## Defensive Programming: a cheap 'n' nasty web browser

- The lessons in this case study include:
  - more packages, inner classes & programming to an interface
  - UML class diagrams
  - the Command design pattern
  - Java I/O (console, file and URL)
  - defensive programming & Java exception handling for data that comes in from outside the system (console, files, URLs)
  - defensive programming, design-by-contract (preconditions) & Java exception handling for data that comes in from client code



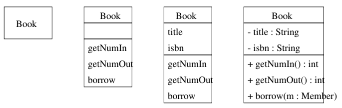
## Unified Modeling Language, UML

- "... is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system."
- Unified?
  - 'method wars' of the early '90s
  - the 3 amigos combine efforts (mid-'90s); one outcome is a unified language
  - adopted by the Object Management Group (1997) and then subsequent wide adoption & revision (now UML 1.3)
  - the language can be used with any method, but the 3 amigos have designed a method, the Unified Process (not in wide use)
- UML has 9 types of diagram.



## Class diagrams

- Class diagrams show classes and their relationships
- They model parts of the *structure* of your problem-domain or your software system
- Classes: name, attributes (variables), operations (methods)
- Do not have to show all or any attributes/operations; an empty compartment does not mean 'none'





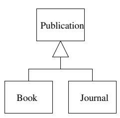

## Refining the diagram during software development

- Refine the clasess that you show:
  - Analysis classes: problem-domain ('real-world' classes)
  - Design classes: architecture (your high-level software solution, e.g. GUI/design pattern classes)
  - Implementation classes: software-internal (data structures, GUI classes)
- Refine what you show about each class:
  - Classes: add name, then add public operations, then add private operations & attributes
  - Operations: add name, then add parameter lists, return types & visibility
  - Attributes: add name, then add type & visibility



## Generalisation

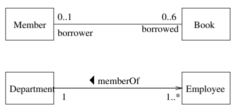
- Class B is a subclass of A; B is a A; B is a kind of A; every B is a A
- Objects of class B may be used anywhere that objects of class A may appear, but not the reverse
- B inherits the attributes and operations of A, and may have attributes and operations of its own





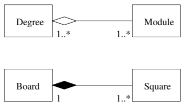

## Association

- Objects of class A are connected to objects of class B
- Draw as a line and optionally adorn with:
  - name of the association (sometimes with a solid triangle showing how to read the name)
  - role names for the participants in the association
  - multiplicities, e.g. 1, 0..1, 0..*, 1..*, etc.
  - navigability arrows to denote efficiency of traversal



**Special kinds of association: aggregation & composition**

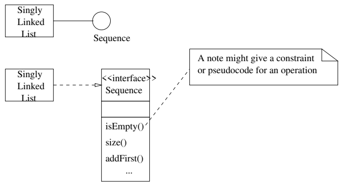
- Aggregation
  - associates a whole with its parts: has, is part of
- Composition
  - a special kind of aggregation, where the part dies with the whole






**Other things that can appear on class diagrams**

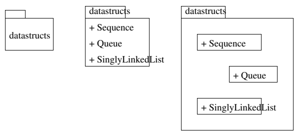
- Notes: can add these to any of the 9 diagrams
- Interfaces and the 'implements' relationship
  - draw simply as a circle, or like a class (when you want to show the operations)

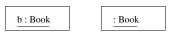





**Other things that can appear on class diagrams**

- Packages: draw with just a name or draw around a whole class diagram
- Objects (instances of classes): can be named or anonynous






KopyRight is a company which supplies photocopiers on a national basis. To deal with customer problems, KopyRight has a set of Help Desks, one per province.

Customers are organisations, and they will operate from a number of locations across Ireland. At each location, they will have a number of photocopiers.

When a photocopier at some location needs attention, the Help Desk for that location's province is rung.

Each Help Desk has a number of staff, some of whom are Help Desk Operators (HDOs), others of whom are Photocopier Engineers (PEs).

It will be an HDO who initially answers the phone. S/he becomes the 'primary owner' of that call: as such, it is his/her responsibility to make sure that the problem eventually gets resolved.

S/he will start by recording details of the problem. Sometimes s/he will be able to resolve the problem immediately over the phone. In this case, s/he records the status of the call as 'resolved'. Often, however, s/he will need to take a so-called referral action.

