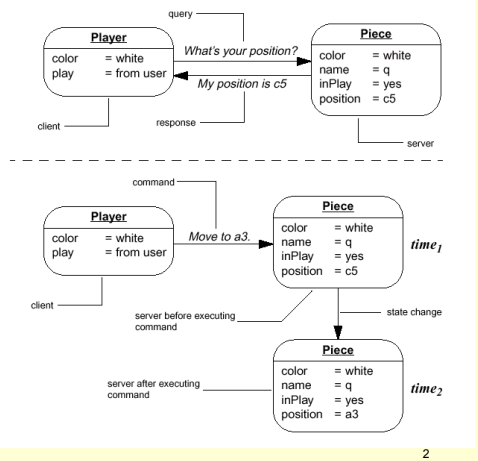
Programming By Contract

Specifying method preconditions and postconditions

Client and Server relationship

- A **client** queries and commands a **server**:
  - **Queries** ascertain values of properties
  - **Commands** change its state
- A client **uses** a server.



Specification and implementation

- **Specification**
  - an object's features, as seen by its clients
- **Implementation**
  - the "internals" that make up the features
- Specifications isolate you from the details of the implementation:
  - "I don't care how you do it, just get the job done" (as long as it meets the specifications).
  - How the features are actually implemented by the server, is of no concern to the client.
- Preserving the distinction between specification and implementation is absolutely essential:
  - Java syntax does not allow us to separate a class specification from its implementation.
  - We will make this distinction using Javadoc comments.

Programming by contract

- **postcondition**
  - a condition the implementor (server) guarantees will hold when a method completes execution
- **invariant**
  - a condition that always holds true
- **class invariant**
  - an invariant regarding properties of class instances: that is, a condition that will always be true for all instances of a class

A counter example

- Enumerate the object's responsibilities:
  - Identify queries:
    - properties of the object that define its state
  - Identify commands:
    - ways in which an object can change state
- A simple counter's responsibilities:
  - Know (two queries):
    - The value of the count
      - `currentCount()`
        - a non-negative integer
    - Is the count zero?
      - `isZero()`
        - boolean result
  - Do (two commands):
    - Set the count to 0
      - `reset()`
        - set the count to 0
    - Increment the count by 1
      - `incrementCount()`
        - increments the count by 1

Counter class

- A **class invariant** for `Counter` is that the instance (component) variable `count` will always be greater than or equal to zero.
- Class invariants for instance variables will appear as **postconditions** to their accessor methods (e.g., `currentCount`() for `count`).
- Each class invariant must also hold true for all of the methods of the class.
- Thus, the **postcondition** to the `incrementCount()` method must make sure that `count` is greater or equal to zero when it completes execution.

Counter class ...

- Class invariants and postconditions are part of class specification but not the implementation.
- They should be included in comments but not in the implementation.
- Consider an object's data areas.
- We will add a simple line comment to indicate constraints on each instance variable:

```
private int count; //current count
                   //  invariant:
                   //    count >= 0
```


Counter class ...

- Postconditions are specified using the `@ensure` Javadoc tag:

```
/**
 * The number of items counted.
 *
 * @ensure result >= 0
 */
public int currentCount () {
   return count;
}
```

- Note the use of the special "keyword" `result` to represent the value that is returned from a method.

Counter class ...

- Now consider the postcondition to the `reset()` method:

```
/**
 * Reset the count to 0.
 *
 * @ensure currentCount() == 0
 */
public void reset () {
   count = 0;
}
```


Counter class ...

- Note that the postcondition we have given for `reset()` is:

```
@ensure currentCount() == 0
```

- We have said `currentCount() == 0` instead of `count == 0` because `count` is a private variable and therefore not known to the client.
- Clearly, this postcondition is accurate and valid.
- Furthermore, this postcondition also maintains the class invariant because `currentCount() == 0` implies `currentCount() >= 0`.

Counter class ...

- Now consider the postcondition to the `incrementCount()` method:

```
/**
 * Increments the count by 1.
 *
 * @ensure currentCount() >= 0
 */
public void incrementCount () {
   ...
}
```


Counter class ...

