Oracle9*i*

JPublisher ユーザーズ・ガイド

リリース 2（9.2）

2002 年 7 月

部品番号：J06307-01

ORACLE®

Oracle9*i* JPublisher ユーザーズ・ガイド , リリース 2（9.2）

部品番号：J06307-01

原本名：Oracle9*i* JPublisher User's Guide Release 2 (9.2)

原本部品番号：A96658-01

原本著者：Brian Wright, Ekkehard Rohwedder, Thomas Pfaeffle, P. Alan Thiesen

原本協力者：Janice Nygard, Quan Wang, Prabha Krishna, Ellen Barnes

目次

2 JPublisher の概念

3 コマンドライン・オプションおよび入力ファイル

4 JPublisher の例

索引

はじめに

ここでは、このマニュアルの対象読者、構成および表記規則について説明します。また、関連する Oracle マニュアルのリストも記載されています。

JPublisher ユーティリティによって、ユーザー定義の SQL オブジェクト型と PL/SQL パッケージが Java クラスに変換されます。アプリケーションで、データベースのオブジェクト型、VARRAY 型、ネストした表型、オブジェクト参照型、OPAQUE 型または PL/SQL パッケージに対応する Java クラスを必要とする SQLJ プログラマ、JDBC プログラマおよび J2EE プログラマが、この JPublisher ユーティリティを使用します。

この章の内容は、次のとおりです。

対象読者

このマニュアルは、アプリケーションでオブジェクト型、VARRAY 型、ネストした表型、オブジェクト参照型、OPAQUE 型または PL/SQL パッケージに対応する Java クラスを必要とする JDBC プログラマおよび SQLJ プログラマを対象としています。

また、Oracle データベース、SQL、PL/SQL、JDBC および SQLJ の知識と実務経験を持つ Java プログラマを対象としています。全般的な知識で十分ですが、Oracle 固有の機能に関する知識があれば役立ちます。

このマニュアルの構成

このマニュアルの構成は、次のとおりです。

第 1 章「JPublisher の概要」

JPublisher ユーティリティの概要と例、このリリースでの新機能および JPublisher 操作の概要について説明します。

第 2 章「JPublisher の概念」

データ型マッピング、出力クラスの生成、継承のサポート、移行と下位互換性および JPublisher の制限事項など、JPublisher の概念と使用方法全般のバックグラウンドと詳細について説明します。

第 3 章「コマンドライン・オプションおよび入力ファイル」

JPublisher のコマンドライン構文、コマンドライン・オプションおよび入力ファイル形式の詳細について説明します。

第 4 章「JPublisher の例」

各種のオブジェクト型、ラッパー・メソッドの使用例とその出力について説明します。

関連文書

- 『Oracle9*i* Java Developer's Guide』

  このマニュアルでは、Oracle9*i* における Java の基本概念と、サーバー側の構成および機能性の概要について説明しています。このマニュアルには、JDBC や SQLJ などの特定の製品ではなく、Oracle データベースの Java 環境全般に関する情報が記載されています。

- 『Oracle9*i* JDBC 開発者ガイドおよびリファレンス』

  このマニュアルでは、JDBC（Java Database Connectivity）標準の Oracle 実装のプログラミング構文と機能について説明しています。Oracle JDBC ドライバの概要、JDBC 1.22、2.0 および 3.0 機能の Oracle の実装の詳細、Oracle JDBC の型の拡張とパフォーマンスの拡張の説明が記載されています。

- 『Oracle9*i* SQLJ 開発者ガイドおよびリファレンス』

  このマニュアルでは、SQLJ を使用して静的 SQL 操作を Java コードに直接埋め込む方法、SQLJ 言語構文および SQLJ Translator のオプションと機能について説明しています。標準的な SQLJ 機能と Oracle 固有の SQLJ 機能の両方が記載されています。

- 『Oracle9*i* JavaServer Pages サポート・リファレンス』

  このマニュアルでは、JavaServer Pages テクノロジを使用して、HTML ページに Java コードと JavaBeans コールを埋め込む方法について説明しています。標準的な JSP 機能と Oracle 固有の機能の両方が記載されています。Oracle9*i* リリース 2（9.2）の Apache JServ 環境に関する考慮事項と、サーブレット 2.2 環境の機能および JServ 用 Oracle JSP コンテナによる一部の機能のエミュレーションについて説明しています。

- 『Oracle9*i* Java Stored Procedures Developer's Guide』

  このマニュアルでは、Oracle9*i* データベースで直接実行するプログラムである Java ストアド・プロシージャについて説明しています。ストアド・プロシージャ（ファンクション、プロシージャ、トリガーおよび SQL メソッド）により、Java 開発者はビジネス・ロジックをサーバー・レベルで実装して、アプリケーション・パフォーマンス、拡張性およびセキュリティを改善できます。

- 『Oracle9*i*AS Containers for J2EE ユーザーズ・ガイド』

  このマニュアルでは、OC4J の概要と一般的な構成および配置手順について説明しています。特に、サーブレット、JSP ページおよび EJB については、個々の章に記載されています。

- 『Oracle9*i*AS Containers for J2EE JavaServer Pages サポート・リファレンス』

  このマニュアルは、OC4J でページを実行する必要のある JSP 開発者を対象としています。JSP 規格の概要とプログラミングの考慮事項、Oracle の付加価値機能と OC4J 環境の使用開始手順を説明しています。

- 『Oracle9*i*AS Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』

  このマニュアルでは、タグ・ライブラリ、JavaBeans および OC4J で提供される他の Java ユーティリティの概要と詳細な構文および使用方法を説明しています。

- 『Oracle9*i*AS Containers for J2EE Servlet 開発者ガイド』

  このマニュアルは、サーブレット開発者を対象としており、OC4J におけるサーブレットとサーブレット・コンテナの使用方法を説明しています。また、関連する OC4J 構成ファイルも記載されています。

- 『Oracle9*i*AS Containers for J2EE サービス・ガイド』

  このマニュアルでは、JTA、JNDI および Oracle9*i* Application Server Java Object Cache など、OC4J とともに提供される基本的な Java サービスについて説明しています。

- 『Oracle9*i*AS Containers for J2EE Enterprise JavaBeans 開発者ガイドおよびリファレンス』

  このマニュアルでは、OC4J における EJB 実装と EJB コンテナについて説明しています。

- 『Oracle9*i* XML データベース開発者ガイド - Oracle XML DB』
- 『Oracle9*i* XML Developer's Kit ガイド - XDK』
- 『Oracle9*i* アプリケーション開発者ガイド - 基礎編』
- 『Oracle9*i* アプリケーション開発者ガイド - ラージ・オブジェクト』
- 『Oracle9*i* アプリケーション開発者ガイド - オブジェクト・リレーショナル機能』
- 『Oracle9*i* Java パッケージ・プロシージャ・リファレンス』
- 『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』
- 『PL/SQL ユーザーズ・ガイドおよびリファレンス 』
- 『Oracle9*i* SQL リファレンス 』
- 『Oracle9*i* Net Services 管理者ガイド』
- 『Oracle Advanced Security 管理者ガイド』
- 『Oracle9*i* Database グローバリゼーション・サポート・ガイド』
- 『Oracle9*i* データベース・リファレンス 』
- 『Oracle9*i* データベース・エラー・メッセージ』

- 『Oracle9*i* サンプル・スキーマ』
- 『Oracle9*i* Application Server 管理者ガイド』
- 『Oracle Enterprise Manager 管理者ガイド』
- 『Oracle9*i* Application Server Oracle HTTP Server 管理ガイド』
- 『Oracle9*i* Application Server パフォーマンス・ガイド』
- 『Oracle9*i* Application Server グローバリゼーション・サポート・ガイド』
- 『Oracle9*i*AS Web Cache 管理および配置ガイド』
- 『Oracle9*i* Application Server Oracle Application Server からの移行』
- JDeveloper オンライン・ヘルプ
- OTN-J（Oracle Technology Network Japan）上の JDeveloper マニュアル

  `http://otn.oracle.co.jp`

リリース・ノート、インストレーション・マニュアル、ホワイト・ペーパー、またはその他の関連文書は、OTN-J（Oracle Technology Network Japan）に接続すれば、無償でダウンロードできます。OTN-J を使用するには、オンラインでの登録が必要です。次の URL で登録できます。

`http://otn.oracle.co.jp/membership/`

OTN-J のユーザー名とパスワードを取得済みであれば、次の OTN-J Web サイトの文書セクションに直接接続できます。

`http://otn.oracle.co.jp/document/`

表記規則

このマニュアル・セットの本文とコード例に使用されている表記規則について説明します。

本文の表記規則

本文中には、特別な用語が一目でわかるように様々な表記規則が使用されています。次の表は、本文の表記規則と使用例を示しています。

表記規則	意味	例
固定幅フォントの大文字	固定幅フォントの大文字は、システムにより指定される要素を示します。この要素には、パラメータ、権限、データ型、Recovery Manager キーワード、SQL キーワード、SQL*Plus またはユーティリティ・コマンド、パッケージとメソッドの他、システム指定の列名、データベース・オブジェクトと構造体、ユーザー名およびロールがあります。	この句は、NUMBER 列に対してのみ指定できます。 BACKUP コマンドを使用すると、データベースのバックアップを作成できます。 USER_TABLES データ・ディクショナリ・ビューの TABLE_NAME 列を問い合せます。 DBMS_STATS.GENERATE_STATS プロシージャを使用します。
固定幅フォントの小文字	固定幅フォントの小文字は、実行可能ファイル、ファイル名、ディレクトリ名およびサンプルのユーザー指定要素を示します。この要素には、コンピュータ名とデータベース名、ネット・サービス名、接続識別子の他、ユーザー指定のデータベース・オブジェクトと構造体、列名、パッケージとクラス、ユーザー名とロール、プログラム・ユニットおよびパラメータ値があります。 **注意 :** 一部のプログラム要素には、大文字と小文字の両方が使用されます。この場合は、記載されているとおりに入力してください。	sqlplus と入力して SQL*Plus をオープンします。 パスワードは orapwd ファイルに指定されています。 データ・ファイルと制御ファイルのバックアップを /disk1/oracle/dbs ディレクトリに作成します。 department_id、department_name および location_id の各列は、hr.departments 表にあります。 QUERY_REWRITE_ENABLED 初期化パラメータを true に設定します。 oe ユーザーで接続します。 これらのメソッドは JRepUtil クラスに実装されます。
固定幅フォントの小文字のイタリック	固定幅フォントの小文字のイタリックは、プレースホルダまたは変数を示します。	*parallel_clause* を指定できます。 *old_release*.SQL を実行します。*old_release* は、アップグレード前にインストールしたリリースです。

コード例の表記規則

コード例は、SQL、PL/SQL、SQL*Plus またはその他のコマンドラインを示します。次のように、固定幅フォントで、通常の本文とは区別して記載されています。

```
SELECT username FROM dba_users WHERE username = 'MIGRATE';
```

次の表は、コード例の記載上の表記規則と使用例を示しています。

表記規則	意味	例
[ ]	大カッコで囲まれている項目は、1 つ以上のオプション項目を示します。大カッコ自体は入力しないでください。	DECIMAL (*digits* [, *precision*])
|	縦線は、大カッコまたは中カッコ内の複数のオプションを区切るために使用します。オプションのうち 1 つを入力します。縦線自体は入力しないでください。	{ENABLE | DISABLE} [COMPRESS | NOCOMPRESS]
. . .	水平の省略記号は、次のどちらかを示します。	
	■ 例に直接関係のないコード部分が省略されていること。	CREATE TABLE ... AS *subquery*;
	■ コードの一部が繰返し可能であること。	SELECT *col1*, *col2*, ... , *coln* FROM employees;
その他の表記	大カッコ、中カッコ、縦線および省略記号以外の記号は、示されているとおりに入力してください。	acctbal NUMBER(11,2); acct CONSTANT NUMBER(4) := 3;
イタリック	イタリックの文字は、特定の値を指定する必要のあるプレースホルダまたは変数を示します。	CONNECT SYSTEM/*system_password* DB_NAME = *database_name*
大文字	大文字は、システムにより指定される要素を示します。これらの用語は、ユーザー定義用語と区別するために大文字で記載されています。大カッコで囲まれている場合を除き、記載されているとおりの順序とスペルで入力してください。ただし、この種の用語は大 / 小文字区別がないため、小文字でも入力できます。	SELECT last_name, employee_id FROM employees; SELECT * FROM USER_TABLES; DROP TABLE hr.employees;

表記規則	意味	例

小文字	小文字は、ユーザー指定のプログラム要素を示します。たとえば、表名、列名またはファイル名を示します。
	
	注意：一部のプログラム要素には、大文字と小文字の両方が使用されます。この場合は、記載されているとおりに入力してください。
		`SELECT last_name, employee_id FROM employees;`
		
		`sqlplus hr/hr`
		
		`CREATE USER mjones IDENTIFIED BY ty3MU9;`

1

JPublisher の概要

この章では、JPublisher ユーティリティの概要と例に続いて、より詳細な概要を説明します。この章の内容は、次のとおりです。

- JPublisher の機能の概要
- JPublisher について
- JPublisher の操作

JPublisher を初めて使用する場合は、1-2 ページの「JPublisher の概要」を参照してください。JPublisher の使用経験がある場合は、1-9 ページの「Oracle9i リリース 2（9.2）の JPublisher の新機能」に進んでかまいません。

JPublisher の機能の概要

この項では、Oracle9*i* リリース 2（9.2）の基本機能と新機能の概要について説明します。

JPublisher の概要

JPublisher はすべて Java で記述されたユーティリティで、次のユーザー定義データベース・エンティティを Java プログラムで表すために Java クラスを生成します。

- SQL オブジェクト型
- オブジェクト参照型（REF 型）
- SQL コレクション型（VARRAY 型またはネストした表型）
- PL/SQL パッケージ

JPublisher を使用すると、強い型指定の方法で SQL オブジェクト型、オブジェクト参照型およびコレクション型（VARRAY またはネストした表）の Java クラスへのマッピングを指定およびカスタマイズできます。

JPublisher では、オブジェクト型の属性ごとに `get`*XXX*`()` および `set`*XXX*`()` アクセッサ・メソッドが生成されます。オブジェクト型にストアド・プロシージャがある場合は、JPublisher でストアド・プロシージャをコールするラッパー・メソッドを生成できます。ラッパー・メソッドは、Oracle9*i* で実行されるストアド・プロシージャをコールするメソッドです。

また、JPublisher では PL/SQL パッケージ用のクラスも生成できます。これらのクラスには、PL/SQL パッケージのストアド・プロシージャをコールするためのラッパー・メソッドが含まれます。

JPublisher で生成されるラッパー・メソッドには SQLJ コードが含まれているため、ラッパー・メソッドの生成時には、通常 `.sqlj` ソース・ファイルが生成されます。これは、JPublisher に対してラッパー・メソッドを生成しないように（`-methods` オプションを通じて）指定しないかぎり、メソッドを定義する PL/SQL パッケージやオブジェクト型を表すクラスの場合も同じです。

ラッパー・メソッドが生成されない場合、JPublisher は `.java` ソース・ファイルを生成します。これは、メソッドを持たないオブジェクト型、オブジェクト参照型またはコレクション型を表すクラスや、`-methods` オプションがオフになっているクラスの場合も同じです。

JPublisher で生成されたクラスを直接使用しない場合、次の方法を実行できます。

- 生成されたクラスを拡張します。この作業は簡単です。JPublisher では、サブクラスの初期バージョンも自動的に生成されるため、そこに必要な動作を追加できます。
- JPublisher を使用せずに、ユーザーが Java クラスを記述します。このアプローチは非常に柔軟ですが、時間がかかりエラーも起こりやすくなります。

- 汎用クラスを使用して、オブジェクト型、オブジェクト参照型およびコレクション型を表します。oracle.sql パッケージには、オブジェクト型、オブジェクト参照型およびコレクション型を表す、弱い型指定を持つ汎用クラスが含まれます。これらのクラスがユーザーの要件を満たす場合は、JPublisher を使用する必要はありません。通常、このアプローチを使用するのは、いずれかの SQL オブジェクト型、コレクション型、参照型または OPAQUE 型を汎用的に処理できるようにする必要がある場合です。

また、JPublisher では、Java から PL/SQL 型へのアクセスのみが簡素化されます。PL/SQL 型と SQL 型の間で事前定義済みまたはユーザー定義のマッピングを利用できるのみでなく、この 2 種類の型の間で PL/SQL 変換ファンクションを使用できます。この型の対応関係を使用することで、JPublisher は必要な Java コードと PL/SQL コードをすべて自動的に生成できます。

JPublisher スタート・ガイド

JPublisher は、Oracle SQLJ Translator とともに配布されます。Oracle Installer を使用して SQLJ をインストールした場合、すでにセットアップは完了しています。ただし、Oracle SQLJ のバージョンを手動でダウンロードした場合は、SQLJ と JPublisher を使用するための手順を手動で実行する必要があります。指示については、『Oracle9*i* SQLJ 開発者ガイドおよびリファレンス』を参照してください。

次のことを確認する必要があります。

- Sun 社の JDK のバージョンがインストール済みで、javac コンパイラをコマンドラインから起動できること。
- Oracle JDBC Drivers がインストール済みで、CLASSPATH に指定されていること、通常は [Oracle_Home]/jdbc/classes*XX*.jar です。
- Oracle SQLJ Translator とランタイムが CLASSPATH に指定されていること、通常は、[Oracle_Home]/sqlj/translator.jar および [Oracle_Home]/sqlj/runtime*XX*.jar です。
- 起動スクリプトまたは実行可能ファイル（jpub または jpub.exe、sqlj または sqlj.exe）がファイル・パスに指定されていること。通常は、[Oracle_Home]/bin または [Oracle_Home]/sqlj/bin（手動ダウンロードの場合）です。

  適切にセットアップされていれば、コマンドラインに jpub と入力すると、共通の JPublisher オプションおよび入力設定の情報が表示されます。

また、リリース 2（9.2）以上の Oracle データベースに対してリリース 9.2.0 以上の JPublisher を使用する場合は、PL/SQL パッケージ SYS.SQLJUTL をインストールする必要があります。データベースが Java 対応の場合は、このパッケージがすでにインストールされています。インストールされていない場合は、SYS スキーマに SQL スクリプト [Oracle_Home]/sqlj/lib/sqljutl.sql をインストールするように、データベース管理者に依頼してください。

注意： これ以降は、全般的な説明と例が記載されています。詳細な例は、Oracle インストールの [Oracle_Home]/sqlj/demo/jpub にアクセスしてください。

SQL オブジェクト型の公開

JPublisher を使用すると、SQL オブジェクトとパッケージを Java クラスとして簡単に公開できます。ここでは、Oracle9*i* サンプル・スキーマの一部である OE（Order Entry）スキーマの例をあげて説明します（詳細は、『Oracle9*i* サンプル・スキーマ』を参照）。サンプル・スキーマをインストールしていなくても、公開対象となる独自のオブジェクト型またはパッケージがある場合は、ユーザー名、パスワードおよびオブジェクト名またはパッケージ名を独自の名前に置き換えてください。

OE スキーマのパスワードが OE の場合、SQL オブジェクト型 CATEGORY_TYP を公開するには次のようにします。

```
jpub -user=OE/OE -sql=CATEGORY_TYP:CategoryTyp
```

JPublisher の -user オプションを使用して、ユーザー名（スキーマ名）とパスワードを指定します。-sql オプションでは、公開する型とパッケージを指定します。CATEGORY_TYP は SQL 型の名前、CategoryTyp は生成対象となる対応する Java クラスの名前で、両者をコロン（:）で区切って指定します。JPublisher では、公開対象となる SQL 型およびパッケージの名前が標準出力にエコーされます。

```
OE.CATEGORY_TYP
```

カレント・ディレクトリ内のファイルをリストすると、予期したファイル CategoryTyp.java に加えて、ファイル CategoryTypeRef.java も生成されたことがわかります。これは、OE.CATEGORY_TYP への SQL オブジェクト参照に対する強い型指定を持つラッパーを表します。どちらのファイルも、Java コンパイラ javac によるコンパイルの準備が完了しています。

別の例として、CUSTOMER_TYP 型に、-user= の短縮形 -u（続いて空白）と -sql= の短縮形 -s を使用する場合は、次のようにします。

```
jpub -u OE/OE -s CUSTOMER_TYP:CustomerTyp
```

JPublisher では次のように出力されます。

```
OE.CUSTOMER_TYP
OE.CUST_ADDRESS_TYP
OE.PHONE_LIST_TYP
OE.ORDER_LIST_TYP
OE.ORDER_TYP
OE.ORDER_ITEM_LIST_TYP
OE.ORDER_ITEM_TYP
OE.PRODUCT_INFORMATION_TYP
```

```
OE.INVENTORY_LIST_TYP
OE.INVENTORY_TYP
OE.WAREHOUSE_TYP
```

JPublisher では、SQL オブジェクト型のリストがレポートされます。これは、新しいオブジェクト型を検出するたびに（それが属性、オブジェクト参照、またはそれ自体がオブジェクトかコレクションの要素型を持っているコレクションであるかを問わず）、その型のラッパー・クラスも自動的に生成されるためです。この例では、オブジェクト型ごとに、1）オブジェクト型のインスタンスを表す CustomerTyp などの Java クラス、2）オブジェクト型の参照を表す CustomerTypeRef などの参照クラスという、2 つのラッパー・ファイルが生成されます。また、JPublisher によってデフォルトで使用されるネーミング計画もわかります。たとえば、SQL 型 OE.PRODUCT_INFORMATION_TYP は、Java クラス ProductInformationTyp となります。

JPublisher では埋込み型のラッパーは自動的に生成されますが、特定のオブジェクト型のサブタイプが自動的に生成されることはありません。この場合は、公開するサブタイプすべてを明示的に列挙する必要があります。CATEGORY_TYP 型には、LEAF_CATEGORY_TYP、COMPOSITE_CATEGORY_TYP および CATALOG_TYP の 3 つのサブタイプがあります。これらのオブジェクト型を公開するための JPublisher コマンドラインは、次のようになります。

```
jpub  -u OE/OE  -s COMPOSITE_CATEGORY_TYP:CompositeCategoryTyp
      -s LEAF_CATEGORY_TYP:LeafCategoryTyp,CATALOG_TYP:CatalogTyp
```

JPublisher では次のように出力されます。

```
OE.COMPOSITE_CATEGORY_TYP
OE.SUBCATEGORY_REF_LIST_TYP
OE.LEAF_CATEGORY_TYP
OE.CATALOG_TYP
OE.CATEGORY_TYP
OE.PRODUCT_REF_LIST_TYP
```

次の点に注意してください。

- 複数の型を解析解除する場合は、そのすべてをカンマで区切って -sql（-s）オプションに指定する方法と、コマンドラインで複数の -sql オプションを指定する方法があります。また、両方使用してもかまいません。
- JPublisher では、すべてのサブクラスのラッパーが自動的に生成されることはありませんが、すべてのスーパークラスのラッパーは生成されます。
- 以前に JPublisher を実行して CatalogTyp を生成している場合は、.java ファイルが出力されています。ただし、このときに生成されたのは、CATALOG_TYP とその 3 つのサブタイプ用の .sqlj ファイルです。

  これは、SQLJ では Java から SQL を起動するためのコーディングが簡素化されているためです。SQL オブジェクト型にメソッドが含まれている場合、JPublisher によってデフォルトで生成される .sqlj ファイルには、これらのメソッドのラッパーも含まれま

す。Oracle SQLJ Translator で次のように入力すると、.sqlj ファイルと .java ファイルの両方をすぐに変換してコンパイルできます。

```
sqlj *.sqlj *.java
```

SQL 型階層の Java ラッパーを生成する場合に、1 つ以上の型にメソッドが（この例のように）含まれていると、階層内のすべての型の .sqlj ファイルが自動的に生成されます。JPublisher オプション -methods=false を使用すると、メソッド・ラッパーの生成を必要に応じて抑制できることに注意してください。これにより、.sqlj ファイルの生成も抑制されます。

JPublisher によって生成されたコードで必要な機能や動作が得られない場合は、生成されたラッパー・クラスをサブクラス化することで、その機能をオーバーライドしたり補完することができます。次の例を考えます。

```
jpub -u OE/OE -s WAREHOUSE_TYP:JPubWarehouse:MyWarehouse
```

JPublisher では次のように出力されます。

```
OE.WAREHOUSE_TYP
```

このコマンドを実行すると、JPublisher では JPubWarehouse.java と MyWarehouse.java が生成されます。ファイル JPubWarehouse.java は、このコマンドを実行するたびに生成されます。ファイル MyWarehouse.java は、カスタマイズできるように作成され、後でこの JPublisher コマンドを実行しても上書きされません。また、MyWarehouse.java に新規メソッドを追加することも、JPubWarehouse.java からのメソッド実装をオーバーライドすることも、あるいはその両方を行うこともできます。Java で WAREHOUSE_TYP インスタンスの実体化に使用されるクラスは、特殊クラス MyWarehouse です。オブジェクト型階層のすべての型にユーザー固有のサブクラスが必要な場合は、すべての階層メンバーについて、前述したように *SQL_TYPE*:*JPubClass*:*UserClass* 形式の 3 つ 1 組を指定する必要があります。

Java ラッパー・クラスをいくつか生成してコンパイルした後に、それを実際に Java プログラムで使用するとします。

JPublisher で生成してコンパイルした Java ラッパー・クラスを使用するのは、SQLJ でプログラミングしている場合は特に簡単で、オブジェクト・ラッパーを直接使用できます。次の例では、IN OUT パラメータとして WAREHOUSE_TYPE インスタンスを取る PL/SQL ストアド・プロシージャをコールします。

```
java.math.BigDecimal location = ...;
java.math.BigDecimal warehouseId = ...;
MyWarehouse w = new MyWarehouse(warehouseId,"Industrial Park",location);
...
#sql { call register_warehouse(:INOUT w) };
```

JDBC では、通常、SQL 型名と対応する Java クラスとの関係を接続インスタンス用の型マップに登録します。この登録は、次の例に示すように接続ごとに必要です。

```
java.util.Map typeMap = conn.getTypeMap();
typeMap.put("OE.WAREHOUSE_TYP", MyWarehouse.class);
conn.setTypeMap(typeMap);
```

次の JDBC コードは、前述の #sql 文に対応しています。

```
CallableStatement cs = conn.prepareCall("{call register_warehouse(?)}");
((OracleCallableStatement)cs).registerOutParameter
                (1,oracle.jdbc.OracleTypes.STRUCT,"OE.WAREHOUSE_TYP");
cs.setObject(w);
cs.executeUpdate();
w = cs.getObject(1);
```

PL/SQL パッケージの公開

前項に示したように、SQLJ コードでは PL/SQL ストアド・プロシージャまたはファンクションを簡単にコールできます。ただし、PL/SQL パッケージ全体を Java クラスにカプセル化する必要がある場合に備えて、JPublisher にはこの機能も用意されています。

PL/SQL ファンクションおよびプロシージャを Java メソッドとして表すという概念には問題があります。この種のファンクションやプロシージャの引数には PL/SQL モード OUT または IN OUT が使用されている可能性がありますが、Java には引数を渡すための等価モードが存在しません。たとえば、int 引数を取るメソッドでは、コール元が新規の値を受け取れるようにこの引数を変更することができません。回避策として、JPublisher では OUT 引数と IN OUT 引数について単一要素の配列が生成されます。たとえば、配列 int[] abc の場合、入力値は abc[0] に指定され、変更後の出力値も abc[0] で戻されます。SQL オブジェクト型のメソッドのコード生成時にも、同様のパターンが使用されます。

次のコマンドラインでは、SYS.DBMS_LOB パッケージが Java に公開されます。

```
jpub  -u SCOTT/TIGER  -s SYS.DBMS_LOB:DbmsLob
```

JPublisher では次のように出力されます。

```
SYS.DBMS_LOB
```

DBMS_LOB はパブリックに参照できるため、SCOTT などの異なるスキーマからアクセスできます。この JPublisher 起動コードでは、PL/SQL パッケージのコールを含む SQLJ ソース・ファイル DbmsLob.sqlj が作成されることに注意してください。生成される Java メソッドは、実際はすべてのインスタンスのメソッドです。これは、JDBC 接続または SQLJ 接続コンテキストを使用してパッケージのインスタンスを作成し、そのインスタンス上でメソッドをコールするという考え方です。

Java 基本数値型にかわるオブジェクト型の使用 生成されたコードを調べると、JPublisher によって各種メソッドの引数として java.lang.Integer が生成されたことがわかります。int などの Java 基本型のかわりに Integer などの Java オブジェクト型を使用すると、SQL の NULL 値を Java の null として直接表すことができ、JPublisher ではこの値がデフォ

ルトで生成されます。ただし、DBMS_LOB パッケージの場合、実際にはオブジェクト型 Integer ではなく int が使用されています。-numbertypes オプションを使用すると、JPublisher 起動コードを次のように変更できます。

```
jpub -numbertypes=jdbc  -u SCOTT/TIGER  -s SYS.DBMS_LOB:DbmsLob
```

JPublisher では次のように出力されます。

```
SYS.DBMS_LOB
```

SQL のトップ・レベルにあるプロシージャのラッパー・コード JPublisher では、SQL のトップ・レベルにあるファンクションとプロシージャのラッパー・コードも生成できます。次の例のように、特殊パッケージ名 TOPLEVEL を使用します。

```
jpub  -u SCOTT/TIGER  -s TOPLEVEL:SQLTopLevel
```

JPublisher では次のように出力されます。

```
SCOTT.<top-level_scope>
```

SQL のトップ・レベルの有効範囲にストアド・ファンクション、またはストアド・プロシージャがないと警告が表示されます。

ストアド・プロシージャまたはファンクションで使用されている型が PL/SQL に固有で、Java でサポートされていない場合は、警告メッセージが表示され、対応する Java メソッドは生成されません。ただし、PL/SQL 型を対応する SQL 型および対応する Java 型にマップできる場合があります。その場合は、JPublisher で適切な Java コード、および可能であれば PL/SQL コードを生成し、Java からこれらの型へのアクセスを取得できます。（2-7 ページの「JDBC でサポートされないデータ型の使用」を参照。）

Oracle9*i* リリース 2（9.2）の JPublisher の新機能

Oracle9*i* リリース 2（9.2）の JPublisher は、実際には Oracle JDBC Drivers で使用できる型をすべてサポートしています。また、PL/SQL 変換のサポートを介して、ストアド・プロシージャやオブジェクト・メソッドのシグネチャ内で PL/SQL 型を使用できます。次の Oracle JDBC の型が直接サポートされるようになりました。

- NCHAR 型
- TIMESTAMP 型
- SQLJ オブジェクト型
- SQL OPAQUE 型

  特に、OPAQUE 型 `SYS.XMLTYPE` は Java の `oracle.xdb.XMLType` 型を介してサポートされます。SQL の OPAQUE 型は、事前定義済みの型の対応関係を介してサポートするか、JPublisher コード生成をトリガーできます。（2-7 ページの「OPAQUE 型の型マッピング・サポート」を参照。）

PL/SQL ラッパー・ファンクションおよびプロシージャの自動生成と次のメカニズムを介して、JPublisher コードからネイティブの PL/SQL 型に従来より簡単にアクセスできるようになりました。

- PL/SQL の `BOOLEAN` と Java の `boolean` の間、または PL/SQL の `INTERVAL` と Java の `String` の間など、事前定義済みの型変換

  2-10 ページの「PL/SQL 変換ファンクションを介した型マッピングのサポート」を参照してください。

- PL/SQL の索引付き表と JDBC OCI ドライバを併用するためのユーザー定義マッピング

  2-8 ページの「JDBC OCI を使用したスカラーの索引付き表の型マッピング・サポート」を参照してください。

- PL/SQL の RECORD 型とレコードの表を SQL のオブジェクト型とコレクション型にマップし、さらに最終的には Java にマップするための、ユーザー定義変換ファンクション

  2-13 ページの「PL/SQL RECORD 型の型マッピング・サポート」を参照してください。

JPublisher の機能と生成コードの柔軟性は、次のように改善されています。

- SQL オブジェクト型について属性ベースのコンストラクタが生成されます。
- 生成されるクラスに新規 API が用意されており、強い型指定を持つ参照の間で変換し、オブジェクト間で接続情報を転送できます。
- SQL オブジェクト型に対して生成される Java ラッパーをシリアル化可能にすることができます。

  3-24 ページの「生成されたオブジェクト・ラッパーのシリアライズ可能性（-serializable）」を参照してください。

- オブジェクト値をレポートする `toString()` メソッドを作成できます。

  3-27 ページの「オブジェクト・ラッパーの toString() メソッドの生成（-tostring）」を参照してください。

JPublisher では、次のようにプログラミングの労力がさらに軽減されます。

- JPublisher 生成のクラスをユーザー・サブクラス化するように要求すると、これらのユーザー・サブクラスの初期バージョンは JPublisher によって自動的に生成されます。
- 継承階層をユーザー・アプリケーションで初期化する必要はありません。
- JPublisher と `Make` 環境との相互作用が改善されており、生成されたファイルが不必要に上書きされることがありません。
- JPublisher プロパティ・ファイルの構文が拡張されたことにより、JPublisher ディレクティブを SQL スクリプトに埋め込むことができます。

JPublisher について

この項では、JPublisher の概要と実行できる操作について説明します。この項の内容は、次のとおりです。

- JPublisher のオブジェクト型マッピングと PL/SQL マッピング
- JPublisher による処理
- JPublisher で生成される内容
- JPublisher の要件
- JPublisher の入力および出力
- データ型マッピングの概要
- データベースでの型およびパッケージの作成

JPublisher のオブジェクト型マッピングと PL/SQL マッピング

JPublisher には、次の SQL エンティティから Java クラスへのマッピングが用意されています。

- SQL のオブジェクト型、コレクション型、参照型および OPAQUE 型
- PL/SQL のパッケージおよび型

オブジェクト型と JPublisher

JPublisher を使用すると、Java 言語アプリケーションで Oracle9*i* のユーザー定義オブジェクト型を使用できます。Java 言語アプリケーションでオブジェクト・データにアクセスする場合は、そのデータを Java 形式で表す必要があります。JPublisher により、オブジェクト型と Java クラス、およびオブジェクト属性の型とそれに対応する Java 型の間にマッピングを作成してデータを Java 形式で表現できます。

JPublisher により生成されるクラスでは、JPublisher オプションの設定方法に応じて、`oracle.sql.ORAData` インタフェースまたは `java.sql.SQLData` インタフェースが実装されます。どちらのインタフェースの場合も、データベースと Java プログラム間でオブジェクト型のインスタンスが転送されます。`ORAData` および `SQLData` インタフェースの詳細は、『Oracle9*i* JDBC 開発者ガイドおよびリファレンス』を参照してください。

PL/SQL パッケージと JPublisher

PL/SQL パッケージ内のストアド・プロシージャを Java アプリケーションからコールする場合があります。ストアド・プロシージャは、PL/SQL サブプログラムまたは SQL に対して公開されている Java メソッドに変換できます。Java 引数およびファンクションはストアド・プロシージャに渡され、そのストアド・プロシージャから戻されます。

これを実行するために、パッケージ内のサブプログラムごとにラッパー・メソッドを含むクラスを作成するように、JPublisher に対して指示できます。JPublisher で生成されたラッパー・メソッドにより、Java コードからの PL/SQL ストアド・プロシージャのコールまたはクライアントの Java プログラムからの Java ストアド・プロシージャのコールが簡単に実行できます。

トップレベルのサブプログラム（どの PL/SQL パッケージにもないサブプログラム）を含む PL/SQL コードをコールすると、JPublisher では、トップレベルのすべてのプロシージャとファンクション、または要求したトップレベルのサブプログラムのサブセット用のラッパー・メソッドを含む単一クラスが生成されます。

PL/SQL 型と JPublisher

Java プログラムで SQL 型を使用できるのは、PL/SQL ストアド・プロシージャまたはファンクションをコールする場合のみです。BOOLEAN、PL/SQL RECORD 型および PL/SQL 索引付き表など、PL/SQL でのみサポートされている型には、JDBC プログラムではアクセスできません。ただし、スカラー型の PL/SQL 索引付き表は、現在、クライアント側 JDBC OCI ドライバでのみサポートされています。

JPublisher では、次のような型を含むストアド・プロシージャとファンクションの起動が簡素化されます。つまり、必要に応じて、PL/SQL 型を含むシグネチャと、Java プログラムから使用可能で SQL 型のみを参照する対応するシグネチャの間で変換を行うために、PL/SQL ラッパー・プロシージャおよびファンクションを含むパッケージが自動的に作成されます。BOOLEAN 型の場合は、マッピングが事前に定義されています。ただし、通常、JPublisher でコード生成に特定の PL/SQL 型を取り込むには、SQL と PL/SQL との対応付けと変換を提供する必要があります。

JPublisher による処理

JPublisher はデータベースに接続し、コマンドラインまたは入力ファイルから指定した SQL オブジェクト型または PL/SQL パッケージの記述を取り出します。JPublisher はデフォルトで JDBC OCI ドライバを使用してデータベースに接続します。この場合、Oracle9*i* Net と必要なサポート・ファイルを含む Oracle クライアントをインストールしたマシンが必要です。Oracle クライアント・マシンがない場合、JPublisher では Oracle JDBC Thin ドライバを使用できます。

JPublisher では、変換対象の SQL オブジェクト型ごとに Java クラスが生成されます。Java クラスには、オブジェクトをデータベースに対して読取り / 書込みを行うために必要なコードが含まれます。JPublisher で生成したクラスを配置する際には、JDBC ドライバも含めて、必要なランタイム・ファイルをマシンにインストールする必要があります。ラッパー・メソッド（SQL オブジェクト型のストアド・プロシージャまたはファンクションをラップする Java メソッド）を作成する場合、JPublisher では SQLJ ソース・コードが生成されるため、SQLJ ランタイム・ライブラリも必要になります。

ラッパー・メソッドをコールすると、オブジェクトの SQL 値が IN または IN OUT 引数とともにサーバーに送られます。次に、メソッド（ストアド・プロシージャまたはファンクション）がコールされ、新規オブジェクト値が OUT または IN OUT 引数とともにクライアントに戻されます。この結果、データベースとのラウンドトリップが発生するため注意してください。メソッドのコールでオブジェクトに対する単一の状態変更のみが実行される場合は、状態変更にローカルに影響する等価の Java を記述して使用する方がはるかに効率的です。

JPublisher では、変換対象の PL/SQL パッケージごとに 1 つのクラスが生成されます。そのクラスには、サーバーでパッケージ・メソッドをコールするためのコードが含まれます。メソッドの IN 引数は、クライアントからサーバーへ送られ、OUT 引数および結果はサーバーからクライアントへ戻されます。また JPublisher では、必要に応じて PL/SQL 型を含むシグネチャを SQL 型しか含まない、対応するシグネチャに変換するために、PL/SQL ラッパー・パッケージも生成できます。

次の項では、JPublisher でオブジェクト型と PL/SQL パッケージに対して作成されるソース・ファイルの概要を説明します。

JPublisher で生成される内容

JPublisher で生成されるファイル数は、ORAData クラス（oracle.sql.ORAData インタフェースを実装するクラス）または SQLData クラス（標準の java.sql.SQLData インタフェースを実装するクラス）のどちらを要求するかによって異なります。

ORAData インタフェースでは、SQL オブジェクト型、オブジェクト参照型、コレクション型および OPAQUE 型が強い型指定の方法でサポートされます。つまり、データベース内の特定のオブジェクト型、オブジェクト参照型、コレクション型または OPAQUE 型ごとに、対応する Java の型があります。一方、SQLData インタフェースでは、SQL オブジェクト型のみが強い型指定の方法でサポートされます。すべてのオブジェクト参照型は java.sql.Ref インスタンスとして汎用的に表され、すべてのコレクション型は java.sql.Array インスタンスとして汎用的に表されます。したがって、JPublisher では、ORAData クラスの生成時にのみ、オブジェクト参照型、コレクション型および OPAQUE 型のクラスが生成されます。

ユーザー定義のオブジェクト型に対して JPublisher を実行して ORAData クラスを要求すると、自動的に次のクラスが生成されます。

- Java プログラムで Oracle オブジェクト型のインスタンスを表すオブジェクト・クラス。
- Oracle オブジェクト型用のオブジェクト参照用の参照クラス。
- トップレベル・オブジェクト内で直接または間接的にネストされたオブジェクト、コレクションまたは OPAQUE の属性用の Java クラス。

  この自動作成が必要なのは、トップレベル・クラスのインスタンスが実体化されるたびに、Java で属性を実体化できるようにするためです。SQL OPAQUE 型や PL/SQL 型など、属性の型が事前にマップされている場合、JPublisher ではマップからのターゲット Java 型が使用されます。

注意： ORAData 実装の場合は、SQL オブジェクト型で参照が使用されるかどうかに関係なく、強い型指定を持つ参照クラスが常に生成されます。

弱い型指定の参照のかわりに強い型指定の参照を使用するメリットについては、1-23 ページの「ORAData 実装の強い型指定を持つオブジェクト参照」を参照してください。

一方、SQLData クラスを要求すると、JPublisher ではオブジェクト参照クラス、ネストされたコレクション属性用のクラスまたは OPAQUE 属性用のクラスのどちらも生成されません。

ユーザー定義コレクション型に対して JPublisher を実行している場合は、ORAData クラスを要求する必要があります。JPublisher では自動的に次のクラスが生成されます。

- Oracle コレクション型に対応する型定義として動作するコレクション・クラス。
- コレクションの要素がオブジェクトの場合は、要素型用の Java クラスと、要素型の中で直接または間接的にネストされたオブジェクト属性またはコレクション属性用の Java クラス。

  この自動生成が必要なのは、コレクションのインスタンスが実体化されるたびに、Java でオブジェクト要素を実体化できるようにするためです。

OPAQUE 型に対して JPublisher を実行している場合は、ORAData クラスを要求する必要があります。JPublisher では、次のクラスが自動的に作成されます。

- OPAQUE 型のラッパーとして動作する Java クラス（OPAQUE 型のメソッドの Java バージョンを提供）と、サブクラス内で OPAQUE 型の表現にアクセスするための保護付き API。

  ただし、通常、SQL OPAQUE 型用の Java ラッパー・クラスは、たとえば、SQL OPAQUE 型 SYS.XMLTYPE の場合は oracle.xdb.XMLType など、OPAQUE 型のプロバイダから提供されます。この場合、SQL の型と Java の型との対応付けは、JPublisher に対して事前定義されていることが保証されます。

PL/SQL パッケージに対して JPublisher が実行されている場合は、次のクラスが自動的に生成されます。

- パッケージのストアド・プロシージャをコールするラッパー・メソッド付きの Java クラス。
- 必要な場合は、PL/SQL のシグネチャから SQL 型のみを含むシグネチャへの変換に必要なファンクションと、プロシージャを含む PL/SQL パッケージ定義。

  この PL/SQL パッケージ定義は、オブジェクト型のメソッドを変換するときに、PL/SQL と SQL の引数間での変換に PL/SQL ラッパーが必要な場合にも生成されます。

JPublisher の要件

JPublisher を使用するには、システムに Oracle SQLJ と Oracle JDBC もインストールし、CLASSPATH に適切に追加する必要があります。また、次のライブラリが必要です（.jar または .zip ファイルとして用意されています）。

- SQLJ Translator クラス（translator）
- SQLJ ランタイム・クラス（runtime12、runtime12ee または runtime11）
- JDBC クラス（classes12、ojdbc14 または classes111）

"12" は JDK 1.2.x 以上のリリース、"14" は JDK 1.4.x 以上のリリースを指します。"11" および "111" は、JDK 1.1.x のリリースを指します。これらのファイルの詳細は、『Oracle9*i* SQLJ 開発者ガイドおよびリファレンス』を参照してください。

Oracle9*i* リリース 2（9.2）以上のデータベースを使用している場合は、パッケージ SQLJUTL もインストールし、SYS スキーマ内でパブリックにアクセスできるようにする必要があります。この作業を行わないと、JPublisher の起動時に次の警告メッセージが表示されます。

```
Warning: Cannot determine what kind of type is <schema>.<type.> You
likely need to install SYS.SQLJUTL. The database returns: ORA-06550:
line 1, column 7:

PLS-00201: identifier 'SYS.SQLJUTL' must be declared
```

この場合は、SQL ファイル [Oracle Home]/sqlj/lib/sqljutl.sql を SYS スキーマにインストールしてパブリックにアクセス可能にするように、データベース管理者に依頼してください。これにより、前述の警告メッセージは表示されなくなります。

JPublisher と SQLJ は常に一緒にインストールされるため、Oracle9*i* JPublisher を使用する場合、通常は SQLJ も同等のバージョンを使用します。JPublisher のすべての機能を使用するには、次の機能も必要です。

- Oracle9*i*（またはリリース 8.1.7 か 8.1.6）
- Oracle9*i* JDBC ドライバ（またはリリース 8.1.7 か 8.1.6）
- Java Development Kit（JDK）バージョン 1.2 以上

JPublisher の一部の機能のみを使用する場合は、要件が緩和されます。

- SQLData クラスを生成せず、java.sql.Blob クラスおよび java.sql.Clob クラスも使用しない場合は、JDK 1.2.x ではなく JDK リリース 1.1.x を使用できます。
- Oracle 固有の ORAData インタフェース（または使用不可の CustomDatum インタフェース）を実装するクラスを生成しない場合は、Oracle 以外の JDBC ドライバまたは Oracle 以外の SQLJ 実装を使用できます。JPublisher で生成されたコードを実行している場合は、Oracle 以外のデータベースとも接続できますが、JPublisher 自体は Oracle データベースと接続する必要があります。Oracle では、Oracle 以外のコンポーネントを使用する構成のテストもサポートもしません。

- JPublisher に対して（-methods=false を設定して）ラッパー・メソッドを生成しないように指示した場合や、オブジェクト型でメソッドが定義されていない場合、JPublisher ではラッパー・メソッドは生成されず、.sqlj ファイルも生成されません。この場合、SQLJ Translator は不要です。-methods オプションの詳細は、3-20 ページの「パッケージのクラスおよびラッパー・メソッドの生成（-methods）」を参照してください。
- JPublisher を使用して SQL OPAQUE 型のラッパーを生成する場合は、Oracle 9*i* リリース 2（9.2）以上のデータベースと JDBC ドライバを使用する必要があります。
- JPublisher を使用して、使用不可の CustomDatum インタフェースを実装するカスタム・オブジェクト・クラスのみを生成する場合は、Oracle データベース リリース 8.1.5 を JDBC リリース 8.1.5 および JDK リリース 1.1.x 以上とともに使用できます。（ただし、ORAData インタフェースへのアップグレードをお薦めします。このインタフェースには、Oracle9*i* 以上の JDBC 実装が必要です。）

