Exceptions
No method can deal with every possible contingency.
Examples

- `compareTo(Object o)`
  - How can you compare instances of different classes? (Recall the contract!)

- `Integer.parseInt("skål");`
  - what `int` could this sensibly return?

- `new FileInputStream("bogus.txt");`
  - what if `bogus.txt` doesn’t exist?
Distinguished values

• One idea: use distinguished return values to identify error conditions

• For example, consider the getClicks() method in Clicker

• could return -1 to indicate Clicker malfunction
In many cases, the return value is an inadequate way to signal what went wrong.
In many cases, the return value is an inadequate way to signal what went wrong.

(Why?)
How do you distinguish between valid and error ("out-of-band") results?
How do you distinguish between valid and error ("out-of-band") results?

(You’ll need a return type that can represent more values than are in the range of your function!)
How do you tell the user to check for errors after each method call?
How do you tell the user to check for errors after each method call?

Not with a straight face, I hope.
(Even worse: handling an error may require many methods to return.)
main(String[] args)
doStuff()
doUsefulStuff()
doCriticalStuff()
doVitalStuff()
doAbsolutelyNecessaryStuff()
Consider this call stack.

```
doAbsolutelyNecessaryStuff()
    doVitalStuff()
    doCriticalStuff()
    doUsefulStuff()
    doStuff()
    main(String[] args)
```
Consider this call stack.

Assume `doStuff` cannot succeed unless every method it calls succeeds, and so on.
Consider this call stack.

Assume `doStuff` cannot succeed unless every method it calls succeeds, and so on.

What happens if there is an error in `doAbsolutelyNecessaryStuff`?
Consider this call stack.

Assume `doStuff` cannot succeed unless every method it calls succeeds, and so on.

What happens if there is an error in `doAbsolutelyNecessaryStuff`?

A big pain for the programmer, that’s what happens.
How do you ensure that the user actually checks for errors?
How do you ensure that the user actually checks for errors?

Stern warnings?  Strongly-worded documentation?  Shame?  Bribes?
...and let’s not get started on constructors!
...and let’s not get started on constructors!

(OK, twist my arm. Why are constructors even harder to deal with than methods?)
It should be apparent that error-handling is a hard problem.
A solution should...

- Provide some way to distinguish between legitimate return values and error notifications,
- Not excessively burden method callers with error code checking,
  - allow returning “up” multiple methods on the call stack, and
- Force the user to deal with (at least some kinds of) errors.
One solution: pass the buck.
“I am unhappy with the service that your company is providing me because of blah blah blah....”
“I am unhappy with the service that your company is providing me because of blah blah blah…”

“I can’t address your specific complaint, but please remember that we value your business!”
“I am unhappy with the service that your company is providing me because of *blah blah blah*....”

“I can’t address your specific complaint, but please remember that we value your business!”

“Indeed, I plan to cancel my service and contract with your fiercest competitor!”

Copyright © 2005-2007 William C. Benton
“I am unhappy with the service that your company is providing me because of *blah blah blah*....”

“I can’t address your specific complaint, but please remember that we value your business!”

“Indeed, I plan to cancel my service and contract with your fiercest competitor!”

“Sir/madam, let me transfer you to my supervisor. She may be able to help you.”
(Passing the buck has rich and lengthy precedent.)
Exceptions

- The "may I speak to your supervisor?" of contemporary programming languages
- Thrown to signal all sorts of error conditions and
- Caught by code that can handle the error.
Idea: try to execute code that may contain an error and catch any problems that may occur.
try
\[ stmt_{try} \]
catch (type id)
\[ stmt_{catch} \]
try

```
stmt_{try}
```

catch \((type \ id)\)

```
stmt_{catch}
```

*(type must be an exception type; i.e. it must be a reference to a class that extends Exception or implements Throwable)*
Scanner s = new Scanner(System.in);
boolean gotInputYet = false;
int result = 0;

do {
    try {
        System.out.print("Please enter a number: ");
        String input = s.nextLine();
        result = Integer.parseInt(input);
        // only reached if parseInt succeeds
        gotInputYet = true;
    }
    catch (NumberFormatException e){
        System.out.println("That's not a number.");
    }
} while(!gotInputYet);
try

\texttt{stmt}\textsubscript{try}

catch (\texttt{type}_1 \texttt{id})

\texttt{stmt}\textsubscript{catch}_1

catch (\texttt{type}_2 \texttt{id})

\texttt{stmt}\textsubscript{catch}_2

// etc.
$\textit{stmt}_{\text{try}}$

$\textit{stmt}_{\text{catch}}$

exception
When an exception is generated, Java transfers control to the nearest enclosing *catch block* that can handle that exception.
When an exception is generated, Java transfers control to the nearest enclosing *catch block* that can handle that exception.