- Note that the postcondition we have given for `incrementCount()` is:

```
@ensure currentCount() >= 0
```

- As with `reset()`, we have used `currentCount()` instead of `count` in the postcondition because `count` is a private variable and therefore not known to the client.
- This postcondition is accurate and valid and it clearly maintains the class invariant.
- However, it doesn't accurately describe the state change resulting from the `incrementCount()` method.
- Can we make the postcondition more precise?

Counter class ...

- For `incrementCount()`, the postcondition should say that the new `count` is 1 more than the old `count`
- More precisely, we say that the new `count` == old `count` + 1
- To avoid confusion, we use the `old` prefix to denote a state prior to execution of a method.
- We could write `count == old.count + 1`
- But `count` is private, not known to the client.
- Postcondition:

```
currentCount() == old.currentCount() + 1
```

- Note that this postcondition implies that the class invariant is maintained.

Counter class ...

- Let's try to implement `incrementCount()`:

```
/**
 * Increment count by 1.
 *
 * @ensure currentCount() == old.currentCount() + 1
 */
public void incrementCount () {
   count = count + 1;
}
```

- This implementation ensures that new `count` is 1 more than the old `count` so we consider it **correct with respect to its specification**

Counter class ...

- For `decrementCount()`, the postcondition should say that the new `count` is 1 less than the old `count`
- Postcondition:

```
currentCount() == old.currentCount() - 1
```

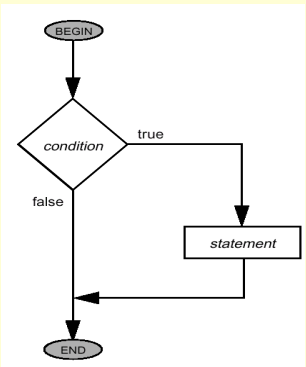
- Note that, unlike the `incrementCount()` case, this does not imply that the class invariant is maintained.
- Thus we need to add `currentCount() >= 0` to the postcondition.
- Updated Postcondition:

```
        currentCount() >= 0 &&
currentCount() == old.currentCount() - 1
```

- What if the `count` is 0 prior to execution?

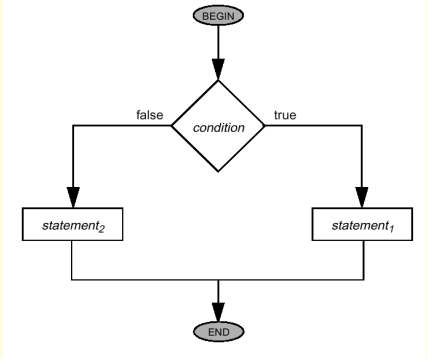
if statements

```
if (condition)
   statement
```



if-else statements

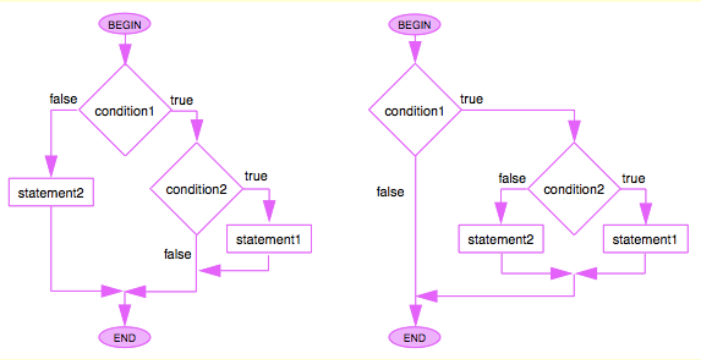
```
if (condition)
  statement1
else
  statement2
```



if-else statements ...

```
if (condition1) {
   if (condition2)
      statement1
} else
   statement2
```

```
if (condition1)
   if (condition2)
      statement1
else
   statement2
```

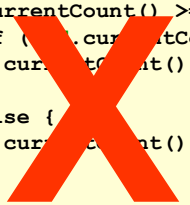


Lab #4

Counter class ...

