# Inform Foundations

## Part of the Well-Versed Informer Series

2/12/2010
Jeff Nyman
Version 0.3

# Contents

# Introduction

## Starting Out

Creating a work of textual IF requires you to understand that you are creating a model world. Did you ever see that movie *The Matrix*? In that film, there was a simulated world that people wandered through and did things in; this was a world that was created by "The Architect." Well, that's sort of what you're doing: you, as the author/architect are creating a simulated world that you will populate with various objects and various characters. Inform helps you create a special such character that the reader is going to control. In gaming terms I guess this would be called the *avatar*. In fiction terms this would be called the *protagonist*.

The model world you create will have an initial state. The **initial state** is what you *assert*, as an author, in your source text. (For the time being, think of "source text" as the word processor into which you are typing your story.) You write *assertions* that say what locations exist, what things exist in those locations, and how those things are situated relative to each other. However, once you introduce characters and events, then your model world will **evolve from that initial state**. This most noticeably occurs when the player starts interacting with your world as the protagonist. You'll also handle all this "evolving" stuff as part of your source text. You'll do that by creating *rules* and *conditional phrases* that you structure to respond to the interactions of the player.

You will be writing assertions, rules and conditions as part of this guide. All of this will constitute what Inform calls your "source text," a term I mentioned a couple of times above. The source text does have certain conventions of how you write all this stuff. Going into this too much now will probably be more confusing than helpful. All you have to know to get started is this: full assertion statements should end with a period. You'll generally have a line break between assertions, although multiple assertions can be placed on the same line as long as each has a period. You can also put a paragraph break between assertions. Any other conventions that Inform throws at you will be covered as I go.

### Create the Project

There's actually a quick way to make what I just said clear and cover a few preliminaries that will get you started. So, first, follow these next steps to get your project started.

1. Start the Inform application.
2. Click the "Start a new project..." button on the introductory screen.
3. Choose a directory where you want the project to be stored.
4. For the name of the project, enter **Learning Inform**.
5. Enter your name as the author.
6. Click the "Start" button.

## Your First Source Text

Once you've done that, you'll have the start of your source text with the story title and author name given in bold. I usually refer to that as the **initial heading text** although I'm not sure that's an official designation. Enter the following under the initial heading text:

**SOURCE TEXT**

**"Learning Inform" by Jeff Nyman**

Use American dialect.
Use full-length room descriptions.

The Learning Lab is a room. "This is the Inform 7 learning lab."

Notice how the statements "Use American dialect" and "Use full length room descriptions" are separated only by a line break. The assertion "The Learning Lab is a room", however, is separated from those with a paragraph break. I could have used line breaks and paragraph breaks here interchangeably. Do what seems most useful and/or readable for you.

**Structure Sometimes Matters**
There are places in Inform where how you structure the source text matters – and matters very much. This includes the spacing between source text elements. I'll cover such issues when they come up.

You may have noticed that I mentioned *statements* and *assertions* above. An **assertion** is any source text that is used to tell Inform about how the model world should be set up (such as what it contains or what some object is called). For right now let's just agree that a **statement** is any line in the source text that is not part of an assertion and that serves to tell Inform some aspect of how the story should be presented or how Inform should operate while the story is being played.

In terms of what I actually added to the above source text, the first thing to understand is that Inform has a certain set of internal options that affect how the story is presented to the player. The first of the two above specify that Inform should use American spellings and conventions rather than British, the latter being the default. The second of the two specifies that Inform should always present a full room description to a reader, regardless of whether or not the reader has already seen that description. This is different from the Inform default which is that the reader sees a full description the first time a room is encountered and a reduced description on subsequent visits.

The next line of source text is actually the only required element so far. Inform requires the source text to contain at least one assertion that specifies a starting location for the protagonist. What that means is that you have to assert the existence of a room, which is what my assertion does. That assertion tells Inform that a "Learning Lab" room exists. Since this is the only location mentioned in my source text, Inform treats this as the starting location for the protagonist.

**The First Room**
An operating convention of Inform is that the first room asserted in your source text is the room that the protagonist will start in. You can modify this by making another assertion in your source text stating a different location where the protagonist starts. Don't worry about this for right now, just know that it's possible.

The text in quotation marks after the assertion of the room is meant to indicate the description that the reader will be presented with regarding the room.

**What the Protagonist Sees**
You're going to use text in double-quotation marks in Inform source text to display information to the reader. You will see this a lot in this guide. Just keep in mind that almost anywhere you are using double-quoted text, you are dealing with something the reader may actually see.

## Play the Project

Before we go too much further, do me a favor and just make sure Inform is working for you. Click the "Go!" button, which you can't miss:

← This is the "Go!" button. It will start your story for you.
You can also press the F5 key.

What this should do is start your story so you could actually play it. This is what you should see:

**STORY TEXT**

```
Learning Lab
This is the Inform 7 learning lab.

>
```

There's absolutely nothing to do here so any attempt to "play" this story at this point is a bit academic. But what this has done is given us some confidence that you'll be able to go through the rest of this guide and actually try things.

**Try It!**
You can certainly act as a reader and try out a few commands just to see that the story has a minimum degree of playability in place. For example, type in the following commands at the prompt:

- LOOK
- INVENTORY
- WAIT

One thing I want to note is that Inform has certain shortcuts that it allows you to take with various aspects of your source text. The danger of this is that it can lead to sloppy and/or inconsistent source text but the positive side is that you have some flexibility in how you structure your text. As an example, I mentioned that the double-quoted text for the "Learning Lab" is a description for that room. But how does Inform know that? Inform makes that association because I've placed that text right after the assertion of the room. You could, however, be a little more explicit by making an assertion like this:

**SOURCE TEXT**

```
The Learning Lab is a room.
The description of the Learning Lab is "This is the Inform 7 learning lab."
```

How you choose to do this is up to you. I tend to be a little more verbose in my source text – thus avoiding some shortcuts – to make sure the intent is clear when doing the equivalent of editing. My personal opinion is that many shortcuts can lead to a bit of ambiguity from a "reading my own source" standpoint. Beyond just my personal opinion, it is a fact that shortcuts can also lead to some ambiguity when Inform tries to parse your source text to create a story but I'm not going to get into that too much since I tend to structure my source text so that such ambiguities are removed.

**Note the Highlighted Text**
You'll notice that I highlighted some of the above source text. It's probably obvious that I do this to make it clear what I've either added or modified from the previous source you already had in place.

So now I have a room and that's nice. But without objects, there's not a whole lot I can do. More importantly, there's not a lot my reader can do! So the next step is to place a few objects in the room. As with the room itself, this can be done by *asserting* what objects exist and where they exist. Let's add the following to the bottom of your source text:

**SOURCE TEXT**

```
A ball is in the Learning Lab.
An air pump is in the Learning Lab.
A flashlight is in the Learning Lab.
```

So I have three objects here that Inform recognizes by the full name I gave them in the source text: "ball", "air pump" and "flashlight." I mentioned before that the "structure" of your source text *can* matter. Here's a perfect example in that Inform does consider your source text positionally in some cases. If you start this story right now, you'll see the following:

**STORY TEXT**

```
Learning Lab
This is the Inform 7 learning lab.

You can see a ball, an air pump and a flashlight here.
```

Notice that Inform listed the contents of the Learning Lab room but also notice that it did so with the same order that I listed the objects in my source text. To prove that this is what's happening, I could simply reorder the items in my source text and start the story again.

**Try It!**
If you want to prove this point to yourself, give it a shot. Change the order of the assertions for the objects and then notice how the story text changes.