JPublisher の入力および出力

コマンドラインおよびプロパティ・ファイルで入力オプションを指定できます。変換されたオブジェクト用に .sqlj および .java ファイルが生成される他に、JPublisher では、変換されたオブジェクトとパッケージの名前を標準出力に書き出します。

JPublisher 入力

すべての JPublisher オプションは、3-2 ページの「JPublisher オプション」を参照してください。

さらに、INPUT ファイルと呼ばれるファイルを使用して、JPublisher で変換するオブジェクト型および PL/SQL パッケージを指定できます。また、JPublisher では生成されたパッケージおよびクラスのネーミングも制御します。INPUT ファイル構文については、3-32 ページの「INPUT ファイルの構造および構文」を参照してください。

プロパティ・ファイルは、JPublisher にオプションを指定できる任意のテキスト・ファイルです。プロパティ・ファイル名は、-props オプションを使用してコマンドラインで指定します。JPublisher では、プロパティ・ファイルの内容がその時点でコマンドラインから順序どおり挿入されたかのように処理されます。柔軟性を高めるために、プロパティ・ファイルを SQL スクリプト・ファイルにして、JPublisher ディレクティブを SQL コメントに埋め込むこともできます。プロパティ・ファイルとファイル形式の詳細は、3-30 ページの「プロパティ・ファイルの構造および構文」を参照してください。

JPublisher 出力

JPublisher では、変換対象のオブジェクト型ごとに Java クラスが生成されます。ORAData 実装であるか SQLData 実装であるかに関係なく、オブジェクト型ごとに、クラス・コード用には <type>.sqlj ファイルが（または、ラッパー・メソッドが抑制されているか存在しない場合、あるいは Publisher の -methods オプションの設定に依存する場合は <type>.java ファイルが）、Java 型の REF クラス・コード用には <type>Ref.java ファイルが生成されます。たとえば、SQL オブジェクト型 `EMPLOYEE` を定義する場合は、JPublisher で `employee.sqlj` ファイル（または `employee.java` ファイル）および `employeeRef.java` ファイルが生成されます。JPublisher で生成される Java クラス名の大 / 小文字区別は、`-case` オプションによって決定されます。3-14 ページの「Java 識別名の大文字小文字の区別（-case）」を参照してください。

JPublisher では、変換されるコレクション型（ネストした表または VARRAY）ごとに、`<type>.java` ファイルが生成されます。ネストした表の場合、生成されたクラスは、配列全体としてネストした表を取得および設定するメソッドと、表の個々の要素を取得および設定するメソッドを持ちます。JPublisher では、コレクション型は `ORAData` クラスの生成時に変換されますが、`SQLData` クラスの生成時には変換されません。JPublisher では、OPAQUE 型用のラッパー・クラスも生成できます。ただし、OPAQUE 型は、`ORAData` インタフェースを実装する、対応する Java クラスに事前にマップされている方が一般的です。

PL/SQL パッケージの場合は、JPublisher では `.sqlj` ファイルとしてラッパー・メソッドを含むクラスが生成されます。

JPublisher でクラス・ファイルおよびラッパーが生成されると、変換された型およびパッケージの名前も標準出力に書き出されます。

データ型マッピングの概要

JPublisher には、SQL から Java へのデータ型マッピングについて、様々なカテゴリが用意されています。これらのマッピングを指定するための JPublisher オプションは、3-7 ページの「データ型マッピングに影響するオプションの詳細説明」を参照してください。

型マッピング・オプションごとに、少なくとも 2 つの可能な値 `jdbc` および `oracle` があります。`-numbertypes` オプションには、さらに 2 つの代替値 `objectjdbc` および `bigdecimal` があります。

これらのマッピングのカテゴリについては、後述します。データ型マッピングの詳細は、2-2 ページの「データ型マッピングの詳細」を参照してください。

JDBC マッピング

JDBC マッピングでは、ほとんどの数値型は int および float などの Java 基本型にマップされ、DECIMAL および NUMBER は java.math.BigDecimal にマップされます。LOB 型と他の数値以外のスカラー型は、java.sql.Blob や java.sql.Timestamp などの標準 JDBC Java 型にマップされます。オブジェクト型の場合、JPublisher では SQLData クラスが生成されます。Oracle 拡張要素である事前定義済みデータ型（BFILE および ROWID など）には JDBC マッピングがないため、oracle.sql.* マッピングのみがサポートされます。

JDBC マッピングで使用される Java 基本型では null 値はサポートされず、整数オーバーフローや浮動小数点での精度損失に対する保護も行われません。JDBC マッピングを使用して、データベース値が null の基本型の属性（short、int、float または double）を取得するためにアクセッサまたはメソッドをコールしようとすると、例外が発生します。基本型が short または int の場合で、値が大きすぎて short または int 変数に入らない場合は、例外が発生します。

Object JDBC マッピング

Object JDBC マッピングでは、ほとんどの数値型は java.lang.Integer および java.lang.Float などの Java ラッパー・クラスにマップされ、DECIMAL および NUMBER は java.math.BigDecimal にマップされます。JDBC マッピングとの違いは、基本型を使用しないことのみです。

Object JDBC マッピングを使用する場合、戻される値はすべてオブジェクトです。値が null の属性を取得しようとすると、null オブジェクトが戻されます。

Object JDBC マッピングで使用される Java ラッパー・クラスでは、整数オーバーフローや浮動小数点での精度損失に対する保護は行われません。アクセッサ・メソッドをコールして java.lang.Integer にマップされる属性を取得する場合で、値が大きすぎて入らない場合は、例外が発生します。

これは数値型のデフォルト・マッピングです。

BigDecimal マッピング

その名前が示すとおり、すべての数値型を java.math.BigDecimal にマップします。null 値および非常に大きい値もサポートします。

Oracle マッピング

JPublisher で任意の数値、LOB またはその他のスカラー型を oracle.sql パッケージ内のクラスにマップします。たとえば、DATE 型は oracle.sql.DATE にマップされ、すべての数値型は oracle.sql.NUMBER にマップされます。オブジェクト型、コレクション型およびオブジェクト参照型の場合、JPublisher では ORAData クラスが生成されます。

Oracle マッピングでは基本型を使用しないため、すべての場合に null 値を Java null として表せます。Oracle マッピングではすべての数値型で oracle.sql.NUMBER クラスを使用するため、データベースに格納できる最大の数値を表せます。

その他のオプション設定

生成されるコードの性質には、その他の多数のオプション設定が影響するため注意してください。たとえば、オプション -compatible は下位互換性のある CustomDatum 型の生成を制御し、-access は生成されるメソッド、コンストラクタおよび属性の可視性を指定します。オプション -serializable は、生成されるオブジェクト・ラッパー・クラスに java.io.Serializable が実装されるかどうかを制御します。

データベースでの型およびパッケージの作成

JPublisher を実行する前に、データベースで必要な新規データ型を作成してください。また、Java からコールする PL/SQL パッケージ、メソッドおよびサブプログラムも、事前に Oracle9*i* にインストールする必要があります。

SQL の CREATE TYPE 文を使用して、オブジェクト型、VARRAY 型およびネストした表型をデータベース内に作成します。JPublisher では、これらのデータ型の Java クラスへのマッピングをサポートします。また、オブジェクト参照（REF）型に対応するクラスも生成されます。REF 型は SQL では明示的に宣言されません。オブジェクト型作成の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

CREATE PACKAGE 文および CREATE PACKAGE BODY 文を使用して、PL/SQL パッケージを作成し、データベースに格納します。PL/SQL には、オブジェクト型に対応付けられたメソッドを実装するのに必要なすべての機能が用意されています。これらのメソッド（ファンクションおよびプロシージャ）はユーザーのスキーマの一部としてサーバー上に常駐します。このメソッドは PL/SQL または Java で実装できます。

通常、パッケージを実装すると次のメリットが得られます。

- 関連プロシージャおよび変数のカプセル化
- public プロシージャと private プロシージャ、変数、定数およびカーソルの宣言
- パフォーマンスの向上

PL/SQL と PL/SQL パッケージ作成の詳細は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

JPublisher の操作

この項では、JPublisher の基本的な使用手順とコマンドライン構文について説明し、最後に変換例の詳細を説明します。この項の内容は、次のとおりです。

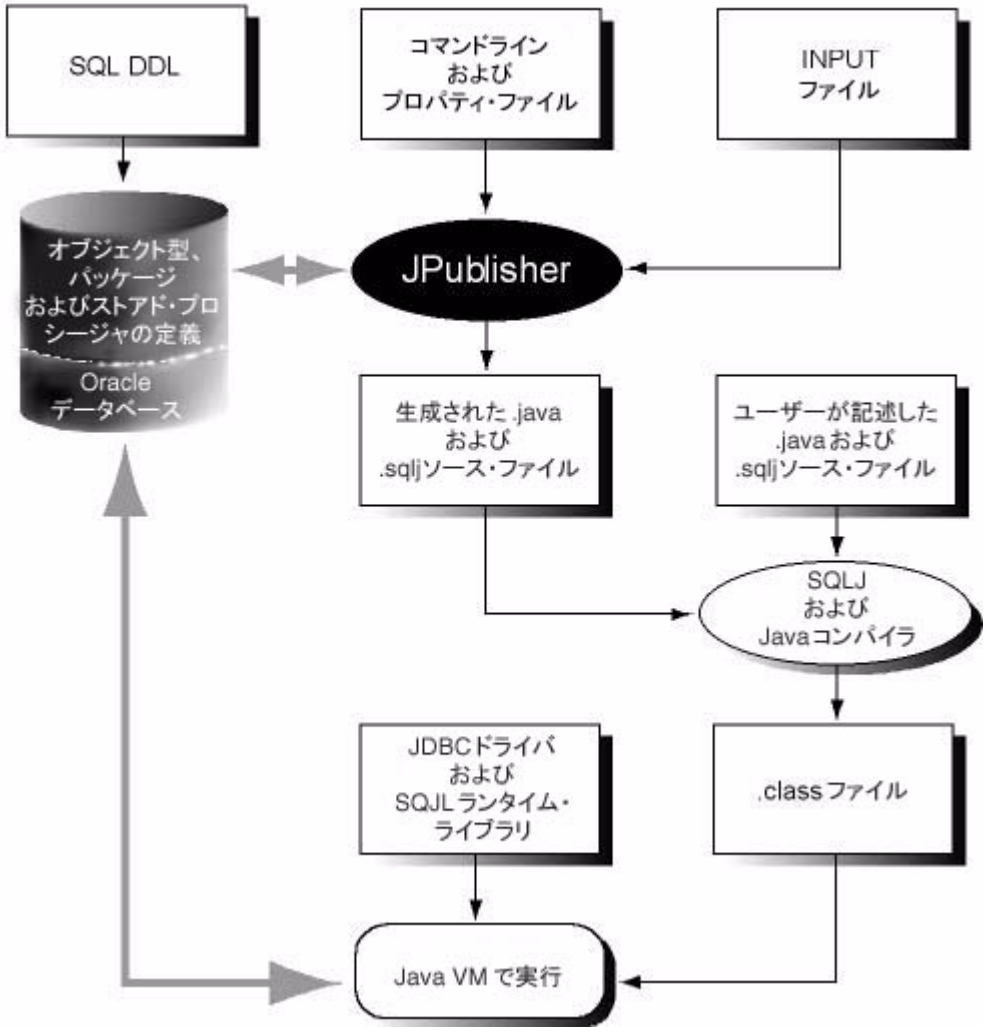
- PL/SQL パッケージおよびユーザー定義型の変換と使用
- Java でのユーザー定義のオブジェクト型、コレクション型および参照型の表現方法
- ORAData 実装の強い型指定を持つオブジェクト参照
- JPublisher コマンドライン構文
- JPublisher 変換の例

PL/SQL パッケージおよびユーザー定義型の変換と使用

この項では、図 1-1 に示すように、ユーザー定義型と PL/SQL パッケージ用のコードを変換して使用する基本的な手順について説明します。ユーザー定義型には、Oracle オブジェクトと Oracle コレクション、つまり VARRAY 型とネストした表型が含まれます。

1. 必要なユーザー定義型および PL/SQL パッケージをデータベース内に作成します。
2. JPublisher では、PL/SQL パッケージ、ユーザー定義型および参照型などを表す Java クラス用のソース・コードが生成され、指定した Java パッケージに配置されます。JPublisher で、オブジェクト参照クラス、VARRAY クラスおよびネストした表クラス用に `.java` ファイルが生成されます。JPublisher でラッパー・メソッドを生成するように指定すると、パッケージとオブジェクト型用に `.sqlj` ファイルが生成されます（オブジェクト型にメソッドがある場合）。JPublisher でラッパー・メソッドを生成しないように指定すると、オブジェクト型用のラッパー・メソッドのない `.java` ファイルが生成され、パッケージ用のクラスは生成されません。メソッドを持たないオブジェクト型の場合、JPublisher では常に `.java` ファイルが生成されます。
3. これらのクラスをユーザーのアプリケーション・コードにインポートします。
4. 生成されたクラスのメソッドを使用して、ユーザー定義型とその属性にアクセスし、操作します。
5. すべてのクラス（JPublisher で生成されたコードとユーザーのコード）をコンパイルします。SQLJ により、`.sqlj` および `.java` ファイルが変換およびコンパイルされます。または、`.java` ファイルのみの場合は Java コンパイラを起動できます。
6. コンパイル済みアプリケーションを実行します。

図 1-1 JPublisher 生成コードの変換および使用



Java でのユーザー定義のオブジェクト型、コレクション型および参照型の表現方法

この項では、ユーザーの Java プログラムでユーザー定義オブジェクト型、コレクション型、オブジェクト参照型および OPAQUE 型を表す方法について説明します。次の 3 つの方法があります。

- `ORAData` インタフェースを実装するクラスを使用します。

  JPublisher では、`oracle.sql.ORAData` インタフェースを実装するクラスが生成されます。(手動で記述することもできますが、通常はお薦めしません。)

- JDBC 2.0 API の説明に従って、`SQLData` インタフェースを実装するクラスを使用します。

  JPublisher では、`java.sql.SQLData` インタフェースを実装する SQL オブジェクト型用のクラスが生成されます。(手動で記述することもできますが、通常はお薦めしません。手動で記述する場合や、オブジェクト型の継承階層用のクラスを生成する場合は、型マップを使用してクラスを登録する必要があることに注意してください。)

  `SQLData` インタフェースを使用する場合、すべてのオブジェクト参照型は汎用的に `java.sql.Ref` のインスタンスとして表され、すべてのコレクション型は汎用的に `java.sql.Array` のインスタンスとして表されます。OPAQUE 型を表すためのメカニズムはありません。

- `oracle.sql.*` クラスを使用します。

  `oracle.sql.*` クラスを使用してオブジェクト型を汎用的に表現できます。クラス `oracle.sql.STRUCT` はすべてのオブジェクト型を表し、クラス `oracle.sql.ARRAY` はすべての VARRAY 型およびネストした表型を表し、クラス `oracle.sql.REF` はすべての REF 型を表し、クラス `oracle.sql.OPAQUE` はすべての OPAQUE 型を表します。これらのクラスは、`java.lang.String` と同様に不変です。

  汎用的な方法でオブジェクト型、コレクション型、参照型または OPAQUE 型を処理するコードを記述する場合は、このオプションを選択する必要があります。

`oracle.sql.*` クラスと比較すると、`ORAData` または `SQLData` を実装するクラスは強い型指定がされています。`ADDRESS` を表す `ORAData` オブジェクトに誤って `PERSON` オブジェクトを選択した場合などは、接続している SQLJ Translator が変換時にエラーを検出します。

JPublisher で生成された、`ORAData` または `SQLData` を実装するクラスには次のメリットもあります。

- クラスは汎用的ではなく、カスタマイズされています。オブジェクトの属性にアクセスするには、そのオブジェクトの特定の属性の後に名前が指定されている `get`*XXX*`()` および `set`*XXX*`()` メソッドを使用します。データに変更があった場合は、データベース内のオブジェクトを明示的に更新する必要があることに注意してください。

- クラスは可変です。通常、オブジェクトの属性またはコレクションの要素は変更できます。ただし、オブジェクト参照型を表す ORAData クラスは不変です。これは、オブジェクト参照には変更できるサブコンポーネントがないためです。ただし、参照オブジェクトの setValue() メソッドを使用すると、参照先のデータベース値を変更できます。
- シリアル化可能な Java ラッパー・クラス、またはオブジェクトとその属性値を出力する toString() メソッドを持つ Java ラッパー・クラスを生成できます。

基本的に、SQLData を実装するクラスに比べて、ORAData を実装するクラスの方が効率的です。これは、ORAData クラスはネイティブな Java 型への不要な変換を行わないためです。SQLData インタフェースと ORAData インタフェースの比較は、『Oracle9*i* JDBC 開発者ガイドおよびリファレンス』を参照してください。

ORAData 実装の強い型指定を持つオブジェクト参照

Oracle ORAData 実装の場合、JPublisher では、弱い型指定の oracle.sql.REF クラスを使用する場合とは異なり、常に強い型指定を持つオブジェクト参照のクラスが生成されます。これは、オブジェクト参照が強い型指定を持ち、SQL での動作を反映して、型保証を強化するためです。強い型指定を持つクラス（名前は、PERSON オブジェクトの参照の場合は PersonRef など）は、実際には oracle.sql.REF クラスのラッパーです。

このような強い型指定を持つ REF ラッパーの場合は、参照される SQL オブジェクトのインスタンスを、対応する Java クラスのインスタンスの形式で生成する getValue() メソッドがあります。（また、継承の場合は、対応する Java クラスのサブクラスのインスタンスとして生成されます。）たとえば、データベースに PERSON オブジェクト型があり、Java クラス Person が対応している場合は、Java クラス PersonRef も生成されます。PersonRef クラスの getValue() メソッドは、データベース内の PERSON オブジェクトのデータを含む Person インスタンスを戻します。また、JPublisher では、PersonRef クラスに対する静的 cast() メソッドを生成することもできます。これにより、他の型指定の参照を PersonRef インスタンスに変換できます。

SQL オブジェクト型が属性としてオブジェクト参照を持っている場合、そのオブジェクト型に対応する Java クラスの属性は、該当する参照型に対応する Java クラスのインスタンスになります。たとえば、MANAGER REF 属性を持つ PERSON オブジェクトがある場合、それに対応する Java クラス Person は ManagerRef 属性を持ちます。

標準的な SQLData 実装の場合、強い型指定を持つオブジェクト参照は、標準に含まれておらず、サポートされません。JPublisher ではカスタム参照クラスは作成されず、java.sql.Ref または oracle.sql.REF を参照型として使用する必要があります。

JPublisher コマンドライン構文

ほとんどのオペレーティング・システムでは、次のように jpub の後に一連のオプション設定を入力し、コマンドラインで JPublisher を起動します。

```
jpub -option1=value1 -option2=value2 ...
```

JPublisher は、データベースに接続してユーザー指定の型またはパッケージの宣言を取得してから、1 つ以上のカスタム Java ファイルを生成し、変換されたオブジェクト型または PL/SQL パッケージの名前を標準出力に書き出します。

JPublisher を起動するコマンドの例を次に示します（ここでは折り返されていますが、1 行のコマンドラインです）。

```
jpub -user=scott/tiger -input=demoin -numbertypes=oracle -usertypes=oracle -dir=demo
-package=corp
```

コマンドは 1 行に入力しますが、必要に応じて改行されます。わかりやすくするために、この章では入力ファイル（-input オプションで指定したファイル）を INPUT ファイルとします（他の種類の入力ファイルと区別するためです）。

このコマンドでは、JPublisher はユーザー名 SCOTT とパスワード TIGER でデータベースに接続し、INPUT ファイル demoin の指示に基づいて、データ型を Java クラスに変換します。-numbertypes=oracle オプションで、オブジェクト属性の型を Oracle が提供する Java クラスにマップするように JPublisher に指示し、-usertypes=oracle オプションで Oracle 固有の ORAData クラスを生成するように指示します。JPublisher では、生成されるクラスはディレクトリ demo 内のパッケージ corp に配置されます。

これらのオプションの詳細は、3-2 ページの「JPublisher オプション」を参照してください。

注意：

- 等号（=）の前後に空白は入力できません。
- コマンドラインでオプションを指定せずに JPublisher を実行すると、オプション・リストが表示されて終了します。

JPublisher 変換の例

この項では、オブジェクト型の JPublisher 変換の例を示します。この時点では、JPublisher で生成されるコードの詳細を気にする必要はありません。JPublisher 入力ファイルと出力ファイル、オプション、データ型マッピングおよび変換の詳細は、このマニュアルの後半で説明します。

次のようにオブジェクト型 EMPLOYEE を作成します。

```
CREATE TYPE employee AS OBJECT
(
    name        VARCHAR2(30),
    empno       INTEGER,
    deptno      NUMBER,
    hiredate    DATE,
    salary      REAL
);
```

INTEGER、NUMBER および REAL 型はすべてデータベース内に NUMBER 型として格納されますが、変換後は選択した -numbertypes オプションの値に基づいて、Java プログラムで別の形で表されます。

JPublisher では、次のようにコマンドラインに従って型を変換します。

```
jpub -user=scott/tiger -dir=demo -numbertypes=objectjdbc -builtintypes=jdbc
-package=corp -case=mixed -sql=Employee
```

ここでは折り返されていますが、1 行のコマンドラインです。これらのオプションの詳細は、3-2 ページの「JPublisher オプション」を参照してください。

EMPLOYEE オブジェクト型ではメソッドが定義されていないため、JPublisher では .sqlj ファイルではなく .java ファイルが生成されることに注意してください。

JPublisher コマンドラインで -dir=demo および -package=corp が指定されているため、変換後のクラス Employee は次のディレクトリにある Employee.java に書き込まれます。

```
./demo/corp/Employee.java           (UNIX)
.¥demo¥corp¥Employee.java           (Windows NT)
```

Employee.java クラス・ファイルには次のコードが含まれます。

注意： JPublisher で生成されるコードの詳細は、今後のリリースで変更されることがあります。特に、非パブリック・メソッド、非パブリック・フィールドおよびすべてのメソッド本体の生成方法が変更されることがあります。

```
package corp;

import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sql.ORAData;
import oracle.sql.ORADataFactory;
import oracle.sql.Datum;
import oracle.sql.STRUCT;
import oracle.jpub.runtime.MutableStruct;

public class Employee implements ORAData, ORADataFactory
{
  public static final String _SQL_NAME = "SCOTT.EMPLOYEE";
  public static final int _SQL_TYPECODE = OracleTypes.STRUCT;

  protected MutableStruct _struct;

  private static int[] _sqlType =  { 12,4,2,91,7 };
  private static ORADataFactory[] _factory = new ORADataFactory[5];
  protected static final Employee _EmployeeFactory = new Employee(false);

  public static ORADataFactory getORADataFactory()
  { return _EmployeeFactory; }

  /* constructor */
  protected Employee(boolean init)
  { if(init) _struct = new MutableStruct(new Object[5], _sqlType, _factory); }
  public Employee()
  { this(true); }
  public Employee(String name, Integer empno, java.math.BigDecimal deptno,
java.sql.Timestamp hiredate, Float salary)
   throws SQLException
  { this(true);
    setName(name);
    setEmpno(empno);
    setDeptno(deptno);
    setHiredate(hiredate);
    setSalary(salary);
  }

  /* ORAData interface */
  public Datum toDatum(Connection c) throws SQLException
  {
    return _struct.toDatum(c, _SQL_NAME);
  }
```

```
  /* ORADataFactory interface */
  public ORAData create(Datum d, int sqlType) throws SQLException
  { return create(null, d, sqlType); }
  protected ORAData create(Employee o, Datum d, int sqlType) throws SQLException
  {
    if (d == null) return null;
    if (o == null) o = new Employee(false);
    o._struct = new MutableStruct((STRUCT) d, _sqlType, _factory);
    return o;
  }
  /* accessor methods */
  public String getName() throws SQLException
  { return (String) _struct.getAttribute(0); }

  public void setName(String name) throws SQLException
  { _struct.setAttribute(0, name); }

  public Integer getEmpno() throws SQLException
  { return (Integer) _struct.getAttribute(1); }

  public void setEmpno(Integer empno) throws SQLException
  { _struct.setAttribute(1, empno); }

  public java.math.BigDecimal getDeptno() throws SQLException
  { return (java.math.BigDecimal) _struct.getAttribute(2); }

  public void setDeptno(java.math.BigDecimal deptno) throws SQLException
  { _struct.setAttribute(2, deptno); }

  public java.sql.Timestamp getHiredate() throws SQLException
  { return (java.sql.Timestamp) _struct.getAttribute(3); }

  public void setHiredate(java.sql.Timestamp hiredate) throws SQLException
  { _struct.setAttribute(3, hiredate); }

  public Float getSalary() throws SQLException
  { return (Float) _struct.getAttribute(4); }

  public void setSalary(Float salary) throws SQLException
  { _struct.setAttribute(4, salary); }

}
```

コード生成に関する注意

- Oracle9*i* リリース 2（9.2）以上の Oracle JPublisher では、オブジェクトの属性に基づいてオブジェクト・コンストラクタも生成されます。
- 他の設定を使用すると、追加の `private` または `public` メソッドも生成できます。たとえば、`-serializable=true` に設定すると、インタフェース `java.io.Serializable` を実装するオブジェクト・ラッパーが生成され、`private writeObject` および `readObject` メソッドが生成されます。`-tostring=true` に設定すると、`public toString()` メソッドが追加生成されます。
- Oracle9*i* リリース（および Oracle8*i* リリース 8.1.7）の場合、JPublisher で生成される SQL オブジェクト型用のコードでは、`_struct` フィールドがプロテクトされています。これは内部クラス `oracle.jpub.runtime.MutableStruct` のインスタンスであり、このインスタンスにはデータがオリジナルの SQL 形式で含まれています。通常、このフィールドを直接参照することはありません。かわりに、JPublisher で `.sqlj` ファイルが生成されるように必要に応じて `-methods=always` または `-methods=named` 設定を使用してから、サブクラス化するときにメソッド `setFrom()` および `setValueFrom()` を使用します。2-35 ページの「setFrom()、setValueFrom() および setContextFrom() メソッド」を参照してください。
- Oracle8*i* 互換モードでは、SQLJ 接続コンテキスト・インスタンスである `_ctx` フィールドもプロテクトされています。詳細は、2-47 ページの「Oracle8i 互換モード」を参照してください。
- Oracle8*i* JPublisher では、`ORAData` および `ORADataFactory` のかわりに、現在では使用不可の `CustomDatum` および `CustomDatumFactory` インタフェースの実装が生成されることに注意してください。実際には、JPublisher の `-compatible` オプションを指定して、使用不可のインタフェースを生成できます。これが必要になるのは、Oracle8*i* JDBC ドライバを使用している場合です。

また、JPublisher では `EmployeeRef.java` クラスも生成されます。ソース・コードは、次のとおりです。

```
package corp;

import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sql.ORAData;
import oracle.sql.ORADataFactory;
import oracle.sql.Datum;
import oracle.sql.REF;
import oracle.sql.STRUCT;

public class EmployeeRef implements ORAData, ORADataFactory
{
  public static final String _SQL_BASETYPE = "SCOTT.EMPLOYEE";
```

```
  public static final int _SQL_TYPECODE = OracleTypes.REF;

  REF _ref;

private static final EmployeeRef _EmployeeRefFactory = new EmployeeRef();

  public static ORADataFactory getORADataFactory()
  { return _EmployeeRefFactory; }
  /* constructor */
  public EmployeeRef()
  {
  }

  /* ORAData interface */
  public Datum toDatum(Connection c) throws SQLException
  {
    return _ref;
  }

  /* ORADataFactory interface */
  public ORAData create(Datum d, int sqlType) throws SQLException
  {
    if (d == null) return null;
    EmployeeRef r = new EmployeeRef();
    r._ref = (REF) d;
    return r;
  }

  public static EmployeeRef cast(ORAData o) throws SQLException
  {
     if (o == null) return null;
     try { return (EmployeeRef) getORADataFactory().create(o.toDatum(null),
OracleTypes.REF); }
     catch (Exception exn)
     { throw new SQLException("Unable to convert "+o.getClass().getName()+" to
EmployeeRef: "+exn.toString()); }
  }

  public Employee getValue() throws SQLException
  {
     return (Employee) Employee.getORADataFactory().create(
       _ref.getSTRUCT(), OracleTypes.REF);
  }
```

```
  public void setValue(Employee c) throws SQLException
  {
    _ref.setValue((STRUCT) c.toDatum(_ref.getJavaSqlConnection()));
  }
}
```

Oracle9*i* リリース 2（9.2）以上では、他の強い型指定を持つ参照から強い型指定を持つ参照のインスタンスにキャストするための `public static cast()` メソッドも生成されることに注意してください。

4-7 ページの「例 : JPublisher のオブジェクト属性のマッピング」で、さらに多くのオブジェクト・マッピングの例を参照できます。

2

JPublisher の概念

この章では、JPublisher の基礎となる概念と操作の詳細を説明します。この章の内容は、次のとおりです。

- データ型マッピングの詳細
- JPublisher で生成されるクラスの概念
- JPublisher による SQLJ クラス（.sqlj）の生成
- JPublisher による Java クラス（.java）の生成
- JPublisher で生成されたクラスのユーザー作成サブクラス
- JPublisher による継承のサポート
- 下位互換性および移行
- JPublisher の制限事項

データ型マッピングの詳細

前述のように、型マッピング・オプション（-builtintypes、-lobtypes、-numbertypes および -usertypes）を使用する場合、次のデータ型マッピングの設定のうちいずれかを指定できます。

- jdbc
- objectjdbc（-numbertypes の場合のみ）
- bigdecimal（-numbertypes の場合のみ）
- oracle

これらのマッピングは、1-17 ページの「データ型マッピングの概要」で説明したように、JPublisher で生成されるメソッドで JPublisher で使用される引数および結果の型に影響を与えます。

JPublisher でオブジェクト型について生成されるクラスには、オブジェクト属性用の get*XXX*() および set*XXX*() メソッドがあります。VARRAY 型またはネストした表型用に JPublisher で生成されるクラスには、配列またはネストした表の要素にアクセスする get*XXX*() メソッドおよび set*XXX*() メソッドがあります。オプション -methods=true を使用すると、オブジェクト型または PL/SQL パッケージ用に JPublisher で生成されるクラスには、オブジェクト型またはパッケージのいくつかのサーバー・メソッドをコールするラッパー・メソッドがあります。マッピング・オプションは、これらのメソッドで使用する引数および結果の型を制御します。

JDBC マッピングおよび Object JDBC マッピングでは、標準の Java 操作で処理可能な通常の Java 型が使用されます。ユーザーの JDBC プログラムでオブジェクト型として格納された Java オブジェクトを操作している場合は、JDBC または Object JDBC マッピングを選択できます。

Oracle マッピングは最も効率のよいマッピングです。oracle.sql 型は Oracle 内部データ型とほぼ一致しているため、Java と SQL フォーマット間のデータ変換はほとんど、あるいはまったく必要ありません。情報を失うことはなく、データの処理およびデータの取出しにおいてより柔軟性があります。標準の SQL 型に対する Oracle マッピングは、データベース内でデータを操作している場合やデータを移動している場合（既存のある表から別の表への SELECT および INSERT 操作を実行している場合など）に最も便利な表現です。データ・フォーマット変換が必要な場合は、oracle.sql.* クラスのメソッドを使用してネイティブな Java 型に変換できます。

使用するマッピング方法を決める場合、データ・フォーマット変換はユーザーのプログラムとサーバー間のデータ転送のコストの一部にすぎない点に注意してください。

Oracle と JDBC の型への SQL と PL/SQL のマッピング

表 2-1 に、Oracle マッピングおよび JDBC マッピングを使用した SQL データ型および PL/SQL データ型の Java 型へのマッピングを示します。PL/SQL メソッドの引数または結果の型として、この表でサポート対象となっているすべてのデータ型を使用できます。オブジェクト属性の型として、2-6 ページの「使用できるオブジェクト属性の型」に示すデータ型のサブセットを使用できます。

SQL および PL/SQL データ型列にはすべての使用可能なデータ型が含まれます。

Oracle マッピング列は、すべての型マッピング・オプションを `oracle` に設定した場合に JPublisher で使用される対応 Java 型を示します。これらの型は Oracle が提供する `oracle.sql` パッケージ内にあり、Oracle データ型を Java 型に変換したときに発生するオーバーヘッドが最小となるように設計されています。`oracle.sql` パッケージの詳細は、『Oracle9*i* JDBC 開発者ガイドおよびリファレンス』を参照してください。

JDBC マッピング列は、すべての型マッピング・オプションを `jdbc` に設定した場合に JPublisher で使用される対応 Java 型を示します。標準 SQL データ型の場合、JPublisher により JDBC 仕様で指定された Java 型が使用されます。Oracle 拡張要素である SQL データ型の場合、JPublisher により `oracle.sql.*` 型が使用されます。型マッピング・オプションを `objectjdbc` に設定すると、対応する型は **JDBC マッピング**列の場合と同じになります。ただし、`int` などの Java 基本型は、`java.lang.Integer` などの対応するオブジェクト型に置き換えられます。PL/SQL の `BOOLEAN` から SQL の `NUMBER`、さらに Java の `boolean` へのマッピングなど、JPublisher の型マップに明示的に定義されている型の対応付けは、マッピング・オプションの設定の影響を受けません。

PL/SQL にのみ関係する型など、JPublisher で直接サポートされないデータ型がいくつかあります。これらの制限事項は、等価の SQL 型と Java 型、および PL/SQL 表現と SQL 表現の間の PL/SQL 変換ファンクションを提供することで回避できます。これらの変換については、後述します。

表 2-1 Oracle および JDBC マッピング・クラスに対する SQL および PL/SQL データ型

SQL および PL/SQL データ型	Oracle マッピング	JDBC マッピング
CHAR、CHARACTER、LONG、STRING、VARCHAR、VARCHAR2	oracle.sql.CHAR	java.lang.String
NCHAR、NVARCHAR2	oracle.sql.NCHAR（注意 1）	oracle.sql.NString（注意 1）
RAW、LONG RAW	oracle.sql.RAW	byte[]
BINARY_INTEGER、NATURAL、NATURALN、PLS_INTEGER、POSITIVE、POSITIVEN、SIGNTYPE、INT、INTEGER	oracle.sql.NUMBER	int
DEC、DECIMAL、NUMBER、NUMERIC	oracle.sql.NUMBER	java.math.BigDecimal

表 2-1 Oracle および JDBC マッピング・クラスに対する SQL および PL/SQL データ型 （続き）

SQL および PL/SQL データ型	Oracle マッピング	JDBC マッピング
DOUBLE PRECISION、FLOAT	oracle.sql.NUMBER	double
SMALLINT	oracle.sql.NUMBER	short
REAL	oracle.sql.NUMBER	float
DATE	oracle.sql.DATE	java.sql.Timestamp
TIMESTAMP、 TIMESTAMP WITH TZ、 TIMESTAMP WITH LOCAL TZ	oracle.sql.TIMESTAMP、 oracle.sql.TIMESTAMPTZ、 oracle.sql.TIMESTAMPLTZ	java.sql.Timestamp
INTERVAL YEAR TO MONTH INTERVAL DAY TO SECOND	String（注意 2）	String（注意 2）
ROWID、UROWID	oracle.sql.ROWID	oracle.sql.ROWID
BOOLEAN	boolean（注意 3）	boolean（注意 3）
CLOB	oracle.sql.CLOB	java.sql.Clob
BLOB	oracle.sql.BLOB	java.sql.Blob
BFILE	oracle.sql.BFILE	oracle.sql.BFILE
NCLOB	oracle.sql.NCLOB（注意 1）	oracle.sql.NCLOB（注意 1）
オブジェクト型	生成されたクラス	生成されたクラス
SQLJ オブジェクト型	型の作成時に定義された Java クラス	型の作成時に定義された Java クラス
OPAQUE 型	生成されるクラスまたは事前定義済みのクラス（注意 4）	生成されるクラスまたは事前定義済みのクラス（注意 4）
RECORD 型	SQL オブジェクト型へのマッピングを使用（注意 5）	SQL オブジェクト型へのマッピングを使用（注意 5）
ネストした表、VARRAY	oracle.sql.ARRAY を使用して実装された生成クラス	java.sql.Array
オブジェクト型の参照	oracle.sql.REF を使用して実装された生成クラス	java.sql.Ref
REF CURSOR	java.sql.ResultSet	java.sql.ResultSet
スカラー（数値または文字）索引付き表	Java 配列へのマッピングを使用（注意 6）	Java 配列へのマッピングを使用（注意 6）
索引付き表	SQL コレクションへのマッピングを使用（注意 7）	SQL コレクションへのマッピングを使用（注意 7）
ユーザー定義サブタイプ	基本型用と同じです	基本型用と同じです

データ型マッピングに関する注意 次の注意事項は、前述の表のマークが付いている項目に対応しています。

1. Java クラス `oracle.sql.NCHAR`、`oracle.sql.NCLOB` および `oracle.sql.NString` は、JDBC の一部ではなく、SQLJ ランタイムとともに配布されます。SQLJ では、これらのクラスを使用して、対応するクラス `oracle.sql.CHAR`、`oracle.sql.CLOB` および `java.lang.String` の使用が NCHAR 形式で表現されます。

2. SQL の `INTERVAL` 型から `VARCHAR2` および Java の `String` へのマッピングは、デフォルトの JPublisher 型マップに定義されています。この場合は、`SYS.SQLJUTL` パッケージからの変換ファンクションが使用されます。2-17 ページの「JPublisher のデフォルトの型マップとユーザー型マップ」も参照してください。

3. PL/SQL の `BOOLEAN` から SQL の `NUMBER` および Java の `boolean` へのマッピングは、デフォルトの JPublisher 型マップに定義されています。この場合は、`SYS.SQLJUTL` パッケージからの変換ファンクションが使用されます。

4. SQL の OPAQUE 型 `SYS.XMLTYPE` から Java クラス `oracle.xdb.XMLType` へのマッピングは、デフォルトの JPublisher 型マップに定義されています。他の OPAQUE 型の場合は、通常、対応する Java クラスがベンダーから提供されます。この場合は、SQL の OPAQUE 型および対応する Java ラッパー・クラス間の対応付けを定義する、JPublisher 型マップ・エントリを指定するのみで済みます。JPublisher では、型マップ・エントリのない OPAQUE 型が検出されると、その OPAQUE 型用の Java ラッパー・クラスが生成されます。2-7 ページの「OPAQUE 型の型マッピング・サポート」も参照してください。

5. PL/SQL の RECORD 型をサポートするには、対応する SQL オブジェクト型と、SQL 型と PL/SQL 型の間でマップする 2 つの PL/SQL 変換ファンクション（変換方向ごとに 1 つのファンクション）を定義する必要があります。また、JPublisher を使用して SQL 型用の Java ラッパー・クラスを公開する必要があります。この時点で、PL/SQL 型、SQL 型および Java 型の対応付けと PL/SQL 変換ファンクションを定義する型マップ・エントリを JPublisher 用に用意できます。これにより、JPublisher では、PL/SQL の RECORD 型を使用する PL/SQL またはメソッドのシグネチャを自動的に公開できます。2-13 ページの「PL/SQL RECORD 型の型マッピング・サポート」も参照してください。

6. JDBC OCI ドライバを使用して PL/SQL ストアド・プロシージャまたはオブジェクト・メソッドをコールする場合は、PL/SQL の TABLE 型とも呼ばれるスカラーの索引付き表が直接サポートされます。この場合は、JPublisher に対して、PL/SQL のスカラーの索引付き表型および対応する Java 配列型を含む型マップ・エントリを指定します。JPublisher では、このスカラーの索引付き表を使用する PL/SQL またはオブジェクト・メソッドのシグネチャを自動的に公開できます。2-8 ページの「JDBC OCI を使用したスカラーの索引付き表の型マッピング・サポート」も参照してください。

7. PL/SQL の索引付き表の型をサポートするには、対応する SQL コレクション型と、SQL 型および PL/SQL 型の間でマップする 2 つの PL/SQL 変換ファンクションを定義する必要があります。また、JPublisher を使用して SQL コレクション型用の Java ラッパー・クラスを公開する必要があります。（索引付き表の要素が PL/SQL レコードの場合は、

これらのレコードと対応する SQL 型および Java 型の間の完全な JPublisher マッピング・サポートも提供する必要があります。）この時点で、PL/SQL 型、SQL 型および Java 型の対応付けと PL/SQL 変換ファンクションを定義する型マップ・エントリを JPublisher 用に用意できます。これにより、JPublisher では、この PL/SQL 索引付き表の型を使用する PL/SQL またはメソッドのシグネチャを自動的に公開できます。2-15 ページの「PL/SQL 索引付き表の型の型マッピング・サポート」も参照してください。

注意： 数値型のみに影響を与える Object JDBC および `BigDecimal` マッピングは、3-9 ページの「数値型のマッピング（-numbertypes）」で詳しく説明します。

使用できるオブジェクト属性の型

オブジェクト属性の型として表 2-1 にリストされている PL/SQL データ型のサブセットを使用できます。この種のデータ型は、表と同じ Oracle マッピングおよび JDBC マッピングを持ちます。

- `CHAR`、`VARCHAR`、`VARCHAR2`、`CHARACTER`
- `NCHAR`、`NVARCHAR2`
- `DATE`
- `DECIMAL`、`DEC`、`NUMBER`、`NUMERIC`
- `DOUBLE PRECISION`、`FLOAT`
- `INTEGER`、`SMALLINT`、`INT`
- `REAL`
- `RAW`、`LONG RAW`
- `CLOB`
- `BLOB`
- `BFILE`
- `NCLOB`
- オブジェクト型、OPAQUE 型、SQLJ オブジェクト型
- ネストした表、VARRAY 型
- 参照型

TIMESTAMP 型 `TIMESTAMP`、`TIMESTAMP WITH TIMEZONE` および `TIMESTAMP WITH LOCAL TIMEZONE` は、JPublisher ではオブジェクト属性としてサポートされます。ただし、Oracle9*i* リリース 2（9.2）の JDBC では、これらの型はオブジェクト属性としてサポートされません。

JDBC でサポートされないデータ型の使用

通常、JPublisher では、サポートされない PL/SQL 型が使用されている PL/SQL ストアド・プロシージャまたはファンクションやオブジェクト型のメソッドが検出されると、エラー・メッセージが発行され、ラッパー・クラスには対応するメソッドが生成されません。ただし、適切な型マッピング情報を指定すると、この種のメソッドも JPublisher で自動的に公開できます。また、JPublisher の型マップ・エントリを使用すると、SQL の OPAQUE 型や特定のスカラーの PL/SQL 索引付き表の型などの型を、対応する Java クラスに対応付けることができます。次の項では、JPublisher で提供される型マッピング・サポートの様々な側面について説明します。

- OPAQUE 型の型マッピング・サポート
- JDBC OCI を使用したスカラーの索引付き表の型マッピング・サポート
- PL/SQL 変換ファンクションを介した型マッピングのサポート
- PL/SQL RECORD 型の型マッピング・サポート
- PL/SQL 索引付き表の型の型マッピング・サポート
- JPublisher のデフォルトの型マップとユーザー型マップ

OPAQUE 型の型マッピング・サポート

Oracle JDBC と Oracle SQLJ には、`oracle.sql.ORAData` インタフェースを実装する Java クラスとして公開される SQL の OPAQUE 型のサポートが用意されています。この種のクラスには、次の `public static` フィールドおよびメソッドも含める必要があります。

```
public static String _SQL_NAME = "SQL_name_of_OPAQUE_type";
public static int _SQL_TYPECODE = OracleTypes.OPAQUE;
public static ORADataFactory getORADataFactory() { ... }
```

Oracle 9*i* リリース 2（9.2）では、SQL の OPAQUE 型 `SYS.XMLTYPE` は、対応する Java ラッパー・クラス `oracle.xdb.XMLType` でサポートされます。

ここで説明する規則に従った SQL の OPAQUE 型用の Java ラッパー・クラスがあれば、次のコマンドライン・オプションを使用して、この対応付けを JPublisher に対して指定できます。

```
-addtypemap=sql_opaque_type:java_wrapper_class
```

この方法では、`XMLTYPE` 用の事前定義の型の対応付けが、JPublisher に対して次のように明示的に指定されている可能性があります。

```
-addtypemap=SYS.XMLTYPE:oracle.xdb.XMLType
```

JPublisher では、型の対応付けが指定されていない SQL の OPAQUE 型が検出されるたびに、実際には Java ラッパー・クラスが公開されます。`SCOTT` スキーマに定義されている次の SQL 型を考えます。

```
CREATE TYPE X_TYP AS OBJECT (xml SYS.XMLTYPE);
```

属性 xml は oracle.xdb.XMLType として公開され、SYS.XMLTYPE に対して事前定義済みの型マッピングに対応していることに注意してください。次のコマンドラインでは、X_TYP が Java クラス XTyp として公開されます。

```
jpub -u scott/tiger -s X_TYP:XTyp
```

JPublisher のデフォルト型マップを消去すると、SYS.XMLTYPE 属性用に追加のラッパー・クラス Xmltype が自動的に生成されます。この動作は、JPublisher を次のように起動することで確認できます。

```
jpub -u scott/tiger -s X_TYP:XTyp -defaulttypemap=
```

オプション -defaulttypemap では、JPublisher のデフォルトの型マップを設定します。前述の例のように、このオプションに値を指定しなければ、デフォルトの型マップは空の文字列に設定（実際には消去）されます。デフォルトの型マップの詳細は、2-17 ページの「JPublisher のデフォルトの型マップとユーザー型マップ」を参照してください。

JDBC OCI を使用したスカラーの索引付き表の型マッピング・サポート

Oracle JDBC OCI ドライバでは、数値または文字の要素を持つ PL/SQL のスカラーの索引付き表が直接サポートされます。（JDBC OCI ドライバを使用しない場合は、2-15 ページの「PL/SQL 索引付き表の型の型マッピング・サポート」を参照。）数値要素を持つ索引付き表は、次の Java 配列型にマップできます。

- int[]
- double[]
- float[]
- java.math.BigDecimal[]
- oracle.sql.NUMBER[]

文字要素を持つ索引付き表は、次の Java 配列型にマップできます。

- String[]
- oracle.sql.CHAR[]

前述のような状況では、索引付き表の型について次の情報を指定する必要があります。

- OUT または IN OUT パラメータの位置に索引付き表を使用するたびに、最大要素数を指定する必要があります。（それ以外の場合、この指定はオプションです。）最大要素数を定義するには、Java 配列割当て用の構文を使用します。たとえば、最大 100 個の要素に対応できる型を示す int[100]、または最大 20 個の要素に対応できる型を示す oracle.sql.CHAR[20] を指定できます。

- 文字要素を持つ索引付き表の場合は、必要に応じて個々の要素の最大サイズ（バイト単位）を指定できます。この設定を定義するには、SQL と同様のサイズ構文を使用します。たとえば、IN 引数に使用する索引付き表の場合は、`String[](30)` を指定できます。また、最大長が 20 で、その各要素が 255 バイト以内である索引付き表の場合は、`oracle.sql.CHAR[20](255)` を指定します。

JPublisher オプション `-addtypemap` を使用してユーザーの型マップに指示を追加し、この指示では、スカラーの索引付き表である PL/SQL 型および対応する Java 配列型の間の対応付けを指定します。前述の構文を使用して指定したサイズ・ヒントは、生成される SQLJ 文に埋め込まれるため、実行時に JDBC に与えられます。

たとえば、スキーマ SCOTT 内の PL/SQL パッケージ INDEXBY の定義から抜粋した次のコード部分を考えます。このコードがファイル `indexby.sql` 内で使用可能であるとします。

```
create or replace package indexby as

--  jpub.addtypemap=SCOTT.INDEXBY.VARCHAR_ARY:String[1000](4000)
--  jpub.addtypemap=SCOTT.INDEXBY.INTEGER_ARY:int[1000]
--  jpub.addtypemap=SCOTT.INDEXBY.FLOAT_ARY:double[1000]

 type varchar_ary IS TABLE OF VARCHAR2(4000) INDEX BY BINARY_INTEGER;
 type integer_ary IS TABLE OF INTEGER        INDEX BY BINARY_INTEGER;
 type float_ary   IS TABLE OF NUMBER         INDEX BY BINARY_INTEGER;

 function get_float_ary RETURN float_ary;
 procedure pow_integer_ary(x integer_ary, y OUT integer_ary);
 procedure xform_varchar_ary(x IN OUT varchar_ary);

end indexby;
/
create or replace package body indexby is ...
/
```

3 つの索引付き表の型のマッピングに必要な `-addtypemap` ディレクティブを次に示します。

```
-addtypemap=SCOTT.INDEXBY.VARCHAR_ARY:String[1000](4000)
-addtypemap=SCOTT.INDEXBY.INTEGER_ARY:int[1000]
-addtypemap=SCOTT.INDEXBY.FLOAT_ARY:double[1000]
```

使用中のオペレーティング・システム・シェルによっては、大カッコ **[...]** や丸カッコ **(...)** を含むオプションを引用符で囲む必要があるため注意してください。または、この種のオプションを次のように JPublisher のプロパティ・ファイルに挿入すると、引用符で囲まずにすみます。

```
jpub.addtypemap=SCOTT.INDEXBY.VARCHAR_ARY:String[1000](4000)
jpub.addtypemap=SCOTT.INDEXBY.INTEGER_ARY:int[1000]
jpub.addtypemap=SCOTT.INDEXBY.FLOAT_ARY:double[1000]
```

プロパティ・ファイルの詳細は、3-30 ページの「プロパティ・ファイルの構造および構文」を参照してください。

また、便利な機能として、プロパティ・ファイル内の JPublisher ディレクティブは、接頭辞 `--`（2 つのハイフン）を付けると認識されますが、`jpub.` または `-- jpub.` で始まらないエントリは単に無視されます。つまり、JPublisher ディレクティブを SQL スクリプトに挿入し、同じ SQL スクリプトを JPublisher のプロパティ・ファイルとして再利用できます。したがって、次のように、`INDEXBY` パッケージを定義するために `indexby.sql` スクリプトを起動した後、JPublisher を実行して、このパッケージを Java クラス `IndexBy` として公開できます。

```
jpub -u scott/tiger -s INDEXBY:IndexBy -props=indexby.sql
```

前述のように、スカラーの索引付き表のマッピングは、JDBC OCI ドライバでのみ使用できます。他のドライバを使用している場合や、ドライバに依存しないコードを作成する場合は、索引付き表の型に対応する SQL 型と、2 つの型の間でマップする変換ファンクションを定義する必要があります。2-15 ページの「PL/SQL 索引付き表の型の型マッピング・サポート」を参照してください。

PL/SQL 変換ファンクションを介した型マッピングのサポート

この項では、Java コード内の PL/SQL 型をサポートするために、対応する SQL 型に変換する PL/SQL ファンクションを介して JPublisher で使用される一般的なメカニズムについて説明します。次の項では、PL/SQL の RECORD 型と PL/SQL の索引付き表の型に固有の、それぞれのマッピングの問題について説明します。

通常、Java プログラムでは、PL/SQL 固有の型のバインドをサポートしていません。（ただし、スカラーの索引付き表は例外です。2-8 ページの「JDBC OCI を使用したスカラーの索引付き表の型マッピング・サポート」を参照してください。）この種の型を Java から使用するには、PL/SQL コードを使用して SQL 型にマップし、これらの SQL 型に Java からアクセスする必要があります。

JPublisher では、このタスクが簡単になります。特定の PL/SQL 型について、JPublisher の型マップ・エントリに次の情報を指定します。

- PL/SQL の型の名前（通常は次の書式を使用）

  `SCHEMA.PACKAGE.TYPE`.

