Moving from Inform 7 to TADS 3

By Jim Aikin

Inform 7 is a very attractive authoring system for people who want to write interactive fiction but have little or no experience with computer programming. Inform's "natural language" programming syntax is easy to grasp if you're new to computer programming, and can be very intriguing even for expert programmers. Inform's beautiful cross-platform IDE (integrated development environment) and active user community are also enticements.

At some point, though, folks who have tried Inform 7 may find themselves wondering whether another development system might be worth investigating. Possibly they've become frustrated in their efforts to create a game with I7, or possibly they've written one or more complete games in I7 and are now considering whether a different feature set might be more to their liking, or more suitable for their next project. People's motivations may vary, and in this tutorial we'll have almost nothing to say about why some authors may have become disenchanted with Inform 7. [*]

Several other development systems are available. Quest and ADRIFT are very different from I7 except in one important respect: They're both designed to be approachable for authors who have little or no experience in computer programming. TADS 3 appears, at first glance, to be just the opposite. Aspiring authors, especially those who are new to programming, can be forgiven for having the impression that T3 is difficult to learn and use.

Make no mistake, T3 is complex. But it's not as difficult as you may think. Writing T3 games can even be fun, once you get the knack of the TADS way of working.

Why might you want to consider switching to TADS 3? Without attempting to diagnose or dwell upon any dissatisfactions that you may (or may not) have with Inform 7, here are some advantages that T3 offers:

- **Code that looks like code.** If you're happy with Inform's approach (code that looks like text), this point won't sway you. But in observing questions brought up by Inform users on the intfiction.org forum over the past couple of years, I've noticed again and again that authors try to use syntax that ought to be perfectly natural, but that doesn't work because the I7 compiler insists that a sentence be structured in one particular way (and not always in a way that's intuitively obvious). T3 code also has to be structured in particular ways, of course, but the syntax is more uniform and less open to ambiguity, so as you learn how to use it you'll encounter fewer "edge cases" that cause confusion. If you have some experience with conventional computer programming, TADS 3 may be easier to learn than Inform 7. Its syntax

* To clear up, at the outset, one possible misconception, I was indeed motivated to write this document partly by the fact that, at this writing, no new version of Inform 7 has been released for more than two years. But while I may have been moved by a disappointment or irritation that I speculate might be shared by other authors, you won't find any editorializing or finger-pointing in these pages. What follows is an entirely straightforward discussion of IF programming techniques.

is very similar in most respects to the syntax of C. But if you don't know C (or Javascript, another popular C-type language), that won't mean anything to you, so there's no reason to get into it here.

- **Only one programming language to learn.** As you delve deeper into I7, you may find that you also need to learn the syntax and conventions of Inform 6, which is entirely different. In T3, all of the library code is written in the same language you'll be using to write your game, so as you become a power user your work will get easier rather than more complicated.
- **T3 gives the author more control.** Ultimately, either system can produce a fully functional game, and you could probably build a complex game that would appear absolutely identical if programmed in both systems. T3 makes fewer assumptions up front about what you, the author, will want to do. This makes for more work when you're just starting out, because you have to specify details that Inform takes for granted, but ultimately the T3 approach makes it easier to do the unusual things you may want to do in your game.
- **Better handling of complex sensing scenarios.** On occasion, your player character may be in an open field and may be able to observe items that are visible but too far away to touch, or may be in an enclosure where she can hear things but not see them. The T3 library handles this type of limited sensing with some standard mechanisms. (There's even an Unthing class, which allows the player to get a sensible response when trying to refer to an object that's not present at all.)
- **A richer set of built-in object types.** I7's library is designed in a manner that is deliberately somewhat spartan, with only a few basic kinds. Many more kinds are available if you download and include extensions, but with T3 there's less need for extensions, because the library includes a wide variety of classes — useful things like Dispenser, Flashlight, Noise, Odor, and FloorlessRoom. Because these items are part of the standard library, they're fully tested by the T3 development team. When a new version of TADS is released, the library classes won't have to be reworked to remain compatible.
- **Tools for building complex NPCs.** Non-player characters are perhaps the most challenging part of IF programming. The T3 library's ActorState and AgendaItem classes, and its array of Ask/Tell/Give/ShowTopic classes, streamline the process of building NPCs who seem alive and responsive.
- **Large games compile faster.** Not a great deal faster, to be sure, but developing a game means writing bits of code and then compiling the game over and over and over, hundreds of times. As your game gets larger, you'll find that T3's separate compilation mechanism enables each compilation to run a bit faster than I7 can manage, so you'll spend less time twiddling your thumbs.
- **Better handling of text manipulations.** Inform's use of "indexed text" is one of its less graceful features. All single-quoted strings in T3 are a single data type, and can be manipulated at run-time using a healthy set of commands.
- **Large source files slow down the I7 IDE.** If you've ever typed a left square bracket to start a comment near the beginning of a long source file, you know what I'm talking about. This is not an issue in T3.
- **No danger of namespace collisions.** Inform will allow you to create an object called the red ball and another object called the big red ball. Thereafter, if you write code about the red ball,

the compiler will probably try to guess which object you're referring to, and it may guess wrong, resulting in hard-to-find bugs. This can't happen in T3, because each object you create must have a unique identifier. Namespace collisions become especially problematical in I7 when you're including a few extensions.