As a matter of going forward, often I'll suggest something in the main text of the guide. When it's something I really want you to see, I'll include these "Try It!" sections but, in general, you should feel free to not take my word for everything. Inform makes it easy to hit the "Go!" button and check things out for yourself.

With all this being said about positional source text, please don't think that you have to worry this much about the positional elements of your source text at all times. For instance, there are ways to get items in a room listed in a different order from the default. It's definitely beyond the discussion I'm having with you right now, but just know that it's possible. (In fact, this is something that I cover fairly extensively in my Description and Locale guide, which is another entry in the Well-Versed Informer series.)

So going back to the source text just added, what's important to note here is that I just asserted the existence of three objects of a very specific kind.

To Inform's way of thinking, every object that makes up the model world has a kind. The Learning Lab, for example, is a kind known as '**room**'. That was fairly obvious since I said as much when I made the assertion. But then what are the ball, flashlight, and air pump? I didn't say *what* they were; I just asserted *that* they were. As it turns out, those objects are a kind known as '**thing**'. I could have made this even less ambiguous in the source text as such:

**SOURCE TEXT**

```
A ball is a thing in the Learning Lab.
An air pump is a thing in the Learning Lab.
A flashlight is a thing in the Learning Lab.
```

It's not entirely inaccurate to say that a '**thing**' is the most generic kind of object that Inform understands. Most other kinds are based off of (or derive from) this generic object. If Inform can't figure out from the source text what kind of object something is then it defaults to '**thing**', which is why I didn't have to specify it in my original assertions for the above three objects.

You might wonder how Inform would "figure out" from the source text what kinds of objects are being asserted. This essentially has to do with how the assertions are written in terms of various concepts like "supporting" and "containing."

**Your Source Asserts a Taxonomy**
The nature of what you type in your source text indicates to Inform what it has to create as part of the model world's initial state. That's an important notion to understand when working with Inform; it's the very basis for why you make assertion statements about the initial state of the model world.

Let's set up a few more objects to make this taxonomy concept a bit more than just an idea. Add the following to your source text:

**SOURCE TEXT**

```
A basket is in the Learning Lab.
An egg is in the basket.
A chair is in the Learning Lab.
A cushion is on the chair.
A glass case is in the Learning Lab.
```

This creates assertions very similar to what you did for the previous three objects. As in the previous case you are here asserting that three objects (a basket, chair and glass case) are **in** the Learning Lab. What you have just specified here is a **relation**: the concept of one object relative to another. Here you are saying that certain objects are "contained by" a certain room.

What you have also asserted here is that certain objects are "contained by" (**in**) other objects. And, as you'll see, this makes a huge difference in how Inform constructs your model world. Start your story right now and you'll see the following:

**STORY TEXT**

```
Learning Lab
This is the Inform 7 learning lab.

You can see a ball, an air pump, a flashlight, a basket (in which is an egg), a chair
(on which is a cushion) and a glass case here.
```

To show you that Inform is treating your objects as specific kinds of things based on these relations, I'm going to have you run a few commands here.

First execute the command `SHOWME BALL`. Your output should look like this:

**STORY TEXT**

```
> SHOWME BALL
ball - thing
location: in the Learning Lab
```

(You'll actually see more output than that but don't worry about that for now.) What that's telling you is that the ball is of the kind '**thing**'.

Now execute the command SHOWME BASKET. Your output should look like this:

**STORY TEXT**

```
> SHOWME BASKET
basket - open container
   egg
location: in the Learning Lab
```

That's showing you that the basket is of the kind '**container**'. The 'open' part is an adjective applying to the container. "Being open" is actually a property of the object. Properties are something I'll talk about later so for right now just focus on the fact that the basket is of the kind '**container**'. The output is also showing you that the basket is associated with an egg. It's not obvious from *that* output but from the previous game output – "a basket (in which is an egg)" – you can see that the basket contains the egg.

Now execute the command SHOWME CHAIR. Your output should look like this:

**STORY TEXT**

```
> SHOWME CHAIR
chair - supporter
   cushion
location: in the Learning Lab
```

Here you can see that the chair is of the kind '**supporter**' and from the game output – "a chair (on which is a cushion)" – you can see that the chair supports a cushion.

The thing to notice here is that you never explicitly told Inform that the chair was a supporter or that the basket was a container. So how did Inform know that? Inform "deduced" this by the relationships of the objects. You asserted that the cushion was **on** the chair. Inform turned that into "the cushion is supported by the chair." You asserted that the egg was **in** the basket. Inform turned that into "the egg is contained by the basket."

So I'm going to throw some more input/output at you here and you should feel free to try out these commands on your own:

**STORY TEXT**

```
> take the ball. put the ball in the basket.
Taken.

You put the ball into the basket.

> take the ball. put the ball on the chair.
You put the ball on the chair.
```

Since Inform recognizes the basket and the chair as certain kinds, it will allow certain actions to be taken with them, such as putting objects in or on them. That's what the above commands are showing you.

The flipside of Inform allowing certain actions is Inform *not* allowing certain actions. The easiest way to see this is to observe the following output (and please notice how the commands are worded):

**STORY TEXT**

```
> take the ball. put the ball on the basket.
Taken.

Putting things on the basket would achieve nothing.

> put the ball in the chair.
That can't contain things.
```

Since the basket is recognized by Inform as a '**container**', you can't put things "on" it. Since the chair is recognized by Inform as a '**supporter**', you can't put things "in" it. Now let's try one more command:

**STORY TEXT**

```
> put the ball in the glass case.
That can't contain things.
```

The glass case is certainly meant to be a container. So why isn't it?

**Think About It**

Before reading on, think about this for a moment. Look at the source text you already put in place and consider how Inform seemed to make distinctions between the objects you asserted, in terms of what they "are."

Now ask yourself: how did Inform "know" that? In the case where you explicitly asserted that something was of a certain kind – like a room – it probably makes sense. But what about where you didn't explicitly say as much?

Well, to figure this out, let's try that SHOWME command again.

**STORY TEXT**

```
> SHOWME GLASS CASE
glass case - thing
location: in the Learning Lab
```

Ah ha! So the glass case is of the '**thing**' kind, according to Inform. That would explain why I can't put the ball in it, even though I, as author, envisioned the glass case as a container. *That* is the key point! I personally envisioned the object to be a certain kind, but Inform can't read my mind. Yet Inform certainly seemed to do well with the chair and the basket, right?

The key thing to understand here is that the glass case is not recognized as a container because nothing was asserted to be in it (which would *implicitly* make it a container) and nothing was explicitly asserted

about it being a container. What that tells you is that Inform works to identify the places and objects being described by the nouns in your source text, and to build a model world based upon what it can determine about the relationships your source text expresses. Since I created a "cushion" object and said it was **on** the "chair" object, Inform determined that the chair was meant to be an object that can support things and thus is a '**supporter**' kind. Since I created an "egg" object and said it was **in** the "basket" object, Inform determined that the basket was meant to be an object that can contain things and thus is a '**container**' kind.

So how do I make my glass case a container?

**Think About It**
How do you think you can do this? Think back to what I said regarding making the source text unambiguous about what is defined as what.

The solution here is that you can make an explicit assertion that the glass case is a container. Here's one way to do it:

**SOURCE TEXT**

```
A glass case is in the Learning Lab.
The glass case is a container.
```

Or you can shorten your code a bit and change the assertion for the glass case to this:

**SOURCE TEXT**

```
A container called the glass case is in the Learning Lab.
```

Or even this:

**SOURCE TEXT**

```
A glass case is a container in the Learning Lab.
```

After putting in this source text (whichever way you choose), a SHOWME command on the glass case will now indicate that the glass case is recognized by Inform as a '**container**' kind.