- 対応する Java（ラッパー）クラスの名前
- PL/SQL の型に対応する SQL 型の名前

  この型を Java ラッパー・クラスに直接マップできるようにする必要があります。たとえば、SQL 型が `NUMBER` の場合、対応する Java クラスの型は `int`、`double`、`Integer`、

Double、java.math.BigDecimal または oracle.sql.NUMBER などとなります。また、SQL 型がオブジェクト型の場合、対応する Java クラスは対応するオブジェクト・ラッパー・クラスとなります。このラッパー・クラスは、通常は JPublisher によって生成され、ORAData または SQLData インタフェースが実装されます。

- SQL 型を PL/SQL 型にマップする PL/SQL ファンクション（変換ファンクション）の名前
- PL/SQL 型を SQL 型にマップする PL/SQL ファンクション（変換ファンクション）の名前

この -addtypemap は、次の書式で指定します。

```
-addtypemap=plsql_type:java_type:sql_type:sql_to_plsql_fun:plsql_to_sql_fun
```

たとえば、PL/SQL 型 BOOLEAN をサポートするための型マップ・エントリを考えます。この型マップ・エントリは次の指定で構成されます。

- PL/SQL 型の名前（BOOLEAN）
- Java の boolean にマップするための指定
- 対応する SQL 型（INTEGER）

  JDBC では、boolean 値は特殊な数値とみなされます。
- SQL から PL/SQL（NUMBER から BOOLEAN）にマップする PL/SQL ファンクションの名前（INT2BOOL）

  このファンクションのコードを次に示します。

```
function int2bool(i INTEGER) return BOOLEAN is
begin if i is null then return null;
      else return i<>0;
      end if;
end int2bool;
```

- PL/SQL から SQL（BOOLEAN から NUMBER）にマップする PL/SQL ファンクションの名前（BOOL2INT）

  このファンクションのコードを次に示します。

```
function bool2int(b BOOLEAN) return INTEGER is
begin if b is null then return null;
      elsif b then return 1;
      else return 0; end if;
end bool2int;
```

前述の指定をすべて次の型マップ・エントリに使用します。

```
-addtypemap=BOOLEAN:boolean:INTEGER:INT2BOOL:BOOL2INT
```

この型マップ・エントリでは、SQL 型、Java 型、2 つの変換ファンクションが、それぞれ SQL、Java および PL/SQL で定義されているものとみなされます。PL/SQL の BOOLEAN に関するエントリは、すでに JPublisher のデフォルトの型マップに存在することに注意してください。2-17 ページの「JPublisher のデフォルトの型マップとユーザー型マップ」を参照してください。前述の型マップ・エントリをテストするため、デフォルトの型マップをオーバーライドする必要があります。そのためには、次のように、JPublisher の -defaulttypemap オプションを使用します。

```
jpub -u scott/tiger -s SYS.SQLJUTL:SQLJUtl
-defaulttypemap=BOOLEAN:boolean:INTEGER:INT2BOOL:BOOL2INT
```

注意： このマニュアルで SQL 型と PL/SQL 型の間のマッピングに関する変換を説明していますが、このアプローチを PL/SQL 限定とするような制限事項はありません。様々な SQL 型の間でもマップできます。事実、JPublisher のデフォルトの型マップでは、VARCHAR2 値との間でマップされる SQL の INTERVAL 型をサポートするために、このマッピングが行われています。（2-17 ページの「JPublisher のデフォルトの型マップとユーザー型マップ」を参照。）

変換対象となった PL/SQL 型が IN パラメータまたはファンクション戻り値として使用されている場合、それ以上の作業は不要です。SQL と PL/SQL の間での 2 つの変換ファンクションがあれば、このような変換要件をすべて満たすことができます。ただし、PL/SQL 型が OUT または IN OUT パラメータの位置に使用されている場合は、問題が発生します。この場合は、この型を使用するオリジナルのプロシージャまたはファンクションをコールする前または後に、PL/SQL 表現と SQL 表現の間で変換が必要になることがあります。つまり、この追加の変換タスクを実行するために、追加の PL/SQL コードをメソッドごとに生成してロードする操作が必要になる可能性があります。JPublisher では、このコードが自動的に作成されます。ただし、この追加の PL/SQL コードをデータベースにインストールする必要があります。

次の JPublisher オプションを使用すると、この PL/SQL コードの作成方法を制御できます。

- -plsqlfile=*filename*

  このオプションでは、JPublisher によって PL/SQL コードが生成されるファイルの名前を指定します。このファイルが存在している場合は、上書きされます。ファイル名を指定しないと、JPublisher ではファイル plsql_wrapper.sql に書き込まれます。PL/SQL ラッパーをデータベースにインストールするには、この SQL スクリプトを実行する必要があるため注意してください。

- -plsqlpackage=*plsql_package*

  このオプションでは、JPublisher によって PL/SQL コードが生成される PL/SQL パッケージの名前を指定します。パッケージ名を指定しないと、JPublisher では JPUB_PLSQL_WRAPPER が使用されます。

- `-plsqlmap=`*`flag`*

  このオプションでは、PL/SQL ラッパー・プロシージャおよびファンクションの生成方法を指定します。*`flag`* には、次のいずれかを設定できます。

  - `true`（デフォルト）。必要に応じて PL/SQL ラッパー・プロシージャおよびファンクションが生成され、対応できる場合にのみ変換ファンクションが使用されます。
  - `false`。PL/SQL ラッパー・プロシージャまたはファンクションは生成されません。変換ファンクションのみではサポートできないシグネチャ内の PL/SQL 型が検出されると、その特定のプロシージャまたはファンクション用の Java コードは生成されません。
  - `always`。PL/SQL 型を使用するストアド・プロシージャまたはファンクションごとに、PL/SQL ラッパー・プロシージャまたはファンクションが生成されます。この設定は、オリジナルの PL/SQL パッケージを補完するプロキシ PL/SQL パッケージを生成する場合に役立ちます。オリジナル・パッケージ内で JDBC または SQLJ からアクセスできないファンクションまたはプロシージャについては、Java でアクセス可能なシグネチャが提供されます。

PL/SQL RECORD 型の型マッピング・サポート

PL/SQL の RECORD 型の公開は、前述のように変換ファンクションの使用を示す特殊なケースです。必要な手順を示すには、具体例を使用するのが最も簡単な方法です。

次の PL/SQL RECORD 型を使用するメソッドのシグネチャがあるとします。このメソッドは、PL/SQL パッケージ `SCHEM.PACK` に定義されています。

```
TYPE plsql_record IS RECORD (
     pls_number   NUMBER,
     pls_name     VARCHAR2(60));
```

また、変換はスキーマ `SCOTT` 内で発生するとします。

実行する手順を次のリストに示します。

1. `PLSQL_RECORD` のマップ先となる SQL 型を定義します。たとえば、次のようにします。

   ```
   create TYPE sql_record as object (
           sql_number   NUMBER,
           sql_name     VARCHAR2(60));
   ```

2. JPublisher を使用して、この SQL 型を Java に公開します。たとえば、次のように `SQL_RECORD` 用の Java ラッパー・クラス `SqlRecord` を作成できます。

   ```
   jpub -u scott/tiger -s SQL_RECORD:SqlRecord
   ```

3. PLSQL_RECORD と SQL_RECORD の間でマップする PL/SQL ストアド・ファンクションを定義します。

```
function plsql_record2sql(r SCHEM.PACK.PLSQL_RECORD)
         return sql_record is
begin
   return sql_record(r.inst_number, r.inst_name);
end plsql_record2sql;

function sql_record2plsql(r sql_record)
         return SCHEM.PACK.PLSQL_RECORD is
 res SCHEM.PACK.PLSQL_RECORD;
begin
 if r IS NOT NULL
 then
   res.plsql_number := r.sql_number;
   res.plsql_name   := r.sql_name;
 end if;
 return res;
end sql_record2plsql;
```

4. JPublisher に対して、PLSQL_RECORD 型を SQL_RECORD 型にマップすることで公開する方法を指示する型マップ・エントリを設定します。たとえば、次の JPublisher プロパティ・ファイル record.properties を作成できます（円記号 ¥ は、行の継続を示します）。

```
# Type map entries have the format:
# jpub.sql=PLSQL_type:Java_type:SQL_type:sql_to_plsql_fun:plsql_to_sql_fun
#
# Note the use of line continuation in the entry below.
jpub.addtypemap=SCHEM.PACK.PLSQL_RECORD:¥
                SqlRecord:¥
                SQL_RECORD:¥
                SQL_RECORD2PLSQL:¥
                PLSQL_RECORD2SQL
```

5. PLSQL_RECORD を参照するパッケージまたは型を公開するときには、この型マップ・エントリを使用します。たとえば、次の JPublisher 起動コードでは、この型マップ・エントリを使用して record.properties を含めています（-users の短縮形 -u および -props の短縮形 -p を使用します）。

```
jpub -u schema/pw_for_schem -p record.properties -s SCHEM.PACK:Pack
```

6. SCHEM.PACK 内で OUT または IN OUT パラメータとして PLSQL_RECORD が使用されている場合、JPublisher では PL/SQL ラッパー定義を含むファイル plsql_wrapper.sql が生成されたことを示す警告も表示されます。生成された Java クラス Pack を使用する前に、このスクリプトを実行してください。また、-plsqlfile、-plsqlpackage および -plsqlmap オプションを使用すると、

JPublisher で作成される PL/SQL スクリプトをカスタマイズできることに注意してください。

PL/SQL 索引付き表の型の型マッピング・サポート

JDBC OCI ドライバを使用しており、スカラーの索引付き表を公開するのみでよい場合は、Java とこれらの型の間で直接のマッピングを使用できます。2-8 ページの「JDBC OCI を使用したスカラーの索引付き表の型マッピング・サポート」を参照してください。他の場合はいずれも、PL/SQL の索引付き表の型との間で変換できるように、SQL コレクション型を定義する必要があります。

この項では、引き続き前項の例を使用し、PLSQL_RECORD 型の要素を持つ索引付き表の型 PLSQL_INDEXBY を追加します。手順は、前述のものと同じです。ここでも、型宣言がパッケージ SCHEM.PACK に定義されているとします。前述の PLSQL_RECORD の宣言に加えて、PLSQL_INDEXBY も次のように定義されています。

```
TYPE plsql_indexby IS TABLE OF plsql_record INDEX BY BINARY_INTEGER;
```

また、変換はスキーマ SCOTT 内で発生するとします。

実行する手順を次のリストに示します。

1. PLSQL_INDEXBY のマップ先となる SQL 型を定義します。たとえば、次のようにします。

   ```
   create TYPE sql_indexby as table of sql_record;
   ```

   この型の要素は、PLSQL_INDEXBY の要素にマップ可能にする必要があることに注意してください。そのために、あらかじめ SQL_RECORD 型を作成して PLSQL_RECORD にマップしてあります。

2. JPublisher を使用して、この SQL 型を Java に公開します。たとえば、SQL_INDEXBY 用の Java ラッパー・クラス SqlIndexby を作成するには、次のように JPublisher を実行します。

   ```
   jpub -u scott/tiger -s SQL_INDEXBY:SqlIndexby
   ```

3. PLSQL_INDEXBY と SQL_INDEXBY の間でマップする PL/SQL ストアド・ファンクションを定義します。次のファンクションは、以前に定義されている変換ファンクション PLSQL_RECORD2SQL および SQL_RECORD2PLSQL とともに動作します。

   ```
   function plsql_indexby2sql (r SCHEM.PACK.PLSQL_INDEXBY)
            return sql_indexby is
     tab sql_indexby := sql_indexby();
   begin
     FOR i IN 1..r.LAST LOOP
        tab(i) := plsql_record2sql(r(i));
     END LOOP;
     return tab;
   ```

```
end plsql_indexby2sql;

function sql_indexby2plsql (r sql_indexby)
         return SCHEM.PACK.PLSQL_INDEXBY is
  res SCHEM.PACK.PLSQL_INDEXBY;
begin
  FOR i IN 1..r.LAST LOOP
    res(i) := sql_record2plsql(r(i));
  END LOOP;
  return res;
end sql_indexby2plsql;
```

4. JPublisher に対して、PLSQL_INDEXBY 型を SQL_INDEXBY 型にマップすることで公開する方法を指示する型マップ・エントリを設定します。たとえば、次の JPublisher プロパティ・ファイル indexby.properties を作成できます。

```
# Type map entries have the format:
# jpub.sql=PLSQL_type:Java_type:SQL_type:sql_to_plsql_fun:plsql_to_sql_fun
#
# Note the use of line continuation in the entry below.
jpub.addtypemap=SCHEM.PACK.PLSQL_INDEXBY:¥
                SqlIndexby:¥
                SQL_INDEXBY:¥
                SQL_INDEXBY2PLSQL:¥
                PLSQL_INDEXBY2SQL
```

5. PLSQL_INDEXBY を参照するパッケージまたは型を公開するときには、この型マップ・エントリを使用します。たとえば、次の JPublisher 起動コマンドでは、この型マップ・エントリとともに indexby.properties ファイルが組み込まれています（このコマンドは折り返されていますが 1 行のコマンドラインです）。

```
jpub -u schem/pw_for_schem -p indexby.properties -p record.properties
     -s SCHEM.PACK:Pack
```

   JPublisher に対して PLSQL_RECORD エンティティのマッピング方法を指示する record.properties ファイルも組み込まれていることに注意してください。これにより、JPublisher では、PLSQL_RECORD 型または PLSQL_INDEXBY 型（あるいはその両方）を含むシグネチャをマップできます。また、すべての型マップ・エントリを組み合せて 1 つのプロパティ・ファイルにすることもできます。

6. SCHEM.PACK 内で OUT または IN OUT パラメータとして PLSQL_INDEXBY または PLSQL_RECORD が使用されている場合、JPublisher では PL/SQL ラッパー定義を含むファイル plsql_wrapper.sql が生成されたことを示す警告も表示されます。生成された Java クラス Pack を使用する前に、このスクリプトを実行してください。また、-plsqlfile、-plsqlpackage および -plsqlmap オプションを使用すると、JPublisher で作成される PL/SQL スクリプトをカスタマイズできることに注意してください。

JPublisher のデフォルトの型マップとユーザー型マップ

JPublisher には、-typemap および -addtypemap オプションによって制御され、最初は空になっているユーザー型マップと、-defaulttypemap および -adddefaulttypemap オプションによって制御され、次のエントリで始まるデフォルトの型マップがあります。

```
jpub.defaulttypemap=SYS.XMLTYPE:oracle.xdb.XMLType
jpub.adddefaulttypemap=BOOLEAN:boolean:INTEGER:¥
SYS.SQLJUTL.INT2BOOL:SYS.SQLJUTL.BOOL2INT
jpub.adddefaulttypemap=INTERVAL DAY TO SECOND:String:CHAR:¥
SYS.SQLJUTL.CHAR2IDS:SYS.SQLJUTL.IDS2CHAR
jpub.adddefaulttypemap=INTERVAL YEAR TO MONTH:String:CHAR:¥
SYS.SQLJUTL.CHAR2IYM:SYS.SQLJUTL.IYM2CHAR
```

JPublisher では、最初にデフォルトの型マップが読み込まれます。デフォルトの型マップに含まれているマッピングをユーザー型マップで再定義しようとすると、JPublisher では警告メッセージが生成され、再定義は無視されます。同様に、以前のマッピングと競合する -adddefaulttypemap または -addtypemap 設定を介してマッピングを追加しようとすると、無視されて警告が生成されます。

カスタム・マッピングを使用するには、次のようにデフォルトの型マップを消去することをお薦めします。

```
-defaulttypemap=
```

その後、-addtypemap オプションを使用して、必要なマッピングをユーザー型マップに追加します。

事前定義済みのデフォルトの型マップでは、OPAQUE 型 SYS.XMLTYPE と Java ラッパー・クラス oracle.xdb.XMLType の間の対応付けが定義されます。また、PL/SQL の BOOLEAN 型は、SYS.SQLJUTL パッケージに定義されている 2 つの変換ファンクションを介して、Java の boolean および SQL の INTEGER にマップされます。最終的に、デフォルトの型マップは SQL の INTERVAL 型と Java の String 型の間のマッピングを提供します。

ただし、たとえば、true 値と false 値のみでなく SQL の NULL 値を取得するために、PL/SQL の BOOLEAN 型から Java のオブジェクト型 Boolean へのマッピングが必要な場合があります。そのためには、次のようにデフォルトの型マップをリセットします（ここでは折り返されていますが 1 行のコードです）。

```
-defaulttypemap=BOOLEAN:Boolean:INTEGER:SYS.SQLJUTL.INT2BOOL:
SYS.SQLJUTL.BOOL2INT
```

これにより、指定した Java 型を boolean から Boolean に変更できます。変換の残りの部分は引き続き有効です。

JDBC でサポートされないデータ型に対するその他の代替方法

JDBC でサポートされない型にアクセスするために JPublisher で使用されるメカニズムについては前述しました。この方法で JPublisher を使用するかわりに、次のいずれかの代替方法を使用できます。

- その型を使用しないように PL/SQL メソッドを書き直します。
- 次のような無名ブロックを記述します。
  - JDBC でサポートされる入力タイプを、PL/SQL メソッドで使用される入力タイプに変換します。
  - PL/SQL メソッドで使用される出力タイプを、JDBC でサポートされる出力タイプに変換します。

  この方法の詳細は、4-66 ページの「例 : JDBC でサポートされないデータ型の使用」を参照してください。

JPublisher で生成されるクラスの概念

この項では、JPublisher で生成されるコードに関する次の基本的な概念について説明します。

- SQL オブジェクト型メソッドと PL/SQL メソッドの出力パラメータの取扱い
- メンバー・メソッドのコール
- オーバーロードされたメソッドの処理

詳細は、次の各項を参照してください。

- 2-22 ページの「JPublisher による SQLJ クラス（.sqlj）の生成」
- 2-29 ページの「JPublisher による Java クラス（.java）の生成」
- 2-36 ページの「JPublisher による継承のサポート」

OUT パラメータ渡し

SQLJ を介してコールされるストアド・プロシージャでは、通常の Java メソッドと同じパラメータ渡しの動作は行われません。これは、JPublisher で生成されるラッパー・メソッドをコールするときに記述するコードに影響します。

通常の Java メソッドをコールすると、Java オブジェクトであるパラメータはオブジェクト参照として渡されます。そのメソッドでオブジェクトを変更できます。

それに対し、SQLJ を介してストアド・プロシージャをコールすると、各パラメータのコピーがストアド・プロシージャに渡されます。プロシージャでパラメータを変更する場合は、変更されたパラメータのコピーがコール元に戻されます。このため、変更されたパラメータの前および後の値は別々のオブジェクトとなります。

JPublisher で生成されるラッパー・メソッドには、ストアド・プロシージャをコールするための SQLJ コードが含まれます。CREATE TYPE または CREATE PACKAGE 宣言で宣言されたストアド・プロシージャに対するパラメータには、3 つの使用可能なパラメータ・モード、IN、OUT および IN OUT があります。ストアド・プロシージャの IN OUT および OUT パラメータは新規に作成されたオブジェクトのラッパー・メソッドに戻されます。これらの新規の値はコール元に戻す必要がありますが、ラッパー・メソッド内の仮パラメータはコール元で参照できる実際のパラメータには影響を与えません。

this パラメータ以外のパラメータ渡し

前述した問題を解決する場合、単一要素配列のラッパー・メソッドに OUT または IN OUT パラメータを渡すのが最も簡単な方法です。配列とは、パラメータを格納するコンテナの一種です。

- そのパラメータの前の値を配列の要素 0 へ割り当てます。
- 配列をラッパー・メソッドへ渡します。
- ラッパー・メソッドはパラメータの後の値を配列の要素 0 へ割り当てます。
- メソッドを実行した後に配列から後の値を抽出します。

次の例では、クラス Person の初期化変数 p があり、x は IN OUT Person 引数を使用するラッパー・メソッド f を持つ JPublisher で生成されたクラスに属するオブジェクトです。配列を作成し、次のようにパラメータを渡します。

```
Person [] pa = {p};
x.f(pa);
p = pa[0];
```

OUT または IN OUT パラメータを渡すためのこの方法では、それぞれのパラメータごとにユーザー・プログラムのコードに何行か追加する必要があります。ユーザーのストアド・プログラムに多くの OUT または IN OUT パラメータがある場合、ラッパー・メソッドよりも SQLJ コードを直接使用してコールする方がよい場合もあります。

this パラメータ渡し

インスタンスのメソッドの this オブジェクトに変更があると、前述のような問題が発生します。

this オブジェクトは異なる方法で渡される追加パラメータです。CREATE TYPE 文で宣言されるそのモードは IN または IN OUT になります。this のモードを明示的に宣言しない場合のモードは、ストアド・プロシージャが結果を戻さない場合には IN OUT、結果を戻す場合は IN になります。

this オブジェクトのモードが IN OUT の場合、ラッパー・メソッドは this の新規の値を戻す必要があります。JPublisher で生成されたコードでは、状況に応じて this が異なる方法で処理されます。

- 結果を戻さないストアド・プロシージャの場合、this の新規の値がラッパー・メソッドの結果として戻されます。

  たとえば、SQL オブジェクト型 MYTYPE に次のメンバー・プロシージャがあるとします。

  ```
  MEMBER PROCEDURE f1(y IN OUT INTEGER);
  ```

  また、JPublisher で対応する Java クラス MyJavaType が生成されるとします。このクラスでは、次のメソッドが定義されます。

  ```
  public MyJavaType f1(int[] y)
  ```

  f1 メソッドは、変更後の this オブジェクトの値を MyJavaType インスタンスとして戻します。

- ストアド・ファンクション（結果を戻すストアド・プロシージャ）の場合、ラッパー・メソッドは結果としてストアド・ファンクションの結果を戻します。this の新規の値は単一要素の配列で戻され、ラッパー・メソッドに追加の引数（最後の引数）として渡されます。

  SQL オブジェクト型 MYTYPE に次のメンバー関数があるとします。

  ```
  MEMBER FUNCTION f2(x IN INTEGER) RETURNS VARCHAR2;
  ```

  対応する Java クラス MyJavaType で次のメソッドが定義されます。

  ```
  public String f2(int x, MyJavaType[] newValue)
  ```

  f2 メソッドは VARCHAR2 関数の戻り値を Java 文字列として戻し、変更後の this オブジェクトの値を MyJavaType 配列の配列要素として戻します。

注意： PL/SQL の静的プロシージャまたはファンクションの場合、JPublisher ではラッパー・クラスに静的メソッドではなくインスタンスのメソッドが生成されます。これは、データベース接続（SQLJ 接続コンテキスト・インスタンスまたは JDBC 接続インスタンス）を、各ラッパー・クラスのインスタンスに対応付けるための方法です。接続インスタンスはラッパー・クラスのインスタンスの初期化に使用されるため、ラッパー・メソッドをコールするときに、接続または接続コンテキスト・インスタンスを明示的に提供する必要はありません。

オーバーロードされたメソッドの変換

PL/SQL では、Java と同様に、オーバーロードされたメソッド、つまり同じ名前でシグネチャが異なる 2 つ以上のメソッドを作成できます。JPublisher を使用して PL/SQL メソッド用のラッパー・メソッドを生成すると、PL/SQL で異なるシグネチャを持つ 2 つのオーバーロードされたメソッドが、Java で同じシグネチャを持つ場合があります。この場合、JPublisher でメソッドの名前を変更して、同じシグネチャを持つ 2 つ以上のメソッドを生成しないでください。たとえば、次のファンクションを含む PL/SQL パッケージまたはオブジェクト型を想定します。

```
FUNCTION f(x INTEGER, y INTEGER) RETURN INTEGER
```

および

```
FUNCTION f(xx FLOAT, yy FLOAT) RETURN INTEGER
```

PL/SQL では、これらのファンクションに異なる引数型があります。ただし、Oracle マッピングを使用して Java に変換されると、この違いはなくなります（`INTEGER` および `FLOAT` は両方とも `oracle.sql.NUMBER` にマップされます）。

JPublisher でコマンドライン設定 `-methods=true` および Oracle マッピングを使用してパッケージまたはオブジェクト型用のクラスを生成するとします。JPublisher では次のコードと同様のコードを生成して応答します。

```
public oracle.sql.NUMBER f_1 (
  oracle.sql.NUMBER x,
  oracle.sql.NUMBER y)
throws SQLException
{
  /* body omitted */
}

public oracle.sql.NUMBER f_4 (
  oracle.sql.NUMBER xx,
  oracle.sql.NUMBER yy)
throws SQLException
{
```

```
  /* body omitted */
}
```

この例では、JPublisher は最初のファンクションの名前を f_1、2 番目のファンクションの名前を f_4 と指定していることに注意してください。それぞれのファンクション名の最後に _<*nn*> が付いています。<*nn*> は、JPublisher により割り当てられる番号です。数値それ自体には意味がありませんが、JPublisher ではその数値を使用して同一のパラメータ・タイプを持つファンクションの名前を確実に一意にします。

JPublisher による SQLJ クラス（.sqlj）の生成

-methods=all（デフォルト）または -methods=true の場合、JPublisher では PL/SQL パッケージとオブジェクト型用の .sqlj ファイルが生成されます。どちらも、ORAData 実装および SQLData 実装です（オブジェクト型でメソッドが定義されている場合は、.java ファイルが生成されます）。そのファイルには、オブジェクト型およびパッケージのいくつかのサーバー・メソッドをコールするラッパー・メソッドが含まれます。SQLJ を実行し、.sqlj ファイルを変換します。

この項では、ユーザーの SQLJ コードでこれらの生成されたクラスを使用する方法を説明します。

SQLJ クラスの生成に関する重要な注意事項

JPublisher で生成される SQLJ クラスについての注意事項は、次のとおりです。

- JPublisher に用意されているクラスには、release() メソッドが含まれています。JPublisher で生成されるラッパー・クラスのインスタンスを作成して使用する場合に、DefaultContext 引数を持つコンストラクタを使用せず、その後に接続コンテキストの引数を取る setConnectionContext() メソッドをコールせずに、ラッパー・メソッドをコールすると、ラッパー・オブジェクトにより DefaultContext インスタンスが暗黙的に構成されます。この場合は、release() メソッドを使用して、不要になった時点で接続コンテキスト・インスタンスを解放する必要があります。

  つまり、次のいずれかの方法をお薦めします。

  – 接続情報を指定せずに、静的な SQLJ のデフォルトの接続コンテキスト・インスタンスを暗黙的に使用します。

  または

  – setConnectionContext() メソッドを介して、オブジェクトを SQLJ 接続コンテキスト・インスタンスに明示的に対応付けます。

  または

  – 明示的に提供する SQLJ 接続コンテキストを使用してオブジェクトを構成します。

詳細は、2-26 ページの「JPublisher により生成された SQLJ コードでの接続コンテキストおよびインスタンスの使用」を参照してください。

- Oracle8*i* JPublisher と JPublisher の Oracle8*i* 互換モードには、`DefaultContext` インスタンスまたはユーザー指定のクラス・インスタンスを取るコンストラクタのかわりに、単に `ConnectionContext` インスタンス（標準 `sqlj.runtime.ConnectionContext` インタフェースを実装するクラスのインスタンス）を取るコンストラクタがあります。

JPublisher で PL/SQL パッケージ用に生成される SQLJ クラスの使用

JPublisher で PL/SQL パッケージ用に生成されるクラスを使用する場合は、次の手順を実行します。

1. クラスのインスタンスを構成します。
2. クラスのラッパー・メソッドをコールします。

クラスのコンストラクタは、データベース接続をクラスのインスタンスに対応付けます。SQLJ `DefaultContext` インスタンス（または、JPublisher の実行時に `-context` オプションで指定したクラスのインスタンス）を取るコンストラクタ、JDBC `Connection` インスタンスを取るコンストラクタ、引数を取らないコンストラクタがあります。引数のないコンストラクタをコールするのは、`DefaultContext` インスタンスを取るコンストラクタに SQLJ のデフォルト・コンテキストを渡すのと同じです。Oracle JDBC には、SQLJ プログラムのコンパイル方法は理解しているが、`DefaultContext` などの SQLJ 概念をよく理解していない JDBC プログラマが使用しやすいように、`Connection` インスタンスを使用するコンストラクタが用意されています。

重要： 2-22 ページの「SQLJ クラスの生成に関する重要な注意事項」を参照してください。

`this` オブジェクトの接続コンテキストがラッパー・メソッドの `#sql` 文で使用されているため、ラッパー・メソッドはすべてインスタンス・メソッドになります。

PL/SQL パッケージ用に生成されたクラスには、接続コンテキスト以外のインスタンス・データがないため、通常は使用する接続コンテキストごとにクラス・インスタンスを 1 つ構成します。デフォルトのコンテキストが、使用する唯一のコンテキストの場合は、引数のないコンストラクタを一度コールできます。ただし、かわりに明示的な接続コンテキスト・インスタンスを使用する理由については、『Oracle9*i* SQLJ 開発者ガイドおよびリファレンス』を参照してください。

PL/SQL パッケージ用に生成されたクラスのインスタンスには、PL/SQL パッケージ変数のコピーは含まれていません。そのクラスのインスタンスは `ORAData` クラスでも `SQLData` クラスでもなく、ホスト変数としても使用できません。

PL/SQL パッケージ用に生成されたクラスを使用する方法は、4-61 ページの「例 : パッケージ用に生成されたクラスの使用」を参照してください。

JPublisher でオブジェクト型用に生成されるクラスの使用

JPublisher で SQL オブジェクト型または SQL OPAQUE 型用に生成された Java クラスのインスタンスを使用するには、最初に Java オブジェクトを初期化する必要があります。そのためには、次の方法があります。

- Java オブジェクトに対して初期化済みの Java オブジェクトを割り当てます。

または

- SQL オブジェクトのコピーを次の方法で Java オブジェクトに取り込みます。そのためには、SQL オブジェクトを `OUT` 引数または JPublisher によって生成されたラッパー・メソッドのファンクション・コールの戻り値として使用する方法、ユーザー記述の `#sql` 文を介して SQL オブジェクトを取り出す方法、ユーザー記述の JDBC コールを介して SQL オブジェクトを取り出す方法があります。

または

- 引数のないコンストラクタで Java オブジェクトを構成し、`setXXX()` メソッドを使用してその属性を設定するか、すべてのオブジェクト属性の値を受け入れるコンストラクタで Java オブジェクトを構成します。通常は、`setConnection()` または `setConnectionContext()` メソッドを使用して、ラッパー・メソッドを起動する前にオブジェクトをデータベース接続に対応付けます。オブジェクトを JDBC または SQLJ 接続に明示的に対応付けずに、そのメソッドを起動すると、メソッドはデフォルトの（静的）SQLJ 接続コンテキストに暗黙的に対応付けられます。

  クラスの他のコンストラクタは接続をクラス・インスタンスに対応付けます。`DefaultContext` インスタンス（または、JPublisher の実行時に `-context` オプションを通じて指定したクラスのインスタンス）を取るコンストラクタと、`Connection` インスタンスを取るコンストラクタがあります。SQLJ プログラムのコンパイル方法は理解しているが、`DefaultContext` などの SQLJ 概念をよく理解していない JDBC プログラマが使用しやすいように、`Connection` インスタンスを使用するコンストラクタが用意されています。

重要 : 2-22 ページの「SQLJ クラスの生成に関する重要な注意事項」を参照してください。

Java オブジェクトを初期化すると、次の作業を実行できます。

- オブジェクトのアクセッサ・メソッドをコールできます。
- オブジェクトのラッパー・メソッドをコールできます。
- オブジェクトを他のラッパー・メソッドに渡すことができます。

- #sql 文でオブジェクトをホスト変数として使用できます。
- JDBC コールでオブジェクトをホスト変数として使用できます。

対応する SQL オブジェクト型の属性ごとに Java 属性があり、属性ごとに get*XXX*() および set*XXX*() アクセッサ・メソッドがあります。アクセッサ・メソッドの名前は、属性 foo の場合は getFoo() および setFoo() という形式になります。JPublisher では属性に対してフィールドは生成されません。

クラスにはデフォルトで、サーバーで実行される対応付けられた Oracle オブジェクト・メソッドをコールするラッパー・メソッドが含まれます。ラッパー・メソッドは、サーバー・メソッドがインスタンス・メソッドかどうかに関係なく、すべてインスタンス・メソッドです。this オブジェクト内の DefaultContext は、ラッパー・メソッドの #sql 文で使用されます。

Oracle マッピングを使用すると、JPublisher では Oracle JDBC ドライバ用に次のメソッドが生成されます。この種のメソッドは、ORAData および ORADataFactory インタフェースで次のように指定されます。

- create()
- toDatum()

この 2 つのメソッドは、通常は直接使用することを意図していません。また、JPublisher では、メソッド setFrom(*otherObject*)、setValueFrom(*otherObject*) および setContextFrom(*otherObject*) が生成されます。これらのメソッドを使用すると、オブジェクト・インスタンス間で値や接続情報をコピーできます。

オブジェクト型用に生成され、ラッパー・メソッドを持つクラスの使用方法は、4-50 ページの「例 : オブジェクト型用に生成されたクラスの使用」の例を参照してください。

JPublisher により生成された SQLJ コードでの接続コンテキストおよびインスタンスの使用

JPublisher で SQLJ 接続コンテキスト・インスタンスの作成に使用されるクラスは、JPublisher の実行時に -context オプションに設定する値に応じて次のように異なります。

- -context=DefaultContext（デフォルト設定）に設定すると、JPublisher では標準 sqlj.runtime.ref.DefaultContext クラスのインスタンスが使用されます。
- ユーザー指定のクラス（つまり、CLASSPATH に指定され、標準 sqlj.runtime.ConnectionContext インタフェースを実装するクラス）を設定すると、JPublisher ではそのクラスのインタフェースが使用されます。
- -context=generated に設定すると、JPublisher で生成されたクラスで次のように宣言されます。

  ```
  #sql static context _Ctx;
  ```

  この場合、JPublisher では、接続コンテキスト・インスタンスに _Ctx クラスのインスタンスが使用されます。

注意： Oracle8*i* JPublisher とは異なり、JPublisher で接続コンテキスト・インスタンス _ctx を宣言するためのルーチンはなくなりました。このルーチンが使用されるのは、Oracle8*i* 互換モード（-compatible=8i または -compatible=both8i）で、_ctx が静的接続コンテキストのクラス _Ctx の protected インスタンスとして宣言されている場合です。

_ctx に依存する従来型コードがない場合に、JPublisher で生成されたクラスで接続コンテキスト・インスタンスを取得して操作するには、getConnectionContext() および setConnectionContext() メソッドを使用してください。この 2 つのメソッドの詳細は、「接続コンテキストおよび接続インスタンスの使用時の考慮点」を参照してください。

-context オプションの詳細は、3-15 ページの「SQLJ 接続コンテキスト・クラス（-context）」を参照してください。

接続コンテキストおよび接続インスタンスの使用時の考慮点

JPublisher で生成されたラッパー・クラスのインスタンス内で、SQLJ 接続コンテキスト・インスタンスまたは JDBC 接続インスタンスを使用する場合は、次のことを考慮してください。

- JPublisher で生成されたラッパー・クラスには、SQLJ 接続コンテキスト・インスタンスを明示的に指定できるように、setConnectionContext() メソッドが用意されています。（コンストラクタを通じて接続コンテキスト・インスタンスをすでに指定している場合、これは不要です。）

このメソッドの定義は次のとおりです。

```
public void setConnectionContext(conn_ctxt_instance);
```

これにより、SQLJ 接続コンテキストとして渡された接続コンテキスト・インスタンスが、オブジェクトのラッパー・インスタンスにインストールされます。この接続コンテキスト・インスタンスは、JPublisher の接続コンテキスト（通常は `DefaultContext`）に対して `-context` オプションを通じて指定したクラスのインスタンスであることが必要です。

基礎となる JDBC 接続には、データベース・オブジェクトを最初にインスタンス化するときに使用される接続との互換性が必要であることに注意してください。特に、一部のオブジェクトは、オブジェクト参照型や BLOB など、特定の接続にのみ有効な属性を持つ場合があります。

注意： `setConnectionContext()` メソッドを使用して接続コンテキスト・インスタンスを明示的に設定すると、接続コンテキストが正常にクローズされないという問題を回避できます。この問題が発生するのは、暗黙的に作成された接続コンテキスト・インスタンスの場合のみです。

- 該当する場合は、オブジェクト・ラッパー・インスタンスの次のいずれかのメソッドを使用して、接続または接続コンテキスト・インスタンスを取得します。
  - `Connection getConnection()`
  - *`ConnCtxtType`* `getConnectionContext()`

`getConnectionContext()` メソッドは、JPublisher の `-context` オプションで指定された接続コンテキスト・クラス（通常は `DefaultContext`）のインスタンスを戻します。

戻される接続コンテキスト・インスタンスは、`setConnectionContext()` メソッドを通じて明示的に設定されたインスタンス、あるいは JPublisher により暗黙的に作成されたインスタンスです。

注意： これらのメソッドを使用できるのは、生成された `.sqlj` ファイル内のみで、生成された `.java` ファイルでは使用できません。必要な場合は、確実に `.sqlj` ファイルが生成されるように `-methods=always` に設定できます。3-20 ページの「パッケージのクラスおよびラッパー・メソッドの生成（-methods）」を参照してください。

- JPublisher で生成されたクラス内のコードで SQLJ 文が使用される場合に、接続コンテキスト・インスタンスを明示的に設定しなければ、`getConnectionContext()` メソッドのコール時に JDBC 接続インスタンスから暗黙的に作成されます。

  この場合は、`release()` メソッドを慎重に使用して、SQLJ ランタイム内で他の場合にはメモリー・リークを発生させるリソースを解放する必要があります。

- 生成された異なるクラスに様々な接続コンテキスト・クラスがある場合は、SQLJ のオンライン・セマンティクス・チェック中に様々なサンプル・スキーマと対照して各クラスをチェックするかどうかをオプションで選択できます。ただし、SQLJ ソースは実際の SQL 型から構成されているため、通常、このチェックは不要です。

関連情報は、次の「接続コンテキストのリソースの解放」および 3-15 ページの「SQLJ 接続コンテキスト・クラス（-context）」を参照してください。

接続コンテキストのリソースの解放

SQLJ ランタイム接続コンテキストのリソースを解放するために、JPublisher で生成されたラッパー・クラスのインスタンスの `release()` メソッドの使用が必要になる場合があります。これに該当するのは、次のような状況です。

- SQLJ クラスの変換に SQLJ 設定 `-codegen=iso` を使用した場合。
- 生成された 1 つ以上のクラスの実行時に、JDK 1.1.x または SQLJ の汎用 `runtime` ライブラリ（`runtime12` や `runtime11` などではなく）を使用する場合。
- `DefaultContext`（または JPublisher の実行時に `-context` オプションで指定した他の接続コンテキスト・クラス）のインスタンスを取るコンストラクタを使用してオブジェクトを作成していない場合。
- ラッパー・インスタンスで 1 つ以上のラッパー・メソッドをコールした場合。
- ラッパー・インスタンスの `setConnectionContext()` メソッドを使用して接続コンテキスト・インスタンスを明示的に設定していない場合。

このような場合、接続コンテキスト・インスタンスはオブジェクトに対して暗黙的に作成されており、そのオブジェクトが有効範囲外になる前に、`release()` メソッドを通じて明示的に解放する必要があります。

（明示的なコンストラクタまたは `setConnectionContext()` メソッドなどを使用して作成された明示的な接続コンテキスト・インスタンスがある場合は、`release()` を使用する必要はありません。）

JPublisher による Java クラス（.java）の生成

-methods=false の場合、または SQL オブジェクト型でメソッドが定義されていない場合、JPublisher ではオブジェクト型のラッパー・メソッドは生成されません。この動作は、ORAdata 実装と SQLData 実装に共通です。また、-methods=false の場合、ラッパー・メソッドがなければ役に立たないため、PL/SQL パッケージのコードも生成されません。（-methods=false の場合、JPublisher では .java ファイルのみが生成されることに注意してください。）

JPublisher では、-methods が FALSE でも TRUE でも、参照型、VARRAY 型およびネストした表型用に同じ Java コードが生成されます。

-methods=false のときのオブジェクト型用や、参照型、VARRAY 型またはネストした表型用に JPublisher で生成されるクラスのインスタンスを使用するには、最初にオブジェクトを初期化する必要があります。

JPublisher で生成される SQLJ クラスの場合と同様に、次のいずれかの方法でオブジェクトを初期化できます。

- Java オブジェクトに対して初期化済みの Java オブジェクトを割り当てます。

または

- SQL オブジェクトのコピーを次の方法で Java オブジェクトに取り込みます。そのためには、SQL オブジェクトを OUT 引数または JPublisher で生成された他のクラスのラッパー・メソッドの、ファンクション・コールの戻り値として使用する方法、ユーザー記述の #sql 文を介して SQL オブジェクトを取り出す方法、ユーザー記述の JDBC コールを介して SQL オブジェクトを取り出す方法があります。

または

- 引数のないコンストラクタで Java オブジェクトを構成して、そのデータを初期化するか、属性値に基づいて Java オブジェクトを構成します。

.sqlj ソース・ファイルに生成されたコンストラクタとは異なり、.java ソース・ファイルに生成されたコンストラクタは接続引数を使用しません。かわりに、オブジェクトが Statement、CallableStatement または PreparedStatement オブジェクトに渡されるか、あるいは戻されると、JPublisher では、Statement、CallableStatement または PreparedStatement オブジェクトの構成に使用する接続が適用されます。

これは、同じオブジェクトを様々な時間に様々な接続で使用できるという意味ではありません。また、常に使用可能とは限りません。オブジェクトは、特定の接続に対してのみ有効な参照または BLOB などのサブコンポーネントを持つ場合があります。

オブジェクトのデータを初期化するには、クラスがオブジェクト型を表す場合は set*XXX*() メソッドを使用し、クラスが VARRAY またはネストした表型を表す場合は setArray() または setElement() メソッドを使用します。クラスが参照型を表す場合に可能なのは、null 参照を構成することのみです。非 null の参照は、すべてデータベースから取り込まれます。