- We need to say, if **count** is 0, then the new **count** remains the same as the old **count**, otherwise the new **count** is 1 less than the old **count**

```
/**
 * Decrement positive count by 1; zero count remains 0.
 *
 * @ensure currentCount() >= 0 &&
 *         if (old.currentCount() == 0) {
 *           currentCount() == old.currentCount();
 *         }
 *         else {
 *           currentCount() == old.currentCount() - 1;
 *         }
 */
public void decrementCount () {
   ...
}
```



- What is wrong with this specification?

Conditional Expressions

- A ***conditional expression*** consists of a **`boolean`** expression and two component expressions:

  booleanExpression **`?`** ***expression*$_1$** **`:`** ***expression*$_2$**

- The **`boolean`** expression is first evaluated.
  - If it evaluates to **`true`** then the value of the conditional expression is the value of ***expression*$_1$**
  - If it evaluates to **`false`** then the value of the conditional expression is the value of ***expression*$_2$**

Conditional Expressions ...

- Note that **`if`** and **`if-else`** are **statements**, not expressions. They do not evaluate to anything, but rather conditionally execute statements.
- *Conditional expressions* are **expression**, not statements. As such, they evaluate to something.
- Conditional expressions give us a way to capture the notion of *choice* within an expression.

Lab #5

Counter class ...

- We can capture the postcondition with a conditional expression:

```
/**
 * Decrement positive count by 1; zero count remains 0.
 *
 * @ensure currentCount() >= 0 &&
 *         (old.currentCount() == 0) ?
 *         (currentCount() == old.currentCount()) :
 *         (currentCount() == old.currentCount() - 1)
 */
public void decrementCount () {
   ...
}
```

Counter class ...

- An alternate way of expressing the postcondition:

```
/**
 * Decrement positive count by 1; zero count remains 0.
 *
 * @ensure currentCount() >= 0 &&
 *         currentCount() == ((old.currentCount() == 0) ?
 *         old.currentCount() :
 *         old.currentCount() - 1)
 */
public void decrementCount () {
   ...
}
```


Counter class ...

- Let's try to implement `decrementCount()`:

```
/**
 * Decrement positive count by 1; zero count remains 0.
 *
 * @ensure currentCount() >= 0 &&
 *         currentCount() == ((old.currentCount() == 0) ?
 *         old.currentCount() :
 *         old.currentCount() - 1)
 */
public void decrementCount () {
   count = count - 1;
}
```

- What's wrong with the implementation?
  - In order to handle the zero count case, we must guard the assignment statement with a conditional statement.

Counter class ...

- Let's fix our implementation of `decrementCount()`:

```
/**
 * Decrement positive count by 1; zero count remains 0.
 *
 * @ensure currentCount() >= 0 &&
 *         currentCount() == ((old.currentCount() == 0) ?
 *         old.currentCount() :
 *         old.currentCount() - 1)
 */
public void decrementCount () {
   if (count > 0)
      count = count - 1;
}
```

- This implementation ensures the validity of its postcondition:
  - **correct with respect to its specification**

Contracts

- We use postconditions (using `@ensure` tags) as part of a contract to guarantee to the client that our methods actually do what they are supposed to do.
- In general, it is difficult (and sometimes even impossible) to make such guarantees in a vacuum.
- We (as the server) need to set some boundaries (or constraints) for the client in order to guarantee our results.
- We do this with **preconditions**.
- A **precondition** is a condition the client of a method must make sure holds when the method is invoked.
- Together, the precondition and the postcondition form a contract between the client and the server.

Counter class ...

- Recall our implementation of the `incrementCount()` method of the `Counter` class:

```
/**
 * Increment count by 1
 *
 * @ensure currentCount() == old.currentCount() + 1
 */
public void incrementCount () {
   count = count + 1;
}
```

- Does this really work for all possible values of `count`?

Counter class ...

- Recall that the data type of `count` is `int`.
- The `int` data type is 4 bytes long and has a range of −2,147,483,648 to +2,147,483,647.
- This determines an upper bound on the range for our counter:
  - It cannot exceed 2,147,483,647.
- Thus, a counter can only be legitimately incremented if its value is less than this limit.
- We can express this a precondition to the `incrementCount()` method using the `@require` tag:

```
@require currentCount() < 2147483647
```


Counter class ...

```
/**
 * Increment count by 1
 *
 * @require currentCount() < 2147483647
 * @ensure  currentCount() == old.currentCount() + 1
 */
public void incrementCount () {
   count = count + 1;
}
```

- This specifies a contract between the server and client such that the server guarantees to correctly increment `count` if and only if the client guarantees that `count` has not already reached the maximum.

