Wouldn’t it be nice if all Java classes that you use, including your own, lived up to their promises? In fact, wouldn’t it be nice if you actually knew exactly what a given class promises? If you agree, read on -- Design by Contract and iContract come to the rescue.

Note: The code source for the examples in this article can be downloaded from Resources.

Design by Contract
The Design by Contract (DBC) software development technique ensures high-quality software by guaranteeing that every component of a system lives up to its expectations. As a developer using DBC, you specify component contracts as part of the component’s interface. The contract specifies what that component expects of clients and what clients can expect of it. Bertrand Meyer developed DBC as part of his Eiffel programming language. Regardless of its origin, DBC is a valuable design technique for all programming languages, including Java.

Central to DBC is the notion of an assertion -- a Boolean expression about the state of a software system. At runtime we evaluate the assertions at specific checkpoints during the system’s execution. In a valid software system, all assertions evaluate to true. In other words, if any assertion evaluates to false, we consider the software system invalid or broken.

DBC's central notion somewhat relates to the #assert macro in C and C++ programming language. However DBC takes assertions a zillion levels further.

In DBC, we identify three different kinds of expressions:

- Preconditions
- Postconditions
- Invariants

Let’s examine each in more detail.

Preconditions
Preconditions specify conditions that must hold before a method can execute. As such, they are evaluated just before a method executes. Preconditions involve the system state and the arguments passed into the method.

Preconditions specify obligations that a client of a software component must meet before it may invoke a particular method of the component. If a precondition fails, a bug is in a software component’s client.

Postconditions
In contrast, postconditions specify conditions that must hold after a method completes. Consequently, postconditions are executed after a method completes. Postconditions involve the old system state, the new system state, the method arguments, and the method’s return value. Postconditions specify guarantees that a software component makes to its clients. If a
postcondition is violated, the software component has a bug.

**Invariants**

An invariant specifies a condition that must hold anytime a client could invoke an object's method. Invariants are defined as part of a class definition. In practice, invariants are evaluated anytime before and after a method on any class instance executes. A violation of an invariant may indicate a bug in either the client or the software component.

**Assertions, inheritance, and interfaces**

All assertions specified for a class and its methods apply to all subclasses as well. You can also specify assertions for interfaces. As such, all assertions of an interface must hold for all classes that implement the interface.

**iContract -- DBC with Java**

So far, we have talked about DBC in general. You probably have some idea by now what I am talking about, but if you are new to DBC, things might still be a bit foggy.

In this section, things will become more concrete. iContract, developed by Reto Kamer, adds constructs to Java that allow you to specify the DBC assertions we talked about earlier.

**iContract basics**

iContract is a preprocessor for Java. To use it, you first process your Java code with iContract, producing a set of decorated Java files. Then you compile the decorated Java code as usual with the Java compiler.

All iContract directives in Java code reside in class and method comments, just like Javadoc directives. In this way, iContract ensures complete backwards-compatibility with existing Java code, and you can always directly compile your Java code without the iContract assertions.

In a typical program lifecycle, you would move your system from a development environment into a test environment, then into a production environment. In the development environment, you would instrument your code with iContract assertions and run it. That way you can catch newly introduced bugs early on. In the test environment you may still want to keep the bulk of the assertions enabled, but you should take them out of performance-critical classes. Sometimes it even makes sense to keep some assertions enabled in a production environment, but only in classes that definitely are in no way critical to your system's performance. iContract allows you to explicitly select the classes that you want to instrument with assertions.

**Preconditions**

In iContract, you place preconditions in a method header using the @pre directive. Here's an example:

```java
/**
 * @pre f >= 0.0
 */
public float sqrt(float f) { ... }
```

The example precondition ensures that the argument \(f\) of function \(\text{sqrt}()\) is greater than or equal to zero. Clients who use that method are responsible for adhering to that precondition. If they don't, we as implementors of \(\text{sqrt}()\) are simply not responsible for the consequences.

The expression after the @pre is a Java Boolean expression.

