to catch exceptions thrown while the **try** part runs
- That includes exceptions thrown by methods called from the **try** part

# Example

```
void f() {  
    try {  
        g();  
    }  
    catch (ArithmeticException a) {  
        ...  
    }  
}
```

- If `g` throws an `ArithmeticException`, that it does not catch, `f` will get it
- In general, the throw and the catch can be separated by any number of method invocations

- If **z** throws an exception it does not catch, **z**'s activation stops...
- ...then **y** gets a chance to catch it; if it doesn't, **y**'s activation stops...
- ...and so on all the way back to **f**



# Long-Distance Throws

- That kind of long-distance throw is one of the big advantages of exception handling
- All intermediate activations between the throw and the catch are stopped and popped
- If not throwing or catching, they need not know anything about it

# Multiple `catch` Parts

```
<try-statement> ::= <try-part> <catch-parts>  
<try-part> ::= try <compound-statement>  
<catch-parts> ::= <catch-part> <catch-parts>  
                | <catch-part>  
<catch-part> ::= catch ( <type> <variable-name> )  
                  <compound-statement>
```

- To catch more than one kind of exception, a **catch** part can specify some general superclass like **RuntimeException**
- But usually, to handle different kinds of exceptions differently, you use multiple **catch** parts

# Example

```
public static void main(String[] args) {
    try {
        int i = Integer.parseInt(args[0]);
        int j = Integer.parseInt(args[1]);
        System.out.println(i/j);
    }
    catch (ArithmeticException a) {
        System.out.println("You're dividing by zero!");
    }
    catch (ArrayIndexOutOfBoundsException a) {
        System.out.println("Requires two parameters.");
    }
}

    This will catch and handle both ArithmeticException
    and ArrayIndexOutOfBoundsException
```

# Example

```
public static void main(String[] args) {
    try {
        int i = Integer.parseInt(args[0]);
        int j = Integer.parseInt(args[1]);
        System.out.println(i/j);
    }
    catch (ArithmeticException a) {
        System.out.println("You're dividing by zero!");
    }
    catch (ArrayIndexOutOfBoundsException a) {
        System.out.println("Requires two parameters.");
    }
    catch (RuntimeException a) {
        System.out.println("Runtime exception.");
    }
}
```

# Overlapping Catch Parts

- If an exception from the **try** part matches more than one of the **catch** parts, only the first matching **catch** part is executed
- A common pattern: **catch** parts for specific cases first, and a more general one at the end
- Note that Java does not allow unreachable **catch** parts, or unreachable code in general

# Outline

- 17.2 Throwable classes
- 17.3 Catching exceptions
- 17.4 Throwing exceptions
- 17.5 Checked exceptions
- 17.6 Error handling
- 17.7 Finally
- 17.8 Farewell to Java

# The **throw** Statement

```
<throw-statement> ::= throw <expression> ;
```

- Most exceptions are thrown automatically by the language system
- Sometimes you want to throw your own
- The *<expression>* is a reference to a throwable object—usually, a new one:

```
throw new NullPointerException();
```

# Custom Throwable Classes

```
public class OutOfGas extends Exception {  
}  
  
system.out.print("1, ");  
try {  
    throw new OutOfGas();  
}  
catch (OutOfGas e) {  
    system.out.print("2, ");  
}  
system.out.println("3");
```

# Using The Exception Object

- The exception that was thrown is available in the catch block—as that parameter
- It can be used to communicate information from the thrower to the catcher
- All classes derived from **Throwable** inherit a method **printStackTrace**
- They also inherit a **String** field with a detailed error message, and a **getMessage** method to access it

# Example

```
public class OutOfGas extends Exception {  
    public OutOfGas(String details) {  
        super(details);  
    }  
}
```

This calls a base-class constructor to initialize the field returned by `getMessage()`.

```
try {  
    throw new OutOfGas("You have run out of gas.");  
} catch (OutOfGas e) {  
    System.out.println(e.getMessage());  
}
```

# About **super** In Constructors

- The first statement in a constructor can be a call to **super** (with parameters, if needed)
- That calls a base class constructor
- Used to initialize inherited fields
- All constructors (except in **Object**) start with a call to another constructor—if you don't include one, Java calls **super()** implicitly

# More About Constructors

- Also, all classes have at least one constructor—if you don't include one, Java provides a no-arg constructor implicitly

```
public class OutOfGas extends Exception {  
}
```

```
public class OutOfGas extends Exception {  
    public OutOfGas() {  
        super();  
    }  
}
```