The purpose of this brief essay is not to explain the details of the TADS 3 programming language or delve into the myriad features of its library. Eric Eve has already written several brilliant and very detailed tutorials showing how T3 works, and once you start looking at T3, his tutorials (along with the rest of the T3 documentation) will quickly become essential reading.

My goal is much more modest. I'll assume you're already conversant with Inform 7 and are wondering whether you might want to switch to TADS 3. If that's the case, I hope to help you get a picture of what it will be like to transfer your authoring skills to the T3 system. Armed with this knowledge, you should be able to move from I7 authoring to T3 authoring, if you should decide to do so, with less confusion and less uncertainty.

Along the way, I hope to convince you that learning and using T3 is not nearly as difficult as you may have thought!

Frequently Asked Questions
(for very small values of "frequently")

Can I write TADS games on a Mac or Linux computer? Yes. The TADS Workbench is Windows-only, but a solid TADS compiler (FrobTads) is available for other operating systems. It's also possible to run Workbench with 99% of its normal functionality in a Windows emulator on MacOS or Linux.

What are the most important differences between T3 and I7? Both systems produce full-featured text games (which can include graphics and sound if desired as well as text), and the games themselves will be very similar. However, there are many differences, and a difference that is important to one author may be less important to another, so providing a list of the "most important" differences would be difficult. Briefly, TADS code is entirely different in appearance and syntax from Inform code. The T3 library has a richer set of built-in features than the I7 library, as noted above. Also as noted above, T3 makes fewer assumptions than I7 about how you, the author, are going to want to structure your game, and that means you'll sometimes need to do a little more programming work — work that will begin to pay dividends as your game becomes more complex.

I'm not sure yet that I want to go to the trouble of downloading and installing the development package. Can I browse the documentation online first, to see what I would be working with? Yes. Go to www.tads.org/t3doc/doc/index.htm.

That's a lot of documentation! Where can I find a brief overview of TADS 3? You're reading one right now. Another essay you should definitely read is Eric Eve's long and insightful comparison of T3 and I7 on Brass Lantern (www.brasslantern.org/writers/iftheory/tads3andi7.html). That article is now three years old, but is still almost entirely accurate with respect to the current versions.

Is there an easy way to get started learning T3 with fewer complexities to trip over? Yes. Eric Eve is currently (as of January 2013) working on a streamlined library called adv3Lite. (The download URL is may change, but your favorite search engine should be able to find it. As of January 2013 it's at http://sdrv.ms/UICBhp.) This library is intended as a simplified replacement for the standard adv3 library included with TADS 3.

I like adding variation to my printed texts with the "[one of] … [or] … [at random]" tags. Can T3 do that? Yes. The in-text brackets are << and >> rather than [and], but the functionality is very similar.

I like using Inform's Scenes to move my story forward. Can I write scenes in T3? Yes. To do this, you need to download an extension (scenes.t, by Eric Eve) from the IF Archive (http://www.ifarchive.org/indexes/if-archiveXprogrammingXtads3XlibraryXcontributions.html).

I like defining Regions in I7 to control what happens in different parts of the map. Can I define regions in T3? The T3 library doesn't provide any direct support for defining regions. You can define your own region sensing mechanism by hand, or you can take advantage of the new adv3Lite library. This library both streamlines TADS 3 and adds support for a few handy I7-type features, including regions, scenes, and a type of action handling that more closely resembles I7's Instead rules.

Does the T3 development environment have anything comparable to I7's World Map or Index? Sadly, no.

Can I write a game in T3 that people can play in their Web browser without downloading the game file and an interpreter? Yes. Unlike Inform games, T3 games must be compiled specifically for Web-based play in order for that to work, but compiling your game for both targets (interpreter and browser) is not difficult.

(If you have any comments on this essay, or if you'd like to suggest that it be revised to include more information on topics that you're curious about, please feel free to send me an email at midiguru23@sbcblobal.net.)

Setting Up

Your experience of T3 will be a bit different depending on whether you use Windows or some other operating system. The T3 IDE, called Workbench, is a Windows program, and there is no equivalent in MacOS or Linux. If you have a Windows emulator (such as Boot Camp, Crossover, or WINE) in your non-Windows machine, you should be able to download, install, and use Workbench.

At most, you'll see an occasional minor glitch when using Workbench in a Windows emulator. The main issues I've noticed are that some options settings are not saved between work sessions, and the links that appear when you search the documentation don't work properly.

The Workbench download (available at http://www.tads.org/tads3.htm) contains everything you need, including the documentation. There's really very little you need to do to get started, other than run the installer and then launch the program. You can launch Workbench with a tutorial game, which is a useful way to take a look at T3 programming.