In referral, the HDO passes the call on to one of the PEs who works for that Help Desk. This makes the PE a 'secondary owner' of the problem. The PE will try to fix the problem either over the phone or by visiting the customer's site. If s/he solves the problem, the PE informs the primary owner that its status is now 'resolved'.

If, however, the PE cannot resolve the problem, s/he informs the primary owner that the call is still 'unresolved'. The primary owner might now try to pass the call to another PE. This PE now becomes the new 'secondary owner' of the problem.

This goes on until the problem is 'resolved'. (A PE might even become the secondary owner several times if s/he is invited to have repeated stabs at solving the problem.)

Draw a class diagram that models this problem domain.

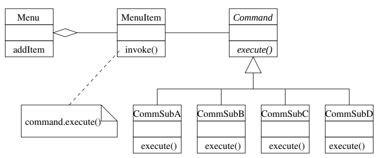


**Menu-driven systems**

- How can we arrange that a menu selection invokes some corresponding processing?
- One way is that the user's choice (an `int`) is used in an `if/switch` statement.
- This is an inflexible solution. It hard-codes the choices and their processing:
  - adding/removing choices involves modifying the `if/switch`
  - re-ordering choices involves modifying the `if/swit`
  - cannot determine menu choices dynamically (e.g. cannot read them in from a file) because you still need to know how to code the `if/switch`



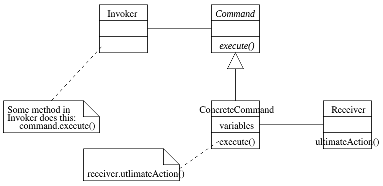
**Decoupling the user interface from the actions**




- Each `MenuItem` contains a `Command` object. But `Command` is abstract. So each `MenuItem` will contain an appropriate object from a subclass of `Command` (polymorphism). Subclasses implement `execute()` in different ways to fulfil different requests.
- When a `MenuItem` object is selected, we run its `invoke()` method. This sends a message to the `MenuItem`'s `Command` object to run `execute()`. The appropriate `execute()` will be run (dynamic method binding).

### Generalising this idea

- In general, the subclasses of `Command` can also contain instance variables and other methods
- `execute()` might make use of these variables and methods. In particular, it might send a message to another object
- In this way, the invoker is completely decoupled (doesn't know anything about) the object that ultimately does the work or the method that is ultimately run






### The Command Design Pattern

- *Name:* Command (also known as Action, Transaction)
- *Problem:* You want to issue requests to objects without knowing anything about the methods being requested or the receiver of the request. This pattern decouples the object that invokes the method from the one having the knowledge to perform it.
- *Solution:* Encapsulate requests as objects that implement some interface. (Typically, by being subclasses of `Command`, they must have an `execute()` method.)



### The Command Design Pattern

- *Consequences:*
  - Commands are objects so we can queue them or log them in data structures/files
  - We can assemble them into composite commands (macros)
  - Changing the set of commands dynamically is easy
  - By storing within them enough information, we can offer an `unexecute()` command (for an undo facility)
  - There is an extra level of indirection, so performance is worsened slightly



### Defensive Programming

- Your code takes responsibility for protecting itself from bad data being given to it (even though the bad data isn't necessarily your fault)
- Especially for data from an external source (user, file, database, Internet)
- But perhaps also for data from other parts of the program



### The defence budget

- E.g. one way to get data from a user:

```
public class MyProg
{ public static void main(String[] args)
  { System.out.println(args[0]);
    System.out.println(args[1]);
  }
}
```

- `java MyProg word1 word2`
  produces
  ```
  word1
  word2
  ```
- Defensiveness brings code bloat and code pollution
- Can we deal with abnormal situations without pollution?



### When things go wrong

- Consider a method and some client code. If something abnormal happens in the method...
- Could 'repair' (e.g. substitute default answer); the client isn't informed
- Or, could simply signal to the client
  1. return a special return value
  2. set a flag
  3. throw an exception

## Special return value

- `s1.indexOf(s2)`
  returns the leftmost position of string `s2` within `s1` or -1 if the search is unsuccessful

- Problems
  - finding a value we can set aside
  - methods that don't naturally return values
  - client programs must remember to check the value
  - if they do, then code pollution



## Setting a flag

- If `pw` is an object of class `PrintWriter`, then `println` will write a string to a file. The client can check for problems (e.g. full disk) by getting the value of an instance variable:

```
pw.println("A line of text");
if (pw.checkError())
{  // deal with problem
}
```