I should also bring up that SHOWME is one of the debugging commands of Inform. A **debugging command** is a command that is meant to help authors look at the "behind the scenes" construction of the story while playing through that story. These are not commands that will be available to readers when you distribute the story to them.

**Explicit and Implicit Assertions**
You have to decide if you prefer implicit or explicit assertions, at least when you have the option. For example, even though I implicitly asserted that the basket was a container and the chair was a supporter, I could have also made that explicit as such:

A basket is a container in the Learning Lab.
A chair is a supporter in the Learning Lab.

The key thing is really making sure you do assert what your objects' kind is. That's the key to making your model world work as you expect.

All of what I just did set up enough situational elements that I can actually work with you to start doing some interesting things with Inform.

## Starting Out: Summary … and Things to Come

At this point you've learned that Inform attempts to construct the simplest model world that's consistent with the information that you provide in the source text. In order to do this Inform will parse through the source text, looking for sufficient clues. One of the first – and probably easiest – things Inform does is attempt to group the nouns in your source text into different categories known as **kinds**.

Then, based on the kinds, Inform will attempt to determine if any **properties** have been established for those kinds. So, for example, you already saw that Inform had some notion of an "open" property for a container. Further, Inform will attempt to determine if any of your assertions use properties and then make sure those are consistent with what properties your kinds can have. As a quick example, just to put this in context, while a '**container**' can have an open property, a '**supporter**' cannot. So if I had tried to somehow indicate to Inform that my chair was "open," Inform would have had a problem with that. Inform would not, however, have a problem with me asserting that my glass case was open.

And, again, all of this is determined from your source text: the assertions you make.

I'll be getting into more about kinds and properties in the next section so fear not. This was just a way to get some of the elements introduced to you earlier. I do want to consider a fundamental point, however.

In crafting Inform source text, you either **specify** (**assert**) or you **query**.

That point above is important. As you've already seen, you're going to be making assertion statements in your source text which essentially serve as a sort of specification of how the model world should be when the story starts. I also mentioned that things can evolve as the story is played out and, as such, what you originally asserted may change. That's why you also have to query. I won't go into details right this second but what that essentially means is that you ask Inform which assertions still hold true and which do not. As a simple example, to put that into context, while my containers may start off as open,

perhaps they get closed during the course of play. In that case, I can query Inform about my container objects and check if they are open or closed at any given time.

So one more final little bit of summary.

**How Inform Works**
Inform turns your description of the initial situation (your **assertions**) into a **model world**, in which the places and items important to the story are created as **objects**.

These objects come in different **kinds**.

All these objects are **related** in order to make up the model world. In terms of those relations, things can then be **in** rooms or containers; **on** supporters; **part of** other things; or **carried by** or **worn by** people.

Lots of detail here – but don't worry. All of this is going to be covered in more detail as this guide goes on. I'm just immersing you in some information here so that you get a feel for what you're going to be learning about.

# World Building

Early on in this guide I mentioned the idea that the reader, acting as the protagonist, will try to do certain things in the model world. I now want to provide some context here as to what the "model world" means in terms of Inform implementation.

## Elements of the Model World

### Kinds

The physical and abstract basis of the Inform model world is predicated upon **kinds**. A small set of kinds are automatically understood by Inform. These are sometimes referred to as the **built-in kinds**. All of these are tabulated in the Kinds index.

**Kinds Index**
To see the Kinds index, you just click the 'Index' tab and then click the 'Kinds' button.

So what you do, as an author, is create a specific object of a given kind. Here "specific object" just means something that's uniquely identifiable. Let's consider this in relation to what you've already defined as part of your source text.

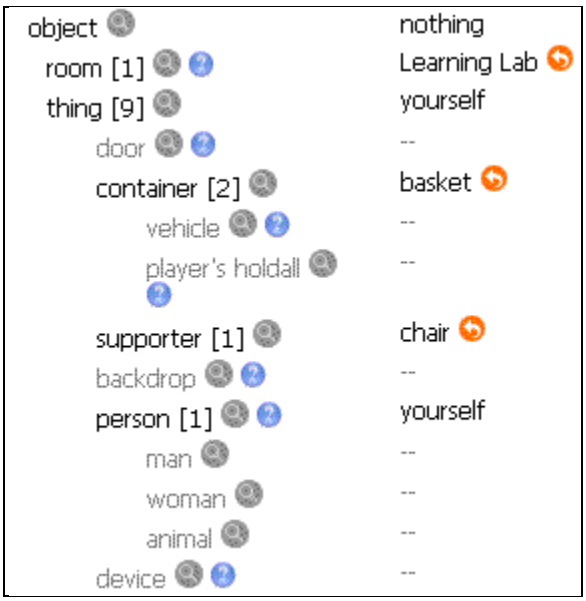
- You don't just have a '**room**', you have a Learning Lab.
- You don't just have three '**things**', you have a ball, an air pump and a flashlight.
- You don't just have two '**containers**', you have a basket and a glass case.
- You don't just have a '**supporter**', you have a chair.

The chair, the basket, the ball – those are *specific* instances of their respective *generic* kind. Specific **instances of kinds** build out the model world. At a certain level of approximation, the model world consists simply of two kinds: **rooms** in which are **things**. Even your containers and supporters are really just a kind of thing, ultimately.

Let's make sure to take a look at the "Kinds" index. As you do this, I do want to bring up one thing here that can be handy as you go through this guide. Sometimes you may want to check how Inform is building out your model world but without necessarily playing the game. That's largely what the "Index" in Inform is for. The "Index" is broken up into different areas. When you press the "Go!" button, you are essentially having Inform build the game – based on the information in your source text – and then presenting it to you so that you can play through it.

However, you can also do a simpler thing which is called "Refeshing the Index" and is currently available by pressing the F7 key. What this does is make Inform build the story as it normally would but skip the playing it part and instead present you with an up-to-date index based on what was understood about your source text.

So with that said, go ahead and refresh your index and then check out the "Kinds" portion of the index. You should see something like this:




Since I digressed a bit, let me just quote myself from above:

> "At a certain level of approximation, the model world consists simply of two kinds: **rooms** in which are **things**. Even your containers and supporters are really just a kind of thing, ultimately."

You can probably now see what I mean by looking at the visual of the Kinds index. At the root of everything is this notion of an **object**. That's about as generic as you can get to Inform. From that Inform creates two base-level kinds: **room** and **thing**. Indented under '**thing**', you can see that Inform lists other kinds, most notably **supporter** and **container**. The bracketed number tells you how many of that kind has been found in your source text.

You've seen that you can create your own instances of kinds. You can also "sub-categorize" a kind. This is, in fact, what Inform does when it shows that a '**container**' is "under" a '**thing**'. What that's saying is that a container is essentially a 'kind of thing.' The same would apply to a supporter and even to a person, as you can see from the Kinds index. Here are some other examples that you could have in your own source text:

> A dead end is a kind of room.
> A trash can is a kind of container.
> A table is a kind of supporter.
> A robot is a kind of person.

The key part of those assertions is the part that says "is a kind". What these assertions do is tell Inform that now rooms have a "sub-kind" called a dead end, and people have a "sub-kind" called robot.

It's fairly important to understand all this so let's break this down a bit.

- When you asserted "The Learning Lab is a room" you were creating an **instance of a kind**. In this case, the kind was '**room**' and the instance was 'Learning Lab'.