If you're not set up to run Windows programs, you can still write T3 games very easily. You'll miss out on a few convenience features, and when starting a new project you'll have to write a bit of extra code, but the T3 game(s) you develop in MacOS or Linux will be absolutely indistinguishable from games written in Workbench, and your workflow as you develop the game will be similar.

Setting Up on the Mac

Basic instructions on setting up your Macintosh so you can write T3 games are available in the *Quick Start Guide*, which is included in the T3 documentation. But unless you know where to look for the instructions, they won't do you any good. So let's go through the process in detail.

A few clicks on the TADS website (www.tads.org) will take you to the download page for the authoring system, which is called FrobTads. Starting on the home page, click the big blue download button, then choose your operating system, then choose "I'd like to write my own game." On this page, download FrobTads.

You'll also want the QTads interpreter (http://qtads.sourceforge.net/). T3 games compiled with FrobTads can be played directly in the Mac's Terminal, but the Terminal will ignore any graphics or audio you've included in your game — it's strictly a text interface, and not a very nice one at that. QTads is a full-featured interpreter, so you'll want to download it and install it in your Applications directory.

The third element you'll need is a text editor for writing your code. You can use TextEdit on the Mac, but it's not really very functional. I recommend TextWrangler (www.barebones.com/products/textwrangler/download.html). It's free. Just as important, it provides a multi-tabbed interface in which you can keep numerous source code files open at the same time.

Here's the process you'll need to go through to get started on the Mac, as outlined in the *Quick Start Guide:*

After downloading FrobTADS, double-click on the .dmg file and run the installer.

Create a directory to hold your projects, and a subdirectory within it to hold your first project. For example, in Documents, create TADS. In TADS, create a MyFirstGame folder. (You'll probably want to name the folder after the game you're intending to create.)

Within the folder for your first project, create a folder called obj. This will hold the object files created by the compiler while it's running. You won't need to be concerned about anything in this folder; it will take care of itself.

Using a text editor (not a word processor), create a .t3m file. For convenience, give the .t3m file the same name as the project, perhaps MyFirstGame.t3m. Copy the following text into your new .t3m file and save the file to the project folder:

```
-D LANGUAGE=en_us
-D MESSAGESTYLE=neu
-Fy obj -Fo obj
-o MyFirstGame.t3
-lib system
-lib adv3/adv3
-source MyFirstGame
```

Replace "MyFirstGame" in the code above with the name of your actual game, if it's different. (Note: As you add more source code files to your project, all you need to do is add more lines beginning with "-source" to the end of the .t3m file. The source code files will have the .t extension on their name, but don't use .t in this list, just the main part of the filename.)

Open a Terminal window. The Terminal program is located in Applications > Utilities. (You may want to make an alias for it and drag it into your Dock, as you'll be using it often.)

Create a starter game file, again as a text file, and save it to the MyFirstGame directory. Your starter game should look more or less like this:

```
#include <adv3.h>
#include <en_us.h>

gameMain: GameMainDef
    initialPlayerChar = me
;

versionInfo: GameID
    name = 'My First Game'
    byline = 'by Bob Author'
    authorEmail = 'Bob Author <bob@myisp.com>'
    desc = 'This is an example of how to start a new game project. '
    version = '1'
    IFID = 'b8563851-6257-77c3-04ee-278ceaeb48ac'
;
```

```
firstRoom: Room 'Starting Room'
    "This is the boring starting room."
;

+ me: Actor
;
```

Fill in those quoted parts under the line reading “versionInfo:GameID” with your own information. Everything in the versionInfo should be self-explanatory, except that last line that starts “IFID =”. That long, random-looking string of letters and numbers is exactly what it appears to be — a long, random string of letters and numbers. Well, almost: it’s actually composed of random hexadecimal (base-16) digits, that is, 0 to 9 plus A to F. The purpose of this random number is to serve as a unique identifier for your game when you upload it to the IF Archive. The format is important, but the individual digits should simply be chosen randomly. For your convenience, tads.org provides an on-line IFID generator at http://www.tads.org/ifidgen/ifidgen.

Launch the Terminal and use the cd (change directory) command to navigate to the folder where your game files are stored. For instance, you might type:

```
cd Documents/TADS/MyFirstGame
```

While the Terminal is logged into this directory, you can compile your game using this command:

```
t3make -d -f MyFirstGame
```

If all goes well, you should see a string of messages in the Terminal window, and a new file (MyFirstGame.t3) will appear in the MyFirstGame directory. This is your compiled game file. If you’ve installed an interpreter program such as QTads that can run TADS games, you’ll be able to double-click the .t3 file and launch the game to test your work. Alternatively, you can run the game directly in the Terminal by typing:

```
frob MyFirstGame.t3
```

As noted earlier, the Terminal provides only a primitive text-only interface for your game. QTads is the way to go.

Another reason not to run your work-in-progress in the Terminal is so you won’t have to retype the t3make command over and over. Each time you need to recompile your code, bring the Terminal window to the front and hit the Up arrow on your keyboard. This will reload the most recent command line (the one containing the t3make command). Hit Return, and the command will be run again, producing a fresh version of the .t3 file.