Counter class ...

- Note that in the precondition:

```
@require currentCount() < 2147483647
```

`currentCount()` refers to the value of `count` when the method is invoked (i.e., before execution of the method).

- In the postcondition:

```
@ensure currentCount() = old.currentCount() + 1
```

`currentCount()` refers to the value of `count` when the method completes (i.e., after execution of the method) whereas `old.currentCount()` refers to the value of `count` when the method is invoked (i.e., before execution of the method).

Counter class ...

- It is important to note that it is not the responsibility of the server to check the precondition:
  - That is the responsibility of the client
- In essence, the precondition is an “escape clause” for the server
  - It allows the server to say in effect that it can do anything it wants if the client fails to ensure the validity of the precondition prior to invoking it
- Thus, **`incrementCount()`** does not need to check the value of **`count`** before it increments it
  - It can assume that **`count < 2147483647`**
- If the **`count == 2147483647`** then **`incrementCount()`** will cause it to wrap into a negative value:
  - That’s fine
  - The problem is not with **`incrementCount()`**, the problem is that the client failed to ensure **`count < 2147483647`**

Counter class ...

- Recall our implementation of the **`decrementCount()`** method:

```
/**
 * Decrement positive count by 1; zero count remains 0
 *
 * @ensure currentCount() >= 0 &&
 *         currentCount() == (old.currentCount() == 0) ?
 *         old.currentCount() :
 *         old.currentCount() - 1
 */
public void decrementCount () {
   if (count > 0)
      count = count - 1;
}
```

- For what values of **`count`** does **`decrementCount()`** work?

Counter class ...

- Because of the guard, the **`count`** is only decremented if it is positive. Otherwise, it remains unchanged.
- The **class invariant** guarantees that **`currentCount() >= 0`**.
- Since there is no need to further restrict the set of possible states that **`Counter`** can be in, we specify the precondition as **`true`** which means that any possible (legal) state of **`Counter`** is acceptable.
- A precondition of **`true`** in essence tells the client that there is actually no precondition to invoking the method – it can always be invoked.

Counter class ...

- We have the following implementation of the **`decrementCount()`** method:

```
/**
 * Decrement positive count by 1; zero count remains 0
 *
 * @require true
 * @ensure  currentCount() >= 0 &&
 *          currentCount() == (old.currentCount() == 0) ?
 *          old.currentCount() :
 *          old.currentCount() - 1
 */
public void decrementCount () {
   if (count > 0)
      count = count - 1;
}
```

Counter class ...

- Since it doesn't matter what the current value of `count` is when the `reset()` method is invoked, its precondition should be `true`:

```
/**
 * Reset the count to 0.
 *
 * @require true
 * @ensure  currentCount() == 0
 */
public void reset () {
   count = 0;
}
```


Specification Documentation

- Tools such as ***javadoc*** generate sets of HTML documents containing specifications extracted from program source files.

Explorer class

- Let's return to the `Explorer` class.
- Consider the explorer's `tolerance`.
- We will consider an explorer with a `tolerance` of 0 to be defeated. Therefore, we will restrict the `tolerance` to be a non-negative integer.
- We will designate that as a class invariant and it needs to be specified as a postcondition to its accessor method `tolerance()`.

Explorer class ...

```
private int tolerance; //current tolerance
                       //invariant:
                       //  tolerance >= 0
 ...

/**
 * Damage (hit points) required to defeat
 * this Explorer.
 *
 * @ensure result >= 0
 */
public int tolerance () {
   return tolerance;
}
```

Explorer class ...

- Now let's consider the `takeThat()` method.
- It must maintain the class invariant `tolerance >= 0`:

```
/**
 * Receive a poke of the specified number
 * of hit points.
 *
 * @ensure tolerance() >= 0
 */
public void takeThat (int hitStrength){
   …
}
```


Explorer class ...