- If you assert "A dead end is a kind of room" you would be creating a **sub-kind**. You have not created an instance yet. To do that you would assert something like "The Alleyway is a dead end."

Now let's consider this in relation to the glass case that you created. When you asserted "The glass case is a container" you were creating an instance of a kind. In this case, the kind was '**container**' and the instance was 'glass case.' But what if you wanted to make a general sort of container, called a case, and then have specific instances of cases?

In that case, you could make the following assertion:

A case is a kind of container.

You just created a sub-kind, called a 'case'. You haven't created an instance yet but now, unlike in the glass case example, you can assert the existence of different kinds of case:

A glass box is is a case in the Learning Lab.
A lead box is a case in the Learning Lab.

**Think About It**
Consider the distinction just brought up between **instances of kinds** and **sub-kinds**. Make sure that distinction makes sense to you. This is one of those things that will soon become natural as you write source text, but I've found it helps to at least call out the distinction early in the learning process.

To make sure this stuff is hitting home, let's actually try this out. Modify your source text as follows:

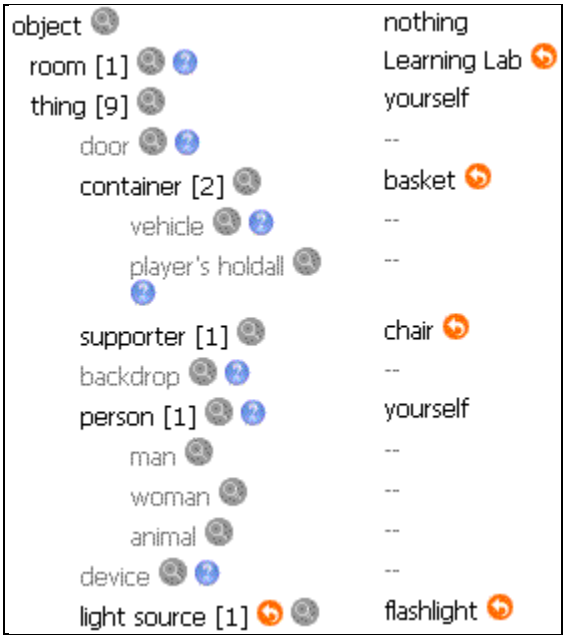
**SOURCE TEXT**

```
...
A light source is a kind of thing.

A ball is a thing in the Learning Lab.
An air pump is a thing in the Learning Lab.
A flashlight is a light source in the Learning Lab.
...
```

**Note the Ellipses**
Any ellipses (...) indicate that the source I'm showing you is appearing in front of, behind, or in between already existing source. The goal is just to provide a relatively simple but effective way in which you can decide where to position additional source text.

All I've done here is had you say that a '**light source**' is a specific kind of thing. A light source is now a sub-kind, by the discussion above. Then I asserted that the flashlight object is, in fact, a '**light source**' rather than just a '**thing**'. Here's what your Kinds index should look like:




Notice at the bottom that Inform now recognizes a new basic kind, called **light source**. That new kind is indented under **thing** because I've said that a '**light source**' is a kind of '**thing**'.

If you execute your story, however, you'll see that there's no apparent difference. The flashlight still shows up being listed as part of the room description. The reader could still take the flashlight, drop the flashlight, and pretty much do whatever they could have done prior to it becoming a slightly more specific kind of thing. The main reason for this is that the flashlight *is* still a '**thing**' and, as such, has the same properties as a generic '**thing**' would have.

## Properties

Assertions can also be used to establish properties of the objects that you set up. All kinds have **properties** appropriate to that kind. Some kinds share properties but some properties are very specific to a given kind. There are also different types of properties. Some properties are either/or ("open" or "closed" but not both and not neither), while others have values (a "matching key" object of a lockable door, for instance, or the "description" text of a room, the latter of which you already have in your source text).

Let's modify the source text a bit:

**SOURCE TEXT**

```
A flashlight is a light source in the Learning Lab. "Flashlight initial appearance."
```

With this addition, there are now two properties to consider in your source text: description and initial appearance.

The **description property** of a room is the text revealed when the reader first enters that room or any time they type a LOOK command in that room. The description text is the double-quoted text that comes after the room assertions in your source text. You've already seen this when you created the Learning Lab. The **initial appearance property** of an object is the text that is reported as the reader's first sight of some object in a room. This text appears as a separate paragraph in the text describing whatever room the object is in. If you run your story right now, you'll see this:

**STORY TEXT**

```
Learning Lab
This is the Inform 7 learning lab.

Flashlight initial appearance.

You can also see a ball, an air pump, a basket (in which is an egg), a chair (on which
is a cushion) and a glass case (empty) here.
```

Notice how the flashlight text is no longer contained within that list of items but instead stands off on its own. This specific text for the flashlight will continue to be used until the first time the reader picks the flashlight up. If the player never does so, then the initial appearance text will be used every time the player receives a description of the Learning Lab. So this property normally describes things in their original, undisturbed ("initial") context. The initial appearance text is the double-quoted text that comes after the object assertions.

It might seem a little confusing that the double-quoted text that comes after rooms is different than the double-quoted text that comes after objects. If you feel that's the case, these properties can be called out a little more explicitly (just as was done with the Learning Lab's description) as such:

**SOURCE TEXT**

```
A flashlight is a light source in the Learning Lab.
The initial appearance of the flashlight is "Flashlight initial appearance."
```

One thing that's important to call out again is that the use of properties depends very much on whether a given kind allows that property to be set. Even this simple example bears that out since rooms do not have an initial appearance. But both rooms and objects can have a description.

Let's try this out just so you can see it for yourself. Add the following source text:

**SOURCE TEXT**

```
A flashlight is a light source in the Learning Lab.
The initial appearance of the flashlight is "Flashlight initial appearance."
The description of the flashlight is "Flashlight description."
```

Unlike the initial appearance – which displays all the time as part of the room description, until the object is moved – the description for an object only appears if the player specifically types a command like `LOOK AT FLASHLIGHT` or `EXAMINE FLASHLIGHT`. What this is telling you is that any object that is a '**thing**' kind (or a kind of thing) can have a description property and an initial appearance property. The same is not true for objects that are a '**room**' (or kind of room) as you can see by adding this source text:

**SOURCE TEXT**

```
The Learning Lab is a room.
The initial appearance of the Learning Lab is "Learning Lab initial appearance."
The description of the Learning Lab is "This is the Inform 7 learning lab."
```

This will get you an error from Inform when you try to play the story or refresh your index. What the error means is that Inform couldn't translate your source text into a working story. The important part of that error is where it says "the property initial appearance for the Learning Lab is not allowed to exist." This is largely because you have not asserted that such a property can exist for a kind of room and that notion is not something that's built-in to Inform.

That last point is important. One thing to get out of what I've shown you here is that Inform has **built-in properties**. A good example is the description property (which Inform asserts can apply to rooms and things) and the initial appearance property (which Inform asserts can apply to things – or anything that is a kind of thing, such as containers, supporters, etc). Beyond this, some of the built-in properties that Inform provides are what I would call "external" and will be established as part of your assertions (such as description and initial appearance) and some of which are more "internal," in that you probably won't use them in assertions but that Inform will use to keep track of how your model world is evolving.

None of the Inform documentation uses that "external / internal" distinction so let me just spell out that internal part a bit more. As one example, Inform uses an either/or **handled property** that is set for any object that has been held by the player. In fact, it's this property which determines whether the initial appearance property text is displayed. Specifically, if the handled property for a given object is set to true, then Inform treats the object as being out of its "initial" state and thus does not display the initial appearance text.