Code Concepts

Without getting too deeply into the details of T3 programming, let's take a quick look at the most important differences Inform authors will need to know about.

Objects

TADS 3 is an object-oriented language. Let's not get too technical about what that means. The practical result for the Inform author is this: In I7, you write rules for how you want your game to operate. The rules are written as free-standing sentences, and can generally be put anywhere in your source text that you find convenient. In T3, you can write much the same rule (though the syntax would be entirely different), but you will always put the rule within the definition of the object that it's intended to operate on.

Let's look at a fairly basic example — again, without going into all the details of how T3's code works, but simply to illustrate the difference in how you would create a rule. (They're not called rules in T3, but let's not worry about that either.) We're going to give the player an umbrella, which shouldn't be opened in the living room. For convenience, we'll make the umbrella an openable container, so that it can be opened and closed, but since it isn't a very practical container we won't let the player put anything in it. For testing, we'll give the player a coin to try to put into the umbrella, and a second room where the player will be allowed open the umbrella.

Here is how you would do that in Inform 7:

```
The Living Room is a room. The Porch is north of the Living Room.

The player carries an umbrella and a coin. The umbrella is a closed openable
container.

Instead of opening the umbrella when the player is in the living room, say "You
shouldn't open the umbrella in the house."

Instead of inserting something into the umbrella, say "The umbrella is not much use
for carrying things."
```

That's a complete Inform game. Below is the TADS 3 version, from which the versionInfo and the #include statements that form the file header (see the previous example) have been omitted. This code will produce exactly the same results as the Inform version when you run the game.

The TADS code is longer, partly because Inform makes some default assumptions about your game that TADS doesn't make. For instance, in Inform you don't have to create the player character or put him in a specific location, because Inform assumes it knows what you want. It will always start the player character in the first room you mention in your code. But in TADS, you need to specify where the player character is at the beginning of the game.

The fact that TADS makes fewer assumptions about what you will want is one of its strengths as an authoring system, but a little more effort will often be needed to get things set up the way you want them. When your game design is simple, TADS will expect you to do more work, but as your game design becomes more complex, you'll probably find that TADS code is clearer and easier to work with than the equivalent Inform code would be.

Here's the TADS version of the Inform example shown above:

```
gameMain: GameMainDef
    initialPlayerChar = me
;

livingRoom: Room 'Living Room'
    north = porch
;

+ me: Actor
;

++ umbrella: OpenableContainer 'umbrella' 'umbrella'
    initiallyOpen = nil
    dobjFor(Open) {
        verify () {
            if (me.isIn(livingRoom)) illogicalNow ('You shouldn\'t open the
umbrella in the house. ');
            else inherited;
        }
    }
    iobjFor(PutIn) {
        verify () {
            illogical (The umbrella is not much use for carrying things. ');
        }
    }
;

++ coin: Thing 'coin' 'coin'
;

porch: Room 'The Porch'
    south = livingRoom
;
```

If you're new to TADS, you can be forgiven for looking at that code and saying, "Ack! How am I ever going to deal with all that?" So let's highlight the main features of this code and explain how they differ from the Inform version.

1) The first two lines tell TADS that the player character will be an object called *me.* (Inform creates the player character automatically, and calls it *yourself*.)
2) Each of the code sections above defines an object, and each object definition ends with a semicolon. By convention, this trailing semicolon goes on a line by itself. This makes the code

easier to read.

3) Although T3 code is usually indented, this is strictly to make it easier for the author to read. The indentations have no meaning to the T3 compiler. Instead, blocks of T3 code are surrounded by curly braces (or, in some cases, parentheses or square brackets).
4) In both Inform and TADS, we have to tell the program that the living room is a room. Inform will then infer that the porch must be a room, because only rooms are connected by relations like north and south. In TADS, we have to mention the class of every object (in ordinary language, the "class" is the type of object we're creating), so we have to say that the porch is also a Room.
5) TADS doesn't assume that just because the porch is north of the living room, the living room must be south of the porch. We have to list each exit from a room explicitly. This is not a bad thing, as you'll discover when you start creating world maps in which the connections between rooms twist and turn. But for this simple case, it does require an extra line of code.
6) The '+' and '++' before the me, umbrella, and coin objects are a quick shorthand for telling TADS where those objects are at the beginning of the game. The me object is directly below the livingRoom object, so a single '+' tells TADS that me is in livingRoom. A dual '++' creates a second level of containment: The umbrella and coin start the game located within me — that is, within the player's inventory.
7) Inform lets you create in-game objects, such as the coin and umbrella, whose printed name and vocabulary are the same as the term by which the object is referred to in the code. TADS doesn't do this. In TADS, you have to specify separately the object's code name, the vocabulary by which the player can refer to it, and the term the game will use when it prints out a message about the object. That's why, in this very simple case, we have:

   ```
   coin: Thing 'coin' 'coin'
   ```

   With such a simple object definition, the information looks redundant and cluttered, but in complex cases involving objects that can be referred to by the player using a variety of nouns and adjectives, this way of creating an object gives the author more flexible control and prevents confusion.
8) T3 code uses both single quotes and double quotes, and the two are not interchangeable. The code shown above happens to use single quotes. That's why the apostrophe in "shouldn't" is preceded by a backslash. The backslash tells T3 that that single quote (the apostrophe) is *not* the end of the data. Instead, it is to be treated as an ordinary apostrophe *within* the data.
9) The line "if (me.isIn(livingRoom))" — and that's pretty readable code, isn't it? — invokes one of the standard library methods of the me object. In fact, every physical object in the world you create in a TADS game has an isIn method. The methods of an object are invoked by typing the name of the object, then a period, then the method being invoked. (A method is basically a function that's part of the code for an object.) We send the isIn method a location, using the parentheses — (livingRoom). The isIn method responds "true" if the object's current location matches the location we're asking about. If they don't match, isIn returns "nil". The value nil is an important one in TADS; it corresponds to the value "false" found in many computer languages, and has some other uses as well. So the value of me.isIn(livingRoom) will always be either true or nil, and that's what we're testing.

10) Each object has a class (that is, a type). The me object is an Actor. The umbrella is an OpenableContainer, which of course is very similar to the I7 openable container kind. The coin is a simple Thing. In Inform, if something is a Thing, you often don't have to mention that fact explicitly, because Inform will guess that it must be a thing if it's in a room, or in a container, or being carried. TADS won't try to guess; it wants you to be specific.

With those preliminaries out of the way, we can look at how the rules we've devised for the umbrella are coded. This is done in the dobjFor (direct object for) and iobjFor (indirect object for) action handlers of the umbrella object. While the code looks different from Inform code, the underlying idea is similar. In Inform, we use an Instead rule. In TADS, we use a verify routine. (T3 also provides a check routine and optional preconditions for each action. This scheme is similar, though not identical, to I7's before/instead/check/carry out/after/report hierarchy. We won't get into the technical details in this tutorial.)

The first rule, that the umbrella shouldn't be opened in the living room, will be needed when the player types OPEN UMBRELLA. This invokes the Open action on the umbrella object. To handle the Open action, we use the statement dobjFor(Open). Within the block of code that this statement introduces, we use T3's standard verify routine, which is part of every action handler. We test whether the player is in the wrong location for opening the umbrella by writing "if (me.isIn(livingRoom))". If this test is true, we use the illogicalNow command to explain to the player why opening the umbrella is, at the moment, a bad idea. illogicalNow also causes the action to fail, very much in the way that an Inform Instead rule interrupts action processing. However, if the action is not currently illogical — that is, if the logical test about the player's location evaluates to nil, meaning the player is *not* in the living room — we need to pass the command on to the standard handler for the Open action, so we use the special word "inherited". This is because there might still be some other reason why the action ought to fail. For instance, the umbrella might already be open, in which case we want TADS to handle the command in its normal manner.

The second rule, that things shouldn't be put into the umbrella even though it's technically a container, will be needed when the player types PUT COIN IN UMBRELLA. In this case, the umbrella is the second object in the command. It's not grammatically an indirect object, but for TADS purposes we can look at it as an indirect object, so we use the iobjFor(PutIn) action handler. Again, we use the verify routine, but in this case the action is always illogical — that is, we never want the player to be able to put anything in the umbrella. So the verify routine has only one command: illogical. This is followed by the text that will explain to the player exactly why the action would be illogical.

In a real game, these objects would be a bit more complex. They would need descriptions, at the very least. In addition, we would probably want to prevent the player from bringing the umbrella into the living room when it's open. To do this, we might use a TravelBarrier object, but explaining how a TravelBarrier is used would take us off on a long side trip. Because this is not a full-on T3 tutorial, we're not going to pause to explain every detail of how the dobjFor and iobjFor action handlers work either, or how exactly you'll write code for them. All of that information is found in the T3 documentation. The point of this example is, more simply, to show you the TADS equivalent of Inform rules.

Classes and the Library

One of the main differences between I7 and T3 is that I7 provides the author with a rather spartan set of kinds — room, door, container, supporter, vehicle, person, device, and a few others. (Many other kinds are defined in Extensions or in the Examples that abound in *Writing With Inform*.) The T3 library, which is installed as part of your TADS 3 download, serves up a much larger set of object classes than core Inform.

One result of this is that there's more that you'll need to learn in order to get the most out of T3. Another result is that you can produce a richer game using only the T3 library, without needing to download or install any extensions. (TADS 3 does have some downloadable extensions, which you'll find at www.ifarchive.org/indexes/if-archiveXprogrammingXtads3XlibraryXcontributions.html. Some of the items on that page are old, and are no longer needed because their features are now part of the standard library. Others are still very useful.)

Some of the T3 classes operate very much like their I7 counterparts. Others, such as the Door class, are similar in function, but must be deployed in slightly different ways. Still others, such as ComplexContainer, are extremely useful but unlike anything the I7 author is likely to have encountered.