**Postconditions**

Postconditions are likewise added to the header comment of the method they belong to. In iContract, the @post directive defines postconditions:

```java
/**
 * @pre f >= 0.0
 * @post Math.abs((return * return) - f) < 0.001
 */
public float sqrt(float f) { ... }
```

In our example, we have added a postcondition that ensures that the \(\text{sqrt}()\) method calculates the square root of \(f\) within a specific margin of error (+/- 0.001).

iContract introduces some specific notations for postconditions. First of all, \(\text{return}\) stands for the return value of the method. At runtime, that will be replaced by the method's return value.
Within postconditions, there often exists a need to differentiate between the value of an argument before execution of the method and afterwards, supported in iContract with the \texttt{@pre} operator. If you append \texttt{@pre} to an expression in a postcondition, it will be evaluated based on the system state before the method executes:

```java
/**
 * Append an element to a collection.
 * @post c.size() = c@pre.size() + 1
 * @post c.contains(o)
 */
public void append(Collection c, Object o) { ... }
```

In the code above, the first postcondition specifies that the size of the collection must grow by 1 when we append an element. The expression \texttt{c@pre} refers to the collection \texttt{c} before execution of the \texttt{append} method.

**Invariants**

With iContract, you can specify invariants in the header comment of a class definition:

```java
/**
 * A PositiveInteger is an Integer that is guaranteed to be positive.
 * @inv intValue() > 0
 */
class PositiveInteger extends Integer {
    ...
}
```

In this example, the invariant guarantees that the \texttt{PositiveInteger}'s value is always greater than or equal to zero. That assertion is checked before and after execution of any method of that class.

**Object Constraint Language (OCL)**

Although the assertion expressions in iContract are valid Java expressions, they are modeled after a subset of the Object Constraints Language (OCL). OCL is one of the standards maintained and coordinated by the Object Management Group, or OMG. (OMG takes care of CORBA and related stuff, in case you miss the connection.) OCL was intended to specify constraints within object modeling tools that support the Unified Modeling Language (UML), another standard guarded by OMG.

Because the iContract expressions language is modeled after OCL, it provides some advanced logical operators beyond Java's own logic operators.

**Quantifiers: forall and exists**

iContract supports \texttt{forall} and \texttt{exists} quantifiers. The \texttt{forall} quantifier specifies that a condition should hold true for every element in a collection:

```java
/*
 * @invariant forall IEmployee e in getEmployees() |
 *           getRooms().contains(e.getOffice())
 */
```

The above invariant specifies that every employee returned by \texttt{getEmployees()} has an office in the collection of rooms returned by \texttt{getRooms()}. Except for the \texttt{forall} keyword, the syntax is the same as that of an \texttt{exists} expression.

Here is an example using \texttt{exists}:

```java
/**
 * ...
 */
```
That postcondition specifies that after the associated method executes, the collection returned by
getRooms() will contain at least one available room. The exists proceeds the Java type of the
collection element -- IRoom in the example. \( r \) is a variable that refers to any element in the
collection. The in keyword is followed by an expression that returns a collection (Enumeration,
Array, or Collection). That expression is followed by a vertical bar, followed by a condition
involving the element variable, \( r \) in the example. Employ the exists quantifier when a condition
must hold true for at least one element in a collection.

Both forall and exists can be applied to different kinds of Java collections. They support
Enumerations, Arrays, and Collections.

**Implications: implies**
iContract provides the implies operator to specify constraints of the form, "If A holds, then B
must hold as well." We say, "A implies B." Example:

```java
/**
 * @invariant getRooms().isEmpty() implies getEmployees().isEmpty() // no
 * rooms, no employees
 */
```

That invariant expresses that when the getRooms() collection is empty, the getEmployees()
collection must be empty as well. Note that it does not specify that when getEmployees() is
empty, getRooms() must be empty as well.

You can also combine the logical operators just introduced to form complex assertions. Example:

```java
/**
 * @invariant forall IEmployee e1 in getEmployees() |
 * forall IEmployee e2 in getEmployees() |
 * (e1 != e2) implies e1.getOffice() != e2.getOffice() // a single office per
 * employee
 */
```

**Constraints, inheritance, and interfaces**
iContract propagates constraints along inheritance and interface implementation relationships
between classes and interfaces.

