Preconditions and Postconditions

A **precondition** is a requirement that the caller of a method must obey. For example, the deposit method of the BankAccount class has a precondition that the amount to be deposited should not be negative. It is the responsibility of the caller never to call a method if one its preconditions is violated. If the method is called anyway, it is not responsible for producing a correct result.

Therefore, a precondition is an important part of the method, and you must document it. Here we document the precondition that the amount parameter must not be negative.

```java
/**
 * Deposits money into this account.
 * @param amount the amount of money to deposit.
 * (precondition: amount >= 0)
 */
```

Some javadoc extensions support a `@precondition` or `@requires` tag, but it is not a part of the standard javadoc program. Because the standard javadoc tool skips all unknown tags, we simply add the precondition to the method explanation or the appropriate `@param` tag.

Preconditions are typically stated for one of two reasons:
- To restrict the parameters of a method
- To require that a method is only called when it is in the appropriate state

For example, once a StringTokenizer has run out of tokens, it is no longer legal to call the `nextToken` method. Thus, a precondition of the `nextToken` methods method is that the `hasMoreTokens` method return true.

A method is responsible only for operating correctly when its caller has fulfilled all preconditions. The method is free to do anything if a precondition is not fulfilled. It would be perfectly legal if the method reformatted the hard disk every time it was called with a wrong input. Naturally, that isn’t reasonable. What should a method actually do when it is called with inappropriate inputs? For example, what should `account.deposit(-1000)` do?

There are two good choices.

1. A method can check for the violation and throw an exception. Then the method does not return to its caller, instead, control is transferred to an exception handler. If no handler is present, then the program terminates.
2. If it is too cumbersome to carry out the check, the method can simply work under the assumption that the preconditions are fulfilled. If they aren’t, then any data corruption (such as a negative balance) or other failures are the caller’s fault.

Later we will see how to program exceptions handlers. Right now, we’ll show you how to throw an exception to indicate that a method has been called with inappropriate parameters.

```java
public double deposit(double amount)
{
    if (amount < 0)
    {
        throw new IllegalArgumentException();
        balance = balance + amount;
    }
}
```

When the method is called with an illegal argument, the program aborts with an error message.
java.lang.IllegalArgumentException at BankAccount.deposit(BankAccount.java:14)

If a method is called when the object is in an an inappropriate state, you can throw an `IllegalStateException` instead.

Many beginning programmers think that it isn't "nice" to abort the program. Why not simply return to the caller instead?

```java
class BankAccount {
    public double deposit(double amount) {
        if (amount < 0)
            return; // not as good as throwing an exception
        balance = balance + amount;
    }
}
```

That is legal – after all, a method can do anything if its preconditions are violated. But it is not good as throwing an exception. If the program called the deposit method has a few bugs that cause it to pass a negative amount as an input value, then the version that throws the exception will make the bugs very obvious during testing – it is had to ignore when the program aborts. The quiet version, on the other hand, will not alert you, and you may not notice that it performs some wrong calculations as a consequence. think of exceptions as the "tough love" approach to precondition checking.

If you test for the precondition, then you might as well throw an exception. However, sometimes the cost of testing is too high. It might slow down the method too much, or it might make it too confusing. Then you may simply go ahead and assume that the precondition is fulfilled:

```java
class BankAccount {
    public double deposit(double amount) {
        // no test, for maximum efficiency
        // if this makes the balance negative; it's the caller's fault
        balance = balance + amount;
    }
}
```

When a method is called in accordance with its precondition, then the method promises to do its job correctly. A different kind of promise that the method makes is called a postcondition. There are tow kinds of postconditions:

1. That return value is computed correctly
2. That the object is in a certain state after the method call is completed

Here is a postcondition that makes a statement about the object state after the deposit method is called.

```java
/**
   * Deposits money into this account.
   * (Postcondition: getBalance() >= 0)
   * @param amount the amount of money to deposit.
   * (precondition: amount >= 0)
   */
```

As long as the precondition is fulfilled, this method guarantees that the balance after the deposit is not negative.
Some javadoc extensions support @postcondition or @ensures tag. However, just as with preconditions, we simply add postconditions to the method explanation or the @return tag, because the standard javadoc program skips all tags that it doesn't know.

Some programmers feel that they must specify a postcondition for every method. When you use javadoc, however, you already specify a part of the postcondition in the @return tag, and you shouldn't repeat it in a postcondition.

```java
// this postcondition statement is overly repetitive
/**
 * Returns the current balance of this account.
 * @return the account balance
 * (Postcondition: the return value equals the account balance)
 */
```

Note that we formulate pre- and postconditions only in terms of the interface of the class. Thus, we state the precondition of the withdraw method as amount <= getBalance(), not amount <= balance. After all, the caller, which needs to check the precondition, has access only to the public interface, not the private implementation.