Details on how to use the more broadly useful Classes are to be found in the documentation, specifically in *Learning TADS 3*. The *TADS 3 Tour Guide* introduces a large number of classes and shows how to use them, but doesn't provide every detail that you may need in order to use a given object in your game. The *Library Reference Manual* provides a great deal of detail, but it's not a how-to guide, it's a reference document. A section of the *Quick Start Guide* offers some concrete tips on how to use the *Library Reference Manual.*

Inheritance

The TADS 3 classes form a hierarchy. That is, classes that provide more detailed behavior are derived from classes that are more general. For example, the ShipboardRoom class is derived from the more general Room class, and the StretchyContainer class is derived from the Container class.

That much may be intuitively obvious. What may be less obvious is that T3 allows a single object to inherit its properties and methods from more than one class. This is called *multiple inheritance,* and it's not found in Inform 7 (though it is a feature of Inform 6). There will be times when you need to create an in-game object that combines the properties of two or even more classes.

The syntax for doing this is easy to use, but it has to be handled with care. We could, for instance, do something like this example from *Learning TADS 3:*

```
peg: RestrictedSurface, Fixture 'wooden peg' 'peg'
    validContents = [floppyHat, brownCoat]
;
```

The point of interest here is that the peg object is both a RestrictedSurface (which means that only certain things can be put on it — the list provided in the validContents property) and a Fixture (which means it won't be mentioned in room descriptions). These two terms form the class list of the peg object. And here's a key point: In some cases, it will make a difference what order the classes are listed in. The details are beyond the scope of this tutorial; if you're curious, you can search for the term "multiple inheritance" in *Learning TADS 3.* Class inheritance can be a bit tricky, and it's something that Inform authors may not have much experience with, but it's a powerful tool, one that will save you a lot of time and trouble if you decide to start using T3.

Templates

The TADS 3 programming language makes extensive use of shortcuts that will make your code easier to type and easier to read. These shortcuts are more formally known as templates and macros. If you're familiar with more standard programming languages of the C family, you may be thrown off by TADS code at first, because code written using a template doesn't look much like standard C code. Since you'll be using templates often (sometimes without even knowing that you're doing so), we need to take a quick look at them here.

Before your source code is compiled into a game that will run, a pre-compiler runs through the code and expands code that's written in template format. Maybe the best way to explain how this works is with an illustration.

All objects in your code — and pretty much everything in TADS code is an object — have properties and methods. A property is just a piece of data that's part of the object. A method, which is also part of the object, is a way of doing something. Like other objects, a Thing (that is, an instance of the Thing class, which includes most of the in-game objects you'll be creating while you write your game) will have some specific properties. For instance, it will have a name, one or more vocabulary words, and a description. These elements are so nearly universal that T3 provides a template that allows you to write them out more quickly.

If you don't use the template, you might provide a definition for the coin object we used in the earlier example (we'll make it slightly more elaborate, in order to illustrate the template concept) like this:

```
coin: Thing
    name = 'gold coin'
    vocabWords = 'gold shiny metal Spanish coin/disk/doubloon'
    desc = "The gold coin appears to be a Spanish doubloon. "
    location = me
;
```

This object has four properties --- name, vocabWords (vocabulary words), desc (description), and an initial location. Each property, as it's being created, is followed by an equals sign and then the data that is contained in the property. (The description is a double-quoted string, a topic to which we'll return shortly.) Note that there are no semicolons after property declarations.

Using the template for the Thing class, we can save time by writing out the coin object like this:

```
coin: Thing 'gold shiny metal Spanish coin/disk/doubloon' 'gold coin' @me
    "The gold coin appears to be a Spanish doubloon. "
;
```

This code is exactly equivalent to the more verbose version. Using the template, we give the code name of the object, then state its class, then list the vocabWords, then provide the name, then (optionally) specify the object's starting location using the @ sign, and finally type out the description. We haven't used the property names (name, vocabWords, desc, and location) at all, but the T3 compiler will understand exactly what each item is.

Using templates is optional, and there are times when spelling out an object's properties is preferable, but this type of code is found throughout every T3 project, so you need to understand what it is when you see it.

Texting

T3 code uses both single-quoted and double-quoted text (better known in computer programming as strings), and the two are not interchangeable. Using a single-quoted string when a double-quoted string is needed, or vice-versa, is guaranteed to make your game misbehave.

A single-quoted string is simply a piece of raw data. It can be manipulated in various ways, and may be used by TADS in various ways while your game is running. A double-quoted string is pretty much the same as an I7 "say" command. When T3 encounters a double-quoted string, it will just go ahead and print it out so the player can read it.

The reality is not quite that simple, because T3 will sometimes need to do some processing on your double-quoted string before printing it to the screen. Also, most single-quoted string data ends up being printed to the interpreter window at some point, in some manner. But if you think of double-quotes as a way of saying, "Print this right now so the player can read it," you won't be far wrong.

When you need to use a single-quote character (an apostrophe) within a single-quoted string, or a double-quote character within a double-quoted string, you need to precede it with a backslash, like this:

```
"Bob glares at you. \"I don't think I like your tone,\" he declares. ";
```

Notice that the apostrophe in "don't" didn't need to be preceded by a backslash, because the context is a double-quoted string.

