Additional Concepts

- Exceptions: What if things don't work out.
- Copying: Copying objects
- Equality: Testing objects for equality
- Interning: Guaranteeing unique objects

Exceptions

- An exception is a kind of "extraordinary" exit from normal control flow.
- Exceptions are used to "catch" occasional situations for which:
  - code would get cluttered if we had to check for them repeatedly
  - the situation may come from inside a method to which we do not have access.
- Such situations "throw" the exception.

Why we need to know about this

- It helps makes your code robust (insensitive to various failures).
- Many library methods throw exceptions; you need to know how to code for these methods.

Exceptions Detail

- Exceptions could include:
  - Divide by 0
  - Arithmetic overflow
  - Error input-output operation
  - Bad input format
  - and others, including programmer-defined ones
- Exceptions in Java are implemented by Exception objects.
- Exceptions can carry values indicating the cause of the exception.
- Exceptions should not be used as a normal value-returning mechanism.

Throwables

- In Java, there is a more general interface, of which Exception is a special case:
  - Throwable is the interface
    - Exception is an implementation, typically used as a base class.
    - Error is another implementation, usually indicating a more serious internal error.

Exception Examples

CloneNotSupportedException
DataFormatException
GeneralSecurityException
IllegalAccessException
InterruptedException
IOException
RuntimeException
UserException

RuntimeException Sub-classes

ArithmeticException
ClassCastException
EmptyStackException
IllegalArgumentException
IndexOutOfBoundsException
NegativeArraySizeException
NoSuchElementException
NullPointerException
SystemException

Exception Lingo

- When an exception occurs it is said to be
  - “thrown”
- If an exception is thrown inside a method, it can either be:
  - “caught”, or
  - “passed”
- sort of like a hot potato

Exception passing

- An exception not otherwise caught will eventually get passed to the top-level main, at which point it will either be:
  - reported, then ignored, or
  - cause the program to terminate

Typical Exception Handling

- Keywords are:
  - **`try`**: execute some code (known as a try-block) in which an exception might be thrown
  - **`catch`**: handle the exception if it is thrown
  - **`finally`**: *optional* code executed after a try-bock whether or not an exception was thrown

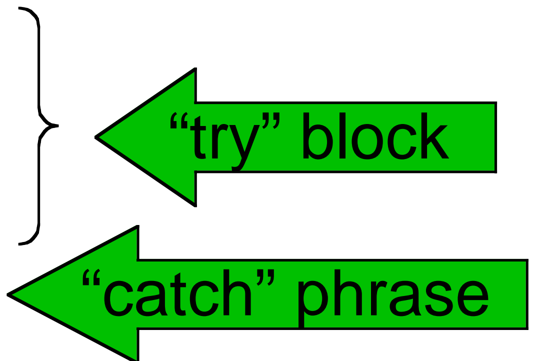
Problem: Opening a File

- File named might not exist
- Attempting to open a non-existent file will throw an

  `IOException`
- Need to catch, or will not compile

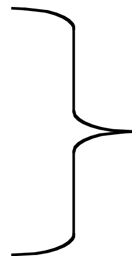
Correct Version

```
InputStream inStream = System.in;

if( arg.length > 0 )
  {
  String filename = arg[0];
  try
    {
    inStream = new FileInputStream(filename);
    }
  catch( IOException e )
    {
    System.err.println("*** unable to open file: " + filename);
    System.exit(1);
    }
  }
```



Generally >1 Exception Type

```
try
   {
   . . .
   }
catch( ExceptionType1 e )
   {
   . . .
   }
catch( ExceptionType2 e )
   {
   . . .
   }
. . .

finally
   {
   . . .
   }
```