- The class invariant `tolerance() >= 0` makes a valid postcondition, but does it really describe the state change resulting from the method?
- What is the purpose of the `takeThat()` method?
- The `takeThat()` method models the act of a `Denizen` (or whatever) poking the `Explorer`.
- When the `Explorer` is poked, his/her `tolerance` will decrease by an amount relative to the argument `hitStrength`. If we assume that `hitStrength >= 0` then we know that the `tolerance() <= old.tolerance()`.

Explorer class ...

- Since `tolerance() <= old.tolerance()` does not imply the class invariant `tolerance() >= 0`, we will add it to the postcondition:

```
/**
 * Receive a poke of the specified number
 * of hit points.
 *
 * @ensure tolerance() <= old.tolerance()
 *         && tolerance() >= 0
 */
public void takeThat (int hitStrength){
   …
}
```


Explorer class ...

- Remember that if `tolerance` reaches 0, an explorer is defeated.
- One possible implementation:

```
public void takeThat (int hitStrength) {
   if (hitStrength <= tolerance)
      tolerance = tolerance - hitStrength;
}
```

- But this rarely lets the `tolerance` reach zero.

Explorer class ...

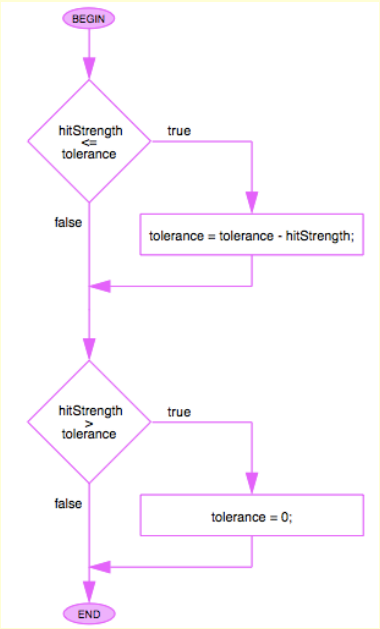
- Another possible approach:

```
public void takeThat (int hitStrength) {
    if (hitStrength <= tolerance)
        tolerance = tolerance - hitStrength;
    if (hitstrength > tolerance)
        tolerance = 0;
}
```

- What is wrong with this approach?

Explorer class ...

- It may meet the first condition and then in its changed state, meet the second condition as well.



Explorer class ...

```
/**
 * Receive a poke of the specified number
 * of hit points.
 *
 * @ensure tolerance() <= old.tolerance()
 *         && tolerance() >= 0
 */
public void takeThat (int hitStrength) {
    if (hitStrength <= tolerance)
        tolerance = tolerance - hitStrength;
    else
        tolerance = 0;
}
```


Explorer class ...

- What should we do if the constructor is called with a negative value for the parameter `tolerance`?

```
public Explorer (String name,
                 int strength,
                 int tolerance) {
    ...
  if (tolerance >= 0)
     this.tolerance = tolerance;
  else
     this.tolerance = 0;
    ...
}
```

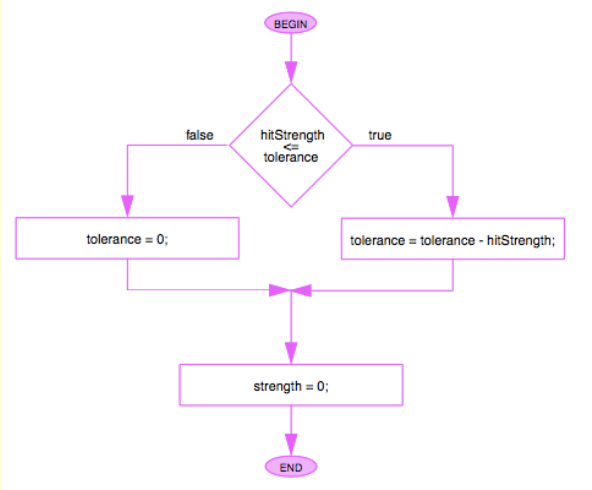
Compound Statements

- Syntax: { $statement_1$ $statement_2$ … }
- Assume that an explorer's strength should be set to 0 whenever his/her tolerance reaches 0.
- Consider the following code fragment:

```
if (hitStrength <= tolerance)
    tolerance = tolerance - hitStrength;
else
    tolerance = 0;
    strength = 0;
```

- What's wrong with this solution?
- The last statement is ***not*** part of the `else` condition.