Most TADS interpreters will print out “curly quotes” rather than the straight up-and-down kind, and will do a reasonable job of guessing when to use which kind. There are ways of coding curly quotes directly into your text, but that’s a power user tip that would take us beyond the scope of this tutorial.

You may notice one other thing about the code above: There’s a space before the closing quote. This is normal and expected with double-quoted strings in T3, and also with single-quoted strings that are intended to be complete messages (sentences, in other words). T3 has a built-in output formatter that assembles the text output before sending it to the interpreter’s window, where it will be displayed. Adding a space before the end of your quoted material insures that the output formatter will always format the text correctly. The places where you don’t want a trailing space are in lower-level text strings like the name property of an object.

Developing and Testing

Inform games are written in one long file of source code, although any extensions that the game makes use of will be in separate files. In TADS, it’s normal to break up a game of any significant size, putting various parts of the code in different files. For instance, if your game is set in an English manor house, you might have separate files for the upstairs area, the downstairs area, the garden area, the butler if he’s an important character, and the in-game hints. These files all have names that end with the .t filename extension.

Having multiple files makes it easy to find sections of code that you’re working on, because in a multi-tabbed editor program, you can switch back and forth and each tab will remember the spot where you’re editing. In addition, as your game becomes larger you’ll find that having multiple source files allows the game to compile more quickly (which speeds up testing), because the TADS compiler only needs to re-compile files that you’ve changed since the last compilation.

Each code file that you create should have these lines at the very top:

```
#charset "us-ascii"
#include <adv3.h>
#include <en_us.h>
```

One small but important point that Inform authors may stumble over: In the Inform IDE, clicking Run will automatically save your source code before compiling it. None of the TADS development environments do this. Before compiling your project, you must remember to save any code files that have changed.

Workflow in Workbench

The Inform 7 IDE is designed — and very nicely — to appeal to non-programmers. It has a lot of functionality, but the user interface is very streamlined. The TADS 3 Workbench is designed more along the lines of the type of user interface that will be familiar to programmers.

From the Help menu, you can select Workbench Help. This brings up a clickable table of contents. There's no need to go through it here, but you might find a few Workbench tips useful:

1) Browsing the help files (both Workbench Help and the display of the various T3 documents) is much like using a Web browser, but the Back and Forward buttons in Workbench are easy to miss. They're small and green, and they're in the toolbar at the upper right corner of the main window.
2) The Edit menu has no commands for commenting or uncommenting blocks of code. To comment or uncomment a block, select it and then click the mysterious, easy-to-miss button in the toolbar that looks like two slash characters: //. (If you're not familiar with this style of commenting your code, you can learn about it in the "Further Programming" page of the *Getting Started Guide.*)
3) You can add source code files to your project by right-clicking on the Source Files folder in the Project pane. When you do this, the file will be added at the top of the list, and that's probably not where you want it. (In fact, your project probably won't compile when one of your code files is up at the top.) After adding it, drag it down to the end of the Source Files list.
4) Searching the documentation using the Search field in the Workbench toolbar does not search *Learning TADS 3*, because it's a PDF. I suggest keeping a shortcut to the PDF on your desktop.

Workflow on the Mac

To develop a T3 game on the Mac (assuming you're not using a Windows emulator and Workbench), you'll need to work back and forth among the following steps:

1) Develop your code in a text editor, such as TextWrangler.
2) Compile the game using the t3make command in Terminal.
3) Run the game in QTads to test it.

Each time you add a new source code file to your project, you'll need to add it to the .t3m file by appending a line like this to the end of the .t3m file:

```
-source gardenMaze
```

(This assumes your file is called gardenMaze.t.)

You'll find the RECORD and REPLAY commands, which are already implemented in your game, very useful if you're working on the Mac. You may want to create a folder within your game code folder in which to store all of the recorded scripts for the game. (See below for more on replay scripts.)

What you won't have access to while writing T3 games on the Mac are the debugging tools in Workbench, specifically the ability to set breakpoints and watch the values of expressions. But these professional programming features are not found in the Inform 7 IDE either, so if you're coming into TADS from the world of Inform, you're not likely to miss them.

Replay Scripts

T3 lacks the complex functionality and attractive graphic display of I7's Skein — but recording and replaying test scripts that quickly run through a number of in-game commands is quite easy. If you're using Workbench, you'll find that each play-through that you do during development is automatically saved in the Scripts panel. You can replay a script at any time by right-clicking it and choosing Replay.

If you're not using Workbench, you still have access to this functionality, but you'll need to operate it by hand. After launching your work-in-progress in an interpreter, type RECORD. You'll be prompted for a filename and save location. All of the commands you use as you go through the game will be stored in this file, and of course you can save as many different run-throughs as you like. Recording will end when you type RECORD OFF. To replay a recorded file, launch the game, type REPLAY, and choose the file you want to replay. These script files can be edited in a text editor, by the way, so you can revise them as needed without having to replay a long sequence of commands in order to create a new script.