There is also a **mentioned property**, which is used in constructing room descriptions to decide what objects should and should not be listed as part of that room description. A **visited property** is used by Inform to keep track of whether or not the protagonist has been to any given room, or – to be more specific – whether the protagonist has ever completed a "looking" action in that room. There are other such properties but for now I just wanted to give you a flavor of what Inform provides.

Before continuing, let's clean up our source text a bit. In the source text below, anything with a strikeout is something that you should remove.

**SOURCE TEXT**

The Learning Lab is a room.
~~The initial appearance of the Learning Lab is "Learning Lab initial appearance."~~
The description of the Learning Lab is "This is the Inform 7 learning lab."
...
A flashlight is a light source in the Learning Lab.
~~The initial appearance of the flashlight is "Flashlight initial appearance."~~
~~The description of the flashlight is "Flashlight description."~~

When making changes like this remember that you can always click "Go!" (F5) or "refresh your index" (F7) to make sure that what you have removed doesn't stop Inform from creating a working story.

So: back to properties. Just as you can create your own kinds you can also create your own properties. Let's talk about that just to make sure the appropriate context is in place. I'll break this down into some specific points.

A property can reflect the way a given kind may behave.

Here's some example source text of what I mean by that:

A dead end is either secret or ordinary.
A room is either indoors or outdoors.
A person is either awake or asleep.

These are what Inform calls **either-or properties** because the general format is this:

*{Some thing}* is **either** *{primary property}* or *{supplementary property}*

You can also think of these as **behavior properties**. The reason I say that is because conceptually you're saying that a person is either awake or asleep. On the one hand, you could argue that's a state (as opposed to a behavior). But I think you could equally argue it's suggestive of a behavior: i.e., a person can be in one two behavioral states. Likewise, a room may 'behave' in different ways based on whether it's considered an indoor location or outdoor location.

Let's modify your source like this:

**SOURCE TEXT**

...
A light source is a kind of thing.
A light source can be switched on or switched off.
...

Notice the slight nomenclature change from what I've been showing you. Instead of saying "A light source **is either** switched on or switched off", I used "**can be**" instead. This is another example of where you can use different wording to achieve the same effect in Inform.

You've just given your light source sub-kind its own specific property. And *now*, unlike before, an object that is a '**light source**' is a little different than an object that's just a generic '**thing**'.

The names of either-or (behavior) properties can be used as **adjectives** – i.e., "A secret room", "An asleep person", and so forth. So now you can have "A switched on flashlight" if you wanted.

The default assumption by Inform is that you don't want the *primary property* to be the starting state. Here, by "primary property" I mean the one that's listed first. For example, with that source text I just provided, the dead end would default to "ordinary" (which would really mean "not secret") and the room would default to "outdoors" (which would really mean "not indoors"). You can alter that default assumption by making assertions like these:

A dead end is usually secret.
A room is usually indoors.
A person is usually awake.

You can also make those assertions a little more certain by replacing "usually" with "always". Saying that a dead end is "*always* secret" is the same as saying that it's "*never* ordinary". By that same logic, saying that a dead end is "*usually* secret" is the same as saying that it's "*seldom* ordinary". All of those are ways that Inform will let you word assertions.

There is something to make clear here. The above are assertions and as you'll hopefully remember those are statements about how the model world starts out. The model world can evolve, however, which means that even something asserted as being "always secret" can eventually become ordinary by actions taken by the reader.

So let's do another addition to the source text:

**SOURCE TEXT**

```
...
A light source is a kind of thing.
A light source can be switched on or switched off.
A light source is usually switched off.
...
```

All I've done here is actually state what Inform already would assume: that the primary property setting ('switched on', in this case) is not operative.

Incidentally, this is why I refer to the second property in an either/or property set as the *supplementary property*; it largely serves as just a negation of the primary property. In other words, Inform is really just keeping track of one aspect of the light source property: switched on. The notion of the switched off property effectively just means "not switched on" to Inform.

Let's consider another point of discussion.

A property can indicate a certain state that a given kind can be in.

Here are some examples of what I mean by that:

A dead end has some text called river sound.
A room has a number called the difficulty rating.
A person has an object called the favorite item.

Notice that instead of "is" (or "can be") in these statements, you have "has". Inform calls these **value properties** and while that may reflect what they actually *are*, conceptually I've found that's not helpful. I prefer to call them **state properties**.

So, with those examples above, a room has a certain number that can be associated with a "difficulty rating." That number could be any number but the point is that the number indicates the particular state of that room. Likewise, the "river sound" can potentially differ between different dead ends. That difference represents the state that a given dead end room may be in. (For example, perhaps one such dead end has the sound being blocked, which indicates a "river sound" state of "silence.")

Let's modify the source text again:

**SOURCE TEXT**

```
...
A light source is a kind of thing.
A light source can be switched on or switched off.
A light source is usually switched off.
A light source has a number called the intensity.
A light source has some text called the light state.
...
```

As with the other examples, what you've done here is give the '**light source**' kind some more specific properties that you can later utilize as part of your source text. Note that this is setting the foundation for a level of flexibility with any objects that you decide are going to be light sources. Right now you only have a flashlight, but you might eventually end up with various objects that can serve as light sources. The behavior and state properties for each such light source object can differ.

**Think About It**

Consider the life and times of the flashlight up to now. The flashlight started out as a generic '**thing**'. I then had you modify it to become a more specific kind of thing: a '**light source**'. Even at that point, however, a flashlight was still basically just a '**thing**.' There was nothing to differentiate it.

Then, however, you added a behavior (either/or) property to say a light source could be switched on or off. Then you added state (value) properties to say that a light source had a number referring to the intensity of the light and some text referring to the state of the light.

The upshot is that now a '**light source**' and a '**thing**' are quite different kinds.

I want to get one more core idea out there.

You can combine the idea of a new kind with a property.

You've already explored that very statement with your flashlight object. But here's another example of what that statement means:

```
A food is a kind of thing.
Food is usually edible.
Food has some text called flavor.
The flavor of food is usually "Not bad."
```

With the source text above, a new kind of thing has been created ("food") and has been given a behavior of "edible" (i.e., capable be being consumed) and a state of "flavor." Further, that state is said to usually be a somewhat generic message. That generic message, however, could be modified for certain food objects in your game when you wanted a more distinct message.

## Kinds of Value

A state property, as you've seen, is established by the "has" element. Those 'text', 'number', and 'object' elements that a kind "has" are actually kinds themselves, referred to as **kinds of value**. So Inform has logic that says a room "has" text called description and any object that is a thing "has" text called initial appearance.

You can also create your own value kinds. Since we're already giving the flashlight quite a workout here, 'et's put this into context with the flashlight. Add the following to your source text:

**SOURCE TEXT**

```
...
A light source has some text called the light state.

Battery charge is a kind of value.
The battery charges are strong, weak, depleted.

...

A flashlight is a light source in the Learning Lab.
The flashlight has a battery charge called flashlight battery charge.
The flashlight battery charge is strong.
...
```

Here you should notice that the flashlight is given a 'battery charge' (a kind of value) that is then referred to as the 'flashlight battery charge'. What you should take from this is the following:

- *Any* objects whose existence you assert in your source text can have a 'battery charge' because you set this up as a kind of value.
- A specific object – the flashlight – has been given a specific kind of value called a 'flashlight battery charge.'