オブジェクトを初期化すると、次を実行できます。

- オブジェクトを他のクラスのラッパー・メソッドに渡すことができます。
- #sql 文でオブジェクトをホスト変数として使用できます。
- JDBC コールでオブジェクトをホスト変数として使用できます。
- オブジェクトの状態を読取り / 書込みを行うメソッドをコールできます。これらのメソッドはユーザーのプログラム内の Java オブジェクトで動作し、データベース内のデータには影響しません。
  - オブジェクト型を表すクラスの場合は、get*XXX*() および set*XXX*() アクセッサ・メソッドをコールできます。
  - VARRAY またはネストした表を表すクラスの場合、getArray()、setArray()、getElement() および setElement() メソッドをコールできます。

    getArray() および setArray() メソッドは配列を全体として戻すか、あるいは変更します。getElement() および setElement() メソッドは配列の個別の要素を戻すか、あるいは変更します。その後、データベースでデータを更新する場合は、Java 配列をデータベースに再挿入します。
- オブジェクト参照は不変なエンティティであるため変更できませんが、getValue() および setValue() メソッドを使用して、参照する SQL オブジェクトの読取り / 書込みができます。

  getValue() メソッドは REF が参照する SQL オブジェクトのコピーを戻します。setValue() メソッドはデータベース内の SQL オブジェクト型インスタンスを更新し、オブジェクト型を表す Java クラスのインスタンスの入力として使用します。オブジェクト型用に生成されたクラスの get*XXX*() および set*XXX*() アクセッサ・メソッドとは異なり、getValue() および setValue() メソッドで SQL オブジェクトの読取り / 書込みを行います。

  getValue() および setValue() メソッドでは、それぞれ参照先である基礎となるデータベース・オブジェクトの値の読取りと書込みのために、データベースとのラウンドトリップが発生することに注意してください。

まだ説明していないメソッドがあります。JDBC コードで getORADataFactory() メソッドを使用すると、ORADataFactory オブジェクトを戻すことができます。この ORADataFactory を、oracle.jdbc パッケージのクラス ArrayDataResultSet、OracleCallableStatement および OracleResultSet 内で Oracle getORAData() メソッドに渡すことができます。Oracle JDBC ドライバでは、ORADataFactory オブジェクトを使用して、JPublisher で生成されたクラスのオブジェクトが作成されます。

さらに、VARRAY およびネストした表を表すクラスには、oracle.sql.ARRAY クラスの機能を実装する次のいくつかのメソッドがあります。

- getBaseTypeName()
- getBaseType()

- `getDescriptor()`

ただし、VARRAY およびネストした表用に JPublisher で生成されたクラスは `oracle.sql.ARRAY` を拡張しません。

Oracle マッピングを使用すると、JPublisher では Oracle JDBC ドライバ用に次のメソッドが生成されます。この種のメソッドは、`ORAData` および `ORADataFactory` インタフェースで次のように指定されます。

- `create()`
- `toDatum()`

この 2 つのメソッドは、通常は直接使用することを意図していません。ただし、2 つのオブジェクト参照ラッパー型の間で変換する場合は使用できます。

4-61 ページの「例 : パッケージ用に生成されたクラスの使用」に示す例には、ラッパー・メソッドを持たないオブジェクト型用に生成されたクラスが含まれます。

JPublisher で生成されたクラスのユーザー作成サブクラス

JPublisher で生成されたカスタム Java クラスの機能を、メソッドおよび一時フィールドを追加して拡張できます。

その方法の 1 つは、JPublisher で生成されたクラスにメソッドを直接追加することです。ただし、将来 JPublisher を実行してクラスを再生成することになると思われる場合、この方法は望ましくありません。この方法で変更されたクラスを再生成すると、変更内容（追加したメソッド）が上書きされます。JPublisher 出力を別のファイルに送る場合でも、変更内容をそのファイルにマージする必要があります。

生成されたクラスの機能を拡張する場合に望ましい方法は、クラスを拡張することです。つまり、JPublisher で生成されたクラスをスーパークラスとして扱い、その機能を拡張するサブクラスを記述してから、オブジェクト型をサブクラスにマップします。（これは、オブジェクト指向の用語では Generation Gap パターンと呼ばれます。）

この項では、その方法について説明します。

JPublisher で生成されたクラスの拡張

たとえば、JPublisher で SQL オブジェクト型 ADDRESS からクラス JAddress を生成するとします。また、クラス MyAddress を記述して ADDRESS オブジェクトを表すと想定します。この MyAddress は JAddress が提供する機能を拡張します。

この使用例では、JPublisher を使用してカスタム Java クラス JAddress とサブクラス MyAddress の初期バージョンを生成し、この初期バージョンに必要な機能を追加できます。次に JPublisher を使用して ADDRESS オブジェクトを JAddress クラスのかわりに MyAddress クラスにマップします。

これを実行するために、JPublisher で生成されるコードが次の方法で変更されます。

- JAddressRef ではなく参照クラス MyAddressRef を生成します。
- SQL 型が ADDRESS の属性を表すため、あるいは SQL 型が ADDRESS の VARRAY およびネストした表の要素を表すために、JAddress クラスではなく MyAddress クラスを使用します。
- ORADataFactory インタフェースを使用して SQL 型が ADDRESS である Java オブジェクトを構成する場合に、JAddress ファクトリではなく MyAddress ファクトリを使用します。
- JAddress クラス用のコードを生成または再生成します。また、MyAddress クラス用のコードの初期バージョンも生成されます。この初期バージョンを変更して、独自の追加機能を挿入できます。ただし、MyAddress クラスのソース・ファイルが存在する場合は、それがそのまま残ります。

代替クラスへのマッピングの構文

JPublisher は代替クラスに対するマッピング処理を効率化する機能を備えています。-sql コマンドライン・オプション設定で次の構文を使用します。

```
-sql=object_type:generated_class:map_class
```

前述の使用例の場合は、次のようになります。

```
-sql=ADDRESS:JAddress:MyAddress
```

-sql オプションの詳細は、3-25 ページの「変換対象のオブジェクト型およびパッケージの宣言（-sql）」を参照してください。

コマンドラインではなく INPUT ファイルに行を入力すると、次のようになります。

```
SQL ADDRESS GENERATE JAddress AS MyAddress
```

INPUT ファイルの詳細は、3-32 ページの「INPUT ファイルの構造および構文」を参照してください。

この構文では、JAddress は JPublisher で生成されるクラスの名前（通常は JAddress.sqlj）を示しますが、MyAddress は実際に ADDRESS にマップするクラスの名

前を指定します。最終的には、MyAddress 内のコードを取り扱う必要があります。カスタム機能を追加するには、このコードを必要に応じて更新します。ADDRESS 属性を持つオブジェクトを取り出すと、この属性は Java で MyAddress のインスタンスとして作成されます。あるいは ADDRESS オブジェクトを直接取り出すと、MyAddress のインスタンスに取り込まれます。

JPublisher を使用して JAddress クラスを生成する方法の例は、4-26 ページの「例 : SQLData クラスの生成」を参照してください。

生成クラスを拡張するクラスの書式化

MyAddress.sqlj など、カスタム・コードを挿入するソース・ファイルの初期バージョンが存在しない場合は、JPublisher によって自動的に生成されます。

生成コードの機能は、次のとおりです。

- クラスには引数のないコンストラクタがあります。適切な初期化オブジェクトを構成する場合は、スーパークラスのコンストラクタを明示的または暗黙的にコールするのが最も簡単な方法です。
- クラスは ORAData インタフェースまたは SQLData インタフェースを実装します。この実装は、スーパークラスから必要なメソッドを継承することで暗黙的に行われます。
- ORAData クラスを拡張すると、サブクラスも ORADataFactory インタフェースを実装します。

  ORADataFactory create() メソッドの実装は、次のようになります。

  ```
  public ORAData create(Datum d, int sqlType) throws SQLException
  {
     return create(new UserClass(),d,sqlType);
  }
  ```

  ただし、クラスが継承階層に含まれている場合、生成されたメソッドは前述の create() と同じシグネチャおよび本体を持つ protected ORAData createExact() に変わります。

  次のコードは、より効率的な実装を示しています。この場合は、UserClass(boolean) コンストラクタを通じて、初期化済みの UserClass インスタンスが作成されます。このコンストラクタは、UserClass が拡張するスーパークラスを含め、JPublisher で生成されるコードにあります。このコンストラクタを使用すると、UserClass インスタンスは、データ・オブジェクトが null の場合に不必要に作成されたり、データ・オブジェクトが非 null の場合に不必要に再初期化されることはありません。

  ```
  protected UserClass(boolean init) { super(boolean); }
  public ORAData create(Datum d, int sqlType) throws SQLException
  {
     return (d==null) ? null : create(new UserClass(false),d,sqlType);
  }
  ```

Oracle9*i* JPublisher で生成されたクラスのユーザー作成サブクラスでの変更

Oracle8*i* JPublisher で、JPublisher で生成されたクラスにユーザー作成のサブクラスを提供していた場合は、Oracle9*i* JPublisher ではコードの生成方法に関連して多数の変更があることに注意する必要があります。Oracle8*i* の機能に対して作成したアプリケーションを Oracle9*i* で使用する場合は、そのアプリケーションに変更を加える必要があります。

注意： `-compatible=both8i` または `8i` オプション設定を使用すると、ここで説明する変更を意識することなく、アプリケーションを従来どおりに使用できます。3-8 ページの「ユーザー定義型用の下位互換性を持つ Oracle マッピング（-compatible）」を参照してください。

ただし、通常は、ユーザー・コードを JPublisher で生成されたクラスの実装の詳細から切り離す上で役立つため、Oracle9*i* JPublisher の機能への変換を行うことをお薦めします。

変更点は次のとおりです。

- 宣言された `_ctx` 接続コンテキスト・フィールドではなく、用意された `getConnectionContext()` メソッドが使用されます。`_ctx` フィールドは、Oracle9*i* ではサポートされなくなりました。
- `create()` メソッドが明示的に実装されるのではなく、スーパークラスの `create()` メソッドがコールされます。

  次の例では、`UserClass` が `BaseClass` を拡張するとします。`UserClass` に次のメソッドを記述するかわりに、

```
public CustomDatum create(Datum d, int sqlType) throws SQLException
{
  if (d == null) return null;
  UserClass o = new UserClass();
  o._struct = new MutableStruct((STRUCT) d, _sqlType, _factory);
  o._ctx = new _Ctx(((STRUCT) d).getConnection());
  return o;
}
```

  次のように指定します。

```
public CustomDatum create(Datum d, int sqlType) throws SQLException
{
  return create(new UserClass(),d,sqlType);
}
```

  または、クラスが継承階層に含まれる場合は、次のように記述します。

```
protected CustomDatum createExact(Datum d, int sqlType) throws SQLException
{
```

```
  return create(new UserClass(),d,sqlType);
}
```

また、JPublisher では、.sqlj ファイル内に、オブジェクトの初期化が必要かどうかを指定するブール引数を持つ protected コンストラクタが生成されます。

```
protected BaseClass(boolean init) { ... }
```

これを使用すると、UserClass コードを最適化できます。2-33 ページの「生成クラスを拡張するクラスの書式化」を参照してください。

- Oracle9*i* JPublisher には、getConnectionContext() メソッドに加えて、オブジェクトに対応付けられた JDBC 接続の取得に使用できる getConnection() メソッドが用意されています。

setFrom()、setValueFrom() および setContextFrom() メソッド

Oracle9*i* JPublisher では、生成される .sqlj ファイルに次のユーティリティ・メソッドが用意されています。

- setFrom(*anotherObject*)

  このメソッドは、接続および接続コンテキスト情報など、同じベース型の別のオブジェクトからコールするオブジェクトを初期化します。コール側のオブジェクトで暗黙的に作成された既存の接続コンテキスト・オブジェクトが解放されます。

- setValueFrom(*anotherObject*)

  このメソッドは、同じベース型の別のオブジェクトからコールするオブジェクトの基礎となるフィールド値を初期化します。このメソッドは、接続または接続コンテキスト情報を転送しません。

- setContextFrom(*anotherObject*)

  このメソッドは、同じベース型の別のオブジェクトの接続設定からコールするオブジェクトの、接続および接続コンテキスト情報を初期化します。コール側のオブジェクトで暗黙的に作成された既存の接続コンテキスト・オブジェクトが解放されます。このメソッドは、オブジェクト値に関連する情報を転送しません。

次のセマンティックと

```
x.setFrom(y);
```

次のセマンティックは同じです。

```
x.setValueFrom(y);
x.setContextFrom(y);
```

JPublisher による継承のサポート

この項では、主に ORAData 型の継承のサポートについて説明します。この項の内容は、次のとおりです。

- JPublisher での継承サポートの実装方法。
- サブタイプの参照クラスではベース型の参照クラスが拡張されない理由、およびある参照型から別の参照型（通常はサブクラスまたはスーパークラス）への変換方法。

これらの項目について説明した後、SQLData 型の標準的な継承のサポートの概要と、詳細情報の参照先について説明します。

ORAData のオブジェクト型および継承

次の SQL オブジェクト型について考えます。

```
CREATE TYPE PERSON AS OBJECT (
...
) NOT FINAL;

CREATE TYPE STUDENT UNDER PERSON (
...
);

CREATE TYPE INSTRUCTOR UNDER PERSON (
...
);
```

また、対応する Java クラスを作成する次の JPublisher コマンドラインについて考えます（ここでは折り返されていますが、1 つのコマンドです）。

```
jpub -user=scott/tiger -sql=PERSON:Person,STUDENT:Student,INSTRUCTOR:Instructor
-usertypes=oracle
```

この例では、JPublisher により Person クラス、Student クラスおよび Instructor クラスが生成されます。STUDENT と INSTRUCTOR は PERSON のサブタイプであるため、Student クラスと Instructor クラスは Person クラスを拡張します。

継承階層のルートにあるクラス（この例では Person）には、継承階層全体のすべての情報が含まれ、その型マップは必要な情報で自動的に初期化されます。JPublisher でクラス階層に必要なクラスをすべて生成すれば、追加のアクションなしでクラス階層の型マップを適切に移入できます。

部分的に生成された型階層を組み合せる場合の事前作業

SQL 型階層に対して JPublisher を数回実行し、そのたびに対応する Java ラッパー・クラスの一部のみを生成する場合は、クラス階層のルートにある型マップが適切に初期化されるように、ユーザー・アプリケーションで事前作業を実行する必要があります。

前述の例で、次の JPublisher コマンドラインを実行したとします。

```
jpub -user=scott/tiger -sql=PERSON:Person,STUDENT:Student -usertypes=oracle
jpub -user=scott/tiger -sql=PERSON:Person,INSTRUCTOR:Instructor -usertypes=oracle
```

この場合は、これらのマップされた型をコードで使用する前に、生成されたクラス（少なくともリーフ・クラス）のインスタンスを作成する必要があります。たとえば、次のようにします。

```
new Instructor(); // required
new Student();    // required
new Person();     // optional
```

この作業が必要な理由を次に示します。

JPublisher で生成されたコードでの型階層のマッピング

`Person` クラスには次のメソッドが含まれます。

```
Person create(oracle.sql.Datum d, int sqlType)
```

このメソッドは、`Datum` インスタンスをカスタム Java オブジェクトとしての表現に変換するもので、`PERSON` として宣言された SQL オブジェクトが `Person` 変数に取り出されるたびに、Oracle JDBC ドライバによりコールされます。ただし、SQL オブジェクトが実際には `STUDENT` オブジェクトである場合があります。この場合、`create()` メソッドでは `Person` インスタンスではなく `Student` インスタンスを作成する必要があります。

通常、この種の状況を処理するには、カスタム Java クラス（クラスが JPublisher で作成されたかどうかに関係なく）の `create()` メソッドで、`oracle.sql.Datum` 引数に対応する SQL オブジェクト型のサブタイプを表すサブクラスのインスタンスを作成できる必要があります。これにより、作成された Java オブジェクトの実際の型が、SQL オブジェクトの実際の型と一致することが保証されます。

カスタム Java クラス階層のルート・クラスにある `create()` メソッドのコードでは、すべてのサブクラスを指定する必要があると考えることができます。ただし、この場合は、新規サブクラスを記述または生成するときに、ベース・クラス用のコードを変更する必要があります。常に JPublisher を使用してクラス階層全体を再生成する場合、このコード変更は自動的に行われますが、そうでない場合があります。たとえば、拡張する Java クラスのソース・コードへのアクセス権がない場合があります。

JPublisher で生成されたコードは、カスタムの Java クラス階層の各サブクラスに静的な初期化ブロックを作成することで、クラス階層の増分拡張を行うことができます。この静的な初期化ブロックでは、ルート・クラスにはサブクラスに関して必要な情報が与えられ、ルー

ト・レベルの Java クラスで宣言された（型マップと等価の）データ構造が初期化されます。サブクラスのインスタンスが実行時に作成されると、その型がデータ構造に登録されます。この暗黙的なマッピング・メカニズムのため、SQLData の使用例とは異なり、明示的な型マップは不要です。

重要： この実装により、既存のクラスを修正せずに拡張できますが、デメリットもあります。つまり、クラス階層を使用してデータベースからオブジェクトを読み取る前に、サブクラスの静的初期化ブロックが実行される必要があります。これが発生するのは、new() をコールして各サブクラスのオブジェクトをインスタンス化する場合です。サブクラスのコンストラクタにより、すぐ上位のスーパークラスのコンストラクタがコールされるため、リーフ・クラスをインスタンス化するのみです。

かわりに、常にクラス階層全体を生成（または再生成）できます。この場合、すべてのリーフ・クラスのインスタンスを作成することで型マップをインスタンス化する必要はありません。

JPublisher で生成されたコードによる継承のサポートを理解するために、この項の冒頭に示したような例を使用して、生成されたコードを調べてください。

ORAData の参照型および継承

この項では、カスタム参照クラス間での変換方法と、JPublisher によってサブタイプ用に生成されるカスタム参照クラスがベース型の参照クラスを拡張しない理由について説明します。

別の参照型への参照型のインスタンスのキャスト

2-36 ページの「ORAData のオブジェクト型および継承」の例では、基礎となるオブジェクト型のラッパーに加えて、強い型指定の参照用に PersonRef、StudentRef および InstructorRef も取得しています。

StudentRef インスタンスはあるが、それを PersonRef インスタンスの必要なコンテキスト内で使用することが必要になる場合があります。この場合は、強い型指定の参照クラスに生成される静的 cast() メソッドを使用します。

```
StudentRef s_ref = ...;  PersonRef p_ref = PersonRef.cast(s_ref);
```

逆に、PersonRef インスタンスがあり、それを InstructorRef インスタンスへと限定できることがわかっている場合があります。

```
PersonRef pr = ...; InstructorRef ir = InstructorRef.cast(pr);
```

次の項では、オブジェクト型階層をミラー化する参照型階層を設定できるようにするのみでなく、cast() ファンクションを使用する必要がある理由について説明します。

参照型の継承がオブジェクト型の継承に従わない理由

ここでは、参照型を関連オブジェクト型の階層に従わせるのが望ましくない理由を、例をあげて説明します。

わかりやすいように、前項と同じ例のサブセットについて考えます。

```
CREATE TYPE PERSON AS OBJECT (
...
) NOT FINAL;

CREATE TYPE STUDENT UNDER PERSON (
...
);

jpub -user=scott/tiger -sql=PERSON:Person,STUDENT:Student -usertypes=oracle
```

JPublisher では、Person.sqlj（または .java）および Student.sqlj（または .java）が生成されるのみでなく、PersonRef.java および StudentRef.java が生成されます。

Student クラスは Person クラスを拡張するため、StudentRef で PersonRef が拡張されると予期できます。ただし、StudentRef クラスは独立クラスとして PersonRef のサブタイプより強いコンパイル時の型保証を提供できるため、これには該当しません。また、PersonRef ではデータベース内の Person オブジェクトを変更できますが、これは StudentRef ではできません。

PersonRef クラスの最も重要なメソッドは次のとおりです。

- Person getValue()
- void setValue(Person c)

StudentRef クラスで対応するメソッドは次のとおりです。

- Student getValue()
- void setValue(Student c)

StudentRef クラスが PersonRef クラスを拡張すると、次の 2 つの問題が発生します。

- Java では、PersonRef クラス内でオーバーライドするメソッドが Person オブジェクトを戻す場合、StudentRef の getValue() メソッドが Student オブジェクトを戻すことは、たとえ適切であっても許可されません。
- StudentRef の setValue() メソッドは PersonRef の setValue() メソッドをオーバーライドしません。これは、この 2 つのメソッドのシグネチャが異なるためです。

StudentRef メソッドに PersonRef メソッドと同じシグネチャおよび結果の型を指定しても、この問題の解消には役立ちません。これは、オブジェクトを PersonRef ではなく StudentRef として宣言することで得られる追加の型保証が失われるためです。

参照型間の手動変換

参照型は関連オブジェクト型の階層に従わないため、JPublisher には 2 つの参照型の間では直接変換できないという制限があります。背景情報として、この項では生成された cast() メソッドによる参照型間の変換操作について説明します。

これらの手動手順に従うことはお薦めしません。ここではあくまでも参考のために説明しています。かわりに、cast() メソッドを使用してください。

たとえば、次のコードを使用すると、参照型 XxxxRef から参照型 YyyyRef に変換できます。

```
java.sql.Connection conn = ...;  // get underlying JDBC connection
XxxxRef xref = ...;
YyyyRef yref = (YyyyRef) YyyyRef.getORADataFactory().
                create(xref.toDatum(conn),oracle.jdbc.OracleTypes.REF);
```

この変換は 2 つのステップで構成され、各ステップに独自のメリットがあります。

1. xref を強い XxxxRef 型から弱い oracle.sql.REF 型に変換します。

   ```
   oracle.sql.REF ref  = (oracle.sql.REF) xref.toDatum(conn);
   ```

2. oracle.sql.REF 型からターゲットの YyyyRef 型に変換します。

   ```
   YyyyRef yref = (YyyyRef) YyyyRef.getORADataFactory().
                              create(ref,oracle.jdbc.OracleTypes.REF);
   ```

次の「例 : 参照型間の手動変換」に、このような変換のサンプル・コードを示します。

> **注意 :** この変換では、タイプ・チェックは行われません。この変換が実際に許可されるかどうかは、使用中のアプリケーションおよび SQL スキーマに応じて異なります。

例 : 参照型間の手動変換

次の例には SQL 定義と Java コードが含まれており、前述のポイントを示しています。

SQL の定義 次の SQL 定義について考えます。

```
create type person_t as object (ssn number, name varchar2 (30), dob date) not final;
/
show errors

create type instructor_t under person_t (title varchar2(20)) not final;
/
show errors

create type instructorPartTime_t under instructor_t (num_hours number);
```

```
/
show errors

create type student_t under person_t (deptid number, major varchar2(30)) not final;
/
show errors

create type graduate_t under student_t (advisor instructor_t);
/
show errors

create type studentPartTime_t under student_t (num_hours number);
/
show errors

create table person_tab  of person_t;

insert into person_tab values (1001, 'Larry', TO_DATE('11-SEP-60'));
insert into person_tab values (instructor_t(1101, 'Smith', TO_DATE ('09-OCT-1940'),
'Professor'));
insert into person_tab values (instructorPartTime_t(1111, 'Myers',
TO_DATE('10-OCT-65'), 'Adjunct Professor', 20));
insert into person_tab values (student_t(1201, 'John', To_DATE('01-OCT-78'), 11,
'EE'));
insert into person_tab values (graduate_t(1211, 'Lisa', TO_DATE('10-OCT-75'), 12,
'ICS', instructor_t(1101, 'Smith', TO_DATE ('09-OCT-40'), 'Professor')));
insert into person_tab values (studentPartTime_t(1221, 'Dave', TO_DATE('11-OCT-70'),
13, 'MATH', 20));
```

JPublisher のマッピング JPublisher の実行時に次のマッピングがあるとします。

```
Person_t:Person,instructor_t:Instructor,instructorPartTime_t:InstructorPartTime,
graduate_t:Graduate,studentPartTime_t:StudentPartTime
```

Java のクラス 2-40 ページの「参照型間の手動変換」に示した参照型間の変換例の Java クラスを次に示します。

```
import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sqlj.runtime.Oracle;
import sqlj.runtime.ref.DefaultContext;
import sqlj.runtime.ResultSetIterator;

public class Inheritance
```

```
{
  public static void main(String[] args) throws SQLException
  {
    System.out.println("Connecting.");
    Oracle.connect("jdbc:oracle:oci:@","scott","tiger");

    // The following is only required in 9.0.1
    // or if the Java class hierarchy was created piecemeal
    System.out.println("Initializing type system.");
    new Person();
        new Instructor();
            new InstructorPartTime();
        new StudentT();
            new StudentPartTime();
            new Graduate();

    PersonRef p_ref;
    InstructorRef i_ref;
    InstructorPartTimeRef ipt_ref;
    StudentTRef s_ref;
    StudentPartTimeRef spt_ref;
    GraduateRef g_ref;

    System.out.println("Selecting a person.");
    #sql { select ref(p) INTO :p_ref FROM PERSON_TAB p WHERE p.NAME='Larry' };

    System.out.println("Selecting an instructor.");
    #sql { select ref(p) INTO :i_ref FROM PERSON_TAB p WHERE p.NAME='Smith' };

    System.out.println("Selecting a part time instructor.");
    #sql { select ref(p) INTO :ipt_ref FROM PERSON_TAB p WHERE p.NAME='Myers' };

    System.out.println("Selecting a student.");
    #sql { select ref(p) INTO :s_ref FROM PERSON_TAB p WHERE p.NAME='John' };

    System.out.println("Selecting a part time student.");
    #sql { select ref(p) INTO :spt_ref FROM PERSON_TAB p WHERE p.NAME='Dave' };

    System.out.println("Selecting a graduate student.");
    #sql { select ref(p) INTO :g_ref FROM PERSON_TAB p WHERE p.NAME='Lisa' };


    // Connection object for conversions
    Connection conn = DefaultContext.getDefaultContext().getConnection();

    // Assigning a part-time instructor ref to a person ref
    System.out.println("Assigning a part-time instructor ref to a person ref");
```

```
    oracle.sql.Datum ref = ipt_ref.toDatum(conn);
    PersonRef pref = (PersonRef) PersonRef.getORADataFactory().
                                          create(ref,OracleTypes.REF);
    // or just use:  PersonRef pref = PersonRef.cast(ipt_ref);

    // Assigning a person ref to an instructor ref
    System.out.println("Assigning a person ref to an instructor ref");
    InstructorRef iref = (InstructorRef) InstructorRef.getORADataFactory().
                          create(pref.toDatum(conn), OracleTypes.REF);
    // or just use:  InstructorRef iref = InstructorRef.cast(pref);

    // Assigning a graduate ref to an part time instructor ref
    // ==> this should actually bomb at runtime!
    System.out.println
              ("Assigning a graduate ref to a part time instructor ref");
    InstructorPartTimeRef iptref =
       (InstructorPartTimeRef) InstructorPartTimeRef.getORADataFactory()
                               .create(g_ref.toDatum(conn), OracleTypes.REF);
    // or just use:  InstructorPartTimeRef iptref =
InstructorPartTimeRef.cast(g_ref);

    Oracle.close();
  }
}
```

SQLData のオブジェクト型および継承

前述のように、JPublisher で `-usertypes=oracle` のかわりに `-usertypes=jdbc` 設定を使用すると、JPublisher で生成されるカスタム Java クラスでは、Oracle `ORAData` インタフェースのかわりに標準 `SQLData` インタフェースが実装されます。`SQLData` の標準 `readSQL()` および `writeSQL()` メソッドには、データの読取りと書込みに関して `ORAData/ORADataFactory` の `create()` および `toDatum()` メソッドと同等の機能があります。

JPublisher で SQL オブジェクト型の階層に対応する `ORAData` クラスが生成される場合と同様に、SQL 階層に対応する `SQLData` クラスの生成時には、Java の型は SQL 型と同じ階層に従います。

ただし、`SQLData` の実装には、JPublisher が `ORAData` クラスに自動的に生成する暗黙的なマッピングのインテリジェント機能はありません（2-36 ページの「ORAData のオブジェクト型および継承」を参照してください）。

`SQLData` の使用例では、SQL オブジェクト型と Java の型の間に適切なマッピングが得られるように、型マップを手動で提供する必要があります。JDBC アプリケーションでは、接続のデフォルトの型マップを正常に初期化するか、`getObject()` の入力パラメータとして型マップを明示的に指定できます。（詳細は、『Oracle9*i* JDBC 開発者ガイドおよびリファレン

ス』を参照してください。）SQLJ アプリケーションでは、プロパティ・ファイルに似た性質を持つ型マップ・リソースを使用します。（詳細は、『Oracle9*i* SQLJ 開発者ガイドおよびリファレンス』を参照してください。）

また、SQLData 実装では、強い型指定のオブジェクト参照に対するサポートがないことに注意してください。すべてのオブジェクト参照は弱い型指定の `java.sql.Ref` インスタンスです。

SQL FINAL、NOT FINAL、INSTANTIABLE、NOT INSTANTIABLE を使用する効果

この項では、SQL 修飾子 `FINAL`、`NOT FINAL`、`INSTANTIABLE` または `NOT INSTANTIABLE` の使用が JPublisher で生成されるラッパー・クラスに与える効果について説明します。

SQL 修飾子 `FINAL` または `NOT FINAL` を SQL 型または SQL 型のメソッドに使用しても、生成される Java ラッパー・コードには影響しません。このため、JPublisher ユーザーは常に、生成された動作をサブクラス化し、オーバーライドして、生成された Java ラッパー・クラスをカスタマイズできます。

SQL 修飾子 `NOT INSTANTIABLE` を SQL 型のメソッドに使用すると、Java ラッパー・クラスにそのメソッド用のコードは生成されません。したがって、そのメソッドをコールするには、インスタンス化できる SQL サブタイプに対応するなんらかのラッパー・クラスにキャストする必要があります。

SQL 型に `NOT INSTANTIABLE` を使用すると、`protected` コンストラクタで対応するラッパー・クラスが生成されます。これにより、インスタンス化できる SQL 型に対応するサブクラス経由でなければ、そのクラスのインスタンスを作成できないことがわかります。

下位互換性および移行

この項では、下位互換性、JDK のバージョン間の互換性、Oracle8*i* リリースと Oracle9*i* リリース間の JPublisher の移行の問題について説明します。

JPublisher の下位互換性

JPublisher ランタイムは、`classes111`、`classes12` または `ojdbc14` ライブラリ内の Oracle JDBC でパッケージ化されています。以前のバージョンの JPublisher で生成されたコードは、次のようになります。

- 現行のリリースの JPublisher ランタイムで引き続き動作します。
- 現行のリリースの JPublisher ランタイムで引き続きコンパイル可能です。

以前のリリースの JPublisher ランタイムと Oracle JDBC をコード生成に使用すると、コードはそのバージョンの JPublisher ランタイムでコンパイル可能になります。特に、Oracle8*i* JDBC ドライバを使用すると、JPublisher では現在使用不可の `CustomDatum` インタフェース用のコードが生成され、代替の `ORAData` インタフェース用のコードは生成されません。

JDK のバージョン間の JPublisher の互換性

一般に、JPublisher で生成された .sqlj ファイルは、JDK 1.1.x（JDBC 2.0 固有の型を使用していない場合）または JDK 1.2.x 以上で変換できます。ただし、変換とコンパイルを別のステップに分ける（.class ファイルではなく .java ファイルのみが生成されるように SQLJ で -compile=false を設定する）場合は、特別な JPublisher オプション設定を使用しないかぎり、コンパイルには変換と同じバージョンの JDK を使用する必要があります。

この場合（変換とコンパイルを別のステップで実行する場合）、JPublisher のデフォルト設定 -context=DefaultContext では、JDK 1.1.x および JDK 1.2.x 以上の間で完全な互換性を持つ .sqlj ファイルが生成されます。（たとえば、この設定では、JDK 1.1.x に対して変換できますが、JDK 1.2.x に対しても正常にコンパイルされます。）

この場合、生成されたすべての .sqlj ファイルでは、すべての接続コンテキストに sqlj.runtime.ref.DefaultContext クラスが使用されます。これに対して、-context=generated 設定の場合、生成された各 .sqlj ファイルでは独自の接続コンテキストのインナー・クラスが宣言されます。これは Oracle8*i* JPublisher のデフォルトの動作であり、変換された .java コードが JDK 1.1.x および JDK 1.2.x 以上の間で非互換になる原因です。

-context オプションの詳細は、3-15 ページの「SQLJ 接続コンテキスト・クラス（-context）」を参照してください。

> **重要：** 一部の JPublisher オプション設定を JDK 1.1.x で使用すると、クローズされていない SQLJ 接続コンテキスト・インスタンスによりメモリー・リークが発生する危険性があります。詳細は、2-28 ページの「接続コンテキストのリソースの解放」を参照してください。

接続コンテキストの概要は、『Oracle9*i* SQLJ 開発者ガイドおよびリファレンス』を参照してください。

Oracle8*i* JPublisher および Oracle9*i* JPublisher 間の移行

Oracle9*i* JPublisher では、デフォルトのオプション設定と生成されるコードの一部の機能が変更されています。リリース 8.1.7 以前の JPublisher を使用してアプリケーションを記述した場合、JPublisher を Oracle9*i* で再実行するだけでは、生成されたクラスをアプリケーションで引き続き動作させることはできません。この項では、JPublisher のオプション設定やアプリケーション・コードを適切に変更する方法について説明します。

> **注意：** JPublisher で生成されるクラスを拡張するクラスに関する Oracle8*i* と Oracle9*i* の機能の違いは、2-34 ページの「Oracle9i JPublisher で生成されたクラスのユーザー作成サブクラスでの変更」も参照してください。

Oracle9*i* JPublisher での動作の変更

Oracle9*i* では、JPublisher の動作が次のように変更されていることに注意してください。

- デフォルトでは、JPublisher は、オブジェクト型ごとに SQLJ 接続コンテキストのインナー・クラス `_Ctx` を宣言しません。かわりに、すべてに接続コンテキスト・クラス `sqlj.runtime.ref.DefaultContext` が使用されます。

  また、ユーザーが記述したコードでは、Oracle8*i* のコード生成時に宣言されていた `_ctx` 接続コンテキスト・フィールドを使用するかわりに、`getConnectionContext()` メソッドをコールして接続コンテキスト・ハンドルを取得する必要があります。`getConnectionContext()` メソッドの詳細は、2-26 ページの「接続コンテキストおよび接続インスタンスの使用時の考慮点」を参照してください。

- `-methods=true` 設定の場合は、基礎となる SQL オブジェクト型または PL/SQL パッケージでメソッドが定義されていなければ、`.sqlj` ファイルではなく `.java` ファイルが生成されます。（ただし、`-methods=always` 設定の場合は、常に `.sqlj` ファイルが生成されます。）

- デフォルトでは、JPublisher は、使用不可の `oracle.sql.CustomDatum` インタフェースではなく、`oracle.sql.ORAData` インタフェースを実装するコードを生成します。

- デフォルトでは、JPublisher は、生成されたコードをカレント・ディレクトリのパッケージ・ディレクトリ階層に格納するのではなく、カレント・ディレクトリに格納します。

Oracle8*i* の動作に戻す方法は、「JPublisher に以前のリリースでの動作を強制する個々の設定」および 2-47 ページの「Oracle8i 互換モード」を参照してください。

JPublisher に以前のリリースでの動作を強制する個々の設定

Oracle9*i* では、JPublisher をリリース 8.1.7 以前と同様に動作させる必要が生じた場合に備えて、個別の下位互換性オプションが多数用意されています。詳細は、表 2-2 を参照してください。各オプションの詳細は、3-12 ページの「JPublisher の一般オプションの詳細説明」を参照してください。

単一の設定で Oracle8*i* JPublisher と同じように動作させる方法は、2-47 ページの「Oracle8i 互換モード」を参照してください。この設定を使用すると、下位互換のコードが生成されるのみでなく、個々のオプション設定を組み合せた場合と同じ動作が得られます。

表 2-2 JPublisher の下位互換性オプション

オプション設定	動作
-context=generated	SQLJ 接続コンテキスト用のインナー・クラス `_Ctx` が宣言されます。このインナー・クラスは、デフォルトの `DefaultContext` クラスまたはユーザー指定の接続コンテキスト・クラスのかわりに使用されます。

表 2-2 JPublisher の下位互換性オプション （続き）

オプション設定	動作
-methods=always	基礎となる SQL オブジェクトまたはパッケージで実際にメソッドが定義されているかどうかに関係なく、JPublisher で生成されるすべてのクラス用に、.sqlj（.java ではなく）ソース・ファイルを強制的に生成させます。
-compatible=customdatum	Oracle 固有のオブジェクト・ラッパーについて、oracle.sql.ORAData および ORADataFactory インタフェースのかわりに、使用不可の（ただし引き続きサポート対象の）oracle.sql.CustomDatum および CustomDatumFactory インタフェースが実装されます。
-dir=.	このオプションを「.」（ピリオド）に設定すると、Oracle8*i* のデフォルトの動作と同様に、階層内のカレント・ディレクトリに出力ファイルが生成されます。

ただし、下位互換性の設定を使用することが避けられない場合以外は、現行のデフォルト（または他の）設定を受け入れることをお薦めします。

Oracle8*i* 互換モード

JPublisher のオプション設定 -compatible=both8i および -compatible=8i は、どちらを使用した場合も、Oracle8*i* 互換モードになります。

このオプションの詳細は、3-8 ページの「ユーザー定義型用の下位互換性を持つ Oracle マッピング（-compatible）」を参照してください。

ただし、このモードを使用するには、次の 1 つ以上の要件に該当する必要があります。

- JPublisher で生成された .sqlj ファイルの変換時に、SQLJ のデフォルトの -codegen=oracle 設定を使用します。

または

- JPublisher で生成されたコードを JDK 1.2 以上で実行し、SQLJ の runtime12 または runtime12ee ライブラリを使用するか、Oracle9*i* リリースのサーバー側 Oracle JVM で実行します。

または

- -methods=false または -methods=none 設定を使用して JPublisher を実行します。

Oracle8*i* 互換モードの JPublisher には、次の機能があります。

- JPublisher では、ORAData インタフェースのかわりに、使用不可の CustomDatum および CustomDatumFactory インタフェースを実装するコードが生成されます（-compatible=customdatum の設定時と同様）。また、設定 -compatible=both8i

を選択すると、生成されたコードは ORAData インタフェースも実装しますが、ORADataFactory は実装しません。

- -methods=true 設定を使用すると、オブジェクト型でメソッドが定義されていない場合でも、常に SQL オブジェクト型用の SQLJ ソース・コードが生成されます（-methods=always の設定時と同様）。
- JPublisher では、次のように、オブジェクト型のラッパーごとに、接続コンテキスト宣言と接続コンテキスト・インスタンスが生成されます（-context=generated の設定時と同様）。

```
#sql static context _Ctx;
protected _Ctx _ctx;
```

- JPublisher には、入力として汎用 ConnectionContext インスタンス（標準 sqlj.runtime.ConnectionContext インタフェースを実装するクラスのインスタンス）を取るラッパー・クラスのコンストラクタが用意されています。Oracle9*i* では、このコンストラクタが受け入れるのは、DefaultContext インスタンス、あるいは JPublisher の実行時に -context オプションで指定されたクラスのインスタンスのみです。
- JPublisher には、JPublisher オブジェクトで暗黙的に作成された接続コンテキスト・インスタンスを解放するための API は用意されていません。

  これに対して、Oracle9*i* JPublisher には、オブジェクト用の接続コンテキスト・インスタンスを明示的に設定するための setConnectionContext() メソッドと、オブジェクトについて暗黙的に作成された接続コンテキスト・インスタンスを解放するための release() メソッドが用意されています。

通常、Oracle8*i* 互換モードの選択が必要な場合は、設定 -compatible=both8i を使用することをお薦めします。これにより、アプリケーションを Oracle9*i* Application Server などの中間層環境で使用でき、JDBC 接続はデータ・ソースを介して取得され、oracle.jdbc.Oracle*Xxxx* インタフェースを使用してラップされます。CustomDatum 実装では、このようにラップされた接続はサポートされません。

注意： 設定 -compatible=both8i を使用するには、Oracle JDBC 9.0.1 以上が必要です。

JPublisher で生成されるコードで接続コンテキスト・インスタンス _ctx を宣言するには、Oracle8*i* 互換モードを使用する必要があります。この特定の Oracle8*i* 動作を指定できる他のオプション設定はありません。_ctx インスタンスが役立つのは、それに依存する従来型コードがあるが、それ以外は getConnectionContext() メソッドを通じて接続コンテキスト・インスタンスを取得する必要がある場合です。

重要： Oracle8*i* 互換モードを使用できない状況があります。環境で次のいずれかを使用している場合です。

- JDK 1.1.x、SQLJ 汎用 `runtime` ライブラリまたは SQLJ `runtime11` ライブラリ

また、次の SQLJ Translator 設定を使用している場合です。

- `-codegen=iso`

さらに、次のいずれかの JPublisher 設定を使用している場合も同様です。

- `-methods=named`（または `some`）、`-methods=true`（または `all`）または `-methods=always`

この条件に該当する場合は、クローズされていない暗黙的な接続コンテキスト・インスタンスによって、重大なメモリー・リークが発生する危険性があります。

このような状況では、`-compatible=8i` および `-compatible=both8i` 設定を使用せず、接続コンテキストの操作には `setConnectionContext()` および `release()` メソッドを使用してください。詳細は、2-26 ページの「JPublisher により生成された SQLJ コードでの接続コンテキストおよびインスタンスの使用」を参照してください。

JPublisher の制限事項

この項では、Oracle9*i* リリース 2（9.2）バージョンの JPublisher の制限事項について説明します。

- 一部のデータ型は、PL/SQL 固有の型を SQL 型にマップする JPublisher の型マップを介して間接的にのみサポートされます。これには、次の型が含まれます。
  - RECORD 型
  - 索引付き表

  JPublisher では、`SYS.SQLJUTL` パッケージの変換ファンクションを使用して PL/SQL の `BOOLEAN` から Java の `boolean` にマップできるように、事前定義済みのサポートがあることに注意してください。通常、JPublisher では、認識されない 1 つ以上のデータ型を使用するラッパー・メソッドが検出されると、対応する Java メソッドは生成されず、1 つ以上のエラー・メッセージが表示されます。

  データ型サポートの詳細は、2-3 ページの「Oracle と JDBC の型への SQL と PL/SQL のマッピング」を参照してください。

- INPUT ファイルのエラー・レポートは不完全な場合があります。

  JPublisher では INPUT ファイル内のほとんどのエラーをレポートしていますが、すべてのエラーではありません。INPUT ファイル内のエラーで、JPublisher でレポートされないエラーは、3-37 ページの「INPUT ファイルの事前注意事項」を参照してください。

- -omit_schema_names オプションはブール型オプションとして動作しますが、他のブール型オプションとは異なり、=true または =false に設定することはできません。このオプションを使用可能にするには、単に -omit_schema_names と指定します。デフォルトでは、このオプションは使用禁止になっています。このオプションの詳細は、3-20 ページの「生成された名前からのスキーマ名の省略（-omit_schema_names）」を参照してください。

3

コマンドライン・オプションおよび入力ファイル

この章では、プログラムの動作を指定する JPublisher のオプション設定と入力ファイルの使用方法と構文の詳細について説明します。この章の内容は、次のとおりです。

JPublisher オプション

この項では、JPublisher のコマンドライン・オプションについて説明します。この項の内容は、次のとおりです。

- JPublisher のオプションのサマリー
- JPublisher オプションのヒント
- 表記法規約
- データ型マッピングに影響するオプションの詳細説明
- JPublisher の一般オプションの詳細説明

JPublisher のオプションのサマリー

表 3-1 に、JPublisher コマンドラインで使用できるオプション、その構文および簡単な説明を示します。「n/a」は適用外を表します。

表 3-1 JPublisher のオプションのサマリー

オプション名	説明	デフォルト値
`-access`	JPublisher により生成されたメソッド定義に組み込まれるアクセス修飾子を指定します。	`public`
`-adddefaulttypemap`	JPublisher のデフォルトの型マップにエントリを追加します。	n/a
`-addtypemap`	JPublisher のユーザー型マップにエントリを追加します。	n/a
`-builtintypes`	非数値、非 LOB のスカラー型のデータ型マッピング（`jdbc` または `oracle`）を指定します。	`jdbc`
`-case`	JPublisher で生成される Java 識別名の大文字小文字を指定します。	`mixed`
`-compatible`	汎用 Oracle8*i* 互換モード、またはユーザー定義型の Oracle マッピング用に生成されたクラスに実装する特定のインタフェースである `ORAData` または `CustomDatum`（下位互換性のためにサポート）を指定し、`-usertypes=oracle` の動作を変更します。	`oradata`
`-context`	JPublisher で接続コンテキストに使用するクラスとして、SQLJ の `DefaultContext` クラス、ユーザー指定クラスまたは JPublisher で生成されるインナー・クラスを指定します。	`DefaultContext`

表 3-1 JPublisher のオプションのサマリー （続き）

オプション名	説明	デフォルト値
-defaulttypemap	JPublisher で使用するデフォルトの型マップを設定します。	2-17 ページの「JPublisher のデフォルトの型マップとユーザー型マップ」を参照
-dir	生成されたファイルまたはパッケージを保持するディレクトリを指定します。空のディレクトリ名を指定すると、すべての生成ファイルはカレント・ディレクトリに格納されます。空でないディレクトリ名を指定すると、クラス階層のルート・ディレクトリとして使用するディレクトリを指定します。	空
-driver	JPublisher でデータベースへの JDBC 接続に使用するドライバのクラスを指定します。	oracle.jdbc.OracleDriver
-encoding	JPublisher の入力ファイルおよび出力ファイルの Java における文字のエンコーディングを指定します。	システム・プロパティ file.encoding の値
-gensubclass	ユーザー・サブクラス用のスタブ・コードを生成するかどうかと生成方法を指定します。	true
-input（または -i）	JPublisher で変換する型およびパッケージをリストするファイルを指定します。	n/a
-lobtypes	JPublisher で BLOB 型および CLOB 型に使用するデータ型マッピング（jdbc または oracle）を指定します。	oracle
-mapping	オブジェクト属性の型とメソッドの引数の型について、生成されたメソッドでサポートするマッピングを指定します。 **注意：**これは、「XXXtypes」マッピング・オプションのために使用不可になっていますが、下位互換性のためにサポートされています。	objectjdbc
-methods	JPublisher で SQL オブジェクトのメソッドと PL/SQL パッケージのメソッド用にラッパー・メソッドを生成するかどうかを指定します。また、副次効果として、JPublisher で .sqlj ファイルと .java ファイルのどちらを生成するかと、PL/SQL ラッパー・クラスを生成するかどうかを指定します。	all
-numbertypes	JPublisher で数値型に使用するデータ型マッピング（jdbc、objectjdbc、bigdecimal または oracle）を指定します。	objectjdbc