Compound Statements …



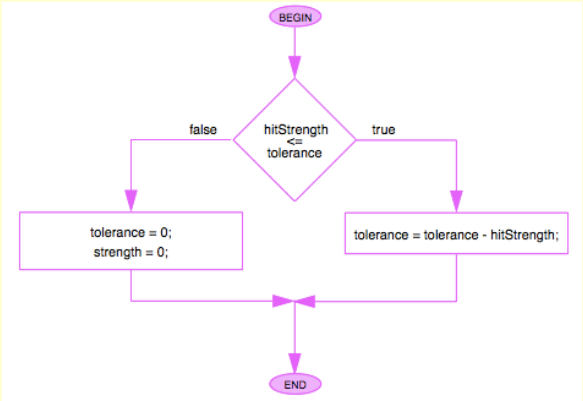
Compound Statements …

- Braces are used to create a ***block*** or ***compound statement***, which is a ***single composite statement***.

```
if (condition) {
    statement1
    …
    statementn
}
```

```
if (condition) {
    statement1
    …
    statementn
}
else {
    statement1
        …
    statementn
}
```


Compound Statements …



Compound Statements ...

```
if (hitStrength <= tolerance)
   tolerance = tolerance - hitStrength;
else {
   tolerance = 0;
   strength = 0;
}
```


Explorer class ...

- Consider the `takeThat()` method of the `Explorer` class again:

```
/**
 * Receive a poke of the specified number
 * of hit points.
 *
 * @ensure tolerance() <= old.tolerance()
 *         && tolerance() >= 0
 */
public void takeThat (int hitStrength) {
   if (hitStrength <= tolerance)
      tolerance = tolerance - hitStrength;
   else {
      tolerance = 0;
      strength = 0;
   }
}
```

- Is the postcondition really accurate?

Explorer class ...

- The postcondition accurately indicates that the explorer's strength decreases when it is attacked by a denizen, but it is not specific about how much damage is inflected.
- Should it be more specific?
- Actually, one could argue either way.
- The authors feel that it is specific enough. They're probably correct.
- But it is possible to strengthen the postcondition to make it describe the outcome of the attack more precisely. Consider the following postcondition:

```
@ensure tolerance() == old.tolerance() - hitstrength
```

- What's wrong with this postcondition?

Explorer class ...

- This postcondition has a problem if hitStrength exceeds the explorer's tolerance. In this situation, the class invariant (which insists that tolerance >= 0) would be violated.
- We can correct this problem and describe the outcome of the attack more precisely with the following postcondition:

```
@ensure tolerance() == (hitStrength <= old.tolerance()) ?
                       old.tolerance() - hitStrength : 0
```

- So we can be more explicit if we want to be, but we'll return to our original postcondition to allow our implementation more flexibility.
  - Maybe our explorer is wearing a magic cape!

Explorer class ...

- So back to the `takeThat()` method:

```
/**
 * Receive a poke of the specified number
 * of hit points.
 *
 * @ensure tolerance() <= old.tolerance()
 *         && tolerance() >= 0
 */
public void takeThat (int hitStrength) {
   if (hitStrength <= tolerance)
      tolerance = tolerance - hitStrength;
   else {
      tolerance = 0;
      strength = 0;
   }
}
```

- Should `hitStrength <= tolerance` be a precondition?

Explorer class ...

- If we make `hitStrength <= tolerance` the precondition to `takeThat()` then it is not possible for a denizen to attack an explorer with a poke that is stronger than really necessary.
- That is probably is too restrictive.
- But clearly the `hitStrength` must not be negative.
- Does it make sense to allow `hitStrength` to be 0?
- Actually, one can argue that `hitStrength` must be strictly positive, but it is also reasonable to allow a poke with `hitStrength` of 0 – it's just a futile attempt by the denizen to poke the explorer.