(It won’t always be in the same method.)
The VM creates a catch block surrounding main().
throw ref;
throw ref;

(ref must be a reference to an instance of a class that extends Exception or implements Throwable)
Typically, you’ll construct `ref` on the line in which you `throw` it.

```java
throw new LameExampleException();
```
Constructing an exception object is just like constructing other objects:

```java
Exception e = new LameExampleException();
```

This does not cause `e` to be thrown. (Indeed, you can do all sorts of things with `e`.)
main(String[] args)
doStuff()
doUsefulStuff()
doCriticalStuff()
doVitalStuff()
doAbsolutelyNecessaryStuff()
catch FileNotFoundException

catch NullPointerException

catch Throwable

loadSomeStuff()
catch NumberFormatException

doSomeStuff()
catch Exception

catch Exception
throws any exception

```java
main(String[] args)
doStuff()
doUsefulStuff()
doCriticalStuff()
doVitalStuff()
doAbsolutelyNecessaryStuff()
catch FileNotFoundException
catch NullPointerException
loadSomeStuff()
catch NumberFormatException
catch Exception
throws any exception
```
```java
main(String[] args)
  doStuff()
  doUsefulStuff()
  doCriticalStuff()
  doVitalStuff()
  doAbsolutelyNecessaryStuff()
  catch FileNotFoundException
  catch NullPointerException
  catch Throwable
  loadSomeStuff()
  catch NumberFormatException
  catch Exception
```
main(String[] args)
doStuff()
doUsefulStuff()
doCriticalStuff()
doVitalStuff()
doAbsolutelyNecessaryStuff()
catch FileNotFoundException

catch NullPointerException

catch Throwable

catch NumberFormatException

loadSomeStuff()
main(String[] args)
doStuff()
doUsefulStuff()
doCriticalStuff()
doVitalStuff()
doAbsolutelyNecessaryStuff()
catch FileNotFoundException

loadSomeStuff()
catch NumberFormatException

throws a FileNotFoundException

catch Throwable

catch NullPointerException

catch NumberFormatException
```java
main(String[] args)
doStuff()
doUsefulStuff()
doCriticalStuff()
doVitalStuff()
doAbsolutelyNecessaryStuff()
catch FileNotFoundException

catch NullPointerException

catch Throwable

catch NumberFormatException
```
main(String[] args)
doStuff()
doUsefulStuff()
doCriticalStuff()
doVitalStuff()
doAbsolutelyNecessaryStuff()
catch FileNotFoundException

catch NumberFormatException
throws a NumberFormatException
```java
main(String[] args)
doStuff()
doUsefulStuff()
catch NullPointerException
catch Throwable
catch NumberFormatException
```
main(String[] args)  
doStuff()  
doUsefulStuff()  
doCriticalStuff()  
doVitalStuff()  
doAbsolutelyNecessaryStuff()  
catch FileNotFoundException  
catch NullPointerException  
catch Throwable  
loadSomeStuff()  
catch NumberFormatException  
throws a NullPointerException  

Copyright © 2005-2007 William C. Benton
main(String[] args)
doStuff()
doUsefulStuff()
catch NullPointerException
catch Throwable
throws an ArrayIndexOutOfBoundsException
throws an ArrayIndexOutOfBoundsException
Exceptions are a one-way street: once you’ve exited main() (or any other method), you can’t go back.

(This program has terminated.)
There are two kinds of exceptions: checked and unchecked.
Checked exceptions **must** be caught or declared to be thrown.

