In the above example the error mode isn't strictly necessary, **but it is advised to add it**. This way the script will not stop with a `Fatal Error` when something goes wrong. And it gives the developer the chance to `catch` any error(s) which are `thrown` as `PDOExceptions`.
What is **mandatory**, however, is the first `setAttribute()` line, which tells PDO to disable emulated prepared statements and use *real* prepared statements. This makes sure the statement and the values aren't parsed by PHP before sending it to the MySQL server (giving a possible attacker no chance to inject malicious SQL).
Although you can set the `charset` in the options of the constructor, it's important to note that 'older' versions of PHP (< 5.3.6) silently ignored the charset parameter in the DSN.

Explanation

What happens is that the SQL statement you pass to `prepare` is parsed and compiled by the database server. By specifying parameters (either a `?` or a named parameter like `:name` in the example above) you tell the database engine where you want to filter on. Then when you call `execute`, the prepared statement is combined with the parameter values you specify.
The important thing here is that the parameter values are combined with the compiled statement, not an SQL string. SQL injection works by tricking the script into including malicious strings when it creates SQL to send to the database. So by sending the actual SQL separately from the parameters, you limit the risk of ending up with something you didn't intend. Any parameters you send when using a prepared statement will just be treated as strings (although the database engine may do some optimization so parameters may end up as numbers too, of course). In the example above, if the `$name` variable contains `'Sarah'; DELETE FROM employees` the result would simply be a search for the string `"'Sarah'; DELETE FROM employees"`, and you will not end up with an empty table.
Another benefit of using prepared statements is that if you execute the same statement many times in the same session it will only be parsed and compiled once, giving you some speed gains.

Oh, and since you asked about how to do it for an insert, here's an example (using PDO):

```
$preparedStatement = $db->prepare('INSERT INTO table (column) VALUES
(:column)');

$preparedStatement->execute(array('column' => $unsafeValue));
```

Can prepared statements be used for dynamic queries?

While you can still use prepared statements for the query parameters, the structure of the dynamic query itself cannot be parametrized and certain query features cannot be parametrized.

For these specific scenarios, the best thing to do is use a whitelist filter that restricts the possible values.

```
// Value whitelist
// $dir can only be 'DESC' otherwise it will be 'ASC'
if (empty($dir) || $dir !== 'DESC') {
    $dir = 'ASC';
}
```