表 3-1 JPublisher のオプションのサマリー （続き）

オプション名	説明	デフォルト値
-omit_schema_names	JPublisher で生成されるすべてのオブジェクト型の名前およびパッケージ名にスキーマ名を含めるかどうかを指定します。	使用禁止（スキーマ名は省略しないでください）
-package	JPublisher で Java ラッパーが生成される Java パッケージの名前を指定します。	n/a
-plsqlfile	JPublisher で PL/SQL ラッパー・ファンクションおよびプロシージャが生成されるファイルを指定します。	plsql_wrapper.sql
-plsqlmap	PL/SQL ラッパー・ファンクションおよびプロシージャを生成するかどうかと、生成方法を指定します。	true
-plsqlpackage	JPublisher でラッパー・ファンクションおよびプロシージャを生成する PL/SQL パッケージを指定します。	JPUB_PLSQL_WRAPPER
-props（または -p）	コマンドラインにリストされているオプション以外の JPublisher オプションを含むファイルを指定します。	n/a
-serializable	オブジェクト型用に生成されたコードで java.io.Serializable を実装するかどうかを指定します。	false
-sql（または -s）	JPublisher でコードを生成するオブジェクト型およびパッケージを指定します。	n/a
-tostring	オブジェクト型用の toString() メソッドを生成するかどうかを指定します。	false
-typemap	JPublisher の型マップ（マッピングのリスト）を指定します。	空
-types	JPublisher でコードを生成するオブジェクト型を指定します。 **注意：** このオプションは -sql のために使用不可になっていますが、下位互換性のためにサポートされています。	n/a

表 3-1 JPublisher のオプションのサマリー （続き）

オプション名	説明	デフォルト値
-url	JPublisher でデータベースの接続に使用する URL を指定します。	jdbc:oracle:oci:@
-user（または -u）	接続に使用する Oracle ユーザー名およびパスワードを指定します。	n/a
-usertypes	JPublisher でユーザー定義の SQL 型に使用する型マッピング（jdbc または oracle）を指定します。	oracle

JPublisher オプションのヒント

JPublisher オプションには、次のような使用上の注意があります。

- JPublisher では、常に -user オプション（または、その短縮形 -u）が必要です。
- オプションはその出現順に処理されます。INPUT ファイルからのオプションは、-input（または -i）オプションが出現した時点で処理されます。同様に、プロパティ・ファイルからのオプションは、-props（または -p）オプションが出現した時点で処理されます。
- 特定のオプションが複数回出現すると、通常、JPublisher ではそのオプションで最後に出現した値が使用されます。ただし、次のオプションの場合は累積されます。

  -sql（または使用不可の -types）

  -addtypemap または -adddefaulttypemap
- 通常、オプションおよび対応するオプション値は、等号（=）で区切る必要があります。ただし、次のオプションをコマンドラインで指定する場合は、セパレータとして空白も使用できます。

  -sql（または -s）、-user（または -u）、-props（または -p）および -input（または -i）
- コマンドラインまたはプロパティ・ファイルで -package オプションを使用して、生成されたクラス用の Java パッケージを指定することをお薦めします。たとえば、コマンドラインでは次のように入力します。

  ```
  jpub -sql=Person -package=e.f ...
  ```

  プロパティ・ファイルの場合は、次のように入力します。

  ```
  jpub.sql=Person
  jpub.package=e.f
  ...
  ```

これらの文により、Java パッケージ `e.f` 内のクラス `Person`（つまり、クラス `e.f.Person`）の作成を JPublisher に指示します。

プロパティ・ファイルの説明は、3-30 ページの「プロパティ・ファイルの構造および構文」を参照してください。

- `INPUT` ファイルまたはコマンドラインで型またはパッケージを指定しない場合、JPublisher ではユーザーのスキーマのすべての型およびパッケージが、コマンドラインまたはプロパティ・ファイルで指定したオプションに従って変換されます。

表記法規約

後続の項で使用される JPublisher オプション構文は、次の表記法規約に従います。

- 山カッコ `<...>` はユーザー指定の文字列を囲みます。
- 中カッコ `{...}` は可能な値のリストを囲みます。中カッコ内の値から 1 つのみを指定します。
- 縦線 | は大カッコまたは中カッコ内の複数の代替選択項目を区切ります。
- イタリックの文字は入力値で、実際の値または文字列を指定します。
- 大カッコ `[...]` はオプション項目を囲みます。ただし、大カッコまたはカッコが構文の一部になっており、そのとおりに入力する必要がある場合があります。その場合、このマニュアルでは太字の **[...]** または **(...)** を使用しています。
- 項目（カッコで囲まれている項目）の直後に省略記号 `...` がある場合は、項目を何回でも繰り返すことができます。
- 前述の句読点記号以外は示されたとおりに入力します。たとえば、「`.`（ピリオド）」や「`@`」などです。

次の項では、データ型マッピングに影響するオプションを説明します。残りのオプションをアルファベット順に説明します。

データ型マッピングに影響するオプションの詳細説明

次のオプションは、JPublisher でオブジェクト型、コレクション型、オブジェクト参照型および PL/SQL パッケージを Java クラスに変換するときに使用するデータ型マッピングを制御します。

- `-usertypes` オプションは、ユーザー定義型に対する JPublisher の動作を制御します（`oracle` マッピング用の `-compatible` オプションと併用できます）。
- `-numbertypes` オプションは、数値型のデータ型マッピングを制御します。
- `-lobtypes` オプションは、`BLOB` 型および `CLOB` 型のデータ型マッピングを制御します。
- `-builtintypes` オプションは、非数値型で非 LOB 型の、事前定義済み SQL 型および PL/SQL 型のデータ型マッピングを制御します。

これらの 4 つのオプションは型マッピング・オプションです。（柔軟性の低いもう 1 つのオプション `-mapping` については後述します。このオプションは使用不可になっていますが、JPublisher の旧リリースとの互換性を保つためにサポートされています。）

また、JPublisher のコード生成は、JPublisher のユーザー型マップまたはデフォルトの型マップのエントリを介して制御されます。これは、主に JPublisher で PL/SQL 型を持つシグネチャへのアクセスを可能にするためです。詳細は、2-7 ページの「JDBC でサポートされないデータ型の使用」を参照してください。

次の各オプションは、JPublisher の型マッピングとともに使用します。一般オプションの項を参照してください。

- 型マッピングを指定するための `-addtypemap`、`-adddefaulttypemap`、`-defaulttypemap` および `-typemap`
- PL/SQL ラッパー・コードの生成を制御するための `-plsqlfile`、`-plsqlmap` および `-plsqlpackage`

オブジェクト型の場合、JPublisher で型マッピング・オプションにより指定したマッピングが、オブジェクトの属性およびそのオブジェクトに含まれるメソッドの引数と結果に適用されます。このマッピングは、生成されたアクセッサ・メソッドでサポートされる型、つまり `get`*`XXX`*`()` メソッドで戻されて、`set`*`XXX`*`()` メソッドで要求される型を制御します。

PL/SQL パッケージの場合、JPublisher ではマッピングはパッケージ内のメソッドの引数および結果に適用されます。

コレクション型の場合、JPublisher ではマッピングはコレクション要素の型に適用されます。

`-usertypes` オプションは、JPublisher で生成されたクラスに、Oracle `ORAData` インタフェースと標準 `SQLData` インタフェースのどちらを実装するかと、JPublisher でコレクション型とオブジェクト参照型用のコードを生成するかどうかを制御します。また、`-usertypes=oracle` の場合は、`-compatible` オプションを使用して、Oracle マッピングに `ORAData` ではなく `CustomDatum` を使用するように指定できます。`CustomDatum` は、Oracle9*i* では `ORAData` に置き換えられて使用不可になっていますが、下位互換性のためにサポートされています。（また、`-compatible` オプションを使用すると、より汎用的な

Oracle8*i* 互換モードを指定できます。2-47 ページの「Oracle8i 互換モード」を参照してください。）

各種データ型マッピングと、使用するマッピングを決定するときに考慮が必要な要素の詳細は、2-2 ページの「データ型マッピングの詳細」を参照してください。

以降の項では、これらの型マッピング・オプションの追加情報を説明します。

ユーザー定義型のマッピング（-usertypes）

```
-usertypes={oracle|jdbc}
```

`-usertypes` オプションは、ユーザー定義型用に生成されたクラスに、Oracle `ORAData` インタフェースと標準 `SQLData` インタフェースのどちらを実装するかを制御します。

`-usertypes=oracle`（デフォルト）の場合、JPublisher でオブジェクト型、コレクション型およびオブジェクト参照型用に `ORAData` クラスが生成されます。

`-usertypes=jdbc` の場合、JPublisher でオブジェクト型用に `SQLData` クラスが生成されます。この場合、コレクション型やオブジェクト参照型用のクラスは生成されません。すべてのコレクション型には `java.sql.Array` を、すべてのオブジェクト参照型には `java.sql.Ref` を使用します。

注意：

- `SQLData` インタフェースは JDBC 2.0 の機能であるため、`-usertypes=jdbc` 設定には JDK 1.2 以上が必要です。
- `-compatible` オプションの特定の設定では、`-usertypes=oracle` に設定すると、`ORAData` インタフェースのかわりに使用不可の `CustomDatum` インタフェースを実装するクラスが生成されます。次の「ユーザー定義型用の下位互換性を持つ Oracle マッピング（-compatible）」を参照してください。

ユーザー定義型用の下位互換性を持つ Oracle マッピング（-compatible）

```
-compatible={oradata|customdatum|both8i|8i}
```

`-usertypes=oracle` の場合は、オプションで `-compatible=customdatum` に設定し、ユーザー定義型用に生成されたクラスに、`ORAData` インタフェースのかわりに `CustomDatum` インタフェースを実装できます。`CustomDatum` は、Oracle9*i* では `ORAData` に置き換えられて使用不可になっていますが、下位互換性のためにサポートされています。`-usertypes=jdbc` の場合、`-compatible` 設定の `customdatum`（または `oradata`）は無視されます。

デフォルト設定は `oradata` です。

また、このオプションにはもう 1 つの操作モードがあります。-compatible=8i または -compatible=both8i に設定すると、汎用的な Oracle8*i* 互換モードを指定できます。このモードでは、CustomDatum インタフェースが使用されるだけでなく、Oracle8*i* JPublisher で生成されるものと同じコードが生成され、Oracle8*i* への下位互換性に関する他の JPublisher オプションを設定した場合と同じ効果が得られます。メソッド生成の動作は -methods=always に設定した場合と同じで、接続コンテキスト宣言の生成は -context=generated に設定した場合と同じです。2-47 ページの「Oracle8i 互換モード」を参照してください。

> **注意：** ORAData インタフェースがサポートされない環境（Oracle8*i* JDBC 8.1.7 以下のリリースなど）で JPublisher を使用する場合は、-usertypes=oracle に設定すると CustomDatum インタフェースが自動的に使用されます。（-compatible=oradata に設定すると警告メッセージが表示されますが、生成は行われます。）
>
> オプション設定 -compatible=both8i では、生成されたオブジェクト型のラッパーにより ORAData インタフェースが追加実装されます。通常、Oracle9*i* Application Server などの中間層で実行するプログラムには、ORAData のサポートが必要であるため、この設定は -compatible=8i 設定より優先されます。ただし、ORAData を使用するには、Oracle 9.0.1 以上の JDBC ドライバが必要であることに注意してください。

数値型のマッピング（-numbertypes）

-numbertypes={jdbc|**objectjdbc**|bigdecimal|oracle}

-numbertypes オプションは、数値 SQL 型および PL/SQL 型のデータ型マッピングを制御します。次の 4 つから選択できます。

- JDBC マッピングでは、ほとんどの数値型は int および float などの Java 基本型にマップされ、DECIMAL および NUMBER は java.math.BigDecimal にマップされます。
- Object JDBC マッピング（デフォルト）では、ほとんどの数値型は java.lang.Integer および java.lang.Float などの Java ラッパー・クラスにマップされ、DECIMAL および NUMBER は java.math.BigDecimal にマップされます。
- BigDecimal マッピングでは、すべての数値型は java.math.BigDecimal にマップされます。
- Oracle マッピングでは、すべての数値型は oracle.sql.NUMBER にマップされます。

表 3-2 に、-numbertypes オプションで影響を受ける SQL 型および PL/SQL 型を示します。また、-numbertypes=jdbc および -numbertypes=objectjdbc（デフォルト）の Java 型マッピングも示します。

表 3-2 -numbertypes オプションの影響を受ける型のマッピング

SQL または PL/SQL データ型	JDBC マッピングの型	Object JDBC マッピングの型
BINARY_INTEGER、INT、INTEGER、NATURAL、NATURALN、PLS_INTEGER、POSITIVE、POSITIVEN、SIGNTYPE	int	java.lang.Integer
SMALLINT	short	java.lang.Integer
REAL	float	java.lang.Float
DOUBLE PRECISION、FLOAT	double	java.lang.Double
DEC、DECIMAL、NUMBER、NUMERIC	java.math.BigDecimal	java.math.BigDecimal

LOB 型のマッピング（-lobtypes）

```
-lobtypes={jdbc|oracle}
```

`-lobtypes` オプションは、LOB 型のデータ型マッピングを制御します。表 3-3 に、`-lobtypes=oracle`（デフォルト）および `-lobtypes=jdbc` の場合のこれらの型のマッピングを示します。

表 3-3 -lobtypes オプションの影響を受ける型のマッピング

SQL または PL/SQL データ型	Oracle マッピングの型	JDBC マッピングの型
CLOB	oracle.sql.CLOB	java.sql.Clob
BLOB	oracle.sql.BLOB	java.sql.Blob
BFILE	oracle.sql.BFILE	oracle.sql.BFILE

注意：

- `BFILE` は Oracle 固有の SQL 型であるため、標準の `java.sql.Bfile` の Java 型はありません。
- `NCLOB` は Oracle 固有の SQL 型です。`CLOB` の使用を NCHAR 形式で示し、SQLJ プログラムでは `oracle.sql.NCLOB` のインスタンスとして表されます。
- `java.sql.Clob` インタフェースおよび `java.sql.Blob` インタフェースは、JDK1.2 での新しいインタフェースです。JDK 1.1 を使用する場合は、`-lobtypes=jdbc` を選択しないでください。

スカラー型のマッピング（-builtintypes）

-builtintypes={**jdbc**|oracle}

-builtintypes オプションは、LOB 型（-lobtypes オプションで制御）および各種数値型（-numbertypes オプションで制御）を除く、すべてのスカラー型のデータ型マッピングを制御します。表 3-4 に、-builtintypes オプションの影響を受けるデータ型と、-builtintypes=oracle および -builtintypes=jdbc（デフォルト）の場合の Java の型マッピングを示します。

表 3-4 -builtintypes オプションの影響を受ける型のマッピング

SQL または PL/SQL データ型	Oracle マッピングの型	JDBC マッピングの型
CHAR、CHARACTER、LONG、STRING、VARCHAR、VARCHAR2	oracle.sql.CHAR	java.lang.String
RAW、LONG RAW	oracle.sql.RAW	byte[ ]
DATE	oracle.sql.DATE	java.sql.Timestamp
TIMESTAMP、TIMESTAMP WITH TZ、TIMESTAMP WITH LOCAL TZ	oracle.sql.TIMESTAMP、oracle.sql.TIMESTAMPTZ、oracle.sql.TIMESTAMPLTZ	java.sql.Timestamp

すべての型のマッピング（-mapping）

-mapping={jdbc|**objectjdbc**|bigdecimal|oracle}

> **注意：** このオプションは、より限定的な型マッピング・オプション -usertypes、-numbertypes、-builtintypes および -lobtypes のために使用不可になっています。ただし、下位互換性のために引き続きサポートされています。

-mapping オプションでは、すべてのデータ型のマッピングを指定するため、型間の柔軟性はほとんど得られません。

-mapping=oracle 設定は、すべての型マッピング・オプションを oracle に設定するのと同等です。表 3-5 に示すように、他の -mapping 設定は、-numbertypes を -mapping の値と同一に設定し、他の型マッピング・オプションをデフォルトに設定するのと同等です。

表 3-5 -mapping 設定と他のマッピング・オプション設定の関係

	-builtintypes=	-numbertypes=	-lobtypes=	-usertypes=
-mapping=oracle	oracle	oracle	oracle	oracle
-mapping=jdbc	jdbc	jdbc	oracle	oracle
-mapping=objectjdbc（デフォルト）	jdbc	objectjdbc	oracle	oracle
-mapping=bigdecimal	jdbc	bigdecimal	oracle	oracle

注意： オプションはコマンドラインに入力した順に処理されるため、-mapping オプションが特定の型マッピング・オプション（-builtintypes、-lobtypes、-numbertypes または -usertypes）の前にある場合、その型マッピング・オプションは、関連する型の -mapping オプションをオーバーライドします。-mapping オプションが特定の型マッピング・オプションの後に続く場合は、その型マッピング・オプションは無視されます。

JPublisher の一般オプションの詳細説明

この項では、データ型マッピング以外の設定に使用する残りの JPublisher オプションについて説明します。この項のオプションは、アルファベット順になっています。

メソッドへのアクセス（-access）

-access={**public**|protected|package}

-access オプションは、JPublisher により生成されたコンストラクタ、属性の setter メソッドと getter メソッド、オブジェクト型ラッパー・クラスのメンバー・メソッドおよび PL/SQL パッケージのメソッドについて組み込まれるアクセス修飾子を設定します。

JPublisher で可能なオプション設定は、次のとおりです。

- public（デフォルト）－ public アクセス修飾子を持つメソッドが生成されます。
- protected － protected アクセス修飾子を持つメソッドが生成されます。
- package － アクセス修飾子は省略されます。つまり、生成されたメソッドはパッケージに対してローカルになります。

生成された JPublisher ラッパー・クラスの使用を制御する必要がある場合は、-access=protected または -access=package 設定を使用します。JPublisher で生成されたクラスのサブクラスとして、カスタマイズしたバージョンのラッパーを提供し、生成されたスーパークラスへのアクセスは提供しないようにします。

コマンドラインまたはプロパティ・ファイルで -access オプションを指定できます。

注意： オブジェクト参照、VARRAY およびネストした表用のラッパーは、-access オプションの値の影響を受けません。

デフォルトの型マップの追加エントリ（-adddefaulttypemap）

-adddefaulttypemap=<*list_of_typemap_entries*>

このオプションを使用すると、JPublisher で使用されるデフォルトの型マップに、1 つのエントリまたはカンマ区切りのエントリ・リストを追加できます。このオプションは、JPublisher 内部でデフォルトの型マップの設定に使用されます。型マップ・エントリの書式は、次の「ユーザー型マップの追加エントリ（-addtypemap）」を参照してください。

注意： デフォルトの型マップとユーザー型マップの競合回避の詳細は、2-17 ページの「JPublisher のデフォルトの型マップとユーザー型マップ」を参照してください。この項では、デフォルトの型マップの初期の内容についても説明します。

ユーザー型マップの追加エントリ（-addtypemap）

-addtypemap=<*list_of_typemap_entries*>

このオプションを使用すると、JPublisher のユーザー型マップに、1 つのエントリまたはカンマ区切りのエントリ・リストを追加できます。エントリの書式は、次のいずれかです。

```
-addtypemap=<opaque_sql_type>:<java_type>
-addtypemap=<numeric_indexed_by_table>:<java_numeric_type>[<max_length>]
-addtypemap=<char_indexed_by_table>:<java_char_type>[<max_length>](<elem_size>)
-addtypemap=<plsql_type>:<java_type>:<sql_type>:<sql_to_plsql_func>:
            <plsql_to_sql_func>
```

[...] と **(...)** が構文の一部であることに注意してください。また、一部のオペレーティング・システムでは、特殊文字を含むコマンドライン・オプションを引用符で囲む必要があります。

最大配列長 <*max_length*> と最大要素サイズ指定 <*elem_size*> は、省略できる場合があります。

-addtypemap オプションと -typemap オプションには、-addtypemap ではユーザー型マップにエントリを追加するのに対して、-typemap では既存の型マップを指定したエントリで置き換えるという違いがあります。3-27 ページの「JPublisher の型マップの置換（-typemap）」を参照してください。

前述の -addtypemap の最初の書式の詳細は、2-7 ページの「OPAQUE 型の型マッピング・サポート」を参照してください。2 番目と 3 番目の書式の説明は、2-8 ページの「JDBC OCI を使用したスカラーの索引付き表の型マッピング・サポート」を参照してください。最後の書式の説明は、2-10 ページの「PL/SQL 変換ファンクションを介した型マッピングのサポート」を参照してください。

注意： デフォルトの型マップとユーザー型マップの競合回避の詳細は、2-17 ページの「JPublisher のデフォルトの型マップとユーザー型マップ」を参照してください。

Java 識別名の大文字小文字の区別（-case）

-case={**mixed**|same|lower|upper}

クラスまたは属性名を INPUT ファイルまたはコマンドラインで指定しない場合、-case オプションは JPublisher で生成される Java 識別子の大 / 小文字区別に影響します。これには、クラス名、メソッド名、getXXX() および setXXX() メソッド名に埋め込まれる属性名、生成されるメソッド名の引数および Java ラッパー名が含まれます。

表 3-6 に、-case オプションで可能な値を示します。

表 3-6 -case オプションの値

-case オプション値	説明
mixed（デフォルト）	クラス名の各単語の 1 文字目、またはメソッド名の最初のワード単位の後の各ワード単位は、大文字になります。他のすべての文字は小文字です。アンダースコア（_）、ドル記号（$）および Java で無効なすべての文字は、ワード単位の境界を構成して暗黙的に削除されます。単語の境界はメソッド名の get または set の後にも出現します。
same	JPublisher でデータベース上での大文字小文字の区別は変更されず、アンダースコアおよびドル記号は保持されます。Java で無効な他の文字は削除され、警告メッセージが発行されます。
upper	JPublisher で小文字は大文字に変換され、アンダースコアおよびドル記号は保持されます。Java で無効な他の文字は削除され、警告メッセージが発行されます。
lower	JPublisher で大文字は小文字に変換され、アンダースコアおよびドル記号は保持されます。Java で無効な他の文字は削除され、警告メッセージが発行されます。

-sql オプションを使用して入力したクラス名や属性名、あるいは INPUT ファイルのクラス名の場合、JPublisher ではその名前の文字の大 / 小文字区別が保持され、-case オプションがオーバーライドされます。

JPublisher では、コマンドラインまたは INPUT ファイルで指定したオブジェクト型の Java クラス識別名の大文字小文字は書き込まれたとおりに保持されます。たとえば、コマンドラインに次のように含まれていると、

```
-sql=Worker
```

次のように生成されます。

```
public class Worker ... ;
```

INPUT ファイルのエントリを次のように書き込むと、

```
SQL wOrKeR
```

次に示すように、識別名の大文字小文字は INPUT ファイルに入力されたとおりに生成されます。

```
public class wOrKeR ... ;
```

SQLJ 接続コンテキスト・クラス（-context）

-context={generated|**DefaultContext**|*user-specified*}

-context オプションは、JPublisher で使用でき、ユーザー定義オブジェクト型と PL/SQL パッケージ用の .sqlj ラッパーについて宣言できる接続コンテキスト・クラスを制御します。

-context=DefaultContext 設定はデフォルトで、JPublisher で生成された .sqlj ソース・ファイルでは、すべての接続コンテキストに SQLJ のデフォルト接続コンテキスト・クラス sqlj.runtime.ref.DefaultContext が使用されます。

または、標準 sqlj.runtime.ConnectionContext インタフェースを実装する、CLASSPATH に存在するクラスを指定できます。指定したクラスがすべての接続コンテキストに使用されます。

注意： ユーザー指定のクラス設定を使用する場合は、そのクラスのインスタンスを getConnectionContext() メソッドからの出力または setConnectionContext() メソッドへの入力に使用する必要があります。この 2 つのメソッドの詳細は、2-26 ページの「接続コンテキストおよび接続インスタンスの使用時の考慮点」を参照してください。

-context=generated 設定では、JPublisher で生成されたすべての .sqlj ファイルに次のインナー・クラス宣言が含まれます。

```
#sql static context _Ctx;
```

これは、各 PL/SQL パッケージと各オブジェクト型のラッパーで、独自の SQLJ 接続コンテキスト・クラスを使用することを意味します。（2-26 ページの「JPublisher により生成された SQLJ コードでの接続コンテキストおよびインスタンスの使用」も参照してください）。

`DefaultContext` 設定またはユーザー指定クラスの設定を使用すると、次のような利点があることに注意してください。

- 追加のコンテキスト・クラスは生成されません。
- `.sqlj` ファイルの変換とコンパイルを別々のステップで実行（SQLJ `-compile=false` 設定で変換）する場合に、大きな柔軟性が得られます。JDK 1.2 固有の型（`java.sql.BLOB`、`CLOB`、`Struct`、`Ref` または `Array` など）を使用しない場合、生成された `.java` ファイルは JDK 1.1.x または JDK 1.2.x 以上でコンパイルできます。ただし、`-context=generated` 設定を使用する場合は該当しません。これは、JDK 1.1.x での SQLJ 接続コンテキストはオブジェクト型のマップに `java.util.Dictionary` インスタンスを使用しますが、JDK 1.2 以上の SQLJ 接続コンテキストは `java.util.Map` インスタンスを使用するためです。

ただし、`generated` 設定を使用する利点は、SQLJ Translator によるオンライン・チェックの実行方法を全面的に制御できることです。特に、各オブジェクト型と各 PL/SQL パッケージを、各自のサンプル・データベース・スキーマに対してチェックできます。ただし、JPublisher では既存のスキーマから `.sqlj` ファイルが生成されるため、生成されたコードが適切かどうかは、そのスキーマからの構成を通じてすでに検証済みです。

ユーザー指定クラスの設定を使用すると、`generated` を柔軟に設定できる一方、`DefaultContext` 設定の利点も得られます。

コマンドラインまたはプロパティ・ファイルで `-context` オプションを指定できます。

SQLJ 接続コンテキストの概要は、『Oracle9*i* SQLJ 開発者ガイドおよびリファレンス』を参照してください。

JPublisher のデフォルトの型マップ（-defaulttypemap）

`-defaulttypemap=[<list_of_typemap_entries>]`

このオプションは、JPublisher 内部で事前定義済みの型マップ・エントリの設定に使用されます。この型マップ・エントリは、`-addtypemap` または `-typemap` で指定するユーザー型マップ・エントリとは区別されます。デフォルトの型マップを消去する場合は、次のオプション設定を使用できます。

```
-defaulttypemap=
```

注意： デフォルトの型マップとユーザー型マップの競合回避の詳細は、2-17 ページの「JPublisher のデフォルトの型マップとユーザー型マップ」を参照してください。この項では、デフォルトの型マップの初期の内容についても説明します。

生成されたファイルの出力ディレクトリ（-dir）

```
-dir=<directory name>
```

空でない -dir オプション設定では、JPublisher で Java ソース・ファイルおよび SQLJ ソース・ファイルを配置するディレクトリ・ツリーのルートを指定します。JPublisher で生成されたパッケージはこのディレクトリでネストされます。「.」（ピリオドまたはドット）を設定すると、カレント・ディレクトリがディレクトリ・ツリーのルートとして指定されます。

ただし、空の設定では、生成されたファイルはすべてカレント・ディレクトリに直接インストールされ、この場合、階層はなくなります。これはデフォルト設定ですが、次のように明示的に指定することもできます。

```
-dir=
```

設定に値を指定すると、JPublisher ではそのディレクトリ、-package オプションを使用して指定したパッケージ名、および INPUT ファイルで SQL 文に組み込まれたパッケージ名を組み合せて、.java または .sqlj ファイルを生成する特定のディレクトリが決定されます。詳細は、3-21 ページの「生成されたパッケージの名前（-package）」を参照してください。

次のコマンドラインの例を考えます（ここでは折り返されていますが、1 行のコマンドラインです）。

```
jpub -user=scott/tiger -input=demoin -mapping=oracle -case=lower -sql=employee
-package=corp -dir=demo
```

demo ディレクトリは、JPublisher で INPUT ファイル demoin で指定されたオブジェクト型用に生成されるパッケージのベース・ディレクトリになります。

コマンドラインまたはプロパティ・ファイルで -dir を指定できます。-dir オプションのデフォルト値は空です。

データベース接続用の JDBC ドライバ（-driver）

```
-driver=<driver_class_name>
```

-driver オプションは、JPublisher でデータベースとの JDBC 接続に使用するドライバのクラスを指定します。デフォルトは次のとおりです。

```
-driver=oracle.jdbc.OracleDriver
```

この設定はあらゆる Oracle JDBC ドライバに適合します。

Java の文字エンコーディング（-encoding）

-encoding=<*name_of_character_encoding*>

-encoding オプションは、JPublisher で読み込む INPUT ファイルと、JPublisher で書き込む .sqlj ファイルおよび .java ファイルの Java における文字のエンコーディングを指定します。デフォルトのエンコーディングはシステム・プロパティの file.encoding の値か、このプロパティが設定されていない場合は 8859_1（ISO Latin-1）になります。

通常、SQLJ および Java コンパイラをコールするときに -encoding オプションを指定しないかぎり、このオプションを指定する必要はありません。この場合、JPublisher で同じ -encoding オプションを使用する必要があります。

-encoding オプションを使用して、Java 環境でサポートされる文字のコードを指定できます。Sun 社の JDK を使用している場合、これらのオプションは次の URL にある native2ascii ドキュメントにリストされています。

http://www.javasoft.com/products/jdk/1.2/docs/tooldocs/solaris/native2ascii.html

または

http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/native2ascii.html

---

注意： エンコーディング設定は、JPublisher の -encoding オプションと Java の file.encoding 設定のどちらを使用して設定する場合も、-props オプションで指定したファイルなどの Java プロパティ・ファイルには適用しないでください。プロパティ・ファイルでは、常にエンコーディング 8859_1 が使用されます。これは、JPublisher 固有の機能ではなく Java 全般の機能です。ただし、プロパティ・ファイルでは Unicode エスケープ・シーケンスを使用できます。

ユーザー・サブクラスの生成（-gensubclass）

-gensubclass={**true**|false|force|call-super}

-gensubclass オプションの値によって、JPublisher でユーザー指定のサブクラスの初期ソース・ファイルが生成されるかどうかと、生成されるサブクラスのフォーマットが決定されます。

-gensubclass=true（デフォルト）に設定すると、ユーザー・サブクラスのソース・ファイル（.java または .sqlj）が存在しないことが検出された場合にのみ、サブクラス用のコードが生成されます。

-gensubclass=false に設定すると、JPublisher ではユーザー・サブクラス用のコードは生成されません。

-gensubclass=force に設定すると、常にユーザー・サブクラス用のコードが生成されます。対応する .java または .sqlj ファイルに既存のコードが存在する場合は、生成されたコードによって上書きされます。この設定は慎重に使用してください。

設定 -gensubclass=call-super には -gensubclass=true に設定したのと同じ効果がありますが、JPublisher で生成されるコードはやや異なります。デフォルトでは、ORAData インタフェースなどの実装に必要なコンストラクタとメソッドのみが生成されます。スーパークラスのメソッドや属性の setter メソッドと getter メソッドのコール方法は示されますが、このコードがコメントに挿入されることはありません。call-super 設定では、メソッド、getter および setter がすべてコードとして生成されます。そこで、クラスのイントロスペクションに基づく Java 開発ツールを使用している場合に、この設定を指定できます。通常、必要となるのは SQL オブジェクト属性と SQL オブジェクト・メソッドに関連するメソッドのみですが、JPublisher の実装の詳細は隠されたままにする必要があります。この場合は、生成されたユーザー・サブクラスでツールを指すことができます。

コマンドラインまたはプロパティ・ファイルで -gensubclass オプションを指定できます。

変換対象のオブジェクトとパッケージの名前を含むファイル（-input）

```
-input=<filename>
-i <filename>
```

この 2 つの書式はシノニムです。2 番目の書式は、コマンドラインの短縮形として使用できるように用意されています。

-input オプションは、JPublisher で読み込む変換対象のオブジェクト型と PL/SQL パッケージの名前を含むファイル名、およびその変換に必要な他の情報を指定します。JPublisher ではリスト内の各オブジェクト型およびパッケージが変換されます。INPUT ファイルは、Java クラス定義を必要とする型を示す型宣言の Make ファイルとみなすことができます。

JPublisher では、INPUT ファイルにないクラスの変換が必要な場合があります。これは、変換実行前に INPUT ファイル内の型の依存関係が分析されて、必要に応じて他の型が変換されるためです。このトピックの詳細は、3-35 ページの「追加の型変換」を参照してください。

INPUT ファイルまたはコマンドラインでパッケージもオブジェクト型も指定していない場合、JPublisher は、接続先のデータベース・スキーマで宣言されているすべてのオブジェクト型およびパッケージを変換します。

INPUT ファイル構文の詳細は、3-32 ページの「INPUT ファイルの構造および構文」を参照してください。

パッケージのクラスおよびラッパー・メソッドの生成（-methods）

```
-methods={true|all|always|named|some|false|none}
```

JPublisher でオブジェクト型および PL/SQL パッケージのメソッドにラッパー・メソッドを生成するかどうかは、-methods オプションの値で決まります。

-methods=true または等価の -methods=all（デフォルト）の場合、JPublisher では処理対象のオブジェクト型および PL/SQL パッケージのすべてのメソッドにラッパー・メソッドが生成されます。Oracle9*i* では、これにより、基礎となる SQL オブジェクトまたはパッケージで実際にメソッドが定義されている場合は、常に .sqlj ソース・ファイルが生成されますが、メソッドが定義されていない場合は .java ソースが生成されます。（以前のリリースでは、設定が TRUE でも all でも、常に .sqlj ソース・ファイルが生成されました。）

-methods=always 設定でもラッパー・メソッドは生成されますが、以前のバージョンの JPublisher への下位互換性を保つために、型でメソッドが定義されているかどうかに関係なく、すべての SQL オブジェクト型について常に .sqlj ファイルが生成されます。

-methods=named または等価の -methods=some の場合、JPublisher により INPUT ファイルで明示的に指定されたメソッドのみにラッパー・メソッドが生成されます。

-methods=false または等価の -methods=none の場合、JPublisher でラッパー・メソッドは生成されません。この場合、ラッパー・メソッドがなければ役に立たないため、PL/SQL パッケージのクラスも生成されません。

デフォルトは -methods=all です。

コマンドラインまたはプロパティ・ファイルで -methods オプションを指定できます。

生成された名前からのスキーマ名の省略（-omit_schema_names）

```
-omit_schema_names
```

-omit_schema_names の指定によって、JPublisher で生成された特定のオブジェクト型の名前にスキーマ名が含まれるかどうかが決定されます。スキーマ名を省略すると、JPublisher のコール時に使用したスキーマ以外のスキーマに接続するときに、使用しているオブジェクト型とパッケージがその 2 つのスキーマで同一に宣言されている場合に限り、JPublisher で生成されたクラスを使用できます。

JPublisher で生成された ORAData および SQLData クラスには、生成されたクラスと一致する SQL オブジェクト型を指定する static final String が含まれます。JPublisher で生成されたコードを実行すると、生成されたコードのオブジェクト型の名前を使用して、データベース内のオブジェクト型が検索されます。オブジェクト型の名前にスキーマ名が含まれない場合は、JPublisher で生成されたコードが実行されるときの接続と対応付けられたスキーマ内で型が検索されます。オブジェクト型にスキーマ名が含まれている場合は、そのスキーマ内で型が検索されます。

-omit_schema_names を指定すると、JPublisher で生成されたすべてのオブジェクト型またはラッパー名がスキーマ名で修飾されます。

-omit_schema_names を指定しないと、次の場合に限り JPublisher で生成されたオブジェクト型またはラッパー名がスキーマ名で修飾されます。

- オブジェクト型またはラッパーを JPublisher の接続先のスキーマ以外のスキーマで宣言した場合

または

- コマンドラインまたは INPUT ファイルでスキーマ名を指定してオブジェクト型またはラッパーを宣言した場合

つまり、別のスキーマからのオブジェクト型またはラッパーにはそれを識別するためのスキーマ名が必要で、コマンドラインまたは INPUT ファイルで型またはパッケージにスキーマ名を使用すると -omit_schema_names オプションがオーバーライドされます。

注意： このオプションはブール型オプションとして動作しますが、Oracle9*i* リリース 2（9.2）では =true または =false には設定できません。使用可能にする場合は -omit_schema_names を指定します。未指定の場合は、使用禁止になります。

生成されたパッケージの名前（-package）

-package=<*package_name*>

-package オプションでは、JPublisher で生成されるパッケージの名前を指定します。パッケージの名前は .java または .sqlj ファイルごとのパッケージ宣言に示されます。ディレクトリ構造もパッケージ名に影響を与えます。INPUT ファイルにある明示的な名前は、-sql オプションの後に、-package オプションに指定された値をオーバーライドします。

例 1 コマンドラインに次の行が含まれているとします。

```
-dir=/a/b -package=c.d -case=mixed
```

また、INPUT ファイルに次の行が含まれている（さらに、SQL 型 PERSON でメソッドが定義されている）とします。

```
SQL PERSON AS Person
```

この場合、次の例では、JPublisher で /a/b/c/d/Person.sqlj ファイルが作成されます。

```
-sql=PERSON:Person
-sql=PERSON
SQL PERSON AS Person
SQL PERSON
```

Person.sqlj ファイルには、次のパッケージ宣言が他の内容とともに含まれます。

```
package c.d;
```

例 2 コマンドラインに次の行が含まれているとします。

```
-dir=/a/b -package=c.d -case=mixed
```

その後に次の行を含む INPUT ファイルが指定されているとします。

```
-sql=PERSON:e.f.Person
SQL PERSON AS e.f.Person
```

この場合は、INPUT ファイル内のパッケージ情報でコマンドラインの -package オプションがオーバーライドされます。JPublisher では、次のパッケージ宣言を含むファイル a/b/e/f/Person.sqlj が作成されます。

```
package e.f;
```

この項で説明したいずれかの方法でクラスにパッケージ名を指定しない場合、JPublisher ではクラスを含むパッケージの名前が提供されません。また、JPublisher ではパッケージ宣言は生成されず、クラスの宣言を含むファイルは -dir オプションで指定されたディレクトリに置かれます。

JPublisher では、変換を必要とする別の型で使用されている型があるため、INPUT ファイルに明示的にリストされていない型を変換することが必要な場合もあります。この場合、要求された型を宣言するファイルはコマンドライン、プロパティ・ファイルまたは INPUT ファイルで指定したデフォルト・パッケージ内に置かれます。JPublisher では、パッケージは他のパッケージに依存しないため、未指定のパッケージは変換されません。

生成された PL/SQL ラッパー・コード用のファイル（-plsqlfile）

-plsqlfile=<*name_of_file_for_generated_PLSQL_code*>

-plsqlfile オプションでは、JPublisher によって PL/SQL ラッパーのストアド・プロシージャおよびファンクションが書き込まれるファイルの名前を指定します。このファイルが存在している場合は、暗黙的に上書きされます。デフォルトでは、PL/SQL コードはファイル plsql_wrapper.sql に書き込まれます。

また、生成されたファイルは（SQL*Plus などを使用して）データベースにロードする必要があることに注意してください。

PL/SQL ラッパー・コードの生成（-plsqlmap）

```
-plsqlmap={true|false|always}
```

-plsqlmap オプションでは、PL/SQL ラッパー・プロシージャおよびファンクションの生成方法を指定します。

このオプションを true（デフォルト）に設定すると、必要に応じて PL/SQL ラッパー・プロシージャおよびファンクションが生成され、可能な場合は変換ファンクションのみが使用されます。

このオプションを false に設定すると、PL/SQL ラッパー・プロシージャやファンクションは生成されません。シグネチャ内で、変換ファンクションのみではサポートできない（つまり、PL/SQL ラッパーの生成を必要とする）PL/SQL 型が見つかると、そのプロシージャまたはファンクション用の Java コードは生成されません。

always に設定すると、PL/SQL 型を使用するストアド・プロシージャまたはファンクションごとに、PL/SQL ラッパー・プロシージャまたはファンクションが生成されます。このオプションは、オリジナルの PL/SQL パッケージを補完するプロキシ PL/SQL パッケージの生成に役立ちます。プロキシは、オリジナル・パッケージ内の JDBC または SQLJ からは直接アクセスできないファンクションやプロシージャについて、Java でアクセス可能なシグネチャを提供します。

生成された PL/SQL ラッパー・コード用のパッケージ（-plsqlpackage）

```
-plsqlpackage=<name_of_PLSQL_package_to_hold_generated_PLSQL_code>
```

-plsqlpackage オプションでは、JPublisher によって生成された PL/SQL ラッパーのストアド・プロシージャおよびファンクションが格納される PL/SQL パッケージの名前を指定します。デフォルトでは、パッケージ JPUB_PLSQL_WRAPPER が使用されます。

このパッケージは、JPublisher によって生成された SQL スクリプトを実行してデータベースに作成する必要があることに注意してください。3-22 ページの「生成された PL/SQL ラッパー・コード用のファイル（-plsqlfile）」を参照してください。

入力プロパティ・ファイル（-props）

```
-props=<filename>
-p <filename>
```

この 2 つの書式はシノニムです。2 番目の書式は、コマンドラインの短縮形として使用できるように用意されています。

-props オプションをコマンドラインで入力し、一般に使用されるオプションの値をリストする JPublisher プロパティ・ファイルの名前を指定します。JPublisher では、プロパティ・ファイルの内容がその時点でコマンドラインから順序どおり挿入されたかのように処理されます。

複数のプロパティ・ファイルがコマンドラインにある場合、JPublisher では他のコマンドライン・オプションとともに、それらのプロパティ・ファイルの出現順に処理されます。

プロパティ・ファイルの内容の詳細は、3-30 ページの「プロパティ・ファイルの構造および構文」を参照してください。

注意： エンコーディング設定は、JPublisher の `-encoding` オプションと Java の `file.encoding` 設定のいずれを使用して設定する場合も、JPublisher の `-props` オプションで指定したファイルなどの、Java プロパティ・ファイルには適用しないでください。プロパティ・ファイルでは、常にエンコーディング `8859_1` が使用されます。これは、JPublisher 固有の機能ではなく Java 全般の機能です。ただし、プロパティ・ファイルでは Unicode エスケープ・シーケンスを使用できます。

生成されたオブジェクト・ラッパーのシリアライズ可能性（-serializable）

`-serializable={true|`**`false`**`}`

ブール型オプション `-serializable` では、JPublisher によって SQL オブジェクト型用に生成される Java クラスで `java.io.Serializable` インタフェースを実装するかどうかを指定します。デフォルト設定は `-serializable=false` です。`-serializable=true` に設定するように選択する場合は、次のことに注意してください。

- すべてのオブジェクト属性がシリアル化可能ではありません。特に、`oracle.sql.BLOB`、`oracle.sql.CLOB` または `oracle.sql.BFILE` などの Oracle LOB 型は、いずれもシリアライズできません。このような属性を持つオブジェクトをシリアライズすると、対応する属性値はデシリアライズ後に `null` に初期化されます。
- `java.sql.Blob` 型または `java.sql.Clob` 型のオブジェクト属性を使用する場合、JPublisher で生成されるコードでは Oracle JDBC 行セット実装を CLASSPATH で使用可能にする必要があります。これは、`[Oracle Home]/jdbc/lib` にある `ocrs12.jar` ライブラリに用意されています。この場合、`Clob` および `Blob` オブジェクトの基礎となる値は実体化され、シリアライズされてから取り出されます。
- オブジェクト参照である属性を含むオブジェクトをデシリアライズするたびに、基礎となる接続に負荷がかかり、参照に対する `setValue()` または `getValue()` コールを発行できません。このため、`-serializable=true` を指定すると、JPublisher で Java クラスに次のメソッドが生成されます。

  ```
  public void restoreConnection(Connection)
  ```

  デシリアライズ後に、このメソッドを特定のオブジェクト参照またはオブジェクトに対して 1 度コールし、参照への現行の接続、または一時的に埋め込まれたすべての参照への現行の接続をそれぞれリストアします。

変換対象のオブジェクト型およびパッケージの宣言（-sql）

```
-sql={toplevel|object type and package translation syntax}
-s {toplevel|object type and package translation syntax}
```

この 2 つの書式はシノニムです。2 番目の書式は、コマンドラインの短縮形として使用できるように用意されています。

INPUT ファイルを汎用的に使用する必要がない場合は、-sql オプションを使用できます。-sql オプションを使用すると、JPublisher で変換する、SQL で宣言された 1 つ以上のデータベース・エンティティをリストできます。（または、同じコマンドラインに複数の -sql オプションを指定する方法や、プロパティ・ファイル内で複数の jpub.sql オプションを指定する方法があります。）現在、JPublisher ではオブジェクト型とパッケージの変換がサポートされています。また、PL/SQL パッケージでサブプログラムが変換されるのと同じように、スキーマでパッケージに属さないストアド・プロシージャやストアド・ファンクションが変換されます。

同じ -sql 宣言内では、オブジェクト型の名前とパッケージ名を混在できます。JPublisher では、それぞれの項目がオブジェクト型かパッケージであるかを検出できます。

また、キーワード toplevel 付きの -sql オプションを使用して、スキーマ内のすべてのパッケージに属さない PL/SQL プロシージャ・ファンクションを変換できます。toplevel キーワードは大文字小文字を区別しません。toplevel キーワードの詳細は後述します。

INPUT ファイルまたはコマンドラインに変換対象の型やパッケージを入力しない場合、JPublisher では接続先のスキーマ内にあるすべての型およびパッケージが変換されます。

この項では、-sql オプションを等価の INPUT ファイル構文に変換して説明します。INPUT ファイル構文は、3-32 ページの「変換文について」を参照してください。

-sql の JPublisher コマンドライン構文では、次の 3 つの型変換を指定できます。

- `-sql=name_a`

  JPublisher ではこの構文は次のように解析されます。

  ```
  SQL name_a
  ```

- `-sql=name_a:name_c`

  JPublisher ではこの構文は次のように解析されます。

  ```
  SQL name_a AS name_c
  ```

- `-sql=name_a:name_b:name_c`

  JPublisher ではこの構文は次のように解析されます。

  ```
  SQL name_a GENERATE name_b AS name_c
  ```

  この場合、*name_a* はオブジェクト型を表す必要があります。

重要： JPublisher のコマンドラインでサポートされるのは、大 / 小文字区別のない SQL 名のみです。SQL でユーザー定義型が大 / 小文字を区別して引用符で囲んで定義されている場合は、その名前をコマンドラインではなく JPublisher の INPUT ファイルに指定する必要があります。詳細は、3-32 ページの「INPUT ファイルの構造および構文」を参照してください。

