```
public class Purse {

    final int MAX_BALANCE;
    int balance;                                                          1
    //@ invariant 0 <= balance && balance <= MAX_BALANCE;

    byte[] pin;
    /*@ invariant pin != null && pin.length == 4
      @           && (\forall int i; 0 <= i && i < 4;
      @                              0 <= pin[i] && pin[i] <= 9);         2
      @*/

    /*@ requires   amount >= 0;
      @ assignable balance;
      @ ensures    balance == \old(balance) - amount
      @            && \result == balance;                                 3
      @ signals (PurseException) balance == \old(balance);
      @*/
    int debit(int amount) throws PurseException {
        if (amount <= balance) { balance -= amount; return balance; }
        else { throw new PurseException("overdrawn by " + amount); }
    }

    /*@ requires   p != null && p.length >= 4;
      @ assignable \nothing;
      @ ensures    \result <==> (\forall int i; 0 <= i && i < 4;          4
      @                                         pin[i] == p[i]);
      @*/
    boolean checkPin(byte[] p) {
        boolean res = true;
        for (int i=0; i < 4; i++) { res = res && pin[i] == p[i]; }
        return res;
    }

    /*@ requires   0 < mb && 0 <= b && b <= mb
      @            && p != null && p.length == 4                          5
      @            && (\forall int i; 0 <= i && i < 4;
      @                               0 <= p[i] && p[i] <= 9);
      @ assignable MAX_BALANCE, balance, pin;
      @ ensures    MAX_BALANCE == mb && balance == b
      @            && (\forall int i; 0 <= i && i < 4; p[i] == pin[i]);
      @*/
    Purse(int mb, int b, byte[] p) {
        MAX_BALANCE = mb; balance = b; pin = (byte[]) p.clone();
    }
}
```

Fig. 1. Example JML specification

ning the Java code and testing for violations of JML assertions. Such runtime assertion checks are accomplished by using the JML compiler *jmlc* (Section 4.1).

Given that one often wants to do runtime assertion checking in the testing phase, there is also a *jmlunit* tool (Section 4.2), which combines runtime assertion checking with unit testing.

3.2 Static checking and verification

More ambitious than testing if the code satisfies the specifications at runtime is verifying that the code satisfies its specification statically. This can give more assurance in the correctness of code as it establishes the correctness for all possible execution paths, whereas runtime assertion checking is limited by the execution paths exercised by the test suite being used. Of course, correctness of a program with respect to a given specification is not decidable in general. Any verification tool must trade