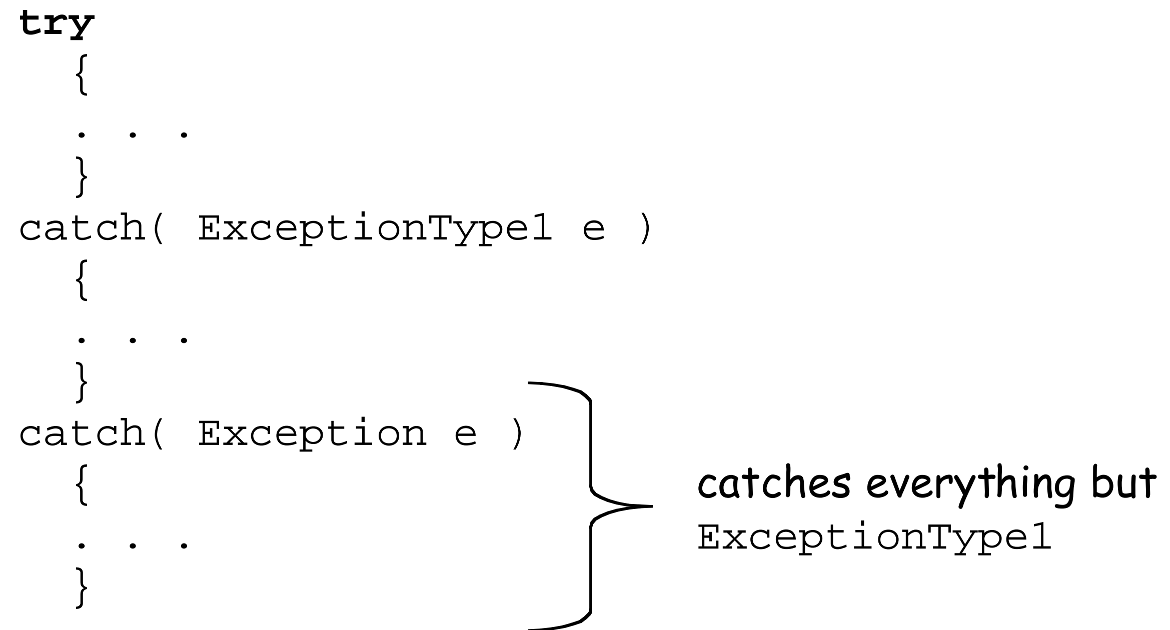


optional, always executed if present whether or not there is an exception

`finally` Code Example

```
FileInputStream stream = null;
StreamTokenizer input;
try
  {
  stream = new FileInputStream(filename);
  input = new StreamTokenizer(stream);
  boolean found = false;
  while( input.nextToken() != StreamTokenizer.TT_EOF )
    {
    if( word.equals(input.sval) )
      {
      found = true;
      break;
      }
    }
  if( found )
    {
    System.out.println("File " + filename + " contains " + word);
    }
  else
    {
  System.out.println("File " + filename + " does not contain " + word);
    }
  }
catch( IOException e )
  {
  System.err.println("IO exception opening or reading file " + filename);
  }
finally
  {
  if( stream != null )
    {
    try
      {
      stream.close();
      }
    catch( IOException e )
      {
      System.err.println("IO exception closing file " + filename);
      }
    }
  }
}
```

Catch-all for Exceptions



Declaring

If a method throws an exception, this fact must be declared:

```
void myMethod() throws MyException
{
… throw new MyException(msg);
}
```

Declaring

If a method *passes* on an exception, this fact must be declared:

```
void myMethod() throws IOException
{
inStream = new FileInputStream(filename);
}
```

Stopping the buck from being passed

If a method catches an exception, do not declare that it throws it, unless it does:

```
void myMethod() throws IOException
{
try
  {
   inStream = new FileInputStream(filename);
  }
catch( IOException e)
  {
  }
}
```

Exception on Declaration Rule

- The subclass of `RuntimeException` does not have to be declared.
- Example: `NoSuchElementException` (used in OpenList)

Copying

- Suppose we want to make a copy of an object with the following structure:

```
class MyClass
{
OpenList List1, List2;
}
```

Ways to Copy

- Copy constructor
- static copy method
- clone() method (returns copy)

- Any of these can be defined in terms of the other.

Copy constructor

- Add constructor

```
MyClass(MyClass orig)
  {
  …
  }
```

- Use constructor

```
MyClass newObj = new MyClass(orig);
```

static copy method

- Add method

```
static MyClass copy(MyClass orig)
    {
    …
    }
```

- Use method

```
MyClass newObj = MyClass.copy(orig);
```

clone() method

- Add method

```
public MyClass clone()
  {
  …
  }
```

- Use method

```
MyClass newObj = orig.clone();
```

Body of copy method, etc.