These are equivalent!

```

public class OutOfGas extends Exception {
    private int miles;
    public OutOfGas(String details, int m) {
        super(details);
        miles = m;
    }
    public int getMiles() {
        return miles;
    }
}

try {
    throw new OutOfGas("You have run out of gas.",19);
}
catch (OutOfGas e) {
    System.out.println(e.getMessage());
    System.out.println("Odometer: " + e.getMiles());
}
}

```

# Outline

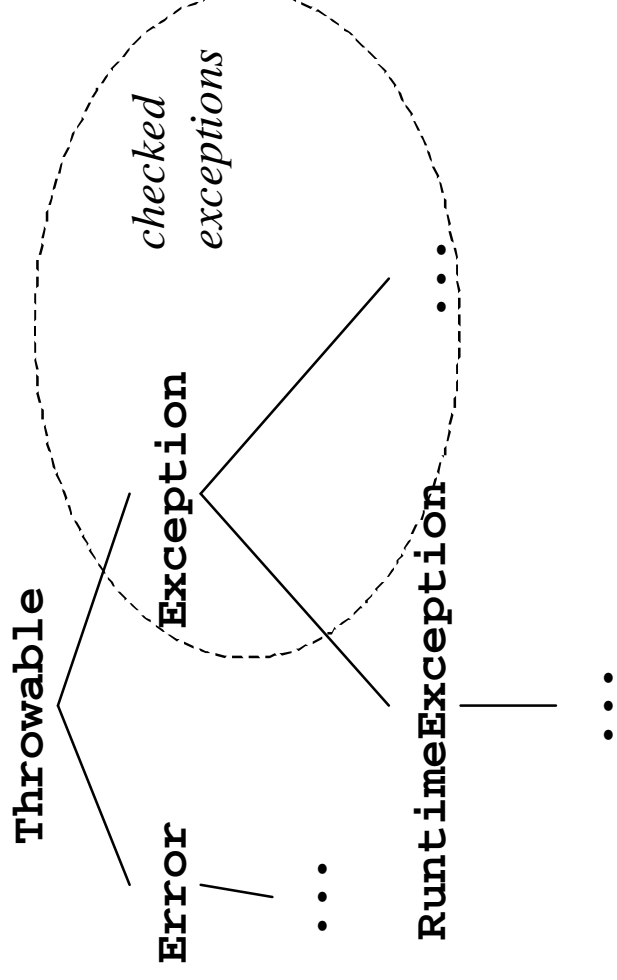
- 17.2 Throwable classes
- 17.3 Catching exceptions
- 17.4 Throwing exceptions
- 17.5 Checked exceptions
- 17.6 Error handling
- 17.7 Finally
- 17.8 Farewell to Java

# Checked Exceptions

```
void z() {  
    throw new OutOfGas("You have run out of gas.", 19");  
}
```

- This method will not compile: “The exception **OutOfGas** is not handled”
- Java has not complained about this in our previous examples—why now?
- Java distinguishes between two kinds of exceptions: checked and unchecked

# Checked Exceptions



The checked exception classes are **Exception** and its descendants, excluding **RuntimeException** and its descendants

# What Gets Checked?

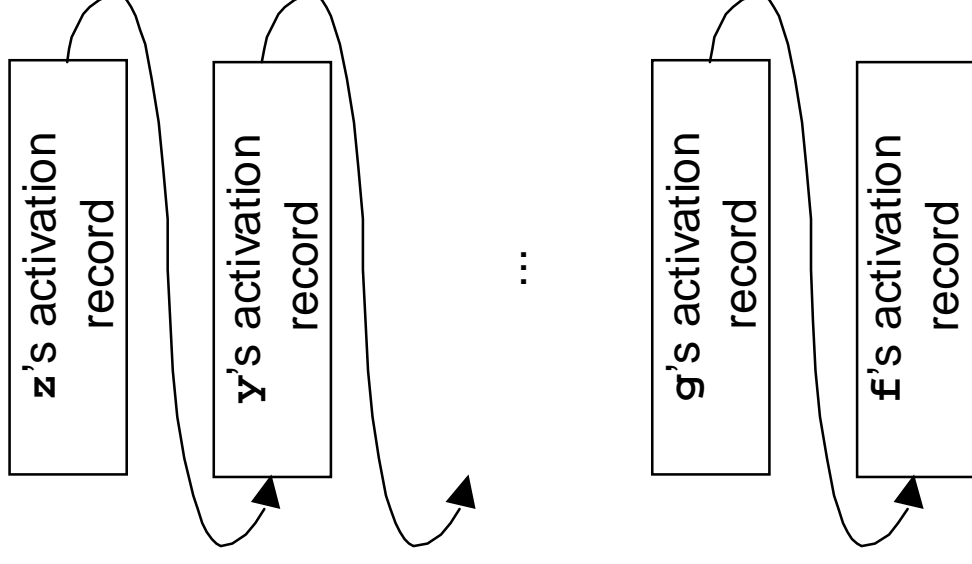
- A method that can get a checked exception is not permitted to ignore it
- It can catch it
  - That is, the code that generates the exception can be inside a **try** statement with a **catch** part for that checked exception
- Or, it can declare that it does *not* catch it
  - Using a **throws** clause