- Problems
  - coupling due to shared data
  - problems in concurrent systems
  - client programs must remember to check the value
  - if they do, then code pollution



## Throwing an exception: Java exception-handling

- A systematic way of representing, transmitting, capturing and handling well-defined abnormal situations
- When the method detects the abnormal situation, it creates an object, known as an exception
- It throws the exception, which may be caught and handled by an appropriate exception-handler, e.g. in the client code
- This is an abnormal return of the flow of control from the method to the client
- It cannot, in general, be ignored by the client
- But handling is separated from detecting/throwing, so less code pollution



## Example: `Integer.parseInt`

```
String str = // e.g. get user's input
int i = Integer.parseInt(str.trim());
```

- What if in wrong format? `parseInt` doesn't substitute a default value. Doesn't return special value. Doesn't set a flag. Creates and throws a `NumberFormatException`

- The client code can catch & handle the exception:

```
try
{  ...
   String str = // e.g. get user's input
   int i = Integer.parseInt(str.trim());
   ...
}
catch (NumberFormatException e)
{  System.out.println("Problem!");
   ...
}
```



## Classes of exceptions

```
Exception
   |
   |
   +----CloneNotSupportedException
   |
   +----IOException
   |       |
   |       +----EOFException
   |       |
   |       +----FileNotFoundException
   |       |
   |       +----MalformedURLException
   |
   +----InterruptedException
   |
   +----RuntimeException (unchecked)
           |
           +----IllegalArgumentException
           |       |
           |       +----NumberFormatException
           |
           +----IndexOutOfBoundsException
           |       |
           |       +----ArrayIndexOutOfBoundsException
           |       |
           |       +----StringIndexOutOfBoundsException
           |
           +----NullPointerException
```



## Defining your own exceptions

- Makes your program more self-documenting

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

   public MyException(String msg)
   {  super(msg);
   }
}
```

- Inherits, e.g., `getMessage` and `printStackTrace`

- Your definition can be more complex: further constructors, variables and methods.

### Writing the method that detects the abnormal situation

- Create a new object from a suitable exception class definition and throw it using `throw` (N.B. `throw` not `throws`)

- E.g. somewhere in the `parseInt` method, the programmer included the following:

```
throw new NumberFormatException();
```

- E.g.

```
if (x != 0)
{  return y / x;
}
else
{  throw new MyException(
        "You're trying to divide by zero!")
}
```



### `try` and `catch`

```
try
{  some statements
}
catch (SomeException e)
{  handle the exception
}
catch (SomeOtherException e)
{  handle the exception
}
more statements
```

- Try to execute the `try` block. If no exceptions produced, the `catch` blocks are ignored

- If any statement in the `try` block throws an exception, the rest of the `try` block is not executed. A `catch` blocks for that class of exception is executed (one of *these* `catch` blocks if they catch the right class of exception, else some other `catch` block)



### The catch or specify principle (for checked exceptions)

- Every (checked) exception that may be passed to your method must either be *caught* in a suitable `catch` clause or *specified* in a `throws` clause (N.B. `throws` not `throw`)

```
public void myMethod throws AnException
{  try
   {  some statements
   }
   catch (SomeException e)
   {  handle the exception
   }
   catch (SomeOtherException e)
   {  handle the exception
   }
   more statements
}
```

- The *compiler* enforces the principle



### Which `catch` block is executed?

- Java looks through the `catch` clauses sequentially

- It executes the first that matches (is equal to or a superclass of)

- If none match, the exception object is passed up the *call chain*

- If no method in the call chain catches the exception, the Java run-time system catches it, displays a message & terminates the program (except for applets and applications with GUIs)



### Unchecked exceptions

- `RuntimeException` and its subclasses are special

- The catch or specify principle, and its enforcement by the compiler, doesn't apply

- If you don't catch, you're not forced to specify

- E.g. no need to write `throws NumberFormatException`

- But the compiler can't then check that clients are handling in some way



### Some principles of good exception handling

- Define your own classes of exceptions — makes your program more self-documenting

- Your own classes should be checked exceptions, whenever reasonable

- The method in which an abnormal situation first arises often cannot handle the situation — it does not know what its many clients would want it to do. So it must throw the exception.

- But do not allow too much 'passing the buck' (throwing up the call chain) — if handler is distant from where exception is first thrown, then distant parts of your program are coupled

### Continued

- To avoid code pollution, keep `try` blocks quite large (larger than might be necessary)

- Do not use exception handling as a way of doing normal flow of control — it is often more complicated than using `if`s, etc. and it has poor performance, so it should be used for abnormal situations only

- Stick to implementing only ‘standard’ practices:

  1. catch and retry (perhaps retrying only a limited number of times)

  2. catch, clean up, report & throw to caller



### The `File` class

- Objects that represent the names of files/directories on the host file system. (These are objects in memory; they may or may not correspond to actual files/directories)