You might not like the idea of having to confuse the issue between a generic 'battery charge' and a specific '*flashlight* battery charge.' You can shorten this if you wish by modifying your source text slightly:

**SOURCE TEXT**

```
...
A flashlight is a light source in the Learning Lab.
~~The flashlight has a battery charge called flashlight battery charge.~~
~~The flashlight battery charge is strong.~~
The flashlight has a battery charge.
The flashlight is strong.
```

Here I've kept the text to that's to be modified in place but struck it out, just to make it clear what needs to change.

Notice here that the kind of value ('battery charge') is specifically applied to a certain object, in this case the flashlight. As is usual with Inform, it's up to you in terms of what you prefer.

You can also make a kind of value a more wide-ranging effect. Here's an example:

```
Texture is a kind of value.
The textures are rough, stubbly, and smooth.
Everything has a texture.
```

Now *every* object in the story has a kind of value ("texture"). I like to refer to these sorts of things as **value conditions**, although that's not a term Inform uses. What I mean by the term is that an object (or a given set of objects) has a conditional element placed on it that allows it to take on one of a set of possible values. In a similar way, in your own source text you could have said 'Everything has a battery charge' but that's probably not what you would want since it most likely is *not* the case that everything in your model world will be a source of light and thus would be subject to a battery charge.

You can also supply kinds of value directly to just one single object without creating an entirely new kind of value. This, in contrast to what I just described, is referred to by Inform as an **object condition**. Here's an example:

> The glass case is either shatterable, crackable or invulnerable.

Inform deals with this by creating a new kind of value, whose possibilities are the named options supplied, and then giving the named object (the glass case, in this case) a property whose value *must* be one of the possibilities listed. The kind of value and the property are usually given the name of the object with the word "condition" tacked on: so in this case "glass case condition".

The named possibilities for a new kind of value can serve as **adjectives** – i.e., "A strong flashlight"; "a depleted flashlight"; "a smooth stone"; "a shatterable glass case", and so on.

## Objects and Values

Let's think about what you've just read about for a second. What you've really seen presented to you so far is two different sorts of noun:

- **Objects** – things, rooms, doors, people, animals, etc.
- **Values** – numbers, text, battery charges, etc.

So why make the distinction? The reason is that Inform provides slightly different facilities for handling objects than it does for handling values. Okay, but why is that? The main reason is because Inform makes the assumption that you have different needs for objects and values. In some cases that's true but in many other cases values often act very much like objects. So now set up this source text:

**SOURCE TEXT**

```
...
Battery charge is a kind of value.
The battery charges are strong, weak, depleted.

Light strength is a kind of value.
The light strengths are steady, dim, and flickering.
...
```

These are values, but you can create them freely, giving each of them names, just as you do with objects. This means you could have the following source text:

**SOURCE TEXT**

```
...
Light strength is a kind of value.
The light strengths are steady, dim, and flickering.
A light strength can be adequate or inadequate.
A light strength is usually adequate.
Flickering is inadequate.
...
```

Just as open and closed refer to an either-or property for containers – which are objects – now adequate and inadequate serve as an either-or property for light strengths – which are values. And values, just like objects, can also have value properties of their own. For example, you can add the following to your source text:

**SOURCE TEXT**

```
...
Flickering is inadequate.

A light strength has a number called effect radius.
The effect radius of steady is 10.
The effect radius of dim is 5.
The effect radius of flickering is 2.
...
```

With this, effect radius is a numerical property of a light strength – which is a value – in just the same way that intensity is a numerical property of a light source – which is an object (a kind of thing). You can add the new value to your flashlight object as such:

**SOURCE TEXT**

```
...
A flashlight is a light source in the Learning Lab.

The flashlight has a battery charge.
The flashlight has a light strength.
The flashlight is strong.
The flashlight is steady.
```

So far you've been setting a lot of properties or establishing values and making sure that all of these apply to a given object. That's one part of what you want to do as part of establishing your model world. The power of being able to do this, however, really shows when you all that information is used by Inform to describe the model world.

## Describing the Model World

Inform's language structure has a concept of descriptions. A **description** serves as the basis for specifying the circumstances in the model world as well as the object those circumstances can happen to or with. *Important distinction there*! The description is *not* the circumstances, but it *is* the basis for stating the circumstances.

A description can be any source text which describes one or more objects. A description might be as simple as "the Learning Lab", or as complicated as "open containers which are in the Learning Lab". More or less the only restriction is that a description must be unambiguous as to what counts and what does not count in terms of being applicable for the description. So, as an example, "open containers" is considered ambiguous as a description because it does not say *which* open containers, whereas "open containers which are in the Learning Lab" is not ambiguous because even though it doesn't give the names of specific containers, it does refer to *any* containers that are in a *specific* location.

Inform maintains an alphabetical list of nouns and adjectives which can be used in descriptions.

And where does Inform get that list of nouns and adjectives? Why, from your own source text, of course. You've just seen that by how you've created nouns and adjectives as you've gone through these examples with me. Inform, for its part, takes all of that information and allows you to **describe circumstances** based on that information. In other words, you could say that you want something to happen to the protagonist when a certain described circumstance – like a "dim flashlight" – is operative in the model world.

A description is essentially a noun phrase, which either refers to a distinct thing ("a dim flashlight") or to a kind of thing ("a switched on light source").

Nouns refer to:

- Specific things ("flashlight")
- Specific kinds ("light source")
- Specific kinds of value ("intensity")

A noun phrase itself can be modified with an adjective ("*flickering* flashlight").

I mentioned adjectives a couple of times above but let's look at them a bit more.

### Adjectives

Adjectives are important in that they provide an additional element to a description. You can apply an adjective to an object or a value. First let's consider **applying an adjective to an object**. Add the following to your source text:

**SOURCE TEXT**

...
The effect radius of flickering is 2.

Definition: A light source is active if it is switched on.
...

What you've just added is a **definition declaration**. The purpose of such a declaration is used to create a new adjective. This allows a light source to be described as an "active light source." The opposite adjective here is really "not active" but you could also provide a specific set of adjectives if you modify the above definition slightly:

**SOURCE TEXT**

Definition: A light source is active rather than inactive if it is switched on.

A key thing to realize here is that the definition is not just providing the adjective but providing a conditional regarding when the adjective should be considered an applicable descriptive element. In the above case, that condition is specified as such:

- A light source is allowed to be described as 'active' if the light source is set to its 'switched on' property.
- A light source is allowed to be described as 'inactive' if the light source is set to its 'switched off' property.

That conditional element, however, must actually be something that Inform understands and that can be resolved to applying to the object you are trying to provide an adjective for. Here are two more examples of what you could use in your own source text:

- Definition: A light source is active rather than inactive if it has an intensity greater than 5.
- Definition: A light source is active rather than inactive if its light state is "on".

Those work because they are consistent with what you've told Inform applies to any object that is a light source. Here are some other examples of adjectives applied to objects:

Definition: A supporter is occupied if something is on it.
Definition: A room is occupied if a person is in it.
Definition: A room is occupied rather than unoccupied if a person is in it.
Definition: a direction (called target) is viable if the room target from the location is a room.
Definition: a person is another if it is not the player.

Now let's consider **applying adjectives to values**. Here's an example that you can add to your source text:

**SOURCE TEXT**

```
...
Definition: A light source is active rather than inactive if it is switched on.
Definition: A light source is high-powered if its intensity is greater than 5.
...
```

There's something I want to point out here because it can be a common source of confusion. You could not have used the following declaration in place of the one above:

> Definition: An intensity is high-powered if it is greater than 5.