- Various meanings of "copy"
- Think of object as a tree
  - Deep copy: Copy all the way down to the leaves.
  - Shallow copy: Copy references only.
- Illustrate with copy constructors

Shallow Copy

- ```
  class MyClass
  {
  OpenList List1, List2;


  MyClass(MyClass orig)
          {
          List1 = orig.List1;
          List2 = orig.List2;
          }
  }
  ```
- The copy shares the lists with the original.
  ```

Deep Copy

- ```
  class MyClass
  {
  OpenList List1, List2;


  MyClass(MyClass orig)
       {
       List1 = copyOpenList(orig.List1);
       List2 = copyOpenList(orig.List2);
       }
  }
  ```
- The copy has copies of the original lists, *provided* that copyList makes such copies.
  ```

copyList, slightly deeper

- ```
  static OpenList copyOpenList(OpenList orig)
      {
      if( orig.isEmpty() )
              {
              return OpenList.nil;
              }
      else
              {
              return cons(orig.first(),
                              copyOpenList(orig.rest()));
              }
      }
  ```

- Still does not copy individual list elements (which could be lists themselves).
  ```

copyList, with copied Elements

- ```
  static OpenList copyOpenList(OpenList orig)
      {
      if( orig.isEmpty() )
            {
            return OpenList.nil;
            }
      else
            {
            return cons(copy(orig.first()),
                        copyOpenList(orig.rest()));
            }
      }
  ```
  ```

Copying elements that are lists recursively

```
static Object copy(Object ob)
  {
  if( ob instanceof OpenList )
    {
    return copyOpenList((OpenList) ob);
    }
  else
    {
    return ob;
    }
  }
```

A *non-list* item will not be copied, but rather will be referenced as is.

clone()

- `clone()` is defined for every object.
- It is over-ridable.
- There is an interface `Cloneable:` to over-ride `clone()` a class must declare that it

  `implements Cloneable`
- To get an Object to clone itself, Java says to return:

  `super.Clone()`
- If the Object is not Cloneable, it will throw

  `CloneNotSupportedException`

clone() limitations

- You cannot do this:

```
static Object cloneIt(Object ob)
{
try
    {
    return ob.clone();
    }
  catch( CloneNotSupportedException e)
    {
    return ob;
    }
}
```

  as `clone()` is not visible for Objects in general.

- I could find no way to implement a general static method that will check dynamically whether an Object implements `Cloneable`, then call `clone()`. There may be a kludgy way.

Checking Equality

- Defining `equals()` is at programmer's discretion
- By analogy with copying:
  - Equality checking can be deep or shallow.
- *Semantic* equality may be taken into account:
  - e.g. allowing an integer value to be *equal* to a floating value.

Default `equals()`

- `equals()` is defined in the base class `Object`.
- It may/should be over-ridden.
- The default implementation is that

  `x.equals(y)`

  if, and only if, `x` and `y` are the *same* Object.

Interning principle

- Suppose we could guarantee that two objects are semantically equal only if they are the *same* object.
- Then computing equals would be very fast: only need to compare references.
- Such a guarantee can be made if we intern all of the objects in the class.

Interning principle

- To *intern* objects in a class, we need a way to get to all objects in that class that were ever created.
- The client does not use the constructor, but rather uses a factory method that returns objects.
- The factory method checks to see whether the prescribed object has already been created
  - If so, it returns the existing object's reference.
  - If not, it creates a new object (using the constructor) and returns the reference to that object.

Interning Example: an Interning cons

```
OpenList cons(Object F, OpenList R)
  {
  OpenList found = find(F, R);
  if( found == null )
    {
    found = new OpenList(new Cell(F, R));
    remember(found);
    }
  return found;
  }
```

Clearly, sharing is intended.

`find` and `remember` must be implemented.

Implementing `find` and `remember`

- We must store every OpenList ever produced (sharing tails, etc.)
- We must be able to find a list with equal firsts and rests.
- A naïve implementation would store an internal list of all OpenLists (not itself a OpenList) and search the list with `find`.
- This process can become slow.

Faster
Implemention of `find` and `remember`

- Use the concept of hashing.
- Hashing computes a numeric signature for the proposed new item.
- It then uses the signature to access an array (called a hash table) of "equivalence classes" of items.
- Each equivalence class ideally has a relatively *small* number of items in it.
- The only searching needed is that of searching the small equivalence class, not the whole universe.

How Java Helps

- Java provides method `hashCode()` of `Object`.
- This gives a (generally large) number for any object.
- By dividing the hash code by the hash table size, equivalence classes are thereby formed.
- All objects with the same hash code *modulo* the table size are considered equivalent.
- Java also provides a class `HashSet` that can be used to implement `find` and `remember`.

Further Investigation

- Examine these classes/*interfaces* on the Java API web page:
  - `Exception`
  - `Object`
  - *`Cloneable`*
  - `HashSet`
  - *`Collection`*
  - `HashMap`