# The Throws Clause

```
void z() throws OutOfGas {  
    throw new OutOfGas("You have run out of gas.", 19);  
}
```

- A **throws** clause lists one or more throwable classes separated by commas
- This one always throws, but in general, the throws clause means *might* throw
- So any caller of **z** must catch **OutOfGas**, or place it in its own **throws** clause

- If **z** declares that it **throws OutOfGas**...
- ...then **y** must catch it, or declare it **throws** it too...
- ...and so on all the way back to **f**



# Why Use Checked Exceptions

- The **throws** clause is like documentation: it tells the reader that this exception can result from a call of this method
- But it is *verified* documentation; if any checked exception can result from a method call, the compiler will insist it be declared
- This can make programs easier to read and more likely to be correct

# How To Avoid Checked Exceptions

- You can always define your own exceptions using a different base class, such as **Error** or **Throwable**
- Then they will be unchecked
- Weigh the advantages carefully

# Outline

- 17.2 Throwable classes
- 17.3 Catching exceptions
- 17.4 Throwing exceptions
- 17.5 Checked exceptions
- 17.6 Error handling
- 17.7 Finally
- 17.8 Farewell to Java

# Handling Errors

- Example: popping an empty stack
- Techniques:
  - Preconditions only
  - Total definition
  - Fatal errors
  - Error flagging
  - Using exceptions

# Preconditions Only

- Document preconditions necessary to avoid errors
- Caller must ensure these are met, or explicitly check if not sure

```

/**
 * Pop the top int from this stack and return it.
 * This should be called only if the stack is
 * not empty.
 * @return the popped int
 */
public int pop() {
    Node n = top;
    top = n.getLink();
    return n.getData();
}

    if (s.hasMore()) x = s.pop();
    else ...

```

# Drawbacks

- If the caller makes a mistake, and pops an empty stack: **NullPointerException**
  - If that is uncaught, program crashes with an unhelpful error message
  - If caught, program relies on undocumented internals; an implementation using an array would cause a different exception

# Total Definition

- We can change the definition of `pop` so that it always works
- Define some standard behavior for popping an empty stack
- Like character-by-character file I/O in C: an EOF character at the end of the file
- Like IEEE floating-point: NaN and signed infinity results

```
/**
 * Pop the top int from this stack and return it.
 * If the stack is empty we return 0 and leave the
 * stack empty.
 * @return the popped int, or 0 if the stack is empty
 */
public int pop() {
    Node n = top;
    if (n==null) return 0;
    top = n.getLink();
    return n.getData();
}
```

# Drawbacks

- Can mask important problems
- If a client pops more than it pushes, this is probably a serious bug that should be detected and fixed, not concealed

# Fatal Errors

- The old-fashioned approach: just crash!
- Preconditions, plus decisive action
- At least this does not conceal the problem...

```

/**
 * Pop the top int from this stack and return it.
 * This should be called only if the stack is
 * not empty. If called when the stack is empty,
 * we print an error message and exit the program.
 * @return the popped int
 */
public int pop() {
    Node n = top;
    if (n==null) {
        system.out.println("Popping an empty stack!");
        system.exit(-1);
    }
    top = n.getLink();
    return n.getData();
}

```

# Drawbacks

- Not an object-oriented style: an object should do things to itself, not to the rest of the program
- Inflexible: different clients may want to handle the error differently
  - Terminate
  - Clean up and terminate
  - Repair the error and continue
  - Ignore the error
  - Etc.