```
File f = new File("myDir/myFileName");
if (f.exists()) ...
if (f.isDirectory()) ...
if (f.isFile()) ...
if (f.canRead()) ...
if (f.canWrite()) ...
String parDir = f.getParent();
String[] contents = parDir.list();
String absPath = f.getAbsolutePath());
File newF = new File("myNewFileName");
f.rename(newF);
f.delete();
File newDir = new File("newDir");
File file = new File(newDir, "fname");
boolean didCreate = newDir.mkdir();
```



### File I/O: sequential v. random

- There are *sequential-access* files (using streams) and *random-access* files

- A stream is an object representing an ordered sequence of data with a source (input streams) or a destination (output streams); much used in Java (and CompSci) for moving data between files, strings, arrays, etc.

- Streams can be ‘pipelined’ together

- When random-access files are opened, a pointer keeps track of your position in the file (the number of bytes from start of file) and this is where read/writes occur

- A stream can be opened for either reading or writing; random-access files can be opened for reading, writing or both



### File I/O: binary v. text

- At one level, all files are binary files

- But by *binary files* we mean files that contain data not designed for human consumption; written by programs; read by programs

- *Text files* contain data designed for human consumption; can be read/written using a text editor, as well as read/written by other programs; so the format will be some text-editor format (e.g. ASCII)

| | Sequential | Random |
|---|---|---|
| Binary | Subclasses of abstract classes *`InputStream`* & *`OutputStream`* | Class `RandomAccessFile` |
| Text | Subclasses of abstract classes *`Reader`* & *`Writer`* | N/A |



### Writing data to binary files

```
OutputStream
  |
  +----ByteArrayOutputStream
  |
  +----FileOutputStream
  |
  +----FilterOutputStream
  |        |
  |        +----BufferedOutputStream
  |        |
  |        +----CheckedOutputStream
  |        |
  |        +----DataOutputStream
  |        |
  |        +----DeflaterOutputStream
  |        |        |
  |        |        +----GZIPOutputStream
  |        |        |
  |        |        +----ZipOutputStream
  |        |
  |        +----DigestOutputStream
  |        |
  |        +----PrintStream
  |                 |
  |                 +----LogStream
  |
  +----ObjectOutputStream
  |
  +----PipedOutputStream
```



### Writing data to binary files

- Creating the stream also opens it

```
stream = new FileOutputStream("filename")
stream = new FileOutputStream(
                   new File("filename"));
int i = 23;
stream.write(i); // writes the least
                 // sig 8 bits (a byte)
```

- Program termination closes the file (but can use `stream.close()`); opening/writing/closing can throw `IOException`

- Writing data is painful, e.g. to write an `int` takes, e.g., four `write`s and three shifts

- So, stick another stream ‘on top’ that already provides the functionality you need (a subclass of `FilterOutputStream`)

**Writing data to binary files**

- You'll probably use a `DataOutputStream`, which can write any of Java's primitive types:

```
stream = new DataOutputStream(
      new FileOutputStream(...));
int i = 23;
stream.writeInt(i); // writes all 32 bits
```

- `writeBoolean`, `writeByte`, `writeChar` (16 bits), `writeDouble`, `writeFloat`, `writeInt`, `write-Long`, `writeShort`

- Can even write strings: `stream.writeUTF("Hello")` writes the string in a special encoding called UTF (Unicode Transmission Format), designed for compact transmission of Unicode characters



**Reading data from binary files**

```
InputStream
   |
   +----ByteArrayInputStream
   |
   +----FileInputStream
   |
   +----FilterInputStream
   |       |
   |       +----BufferedInputStream
   |       |
   |       +----CheckedInputStream
   |       |
   |       +----DataInputStream
   |       |
   |       +----InflaterInputStream
   |       |       |
   |       |       +----GZIPInputStream
   |       |       |
   |       |       +----ZipInputStream
   |       |
   |       +----DigestInputStream
   |       |
   |       +----LineNumberInputStream
   |       |
   |       +----PushbackInputStream
   |
   +----ObjectInputStream
   |
   +----PipedInputStream
   |
   +----SequenceInputStream
   |
   +----StringBufferInputStream
```



**Reading data from binary files**

- Creating the stream also opens it