Explorer class ...

- So let's make the precondition to `takeThat()` be `hitStrength >= 0`:

```
/**
 * Receive a poke of the specified number
 * of hit points.
 *
 * @require hitStrength >= 0
 * @ensure  tolerance() <= old.tolerance()
 *          && tolerance() >= 0
 */
public void takeThat (int hitStrength) {
   if (hitStrength <= tolerance)
      tolerance = tolerance - hitStrength;
   else {
      tolerance = 0;
      strength = 0;
   }
}
```


Explorer class ...

- Consider the `poke()` method:

```
/**
 * Poke the specified Denizen.
 *
 * @require ???
 * @ensure  ???
 */
public void poke (Denizen opponent) {
   opponent.takeThat(strength);
}
```

- What would be an appropriate precondition and postcondition for this method?

Explorer class ...

- What would be an appropriate precondition and postcondition for the other methods of the **Explorer** class?
  - **name()**
  - **strength()**
  - **tolerance()**
- What about the constructor for **Explorer**?

Explorer class ...

```
/**
 * Create a new Explorer with specified name,
 * strength, and tolerance.
 *
 * @require strength >= 0 && tolerance >= 0
 * @ensure  name() == name &&
 *          strength() == strength &&
 *          tolerance() == tolerance
 */
public Explorer (String name,
                 int strength,
                 int tolerance) {
   this.name = name;
   this.strength = strength;
   this.tolerance = tolerance;
}
```


Using the Debugger in DrJava

- Let's bring up Explorer.java in DrJava
- PP. 185-186

A lock example

- We want to model a simple lock with an integer combination
- A combination is set into the lock when it is created
- To open a closed lock, the client must provide the correct combination
- A lock must know its combination and whether it is locked or unlocked
- It must be able to lock itself and also unlock itself when it is provided with the proper combination
- We will define a single class **CombinationLock**

CombinationLock Responsibilities

- Know:
  - the combination
  - whether opened or closed (i.e., unlocked or locked)
- Do:
  - close (lock)
  - open (unlock), when given proper combination

CombinationLock Responsibilities ...

- Class: **CombinationLock**
- Query:
  - **isOpen** whether or not the lock is open
- Commands:
  - **close** lock the lock
  - **open** unlock the lock (*combination*)
- Notice that we did not supply a query for the combination. Why not?

CombinationLock Specification

- Component variables:

```
private int combination; // lock's combination
                         // invariant:
                         //    0 <= combination &&
                         //    combination <= 999

private boolean isOpen;  // the lock is unlocked
```


CombinationLock Specification ...

- Let's consider the constructor
- What should the initial state be for a **CombinationLock**?
- The constructor specification leaves a couple questions:
  - Is any integer a legal combination?
  - Does a newly minted lock start life opened or closed?
- Let's assume that a lock should initially be open
  - We can express this as a postcondition
- Let's further assume that only combinations in the range 0-999 are legal
  - We can express this as a precondition (and it will be a class invariant as well)

CombinationLock Specification ...

- Constructor:

```
/**
 * Create a lock with the specified
 * combination.
 *
 * @require 0 <= combination &&
 *          combination <= 999
 * @ensure  isOpen()
 */
public CombinationLock (int combination)
```


CombinationLock Specification ...

- Query:

```
/**
 * This CombinationLock is unlocked.
 *
 * @require true
 * @ensure  true
 */
public boolean isOpen ()
```


CombinationLock Specification ...

- Commands:

```
/**
 * Lock this CombinationLock.
 *
 * @require true
 * @ensure  !isOpen()
 */
public void close ()
```


CombinationLock Specification ...

- Commands:

```
/**
 * Unlock this CombinationLock if the correct
 * combination is provided.
 *
 * @require 0 <= combinationToTry &&
 *          combinationToTry <= 999
 * @ensure  isOpen() == (old.isOpen() ||
 *          combinationToTry == combination)
 */
public void open (int combinationToTry)
```

CombinationLock Specification ...