注意： *name_a*:*name_b*:*name_c* 変換構文は、*name_a* がパッケージを表す場合は無効です。

「-sql=」と入力した後に、JPublisher で変換する 1 つ以上のオブジェクト型およびパッケージ（トップレベルのパッケージを含む）を入力します。複数の変換項目を入力する場合は、それらの項目をカンマで区切る必要があります。空白は使用できません。次の例では、CORPORATION をパッケージ、EMPLOYEE および ADDRESS をオブジェクト型と想定します。

```
-sql=CORPORATION,EMPLOYEE:oracleEmployee,ADDRESS:JAddress:MyAddress
```

JPublisher では次のように解析されます。

```
SQL CORPORATION
SQL EMPLOYEE AS oracleEmployee
SQL ADDRESS GENERATE JAddress AS MyAddress
```

JPublisher では次の操作が実行されます。

- CORPORATION パッケージにラッパーが作成されます。
- オブジェクト型 EMPLOYEE が oracleEmployee として変換されます。
- ADDRESS が JAddress として変換され、ADDRESS オブジェクトを JAddress を拡張するために記述する MyAddress クラスで表すためのコードが生成されます。
- JAddress を拡張するために記述する MyAddress クラスへの参照が作成されます。

JPublisher で接続先スキーマにあるトップレベルの PL/SQL サブプログラムをすべて変換する場合は、-sql オプションに続けてキーワード toplevel を入力します。JPublisher では、トップレベルの PL/SQL サブプログラムがパッケージ内にあるかのように扱われます。たとえば、次のようにします。

```
-sql=toplevel
```

JPublisher ではパッケージに属さないプロシージャやファンクション用に toplevel というラッパー・クラスが生成されます。クラスを別の名前で生成する場合は、-sql=*name_a*:*name_b* 構文を使用して名前を宣言できます。たとえば、次のようにします。

```
-sql=toplevel:myClass
```

これは INPUT ファイル構文のシノニムであることに注意してください。

```
SQL toplevel AS myClass
```

同様に、JPublisher で他のスキーマにあるトップレベルの PL/SQL サブプログラムをすべて変換する場合は、次のように入力します。

```
-sql=<schema_name>.toplevel
```

この例で、<*schema_name*> は、トップレベルのサブプログラムを含むスキーマの名前です。

トップレベルのサブプログラムの生成を要求する場合は、名前のリストも指定できます。その場合は、リストに指定したトップレベルのファンクションまたはプロシージャ用のコードのみが生成されます。名前のリストの前に TOPLEVEL トークンを付けて **(...)** で囲み、ファンクション名は **+**（プラス記号）で区切る必要があります。次の例を考えます。

```
-sql=toplevel(BOOL2INT+INT2BOOL):Conversions
```

リストに指定するファンクション名とプロシージャ名には、大 / 小文字区別があります。大 / 小文字区別なしに定義されている場合は、大文字で指定する必要があります。また、このオプションを使用する場合、オペレーティング・システム・シェルによっては、JPublisher のコマンドラインで引用符で囲む必要があるため注意してください。

オブジェクト・ラッパーの toString() メソッドの生成（-tostring）

```
-tostring={true|false}
```

ブール型オプション -tostring を使用すると、オブジェクト値の出力用に toString() メソッドを追加生成するように、JPublisher に対して指定できます。出力は、オブジェクトの構成に使用する SQL コードに似ています。デフォルト設定は false です。

JPublisher の型マップの置換（-typemap）

```
-typemap=[<list_of_typemap_entries>]
```

-addtypemap オプションと -typemap オプションには、-addtypemap ではユーザー型マップにエントリを追加するのに対して、-typemap では既存の型マップを指定したエントリで置き換えるという違いがあります。したがって、ユーザー型マップを消去する場合は、次のオプション設定を使用できます。

```
-typemap=
```

このオプションでは、デフォルトの型マップの内容は消去されないため注意してください。デフォルトの型マップは、ユーザー型マップに関係なく -defaulttypemap および -adddefaulttypemap オプションで制御されます。型マップ・エントリの書式は、3-13 ページの「ユーザー型マップの追加エントリ（-addtypemap）」を参照してください。

注意： デフォルトの型マップとユーザー型マップの競合回避の詳細は、2-17 ページの「JPublisher のデフォルトの型マップとユーザー型マップ」を参照してください。

変換対象のオブジェクト型の宣言（-types）

`-types=<type_translation_syntax>`

注意： `-types` オプションは互換性を保つために現在サポートされていますが、他の指定の方が優先します。かわりに `-sql` オプションを使用してください。

INPUT ファイルを汎用的に使用する必要がない場合は、オブジェクト型にのみ `-types` オプションを使用できます。`-types` オプションを使用すると、JPublisher で変換する 1 つ以上の個別オブジェクト型をリストできます。`-types` オプションは PL/SQL パッケージをサポートしない点を除いては、`-sql` オプションと同一です。

INPUT ファイルまたは `-types` か `-sql` オプションで、型またはパッケージを入力せずに変換すると、JPublisher では接続先のスキーマ内にあるすべての型およびパッケージが変換されます。

コマンドライン構文では、次の 3 つの型変換を指定できます。

- `-types=name_a`

  JPublisher ではこの構文は次のように解析されます。

  ```
  TYPE name_a
  ```

- `-types=name_a:name_b`

  JPublisher ではこの構文は次のように解析されます。

  ```
  TYPE name_a AS name_b
  ```

- `-types=name_a:name_b:name_c`

  JPublisher ではこの構文は次のように解析されます。

  ```
  TYPE name_a GENERATE name_b AS name_c
  ```

TYPE、TYPE...AS および TYPE...GENERATE...AS 構文の機能は、SQL、SQL...AS および SQL...GENERATE...AS 構文と同じです。3-32 ページの「変換文について」を参照してください。

コマンドラインで「-types=」を入力した後、JPublisher で実行する 1 つ以上のオブジェクト型変換を入力します。複数項目を入力する場合は、それらの項目をカンマで区切る必要があります。空白は使用できません。たとえば、次のように入力すると

```
-types=CORPORATION,EMPLOYEE:oracleEmployee,ADDRESS:JAddress:MyAddress
```

JPublisher では次のように解析されます。

```
TYPE CORPORATION
TYPE EMPLOYEE AS oracleEmployee
TYPE ADDRESS GENERATE JAddress AS MyAddress
```

ターゲット・データベースへの接続 URL（-url）

```
-url=<url>
```

-url オプションを使用して、接続先のデータベースの URL を指定できます。デフォルトは次のとおりです。

```
-url=jdbc:oracle:oci:@
```

「@」記号の後に Oracle SID を指定できます。

Thin ドライバを指定する場合は、次のように入力します。

```
-url=jdbc:oracle:thin:@host:port:sid
```

この例で、*host* はデータベースを実行するホストの名前、*port* はポート番号および *sid* は Oracle SID です。

> **注意：** Oracle9*i* の場合、新規コードでは Oracle JDBC OCI ドライバ用の接続文字列に「oci」を使用します。ただし、下位互換性のために「oci8」も引き続き受け入れられます。（また、Oracle7 リリース 7.3.4 の場合は「oci7」が受け入れられます。）

データベース接続用のユーザー名とパスワード（-user）

```
-user=<username/password>
-u <username/password>
```

この 2 つの書式はシノニムです。2 番目の書式は、コマンドラインの短縮形として使用できるように用意されています。

JPublisher には、データベースに接続できるように、Oracle ユーザー名とパスワードを指定する -user オプションが必要です。-user オプションを入力しない場合、JPublisher ではエラー・メッセージが発行されて実行が停止します。

たとえば、次のコマンドラインではユーザー名 scott とパスワード tiger を使用してデータベースに接続するよう JPublisher に指示します。

```
jpub -user=scott/tiger -input=demoin -dir=demo -mapping=oracle -package=corp
```

JPublisher 入力ファイル

次の項では、JPublisher の入力ファイルの構造および内容を説明します。

- プロパティ・ファイルの構造および構文
- INPUT ファイルの構造および構文
- INPUT ファイルの事前注意事項

プロパティ・ファイルの構造および構文

プロパティ・ファイルは、頻繁に使用するオプションを指定できるオプションのテキスト・ファイルです。JPublisher コマンドラインで -props オプションを使用して、プロパティ・ファイルの名前を指定します。（また、プロパティ・ファイル内で指定できないオプションは、-props のみです。）

プロパティ・ファイルでは、各行に 1 つのオプションを、それに対応する値を指定して入力します。各オプション名に次の接頭辞を付けて入力します（ピリオドも入力してください）。この場合、大 / 小文字区別があります。

```
jpub.
```

空白を使用できるのは jpub. の直前のみで、オプション行の他の位置には空白を使用できません。

また、JPublisher では、SQL 行コメントの構文に似た次の接頭辞を使用してオプションを指定できます。

```
-- jpub.
```

前述のどちらかの接頭辞で始まらない行は、単に JPublisher で無視されます。

また、行の継続を使用すると、プロパティ・ファイル内で複数行にまたがる JPublisher オプションを指定できます。継続行の末尾には、行テキストの直後に ¥（円記号）を付ける必要があります。次行の先頭の空白や --（SQL コメント指定）は無視されます。次のサンプル・エントリを考えます。

```
/* The next three lines represent a JPublisher option
   jpub.sql=SQL_TYPE:JPubJavaType:MyJavaType,¥
            OTHER_SQL_TYPE:OtherJPubType:MyOtherJavaType,¥
            LAST_SQL_TYPE:My:LastType
*/
-- The next two lines represent another JPublisher option
```

```
-- jpub.addtypemap=PLSQL_TYPE:JavaType:SQL TYPE¥
--                :SQL_TO_PLSQL_FUNCTION:PLSQL_TO_SQL_FUNCTION
```

この機能により、SQL スクリプトに JPublisher オプションを簡単に埋め込むことができます。これは、PL/SQL から SQL への型マッピングを設定するときに役立ちます。

JPublisher では、-props オプションが指定された場所で、プロパティ・ファイルの内容がコマンドラインから挿入されたかのように、オプションが順番に読み込まれます。1 つのオプションを 2 度以上指定すると、最後に指定した値によって前の値がオーバーライドされます。ただし、次のオプションの場合は値が累積されます。

- jpub.sql
- jpub.type
- jpub.addtypemap
- jpub.adddefaulttypemap

次のコマンドラインの例を考えます（ここでは折り返されていますが、1 行のコマンドラインです）。

```
jpub -user=scott/tiger -sql=employee -mapping=oracle -case=lower -package=corp
-dir=demo
```

これは次の 1 行と同等です。

```
jpub -props=my_properties
```

my_properties が次のように定義されているとします。

```
-- jpub.user=scott¥
--          /tiger
// jpub.user=cannot_use/java_line_comments
jpub.sql=employee
/*
jpub.mapping=oracle
*/
Jpub.notreally=a jpub option
   jpub.case=lower
jpub.package=corp
    jpub.dir=demo
```

この場合は、各オプション名を jpub. 接頭辞（ピリオドも含む）で始める必要があります。オプション名の前に空白または -- 以外の文字列を入力すると、JPublisher で行全体が無視されます。

また、この例は、Jpub.notreally=a jpub option のように jpub. 接頭辞をすべて小文字で指定しないと、無視されることを示しています。

すべての JPublisher オプションは、3-2 ページの「JPublisher オプション」を参照してください。

INPUT ファイルの構造および構文

JPublisher コマンドラインで -input オプションを使用して、INPUT ファイルの名前を指定します。このファイルは、JPublisher で変換するオブジェクト型および PL/SQL パッケージを識別します。また、生成されたクラスおよびパッケージのネーミングも制御します。-sql コマンドライン・オプションを使用してオブジェクト型とパッケージを指定できますが、INPUT ファイルを使用すると JPublisher でのオブジェクト型と PL/SQL パッケージの変換をさらに詳細に制御できます。

INPUT ファイルまたはコマンドラインで型やパッケージを指定しない場合、JPublisher では、接続先のスキーマにあるすべてのオブジェクト型および PL/SQL パッケージが変換されます。

変換文について

INPUT ファイル内の変換文は、JPublisher で変換するオブジェクト型および PL/SQL パッケージの名前を識別します。変換文ではオプションで型またはパッケージの Java 名、属性識別子の Java 名および拡張されたクラスがあるかどうかを指定できます。

1 つ以上の変換文を INPUT ファイルに入れることができます。変換文の構造は次のとおりです。

```
( SQL <name>
| SQL [<schema_name>.]toplevel  [(<name_list>)]
| TYPE <type_name>)
[GENERATE <java_name_1>]
[AS <java_name_2>]
[TRANSLATE
     <database_member_name> AS <simple_java_name>
 { , <database_member_name> AS <simple_java_name>}*
]
```

次の項では、変換文のコンポーネントを説明します。

SQL < 名前 > | TYPE < 型名 > 句 SQL < 名前 > を入力し、JPublisher で変換するオブジェクト型または PL/SQL パッケージを識別します。JPublisher では < 名前 > の検査でオブジェクト型かパッケージ名かが判断され、適切に処理されます。< 名前 > のかわりに、予約語 toplevel を使用すると、JPublisher ではその接続先のスキーマ内のパッケージに属さないストアド・プロシージャやストアド・ファンクションが変換されます。

オブジェクト型のみを指定する場合は、SQL のかわりに TYPE < 型名 > を入力できます。ただし、TYPE 構文は Oracle9*i* では使用不可です。

また、< 名前 > に < スキーマ名 >.< 名前 > を入力すると、オブジェクト型またはパッケージが所属するスキーマを指定できます。< スキーマ名 >.toplevel を入力すると、JPublisher

ではスキーマ < スキーマ名 > のパッケージに属さないストアド・プロシージャやストアド・ファンクションが変換されます。TOPLEVEL とともに、公開する名前のカンマ区切りのリストをカッコで囲み、(*<name_list>*) と指定することもできます。JPublisher では、このリストと一致するトップレベルのファンクションおよびプロシージャのみが考慮されます。このリストを指定しなければ、トップレベルのすべてのサブプログラム用のコードが生成されます。

重要： SQL でユーザー定義型が大 / 小文字区別を使用して（引用符で囲んで）定義されている場合は、名前を引用符で囲んで指定する必要があります。たとえば、次のようにします。

```
SQL "CaseSenstiveType" AS CaseSensitiveType
```

大 / 小文字区別のないスキーマ名を指定する場合は、次のようにします。

```
SQL SCOTT."CaseSensitiveType" AS CaseSensitiveType
```

大 / 小文字区別のあるスキーマ名を指定する場合は、次のようにします。

```
SQL "Scott"."CaseSensitiveType AS CaseSensitiveType
```

AS 句は後述のようにオプションです。

スキーマ名や型名にはドット（.）を使用しないでください。

注意： TYPE 構文は互換性を保つために現在サポートされていますが、他の指定の方が優先します。かわりに SQL 構文を使用してください。

AS <Java 名 _2> 句 この句はオブジェクト型また PL/SQL パッケージを表す Java クラスの名前をオプションで指定します。<Java 名 _2> には任意の正当な Java 名を指定でき、パッケージ識別子を組み込めます。Java 名の大文字小文字の区別により、-case オプションの値がオーバーライドされます。パッケージ命名の詳細は、3-35 ページの「INPUT ファイルでのパッケージ命名規則」を参照してください。

GENERATE 句を使用せずに AS 句を使用すると、AS 句に指定したクラスが JPublisher で生成され、SQL 型にマップされます。

AS 句と GENERATE 句を使用すると、JPublisher では GENERATE 句に指定したクラスが生成されますが、SQL 型は AS 句に指定したクラスにマップされます。JPublisher で生成されたクラスを拡張するクラスを、AS 句を使用して手動で作成します。

2-32 ページの「JPublisher で生成されたクラスの拡張」も参照してください。

GENERATE <Java 名 _1> 句 この句では、マッピング用のサブクラスを作成する場合に、JPublisher で生成されるクラスの名前を指定します。GENERATE 句は AS 句とともに使用します。JPublisher では、GENERATE 句に指定したクラスが生成されます。AS 句では、作成し

て Java プログラムで SQL オブジェクト型の表現に使用されるサブクラスの名前を指定します。

<Java 名 _1> には任意の正当な Java 名を指定でき、パッケージ識別子を組み込めます。その大文字小文字の区別により、-case オプションの値がオーバーライドされます。

GENERATE 句は、オブジェクト型を変換する場合にのみ使用してください。オブジェクト型を変換している場合、JPublisher で生成されたコードは、生成されるクラスの名前と、ユーザーの Java プログラムで SQL オブジェクト型の表現に使用されるクラスの名前の両方を意味します。これらが 2 つの異なる句の場合は、GENERATE...AS を使用してください。

PL/SQL パッケージを変換する場合は、この句を使用しないでください。PL/SQL パッケージを変換している場合は、JPublisher で生成されたコードは、生成されるクラスの名前のみを意味するため、この場合は GENERATE 句を使用する必要はありません。

2-32 ページの「JPublisher で生成されたクラスの拡張」も参照してください。

TRANSLATE < データベース・メンバー名 > AS < 単純 Java 名 > 句 この句は属性またはメソッドに対して別の名前をオプションで指定します。< データベース・メンバー名 > はオブジェクト型の属性名または型かパッケージのメソッドの名前で、次の < 単純 Java 名 > に変換されます。< 単純 Java 名 > には任意の正当な Java 名を指定でき、その大文字小文字の区別により、-case オプションの値がオーバーライドされます。この名前にはパッケージ名を含めません。

TRANSLATE...AS を使用せずに属性またはメソッドを改名する場合、または JPublisher で INPUT ファイルにリストされていないオブジェクト型を変換する場合、JPublisher では -case オプションの値に従って変更された Java 名として、属性またはメソッドのデータベース名が使用されます。属性名またはメソッドを改名するのは次の理由があります。

- 名前に文字、数字およびアンダースコア以外の文字が含まれている場合。
- 名前が Java キーワードと競合する場合。
- 型名が同じ有効範囲の別の名前と競合する場合。これは、プログラムで異なるスキーマからの同じ名前を持つ 2 つの型を使用する場合などに起こります。

属性名は get*XXX*() および set*XXX*() メソッド名に埋め込まれて表されます。そのため、属性名の 1 文字目を大文字にすると見やすくなります。たとえば、次のように入力すると

```
TRANSLATE FIRSTNAME AS FirstName
```

getFirstName() メソッドおよび setFirstName() メソッドが生成されます。それに対し、次のように入力すると、

```
TRANSLATE FIRSTNAME AS firstName
```

getfirstName() メソッドおよび setfirstName() メソッドが生成されます。

注意： Java キーワード null は、次のように属性やメソッドのターゲット Java 名として使用される場合は特別な意味を持ちます。

```
TRANSLATE FIRSTNAME AS null
```

SQL メソッドを null にマップすると、JPublisher ではマップ先の Java クラスには対応する Java メソッドが作成されません。SQL オブジェクトの属性を null にマップすると、JPublisher ではマップ先の Java クラスに属性用の getter メソッドと setter メソッドが作成されません。

INPUT ファイルでのパッケージ命名規則 単純な Java 識別名を使用して、INPUT ファイルのクラスに名前を付ける場合は、その完全なクラス名に -package オプションからのパッケージ名が組み込まれます。INPUT ファイル内のクラス名がパッケージ名で修飾されると、そのパッケージ名により -package オプションの値がオーバーライドされ、クラスの完全なパッケージ名になります。

次の点に注意してください。

- 次の構文を入力します。

  ```
  SQL A AS B
  ```

  JPublisher ではコマンドラインまたはプロパティ・ファイルで -package に入力された値が使用されます。

- 次の構文を入力します。

  ```
  SQL A AS B.C
  ```

  JPublisher では B.C が完全なクラス名を表すものと解析されます。

たとえば、コマンドラインに次のように入力した場合に、

```
-package=a.b
```

INPUT ファイルに次の変換文が含まれていると、

```
SQL scott.employee AS e.Employee
```

JPublisher ではクラスが次のように生成されます。

```
e.Employee
```

パッケージ名の決定に関する他の例は、3-21 ページの「生成されたパッケージの名前（-package）」を参照してください。

追加の型変換 JPublisher では INPUT ファイルにリストされていない追加の型を変換することが必要な場合があります。これは、変換実行前に INPUT ファイル内の型の依存関係が分析されて、必要に応じて他の型が変換されるためです。1-25 ページの「JPublisher 変換の例」

の例を思い出してください。EMPLOYEE のオブジェクト型定義に ADDRESS と呼ばれる属性が含まれており、ADDRESS が次の定義で指定されたオブジェクトの場合、

```
CREATE OR REPLACE TYPE address AS OBJECT
(
    street      VARCHAR2(50),
    city        VARCHAR2(50),
    state       VARCHAR2(30),
    zip         NUMBER
);
```

JPublisher で EMPLOYEE 型を定義する必要があるため、ADDRESS が最初に変換されます。さらに、ADDRESS およびその属性は INPUT ファイルで特に指定されていないため、すべて同じ大文字小文字の区別で変換されます。Address.java のクラス・ファイルが生成され、コマンドラインで指定されたパッケージに組み込まれます。

JPublisher ではユーザーが要求しないパッケージは変換されません。パッケージには属性がないため、他のパッケージへの依存関係はありません。

変換文の例

INPUT ファイルの機能を具体的に示すために、1-25 ページの「JPublisher 変換の例」の例をさらに複雑にしたバージョンを考えます。次のコマンドラインの例を考えます（ここでは折り返されていますが、1 行のコマンドラインです）。

```
jpub -user=scott/tiger -input=demoin -dir=demo -numbertypes=oracle -package=corp
-case=same
```

INPUT ファイル demoin に次のコードが含まれているとします。

```
SQL employee AS c.Employee
  TRANSLATE NAME AS Name
            HIRE_DATE AS HireDate
```

-case=same オプションは、生成された Java 識別名でデータベース内と同じ大文字小文字の区別を保持するように指定します。CREATE TYPE または CREATE PACKAGE 宣言内の識別名は、引用符で囲まれていないかぎりデータベース内に大文字で格納されます。ただし、-case オプションは INPUT ファイルで明示的に指定されていない識別名のみに適用されます。そのため、Employee は記述されたとおりに表されます。特別に指定されていない属性識別名（つまり、EMPNO、DEPTNO および SALARY）は大文字のままですが、JPublisher では特別に指定された NAME および HIRE_DATE 属性は示されるとおりに変換されます。

変換文では、変換対象となる SQL オブジェクト型を指定します。この場合は、オブジェクト型 Employee のみが存在します。

AS c.Employee 句でパッケージ名がより詳細に修飾されます。変換された型は次のファイルに書き込まれます。

```
./demo/corp/c/Employee.sqlj           (UNIX)
.¥demo¥corp¥c¥Employee.sqlj           (Windows NT)
```

（これはオブジェクト型でメソッドが定義されている場合です。それ以外の場合は、かわりに Employee.java が生成されます。）

生成されたファイルは、出力ディレクトリ demo にあるパッケージ corp.c に書き込まれます。パッケージ名はディレクトリ構造に反映されることに注意してください。

TRANSLATE...AS 句は、型の Java クラスへの変換時に指定したオブジェクト属性の名前を変更することを指定します。この場合、NAME 属性は Name に変更され、HIRE_DATE 属性は HireDate に変更されます。

INPUT ファイルの事前注意事項

この項では、INPUT ファイルでの一般的なエラーについて説明します。これらのエラーをチェックした後に JPublisher を実行してください。JPublisher では、INPUT ファイル内で検出されたエラーのほとんどがレポートされますが、次に示すエラーはレポートされません。

異なるオブジェクト型に対する同じ Java クラス名の要求

2 つの異なるオブジェクト型に対して同じ Java クラス名を要求すると、2 番目のクラスにより最初のクラスが暗黙的に上書きされます。たとえば、INPUT ファイルに次の内容が含まれる場合、

```
type PERSON1 as Person
TYPE PERSON2 as Person
```

JPublisher では、PERSON1 用にファイル Person.java が作成され、PERSON2 型用に次に作成された同名のファイルにより上書きされます。

異なるオブジェクト属性に対する同じ属性名の要求

2 つの異なるオブジェクト属性に対して同じ属性名を要求すると、JPublisher では警告メッセージを発行せずに両方の属性に対して get*XXX*() および set*XXX*() メソッドが生成されます。生成されたクラスが Java で有効かどうかは、同じ名前の 2 つの get*XXX*() メソッドおよび同じ名前の 2 つの set*XXX*() メソッドに、明確にオーバーロードできる異なる引数型があるかどうかによります。

存在しない属性の指定

TRANSLATE 句で存在しないオブジェクト属性を指定すると、JPublisher は警告メッセージを発行せずにそのオブジェクト属性を無視します。

INPUT ファイルからの次の例を考えます。

```
type PERSON translate X as attr1
```

X が PERSON の属性ではない場合、JPublisher では警告メッセージが発行されません。

4

JPublisher の例

この章では、オブジェクト型および PL/SQL パッケージを変換する際に JPublisher で生成される出力を、例をあげて説明します。この章の内容は、次のとおりです。

- 「例 : 様々なマッピングを使用する JPublisher 変換」では、JPublisher 出力の例を示して、データ型マッピング・パラメータの値のみを変更した場合の各種出力を比較します。
- 「例 : JPublisher のオブジェクト属性のマッピング」では、様々なオブジェクト型を変換する場合の JPublisher 出力の例を示します。
- 「例 : SQLData クラスの生成」では、SQLData インタフェースを実装するクラスを生成する場合の JPublisher 出力の例を示します。
- 「例 : JPublisher クラスの拡張」では、拡張するクラスを生成する場合の JPublisher 出力の例を示します。
- 「例 : オブジェクトのメソッド用に生成されたラッパー」では、オブジェクト型の属性とメソッド用のメソッド・ラッパーを生成する場合の JPublisher 出力の例を示します。
- 「例 : パッケージのメソッド用に生成されたラッパー」では、PL/SQL メソッド用のメソッド・ラッパーを生成する場合の JPublisher 出力の例を示します。
- 「例 : オブジェクト型用に生成されたクラスの使用」では、JPublisher でオブジェクト型用に生成されるクラスを使用する完全なプログラムを示します。
- 「例 : パッケージ用に生成されたクラスの使用」では、JPublisher でオブジェクトおよびパッケージ用にそれぞれ生成されるクラスおよびメソッド・ラッパーを使用する完全なプログラムを示します。
- 「例 : JDBC でサポートされないデータ型の使用」では、PL/SQL 型に対する JPublisher のサポート、PL/SQL の BOOLEAN 値を使用するオブジェクト型の設定を示します。この例では、JPublisher を介して型を直接公開する方法と、型の変換を手動で記述する方法を比較します。

例 : 様々なマッピングを使用する JPublisher 変換

この項では、データ型マッピング・パラメータの値のみが異なる変換の JPublisher からの出力例を示します。この例では、次の型宣言を使用しています。

```
CREATE TYPE employee AS OBJECT
(
    name       VARCHAR2(30),
    empno      INTEGER,
    deptno     NUMBER,
    hiredate   DATE,
    salary     REAL
);
```

また、次のコマンドライン（折り返されていますが 1 行のコマンドです）を使用していますが、2 つの例では -numbertypes および -builtintypes の設定が異なります。

```
jpub -user=scott/tiger -dir=demo -numbertypes=xxxx -builtintypes=xxxx -package=corp
-case=mixed -sql=Employee
```

次の 2 つの例では、JPublisher は次のデータ型マッピングを使用します。

- 最初の例では、-numbertypes=jdbc および -builtintypes=jdbc を使用します。
- 2 番目の例では、-numbertypes=oracle および -builtintypes=oracle を使用します。

JDBC マッピングを使用する JPublisher 変換

ユーザーが数値型に対して Object JDBC マッピングではなく JDBC マッピングを要求するため、getXXX() および setXXX() アクセッサ・メソッドでは Integer 型のかわりに int 型、および Float 型のかわりに float 型が使用されます。

Employee.java ファイルの内容を次に示します。EmployeeRef.java ファイルは、属性の型に依存しないため変更されません。

> **注意 :** JPublisher で生成されるメソッド本体の詳細は、今後のリリースで変更される場合があります。

```
package corp;

import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sql.ORAData;
import oracle.sql.ORADataFactory;
```

```
import oracle.sql.Datum;
import oracle.sql.STRUCT;
import oracle.jpub.runtime.MutableStruct;

public class Employee implements ORAData, ORADataFactory
{
  public static final String _SQL_NAME = "SCOTT.EMPLOYEE";
  public static final int _SQL_TYPECODE = OracleTypes.STRUCT;

  protected MutableStruct _struct;

  private static int[] _sqlType =  { 12,4,2,91,7 };
  private static ORADataFactory[] _factory = new ORADataFactory[5];
protected static final Employee _EmployeeFactory = new Employee(false);

  public static ORADataFactory getORADataFactory()
  { return _EmployeeFactory; }
  /* constructor */
  protected Employee(boolean init)
  { if(init) _struct = new MutableStruct(new Object[5], _sqlType, _factory); }
  public Employee()
  { this(true); }
  public Employee(String name, int empno, java.math.BigDecimal deptno,
                  java.sql.Timestamp hiredate, float salary) throws SQLException
  { this(true);
    setName(name);
    setEmpno(empno);
    setDeptno(deptno);
    setHiredate(hiredate);
    setSalary(salary);
  }

  /* ORAData interface */
  public Datum toDatum(Connection c) throws SQLException
  {
    return _struct.toDatum(c, _SQL_NAME);
  }


  /* ORADataFactory interface */
  public ORAData create(Datum d, int sqlType) throws SQLException
  { return create(null, d, sqlType); }
  protected ORAData create(Employee o, Datum d, int sqlType) throws SQLException
  {
    if (d == null) return null;
    if (o == null) o = new Employee(false);
    o._struct = new MutableStruct((STRUCT) d, _sqlType, _factory);
```

```
    return o;
  }
  /* accessor methods */
  public String getName() throws SQLException
  { return (String) _struct.getAttribute(0); }

  public void setName(String name) throws SQLException
  { _struct.setAttribute(0, name); }


  public int getEmpno() throws SQLException
  { return ((Integer) _struct.getAttribute(1)).intValue(); }

  public void setEmpno(int empno) throws SQLException
  { _struct.setAttribute(1, new Integer(empno)); }


  public java.math.BigDecimal getDeptno() throws SQLException
  { return (java.math.BigDecimal) _struct.getAttribute(2); }

  public void setDeptno(java.math.BigDecimal deptno) throws SQLException
  { _struct.setAttribute(2, deptno); }


  public java.sql.Timestamp getHiredate() throws SQLException
  { return (java.sql.Timestamp) _struct.getAttribute(3); }

  public void setHiredate(java.sql.Timestamp hiredate) throws SQLException
  { _struct.setAttribute(3, hiredate); }


  public float getSalary() throws SQLException
  { return ((Float) _struct.getAttribute(4)).floatValue(); }

  public void setSalary(float salary) throws SQLException
  { _struct.setAttribute(4, new Float(salary)); }

}
```

Oracle マッピングを使用する JPublisher 変換

ユーザーが Oracle 型マッピングを要求しているため、getXXX() および setXXX() アクセッサ・メソッドでは、String のかわりに oracle.sql.CHAR 型、java.sql.Timestamp のかわりに oracle.sql.DATE 型、java.lang.Integer、java.math.BigDecimal および java.lang.Float のかわりに oracle.sql.NUMBER 型が使用されます。

Employee.java ファイルの内容を次に示します。EmployeeRef.java ファイルは、属性の型に依存しないため変更されません。

> **注意 :** JPublisher で生成されるメソッド本体の詳細は、今後のリリースで変更される場合があります。

```
package corp;

import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sql.ORAData;
import oracle.sql.ORADataFactory;
import oracle.sql.Datum;
import oracle.sql.STRUCT;
import oracle.jpub.runtime.MutableStruct;

public class Employee implements ORAData, ORADataFactory
{
  public static final String _SQL_NAME = "SCOTT.EMPLOYEE";
  public static final int _SQL_TYPECODE = OracleTypes.STRUCT;

  protected MutableStruct _struct;

  private static int[] _sqlType =  { 12,4,2,91,7 };
  private static ORADataFactory[] _factory = new ORADataFactory[5];
protected static final Employee _EmployeeFactory = new Employee(false);

  public static ORADataFactory getORADataFactory()
  { return _EmployeeFactory; }
  /* constructor */
  protected Employee(boolean init)
  { if(init) _struct = new MutableStruct(new Object[5], _sqlType, _factory); }
  public Employee()
  { this(true); }
  public Employee(oracle.sql.CHAR name, oracle.sql.NUMBER empno,
oracle.sql.NUMBER deptno,
```

```
                    oracle.sql.DATE hiredate, oracle.sql.NUMBER salary) throws
SQLException
  { this(true);
    setName(name);
    setEmpno(empno);
    setDeptno(deptno);
    setHiredate(hiredate);
    setSalary(salary);
  }

  /* ORAData interface */
  public Datum toDatum(Connection c) throws SQLException
  {
    return _struct.toDatum(c, _SQL_NAME);
  }


  /* ORADataFactory interface */
  public ORAData create(Datum d, int sqlType) throws SQLException
  { return create(null, d, sqlType); }
  protected ORAData create(Employee o, Datum d, int sqlType) throws SQLException
  {
    if (d == null) return null;
    if (o == null) o = new Employee(false);
    o._struct = new MutableStruct((STRUCT) d, _sqlType, _factory);
    return o;
  }
  /* accessor methods */
  public oracle.sql.CHAR getName() throws SQLException
  { return (oracle.sql.CHAR) _struct.getOracleAttribute(0); }

  public void setName(oracle.sql.CHAR name) throws SQLException
  { _struct.setOracleAttribute(0, name); }


  public oracle.sql.NUMBER getEmpno() throws SQLException
  { return (oracle.sql.NUMBER) _struct.getOracleAttribute(1); }

  public void setEmpno(oracle.sql.NUMBER empno) throws SQLException
  { _struct.setOracleAttribute(1, empno); }


  public oracle.sql.NUMBER getDeptno() throws SQLException
  { return (oracle.sql.NUMBER) _struct.getOracleAttribute(2); }

  public void setDeptno(oracle.sql.NUMBER deptno) throws SQLException
  { _struct.setOracleAttribute(2, deptno); }
```

```
  public oracle.sql.DATE getHiredate() throws SQLException
  { return (oracle.sql.DATE) _struct.getOracleAttribute(3); }

  public void setHiredate(oracle.sql.DATE hiredate) throws SQLException
  { _struct.setOracleAttribute(3, hiredate); }


  public oracle.sql.NUMBER getSalary() throws SQLException
  { return (oracle.sql.NUMBER) _struct.getOracleAttribute(4); }

  public void setSalary(oracle.sql.NUMBER salary) throws SQLException
  { _struct.setOracleAttribute(4, salary); }

}
```

例 : JPublisher のオブジェクト属性のマッピング

この項では、様々なオブジェクト属性の型の JPublisher 出力の例をあげて、JPublisher で作成される各種データ型マッピングを示します。

この例では、オブジェクト型 address を定義した後、VARRAY 型 Addr_Array の定義の基礎としてオブジェクト型 address を使用します。また、オブジェクト型 alltypes の定義でも、JPublisher でオブジェクト参照および配列に対して作成されるマッピングを示すために、オブジェクト型 address および VARRAY 型 Addr_Array を使用します（次に示す alltypes オブジェクト定義の attr17、attr18 および attr19 を参照してください）。

```
CONNECT SCOTT/TIGER;

CREATE OR REPLACE TYPE address AS object
(
  street varchar2(50),
  city   varchar2(50),
  state  varchar2(30),
  zip    number
);

CREATE OR REPLACE TYPE Addr_Array AS varray(10) OF address;
CREATE OR REPLACE TYPE ntbl AS table OF Integer;
CREATE TYPE alltypes AS object (
  attr1  bfile,
  attr2  blob,
  attr3  char(10),
  attr4  clob,
  attr5  date,
```

```
  attr6  decimal,
  attr7  double precision,
  attr8  float,
  attr9  integer,
  attr10 number,
  attr11 numeric,
  attr12 raw(20),
  attr13 real,
  attr14 smallint,
  attr15 varchar(10),
  attr16 varchar2(10),
  attr17 address,
  attr18 ref address,
  attr19 Addr_Array,
  attr20 ntbl);
```

この例では、JPublisher は次のコマンドラインでコールされています（ここでは折り返されていますが、1 行のコマンドラインです）。

```
jpub -user=scott/tiger -input=demoin -dir=demo -package=corp -mapping=objectjdbc
-methods=false
```

注意： -mapping オプションは使用不可になっていますが、引き続きサポートされているため例に示してあります。-mapping=objectjdbc 設定は、-builtintypes=jdbc、-numbertypes=objectjdbc、-lobtypes=oracle および -usertypes=oracle の組合せと同等です。詳細は、3-11 ページの「すべての型のマッピング（-mapping）」を参照してください。

---

demo および corp ディレクトリは、あらかじめ作成しておく必要はありません。このディレクトリは JPublisher で作成されます。

demoin ファイルには、次の宣言が含まれています。

```
SQL ADDRESS AS Address
SQL ALLTYPES AS all.Alltypes
```

demoin で明示的に Alltypes 型および Address 型の宣言がリストされるため、JPublisher でそれらの宣言が生成されます。また、Alltypes 型で ntbl 型および AddrArray 型の宣言が要求されるため、それらの宣言も生成されます。

さらに、オブジェクト型ごとに参照型の宣言も生成されるため、JPublisher で AlltypesRef 型および AddressRef 型の宣言も生成されます。参照型は対応するオブジェクト型と同じパッケージ内にあります。参照型は、INPUT ファイルまたはコマンドラインにリストされません。Address 型および AddressRef 型は、-package=corp がコマンドラ

インにあるため、パッケージ corp 内にあります。all.Alltypes の all により -package=corp がオーバーライドされるため、Alltypes および AlltypesRef 型はパッケージ all にあります。残りの型は明示的に指定されていないため、パッケージ corp に置かれます。

したがって、JPublisher ではパッケージ corp に次のファイルが生成されます。

```
./demo/corp/Address.java
./demo/corp/AddressRef.java
./demo/corp/Ntbl.java
./demo/corp/AddrArray.java
```

また、パッケージ all には次のファイルが生成されます。

```
./demo/all/Alltypes.java
./demo/all/AlltypesRef.java
```

JPublisher で生成された Address.java のリストと説明

ファイル ./demo/corp/Address.java の内容は次のとおりです。

> **注意：** JPublisher で生成されるメソッド本体の詳細は、今後のリリースで変更される場合があります。

```
package corp;

import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sql.ORAData;
import oracle.sql.ORADataFactory;
import oracle.sql.Datum;
import oracle.sql.STRUCT;
import oracle.jpub.runtime.MutableStruct;

public class Address implements ORAData, ORADataFactory
{
  public static final String _SQL_NAME = "SCOTT.ADDRESS";
  public static final int _SQL_TYPECODE = OracleTypes.STRUCT;

  protected MutableStruct _struct;

  private static int[] _sqlType =  { 12,12,12,2 };
  private static ORADataFactory[] _factory = new ORADataFactory[4];
protected static final Address _AddressFactory = new Address(false);
```

```
public static ORADataFactory getORADataFactory()
{ return _AddressFactory; }
/* constructor */
protected Address(boolean init)
{ if(init) _struct = new MutableStruct(new Object[4], _sqlType, _factory); }
public Address()
{ this(true); }
public Address(String street, String city, String state,
               java.math.BigDecimal zip) throws SQLException
{ this(true);
  setStreet(street);
  setCity(city);
  setState(state);
  setZip(zip);
}

/* ORAData interface */
public Datum toDatum(Connection c) throws SQLException
{
  return _struct.toDatum(c, _SQL_NAME);
}


/* ORADataFactory interface */
public ORAData create(Datum d, int sqlType) throws SQLException
{ return create(null, d, sqlType); }
protected ORAData create(Address o, Datum d, int sqlType) throws SQLException
{
  if (d == null) return null;
  if (o == null) o = new Address(false);
  o._struct = new MutableStruct((STRUCT) d, _sqlType, _factory);
  return o;
}
/* accessor methods */
public String getStreet() throws SQLException
{ return (String) _struct.getAttribute(0); }

public void setStreet(String street) throws SQLException
{ _struct.setAttribute(0, street); }


public String getCity() throws SQLException
{ return (String) _struct.getAttribute(1); }

public void setCity(String city) throws SQLException
{ _struct.setAttribute(1, city); }
```

```
  public String getState() throws SQLException
  { return (String) _struct.getAttribute(2); }

  public void setState(String state) throws SQLException
  { _struct.setAttribute(2, state); }


  public java.math.BigDecimal getZip() throws SQLException
  { return (java.math.BigDecimal) _struct.getAttribute(3); }

  public void setZip(java.math.BigDecimal zip) throws SQLException
  { _struct.setAttribute(3, zip); }

}
```

Address.java ファイルには、Java ソース・ファイルに関するいくつかのポイントが示されています。JPublisher で生成されるファイルは、生成されたクラスが名前付きパッケージにある場合、常にパッケージ宣言で開始されます。次のいずれかを使用してパッケージを指定できます。

- コマンドラインまたはプロパティ・ファイルで指定した -package パラメータ。
- INPUT ファイルの AS <*Java_identifier*> 句。*Java_identifier* にはパッケージ名が含まれます。

package 宣言の後に Address クラスで指定された特定のクラスおよびインタフェースのインポート宣言が続きます。

クラス定義が import 宣言の後に続きます。JPublisher で生成されるすべてのクラスが public と宣言されます。

SQLJ では _SQL_NAME および _SQL_TYPECODE 文字列を使用して、Address クラスに一致するオブジェクト型を識別します。

引数のないコンストラクタを使用して、_AddressFactory オブジェクトが作成され、このオブジェクトは getORADataFactory() によって戻されます。効率を良くするため、JPublisher では Address オブジェクト用の protected boolean コンストラクタも生成されます。このコンストラクタを Address のサブクラスで使用して、初期化されていない Address オブジェクトを作成できます。他の Address オブジェクトは、create() メソッドで構成されます。protected create(...,...,...) メソッドを使用して、JPublisher で生成された Address クラス内の JPublisher 実装の詳細がカプセル化され、ユーザー指定サブクラスの記述が簡素化されます。静的な _factory フィールドや _struct フィールドの生成など、実装の詳細は実装固有であり、Address のサブクラスでは参照または使用できません。（この実装では、_factory フィールドは Address の属性のファクトリの配列ですが、この場合、Address の属性の型はいずれもファクトリを必要としないため、ファ

クトリは null です。_struct フィールドはオブジェクトのデータを保持する MutableStruct インスタンスです。）

toDatum() メソッドにより、Address オブジェクトは Datum オブジェクト（この場合は STRUCT オブジェクト）に変換されます。JDBC には接続引数が必要ですが、論理的には必要ない場合があります。

getXXX() および setXXX() アクセッサ・メソッドでは、数値属性に objectjdbc マッピングが使用され、その他の属性には jdbc マッピングが使用されます。-case=mixed がデフォルトであるため、メソッド名は大文字小文字の混合表記になります。

JPublisher で生成された AddressRef.java のリスト

ファイル ./demo/corp/AddressRef.java の内容は次のとおりです。

> **注意：** JPublisher で生成されるメソッド本体の詳細は、今後のリリースで変更される場合があります。

```
package corp;

import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sql.ORAData;
import oracle.sql.ORADataFactory;
import oracle.sql.Datum;
import oracle.sql.REF;
import oracle.sql.STRUCT;

public class AddressRef implements ORAData, ORADataFactory
{
  public static final String _SQL_BASETYPE = "SCOTT.ADDRESS";
  public static final int _SQL_TYPECODE = OracleTypes.REF;

  REF _ref;

private static final AddressRef _AddressRefFactory = new AddressRef();

  public static ORADataFactory getORADataFactory()
  { return _AddressRefFactory; }
  /* constructor */
  public AddressRef()
  {
  }
```

```
  /* ORAData interface */
  public Datum toDatum(Connection c) throws SQLException
  {
    return _ref;
  }

  /* ORADataFactory interface */
  public ORAData create(Datum d, int sqlType) throws SQLException
  {
    if (d == null) return null;
    AddressRef r = new AddressRef();
    r._ref = (REF) d;
    return r;
  }

  public static AddressRef cast(ORAData o) throws SQLException
  {
     if (o == null) return null;
     try { return (AddressRef) getORADataFactory().create(o.toDatum(null),
OracleTypes.REF); }
     catch (Exception exn)
     { throw new SQLException("Unable to convert "+o.getClass().getName()+" to
AddressRef: "+exn.toString()); }
  }

  public Address getValue() throws SQLException
  {
     return (Address) Address.getORADataFactory().create(
       _ref.getSTRUCT(), OracleTypes.REF);
  }

  public void setValue(Address c) throws SQLException
  {
    _ref.setValue((STRUCT) c.toDatum(_ref.getJavaSqlConnection()));
  }
}
```

AddressRef クラス内の getValue() メソッドは、AddressRef オブジェクトによって参照されたアドレスを適切な型で戻します。setValue() メソッドは、Address 引数の内容を AddressRef オブジェクトの参照先であるデータベース Address オブジェクトにコピーします。また、AddressRef クラスは、他の型の参照を Address の参照に変換する静的な cast() メソッドを提供します。

JPublisher で生成された Alltypes.java のリスト

ファイル ./demo/all/Alltypes.java の内容は次のとおりです。

> **注意 :** JPublisher で生成されるメソッド本体の詳細は、今後のリリースで変更される場合があります。

```
package all;

import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sql.ORAData;
import oracle.sql.ORADataFactory;
import oracle.sql.Datum;
import oracle.sql.STRUCT;
import oracle.jpub.runtime.MutableStruct;

public class Alltypes implements ORAData, ORADataFactory
{
  public static final String _SQL_NAME = "SCOTT.ALLTYPES";
  public static final int _SQL_TYPECODE = OracleTypes.STRUCT;

  protected MutableStruct _struct;

  private static int[] _sqlType =  {
-13,2004,1,2005,91,3,8,6,4,2,3,-2,7,5,12,12,2002,2006,2003,2003 };
  private static ORADataFactory[] _factory = new ORADataFactory[20];
  static
  {
    _factory[16] = corp.Address.getORADataFactory();
    _factory[17] = corp.AddressRef.getORADataFactory();
    _factory[18] = corp.AddrArray.getORADataFactory();
    _factory[19] = corp.Ntbl.getORADataFactory();
  }
protected static final Alltypes _AlltypesFactory = new Alltypes(false);

  public static ORADataFactory getORADataFactory()
  { return _AlltypesFactory; }
  /* constructor */
  protected Alltypes(boolean init)
  { if(init) _struct = new MutableStruct(new Object[20], _sqlType, _factory); }
  public Alltypes()
  { this(true); }
  public Alltypes(oracle.sql.BFILE attr1, oracle.sql.BLOB attr2, String attr3,
oracle.sql.CLOB attr4,
```

```
                        java.sql.Timestamp attr5, java.math.BigDecimal attr6,
Double attr7, Double attr8,
                Integer attr9, java.math.BigDecimal attr10, java.math.BigDecimal
attr11,
                        byte[] attr12, Float attr13, Integer attr14, String attr15,
String attr16, corp.Address attr17,
                corp.AddressRef attr18, corp.AddrArray attr19, corp.Ntbl attr20)
throws SQLException
  { this(true);
    setAttr1(attr1);
    setAttr2(attr2);
    setAttr3(attr3);
    setAttr4(attr4);
    setAttr5(attr5);
    setAttr6(attr6);
    setAttr7(attr7);
    setAttr8(attr8);
    setAttr9(attr9);
    setAttr10(attr10);
    setAttr11(attr11);
    setAttr12(attr12);
    setAttr13(attr13);
    setAttr14(attr14);
    setAttr15(attr15);
    setAttr16(attr16);
    setAttr17(attr17);
    setAttr18(attr18);
    setAttr19(attr19);
    setAttr20(attr20);
  }

  /* ORAData interface */
  public Datum toDatum(Connection c) throws SQLException
  {
    return _struct.toDatum(c, _SQL_NAME);
  }


  /* ORADataFactory interface */
  public ORAData create(Datum d, int sqlType) throws SQLException
  { return create(null, d, sqlType); }
  protected ORAData create(Alltypes o, Datum d, int sqlType) throws SQLException
  {
    if (d == null) return null;
    if (o == null) o = new Alltypes(false);
    o._struct = new MutableStruct((STRUCT) d, _sqlType, _factory);
    return o;
```

```
}
/* accessor methods */
public oracle.sql.BFILE getAttr1() throws SQLException
{ return (oracle.sql.BFILE) _struct.getOracleAttribute(0); }

public void setAttr1(oracle.sql.BFILE attr1) throws SQLException
{ _struct.setOracleAttribute(0, attr1); }


public oracle.sql.BLOB getAttr2() throws SQLException
{ return (oracle.sql.BLOB) _struct.getOracleAttribute(1); }

public void setAttr2(oracle.sql.BLOB attr2) throws SQLException
{ _struct.setOracleAttribute(1, attr2); }


public String getAttr3() throws SQLException
{ return (String) _struct.getAttribute(2); }

public void setAttr3(String attr3) throws SQLException
{ _struct.setAttribute(2, attr3); }


public oracle.sql.CLOB getAttr4() throws SQLException
{ return (oracle.sql.CLOB) _struct.getOracleAttribute(3); }

public void setAttr4(oracle.sql.CLOB attr4) throws SQLException
{ _struct.setOracleAttribute(3, attr4); }


public java.sql.Timestamp getAttr5() throws SQLException
{ return (java.sql.Timestamp) _struct.getAttribute(4); }

public void setAttr5(java.sql.Timestamp attr5) throws SQLException
{ _struct.setAttribute(4, attr5); }


public java.math.BigDecimal getAttr6() throws SQLException
{ return (java.math.BigDecimal) _struct.getAttribute(5); }

public void setAttr6(java.math.BigDecimal attr6) throws SQLException
{ _struct.setAttribute(5, attr6); }


public Double getAttr7() throws SQLException
{ return (Double) _struct.getAttribute(6); }
```

```
public void setAttr7(Double attr7) throws SQLException
{ _struct.setAttribute(6, attr7); }


public Double getAttr8() throws SQLException
{ return (Double) _struct.getAttribute(7); }

public void setAttr8(Double attr8) throws SQLException
{ _struct.setAttribute(7, attr8); }


public Integer getAttr9() throws SQLException
{ return (Integer) _struct.getAttribute(8); }

public void setAttr9(Integer attr9) throws SQLException
{ _struct.setAttribute(8, attr9); }


public java.math.BigDecimal getAttr10() throws SQLException
{ return (java.math.BigDecimal) _struct.getAttribute(9); }

public void setAttr10(java.math.BigDecimal attr10) throws SQLException
{ _struct.setAttribute(9, attr10); }


public java.math.BigDecimal getAttr11() throws SQLException
{ return (java.math.BigDecimal) _struct.getAttribute(10); }

public void setAttr11(java.math.BigDecimal attr11) throws SQLException
{ _struct.setAttribute(10, attr11); }


public byte[] getAttr12() throws SQLException
{ return (byte[]) _struct.getAttribute(11); }

public void setAttr12(byte[] attr12) throws SQLException
{ _struct.setAttribute(11, attr12); }


public Float getAttr13() throws SQLException
{ return (Float) _struct.getAttribute(12); }

public void setAttr13(Float attr13) throws SQLException
{ _struct.setAttribute(12, attr13); }


public Integer getAttr14() throws SQLException
```

```
{ return (Integer) _struct.getAttribute(13); }

public void setAttr14(Integer attr14) throws SQLException
{ _struct.setAttribute(13, attr14); }


public String getAttr15() throws SQLException
{ return (String) _struct.getAttribute(14); }

public void setAttr15(String attr15) throws SQLException
{ _struct.setAttribute(14, attr15); }


public String getAttr16() throws SQLException
{ return (String) _struct.getAttribute(15); }

public void setAttr16(String attr16) throws SQLException
{ _struct.setAttribute(15, attr16); }


public corp.Address getAttr17() throws SQLException
{ return (corp.Address) _struct.getAttribute(16); }

public void setAttr17(corp.Address attr17) throws SQLException
{ _struct.setAttribute(16, attr17); }


public corp.AddressRef getAttr18() throws SQLException
{ return (corp.AddressRef) _struct.getAttribute(17); }

public void setAttr18(corp.AddressRef attr18) throws SQLException
{ _struct.setAttribute(17, attr18); }


public corp.AddrArray getAttr19() throws SQLException
{ return (corp.AddrArray) _struct.getAttribute(18); }

public void setAttr19(corp.AddrArray attr19) throws SQLException
{ _struct.setAttribute(18, attr19); }


public corp.Ntbl getAttr20() throws SQLException
{ return (corp.Ntbl) _struct.getAttribute(19); }

public void setAttr20(corp.Ntbl attr20) throws SQLException
```

```
  { _struct.setAttribute(19, attr20); }

}
```

宣言されたクラスに別のパッケージからのユーザー定義クラスが必要な場合、JPublisher では oracle.sql パッケージ用の import 宣言に続いて、そのユーザー定義クラス用の import 宣言が生成されます。この場合、JDBC には、パッケージ corp からの Address および AddressRef クラスが必要です。

Address、AddressRef、AddrArray および Ntbl 型を持つ属性にはファクトリの構成が必要です。スタティック・イニシャライザは正しいファクトリを _factory 配列に配置します。

> **注意：** attr14 の SMALLINT SQL 型は Java の short 型にマップされますが、-numbertypes=objectjdbc のマッピングではこれが Integer にマップされることに注意してください。これは、JPublisher の実装によって決定されます。詳細は、3-9 ページの「数値型のマッピング (-numbertypes)」を参照してください。

JPublisher で生成された AlltypesRef.java のリスト

ファイル ./demo/corp/all/AlltypesRef.java の内容は次のとおりです。

> **注意：** JPublisher で生成されるメソッド本体の詳細は、今後のリリースで変更される場合があります。

```
package all;

import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sql.ORAData;
import oracle.sql.ORADataFactory;
import oracle.sql.Datum;
import oracle.sql.REF;
import oracle.sql.STRUCT;

public class AlltypesRef implements ORAData, ORADataFactory
{
  public static final String _SQL_BASETYPE = "SCOTT.ALLTYPES";
  public static final int _SQL_TYPECODE = OracleTypes.REF;

  REF _ref;
```

```
private static final AlltypesRef _AlltypesRefFactory = new AlltypesRef();

  public static ORADataFactory getORADataFactory()
  { return _AlltypesRefFactory; }
  /* constructor */
  public AlltypesRef()
  {
  }

  /* ORAData interface */
  public Datum toDatum(Connection c) throws SQLException
  {
    return _ref;
  }

  /* ORADataFactory interface */
  public ORAData create(Datum d, int sqlType) throws SQLException
  {
    if (d == null) return null;
    AlltypesRef r = new AlltypesRef();
    r._ref = (REF) d;
    return r;
  }

  public static AlltypesRef cast(ORAData o) throws SQLException
  {
     if (o == null) return null;
     try { return (AlltypesRef) getORADataFactory().create(o.toDatum(null),
OracleTypes.REF); }
     catch (Exception exn)
     { throw new SQLException("Unable to convert "+o.getClass().getName()+" to
AlltypesRef: "+exn.toString()); }
  }

  public Alltypes getValue() throws SQLException
  {
     return (Alltypes) Alltypes.getORADataFactory().create(
       _ref.getSTRUCT(), OracleTypes.REF);
  }

  public void setValue(Alltypes c) throws SQLException
  {
    _ref.setValue((STRUCT) c.toDatum(_ref.getJavaSqlConnection()));
  }
}
```

JPublisher で生成された Ntbl.java のリスト

ファイル ./demo/corp/Ntbl.java の内容は次のとおりです。

> **注意 :** JPublisher で生成されるメソッド本体の詳細は、今後のリリースで変更される場合があります。

```
package corp;

import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sql.ORAData;
import oracle.sql.ORADataFactory;
import oracle.sql.Datum;
import oracle.sql.ARRAY;
import oracle.sql.ArrayDescriptor;
import oracle.jpub.runtime.MutableArray;

public class Ntbl implements ORAData, ORADataFactory
{
  public static final String _SQL_NAME = "SCOTT.NTBL";
  public static final int _SQL_TYPECODE = OracleTypes.ARRAY;

  MutableArray _array;

private static final Ntbl _NtblFactory = new Ntbl();

  public static ORADataFactory getORADataFactory()
  { return _NtblFactory; }
  /* constructors */
  public Ntbl()
  {
    this((Integer[])null);
  }

  public Ntbl(Integer[] a)
  {
    _array = new MutableArray(4, a, null);
  }

  /* ORAData interface */
  public Datum toDatum(Connection c) throws SQLException
  {
    return _array.toDatum(c, _SQL_NAME);
```

```
}

/* ORADataFactory interface */
public ORAData create(Datum d, int sqlType) throws SQLException
{
  if (d == null) return null;
  Ntbl a = new Ntbl();
  a._array = new MutableArray(4, (ARRAY) d, null);
  return a;
}

public int length() throws SQLException
{
  return _array.length();
}

public int getBaseType() throws SQLException
{
  return _array.getBaseType();
}

public String getBaseTypeName() throws SQLException
{
  return _array.getBaseTypeName();
}

public ArrayDescriptor getDescriptor() throws SQLException
{
  return _array.getDescriptor();
}

/* array accessor methods */
public Integer[] getArray() throws SQLException
{
  return (Integer[]) _array.getObjectArray();
}

public void setArray(Integer[] a) throws SQLException
{
  _array.setObjectArray(a);
}

public Integer[] getArray(long index, int count) throws SQLException
{
  return (Integer[]) _array.getObjectArray(index, count);
}
```

```
  public void setArray(Integer[] a, long index) throws SQLException
  {
    _array.setObjectArray(a, index);
  }

  public Integer getElement(long index) throws SQLException
  {
    return (Integer) _array.getObjectElement(index);
  }

  public void setElement(Integer a, long index) throws SQLException
  {
    _array.setObjectElement(a, index);
  }

}
```

JPublisher で生成された AddrArray.java のリスト

JPublisher では Alltypes 型に必要な AddrArray 型の宣言が生成されます。ファイル ./demo/corp/AddrArray.java の内容は次のとおりです。

> **注意：** JPublisher で生成されるメソッド本体の詳細は、今後のリリースで変更される場合があります。

```
package corp;

import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sql.ORAData;
import oracle.sql.ORADataFactory;
import oracle.sql.Datum;
import oracle.sql.ARRAY;
import oracle.sql.ArrayDescriptor;
import oracle.jpub.runtime.MutableArray;

public class AddrArray implements ORAData, ORADataFactory
{
  public static final String _SQL_NAME = "SCOTT.ADDR_ARRAY";
  public static final int _SQL_TYPECODE = OracleTypes.ARRAY;

  MutableArray _array;
```

```
private static final AddrArray _AddrArrayFactory = new AddrArray();

  public static ORADataFactory getORADataFactory()
  { return _AddrArrayFactory; }
  /* constructors */
  public AddrArray()
  {
    this((Address[])null);
  }

  public AddrArray(Address[] a)
  {
    _array = new MutableArray(2002, a, Address.getORADataFactory());
  }

  /* ORAData interface */
  public Datum toDatum(Connection c) throws SQLException
  {
    return _array.toDatum(c, _SQL_NAME);
  }

  /* ORADataFactory interface */
  public ORAData create(Datum d, int sqlType) throws SQLException
  {
    if (d == null) return null;
    AddrArray a = new AddrArray();
    a._array = new MutableArray(2002, (ARRAY) d, Address.getORADataFactory());
    return a;
  }

  public int length() throws SQLException
  {
    return _array.length();
  }

  public int getBaseType() throws SQLException
  {
    return _array.getBaseType();
  }

  public String getBaseTypeName() throws SQLException
  {
    return _array.getBaseTypeName();
  }

  public ArrayDescriptor getDescriptor() throws SQLException
  {
```

```
    return _array.getDescriptor();
  }

  /* array accessor methods */
  public Address[] getArray() throws SQLException
  {
    return (Address[]) _array.getObjectArray(
      new Address[_array.length()]);
  }

  public void setArray(Address[] a) throws SQLException
  {
    _array.setObjectArray(a);
  }

  public Address[] getArray(long index, int count) throws SQLException
  {
    return (Address[]) _array.getObjectArray(index,
      new Address[_array.sliceLength(index, count)]);
  }

  public void setArray(Address[] a, long index) throws SQLException
  {
    _array.setObjectArray(a, index);
  }

  public Address getElement(long index) throws SQLException
  {
    return (Address) _array.getObjectElement(index);
  }

  public void setElement(Address a, long index) throws SQLException
  {
    _array.setObjectElement(a, index);
  }

}
```

例 : SQLData クラスの生成

この例は、JPublisher で ORAData クラスではなく SQLData クラスが生成される点を除き、前述の例と同じです。この例のコマンドラインは次のようになります。

```
jpub -user=scott/tiger -input=demoin -dir=demo -package=corp -mapping=objectjdbc
-usertypes=jdbc -methods=false
```

（ここでは折り返されていますが、1 行のコマンドラインです。）

注意 : -mapping オプションは使用不可になっていますが、引き続きサポートされているため例に示してあります。-mapping=objectjdbc 設定は、-builtintypes=jdbc、-numbertypes=objectjdbc、-lobtypes=oracle および -usertypes=oracle の組合せと同等です。ただし、このコマンドラインにより、-usertypes=oracle 設定が -usertypes=jdbc 設定でオーバーライドされます。-mapping オプションの詳細は、3-11 ページの「すべての型のマッピング（-mapping）」を参照してください。

オプション -usertypes=jdbc の指示により、JPublisher では SQLData インタフェースを実装するクラスが生成されます。SQLData インタフェースは、ユーザー定義のクラスではなく汎用型 java.sql.Ref および java.sql.Array を使用して、参照クラスおよびコレクション・クラスを包括的にサポートします。したがって、JPublisher では次の 2 つのクラスのみが生成されます。

```
./demo/corp/Address.java
./demo/all/Alltypes.java
```

JPublisher で生成された Address.java のリスト

この例では -usertypes=jdbc が指定されているため、Address クラスでは、oracle.sql.ORAData インタフェースではなく java.sql.SQLData インタフェースが実装されます。ファイル ./demo/corp/Address.java の内容は次のとおりです。

> **注意：** JPublisher で生成されるメソッド本体の詳細は、今後のリリースで変更される場合があります。

```
package corp;

import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sql.ORAData;
import oracle.sql.ORADataFactory;
import oracle.sql.Datum;
import oracle.sql.STRUCT;
import oracle.jpub.runtime.MutableStruct;

public class Address implements ORAData, ORADataFactory
{
  public static final String _SQL_NAME = "SCOTT.ADDRESS";
  public static final int _SQL_TYPECODE = OracleTypes.STRUCT;

  protected MutableStruct _struct;

  private static int[] _sqlType =  { 12,12,12,2 };
  private static ORADataFactory[] _factory = new ORADataFactory[4];
protected static final Address _AddressFactory = new Address(false);

  public static ORADataFactory getORADataFactory()
  { return _AddressFactory; }
  /* constructor */
  protected Address(boolean init)
  { if(init) _struct = new MutableStruct(new Object[4], _sqlType, _factory); }
  public Address()
  { this(true); }
  public Address(String street, String city, String state, java.math.BigDecimal zip)
throws SQLException
  { this(true);
    setStreet(street);
    setCity(city);
    setState(state);
    setZip(zip);
```

```
}

/* ORAData interface */
public Datum toDatum(Connection c) throws SQLException
{
  return _struct.toDatum(c, _SQL_NAME);
}


/* ORADataFactory interface */
public ORAData create(Datum d, int sqlType) throws SQLException
{ return create(null, d, sqlType); }
protected ORAData create(Address o, Datum d, int sqlType) throws SQLException
{
  if (d == null) return null;
  if (o == null) o = new Address(false);
  o._struct = new MutableStruct((STRUCT) d, _sqlType, _factory);
  return o;
}
/* accessor methods */
public String getStreet() throws SQLException
{ return (String) _struct.getAttribute(0); }

public void setStreet(String street) throws SQLExcept
```

JPublisher で生成された Alltypes.java のリスト

この例では -usertypes=jdbc が指定されているため、Alltypes クラスでは、oracle.sql.ORAData インタフェースではなく java.sql.SQLData インタフェースが実装されます。SQLData インタフェースはベンダーに依存しない標準ですが、Alltypes クラスには Oracle 固有のコードがあります。これは、oracle.sql.BFILE および oracle.sql.CLOB など、Oracle 固有の型を使用するためです。ファイル ./demo/corp/Alltypes.java の内容は次のとおりです。

> **注意 :** JPublisher で生成されるメソッド本体の詳細は、今後のリリースで変更される場合があります。

```
package all;

import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sql.ORAData;
import oracle.sql.ORADataFactory;
import oracle.sql.Datum;
```

```
import oracle.sql.STRUCT;
import oracle.jpub.runtime.MutableStruct;

public class Alltypes implements ORAData, ORADataFactory
{
  public static final String _SQL_NAME = "SCOTT.ALLTYPES";
  public static final int _SQL_TYPECODE = OracleTypes.STRUCT;

  protected MutableStruct _struct;

  private static int[] _sqlType =  {
-13,2004,1,2005,91,3,8,6,4,2,3,-2,7,5,12,12,2002,2006,2003,2003 };
  private static ORADataFactory[] _factory = new ORADataFactory[20];
  static
  {
    _factory[16] = corp.Address.getORADataFactory();
    _factory[17] = corp.AddressRef.getORADataFactory();
    _factory[18] = corp.AddrArray.getORADataFactory();
    _factory[19] = corp.Ntbl.getORADataFactory();
  }
protected static final Alltypes _AlltypesFactory = new Alltypes(false);

  public static ORADataFactory getORADataFactory()
  { return _AlltypesFactory; }
  /* constructor */
  protected Alltypes(boolean init)
  { if(init) _struct = new MutableStruct(new Object[20], _sqlType, _factory); }
  public Alltypes()
  { this(true); }
  public Alltypes(oracle.sql.BFILE attr1, oracle.sql.BLOB attr2, String attr3,
oracle.sql.CLOB attr4,
                        java.sql.Timestamp attr5, java.math.BigDecimal attr6,
Double attr7, Double attr8,
                         Integer attr9, java.math.BigDecimal attr10,
java.math.BigDecimal attr11, byte[] attr12,
                         Float attr13, Integer attr14, String attr15, String
attr16, corp.Address attr17,
                         corp.AddressRef attr18, corp.AddrArray attr19,
corp.Ntbl attr20) throws SQLException
  { this(true);
    setAttr1(attr1);
    setAttr2(attr2);
    setAttr3(attr3);
    setAttr4(attr4);
    setAttr5(attr5);
    setAttr6(attr6);
    setAttr7(attr7);
```

```
  setAttr8(attr8);
  setAttr9(attr9);
  setAttr10(attr10);
  setAttr11(attr11);
  setAttr12(attr12);
  setAttr13(attr13);
  setAttr14(attr14);
  setAttr15(attr15);
  setAttr16(attr16);
  setAttr17(attr17);
  setAttr18(attr18);
  setAttr19(attr19);
  setAttr20(attr20);
}

/* ORAData interface */
public Datum toDatum(Connection c) throws SQLException
{
  return _struct.toDatum(c, _SQL_NAME);
}


/* ORADataFactory interface */
public ORAData create(Datum d, int sqlType) throws SQLException
{ return create(null, d, sqlType); }
protected ORAData create(Alltypes o, Datum d, int sqlType) throws SQLException
{
  if (d == null) return null;
  if (o == null) o = new Alltypes(false);
  o._struct = new MutableStruct((STRUCT) d, _sqlType, _factory);
  return o;
}
/* accessor methods */
public oracle.sql.BFILE getAttr1() throws SQLException
{ return (oracle.sql.BFILE) _struct.getOracleAttribute(0); }

public void setAttr1(oracle.sql.BFILE attr1) throws SQLException
{ _struct.setOracleAttribute(0, attr1); }


public oracle.sql.BLOB getAttr2() throws SQLException
{ return (oracle.sql.BLOB) _struct.getOracleAttribute(1); }

public void setAttr2(oracle.sql.BLOB attr2) throws SQLException
{ _struct.setOracleAttribute(1, attr2); }
```

```
public String getAttr3() throws SQLException
{ return (String) _struct.getAttribute(2); }

public void setAttr3(String attr3) throws SQLException
{ _struct.setAttribute(2, attr3); }


public oracle.sql.CLOB getAttr4() throws SQLException
{ return (oracle.sql.CLOB) _struct.getOracleAttribute(3); }

public void setAttr4(oracle.sql.CLOB attr4) throws SQLException
{ _struct.setOracleAttribute(3, attr4); }


public java.sql.Timestamp getAttr5() throws SQLException
{ return (java.sql.Timestamp) _struct.getAttribute(4); }

public void setAttr5(java.sql.Timestamp attr5) throws SQLException
{ _struct.setAttribute(4, attr5); }


public java.math.BigDecimal getAttr6() throws SQLException
{ return (java.math.BigDecimal) _struct.getAttribute(5); }

public void setAttr6(java.math.BigDecimal attr6) throws SQLException
{ _struct.setAttribute(5, attr6); }


public Double getAttr7() throws SQLException
{ return (Double) _struct.getAttribute(6); }

public void setAttr7(Double attr7) throws SQLException
{ _struct.setAttribute(6, attr7); }


public Double getAttr8() throws SQLException
{ return (Double) _struct.getAttribute(7); }

public void setAttr8(Double attr8) throws SQLException
{ _struct.setAttribute(7, attr8); }


public Integer getAttr9() throws SQLException
{ return (Integer) _struct.getAttribute(8); }

public void setAttr9(Integer attr9) throws SQLException
{ _struct.setAttribute(8, attr9); }
```

```
public java.math.BigDecimal getAttr10() throws SQLException
{ return (java.math.BigDecimal) _struct.getAttribute(9); }

public void setAttr10(java.math.BigDecimal attr10) throws SQLException
{ _struct.setAttribute(9, attr10); }


public java.math.BigDecimal getAttr11() throws SQLException
{ return (java.math.BigDecimal) _struct.getAttribute(10); }

public void setAttr11(java.math.BigDecimal attr11) throws SQLException
{ _struct.setAttribute(10, attr11); }


public byte[] getAttr12() throws SQLException
{ return (byte[]) _struct.getAttribute(11); }

public void setAttr12(byte[] attr12) throws SQLException
{ _struct.setAttribute(11, attr12); }


public Float getAttr13() throws SQLException
{ return (Float) _struct.getAttribute(12); }

public void setAttr13(Float attr13) throws SQLException
{ _struct.setAttribute(12, attr13); }


public Integer getAttr14() throws SQLException
{ return (Integer) _struct.getAttribute(13); }

public void setAttr14(Integer attr14) throws SQLException
{ _struct.setAttribute(13, attr14); }


public String getAttr15() throws SQLException
{ return (String) _struct.getAttribute(14); }

public void setAttr15(String attr15) throws SQLException
{ _struct.setAttribute(14, attr15); }


public String getAttr16() throws SQLException
{ return (String) _struct.getAttribute(15); }
```

```
  public void setAttr16(String attr16) throws SQLException
  { _struct.setAttribute(15, attr16); }


  public corp.Address getAttr17() throws SQLException
  { return (corp.Address) _struct.getAttribute(16); }

  public void setAttr17(corp.Address attr17) throws SQLException
  { _struct.setAttribute(16, attr17); }


  public corp.AddressRef getAttr18() throws SQLException
  { return (corp.AddressRef) _struct.getAttribute(17); }

  public void setAttr18(corp.AddressRef attr18) throws SQLException
  { _struct.setAttribute(17, attr18); }


  public corp.AddrArray getAttr19() throws SQLException
  { return (corp.AddrArray) _struct.getAttribute(18); }

  public void setAttr19(corp.AddrArray attr19) throws SQLException
  { _struct.setAttribute(18, attr19); }


  public corp.Ntbl getAttr20() throws SQLException
  { return (corp.Ntbl) _struct.getAttribute(19); }

  public void setAttr20(corp.Ntbl attr20) throws SQLException
  { _struct.setAttribute(19, attr20); }

}
```

例 : JPublisher クラスの拡張

この項では、2-32 ページの「JPublisher で生成されたクラスの拡張」で説明した使用例を示します。

次に示すコードは、クラス MyAddress.java の初期バージョンです。このコードは JPublisher によって自動的に作成され、ディレクトリ demo/corp に格納されます。JPublisher では、同じコマンドラインを使用して再起動するたびに、MyAddress ではなくスーパークラス JAddress が再生成されるため（存在する場合）、その後このコードを変更できます。

注意 : ここに示す ORADataFactory create() メソッドをコーディングすると、データ・オブジェクトが null の場合にオブジェクト・インスタンスが不必要に作成されたり、データ・オブジェクトが非 null の場合に不必要に再初期化されることがなくなります。この方法については、2-33 ページの「生成クラスを拡張するクラスの書式化」を参照してください。

```
package corp;

import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sql.ORAData;
import oracle.sql.ORADataFactory;
import oracle.sql.Datum;
import oracle.sql.STRUCT;
import oracle.jpub.runtime.MutableStruct;

public class MyAddress extends JAddress implements ORAData, ORADataFactory
{
   private static final MyAddress _MyAddressFactory = new MyAddress();
   public static ORADataFactory getORADataFactory()
   { return _MyAddressFactory; }

   public MyAddress() { super(); }
  public MyAddress(String street, String city, String state,
                   java.math.BigDecimal zip) throws SQLException
  {
    setStreet(street);
    setCity(city);
    setState(state);
    setZip(zip);
  }
   /* ORAData interface */
   public ORAData create(Datum d, int sqlType) throws SQLException
```

```
   { return create(new MyAddress(), d, sqlType); }

  /* superclass accessors */

/*
  public String getStreet() throws SQLException { return super.getStreet(); }
  public void setStreet(String street) throws SQLException {
super.setStreet(street); }
 */

/*
  public String getCity() throws SQLException { return super.getCity(); }
  public void setCity(String city) throws SQLException { super.setCity(city); }
 */

/*
  public String getState() throws SQLException { return super.getState(); }
  public void setState(String state) throws SQLException {
super.setState(state); }
 */

/*
  public java.math.BigDecimal getZip() throws SQLException { return
super.getZip(); }
  public void setZip(java.math.BigDecimal zip) throws SQLException {
super.setZip(zip); }
 */

}
```

次のコマンドラインを入力して、JPublisher でスーパークラス JAddress 用のコードを生成し、JAddress 外部にあるクラス MyAddress 用の初期スタブを生成します。（このスタブが作成されるのは、MyAddress.java が存在しない場合のみです。）

```
jpub -user=scott/tiger -input=demoin -dir=demo -package=corp
```

demoin ファイルの内容が次のとおりであるとします。

```
SQL ADDRESS GENERATE JAddress AS MyAddress
```

JPublisher では次のファイルが生成されます。

```
demo/corp/JAddress.java
demo/corp/MyAddressRef.java
```

ADDRESS オブジェクトは Java プログラムでは MyAddress インスタンスとして表されるため、JPublisher ではクラス JAddressRef ではなく MyAddressRef が生成されます。

次に、JPublisher で常に生成される demo/corp/JAddress.java クラス・ファイルをリストします。

> **注意：** JPublisher で生成されるメソッド本体の詳細は、今後のリリースで変更される場合があります。

```
package corp;

import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sql.ORAData;
import oracle.sql.ORADataFactory;
import oracle.sql.Datum;
import oracle.sql.STRUCT;
import oracle.jpub.runtime.MutableStruct;

public class JAddress implements ORAData, ORADataFactory
{
  public static final String _SQL_NAME = "SCOTT.ADDRESS";
  public static final int _SQL_TYPECODE = OracleTypes.STRUCT;

  protected MutableStruct _struct;

  private static int[] _sqlType =  { 12,12,12,2 };
  private static ORADataFactory[] _factory = new ORADataFactory[4];
protected static final JAddress _JAddressFactory = new JAddress(false);

  public static ORADataFactory getORADataFactory()
  { return _JAddressFactory; }
  /* constructor */
  protected JAddress(boolean init)
  { if(init) _struct = new MutableStruct(new Object[4], _sqlType, _factory); }
  public JAddress()
  { this(true); }
  public JAddress(String street, String city, String state,
                  java.math.BigDecimal zip) throws SQLException
  { this(true);
    setStreet(street);
    setCity(city);
    setState(state);
    setZip(zip);
  }

  /* ORAData interface */
```

```
  public Datum toDatum(Connection c) throws SQLException
  {
    return _struct.toDatum(c, _SQL_NAME);
  }


  /* ORADataFactory interface */
  public ORAData create(Datum d, int sqlType) throws SQLException
  { return create(null, d, sqlType); }
  protected ORAData create(JAddress o, Datum d, int sqlType) throws SQLException
  {
    if (d == null) return null;
    if (o == null) o = new JAddress(false);
    o._struct = new MutableStruct((STRUCT) d, _sqlType, _factory);
    return o;
  }
  /* accessor methods */
  public String getStreet() throws SQLException
  { return (String) _struct.getAttribute(0); }

  public void setStreet(String street) throws SQLException
  { _struct.setAttribute(0, street); }


  public String getCity() throws SQLException
  { return (String) _struct.getAttribute(1); }

  public void setCity(String city) throws SQLException
  { _struct.setAttribute(1, city); }


  public String getState() throws SQLException
  { return (String) _struct.getAttribute(2); }

  public void setState(String state) throws SQLException
  { _struct.setAttribute(2, state); }


  public java.math.BigDecimal getZip() throws SQLException
  { return (java.math.BigDecimal) _struct.getAttribute(3); }

  public void setZip(java.math.BigDecimal zip) throws SQLException
  { _struct.setAttribute(3, zip); }

}
```

次に、JPublisher で生成された demo/corp/MyAddressRef.java クラス・ファイルをリストします。

```
package corp;

import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sql.ORAData;
import oracle.sql.ORADataFactory;
import oracle.sql.Datum;
import oracle.sql.REF;
import oracle.sql.STRUCT;

public class MyAddressRef implements ORAData, ORADataFactory
{
  public static final String _SQL_BASETYPE = "SCOTT.ADDRESS";
  public static final int _SQL_TYPECODE = OracleTypes.REF;

  REF _ref;

private static final MyAddressRef _MyAddressRefFactory = new MyAddressRef();

  public static ORADataFactory getORADataFactory()
  { return _MyAddressRefFactory; }
  /* constructor */
  public MyAddressRef()
  {
  }

  /* ORAData interface */
  public Datum toDatum(Connection c) throws SQLException
  {
    return _ref;
  }

  /* ORADataFactory interface */
  public ORAData create(Datum d, int sqlType) throws SQLException
  {
    if (d == null) return null;
    MyAddressRef r = new MyAddressRef();
    r._ref = (REF) d;
    return r;
  }

  public static MyAddressRef cast(ORAData o) throws SQLException
  {
```

```
    if (o == null) return null;
    try { return (MyAddressRef) getORADataFactory().create(o.toDatum(null),
OracleTypes.REF); }
    catch (Exception exn)
    { throw new SQLException("Unable to convert "+o.getClass().getName()+" to
MyAddressRef: "+exn.toString()); }
  }

  public MyAddress getValue() throws SQLException
  {
     return (MyAddress) MyAddress.getORADataFactory().create(
       _ref.getSTRUCT(), OracleTypes.REF);
  }

  public void setValue(MyAddress c) throws SQLException
  {
    _ref.setValue((STRUCT) c.toDatum(_ref.getJavaSqlConnection()));
  }
}
```

例 : オブジェクトのメソッド用に生成されたラッパー

> **注意 :** JPublisher でストアド・プロシージャをコールするために生成されるラッパー・メソッドは、SQLJ コードで生成されます。したがって、ラッパー・メソッドを含む生成クラスは、SQLJ Translator で処理する必要があります。

この項では、メソッドを含む SQL 型に次の定義が指定された場合の JPublisher 出力の例について説明します。この例では、属性 numerator と denominator を持つオブジェクト型 Rational、および次のファンクションとプロシージャを定義します。

- MEMBER FUNCTION toReal: 2 つの整数を指定すると、このファンクションは有理数を実数に変換して実数を戻します。
- MEMBER PROCEDURE normalize: 分子と分母を表す 2 つの整数を指定すると、このプロシージャは分子と分母をその最大公約数で割って分数を約分します。
- STATIC FUNCTION gcd: 2 つの整数を指定すると、このファンクションはその最大公約数を戻します。
- MEMBER FUNCTION plus: このファンクションは 2 つの有理数を加算して結果を戻します。

rational.sql のコードは次のようになります。

```
CREATE TYPE Rational AS OBJECT (
 numerator INTEGER,
 denominator INTEGER,
 MAP MEMBER FUNCTION toReal RETURN REAL,
 MEMBER PROCEDURE normalize,
 STATIC FUNCTION gcd(x INTEGER,
                     y INTEGER) RETURN INTEGER,
 MEMBER FUNCTION plus ( x Rational) RETURN Rational
);

CREATE TYPE BODY Rational AS

MAP MEMBER FUNCTION toReal RETURN REAL IS
-- convert rational number to real number
BEGIN
  RETURN numerator / denominator;
END toReal;

MEMBER PROCEDURE normalize IS
 g INTEGER;
BEGIN
 g := Rational.gcd(numerator, denominator);
 numerator := numerator / g;
 denominator := denominator / g;
END normalize;

STATIC FUNCTION gcd(x INTEGER,
                    y INTEGER) RETURN INTEGER IS
-- find greatest common divisor of x and y
ans INTEGER;
z INTEGER;
BEGIN
IF x < y THEN
   ans := Rational.gcd(y, x);
ELSIF (x MOD y = 0) THEN
   ans := y;
ELSE
   z := x MOD y;
   ans := Rational.gcd(y, z);
END IF;
RETURN ans;
END gcd;

MEMBER FUNCTION plus (x Rational) RETURN Rational IS
BEGIN
 return Rational(numerator * x.denominator + x.numerator * denominator,
```

```
                denominator * x.denominator);
END plus;
END;
```

この例では、JPublisher を次のコマンドラインで起動します。

```
jpub -user=scott/tiger -sql=Rational -methods=true
```

JPublisher は `-user` パラメータの指示により、ユーザー `scott` でパスワード `tiger` を使用してデータベースにログインします。`-methods` パラメータの指示により、JPublisher は `Rational` 型に含まれるメソッド用のラッパーが生成されます。`-methods=true` パラメータはデフォルトのため省略できます。

JPublisher で生成された Rational.sqlj のリストと説明

JPublisher では、ファイル `Rational.sqlj` が生成されます。このファイルの内容は次のとおりです。

注意 :

- JPublisher で生成されるメソッド本体の詳細は、今後のリリースで変更される場合があります。
- `release()` コールは、SQLJ 接続コンテキストに関連するメモリー・リークを回避するためのコールであることに注意してください。詳細は、2-28 ページの「接続コンテキストのリソースの解放」を参照してください。

```
import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sql.ORAData;
import oracle.sql.ORADataFactory;
import oracle.sql.Datum;
import oracle.sql.STRUCT;
import oracle.jpub.runtime.MutableStruct;
import sqlj.runtime.ref.DefaultContext;
import sqlj.runtime.ConnectionContext;
import java.sql.Connection;

public class Rational implements ORAData, ORADataFactory
{
  public static final String _SQL_NAME = "SCOTT.RATIONAL";
  public static final int _SQL_TYPECODE = OracleTypes.STRUCT;
```

```
  /* connection management */
  protected DefaultContext __tx = null;
  protected Connection __onn = null;
  public void setConnectionContext(DefaultContext ctx) throws SQLException
  { release(); __tx = ctx; }
  public DefaultContext getConnectionContext() throws SQLException
  { if (__tx==null)
    { __tx = (__onn==null) ? DefaultContext.getDefaultContext() : new
DefaultContext(__onn); }
    return __tx;
  };
  public Connection getConnection() throws SQLException
  { return (__onn==null) ? ((__tx==null) ? null : __tx.getConnection()) : __onn;
}
  public void release() throws SQLException
  { if (__tx!=null && __onn!=null)
__tx.close(ConnectionContext.KEEP_CONNECTION);
    __onn = null; __tx = null;
  }

  protected MutableStruct _struct;

  private static int[] _sqlType =  { 4,4 };
  private static ORADataFactory[] _factory = new ORADataFactory[2];
protected static final Rational _RationalFactory = new Rational(false);

  public static ORADataFactory getORADataFactory()
  { return _RationalFactory; }
  /* constructors */
  protected Rational(boolean init)
  { if (init) _struct = new MutableStruct(new Object[2], _sqlType, _factory); }
  public Rational()
  { this(true); __tx = DefaultContext.getDefaultContext(); }
  public Rational(DefaultContext c) /*throws SQLException*/
  { this(true); __tx = c; }
  public Rational(Connection c) /*throws SQLException*/
  { this(true); __onn = c; }
  public Rational(Integer numerator, Integer denominator) throws SQLException
   {
    this(true);
    setNumerator(numerator);
    setDenominator(denominator);
  }

  /* ORAData interface */
  public Datum toDatum(Connection c) throws SQLException
  {
```

```
  if (__tx!=null && __onn!=c) release();
  __onn = c;
  return _struct.toDatum(c, _SQL_NAME);
}


/* ORADataFactory interface */
public ORAData create(Datum d, int sqlType) throws SQLException
{ return create(null, d, sqlType); }
public void setFrom(Rational o) throws SQLException
{ setContextFrom(o); setValueFrom(o); }
protected void setContextFrom(Rational o) throws SQLException
{ release(); __tx = o.__tx; __onn = o.__onn; }
protected void setValueFrom(Rational o) { _struct = o._struct; }
protected ORAData create(Rational o, Datum d, int sqlType) throws SQLException
{
  if (d == null) { if (o!=null) { o.release(); }; return null; }
  if (o == null) o = new Rational(false);
  o._struct = new MutableStruct((STRUCT) d, _sqlType, _factory);
  o.__onn = ((STRUCT) d).getJavaSqlConnection();
  return o;
}
/* accessor methods */
public Integer getNumerator() throws SQLException
{ return (Integer) _struct.getAttribute(0); }

public void setNumerator(Integer numerator) throws SQLException
{ _struct.setAttribute(0, numerator); }


public Integer getDenominator() throws SQLException
{ return (Integer) _struct.getAttribute(1); }

public void setDenominator(Integer denominator) throws SQLException
{ _struct.setAttribute(1, denominator); }


public Integer gcd (
  Integer x,
  Integer y)
throws SQLException
{
  Integer __jPt_result;
  #sql [getConnectionContext()] __jPt_result = { VALUES(SCOTT.RATIONAL.GCD(
    :x,
    :y)) };
  return __jPt_result;
```

```
  }

  public Rational normalize ()
  throws SQLException
  {
    Rational __jPt_temp = this;
    #sql [getConnectionContext()] {
      BEGIN
      :INOUT __jPt_temp.NORMALIZE();
      END;
    };
    return __jPt_temp;
  }

  public Rational plus (
    Rational x)
  throws SQLException
  {
    Rational __jPt_temp = this;
    Rational __jPt_result;
    #sql [getConnectionContext()] {
      BEGIN
      :OUT __jPt_result := :__jPt_temp.PLUS(
      :x);
      END;
    };
    return __jPt_result;
  }

  public Float toreal ()
  throws SQLException
  {
    Rational __jPt_temp = this;
    Float __jPt_result;
    #sql [getConnectionContext()] {
      BEGIN
      :OUT __jPt_result := :__jPt_temp.TOREAL();
      END;
    };
    return __jPt_result;
  }
}
```

JPublisher で生成されるすべてのメソッドは、対応する PL/SQL メソッドをコールしサーバー上で実行されます。

JPublisher では、オブジェクトの *sql_name* が SCOTT.RATIONAL として宣言され、その *sql_type_code* が OracleTypes.STRUCT として宣言されます。デフォルトでは、SQLJ 接続コンテキスト・クラス sqlj.runtime.ref.DefaultContext が使用されます。オブジェクト属性 numerator および denominator 用にアクセッサ・メソッド getNumerator()、setNumerator()、getDenominator() および setDenominator() が作成されます。

JPublisher ではスタティック・ファンクション gcd 用のソース・コードが生成され、2 つの Integer 値を入力として使用して、Integer の結果が戻されます。このファンクション gcd は、IN ホスト変数 :x および :y を使用してストアド・ファンクション RATIONAL.GCD をコールします。

JPublisher ではメンバー・プロシージャ normalize 用のソース・コードが生成され、SQLJ 文内に IN OUT パラメータを含む PL/SQL ブロックを定義します。this パラメータにより、値が PL/SQL ブロックに渡されます。

JPublisher では、メンバー関数 plus 用のソース・コードが生成され、Rational 型のオブジェクト x を使用して Rational 型のオブジェクトが戻されます。これは、SQLJ 文内の PL/SQL ブロックを定義します。IN ホスト変数は :x で、this のコピーです。ファンクションの結果は OUT ホスト変数になります。

JPublisher では、メンバー関数 toReal 用のソース・コードが生成され、Float 値が戻されます。ファンクションで戻された値を割り当てるホスト OUT 変数を定義します。this オブジェクトのコピーは IN パラメータになります。

例 : パッケージのメソッド用に生成されたラッパー

注意 : JPublisher でストアド・プロシージャをコールするために生成されるラッパー・メソッドは、SQLJ コードで生成されます。したがって、ラッパー・メソッドを含む生成クラスは、SQLJ Translator で処理する必要があります。

この項では、JPublisher に対して次の定義でメソッドを含む PL/SQL パッケージを指定した場合の出力の例を説明します。この例では、分数の分子と分母を操作する次のファンクションとプロシージャでパッケージ RationalP を定義します。

- FUNCTION toReal: 2 つの整数を指定すると、このファンクションは有理数を実数に変換して実数を戻します。
- PROCEDURE normalize: 分子と分母を表す 2 つの整数を指定すると、このプロシージャは分子と分母をその最大公約数で割って分数を約分します。
- FUNCTION gcd: 2 つの整数を指定すると、このファンクションはその最大公約数を戻します。
- PROCEDURE plus: 2 つの有理数を加算して結果を戻します。

RationalP.sql のコードは次のようになります。

```
CREATE PACKAGE RationalP AS

 FUNCTION toReal(numerator   INTEGER,
                 denominator INTEGER) RETURN REAL;

 PROCEDURE normalize(numerator   IN OUT INTEGER,
                     denominator IN OUT INTEGER);

 FUNCTION gcd(x INTEGER, y INTEGER) RETURN INTEGER;

 PROCEDURE plus (n1 INTEGER, d1 INTEGER,
                 n2 INTEGER, d2 INTEGER,
                 n3 OUT INTEGER, d3 OUT INTEGER);
END rationalP;

/

CREATE PACKAGE BODY rationalP AS

  FUNCTION toReal(numerator INTEGER,
                  denominator INTEGER) RETURN real IS
  -- convert rational number to real number
  BEGIN
    RETURN numerator / denominator;
  END toReal;

  FUNCTION gcd(x INTEGER, y INTEGER) RETURN INTEGER IS
  -- find greatest common divisor of x and y
  ans INTEGER;
  BEGIN
  IF x < y THEN
     ans := gcd(y, x);
  ELSIF (x MOD y = 0) THEN
     ans := y;
  ELSE
     ans := gcd(y, x MOD y);
  END IF;
  RETURN ans;
  END gcd;

  PROCEDURE normalize( numerator IN OUT INTEGER,
                       denominator IN OUT INTEGER) IS
   g INTEGER;
   BEGIN
   g := gcd(numerator, denominator);
   numerator := numerator / g;
```

```
   denominator := denominator / g;
   END normalize;

  PROCEDURE plus (n1 INTEGER, d1 INTEGER,
                  n2 INTEGER, d2 INTEGER,
                  n3 OUT INTEGER, d3 OUT INTEGER) IS
  BEGIN
  n3 := n1 * d2 + n2 * d1;
  d3 := d1 * d2;
  END plus;

END rationalP;
```

この例では、JPublisher を次のコマンドラインで起動します。

```
jpub -user=scott/tiger -sql=RationalP -methods=true
```

JPublisher は -user パラメータの指示により、ユーザー scott でパスワード tiger を使用してデータベースにログインします。JPublisher では -methods パラメータの指示により、パッケージ RationalP のメソッド用のラッパーが生成されます。-methods=true パラメータはデフォルトのため省略できます。

JPublisher で生成された RationalP.sqlj のリストと説明

JPublisher では、次のファイル RationalP.sqlj が生成されます。

> **注意 :** JPublisher で生成されるメソッド本体の詳細は、今後のリリースで変更される場合があります。

```
import java.sql.SQLException;
import sqlj.runtime.ref.DefaultContext;
import sqlj.runtime.ConnectionContext;
import java.sql.Connection;

public class RationalP
{
  /* connection management */
  protected DefaultContext __tx = null;
  protected Connection __onn = null;
  public void setConnectionContext(DefaultContext ctx) throws SQLException
  { release(); __tx = ctx; }
  public DefaultContext getConnectionContext() throws SQLException
  { if (__tx==null)
    { __tx = (__onn==null) ? DefaultContext.getDefaultContext() : new
DefaultContext(__onn); }
```

```
    return __tx;
  };
  public Connection getConnection() throws SQLException
  { return (__onn==null) ? ((__tx==null) ? null : __tx.getConnection()) : __onn; }
  public void release() throws SQLException
  { if (__tx!=null && __onn!=null)
__tx.close(ConnectionContext.KEEP_CONNECTION);
    __onn = null; __tx = null;
  }

  /* constructors */
  public RationalP() throws SQLException
  { __tx = DefaultContext.getDefaultContext();
 }
  public RationalP(DefaultContext c) throws SQLException
  { __tx = c; }
  public RationalP(Connection c) throws SQLException
  {__onn = c; __tx = new DefaultContext(c); }

  public Integer gcd (
    Integer x,
    Integer y)
  throws SQLException
  {
    Integer __jPt_result;
    #sql [getConnectionContext()] __jPt_result = { VALUES(SCOTT.RATIONALP.GCD(
      :x,
      :y)) };
    return __jPt_result;
  }
  public void plus (
    Integer n1,
    Integer d1,
    Integer n2,
    Integer d2,
    Integer n3[],
    Integer d3[])
  throws SQLException
  {
    #sql [getConnectionContext()] { CALL SCOTT.RATIONALP.PLUS(
      :n1,
      :d1,
      :n2,
      :d2,
      :OUT (n3[0]),
      :OUT (d3[0])) };
  }
```

```
  public Float toreal (
    Integer numerator,
    Integer denominator)
  throws SQLException
  {
    Float __jPt_result;
    #sql [getConnectionContext()] __jPt_result = {
VALUES(SCOTT.RATIONALP.TOREAL(
      :numerator,
      :denominator)) };
    return __jPt_result;
  }

  public void normalize (
    Integer numerator[],
    Integer denominator[])
  throws SQLException
  {
    #sql [getConnectionContext()] { CALL SCOTT.RATIONALP.NORMALIZE(
      :INOUT (numerator[0]),
      :INOUT (denominator[0])) };
  }
}
```

JPublisher で生成されるすべてのメソッドは、対応する PL/SQL メソッドをコールしサーバー上で実行されます。

デフォルトでは、JPublisher は既存の SQLJ 接続コンテキスト・クラス `sqlj.runtime.ref.DefaultContext` を使用して、そのインスタンスを `RationalP` パッケージに対応付けます。

JPublisher では、ファンクション `gcd` 用のソース・コードが生成され、2 つの `BigDecimal` 値 `x` および `y` を使用して `BigDecimal` の結果が戻されます。このファンクション `gcd` は、`IN` ホスト変数 `:x` および `:y` を使用して、ストアド・ファンクション `RATIONALP.GCD` をコールします。

JPublisher では、プロシージャ `normalize` 用のソース・コードが生成され、2 つの `BigDecimal` 値 `numerator` および `denominator` が使用されます。このプロシージャ `normalize` は、`IN OUT` ホスト変数 `:numerator` および `:denominator` を使用して、ストアド・プロシージャ `RATIONALP.NORMALIZE` をコールします。`IN OUT` パラメータがあるため、その値が配列の最初の要素として渡されます。

JPublisher では、プロシージャ `plus` 用のソース・コードが生成され、4 つの `BigDecimal` `IN` パラメータおよび 2 つの `BigDecimal` `OUT` パラメータが使用されます。このプロシージャ `plus` は、`IN` ホスト変数 `:n1`、`:d1`、`:n2`、`:d2` を使用して、ストアド・プロシージャ `RATIONALP.PLUS` をコールします。また、`OUT` ホスト変数 `:n3` および `:d3` も定義します。`OUT` 変数があるため、その値がそれぞれ配列の最初の要素として渡されます。

JPublisher では、ファンクション toReal 用のソース・コードが生成され、2 つの BigDecimal 値 numerator および denominator を使用して、結果 BigDecimal が戻されます。このファンクション toReal は IN ホスト変数 :numerator および :denominator を使用して、ストアド・ファンクション RATIONALP.TOREAL をコールします。

例 : オブジェクト型用に生成されたクラスの使用

この項では、JPublisher でオブジェクト型用に生成されるクラスの使用例を示します。属性とメソッドを含む SQL オブジェクト型を定義したと想定します。JPublisher を使用して、オブジェクト型用に < 名前 >.sqlj ファイルおよび < 名前 >Ref.java ファイルを生成します。JPublisher でオブジェクト型用に生成された Java クラスの機能を向上させるために、クラスを拡張できます。クラスを変換（該当する場合）およびコンパイルすると、プログラムで使用できます。このトピックの詳細は、2-24 ページの「JPublisher でオブジェクト型用に生成されるクラスの使用」を参照してください。

次のステップは、前述の使用例を実行します。この例では、numerator および denominator 属性と有理数を操作するいくつかのメソッドを含む SQL オブジェクト型 Rational0 を定義します。JPublisher を使用して JPubRational0.sqlj ファイル、Rational0Ref.java ファイルおよび Rational0.sqlj ファイルの初期バージョンを生成した後、Rational0.sqlj ファイルを編集して JPubRational0 クラスの機能を拡張および向上させます。必要なファイルを変換してコンパイルした後に、テスト・ファイルで Rational0.java クラスのパフォーマンスをテストします。

手順の後に、ファイルのリストを示します。

1. SQL オブジェクト型 Rational0 を作成します。4-52 ページの「Rational0.sql のリスト（オブジェクト型の定義）」に Rational0.sql ファイルのコードが示されています。
2. JPublisher を使用してオブジェクト用の Java クラス、つまりサブクラス用の JPubRational0.sqlj ファイル、Rational0Ref.java ファイルおよび初期 Rational0.sqlj ファイルを生成します。次のコマンドラインを使用します。

   ```
   jpub -props=Rational0.props
   ```

   プロパティ・ファイル Rational0.props に次の内容が含まれているとします。

   ```
   jpub.user=scott/tiger
   jpub.sql=Rational0:JPubRational0:Rational0
   jpub.methods=true
   ```

   プロパティ・ファイルに従い、JPublisher はユーザー名 scott およびパスワード tiger を使用してデータベースにログインします。sql パラメータの指示により、JPublisher でオブジェクト型 Rational0（Rational0.sql で宣言済み）が変換され、JPubRational0 が Rational0 として生成されます。この 2 つ目の Rational0 は、元の Rational0 の機能を拡張するサブクラス（Rational0.sqlj）です。methods パラメータの値は、JPublisher で PL/SQL パッケージおよびラッパー・メソッド用のクラスを生成することを示します。

JPublisher では次のファイルが生成されます。

```
JPubRational0.sqlj
Rational0Ref.java
Rational0.sqlj
```

これらのファイルのリストは、次の項を参照してください。

3. Rational0.sqlj を編集し、JPubRational0.sqlj の機能を拡張して向上させます。特に、toString() メソッドのコードを追加します。このメソッドは、テスト・プログラム TestRational0.java で最後の 2 つの System.out.println() コールで使用します。

4. SQLJ を使用して必要なファイルを変換およびコンパイルします。次のように入力します。

   ```
   sqlj JPubRational0.sqlj Rational0.sqlj
   ```

   これにより、JPubRational0.sqlj ファイルと Rational0.sqlj ファイルが変換およびコンパイルされます。

5. Rational0 クラスを使用するプログラム TestRational0.java を作成します。TestRational0.java のコードは、4-60 ページの「ユーザー作成の TestRational0.java のリスト」を参照してください。

6. ファイル connect.properties を作成します。このファイルは、TestRational0 でデータベースへの接続方法を決定するときに使用します。このファイルの内容は次のとおりです。

   ```
   sqlj.user=scott
   sqlj.password=tiger
   sqlj.url=jdbc:oracle:oci:@
   sqlj.driver=oracle.jdbc.driver.OracleDriver
   ```

7. TestRational0 をコンパイルして実行します。

   ```
   javac TestRational0.java
   java TestRational0
   ```

   プログラムでは次の出力が生成されます。

   ```
   gcd: 5
   real value: 0.5
   sum: 100/100
   sum: 1/1
   ```

Rational0.sql のリスト（オブジェクト型の定義）

この項では、SQL オブジェクト型 Rational0 を定義するコードを示します。

```
CREATE TYPE Rational0 AS OBJECT (
   numerator INTEGER,
   denominator INTEGER,
   MAP MEMBER FUNCTION toReal RETURN REAL,
   MEMBER PROCEDURE normalize,
   STATIC FUNCTION gcd(x INTEGER,
                       y INTEGER) RETURN INTEGER,
   MEMBER FUNCTION plus ( x Rational0) RETURN Rational0
);

CREATE TYPE BODY Rational0 AS

  MAP MEMBER FUNCTION toReal RETURN REAL IS
  -- convert rational number to real number
  BEGIN
    RETURN numerator / denominator;
  END toReal;

  MEMBER PROCEDURE normalize IS
   g BINARY_INTEGER;
  BEGIN
   g := Rational0.gcd(numerator, denominator);
   numerator := numerator / g;
   denominator := denominator / g;
  END normalize;

  STATIC FUNCTION gcd(x INTEGER,
                      y INTEGER) RETURN INTEGER IS
  -- find greatest common divisor of x and y
  ans BINARY_INTEGER;
  BEGIN
  IF x < y THEN
     ans := Rational0.gcd(y, x);
  ELSIF (x MOD y = 0) THEN
     ans := y;
  ELSE
     ans := Rational0.gcd(y, x MOD y);
  END IF;
  RETURN ans;
  END gcd;

  MEMBER FUNCTION plus (x Rational0) RETURN Rational0 IS
  BEGIN
   return Rational0(numerator * x.denominator + x.numerator * denominator,
```

```
                    denominator * x.denominator);
  END plus;
END;
```

JPublisher で生成された JPubRational0.sqlj のリスト

この項では、JPublisher で生成される JPubRational0.java のコードをリストします。

```
import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sql.ORAData;
import oracle.sql.ORADataFactory;
import oracle.sql.Datum;
import oracle.sql.STRUCT;
import oracle.jpub.runtime.MutableStruct;
import sqlj.runtime.ref.DefaultContext;
import sqlj.runtime.ConnectionContext;
import java.sql.Connection;

public class JPubRational0 implements ORAData, ORADataFactory
{
  public static final String _SQL_NAME = "SCOTT.RATIONAL0";
  public static final int _SQL_TYPECODE = OracleTypes.STRUCT;

  /* connection management */
  protected DefaultContext __tx = null;
  protected Connection __onn = null;
  public void setConnectionContext(DefaultContext ctx) throws SQLException
  { release(); __tx = ctx; }
  public DefaultContext getConnectionContext() throws SQLException
  { if (__tx==null)
    { __tx = (__onn==null) ? DefaultContext.getDefaultContext() : new
DefaultContext(__onn); }
    return __tx;
  };
  public Connection getConnection() throws SQLException
  { return (__onn==null) ? ((__tx==null) ? null : __tx.getConnection()) : __onn;
}
  public void release() throws SQLException
  { if (__tx!=null && __onn!=null)
__tx.close(ConnectionContext.KEEP_CONNECTION);
    __onn = null; __tx = null;
  }

  protected MutableStruct _struct;
```

```
  private static int[] _sqlType =  { 4,4 };
  private static ORADataFactory[] _factory = new ORADataFactory[2];
  protected static final JPubRational0 _JPubRational0Factory = new
JPubRational0(false);

  public static ORADataFactory getORADataFactory()
  { return _JPubRational0Factory; }
  /* constructors */
  protected JPubRational0(boolean init)
  { if (init) _struct = new MutableStruct(new Object[2], _sqlType, _factory); }
  public JPubRational0()
  { this(true); __tx = DefaultContext.getDefaultContext(); }
  public JPubRational0(DefaultContext c) /*throws SQLException*/
  { this(true); __tx = c; }
  public JPubRational0(Connection c) /*throws SQLException*/
  { this(true); __onn = c; }
  public JPubRational0(Integer numerator, Integer denominator) throws SQLException
   {
    this(true);
    setNumerator(numerator);
    setDenominator(denominator);
  }

  /* ORAData interface */
  public Datum toDatum(Connection c) throws SQLException
  {
    if (__tx!=null && __onn!=c) release();
    __onn = c;
    return _struct.toDatum(c, _SQL_NAME);
  }

  /* ORADataFactory interface */
  public ORAData create(Datum d, int sqlType) throws SQLException
  { return create(null, d, sqlType); }
  public void setFrom(JPubRational0 o) throws SQLException
  { setContextFrom(o); setValueFrom(o); }
  protected void setContextFrom(JPubRational0 o) throws SQLException
  { release(); __tx = o.__tx; __onn = o.__onn; }
  protected void setValueFrom(JPubRational0 o) { _struct = o._struct; }
  protected ORAData create(JPubRational0 o, Datum d, int sqlType) throws
SQLException
  {
    if (d == null) { if (o!=null) { o.release(); }; return null; }
    if (o == null) o = new JPubRational0(false);
    o._struct = new MutableStruct((STRUCT) d, _sqlType, _factory);
    o.__onn = ((STRUCT) d).getJavaSqlConnection();
```

```
  return o;
}
/* accessor methods */
public Integer getNumerator() throws SQLException
{ return (Integer) _struct.getAttribute(0); }

public void setNumerator(Integer numerator) throws SQLException
{ _struct.setAttribute(0, numerator); }

public Integer getDenominator() throws SQLException
{ return (Integer) _struct.getAttribute(1); }

public void setDenominator(Integer denominator) throws SQLException
{ _struct.setAttribute(1, denominator); }

public Integer gcd (
  Integer x,
  Integer y)
throws SQLException
{
  Integer __jPt_result;
  #sql [getConnectionContext()] __jPt_result = { VALUES(SCOTT.RATIONAL0.GCD(
    :x,
    :y)) };
  return __jPt_result;
}

public Rational0 normalize ()
throws SQLException
{
  Rational0 __jPt_temp = (Rational0) this;
  #sql [getConnectionContext()] {
    BEGIN
    :INOUT __jPt_temp.NORMALIZE();
    END;
  };
  return __jPt_temp;
}

public Rational0 plus (
  Rational0 x)
throws SQLException
{
  JPubRational0 __jPt_temp = this;
  Rational0 __jPt_result;
  #sql [getConnectionContext()] {
    BEGIN
```

```
      :OUT __jPt_result := :__jPt_temp.PLUS(
      :x);
      END;
    };
    return __jPt_result;
  }

  public Float toreal ()
  throws SQLException
  {
    JPubRational0 __jPt_temp = this;
    Float __jPt_result;
    #sql [getConnectionContext()] {
      BEGIN
      :OUT __jPt_result := :__jPt_temp.TOREAL();
      END;
    };
    return __jPt_result;
  }
}
```

JPublisher で生成された RationalORef.java のリスト

この項では、JPublisher で生成される `RationalORef.java` のコードをリストします。

> **注意 :** JPublisher で生成されるメソッド本体の詳細は、今後のリリースで変更される場合があります。

```
import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sql.ORAData;
import oracle.sql.ORADataFactory;
import oracle.sql.Datum;
import oracle.sql.REF;
import oracle.sql.STRUCT;

public class RationalORef implements ORAData, ORADataFactory
{
  public static final String _SQL_BASETYPE = "SCOTT.RATIONALO";
  public static final int _SQL_TYPECODE = OracleTypes.REF;

  REF _ref;
```

```
private static final RationalORef _RationalORefFactory = new RationalORef();

  public static ORADataFactory getORADataFactory()
  { return _RationalORefFactory; }
  /* constructor */
  public RationalORef()
  {
  }

  /* ORAData interface */
  public Datum toDatum(Connection c) throws SQLException
  {
    return _ref;
  }

  /* ORADataFactory interface */
  public ORAData create(Datum d, int sqlType) throws SQLException
  {
    if (d == null) return null;
    RationalORef r = new RationalORef();
    r._ref = (REF) d;
    return r;
  }

  public static RationalORef cast(ORAData o) throws SQLException
  {
     if (o == null) return null;
     try { return (RationalORef) getORADataFactory().create(o.toDatum(null),
OracleTypes.REF); }
     catch (Exception exn)
     { throw new SQLException("Unable to convert "+o.getClass().getName()+" to
RationalORef: "+exn.toString()); }
  }

  public RationalO getValue() throws SQLException
  {
     return (RationalO) RationalO.getORADataFactory().create(
       _ref.getSTRUCT(), OracleTypes.REF);
  }

  public void setValue(RationalO c) throws SQLException
  {
    _ref.setValue((STRUCT) c.toDatum(_ref.getJavaSqlConnection()));
  }
}
```

JPublisher で生成されてユーザーによって変更される Rational0.sqlj のリスト

この項では、JPublisher によって生成されたスーパークラス JpubRational0 を拡張する Rational0 クラスのコードをリストします。これはデフォルト・モード (-gensubclass=true) の場合で、クラスの初期 .sqlj ソース・ファイルが生成されます。このソース・ファイルを必要に応じて変更します。

通常、ユーザー記述のサブクラスでは、次のことを行う必要があります。

- ファクトリ・オブジェクト _JPubRational0 を宣言します。
- getORADataFactory() メソッドを実装します。
- create() メソッドを実装します。
- スーパークラスでコンストラクタをコールして、コンストラクタを実装します。

また、このサブクラスには、toString() メソッドも必要です。このメソッドは、TestRational0.java の最後の 2 つの System.out.println() コールで使用します (4-60 ページの「ユーザー作成の TestRational0.java のリスト」を参照してください)。生成されたコードの最後にある「手動でコーディングする toString() メソッド」を参照してください。

JPublisher によって生成されるコード

この項では、JPublisher によって生成される Rational0.sqlj ソース・コードをリストします。

```
import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sql.ORAData;
import oracle.sql.ORADataFactory;
import oracle.sql.Datum;
import oracle.sql.STRUCT;
import oracle.jpub.runtime.MutableStruct;
import sqlj.runtime.ref.DefaultContext;
import sqlj.runtime.ConnectionContext;
import java.sql.Connection;

public class Rational0 extends JPubRational0 implements ORAData, ORADataFactory
{
   private static final Rational0 _Rational0Factory = new Rational0(false);
   public static ORADataFactory getORADataFactory()
   { return _Rational0Factory; }

   public Rational0() { super(); }
   public Rational0(Connection conn) throws SQLException { super(conn); }
   public Rational0(DefaultContext ctx) throws SQLException { super(ctx); }
   protected Rational0(boolean init) { super(init); }
```

```
  public Rational0(Integer numerator, Integer denominator) throws SQLException
  {
    setNumerator(numerator);
    setDenominator(denominator);
  }
   /* ORAData interface */
   public ORAData create(Datum d, int sqlType) throws SQLException
   { return create(new Rational0(false), d, sqlType); }

  /* superclass accessors */

/*
  public Integer getNumerator() throws SQLException { return super.getNumerator(); }
  public void setNumerator(Integer numerator) throws SQLException {
super.setNumerator(numerator); }
 */

/*
  public Integer getDenominator() throws SQLException { return
super.getDenominator(); }
  public void setDenominator(Integer denominator) throws SQLException {
super.setDenominator(denominator); }
 */

  /* superclass methods */
/*
  public Integer gcd(Integer x, Integer y) throws SQLException
  { return super.gcd(x, y); }
 */
/*
  public Rational0 normalize() throws SQLException
  { return super.normalize(); }
 */
/*
  public Rational0 plus(Rational0 x) throws SQLException
  { return super.plus(x); }
 */
/*
  public Float toreal() throws SQLException
  { return super.toreal(); }
 */
}
```

手動でコーディングする toString() メソッド

この項では、TestRational0 に必要な toString() メソッドを示します。この例では、JPublisher で生成された Rational0.sqlj ソース・ファイルに、このメソッドの定義を追加する必要があります。

または、JPublisher オプション設定 -tostring=true を使用すると、JPublisher で Java オブジェクト型ラッパーに toString() メソッドが自動的に生成されます。

```
/* additional method not in base class */
public String toString()
{
  try
  {
     return getNumerator().toString() + "/" + getDenominator().toString();
  }
  catch (SQLException e)
  {
     return null;
  }
}
```

ユーザー作成の TestRational0.java のリスト

この項では、numerator および denominator の初期値を指定して、Rational0 クラスのパフォーマンスをテストする、ユーザー作成ファイル TestRational0.java の内容をリストします。TestRational0.java ファイルには、次のタスクの実行方法も示されています。

- Oracle.connect() メソッドをコールしてデータベースに接続します。
- SQL オブジェクト型を表す Java オブジェクトを宣言し、その属性を設定して初期化します。
- オブジェクトを使用してサーバー・メソッドをコールします。

```
import oracle.sqlj.runtime.Oracle;
import oracle.sql.Datum;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Driver;

public class TestRational0
{

  public static void main(String[] args)
  throws java.sql.SQLException
```

```
  {
    Oracle.connect(new TestRational0().getClass(),
                   "connect.properties");

    Rational0 r = new Rational0();

    Integer n = new Integer(5);
    Integer d = new Integer(10);

    r.setNumerator(n);
    r.setDenominator(d);

    Integer g = r.gcd(n, d);
    System.out.println("gcd: " + g);

    Float f = r.toreal();
    System.out.println("real value: " + f);

    Rational0 s = r.plus(r);
    System.out.println("sum: " + s);

    s = s.normalize();
    System.out.println("sum: " + s);
  }
}
```

例 : パッケージ用に生成されたクラスの使用

この項では、オブジェクトおよびパッケージに対して JPublisher で生成されるクラスおよびラッパー・メソッドの使用例を示します。属性を含む SQL オブジェクト型およびメソッドを含むパッケージを定義すると想定します。JPublisher を使用して、オブジェクトおよびパッケージ用の < 名前 >.sqlj ファイルを生成します。クラスを変換した後、プログラムで使用可能になります。このトピックの詳細は、2-23 ページの「JPublisher で PL/SQL パッケージ用に生成される SQLJ クラスの使用」を参照してください。

次のステップは、前述の使用例を実行します。ここでは、有理数を操作する、numerator および denominator 整数属性を含む SQL オブジェクト型 Rational とメソッドを持つパッケージ RationalP を定義します。JPublisher で Rational.sqlj ファイルおよび RationalP.sqlj ファイルを生成した後に、それらを SQLJ で変換した後、テスト・ファイルで Rational クラスおよび RationalP クラスのパフォーマンスをテストします。

手順の後に、ファイルのリストを示します。

1. SQL オブジェクト型 Rational およびパッケージ RationalP を作成します。4-63 ページの「RationalP.sql のリスト（オブジェクト型およびパッケージの定義）」に RationalP.sql ファイルの SQL コードが示されています。

2. JPublisher を使用して、オブジェクト用およびパッケージ用の Java クラス・ファイルと SQLJ クラス・ファイル（Rational.java および RationalP.sqlj）をそれぞれ生成します。次のコマンドラインを使用します。

   ```
   jpub -props=RationalP.props
   ```

   プロパティ・ファイル RationalP.props に次の内容が含まれているとします。

   ```
   jpub.user=scott/tiger
   jpub.sql=RationalP,Rational
   jpub.mapping=oracle
   jpub.methods=true
   ```

   プロパティ・ファイルに従い、JPublisher はユーザー名 scott およびパスワード tiger を使用してデータベースにログインします。sql パラメータの指示により、JPublisher でオブジェクト型 Rational および RationalP.sql で宣言されたパッケージ RationalP が変換されます。JPublisher では、oracle マッピングに従って型およびパッケージが変換されます。methods パラメータの値は、JPublisher でラッパー・メソッドを含む PL/SQL パッケージ用のクラスを生成することを示します。オブジェクト型 Rational にはメンバー関数がないため、JPublisher では .sqlj ファイルではなく .java ファイルに変換されます。ただし、JPublisher 用の -methods=always 設定を使用すると、常に .sqlj ファイルを生成するように要求できます。詳細は、3-20 ページの「パッケージのクラスおよびラッパー・メソッドの生成（-methods）」を参照してください。

   JPublisher では次のファイルが生成されます。

   ```
   Rational.java
   RationalP.sqlj
   ```

3. RationalP.sqlj ファイルおよび Rational.java ファイルを次のように変換してコンパイルします。

   ```
   sqlj RationalP.sqlj Rational.java
   ```

4. RationalP クラスを使用するプログラム TestRationalP.java を作成します。

5. connect.properties を作成します。このファイルは、TestRationalP.java でデータベースへの接続方法を決定する際に使用します。このファイルの内容は次のとおりです。

   ```
   sqlj.user=scott
   sqlj.password=tiger
   sqlj.url=jdbc:oracle:oci:@
   sqlj.driver=oracle.jdbc.driver.OracleDriver
   ```

6. TestRationalP をコンパイルして実行します。

   ```
   javac TestRationalP.java
   java TestRationalP
   ```

プログラムでは次の出力が生成されます。

```
gcd: 5
real value: 0.5
sum: 100/100
sum: 1/1
```

RationalP.sql のリスト（オブジェクト型およびパッケージの定義）

この項では、Rational SQL オブジェクト型および RationalP パッケージを定義するファイル RationalP.sql の内容をリストします。

```
CREATE TYPE Rational AS OBJECT (
   numerator INTEGER,
   denominator INTEGER
);
/
CREATE PACKAGE RationalP AS

 FUNCTION toReal(r Rational) RETURN REAL;

 PROCEDURE normalize(r IN OUT Rational);

 FUNCTION gcd(x INTEGER, y INTEGER) RETURN INTEGER;

 FUNCTION plus (r1 Rational, r2 Rational) RETURN Rational;

END rationalP;
/
CREATE PACKAGE BODY rationalP AS

  FUNCTION toReal(r Rational) RETURN real IS
  -- convert rational number to real number
  BEGIN
    RETURN r.numerator / r.denominator;
  END toReal;

  FUNCTION gcd(x INTEGER, y INTEGER) RETURN INTEGER IS
  -- find greatest common divisor of x and y
  result INTEGER;
  BEGIN
  IF x < y THEN
     result := gcd(y, x);
  ELSIF (x MOD y = 0) THEN
     result := y;
  ELSE
     result := gcd(y, x MOD y);
```

```
  END IF;
  RETURN result;
  END gcd;

  PROCEDURE normalize( r IN OUT Rational) IS
   g INTEGER;
   BEGIN
   g := gcd(r.numerator, r.denominator);
   r.numerator := r.numerator / g;
   r.denominator := r.denominator / g;
   END normalize;

  FUNCTION plus (r1 Rational,
                 r2 Rational) RETURN Rational IS
  n INTEGER;
  d INTEGER;
  result Rational;
  BEGIN
  n := r1.numerator * r2.denominator + r2.numerator * r1.denominator;
  d := r1.denominator * r2.denominator;
  result := Rational(n, d);
  RETURN result;
  END plus;

END rationalP;
/
```

ユーザー作成の TestRationalP.java のリスト

テスト・プログラム TestRationalP.java では、パッケージ RationalP およびメソッドを持たないオブジェクト型 Rational を使用します。テスト・プログラムでは、パッケージ RationalP のインスタンスおよび 2 つの Rational オブジェクトが作成されます。

TestRationalP は、Oracle SQLJ の Oracle.connect() メソッドを介してデータベースに接続します。この例では、Oracle.connect() は次の接続プロパティを含むファイル connect.properties を指定します。

```
sqlj.url=jdbc:oracle:oci:@
sqlj.user=scott
sqlj.password=tiger
```

次に、TestRationalP.java をリストします。

```
import oracle.sql.Datum;
import oracle.sql.NUMBER;
import java.math.BigDecimal;
import sqlj.runtime.ref.DefaultContext;
```

```
import oracle.sqlj.runtime.Oracle;
import java.sql.Connection;

public class TestRationalP
{

  public static void main(String[] args)
  throws java.sql.SQLException
  {

    Oracle.connect(new TestRationalP().getClass(),
                   "connect.properties");

    RationalP p = new RationalP();

    NUMBER n = new NUMBER(5);
    NUMBER d = new NUMBER(10);
    Rational r = new Rational();
    r.setNumerator(n);
    r.setDenominator(d);

    NUMBER f = p.toreal(r);
    System.out.println("real value: " + f.stringValue());

    NUMBER g = p.gcd(n, d);
    System.out.println("gcd: " + g.stringValue());

    Rational s = p.plus(r, r);
    System.out.println("sum: " + s.getNumerator().stringValue() +
                           "/" + s.getDenominator().stringValue());

    Rational[] sa = {s};
    p.normalize(sa);
    s = sa[0];
    System.out.println("sum: " + s.getNumerator().stringValue() +
                           "/" + s.getDenominator().stringValue());
  }
}
```

例 : JDBC でサポートされないデータ型の使用

JPublisher には、PL/SQL 固有で Java から直接アクセスできない型を使用できるように、多数のメカニズムが用意されています。この例では、オブジェクト・メソッドに PL/SQL の BOOLEAN 型を使用する SQL オブジェクト型を設定します。

この型を JPublisher で直接公開する場合と、この型の変換を手動で記述する場合を比較します。JPublisher では BOOLEAN 型を自動的に処理できるため、最も迅速に結果が得られるアプローチについて疑問の余地はありません。ただし、手動によるアプローチは、JPublisher でも利用されている基本的な変換の概念の好例です。また、事前定義済みの変換がない型の場合は、対応する SQL 型と変換ファンクションを作成する必要があることに注意してください。この操作を特定の型について行うと、JPublisher に型マップ・エントリを提供できます。JPublisher では、この情報を使用して型が発生するすべてのメソッドが適切にマップされます。

ユーザー定義の BOOLEANS データ型

次の .sql ファイルでは、PL/SQL の BOOLEAN 引数を使用するメソッドを持つオブジェクト型を定義します。このプログラムで使用される基本的なメソッドは、引数が正しく渡されることを示すために使用されています。

> **注意 :** ユーザー定義の BOOLEANS オブジェクト型を PL/SQL の BOOLEAN 型と混同しないでください。

```
CREATE TYPE BOOLEANS AS OBJECT (
 iIn     INTEGER,
 iInOut  INTEGER,
 iOut    INTEGER,

 MEMBER PROCEDURE p(i1 IN BOOLEAN,
                    i2 IN OUT BOOLEAN,
                    i3 OUT BOOLEAN),

 MEMBER FUNCTION f(i1 IN BOOLEAN) RETURN BOOLEAN
);

CREATE TYPE BODY BOOLEANS AS

MEMBER PROCEDURE p(i1 IN BOOLEAN,
                   i2 IN OUT BOOLEAN,
                   i3 OUT BOOLEAN) IS
BEGIN
  iOut := iIn;

  IF iInOut IS NULL THEN
```

```
    iInOut := 0;
  ELSIF iInOut = 0 THEN
    iInOut := 1;
  ELSE
    iInOut := NULL;
  END IF;

  i3 := i1;
  i2 := NOT i2;
END;

MEMBER FUNCTION f(i1 IN BOOLEAN) RETURN BOOLEAN IS
BEGIN
  return i1 = (iIn = 1);
END;

END;
```

代替方法 1: プロセス全体への JPublisher の使用

BOOLEAN 用の変換は PL/SQL の BOOLEAN と SQL の INTEGER の間で変換する SYS.SQLJUTL パッケージ内で定義されているため、BOOLEANS オブジェクト型を次の JPublisher コマンドラインのように直接公開できます。また、SQL の INTEGER 自体は Java の boolean に直接マップ可能であるため、もともとの対応付けがあります。JPublisher で Boolean.sqlj に生成される SQLJ コードを使用する前に、PL/SQL ラッパー・スクリプトをインストールする必要があることに注意してください。

```
jpub -u scott/tiger -s BOOLEANS:Booleans -plsqlfile=BWrap.sql
-plsqlpackage=B_WRAP
sqljplus scott/tiger @BWrap.sql
```

2-10 ページの「PL/SQL 変換ファンクションを介した型マッピングのサポート」で説明したように、JPublisher のデフォルトの型マップでは、PL/SQL の BOOLEAN が Java の boolean に関連付けられています。null データの表現機能を保つために、かわりに Java オブジェクト型 Boolean へのマッピングを使用することが必要な場合があります。そのためには、デフォルトの型マップを再定義します。

完全を期すために、JPublisher で生成されるファイル Booleans.sqlj の内容を次に示します。

```
import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sql.ORAData;
import oracle.sql.ORADataFactory;
import oracle.sql.Datum;
```

```
import oracle.sql.STRUCT;
import oracle.jpub.runtime.MutableStruct;
import sqlj.runtime.ref.DefaultContext;
import sqlj.runtime.ConnectionContext;
import java.sql.Connection;

public class Booleans implements ORAData, ORADataFactory
{
  public static final String _SQL_NAME = "SCOTT.BOOLEANS";
  public static final int _SQL_TYPECODE = OracleTypes.STRUCT;

  /* connection management */
  protected DefaultContext __tx = null;
  protected Connection __onn = null;
  public void setConnectionContext(DefaultContext ctx) throws SQLException
  { release(); __tx = ctx; }
  public DefaultContext getConnectionContext() throws SQLException
  { if (__tx==null)
    { __tx = (__onn==null) ? DefaultContext.getDefaultContext() : new
DefaultContext(__onn); }
    return __tx;
  };
  public Connection getConnection() throws SQLException
  { return (__onn==null) ? ((__tx==null) ? null : __tx.getConnection()) : __onn;
}
  public void release() throws SQLException
  { if (__tx!=null && __onn!=null)
__tx.close(ConnectionContext.KEEP_CONNECTION);
    __onn = null; __tx = null;
  }

  protected MutableStruct _struct;

  private static int[] _sqlType =  { 4,4,4 };
  private static ORADataFactory[] _factory = new ORADataFactory[3];
protected static final Booleans _BooleansFactory = new Booleans(false);

  public static ORADataFactory getORADataFactory()
  { return _BooleansFactory; }
  /* constructors */
  protected Booleans(boolean init)
  { if (init) _struct = new MutableStruct(new Object[3], _sqlType, _factory); }
  public Booleans()
  { this(true); __tx = DefaultContext.getDefaultContext(); }
  public Booleans(DefaultContext c) /*throws SQLException*/
  { this(true); __tx = c; }
  public Booleans(Connection c) /*throws SQLException*/
```

```
{ this(true); __onn = c; }
public Booleans(Integer iin, Integer iinout, Integer iout) throws SQLException
 {
  this(true);
  setIin(iin);
  setIinout(iinout);
  setIout(iout);
}

/* ORAData interface */
public Datum toDatum(Connection c) throws SQLException
{
  if (__tx!=null && __onn!=c) release();
  __onn = c;
  return _struct.toDatum(c, _SQL_NAME);
}


/* ORADataFactory interface */
public ORAData create(Datum d, int sqlType) throws SQLException
{ return create(null, d, sqlType); }
public void setFrom(Booleans o) throws SQLException
{ setContextFrom(o); setValueFrom(o); }
protected void setContextFrom(Booleans o) throws SQLException
{ release(); __tx = o.__tx; __onn = o.__onn; }
protected void setValueFrom(Booleans o) { _struct = o._struct; }
protected ORAData create(Booleans o, Datum d, int sqlType) throws SQLException
{
  if (d == null) { if (o!=null) { o.release(); }; return null; }
  if (o == null) o = new Booleans(false);
  o._struct = new MutableStruct((STRUCT) d, _sqlType, _factory);
  o.__onn = ((STRUCT) d).getJavaSqlConnection();
  return o;
}
/* accessor methods */
public Integer getIin() throws SQLException
{ return (Integer) _struct.getAttribute(0); }

public void setIin(Integer iin) throws SQLException
{ _struct.setAttribute(0, iin); }


public Integer getIinout() throws SQLException
{ return (Integer) _struct.getAttribute(1); }

public void setIinout(Integer iinout) throws SQLException
{ _struct.setAttribute(1, iinout); }
```

```
  public Integer getIout() throws SQLException
  { return (Integer) _struct.getAttribute(2); }

  public void setIout(Integer iout) throws SQLException
  { _struct.setAttribute(2, iout); }


  public boolean  f (
    boolean  i1)
  throws SQLException
  {
    Booleans __jPt_temp = this;
    boolean  __jPt_result;
    #sql [getConnectionContext()] {
      BEGIN
      :OUT __jPt_result := SYS.SQLJUTL.BOOL2INT(:__jPt_temp.F(
      SYS.SQLJUTL.INT2BOOL(:i1)));
      END;
    };
    return __jPt_result;
  }

  public Booleans p (
    boolean  i1,
    boolean  i2[],
    boolean  i3[])
  throws SQLException
  {
    Booleans __jPt_temp = this;
    #sql [getConnectionContext()] {
      BEGIN
      B_WRAP.BOOLEANS$P(:INOUT __jPt_temp,
      :i1,
      :INOUT (i2[0]),
      :OUT (i3[0]));
      END;
    };
    return __jPt_temp;
  }
}
```

JPublisher で生成されるファイル `BWrap.sql` の内容は、次のとおりです。このファイルには、PL/SQL ラッパー・コードが含まれています。JPublisher でラッパーを生成する必要が

あるのは、PL/SQL 引数が IN OUT または OUT パラメータとして使用される場合のみであることに注意してください。

```
CREATE OR REPLACE PACKAGE B_WRAP AS
   PROCEDURE BOOLEANS$P (SELF_ IN OUT SCOTT.BOOLEANS,I1 INTEGER,I2 IN OUT INTEGER,I3
OUT INTEGER);
END B_WRAP;
/
CREATE OR REPLACE PACKAGE BODY B_WRAP IS

   PROCEDURE BOOLEANS$P (SELF_ IN OUT SCOTT.BOOLEANS,I1 INTEGER,I2 IN OUT INTEGER,I3
OUT INTEGER) IS
      I1_ BOOLEAN;
      I2_ BOOLEAN;
      I3_ BOOLEAN;
   BEGIN
      I1_ := SYS.SQLJUTL.INT2BOOL(I1);
      I2_ := SYS.SQLJUTL.INT2BOOL(I2);
      SELF_.P(I1_,I2_,I3_);
      I2 := SYS.SQLJUTL.BOOL2INT(I2_);
      I3 := SYS.SQLJUTL.BOOL2INT(I3_);
   END BOOLEANS$P;

END B_WRAP;
/
```

代替方法 2: 手動変換

JDBC でサポートされないデータ型を使用する代替方法に、無名 PL/SQL ブロックを作成して JDBC でサポートされる入力タイプを PL/SQL メソッドで使用する入力タイプに変換する方法があります。作成後、PL/SQL メソッドで使用される出力タイプを、JDBC でサポートされる出力タイプに変換します。このトピックの詳細は、2-7 ページの「JDBC でサポートされないデータ型の使用」を参照してください。

次のステップでは、この変換を実行するための一般的な方法を示します。このステップでは、JDBC でサポートされていないデータ型の引数を含むメソッドを持つオブジェクト型を JPublisher で変換したと想定します。このステップでは JDBC でサポートされていないデータ型を使用するために必要な変更を説明します。クラスを拡張するか、または生成されたファイルを修正して変更を加えることができます。通常は、クラスを拡張するのがよりよい方法ですが、この例では生成されたファイルを変更します。

1. Java で、JDBC でサポートされない型を持つ各 `IN` または `IN OUT` 引数を、JDBC でサポートされる Java 型に変換します。
2. 各 `IN` または `IN OUT` 引数を PL/SQL ブロックに渡します。

3. PL/SQL ブロックでは、各 IN または IN OUT 引数を PL/SQL メソッドに対して適切な型に変換します。
4. PL/SQL メソッドをコールします。
5. 各 OUT 引数、IN OUT 引数またはファンクションの結果を、PL/SQL 上で、JDBC でサポートされない型から JDBC でサポートされる適切な型に変換します。
6. 各 OUT 引数、または IN OUT 引数、あるいはファンクションの結果を PL/SQL ブロックから戻します。
7. Java で、各 OUT 引数または IN OUT 引数あるいはファンクションの結果を、JDBC でサポートされる型から JDBC でサポートされない型に変換します。

次に、JDBC で直接サポートされない引数型の処理例を示します。この例では、JDBC でサポートされない型（Boolean/BOOLEAN）と JDBC でサポートされる型（String/VARCHAR2）間での変換を示します。

次の .sqlj ファイルは最初に JPublisher で生成され、前述のステップに従ってユーザーが変更したものです。ラッパー・メソッドは次の作業を実行します。

- 各引数を Boolean から Java の String に変換します。
- 各引数を PL/SQL ブロックに渡します。
- 引数を PL/SQL で VARCHAR2 から BOOLEAN に変換します。
- PL/SQL メソッドをコールします。
- 各 OUT 引数、または IN OUT 引数、あるいはファンクションの結果を PL/SQL で BOOLEAN から VARCHAR2 に変換します。
- 各 OUT 引数、または IN OUT 引数、あるいはファンクションの結果を PL/SQL ブロックから戻します。
- 最後に、各 OUT 引数、または IN OUT 引数、あるいはファンクションの結果を変換します。

このコードを次に示します。

```
import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sql.ORAData;
import oracle.sql.ORADataFactory;
import oracle.sql.Datum;
import oracle.sql.STRUCT;
import oracle.jpub.runtime.MutableStruct;
import sqlj.runtime.ref.DefaultContext;
import sqlj.runtime.ConnectionContext;
import java.sql.Connection;
```

```
public class Booleans implements ORAData, ORADataFactory
{
  public static final String _SQL_NAME = "SCOTT.BOOLEANS";
  public static final int _SQL_TYPECODE = OracleTypes.STRUCT;

  /* connection management */
  protected DefaultContext __tx = null;
  protected Connection __onn = null;
  public void setConnectionContext(DefaultContext ctx) throws SQLException
  { release(); __tx = ctx; }
  public DefaultContext getConnectionContext() throws SQLException
  { if (__tx==null)
    { __tx = (__onn==null) ? DefaultContext.getDefaultContext() : new
DefaultContext(__onn); }
    return __tx;
  };
  public Connection getConnection() throws SQLException
  { return (__onn==null) ? ((__tx==null) ? null : __tx.getConnection()) : __onn;
}
  public void release() throws SQLException
  { if (__tx!=null && __onn!=null)
__tx.close(ConnectionContext.KEEP_CONNECTION);
    __onn = null; __tx = null;
  }

  protected MutableStruct _struct;

  static int[] _sqlType =
  {
    4, 4, 4
  };

  static ORADataFactory[] _factory = new ORADataFactory[3];

  static final Booleans _BooleansFactory = new Booleans(false);
  public static ORADataFactory getORADataFactory()
  {
    return _BooleansFactory;
  }

  /* constructors */
  protected Booleans(boolean init)
  { if (init) _struct = new MutableStruct(new Object[3], _sqlType, _factory); }
  public Booleans()
  { this(true); __tx = DefaultContext.getDefaultContext(); }
  public Booleans(DefaultContext c) throws SQLException
  { this(true); __tx = c; }
```

```
public Booleans(Connection c) throws SQLException
{ this(true); __onn = c; }

/* ORAData interface */
public Datum toDatum(Connection c) throws SQLException
{
  if (__tx!=null && __onn!=c) release();
  __onn = c;
  return _struct.toDatum(c, _SQL_NAME);
}

/* ORADataFactory interface */
public ORAData create(Datum d, int sqlType) throws SQLException
{ return create(null, d, sqlType); }
public void setFrom(Booleans o) throws SQLException
{ release(); _struct = o._struct; __tx = o.__tx; __onn = o.__onn; }
protected void setValueFrom(Booleans o) { _struct = o._struct; }
protected ORAData create(Booleans o, Datum d, int sqlType) throws SQLException
{
  if (d == null) { if (o!=null) { o.release(); }; return null; }
   if (o == null) o = new Booleans(false);
  o._struct = new MutableStruct((STRUCT) d, _sqlType, _factory);
  o.__onn = ((STRUCT) d).getJavaSqlConnection();
  return o;
}

/* accessor methods */
public Integer getIin() throws SQLException
{ return (Integer) _struct.getAttribute(0); }

public void setIin(Integer iin) throws SQLException
{ _struct.setAttribute(0, iin); }


public Integer getIinout() throws SQLException
{ return (Integer) _struct.getAttribute(1); }

public void setIinout(Integer iinout) throws SQLException
{ _struct.setAttribute(1, iinout); }


public Integer getIout() throws SQLException
{ return (Integer) _struct.getAttribute(2); }

public void setIout(Integer iout) throws SQLException
{ _struct.setAttribute(2, iout); }
```

```
/* Unable to generate method "f"
   because it uses a type that is not supported

  public <unsupported type> f (
    <unsupported type> i1)
  throws SQLException
  {
    Booleans __jPt_temp = this;
    <unsupported type> __jPt_result;
    #sql [getConnectionContext()] {
      BEGIN
      :OUT __jPt_result := :__jPt_temp.F(
      :i1);
      END;
    };
    return __jPt_result;
  } */

  public Boolean f (
    Boolean i1)
  throws SQLException
  {
    Booleans _temp = this;
    String _i1 = null;
    String _result = null;

    if (i1 != null) _i1 = i1.toString();

    #sql [getConnectionContext()] {
      DECLARE
      i1_ BOOLEAN;
      result_ BOOLEAN;
      t_ VARCHAR2(5);

      BEGIN
      i1_ := :_i1 = 'true';

      result_ := :_temp.F(i1_);

      IF result_ THEN
        t_ := 'true';
      ELSIF NOT result_ THEN
        t_ := 'false';
      ELSE
        t_ := NULL;
```

```
      END IF;
      :OUT _result := t_;

      END;
    };

    if (_result == null)
       return null;
    else
       return new Boolean(_result.equals("true"));
  }


/* Unable to generate method "p"
   because it uses a type that is not supported

  public Booleans p (
    <unsupported type> i1,
    <unsupported type> i2[],
    <unsupported type> i3[])
  throws SQLException
  {
    Booleans __jPt_temp = this;
    #sql [getConnectionContext()] {
      BEGIN
      :INOUT __jPt_temp.P(
      :i1,
      :INOUT (i2[0]),
      :OUT (i3[0]));
      END;
    };
    return __jPt_temp;
  } */

  public Booleans p (
    Boolean i1,
    Boolean i2[],
    Boolean i3[])
  throws SQLException
  {
    String _i1 = (i1 == null) ? null
                              : i1.toString();

    String _i2 = (i2[0] == null) ? null
                                 : i2[0].toString();

    String _i3 = (i3[0] == null) ? null
```

```
                                : i3[0].toString();

    Booleans _temp = this;

    #sql [getConnectionContext()] {
      DECLARE
      i1_ BOOLEAN;
      i2_ BOOLEAN;
      i3_ BOOLEAN;
      t_ VARCHAR2(5);

      BEGIN
      i1_ := :_i1 = 'true';
      i2_ := :_i2 = 'true';

      :INOUT _temp.P( i1_, i2_, i3_);

      IF i2_ THEN
        t_ := 'true';
      ELSIF NOT i2_ THEN
        t_ := 'false';
      ELSE
        t_ := NULL;
      END IF;
      :OUT _i2 := t_;

      IF i3_ THEN
        t_ := 'true';
      ELSIF NOT i3_ THEN
        t_ := 'false';
      ELSE
        t_ := NULL;
      END IF;
      :OUT _i3 := t_;

      END;
    };

    i2[0] = (_i2 == null) ? null
                          : new Boolean(_i2.equals("true"));
    i3[0] = (_i3 == null) ? null
                          : new Boolean(_i3.equals("true"));
    return _temp;
  }
}
```

注意 : SQLJ パラメータの意味により、各出力パラメータへの割当てはブロック内で 1 回のみ実行する必要があります。

索引

お

か

き

く

け

こ

ふ

へ

ま

め

ゆ

ら