# Error Flagging

- The method that detects the error can flag it somehow
  - By returning a special value (like C `malloc`)
  - By setting a global variable (like C `errno`)
  - By setting an instance variable to be checked by a method call (like C `error(f)`)
- Caller must explicitly test for error

```

/**
 * Pop the top int from this stack and return it.
 * This should be called only if the stack is
 * not empty. If called when the stack is empty,
 * we set the error flag and return an undefined
 * value.
 * @return the popped int if stack not empty
 */
public int pop() {
    Node n = top;
    if (n==null) {
        error = true;
        return 0;
    }
    top = n.getLink();
    return n.getData();
}

```

```

/**
 * Return the error flag for this stack. The error
 * flag is set true if an empty stack is ever popped.
 * It can be reset to false by calling resetError().
 * @return the error flag
 */
public boolean getError() {
    return error;
}

/**
 * Reset the error flag. We set it to false.
 */
public void resetError() {
    error = false;
}

```

```

/**
 * Pop the two top integers from the stack, divide
 * them, and push their integer quotient. There
 * should be at least two integers on the stack
 * when we are called. If not, we leave the stack
 * empty and set the error flag.
 */
public void divide() {
    int i = pop();
    int j = pop();
    if (getError()) return;
    push(i/j);
}

```

The kind of explicit error check required by an error flagging technique.

Note that **divide**'s caller may also have to check it, and its caller, and so On...

# Using Exceptions

- The method that first finds the error throws an exception
- May be checked or unchecked
- Part of the documented behavior of the method

```
/**
 * Pop the top int from this stack and return it.
 * @return the popped int
 * @exception EmptyStack if stack is empty
 */
public int pop() throws EmptyStack {
    Node n = top;
    if (n==null) throw new EmptyStack();
    top = n.getLink();
    return n.getData();
}
```

```
/**
 * Pop the two top integers from the stack, divide
 * them, and push their integer quotient.
 * @exception EmptyStack if stack runs out
 */
public void divide() throws EmptyStack {
    int i = pop();
    int j = pop();
    push(i/j);
}
```

Caller makes no error check—just passes  
the exception along if one occurs

# Advantages

- Good error message even if uncaught
- Documented part of the interface
- Error caught right away, not masked
- Caller need not explicitly check for error
- Error can be ignored or handled flexibly

# Outline

- 17.2 Throwable classes
- 17.3 Catching exceptions
- 17.4 Throwing exceptions
- 17.5 Checked exceptions
- 17.6 Error handling
- 17.7 Finally
- 17.8 Farewell to Java

# The Full **try** Syntax

```
<try-statement> ::= <try-part> <catch-parts>  
                | <try-part> <catch-parts> <finally-part>  
                | <try-part> <finally-part>  
<try-part> ::= try <compound-statement>  
<catch-parts> ::= <catch-part> <catch-parts> | <catch-part>  
<catch-part> ::= catch ( <type> <variable-name> )  
                  <compound-statement>  
<finally-part> ::= finally <compound-statement>
```

- There is an optional **finally** part
- No matter what happens, the **finally** part is always executed at the end of the **try** statement

# Using `finally`

```
file.open();  
try {  
    workWith(file);  
}  
finally {  
    file.close();  
}
```

- The `finally` part is usually used for cleanup operations
- Whether or not there is an exception, the file is closed

# Example

```
System.out.print("1");  
try {  
    System.out.print("2");  
    if (true) throw new Exception();  
    System.out.print("3");  
}  
catch (Exception e) {  
    System.out.print("4");  
}  
finally {  
    System.out.print("5");  
}  
System.out.println("6");
```

What does this print?

What if we change

**new Exception()** to  
**new Throwable()**?

# Outline

- 17.2 Throwable classes
- 17.3 Catching exceptions
- 17.4 Throwing exceptions
- 17.5 Checked exceptions
- 17.6 Error handling
- 17.7 Finally
- 17.8 Farewell to Java

# Parts We Skipped

- Fundamentals
  - Primitive types: **byte**, **short**, **long**, **float**
  - Statements: **do**, **for**, **break**, **continue**, **switch**
- Refinements
  - Inner classes: **define classes in any scope:**  
**inside other classes, in blocks, in expressions**
  - **assert** statement

# More Parts We Skipped

- Packages
  - Classes are grouped into packages
  - In many Java systems, the source files in a directory correspond to a package
  - Default access (without **public**, **private** or **protected**) is package-wide
- Concurrency
  - Synchronization constructs for multiple threads
  - Parts of the API for creating threads

# More Parts We Skipped

- The vast API
  - containers (stacks, queues, hash tables, etc.)
  - graphical user interfaces
  - 2D and 3D graphics
  - math
  - pattern matching with regular expressions
  - file IO
  - network IO and XML
  - encryption and security
  - remote method invocation
  - interfacing to databases and other tools