One of the newer T3 extensions in the IF Archive (you'll find it at www.ifarchive.org/indexes/if-archiveXprogrammingXtads3XlibraryXcontributions.html) is a file called tests.t. By including this in your game, you'll add the ability to write Inform-style TEST ME scripts that will run through any sequence of commands you need to execute. This extension saves tons of time and typing during game development, and I recommend it highly.

Using the Documentation

If you're using Workbench, you can access the T3 documentation directly from the toolbar at the top of the main window by clicking on any of the book icons. If you're on a Mac or Linux machine, you may want to dive into the doc folder, find the file called index.htm, make an alias (shortcut) for it, and drag the alias to your desktop. When you double-click the alias, index.htm will open in your default Web browser. This page displays the covers of and links to the eight main T3 documents, so the alias will give you instant access to all of the information you'll need.

If you're using Workbench, you can quickly search the documentation for terms using the search field at the top of the main window. Non-Workbench users will have a bit more trouble searching, so getting to know what's in the documentation will be useful when you need to research the answer to a specific question. The good news is, *Learning TADS 3* is a PDF file, and *Getting Started* and the *Tour Guide* can optionally be viewed as PDFs. Your PDF reader software will let you search a PDF file for a particular term. Finding information by searching *Learning TADS 3* can be very fast, once you're familiar enough with T3 to know what you're looking for.

When you're starting your journey into T3, the most important documents are likely to be *Getting Started in TADS 3* and/or *Learning TADS 3*. The *Quick Start Guide* is shorter and will get you going more quickly. It contains a working sample game, but it doesn't explain many of the concepts you'll need to grasp as you move forward. Personally, I feel that *Learning TADS 3* is the best place to start; it's very detailed and systematic. However, it doesn't walk you through the development of a complete sample game. The *Quick Start Guide, Getting Started,* and the *Tour Guide* all have sample games (increasingly complex) that you can copy and edit if you want to.

The *Technical Manual* is a bit of a grab bag, but it's packed with useful articles on important subjects, so you should make a point of reading its table of contents to find out what's there, and skimming a few of the articles. As you're learning T3, you'll often need to consult articles in this volume such as "How to Create Verbs." The *System Manual* is also a grab bag. Some sections are useful for day-to-day programming challenges, others less so. Part VII of the *System Manual,* "Playing on the Web," will be of vital interest if you're planning to write a game that can be played online in a Web browser.

The *TADS 3 Tour Guide* is a sequel of sorts to *Getting Started*. It introduces most of the classes (types of objects) you're likely to use in developing a game, and illustrates their basic features with example code. It does not, however, give full discussions of the finer details of most of the classes. For that, you'll need to learn to use the *Library Reference Manual*.

When you want details on the features of the various T3 classes (Thing, OpenableContainer, Door, Room, Actor, Distant, Readable, and many others), you'll find the *Library Reference Manual* indispensable. However, I'd be lying if I didn't admit that at first this document is a bit intimidating — even bewildering. A brief description of what you'll find in the *Library Reference Manual,* and a few suggestions on how to use it, are to be found in the *Quick Start Guide,* so there's no need to reiterate them here.

Introduction to HTML TADS is a guide to some features with which you can customize the appearance of your game. For instance, you can turn the directional words (north, south, and so on) in your room descriptions into clickable links. You need to be aware, however, that HTML TADS is *not* standard HTML, and certain things you can do with it will not work if you're planning to deploy your game for online play in a browser using the Web UI.

A Word to the Wise

I've seen more than one aspiring TADS 3 author start out by attempting, at the very beginning, to do a complex bit of programming and quickly get discouraged and give up. The first time I tried to learn T3, in fact, I did exactly that. I started looking at the Heidi example game in the *Getting Started Guide* and got terribly confused by (and indignant about) the fact that I couldn't see how to let the player look through the window of the cottage, when that should be a perfectly natural thing to want to do with a window. T3 provides an Occluder class, which is useful in such cases, but I couldn't see how to use it in the way that I wanted.

The way to learn T3 is, I think, to proceed in a sensible step-by-step manner, doing the easy stuff first. Save the challenging features of your game for later, after you've become comfortable with basic coding. That may mean writing code for all of the rooms in your game before you start adding movable objects, gadgets that can be switched on and off, and so forth. NPCs are easily the most complex aspect of programming any text game, and in this introductory tutorial I've mostly steered clear of them, though the features for programming lifelike NPCs in T3 are very powerful. When you’re just starting your adventure in TADS 3 programming, I suggest that you do likewise. Customizing the library’s default messages — likewise, TADS has great tools for doing this, but if the first programming challenge you give yourself is, “I’ve got to set up the player’s inventory so that it’s displayed in alphabetical order,” you’ll have to learn about Listers, which are one of the deeper and less intuitive features of T3. Don’t worry about using Listers, Occluders, and TravelBarriers for a few weeks. If you stick with it, you’ll get there, and once you’re comfortable with the basics of TADS 3 programming, you’ll quickly get the knack of using these and many other powerful features.