Suppose class \( \texttt{B} \) extends class \( \texttt{A} \). Class \( \texttt{A} \) defines a set of invariants, preconditions, and
postconditions. In that case, the invariants and preconditions of class \( \texttt{A} \) apply to class \( \texttt{B} \) as well,
and methods in class \( \texttt{B} \) must satisfy the same postconditions that class \( \texttt{A} \) satisfies. You may add
more restrictive assertions to class \( \texttt{B} \).

The aforementioned mechanism works for interfaces and implementations as well. Suppose \( \texttt{A} \) and
\( \texttt{B} \) are interfaces and class \( \texttt{C} \) implements both. In that case, \( \texttt{C} \) is subject to invariants,
preconditions, and postconditions of both interfaces, \( \texttt{A} \) and \( \texttt{B} \), as well as those defined directly in
class \( \texttt{C} \).

**Beware of side effects!**
iContract will improve the quality of your software by allowing you to catch many possible bugs
eyearly on. But you can also shoot yourself in the foot (that is, introduce new bugs) using iContract.
That can happen when you invoke functions in your iContract assertions that engender side effects
that alter your system's state. That leads to unpredictable behavior because the system will behave
differently once you compile your code without iContract instrumentation.

**The stack example**
Let us take a look at a complete example. I have defined the Stack interface, which defines the
familiar operations of my favorite data structure:
/**
 * @inv !isEmpty() implies top() != null // no null objects are allowed
 */
public interface Stack
{
    /**
    * @pre o != null
    * @post !isEmpty()
    * @post top() == o
    */
    void push(Object o);

    /**
    * @pre !isEmpty()
    * @post @return == top@pre
    */
    Object pop();

    /**
    * @pre !isEmpty()
    */
    Object top();

    boolean isEmpty();
}

We provide a simple implementation of the interface:

import java.util.*;

/**
 * @inv isEmpty() implies elements.size() == 0
 */
public class StackImpl implements Stack
{
    private final LinkedList elements = new LinkedList();

    public void push(Object o)
    {
        elements.add(o);
    }

    public Object pop()
    {
        final Object popped = top();
        elements.removeLast();
        return popped;
    }

    public Object top()
    {
        return elements.getLast();
    }

    public boolean isEmpty()
    {
As you can see, the Stack implementation does not contain any iContract assertions. Rather, all assertions are made in the interface, meaning that the component contract of the Stack is defined in the interface in its entirety. Just by looking at the Stack interface and its assertions, the Stack's behavior is fully specified.

Now we add a small test program to see iContract in action:

```java
public class StackTest {
    public static void main(String[] args) {
        final Stack s = new StackImpl();
        s.push("one");
        s.pop();
        s.push("two");
        s.push("three");
        s.pop();
        s.pop();
        s.pop(); // causes an assertion to fail
    }
}
```

Next, we run iContract to build the stack example:

```
java -cp %CLASSPATH%;src;_contract_db;instr
com.reliablesystems.iContract.Tool -Z -a -v -minv,pre,post
>b"javac -classpath %CLASSPATH%;src" -c"javac -classpath %CLASSPATH%;instr" > -n"javac -classpath %CLASSPATH%;_contract_db;instr" -oinstr/@p/@f.@e -k_contract_db/@p src/*.java
```

The statement above warrants a little bit of an explanation.

- You invoke iContract, a command line tool written in Java, by starting the class `com.reliablesystems.iContract.Tool`.
- The argument `-minv,pre,post` specifies that we want to instrument our code with invariants and preconditions as well as postconditions.
- The `-b"javac -classpath %CLASSPATH%;src"` option specifies the command to use to compile Java classes before iContract instruments them. iContract needs to do that because it uses Java reflection to extract some of the properties of your Java classes. So in order to do its job, it first needs a compiled version of the uninstrumented Java class.
- The `-c"javac -classpath %CLASSPATH%;instr"` option specifies the Java compiler command to compile the instrumented Java files.
- The `-n"javac -classpath %CLASSPATH%;_contract_db;instr"` option specifies the Java compiler to compile the repository classes. iContract uses a set of Java classes as a repository to resolve inheritance and interface implementation relationships.
- The option `-oinstr/@p/@f.@e` specifies the location of the instrumented Java files.