Unchecked exceptions **may** be caught.
Why do we have this distinction?
You **must** either catch checked exceptions or declare them in a *throws clause*.

```java
public void doDangerousStuff()
    throws HazardException,
            ComputerCombustionException {

    // ...

}
```
Signature vs. prototype

• Method signature: name of method + list of parameter types

• Method prototype: signature + throws clause

• Java compiler uses prototype to determine which method calls might throw exceptions
Any questions?
Rolling your own

• You can make your own exception types by extending Exception or RuntimeException

• You can also implement Throwable

• What are some exceptions you might use in your programs?
Exercise

• Say you need to develop an ice-cream cone class with an eat(int) method

• Write code to throw an exception if a class user attempts to eat a negative number of scoops

• Should this be checked or not?
Sometimes, we want to execute some code after a block, whether or not it causes an exception.
Scanner f = null;

try {
    f = new Scanner("file.txt");
} catch (IOException ioe) {
    if (f != null) f.close();
}

if (f != null) {
    try {
        f.close();
    } catch (Exception e) {}
}

if (f != null) {
    try {
        f.close();
        f.close();
    } catch (Exception e) {}
}
Statements in a *finally block* are executed whether or not the *try block* threw an exception.
try
stmt_{try}
finally
stmt_{finally}
Why are we interested in this capability?
Review

What is an exception?
How do we generate an exception?
throw ref;
throw ref;

(ref must be a reference to an instance of a class that extends Exception or implements Throwable)
How do we handle exceptional conditions?
try
   \texttt{stmt}_{\text{try}}
catch \ (\texttt{type \ id})
   \texttt{stmt}_{\text{catch}}
try

\textit{stmt}_{\textit{try}}

catch \ (\textit{type id})

\textit{stmt}_{\textit{catch}}

\textit{(type} must be an exception type; \textit{i.e.} compatible with \textit{Exception} or \textit{Throwable})
What happens when an exception is thrown?
Control transfers to the nearest enclosing catch block that can catch that type of exception.
main(String[] args)
doStuff()
doUsefulStuff()
doCriticalStuff()
doVitalStuff()
doAbsolutelyNecessaryStuff()
catch FileNotFoundException

catch NullPointerException

catch Throwable

catch NumberFormatException

loadSomeStuff()
throws a FileNotFoundException

main(String[] args)
doStuff()
doUsefulStuff()
doCriticalStuff()
doVitalStuff()
doAbsolutelyNecessaryStuff()
catch FileNotFoundException
catch NullPointerException
catch Throwable
loadSomeStuff()
catch NumberFormatException
throws a FileNotFoundException

Copyright © 2005-2007 William C. Benton
main(String[] args)
doStuff()
doUsefulStuff()
doCriticalStuff()
doVitalStuff()
doAbsolutelyNecessaryStuff()
catch FileNotFoundException

catch NullPointerException

catch Throwable

catch NumberFormatException
main(String[] args)
doStuff()
doUsefulStuff()
doCriticalStuff()
doVitalStuff()
doAbsolutelyNecessaryStuff()
catch FileNotFoundException
catch NullPointerException
  catch NumberFormatException
throws a NumberFormatException
main(String[] args)
doStuff()
doUsefulStuff()
catch NullPointerException
catch Throwable

catch NumberFormatException
main(String[] args)
doStuff()
doUsefulStuff()
doCriticalStuff()
doVitalStuff()
doAbsolutelyNecessaryStuff()
catch FileNotFoundException

catch NullPointerException

catch Throwable

catch NumberFormatException
throws a NullPointerException

loadSomeStuff()
main(String[] args)
doStuff()
doUsefulStuff()
catch NullPointerException
catch Throwable

Copyright © 2005-2007 William C. Benton
What are the two kinds of exceptions?
Checked exceptions **must** be caught or declared to be thrown.

Unchecked exceptions **may** be caught.
Rolling your own

- You can make your own exception types by extending Exception or RuntimeException
- You can also implement Throwable
- What are some exceptions you might use in your programs?
Exercise

• Say you need to develop an ice-cream cone class with an eat(int) method

• Write code to throw an exception if a class user attempts to eat a negative number of scoops

• Should this be checked or not?
Sometimes, we want to execute some code after a block, whether or not it causes an exception.
Scanner f = null;

try {
    f = new Scanner("file.txt");
} catch (IOException ioe) {
    if (f != null) f.close();
}

if (f != null) {
    try {
        f.close();
    } catch (Exception e) {
    }
}

/* what happens if some other exception is thrown? */
Statements in a *finally* block are executed whether or not the *try* block threw an exception.
try
stmt
finally
try
stmt
finally
try
catch (type id)
stmt
finally
try
stmt
finally
try
stmt
finally
Why are we interested in this capability?