```
stream = new FileInputStream("filename");
stream = new FileInputStream(
                 new File("filename"));
int i = stream.read(); // read a byte
```

- Program termination closes the file (but can use `stream.close()`); opening/reading/closing can throw `IOException`; `read` returns -1 if reading beyond end of file

- Reading data is painful, e.g. to read an `int` takes, e.g., four `read`s and three shifts

- So, stick another stream 'on top' that already provides the functionality you need (a subclass of `FilterInputStream`)



**Reading data from binary files**

- You'll probably use a `DataInputStream`, which can read any of Java's primitive types:

```
stream = new DataInputStream(
            new FileInputStream(...));
int i = stream.readInt(); // reads 32 bit
```

- `readBoolean`, `readByte`, `readChar` (16 bits), `readDouble`, `readFloat`, `readInt`, `readLong`, `readShort`, `readUTF()`

- These read methods throw `EOFException` if end of file is reached (including if it is reached when trying to read in the appropriate number of bytes)



**Writing/reading objects to files**

- If reading/writing whole objects, it is painful to have to write/read each instance variable yourself

- `ObjectOuputStream` and `ObjectInputStream` write & read objects

- Any class whose objects are to be written/read must implement the `Serializable` interface

- The contents of each instance variable are written/read. If they refer to other objects, these other objects are written/read too (if `Serializable`) in a way that preserves the original structure



**Writing/reading objects to files**

- Assume `sll` references a singly linked list and that `SinglyLinkedList` and `Node` both implement `Serializable`:

```
stream = new ObjectOutputStream(
            new FileOutStream("..."));
stream.writeObject(sll);
...
stream = new ObjectInputStream(
            new FileInputStream("..."));
sll = stream.readObject(sll);
```

  — writes/reads whole list structures

- Many more details (e.g. exceptions, static/transient variables,...)

### Writing data to text files

```
Writer
   |
   +----BufferedWriter
   |
   +----CharArrayWriter
   |
   +----FilterWriter
   |
   +----OutputStreamWriter
   |       |
   |       +----FileWriter
   |
   +----PipedWriter
   |
   +----PrintWriter
   |
   +----StringWriter
```



### Writing data to text files

- Creating the stream also opens it

```
stream = new FileWriter("filename");
stream = new FileWriter(
                  new File("filename"));
char c = 'a';
stream.write(c);
```

- Program termination closes the file (but can use `stream.close()`); opening/writing/closing can throw `IOException`

- This writes one character at a time (using a specified encoding); the writing is actually buffered, but the conversion to the encoding is not



### Writing data to text files

- For efficiency, include a `BufferedWriter` to buffer the conversions

```
stream = new BufferedWriter(
            new FileWriter(...));
char c = 'a';
stream.write(c);
```

- For greater convenience, include a `PrintWriter`, which gives you `print` and `println` methods for all Java primitive types and objects (using `toString`)

```
stream = new PrintWriter(
            new FileWriter(...));
stream.println("The answer is " + 42);
```

(but throws no exceptions; use `checkError`)



### Reading data from text files

```
Reader
   |
   +----BufferedReader
   |       |
   |       +----LineNumberReader
   |
   +----CharArrayReader
   |
   +----FilterReader
   |       |
   |       +----PushbackReader
   |
   +----InputStreamReader
   |       |
   |       +----FileReader
   |
   +----PipedReader
   |
   +----StringReader
```



### Reading data from text files

- Creating the stream also opens it

```
stream = new FileReader("filename");
stream = new FileReader(
                new File("filename"));
int i = stream.read(); // read a char
char c = (char) i;
```

- Program termination closes the file (but can use `stream.close()`); opening/reading/closing can throw `IOException`; `read` returns -1 if reading beyond end of file

- Stick another stream 'on top' for efficient and convenient reading of lines



### Reading data from text files

- `BufferedReader`s fill a buffer with a single disk access, instead of making numerous individual read requests