As you can see, iContract commands can be quite involved, so it’s best to create a small build script for every project.

After you execute that command, we will have two versions of the compiled class files of our sample project. The `src` directory will contain the uninstrumented class, while the `instr` directory will contain the instrumented version.
We can now run our example, then see the assertions in action. Let's run the uninstrumented version first:

```
$ java -cp ./src StackTest
Exception in thread "main" java.util.NoSuchElementException
  at java.util.LinkedList.getLast(LinkedList.java:107)
  at StackImpl.top(StackImpl.java:24)
  at StackImpl.pop(StackImpl.java:17)
  at StackTest.main(StackTest.java:14)
```

As you can see, something went terribly wrong! Let's see if the instrumented code can shed some light.

```
$ java -cp ./instr StackTest
Exception in thread "main" java.lang.RuntimeException:
  java.lang.RuntimeException: src/StackImpl.java:22: error:
  precondition violated (StackImpl::top()): (/*declared in Stack::top()*/ (!isEmpty()))
    at StackImpl.top(StackImpl.java:210)
    at StackImpl.pop(StackImpl.java:124)
    at StackTest.main(StackTest.java:15)
```

Well, that's better, isn't it? We now know exactly what went wrong. The precondition !isEmpty() of the StackImpl.top() method was violated. We were calling top() on an empty Stack.

**Conclusion**

I hope that you feel our little journey into DBC in Java was worthwhile. You should have a good idea of how useful iContract as a tool is to your development efforts. Please be aware that iContract is still in beta (and has been for some time). Some bugs in the tool are laid out on the iContract Webpage.

You'll want to check out a number of iContract supporting tools:

- **iDoclet** integrates iContract assertions into the Javadoc documentation
- **iDarwin** builds on top of iContract to add constraints at the architectural level of software projects

**About the author**

Oliver Enseling is the principal Java consultant of Martian Software. Programming has been his passion since he was sixteen. He is a Sun certified Java developer and works on distributed systems development in Minnesota's Twin Cities area.

**Resources**

- To download the complete source code for this article, please visit: [http://www.javaworld.com/jw-02-2001/cooltools/jw-0216-cooltools.zip](http://www.javaworld.com/jw-02-2001/cooltools/jw-0216-cooltools.zip)
- For a freewheeling discussion on programming theory, visit JavaWorld's Programming Theory & Practice discussion, moderated by the intrepid Allen Holub: [http://www.itworld.com/jump/jw-0216-cooltools/forums.itworld.com/webx?14@@.ee6b806](http://www.itworld.com/jump/jw-0216-cooltools/forums.itworld.com/webx?14@@.ee6b806)
- Sign up for the JavaWorld This Week free weekly email newsletter and keep up with what's new at JavaWorld: [http://www.idg.net/jw-subscribe](http://www.idg.net/jw-subscribe)
- For more on iContract, go directly to the source: [http://www.reliable-systems.com/tools/iContract/iContract.htm](http://www.reliable-systems.com/tools/iContract/iContract.htm)
Download JavaCC from Reliable Systems' Website:

- Get an intro to Design by Contract at:
- For another introduction to DBC, visit:
  http://www.awl.com/cseng/titles/0-201-89542-0/techniques/designByContract.htm
  http://www.amazon.com/exec/obidos/ASIN/0136291554/javaworld
- For more on the Eiffel programming language, go to:
  http://www.eiffel.com/
- Read up on DBC in Java at:
  http://w3.one.net/~jweirich/java/javadbc/java-dbc.html
- Get an idea on OCL at:
- Download iDoclet at:
  http://home.sol.no/~hellesoy/icplus.html
- Download iDarwin at: