Learning Apex Programming

Starting with the syntax and common use cases for developing in Salesforce, this book will teach you how to create and deploy applications on the fly. You will learn how to approach new projects and learn the basics of Apex, including how to use Eclipse and the sandbox.

Salesforce limits and batch processes are explained in an easy-to-understand format, and you will gain a deep understanding of triggers, classes, test coverage, and Visualforce pages. You will also discover how you can share your Force.com data and code with the outside world and how to integrate with Google calendar using Apex.

Finally, the book covers the creation of real-world web service integrations with Salesforce.com; you will create a Force.com application from start to finish using Apex as the business logic. By the end of this book, you will know all the best practices for development using Apex.

Who this book is written for

If you are a developer who has some object-oriented programming experience, Learning Apex Programming is the perfect book for you. This book is most appropriate for developers who wish to gain an understanding of the Force.com platform and how to use Apex to create business applications.

What you will learn from this book

- Create an Eclipse workspace and a sandbox, and learn about IDE best practices
- Write code within the limits of the platform and discover the best practices to stay out of trouble with queries
- Understand transactional and batch processes
- Discover classes and triggers and the best practices for using both
- Design a Visualforce page using Apex and JavaScript
- Customize sites to display Visualforce pages to the world
- Integrate Google and Salesforce calendars

Foreword by Mike Rosenbaum, Executive Vice President, Salesforce Platform
In this package, you will find:

- The authors biography
- A preview chapter from the book, Chapter 8 “Creating a Property Management Application”
- A synopsis of the book’s content
- More information on Learning Apex Programming

About the Authors

Matt Kaufman is no stranger to the Salesforce1 Platform, as he is one of the early employees at salesforce.com. Since 2002, Matt Kaufman has worked with hundreds of businesses to improve their efficiency through Salesforce. He is a certified Salesforce Advanced Administrator, Sales Cloud Consultant, Service Cloud Consultant, Advanced Developer, and Advanced Developer Instructor. His extensive experience and knowledge of salesforce.com technologies cause him to regularly be referred to as Mr. Salesforce, Salesforce wizard, Salesforce genie, and other magically endowed names.

Matt is currently the chief technology officer of MK Partners, Inc. (www.mkpartners.com), the leading salesforce.com implementation partner in southern California. He regularly provides training and talks on cloud technologies and development. He has also written other publications including Salesforce.com for Dummies and Salesforce.com's Service Cloud for Dummies, Wiley Publishing.

I would like to thank my wife and three children for putting up with me and embracing my passion for technology. I also give special thanks to the dedicated team of experts at MK Partners, who worked hard so that I could get the time to write. Finally, I cannot thank Justin Davis enough, who is the ultimate efficiency expert—without him, this book, and so much more, would not have happened.
Michael Wicherski, for the past several years, has been applying his business sense and knowledge towards designing, developing, and implementing custom solutions for hundreds of Salesforce customers, and has worked with clients of varying sizes—from mom-and-pop shops that are just opening to Fortune 100 companies.

During his time at MK Partners, Michael honed his skills for translating business processes into business logic. He is currently the senior Salesforce developer at The Agency RE, a real-estate firm in the heart of Beverly Hills, where he oversees the day-to-day use of Salesforce; plans, designs, and implements new functionality enhancements; and optimizes those currently in place.

An avid developer and a nerd at heart, Michael is constantly searching for those bleeding-edge technology innovations that can squeeze that extra bit of efficiency into his work.

Michael has also collaborated with peers on other publications related to Salesforce, most notably as the technical editor of Salesforce.com for Dummies, 5th Edition.

I would like to thank all those friends and family who have supported me in my endeavors and made sure I never settled and always strived for something more. I would also like to thank Matt Kaufman, Justin Davis, Chris Rodriguez, and the rest of the MK Partners team for their invaluable teamwork over the years, which allowed me to gain and apply the knowledge necessary for writing this book. To all who know me, thank you for embracing my nerd side, without which I would never have written a text about coding.
Learning Apex Programming

Welcome aboard! We're about to take a flight together into the realm of Salesforce1 development. We hope that our engaging banter will keep you on the edge of your seat through our thrilling hands-on exploration of the platform, its capabilities, as and its limitations. Our hope is that upon completing this book, you will walk away with the necessary knowledge to build the best application on App Exchange to date!

We ask that you please remain seated until the back cover has fully closed, lock your chairs in the upright position, and prepare yourselves to code like never before! The following sections will now provide you with an overview of our course and brief you on the features of this paper-bound vessel.

We know you have many choices when choosing a learning partner, and we thank you for the chance to become yours.

What This Book Covers

Chapter 1, Apex Assumptions and Comparisons, serves to introduce you to the Salesforce1 Platform. The Apex programming language, its similarity to Java, and other core concepts will be covered, as well as how to set up a development environment, which will be used throughout the book.

Chapter 2, Apex Limits, covers the limitations of the Salesforce1 Platform, specifically runtime limitations due to the shared tenant architecture. It also covers optimization methods to ensure that these limits are avoided.

Chapter 3, More and Later, which is an extension to the previous chapter, reviews how to process data asynchronously in bulk—leveraging the increased limits in asynchronous processing.

Chapter 4, Triggers and Classes, explains how to create Apex triggers and their respective Apex classes for database manipulation logic.

Chapter 5, Visualforce Development with Apex, allows you to explore how the Apex programming language can interact with the Visualforce markup language to create user interfaces on the Salesforce1 Platform. The chapter progresses from simple examples to more complex, dynamic interactions.
Chapter 6, *Exposing Force.com to the World*, teaches you how to create public-facing web services and expose your data to the whole wide world through Apex web services hosted on the Salesforce1 Platform.

Chapter 7, *Use Case – Integration with Google Calendar*, expands upon the knowledge gained in the previous chapter to consume the Google Calendar web service through Apex in order to create a synchronization process between the Salesforce and Google calendars.

Chapter 8, *Creating a Property Management Application*, covers how to create a property management application from start to finish, complete with credit card processing through the Authorize.net payment gateway.

Chapter 9, *Test Coverage*, completes your introduction to Apex programming with a fundamental understanding of Test Coverage, its requirements and limitations, and how to write proper tests.
Creating a Property Management Application

At this point, we've seen many different aspects of development on the Salesforce1 Platform and should be fairly familiar with both declarative and programmatic solutions to our everyday problems, as well as when to use which solution. In this chapter, we're going to mesh many of the things we have previously learned into a single application on the Salesforce1 Platform after doing a quick analysis run-through.

A big selling point for salesforce.com and the Salesforce1 Platform in general is how transparent the data is and how easy it is to run complex reports using the report builder. Although the process of building a report is outside the scope of this book, it is an important consideration when designing a data model to know your reporting capabilities. You don't want to spend time and effort into designing a perfectly running application that you can't run any meaningful reports on.

So here's the agenda—we're going to discuss what we want to keep track of, who is involved in the process, and what pieces of data are important for us to know about. On top of this, we're going to toss on some functionality to allow for very basic credit card processing (nothing is stopping you from making it more complex and functional!) and present the information in an easy-to-read manner.

Let's get started with using the trusty model of who, what, where, when, why, and how (not necessarily in that order). Note that this will be an overly simplistic model, one that does not account for every little nuance that exists with regard to property management. It is intended to demonstrate the functionality of the Salesforce1 Platform.
Why
Why, exactly, are we trying to do something? That should be the first question you always ask yourself. My typical approach is to gather a high-level overview first; this way, you never lose sight of the big picture. So, at the high-level overview, we want to track rental units, who's renting them, and whether or not they have paid up. Then, you should get into slightly more specific details. On a more in-depth look, we'd also like to know who actually owns the unit, whether or not it's part of a complex system, what additional fees (if any) are accumulated, possibly take a look at vendors who render services to the complex, as well as payment information. Finally, the actual data elements (fields) that we need to collect in order for our app to be as useful as possible and keep the accountants off our backs by offering a high return on investment on licensing/development work. We'll cover fields a little later. For now, let's take a look at the key things and people/entities involved.

What
We're interested in our rental units, right? We also want to be able to aggregate information across the rental units if they are part of a complex to determine the complex's overall performance. Who owns the units can also be a good aggregate piece of information, and let's not forget the actual fees/deposit/rent to be charged as well as their payment status and associated payments.

Who
So we know what we are interested in. Who we are interested in are the owners of the properties, those renting them, any vendors who render services to the property, and any pertinent contact information for each as well as any relationships that might exist related to these.

How
We'll use Accounts and Contacts to handle our who including owners, renters (households), as well as vendors. Then, we'll use custom objects to handle our what as well as a custom junction object to link our Contacts to rental units.

Although not a topic for discussion, you can set workflows to create tasks as reminders to our staff to calculate monthly dues for our renters, and e-mail reminders to the renters if their bill is due in one day as well as when it becomes past due.
For the payment processing aspect, we will use a very basic, charge-only, Authorize.net integration. We will also allow payments to be saved for cash and check payments via manual entry. More workflows here can send out e-mail notifications to the renter that their bill is now available as well as receipts when it has been paid. For our code in the mix, we’ll send the receipts for Authorize.net transactions from Apex.

We’ll also show you how to use Apex to do roll-up summaries once you pass the limit (there is a limit on the number of roll-up summaries you can have per object), or want to roll up to the parent of the parent, or there are other limiting factors that prevent you from using the inherent ability of master-detail relationships, which we’ll cover. These are all commonplace scenarios in data-intensive organizations.

**When and Where**

We couldn't just forget about these! Whenever you want and from within Salesforce!

**A custom model**

No colorful step-by-step instructions here, or pop ups sadly, but we do have pages and pages of text and cryptic code for you! All of it interlaced with the best comments you can have to guide you through to success.

Let’s break down all of the objects and data elements (fields) that we need to make this work.

![💡 Review previous chapters for step-by-step instructions on how to create custom fields and settings.](image)

**Account (standard object)**

These record types will allow us to track Accounts used for different purposes:

- Household (to contain owners/renters)
- Vendor (for all types of vendors)

**Contact (standard object)**

We want to create these custom fields to track information about each person:

- Total number of properties owned (roll-up summary: `COUNT` properties)
- Total number of units owned (roll-up summary: `SUM` property: Total Units)
Creating a Property Management Application

One thing we have not yet covered is how to create a custom object. It's a fairly straightforward process that should be familiar to you now that you have made other custom data elements on the Salesforce1 Platform. To create a new custom object, perform the following steps:

1. Click on the **Setup** link in the top-right corner of the Force.com GUI.
2. Under the **Create** section, click on **Objects**.
3. Click on **New Custom Object**.
4. Enter in the name (label).
5. Enter in the plural label; usually this just adds an "s" and is used when referencing multiple records of this object.
6. **Object Name** (the API name) is autogenerated from the name.
7. For **Record Name**, determine whether an autonumber or free text is more appropriate.
8. Select **Allow Reports**, **Activities**, and **Track Field History** as appropriate.
9. Check **Add Notes and Attachments related list** as appropriate (you can always add this later if you decide you want notes/attachments on your object).
10. Check **Launch new Custom Tab Wizard after saving this custom object** to make a tab for your new object. Note that not all objects require a tab.

Remember that in order to create roll-up summary fields, you will first have to create both the master and detail (parent/child) objects and required fields, and then return to the master object to create the roll-up summary field.

**Property (custom object)**

The Property object will house the physical location of our property assets, be it an apartment building, a housing complex, or a single unit (in which case, the property will have one and only one rental unit).

When you create an object, you’ll be presented with some options; go ahead and configure the Property object as follows:

- **Allow Reports**
- **Allow Activities**
- **Create New Tab**
An object without fields is like a car without wheels; so let's set up some fields:

- **Owner** (*Master-Detail: Contact*)
- **Physical Street** (*Text Area* length 255)
- **Physical City** (*Text* length 255)
- **Physical State** (*Text* length 2)
- **Physical Postal Code** (*Text* length 10)
- **Laundry Vendor** (*Lookup Account*)
- **Vending Machines Vendor** (*Lookup Account*)
- **Phone Vendor** (*Lookup Account*)
- **TV Vendor** (*Lookup Account*)
- **Internet Vendor** (*Lookup Account*)
- **Security Vendor** (*Lookup Account*)
- **Total Units** (*Roll-up Summary* `COUNT (rental unit)*)
- **Year Built** (*Text* length 4)
- **Community Features** (*Multiselect picklist*)
  - Pool
  - Tennis Court
  - Gym
  - Track
  - Basketball Court
  - Community Area
  - Play Area
- **Gated** (*Checkbox*)

Each type of Property has different attributes; so we're going to set up these record types:

- Apartment Building
- Housing Complex
- Condominium
- House
- Duplex
- Other Multi-tenant Housing
Rental Unit

Rental units will be the actual units for rent. This object will define all of the attributes such as bathrooms, bedrooms, square footage, as well as other information typically collected when renting/owning property.

The Rental Units object should be set up with these options:

- Allow Reports
- Allow Activities

We have to know more about each unit; so set up these custom fields:

- Unit # (Text, length 255)
- Number of Bedrooms (Number 2, 0 decimals)
- Number of Bathrooms (Number 2, 2 decimals)
- Floors (Number 2, 0 decimals)
- Garage (Picklist):
  - Attached
  - Detached
  - Assigned Parking
- Number of Parking Spaces (Number 2, 0 decimals)
- Laundry (Picklist)
  - In-unit hook-ups
  - Community
- Appliances Included (Checkbox)
- Furniture Included (Checkbox)
- Total Square Feet (Number 15, 3 decimals)
- Air Conditioning (Picklist)
  - Window Unit
  - Ceiling Fan
  - Central
- Heating (Picklist)
  - Forced Air
  - Central
• Pets Allowed (Checkbox)
• Utilities Included (Multiselect Picklist)
  ° Cold Water
  ° Hot Water
  ° Gas
  ° Electric
  ° Sewage
  ° Trash

**Rental Agreement (custom junction object)**

The Rental Agreement will store specific information for the rental term, including the agreed upon rental amount, deposits, and balances for the unique relationship of renter to rental unit.

We hope you're seeing a pattern; we like being able to take advantage of the built-in features such as reporting, so set up these options on this object:

• Allow Reports
• Allow Activities
• Track Field History
• Show Notes and Attachments Related List

This is no standard junction object. It serves a very important purpose, so we need to set up these key fields on it:

• Rental Unit (Master-Detail: Rental Unit)
• Renter (Primary) (Master-Detail: Contact)
• Pet Deposit (Currency 16, 2 decimals)
• Renter Has Pets (Checkbox)
• Deposit (Currency 16, 2 decimals)
• Rental Amount (Currency 16, 2 decimals)
• Total Deposit (Formula, Currency 16, 2 decimals)
  Pet_Deposit__c + Deposit__c
• Total Invoiced (Currency 16, 2 decimals). This will perform the Apex roll-up of Statement amounts from Statement here
• Total Payments (Currency: 16, 2 decimals). This will perform the Apex roll-up of Payments from Statement here
• Balance (Formula, Currency 16, 2 decimals)
  Total_Invoiced__c – Total_Payments__c

**Statement (custom object)**

Statements will house our invoices, or in other words, what we expect our renter to pay for this rental property, plus any additional fees incurred.

Here are the options to be set up for the Statement object:

• Allow Reports
• Allow Activities
• Track Field History

Statements are about money; so we need some fields to track it:

• Amount (Currency 16, 2 decimals)
• Additional Fees (Currency 16, 2 decimals, the default value is 0)
• Total Amount (Formula, Currency 16, 2)
  Amount__c + Additional_Fees__c
• Rental Agreement (Lookup, Rental Agreement)

You would think this would be a Master-Detail relationship. However, because Rental Agreement is a junction object (with two masters), you cannot set the Statement as a detail object; details of junctions are not allowed on the Salesforce1 Platform.

• Statement Date (Date, the default value is TODAY())
• Due Date (Date, the default value is TODAY()+15)
• Status (Picklist)
  ° Issued
  ° Paid
  ° Past Due
• Amount Paid (Roll-up Summary: SUM(Payment: Amount), where status= Paid
Payment (custom object)

We'll use the Payment object to store individual transaction information, the type of payment, as well as any relevant authorization codes, and so on. We frequently get into academic debates with peers on this type of object with regards to the payment method. Many argue that this should be a picklist option. However, others such as myself believe that record types fulfill this requirement much better, as they can control the page layout of the Payment record; this allows you to only show fields relevant to that payment method. For example, why would you care to see all the credit card fields if you paid by check or cash?

Of course, we want to be able to report on Payments, so set up these options:

- Allow Reports
- Allow Activities
- Track Field History

We're going to need some record types, as not all payments are the same:

- Credit Card
- Check
- Cash

Here are the custom fields you need to create on the Payment object:

- Amount *(Currency 16, 2 decimals)*
- Statement *(Master-Detail: Statement)*
- Status *(Picklist)*
  - Paid
  - Failed
- Payment Date *(Date/Time)*
- Authorize.net Transaction ID *(Text, length 255; External Id)*

By marking the transaction ID as an external ID, it will automatically be indexed for searches, allowing us to quickly find transactions based on this ID.
Creating a Property Management Application

- Authorize.net Authorization Code (Text length 255)
- Authorize.net Response (Text length 255)
- Billing Name (Text length 255)
- Billing Street (Text Area length 255)
- Billing City (Text length 255)
- Billing State (Text length 2)
- Billing Postal Code (Text length 10)
- Check Account Number (Encrypted Text length 128, mask 4)
- Check Routing Number (Encrypted Text length 128, mask 4)
- Credit Card Number (Encrypted Text length 16, mask credit card)
- Credit Card Expiration Month (Picklist)
  - January
  - …
  - December
- Credit Card Expiration Year (Text length 4)
- Credit Card Security Code (Encrypted Text length 4, mask all)

Even though we are marking some of these fields as encrypted, those without the View Encrypted Data permission can still edit these fields to update. The Apex-based functionality is also possible (the code sees the real value).

Let's also create a custom setting while we are in the mindset of clicking around—we will use it later in order to facilitate the Authorize.net functionality; let's make it a public list type setting.
The Authorize.net setting (custom setting)

Remember that custom settings are different from custom objects, but they too have some options:

- List
- Public

We like to use key-value pairs, so you just need to create one easy field:

- Value (Text length 255)

Now that we have our data model set up, let's throw some functionality in.

Paying your dues

We think we can skip the mafia bosses and objects to track broken knees, let's just stick to Authorize.net to process credit cards and the check or cash option via manual entry. Recall from Chapter 6, Exposing Force.com to the World, and Chapter 7, Use Case – Integration with Google Calendar, everything we said about interacting with external services, their terms, and so on—we're about to put it to good use.

Head on over to developer.authorize.net to sign up for a sandbox account, as shown:
Creating a Property Management Application

Once you have signed up for an account, you will get an e-mail from Authorize.net confirming your signup, and you can venture over to sandbox.authorize.net to log in. Once you have logged in, proceed with the following steps:

1. Navigate to the **Account** tab; it should be in the rightmost corner.
2. Click on **API Login ID and Transaction Key**; this is highlighted in the following screenshot:

   ![Authorize.net Settings](image)

3. Once you are on the next screen, you will have to generate a new transaction key. In order to do so, you need to know your secret answer, as shown in the following screenshot:
In fact, it's so secretive that you don't even know what it is! Don't believe me? Try answering, we bet you'll get it wrong. We never actually set this up, so how are we supposed to answer? Well, Authorize.net included it in that sign-up confirmation e-mail for you. It's not indicated in any way, so you'll just have to read through the e-mail for it, as shown here:

In fact, it's so secretive that you don't even know what it is! Don't believe me? Try answering, we bet you'll get it wrong. We never actually set this up, so how are we supposed to answer? Well, Authorize.net included it in that sign-up confirmation e-mail for you. It's not indicated in any way, so you'll just have to read through the e-mail for it, as shown here:

If you're trying to skim through and are having issues retrieving your transaction key because of your secret answer, read the last paragraph, it will save you loads of head-scratching time.
4. Make sure that you save your transaction key. If you lose it, you will have to generate another, and your old ones will become inactive. (Don't try using the combo shown in the text here, as it has been changed; you need to get your own.)

Okay, remember the Authorize.net setting we made earlier? Let's go fill it up:

1. Navigate to **Setup | Develop | Custom Settings**.
2. Click on **Manage** next to the Authorize.net setting.
3. Create an API login record (name) with your API login ID.
4. Create a transaction key record (name) with your transaction key.

Great! Finally, we are done with our point-and-click setup. Now we can look at some code!

The process to attain your live API login ID and transaction key are identical, except that you would log in at http://www.authorize.net/ instead of the sandbox.

The following slew of code is a complete class for the Authorize.net integration we will be using in this chapter. It has many moving parts, with custom classes (wrappers) declared at the bottom of the class, references to static class variables, and other elements that will be much easier to explain if you follow along with the comments:

```csharp
public class api_AuthorizeDotNet {
    //variables to hold our login credentials
    //these will get set later
    public static string APILOGIN;
    public static string APITRANSKEY;

    //method to process a credit card charge
    //takes in a parameter of authnetreq_wrapper, input
    //returns an authnetresp_wrapper
    //both are declared at the end of the class
    public static authnetresp_wrapper authdotnetCharge
```
(authnetreq_wrapper input) {
    // reusable method for setting Authorize.net credentials
    getAuthNetCreds();

    // Construct the request for a charge
    HttpRequest req = new HttpRequest();
    // for testing use the test endpoint
    // otherwise use
    // https://secure.authorize.net/gateway/transact.dll
    req.setEndpoint('https://test.authorize.net/gateway/transact.dll');
    req.setMethod('POST');

    // build message
    Map<string,string> messagestring = new map<String,String>();

    // Default Fields
    // See the authorize.net documentation
    // for more information
    messagestring.put('x_login', APILOGIN);
    messagestring.put('x_tran_key', APITRANSKEY);
    messagestring.put('x_version', '3.1');
    messagestring.put('x_delim_data', 'TRUE');
    // specify delimiter character for response
    messagestring.put('x_delim_char', ',');
    messagestring.put('x_relay_response', 'FALSE');
    // The type of transaction
    messagestring.put('x_type', 'AUTH_CAPTURE');
    // Processing credit card
    messagestring.put('x_method', 'CC');

    // Transaction Specific Information
    // card numbers, expiration, security codes
    messagestring.put('x_card_num', input.ccnum);
    // expiration date in the format mmYYYY
    messagestring.put('x_exp_date', input.ccexp);
    messagestring.put('x_card_code', input.ccsec);

    // transaction amount
    messagestring.put('x_amount', input.amt);
    // description of transaction
    messagestring.put('x_description', 'Your Transaction: '+input.ordername);

    // billing information
    messagestring.put('x_first_name', input.firstname);
Creating a Property Management Application

messagestring.put('x_last_name', input.lastname);
messagestring.put('x_address', input.billstreet);
messagestring.put('x_city', input.billcity);
messagestring.put('x_state', input.billstate);
messagestring.put('x_zip', input.billzip);

//encode the message components
String encodedmsg = ''; 
for (string s : messagestring.keySet()){
    string v = messagestring.get(s);
    //fix null values
    if (string.isblank(v)) v ='';
    encodedmsg += s+'='+
    EncodingUtil.urlEncode(v, 'UTF-8')+'&';
    //debug message value being added
    system.debug('TRACE: message bit '+s+' added');
}
//add message termination
encodedmsg += 'endofdata';
system.debug('TRACE: Encoded Message: 

'+ encodedmsg);

//set the body of the httprequest
req.setBody(encodedmsg);

//send and collect response
Http http = new Http();
string resp = http.send(req).getbody();
//debug response
system.debug(resp);
//split response by our delimiter
list<string> responses = resp.split(';');

//use parsing method to return response wrapper
authnetresp_wrapper parsedResponse =
parseIntoResponseWrapper(responses);

//debug response wrapper
system.debug(parsedResponse);
return parsedResponse;

public static void getAuthNetCreds(){
    //get api login setting value
    //test condition to generate test key
    //during test methods
Authorize_net_Setting__c apiloginsetting =
(test.isRunningTest())?
(new Authorize_net_Setting__c(name='API Login',
value__c = 'test')):
((Authorize_net_Setting__c.getInstance('API Login')));
//get transaction key setting value
//test condition to generate test key during test methods
Authorize_net_Setting__c apitranskeysetting =
(test.isRunningTest())?
(new Authorize_net_Setting__c(name='API TransKey', value__c =
'test')):
((Authorize_net_Setting__c.getInstance('API TransKey')));
APILOGIN = apiloginsetting.value__c;
APITRANSKEY = apitranskeysetting.value__c;
}

/**
 * Method to parse split response into wrapper
 * based on appropriate indeices
 **/ 
public static authNetResp_Wrapper
parseIntoResponseWrapper(list<string> input){
    authNetResp_Wrapper temp = new authNetResp_Wrapper();
    temp.responseCode = input[0];
    temp.ResponseSubcode = input[1];
    temp.ResponseReasonCode = input[2];
    temp.ResponseReasonText = input[3];
    temp.AuthorizationCode = input[4];
    temp.AVSResponse = input[5];
    temp.TransactionID = input[6];
    temp.InvoiceNumber = input[7];
    temp.Description = input[8];
    temp.Amount = input[9];
    temp.Method = input[10];
    temp.TransactionType = input[11];
    temp.CustomerID = input[12];
    temp.FirstName = input[13];
    temp.LastName = input[14];
    temp.Company = input[15];
    temp.Address = input[16];
    temp.City = input[17];
    temp.State = input[18];
    temp.ZIPCode = input[19];
    temp.Country = input[20];
    temp.Phone = input[21];
    temp.Fax = input[22];
temp.EmailAddress = input[23];
temp.ShipToFirstName = input[24];
temp.ShipToLastName = input[25];
temp.ShipToCompany = input[26];
temp.ShipToAddress = input[27];
temp.ShipToCity = input[28];
temp.ShipToState = input[29];
temp.ShipToZIPCode = input[30];
temp.ShipToCountry = input[31];
temp.Tax = input[32];
temp.Duty = input[33];
temp.Freight = input[34];
temp.TaxExempt = input[35];
temp.PurchaseOrderNumber = input[36];
temp.MD5Hash = input[37];
temp.CardCodeResponse = input[38];
temp.CardholderAuthenticationVerificationResponse = input[39];
temp.AccountNumber = input[40];
temp.CardType = input[41];
temp.SplitTenderID = input[42];
temp-RequestedAmount = input[43];
temp.BalanceOnCard = input[44];
return temp;

/**
 * The request wrapper. Holds all information
 * needed by Authorize.net to process a transaction
 * regardless of operation: charge, void, auth, refund
 **/
public string bankaccountname {get; set;}

public authnetreq_wrapper(){}
/**
 * Response wrapper. Holds the full Authorize.net response
 * in an easy to use object variable
 * Attributes are listed by return index rather than
 * alphabetically
 **/
public class authNetResp_Wrapper{
    // value, index in split string list
    public string responseCode {get;set;} //0
    public string ResponseSubcode{get;set;} //1
    public string ResponseReasonCode{get;set;} //2
    public string ResponseReasonText{get;set;} //3
    public string AuthorizationCode{get;set;} //4
    public string AVSResponse{get;set;} //5
    public string TransactionID{get;set;} //6
    public string InvoiceNumber{get;set;} //7
    public string Description{get;set;} //8
    public string Amount{get;set;} //9
    public string Method{get;set;} //10
    public string TransactionType{get;set;} //11
    public string CustomerID{get;set;} //12
    public string FirstName{get;set;} //13
    public string LastName{get;set;} //14
    public string Company{get;set;} //15
    public string Address{get;set;} //16
    public string City{get;set;} //17
    public string State{get;set;} //18
    public string ZIPCode{get;set;} //19
    public string Country{get;set;} //20
    public string Phone{get;set;} //21
    public string Fax{get;set;} //22
    public string EmailAddress{get;set;} //23
    public string ShipToFirstName{get;set;} //24
    public string ShipToLastName{get;set;} //25
    public string ShipToCompany{get;set;} //26
    public string ShipToAddress{get;set;} //27
    public string ShipToCity{get;set;} //28
    public string ShipToState{get;set;} //29
    public string ShipToZIPCode{get;set;} //30
    public string ShipToCountry{get;set;} //31
Creating a Property Management Application

```csharp
public string Tax{get;set;} //32
public string Duty{get;set;} //33
public string Freight{get;set;} //34
public string TaxExempt{get;set;} //35
public string PurchaseOrderNumber{get;set;} //36
public string MD5Hash{get;set;} //37
public string CardCodeResponse{get;set;} //38
public string CardholderAuthenticationVerificationResponse{get;set;} //39
public string AccountNumber{get;set;} //40
public string CardType{get;set;} //41
public string SplitTenderID{get;set;} //42
public string RequestedAmount{get;set;} //43
public string BalanceOnCard{get;set;} //44
public authnetresp_wrapper(){
}
}
```

As can be seen after reading through this class, the wrappers are much more involved than they have to be for the credit card charge method. We’ve included the exploded versions of these wrappers, not to confuse the reader, but to give those reading this book a leg up in their ambitions to develop the missing methods to address the other functions available through Authorize.net such as electronic checks, refunding, voiding, and simple authorizations.

Forgetting something, are we? Yes, indeed we are. We need to authorize the endpoints in order for us to be able to communicate to Authorize.net. Head on over to Setup | Security Controls | Remote Site Settings and add these:

- **Live**: https://secure.authorize.net/gateway/transact.dll
- **Test**: https://test.authorize.net/gateway/transact.dll

**Mine, all mine**

Now that we have a payment processing solution available, let's put it to good use by creating an interactive Visualforce page that will allow us to collect credit card information for processing. We'll make this page and its extension fairly dynamic, because we will make this one single page available internally as an embedded page within the Statement record view page and as a standalone page overriding the New button for payments; we will make it available as an external page that will allow users to pay without logging into Salesforce! We will be using concepts from Chapter 5, Visualforce Development with Apex, and Chapter 6, Exposing Force.com to the World, extensively, so refer to those if the need arises.
Black magic
First steps first. We’ll be using the standard controller for Statement__c, and we’ll need a custom extension called payments_Extension that contains a Payment__c record variable. Go ahead and create this extension now in Eclipse by right-clicking and navigating to src | New | Apex Class. Your extension class should look like this:

```java
public with sharing class payments_Extension {
    //Payment__c record variable to hold payment information
    public Payment__c thisPayment {get;set;}
    public payments_Extension(ApexPages.StandardController scon) {
        thisPayment = new Payment__c();
        // our handling logic will go here.
    }
}
```

We’ll also need our embedded Visualforce page; let’s call it statement_payment.
To create a new Visualforce page in Eclipse, just right-click on the src folder and navigate to New | Visualforce page and save it as this:

```html
<apex:page standardController="Statement__c"
    extensions="payments_Extension" sidebar="false"
    showHeader="false">
    <!-- rest of page content -->
</apex:page>
```

We now have a blank page, which we can embed on our Statement__c view page as well as expose it on a Force.com site for the guest user to see. The next step is to add all of the relevant payment fields and details about the statement and also ensure that these details are only displayed when a guest user is viewing the page (otherwise, the page is being viewed internally on the statement view page and we don't need statement details in this case). So in our extension, let’s determine whether the current user is a guest by adding the following method:

```java
public boolean getIsGuest(){
    //query for profile based on current user's (userinfo) profile
    //userType on the profile determines the license type
    //return true/false of comparison to 'Guest'
    return [
        select id, userType
        from Profile
        where id = :userInfo.getProfileId()
    ].userType == 'Guest';
}
```
Although you can use queries to retrieve information about users, there are more efficient ways that do not use up SOQL query calls. For example, the following line of code has the same effect:

```java
return UserInfo.getUserType() == 'Guest';
```

There are different ways to define a getter in Apex such as declaring a class variable and setting it in the constructor, but the previous example works well for what we are trying to accomplish and demonstrates a simple getter method.

Getters are regarded as transient and do not impact your view state, which becomes important once you have complex pages or pages with large amounts of data.

We can now simply use `{!isGuest}` on our Visualforce page to instantly know which type of user is accessing the page. Here is the complete embedded page:

```xml
<apex:page standardController="Statement__c" extensions="payments_Extension" sidebar="false" showHeader="false">
  <apex:sectionheader title="Statement" subtitle="{!Statement__c.name}" rendered="{!isGuest}"/>
  <apex:pageblock mode="maindetail" title="Statement Details" rendered="{!isGuest}"/>
  <apex:pageblocksection columns="1">
    <apex:outputfield value="{!Statement__c.statement_date__c}"/>
    <apex:outputfield value="{!Statement__c.balance__c}"/>
  </apex:pageblocksection>
</apex:pageblock>
<apex:form>
  <apex:pageblock title="{!IF(isGuest,'Payment Details','Quick Payment')}" mode="edit">
    <!--pageblockbuttons automatically arrange our buttons-->
    <apex:pageblockbuttons location="bottom"/>
    <!--we only want this button to show at the bottom of our page-->
    <apex:commandbutton value="Save Payment" action="{!savePayment}" disabled="{!success}"/>
  </apex:pageblockbuttons>
</apex:form>
```
<apex:pagemessages/>
<apex:pageblocksection id="paymentSection" columns="1">
  <apex:selectlist label="Payment Method" value="{!thisPayment.recordtypeid}" size="1" disabled="{!isGuest}">
    <!--getter method to retrieve our record types for payment__c--> 
    <!-- only payment method available for guests --> 
    <apex:selectOptions value="{!paymentRecordTypes}"/>
    <!--actionsupport allows us to call ajax rerenders--> 
    <!--or controller / extension methods using the--> 
    <!--action attribute on events--> 
    <apex:actionsupport event="onchange" rerender="paymentMethodDetails"/>
  </apex:selectlist>

  <!--amount defaulted to statement amount in extension--> 
  <!--editable if not guest--> 
  <apex:inputfield value="{!thisPayment.amount__c}" rendered="{!NOT(isGuest)}"/>
  <!--if guest, not editable--> 
  <apex:outputfield value="{!thisPayment.amount__c}" rendered="{!isGuest}"/>

  <!--generic billing information--> 
  <!--autopopulated in extension if not guest from renter--> 
  <apex:inputfield value="{!thisPayment.billing_name__c}"/>
  <apex:inputfield value="{!thisPayment.billing_street__c}"/>
  <apex:inputfield value="{!thisPayment.billing_city__c}"/>
  <apex:inputfield value="{!thisPayment.billing_state__c}"/>
  <apex:inputfield value="{!thisPayment.billing_postal_code__c}"/>
</apex:pageblocksection>

  <!--section to contain check/card details--> 
  <!--the id will allow us to ajax rerender this section--> 
  <apex:outputpanel id="paymentMethodDetails">
    <!--check fields--> 
    <apex:pageblocksection columns="1" rendered="{!thisPayment.recordtypeid != null && recordTypeMap[thisPayment.recordtypeid] == 'Check'}">
      <apex:inputfield value="{!thisPayment.check_account_number__c}"/>
      <apex:inputfield value="{!thisPayment.check_routing_number__c}"/>
    </apex:pageblocksection>
  </apex:outputpanel>
</apex:pageblocksection>

<!--card fields-->
Creating a Property Management Application

As mentioned, in order to get the page to save, you'll need the following extension. You can ask yourself why it is being presented in this weird out-of-order manner. The answer is simple: it's cleaner and easier to understand if everything is kept chunked like this rather than adding a couple of lines of code every few sentences and never providing you with the big picture. This extension will handle our error checking, success flagging, payment processing through the Authorize.net API and creation of `Payment__c` records from our embedded page, Force.com sites page, and soon our `payment_Edit` override page:

```java
public with sharing class payments_Extension {
    //Payment__c record variable to hold payment information
    public Payment__c thisPayment {get;set;}
    public Statement__c thisStatement {get;set;}
    public map<id, string> recordTypeMap {get;set;}
```
/boolean to determine if payment was successful
public boolean success {get;set;}

public payments_Extension(ApexPages.StandardController scon) {
    if (scon.getRecord() instanceof Statement__c)
        thisStatement = [select id, name, Rental_Agreement__r.renter__c, Rental_Agreement__r.renter__r.email, balance__c from Statement__c where id = :scon.getid()];
    success = false;
    recordTypeMap = new map<id,string>();
    //query for all payment record types
    //populate map
    for (RecordType r : [select id, name from RecordType where sobjecttype='Payment__c']){
        recordTypeMap.put(r.id,r.name);
    }
    //instantiate payment
    thisPayment = new Payment__c();
    if (scon.getRecord() instanceof Statement__c){
        thisPayment.Statement__c = scon.getid();
        thisPayment.Amount__c = thisStatement.balance__c;
    }
    //if guest user, preset type to credit card
    if (getIsGuest()){}
    for (id i : recordTypeMap.keyset()){
        if (recordTypeMap.get(i) == 'Credit Card'){
            thisPayment.recordtypeid = i;
            break;
        }
    }
    //if not guest, populate billing details from renter
    else if (scon.getRecord() instanceof Statement__c){
        Contact renter = [select id, firstname, lastname, mailingstreet, mailingcity, mailingstate, mailingpostalcode from Contact where id = :thisStatement.Rental_Agreement__r.renter__c];
        thisPayment.Billing_Name__c = renter.firstname+' '+renter.lastname;
        thisPayment.Billing_Street__c = renter.mailingstreet;
        thisPayment.Billing_City__c = renter.mailingcity;
        thisPayment.Billing_State__c = renter.mailingstate;
Creating a Property Management Application

```java
thisPayment.Billing_Postal_Code__c = renter.mailingpostalcode;
}
}

// this method will process and save our payment
// or report any errors in the attempt
public pagereference savePayment(){
    success = false;
    string paymentType = recordTypeMap.get(thisPayment.recordtypeid);
    // check if all fields filled out
    if (validateFields(paymentType)){
        // process credit card payments
        if (paymentType == 'Credit Card'){
            // create a request wrapper for authorize.net
            api_AuthorizeDotNet.authNetReq_Wrapper req = new api_AuthorizeDotNet.authNetReq_Wrapper();
            // set wrapper values
            req.amt = string.valueOf(thisPayment.Amount__c);
            req.firstname = (thisPayment.Billing_Name__c.contains(' '))?thisPayment.Billing_Name__c.substringbefore(' '):thisPayment.Billing_Name__c;
            req.lastname = (thisPayment.Billing_Name__c.contains(' '))?thisPayment.Billing_Name__c.substringafter(' '):thisPayment.Billing_Name__c;
            req.billstreet = thisPayment.Billing_Street__c;
            req.billcity = thisPayment.Billing_City__c;
            req.billstate = thisPayment.Billing_State__c;
            req.billzip = thisPayment.Billing_Postal_Code__c;
            // set wrapper credit card fields
            req.ccnum = thisPayment.Credit_Card_Number__c;
            req.cccexp = monthmap.get(thisPayment.Credit_Card_Expiration_Month__c)+thisPayment.Credit_Card_Expiration_Year__c;
            req.ccsec = thisPayment.Credit_Card_Security_Code__c;

            // give this request a name
            // querying here for the statement name to accommodate later //
            // functionality with payment_Edit
            req.ordername = 'Payment of '+ req.ordername = 'Payment of '+[select id,name from Statement__c where id = thisPayment.Statement__c].name;
```

[238]
// process authorize.net request
api_AuthorizeDotNet.authNetResp_Wrapper res = api_AuthorizeDotNet.authdotnetCharge(req);

// authorize.net data regarding the transaction
thisPayment.Authorize_net_Transaction_ID__c = res.TransactionID;
thisPayment.Authorize_net_Authorization_Code__c = res.AuthorizationCode;

// if the transaction failed
if (res.responseCode != '1' || res.ResponseReasonText != 'This transaction has been approved.'){
    thisPayment.Status__c = 'Failed';
    ApexPages.addMessage(new ApexPages.message(ApexPages.severity.error,'Payment Failed'));
    ApexPages.addMessage(new ApexPages.message(ApexPages.severity.error,'res.ResponseReasonText'));
    return null;
}

// successful transactions
thisPayment.Status__c = 'Paid';
thisPayment.Payment_Date__c = system.now();
upsert thisPayment;
success = true;
ApexPages.addMessage(new ApexPages.message(ApexPages.severity.confirm,'Payment Successful'));

try{
    // if there is an email on the contact
    // send them a confirmation email
    if (thisstatement !=null && thisstatement.Rental_Agreement__r.renter__r.email != null){
        // construct message
        Messaging.SingleEmailMessage msg = new Messaging.SingleEmailMessage();
        // to addresses is a list
Creating a Property Management Application

```java
msg.setToAddresses(new list<string>{thisStatement.Rental_Agreement__r.renter__r.email});
msg.setsubject('Payment Confirmation');
//you can set both html and plaintext bodies in case the recipient does not receive html
msg.setHTMLbody('Your payment of ' + thisPayment.Amount__c + ' has been successfully processed.<br/>Thank you.');
msg.setplaintextbody('Your payment of ' + thisPayment.Amount__c + ' has been successfully processed. Thank you.');
//send the email
Messaging.sendEmail(new list<Messaging.SingleEmailMessage>{msg});
}
}
```

```java
//verify required fields have been filled out
public boolean validateFields(string paymentType){
    boolean valid = true;
    //check common fields
    if (thisPayment.statement__c==null) valid = false;
    if (thisPayment.Amount__c==null) valid = false;
    if (string.isBlank(thisPayment.Billing_Name__c)) valid = false;
    if (string.isBlank(thisPayment.Billing_Street__c)) valid = false;
    if (string.isBlank(thisPayment.Billing_City__c)) valid = false;
    if (string.isBlank(thisPayment.Billing_State__c)) valid = false;
    if (string.isBlank(thisPayment.Billing_Postal_Code__c)) valid = false;
    return valid;
}
```
// check specific fields
if (paymentType == 'Check'){
    if (string.isBlank(thisPayment.Check_Account_Number__c))
        valid = false;
    if (string.isBlank(thisPayment.Check_Routing_Number__c))
        valid = false;
}

// credit card specific fields
else if (paymentType == 'Credit Card'){
    if (string.isBlank(thisPayment.Credit_Card_Number__c)) valid = false;
    if (string.isBlank(thisPayment.Credit_Card_Expiration_Month__c)) valid = false;
    if (string.isBlank(thisPayment.Credit_Card_Expiration_Year__c)) valid = false;
    if (string.isBlank(thisPayment.Credit_Card_Security_Code__c)) valid = false;
}

return valid;

// getter method for guest user determination
public boolean getIsGuest(){
    // match profile based on current user
    return UserInfo.getUserType() == 'Guest';
}

// return select options based on class map
public list<selectoption> getPaymentRecordTypes(){
    list<selectoption> temp = new list<selectoption>();
    // select option structure is value, label, disabled (optional)
    temp.add(new selectoption('', 'Select Payment Method'));
    for (id i : recordTypeMap.keySet()){
        temp.add(new selectOption(i, recordTypeMap.get(i)));
    }
    return temp;
}

// construct select options for this year
// and the next 4 years after that
public list<selectOption> getExpirationYears(){
    list<selectoption> temp = new list<selectoption>();
    for (integer i=0; i<5; i++){
        string y = ''+System.today().addYears(i).year();
        temp.add(new selectOption(y, y));
    }
    return temp;
}
Creating a Property Management Application

   return temp;
}

// map to convert months to 2 digits for authorize.net
public static map<string,string> monthMap = new map<string,string>{
   'January' => '01',
   'February' => '02',
   'March' => '03',
   'April' => '04',
   'May' => '05',
   'June' => '06',
   'July' => '07',
   'August' => '08',
   'September' => '09',
   'October' => '10',
   'November' => '11',
   'December' => '12'
};

In the previous extension, a confirmation e-mail is sent. However, in order to enable e-mails to be sent, the e-mail administration setting must be changed to allow this ([Setup] | [Email Administration] | [Deliverability] | [Access Level] | [Set to "All Email" | [Save]). You can also retrieve RecordType (and other schema-related) information using Schema Describes. In our extension, instead of querying for RecordTypes, you can also loop through the following to save a query:

   Payment__c.SObjectType.getDescribe().getRecordTypeInfos({
   )
Now that we have the extension in place and saved, go ahead and save the page as well. We can now test out our embedded page as well as the Force.com site page.

To test the embedded page, go to Setup | Create | Objects | Statement__c | Page Layouts | Edit Statement Layout. On the page layout editor, drag a new section down and set it to the following:

- Single column
- Width 100%
- Height 475 px
- Show scrollbars (in our case, the page is bigger than we anticipate)

After this, select the Visualforce Pages option from word bank on the top-left corner and drag the statement_payment page into the newly created section. Now, save the layout and you should see the following page:

Keep in mind that embedded pages are only visible on the view page. As such, when you save a new payment, the page will not automatically refresh the entire view page; thus, you will not see the payment in the related list until you refresh the entire page.
The next step is to take a look at our page on the Force.com site. Remember to set all of the relevant Visualforce, Apex class, object, and field permissions for the site guest user profile before attempting this or you will get an error. Here is what the page should look like. Note \( id = \) parameter; this is a `Statement__c` ID and required.

There are, of course, many steps that a developer can take at this point to personalize the experience for backend versus frontend users as explained in previous chapters, but it is easy to see how quickly a single page can be created for use in both worlds.
I did it my way!

The Salesforce1 Platform is a great out-of-the-box user experience; however, for something like our payment processing scenario, the standard create new page will not really work. We're now going to create, very quickly, a payment_Edit page that we will use to override the standard Payment__c new page with. Start by creating a new page called payment_Edit and copying in all of the markup from statement_payment, but don't save it quite yet. We need to change standardController="Statement__c" to:

```
standardController="Payment__c"
```

Next, we will need the ability to set the Statement__c field, so let's add inputfield for this before the payment method:

```
<apex:inputfield value="{!thisPayment.Statement__c}"/>
```

We also need to remove the initial page block titled Statement Details and change sectionheader to the following line of code:

```
<apex:sectionheader title="New Payment"/>
```

Also, changing the showheader and sidebar attributes on the page tag to true will make the page feel more like what users are used to seeing. Now, we can compile and save this page and since our extension is already written in a way that conditionally determines what to do based on the standard controller being used, we can override the new Payment__c button. Proceed with the following steps:

1. Navigate to Setup | Create | Objects | Payment__c.
2. Scroll down to Buttons, Links, and Actions.
3. Click on the Edit link next to New.
4. Select the Visualforce Page option with payment_Edit for the Override with option.
5. The Skip RecordType Selection Page should be checked.
Here is a visual; ensure that you check the **Skip RecordType Selection Page** option:

We can test the functionality out by going to any statement and making a new payment using the related list **New** button. You will get a screen similar to the following screenshot:
Amazing! How easy it is to reuse our functionality and code to quickly adapt to different scenarios, isn't it? Now, if you are one of those advanced developers reading this book for leisure, you'll probably be saying to yourself that there are more efficient ways of doing certain things that have been covered so far. While this might be true, minifying and optimizing the code, while making it cryptic, is not the goal of this entry-level book; that's something to be covered in an advanced text.

Ain't no mountain

Although Force.com has many different limits on things such as number of fields, number of specific types of fields, and those dictating how custom code should be written and can be used, there are ways to work around many of these limitations. Let's take a look at how to write a roll-up summary formula in a trigger, which we will take advantage of if we have reached the maximum allowed on our object, or want to cross several relationship bridges, or want to mimic the roll-up functionality through a lookup relationship rather than Master-Detail. In order to demonstrate this functionality, we will do two Apex roll-ups using aggregate queries to Rental Agreement; one will be based on Statement__c (Total Invoiced) and the other will be based on Payment__c (Total Payments). Create the class before the triggers and this time around rollup_Methods, as shown:

```apex
public class rollup_Methods {

  // for rolling up statements to total invoiced
  public static void rollupStatementsToRentalAgreement(list<Statement__c> newlist) {
    // first we need a set of Rental Agreements involved
    set:id> rentalAgreementSet = new set:id>();
    // loop through every statement and add the related rental agreement id
    for (Statement__c s : newlist)
      rentalAgreementSet.add(s.Rental_Agreement__c);

    // list to store our Rental_Agreement__c for update
    list<Rental_Agreement__c> rentalUpdates = new list<Rental_Agreement__c>();

    // aggregate query to sum total amount from statement
    // group by rental agreement
    for (AggregateResult ar : [select sum(Total_Amount__c)
      totalInvoice,rental_agreement__c from Statement__c where
      Rental_Agreement__c IN :rentalAgreementSet group by
      Rental_Agreement__c]) {
```

[247]
// instantiate an update record for each grouped aggregate result
Rental_Agreement__c r = new Rental_Agreement__c(id=
    string.valueOf(ar.get('rental_agreement__c')));
// set value of total invoiced
r.Total_Invoiced__c =
    double.valueOf(ar.get('totalInvoice'));
// add to update list
rentalUpdates.add(r);
}

// if update list has records, update
if (!rentalUpdates.isEmpty()) update rentalUpdates;

// for rolling up payments to total payments
// note that you can use a map or list from triggers
// use trigger.new and trigger.newmap respectively (same for 'old')
public static void rollupPaymentsToRentalAgreement(map<id,Payment__c> newmap){
    // since we need to navigate two objects to get to rental agreement fields
    // it is best practice to query for the data we need to ensure it is available to us
    // triggers do not automatically contain all fields if invoked from code actions
    set<id> rentalAgreementSet = new set<id>();
    for (Payment__c p : [select id,
        Statement__r.Rental_Agreement__c from Payment__c where id IN
        :newmap.keySet()]){[
        rentalAgreementSet.add(p.Statement__r.Rental_Agreement__c);
    }

    // list to store our Rental_Agreement__c for update
    list<Rental_Agreement__c> rentalUpdates = new
    list<Rental_Agreement__c>();

    // aggregate query to sum total amount from payments
    // group by rental agreement
    for (AggregateResult ar : [select sum(Amount__c) totalPaid,
        Statement__r.Rental_Agreement__c ra from Payment__c where
        Statement__r.Rental_Agreement__c IN :rentalAgreementSet group
        by Statement__r.Rental_Agreement__c]){[
        // instantiate an update record for each grouped aggregate result
        // aliased the rental agreement grouping as 'ra'
    ]}
Rental_Agreement__c r = new Rental_Agreement__c(id=
    string.valueOf(ar.get('ra')));
   //set value of total invoiced
    r.Total_Payments__c = double.valueOf(ar.get('totalPaid'));
   //add to update list
    rentalUpdates.add(r);

    //if update list has records, update
    if (!rentalUpdates.isEmpty()) update rentalUpdates;

The triggers for these are super easy. They should be upsert triggers (acting on both
insert as well as update):

    trigger payment_Trigger on Payment__c (after insert, after update) {
      if(trigger.isAfter && (trigger.isInsert || trigger.isUpdate))
        rollup_Methods.rollupPaymentsToRentalAgreement(trigger.newmap);
    }

The other trigger is as follows:

    trigger statement_Trigger on Statement__c (after insert, after update) {
      if(trigger.isAfter && trigger.isInsert || trigger.isUpdate)
        rollup_Methods.rollupStatementsToRentalAgreement(trigger.new);
    }

Easy, right? If anything about the triggers is confusing, refer to Chapter 4, Triggers and
Classes, for some more in-depth explanations and walkthroughs.

Upgrade to the app status
So far, everything we have created has been a page or class (or trigger),
interconnected, yes, but really still standalone. Keeping in line with the chapter's
title, we're now going to create an actual app, a collection of tabs, objects, pages, and
classes and have it contain our tabs for Property Management Application. Proceed
with the following steps:

1. Navigate to Setup | Create | Apps | New.
2. Select Custom App (if prompted to select between Console and this).
3. Name the app as App Property Management.
4. Upload a custom logo if you want.
5. Add the following tabs:
   - Accounts
   - Contacts
   - Properties
   - Rental Agreements
   - Reports

6. Make them visible on the next screen and finalize.

You should now be able to click on the blue bubble (app selector) in the top-right part of the screen and select the **Property Management** app, as shown in the following screenshot:

![Property Management App Selector](image)

If you want to add any additional tabs, you can; if they are missing, for example, Statements, you can create a tab and then edit the app to add it.

**Summary**

Hello, are you still there? Did we just blow your mind? Are you feeling confused? Elated? Questioning everything you’ve ever known? Don’t worry; this feeling goes away after a few years. It’s hard to believe that in a single chapter of this book, we have walked you through creating a fully functional application that can actually be used for business purposes. Before you run off and try to sell this application, there are some other things to keep in mind. First, you’re not the only reader of this book, and we don’t want to flood the market with property management applications. Second, in order to package or deploy this app, you’re going to need test coverage. So keep reading, because the next chapter is an important one!

In the next and final chapter, we’ll review the concept of test coverage. Test coverage is, at the most basic level, code written to run your code to detect errors. However, as professional developers, we tend to strive for functional tests that test your code by simulating actual use cases.
Where to buy this book

You can buy Learning Apex Programming from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.