You can't use that because intensity is a property. It's a property that happens to be a number and that number is applied to a kind (in this case, a kind of thing). An adjective has to be applied to a kind as well. Yet with the above you would be trying to apply an adjective to a property. If it helps think of it this way: you would use an adjective as a descriptive element for an object ("a high-powered flashlight") rather than as a property ("a high-powered intensity").

Another element you can add to your source text is this:

**SOURCE TEXT**

```
...
Definition: A light source is active rather than inactive if it is switched on.
Definition: A light source is high-powered if its intensity is greater than 5.
Definition: A light strength is high-powered if its effect radius is greater than 5.
...
```

Here are some other examples of adjectives being applied with values:

> Definition: A number is round if the remainder after dividing it by 10 is 0.
> Definition: A time is late rather than early if it is at least 8 PM.

Note here that those adjectives are being applied to *any* number or *any* time while the examples you actually put in your source text were restricted and much more specific.

Adjectives can only be applied to kinds, such as kinds of things (light source) and kinds of values (light strength). Adjectives cannot be applied to properties.

Besides allowing for the qualification of a kind (such as "open container"), adjectives also allow you to provide a context for your assertions. First let's just consider a general example:

```
A person has a number called height.
Definition: A man is tall if his height is 72 or more.
Definition: A woman is tall if her height is 68 or more.
In the Living Room are a tall man and a tall woman.
```

These statements, all taken together, create a man 72 inches tall and a woman 68 inches tall. Now you might add the following to your source text:

**SOURCE TEXT**

```
...
Definition: A light source is active rather than inactive if it is switched on.
Definition: A light source is high-powered if its intensity is greater than 5.
Definition: A light strength is high-powered if its effect radius is greater than 5.
Definition: A light source is blazing if its intensity is 50 or more.
...
```

What that latter statement does is allow you to make an assertion like this:

> A blazing light source called the halogen lamp is in the Learning Lab.

The key additional element there is using the adjective 'blazing' as part of the assertion. This brings up a good distinction though, particularly one that is going to make sense as we go along. The distinction here is between *asserting* something at the start of play and *checking* for a circumstance (via a description) during the course of play. This gets a little tricky at this point because you only have some of the context in place to see what I'm going to be talking about. For that reason, this may seem like a whirlwind discussion without a lot of grounding. Just go through it for now, soaking in the details without necessarily worrying if everything feels completely understood. I'm going to come back to these topics when you do have that extra context.

With that being said, as an example, consider the first of the definitions you currently have in your source text:

> Definition: A light source is active rather than inactive if it is switched on.

With that declaration in place, you might think you could make an assertion like this:

> An active light source called the halogen lamp is in the Learning Lab.

This, however, won't work. (Feel free to try it if you don't believe me.) The reason this won't work is because the adjective 'active' only applies when a given behavior is allowed. In this case, the light source has to be switched on (a behavior property) for the adjective 'active' to make sense. You could, however, do this:

> A switched on light source called the halogen lamp is in the Learning Lab.

That would then allow you to talk about an "active halogen lamp" elsewhere in your source text, such as checking to see whether the lamp is active. At that point 'active' would make sense because the context of 'switched on' has been provided.

Consider the second definition:

> Definition: A light source is high-powered if its intensity is greater than 5.

Even with that declaration in place, you could not have an assertion like this:

> A high-powered light source called the halogen lamp is in the Learning Lab.

The reason is largely the same as above: the notion of 'high-powered' for a light source only makes sense in the case of a given intensity (a property applied to that light source) being above a certain value. You could, however, have a set of assertions like this:

```
A light source called the halogen lamp is in the Learning Lamp.
The intensity of the halogen lamp is 5.
```

That would then allow you to talk about a "high-powered lamp" elsewhere in your source text, such as checking to see if the lamp has its intensity property set to a certain value. In fact, you've already seen this in action in your own source code. Consider your third definition:

> Definition: A light strength is high-powered if its effect radius is greater than 5.

With that in place, you still couldn't do this:

> A steady light source called the halogen lamp is in the Learning Lab.

(Remember that the light strengths are asserted to be steady, dim, and flickering.) It may seem odd that you can't do this but here's why this is the case. You can create "a light source called something," but you can't create a "*steady* light source called something" because, at this point, the notion of a *steady* light source does not have meaning. But you've already seen in your own source text how to handle this:

```
A flashlight is a light source in the Learning Lab.
The flashlight is steady.
```

Just to make sure these concepts are getting drilled in, I'll present you with a few more general examples. Here's another bit of context:

```
Definition: A container is huge if its carrying capacity is 20 or more.
Definition: A container is large if its carrying capacity is 10 or more.
Definition: A container is standard if its carrying capacity is 7.
Definition: A container is small if its carrying capacity is 5 or less.
```

You've already seen that the idea of a container kind is built in to Inform. Along with that, a property called 'carrying capacity' is asserted to be a number that is relevant for containers. With that context provided, you could now make assertions like these:

```
The thimble is a small container in the Living Room.
The matchbox is a standard container in the Living Room.
```

The notions of "small container" and "standard container" would have made no sense to Inform prior to the context that was provided by the definitions.

## Descriptive Context

You've seen adjectives and kinds (and instances of kinds) so far and you've seen how those can be asserted in your source text. You've also just seen adjectives and how those can be defined in your source text and then used to provide more specific assertions. All of this culminates in you being able to provide very specific descriptions to Inform.

Consider these descriptions:

> an open container on the table
> a woman inside a lighted room
> an animal carried by a man
> a woman taller than Mark
> something worn by somebody

The key thing to get out of the above is that you can give Inform complex descriptions. Consider the first one and how it breaks down:

| open | container | on | table |
|---|---|---|---|
| *is an* adjective | *is a* kind | *is a* relation | *is an* instance of kind |

I haven't talked much about relations yet except briefly earlier in this guide, saying how all objects in the world are related in some fashion, and showing a few examples (such as with the chair and the basket). Here the relation being described is that of supporting: a table supporting an open container. Here are some examples that you will eventually be able to use in your own source text:

> anyone carrying a depleted flashlight
> an active blazing light source in the Learning Lab

Why can't you use those now? Well, first, what I want you get out of this is a very important point.

You can provide a descriptive context to Inform based on the state that an object is in, the type of object it is, and the relation of that object to other objects.

This is, in fact, a key element of how you're going to get Inform to handle an evolving model world. The reason you can't use these descriptions now is because you don't have a place to put the descriptive context. Yes, that's right: even the descriptive context has a context!

I'm going to hold off giving you that context right now except to tell you that it involves rules, which are the basis for pretty much everything Inform does. However, that's a bit more of a complicated subject than I can get into right now. Instead I'm going to show you some more supporting elements.

## Managing the Model World

### Phrases

A **phrase** is what you use to make changes to the model world or send messages to the player. Phrases are *active statements* ("award 3 points") rather than merely *descriptive statements* ("a blazing light source"). Here's a way to look at it: phrases only make sense if you know exactly in what circumstances they will happen. That's largely the case for descriptions, as well. Those circumstances are, in fact, the context I mentioned before.

Phrases take place in a certain context and that context is determined by the rules that phrases are a part of.

So here we are again: back at rules. I haven't talked about rules much yet because I'm going to be getting into that as I take you through an extended example. For now, just realize that Inform is all about rules. Rules use descriptions to provide the circumstances (or the context) for when a given set of changes should be made to the model world. The phrases are what actually carry out those changes.

### Conditions

A **condition** is a phrase which describes a situation which might be true or might be false. What you actually have are descriptions that can be used as part of conditional sentences. Earlier I mentioned that you assert and you query your model world. Conditions are how you do the query part.