- Note that we needed to specify a private variable in the postcondition
- Can we avoid this?
  - We certainly don't want to make the combination to the lock publicly available to the client via an accessor, …
  - So we either need to have a ***design variable*** that talks about the concept of the combination, or …
  - We can use a private data area since the client is aware of the combination, he/she simply can't access or change its value

CombinationLock Specification ...

- What is the purpose of an accessor?
  - It enables the client to access the value of a private instance variable
  - It provides a way to specify the value of a private data area in the precondition
    - Note that although we can use a private data area in the specification of the postcondition, we cannot use a private data area in the specification of the precondition
    - Clients need to be able to test the validity of the precondition in order to prevent invoking the method in an invalid state
    - Clients do not need to validate the correctness of the postcondition

CombinationLock Implementation

```
/**
 * Create a lock with the specified
 * combination.
 *
 * @require 0 <= combination &&
 *          combination <= 999
 * @ensure  isOpen()
 */
public CombinationLock (int combination) {
   this.combination = combination;
   isOpen = true;
}
```


CombinationLock Implementation ...

```
/**
 * Indicates if CombinationLock is open.
 *
 * @require true
 * @ensure  true
 */
public boolean isOpen () {
   return isOpen;
}
```

CombinationLock Implementation ...

```
/**
 * Lock this CombinationLock.
 *
 * @require true
 * @ensure  !isOpen()
 */
public void close () {
   isOpen = false;
}
```


CombinationLock Implementation ...

```
/**
 * Unlock this CombinationLock if the correct
 * combination is provided.
 *
 * @require 0 <= combinationToTry &&
 *          combinationToTry <= 999
 * @ensure  isOpen() == (old.isOpen() ||
 *          combinationToTry == combination)
 */
public void ope   (i  combinationToTry) {
  isOpen = combi    on == combinationToTry;
}
```

- What is wrong with this implementation?

CombinationLock Implementation ...

```
/**
 * Unlock this CombinationLock if the correct
 * combination is provided.
 *
 * @require 0 <= combinationToTry &&
 *          combinationToTry <= 999
 * @ensure  isOpen() == (old.isOpen() ||
 *          combinationToTry == combination)
 */
public void open (int combinationToTry) {
   if (combination == combinationToTry)
       isOpen = true;
}
```


Preconditions

- Preconditions must be satisfied by the client when invoking the method:
  - Preconditions are usually used to constrain values that the client can provide as arguments when invoking a method.
  - Occasionally, preconditions are also used to constrain the order in which methods can be invoked or require that an object be in a certain state before a given method can be invoked.

Postconditions

- Postconditions are guarantees made by the server when its method is invoked:
  - Query postconditions generally provide a value to the client using the `result` "keyword".
  - Command postconditions typically describe the new state of the object.
  - Constructor postconditions typically describe the initial state of the newly created object.
- Preconditions and postconditions are part of the specification, forming a **contract** between the client and the server.

`assert` statements

- The `assert` statement was added to Java in release 1.4.
- It can be used to verify preconditions at runtime.
- There are two formats for the `assert` statement:

```
assert booleanExpression;

assert booleanExpression : expression;
```


`assert` statements ...

- Consider the constructor for **`Explorer`** again:

```
/**
 * Create a new Explorer with specified name,
 * strength, and tolerance.
 *
 * @require strength >= 0 && tolerance >= 0
 * @ensure  this.name == name &&
 *          this.strength == strength &&
 *          this.tolerance == tolerance
 */
public Explorer (String name,
                 int strength,
                 int tolerance) {
   assert strength >= 0;
   assert tolerance >= 0 : "precondition: tolerance (" + tolerance + ") >= 0";
   this.name = name;
   this.strength = strength;
   this.tolerance = tolerance;
}
```


`assert` statements ...

- Assertions must be explicitly enabled with command line switches.
- Because of the possibility that a program might be run without precondition testing, some programmers prefer to test preconditions explicitly with *if* statements.
- An *if* statement implies an ordinary, expected case that must be handled by the program.
- A precondition failure, on the other hand, is an error and occurs only in an incorrect program.

`assert` statements ...

- Let's bring up DrJava
- p. 240