```
stream = new BufferedReader(
            new FileReader(...));
int i = stream.read(); // read a char
char c = (char) i;
String str = stream.readLine();
```

- If end of file is reached, `read` returns -1 but `readLine` returns `null`

## Console I/O

- For output, send `print`/`println` messages to the output stream `System.out`
- For input, there is an input stream, `System.in`
- It is typically a `BufferedInputStream`, with methods only for reading bytes
- Stick an `InputStreamReader` 'on top', which bridges from bytes to characters
- Stick a `BufferedReader` onto that for efficiency and convenience of reading lines:

```
stream = new BufferedReader(new
   InputStreamReader(System.in));
String str = stream.readLine();
```



## URL

- Objects that represent URLs (These are objects in memory; they may or may not correspond to actual web resources)

```
URL u = new URL(
"http://www.cs.ucc.ie/~dgb/teaching.html"
```

(Can throw `MalformedURLException`)

```
String fname = u.getFile();
String hname = u.getHost();
int pnum = u.getPort();
String pname = u.getProtocol();
String aname = u.getRef(); // anchor

stream = new BufferedReader(new
   InputStreamReader(u.openStream()));
String str = stream.readLine();
```

(`IOException` and its subclasses may be thrown)



## Defensiveness: preconditions

- Suppose a method of mine, `meth`, has a precondition, $P$, and this is properly documented
- The precondition places an obligation on clients of `meth`
  - it is the responsibility of the client programmer to ensure that their code satisfies $P$ before they invoke `meth`
  - if they do not, they have a bug and, if things go wrong subsequently, e.g. in `meth`, it is their fault
  - if they do and things still go wrong, then I have a bug and it is my fault



## Defensiveness: precondition checking

- So, if there is a documented precondition, it is the responsibility of **client code** to ensure that it is satisfied
- `meth` itself has no obligation to check its precondition, e.g. to check the parameters it receives
- But, if you want to be defensive, then `meth` might *also* do the check and throw an exception if the precondition is violated
- This doesn't relieve the client of its obligation
- The upshot, if client code is written responsibly, will be redundant code ('belt-and-braces')
- But you will be protected in cases where client code is written irresponsibly



## When to be defensive

- For data from external sources, always be defensive. For data from other parts of the program (from client code), a judgement is needed
- Choosing preconditions
  - If it is not a bug in the client to invoke your method with the condition unsatisfied, then the condition is not a precondition, and vice versa
  - Choose the strongest precondition that you can (this increases the cohesion of your methods)
  - Sometimes it is impossible (or very inefficient) for client code to check conditions; these should probably not be preconditions (e.g. checking *whether* something is in an array before invoking a method that finds *where* it is in the array)



## When to be defensive

- If you can afford the small performance degradation, then be defensive everywhere; if not...
- Methods whose client code is known and will not change need not check their preconditions, e.g.
  - `private` methods
  - methods in 'one-off' classes, used in this application but no other
  - methods in classes that won't be placed in libraries
  - methods in classes that won't be re-used
- Do check preconditions in methods in classes that are the opposite to the above (especially utility classes)

## Design-By-Contract

- Design-by-contract is a discipline we can adhere to when writing code;
- a way of documenting our program;
- a way of supporting development, testing, debugging and maintenance; and
- a way of making our programs more fault-tolerant.
- There is language-support for it within certain languages (e.g. Eiffel)



## Design-By-Contract

- View the relationship between client and the class it sends messages to as a formal agreement, a contract
- By understanding the obligations placed on both
  - we will write our method bodies correctly
  - we will know how to handle abnormal situations robustly



## Correctness

- Code is only correct *relative to a specification*
- Some notation: $\{P\}A\{Q\}$
  - $A$ is a sequence of statements (e.g. a method body)
  - $P$ and $Q$ are *assertions*: they express properties about objects, variables, values, etc.
  - $P$ is the *precondition*; $Q$ is the *postcondition*
  - *"Any execution of $A$, starting in a state where $P$ holds, will terminate in a state where $Q$ holds."*
- Our aim, given $P$ and $Q$ is to write statements $A$ such that the above is true.



## Using these conditions

- Before coding, work out $P$ and $Q$
- The more precise they are, the better
- Include them as executable statements (in some suitable syntax) in your method
- At run-time, they are checked to make sure they are true; an exception is raised if not
- Turn them on for debugging; if you can't afford the performance penalty and if performance is more important than robustness, turn some or all of them off for normal execution