You can use what I call **non-ranged conditions and descriptions**:

> if the oaken door is open
> if Jane is carrying an animal

What I mean by "non-ranged" is that these don't refer to a "range" of objects, but rather very specific objects. You can also use what I call **ranged conditions with descriptions**:

> if each door is open
> if anyone is carrying all of the animals
> if everybody is in the Living Room

Here "each", "anyone" and "everybody" are called **determiners**. These are very general determiners in that they refer to groups of objects or people. As you just saw, those can be replaced with specific determiners that refer to a specific object (such as "the oaken door") or a specific person (such as "Jane").

Before I provided some descriptions that I said you could eventually use in your own source text. Those descriptions were:

> anyone carrying a depleted flashlight
> an active blazing light source in the Learning Lab

Here's an example of how you could take those and put those in a condition:

> **if** an active blazing light source is in the Learning Lab **then** do nothing.
> **if** anyone is carrying a depleted flashlight **then** do nothing.

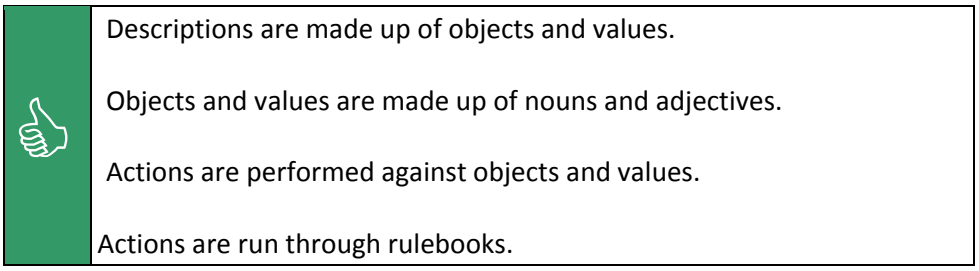
The bolded portions indicate the elements that are the condition. The fact that the above constructs make a complete "sentence" to Inform means that this condition is part of a phrase. What's still lacking for you is the context in which that phrase would be situated. And, once again, that context would be a rule, which is beyond where I want to take you right now.

# The Inform Context

I want to provide a very short summing up of what I talked about so far, but in a very general way: the Inform context. Meaning, the way in which Inform operates and the start of how you can start thinking in terms of how Inform works.

One of the main things that I haven't even mentioned yet is actions. An **action** is essentially something that the reader has the protagonist try to do in the model world. (This is a bit of a simplification but it will do for now.) All you need to understand right now is that most actions involve objects: taking a ball, for example, or switching on a flashlight. Actions might also involve values, or a mixture of the two: turning a dial to 10 would involve both an object (the dial) and a number (10). In all cases Inform has to try to relate the object or the value to something it already understands. What Inform "already understands" is solely predicated upon what you, as the author, have asserted in your source text.

That notion of Inform relating parts of a reader action to what it understands from the soursce text *is* the Inform context, at least at a high-level. The following is how that high-level translates down into Inform concepts:




There is a lot of detail behind all of this. What I hope the above does, however, is provide you with an operating heuristic – a way to guide your further understanding as you learn more about Inform. Part of that learning will take place next as I take you through an extended example, where I'll talk about actions, rules, and cover many of the elements you've already read about in this guide.

# A Working Example

The example source text I work with you on in this section is not going to serve as an actual, usable game and it's most certainly not going to serve as a working story. This example is simply designed to showcase some of the foundational aspects of Inform. I chose to do things this way because it would allow me to focus on concepts and techniques rather than having to worry about some cohesive game.

I think it's possible to break up learning Inform into conceptual points that will serve as a focus for a running example.

**Conceptual Points**

1. Establish the model world.
2. Establish permissible actions.
3. Establish action grammar.
4. Establish how the world model will respond to actions.

You've already handled the first point above with the assertions that you established in the prior sections. The remaining three items from that list will be covered here in this section.

## The Basis for Actions

A reader's exploration of a work of textual IF is made by a sequence of commands. Inform decides if it can understand the command. At the most simple level, this means Inform makes sure it's dealing with a noun and a verb. Assuming Inform can make structural sense of the command, it translates that command into a specific action.

Inform will attempt to translate reader commands into actions.

That's a critical step because Inform's primary job is to respond to those actions and update the model world as and where appropriate. Sometimes that updating means just printing a simple response to the reader. Other times it means opening doors, or allowing the protagonist to move from one place to another, or causing another character in the story to respond a certain way.

Assuming a translation from command to action is possible, Inform will then attempt to execute those actions against the model world. And what *that* really means is that Inform runs the actions through the set of rulebooks that make up the model world and waits to see which rule indicates it will handle the action. The point of this is that, from an authorial standpoint, much of the designing process comes down to responding to the actions that Inform will take in response to reader commands.

Regarding the distinction between command and action that I just brought up, one thing that it might help to mention early on is that a command (supplied by the reader) is an imperative verb (i.e., *take*) that is translated into an action (supplied by Inform or by the author) that is a present participle (i.e.,

*taking*). You'll start to see why this has relevance as I start going through how to write "action handling" elements in your source text.

A wide range of commands are automatically understood by Inform. These are sometimes to as the **built-in actions**. All of these are tabulated in the Actions index in various ways.

**Actions Index**
To see the Actions index, you just click the 'Index' tab and then click the 'Actions' button. An easy glance is provided by the section 'Actions in alphabetical order.'

To get started, let's first do some cleanup of the source text. In order to present this example in a relatively easy manner, without a lot of non-essential source text, let's restructure your source text so it looks like this:

**SOURCE TEXT**

```
"Learning Inform" by Jeff Nyman

Use American dialect.
Use full-length room descriptions.

The Learning Lab is a room. "This is the Inform 7 learning lab."

A ball is a thing in the Learning Lab.
An air pump is a thing in the Learning Lab.
A flashlight is a thing in the Learning Lab.
```

Now add one additional statement near the top of your source text:

**SOURCE TEXT**

```
...
Test me with "actions / take the flashlight / take the ball / drop the ball".

The Learning Lab is a room.
...
```

I created a 'test' statement that will let you see some action processing right away. Here I'm relying on the 'take' command (which Inform translates to the built-in 'taking' action). There are actually a couple of important things being demonstrated with this new statement and I do want to cover those. Incidentally, I put the 'test' statement near the top of the source text for convenience purposes, but it can go anywhere. Some people prefer all such statements to be at the bottom. It's up to you.

In Inform, 'test' statements let you provide a way to automatically run through your story with a series of commands that a reader would normally enter during the course of playing the story. This ability to test is going to come in *really* handy in your development, particularly as you make changes. Part of what I'll focus on from from this point forward is proving that statement to you.

The second thing to focus on is the command I provide at the start of the test list, which is ACTIONS. Remember how I used SHOWME and said it was a debugging command? Well, ACTIONS is a **testing command** that tells Inform that I want to see internal details about what Inform is doing as it processes any given action. Remember that each command is translated into an action by Inform. So what the ACTIONS command is doing is showing me what action is running for each command that is processed.

To see this in action, I can just start the story and enter the command TEST ME. (Incidentally, you don't need to use all capitals for commands. I do that in this guide just to separate commands from normal text.)

**Try It!**
When I say "I can just start the story," that should be read as "Hey, *you* can start the story and see what I'm talking about." So try it! Start your story and enter the command TEST ME.

You should see the following output:

**STORY TEXT**

```
Learning Lab
This is the Inform 7 learning lab.

You can see a ball, an air pump and a flashlight here.

> test me
(Testing.)

> [1] actions
Actions listing on.

> [2] take the flashlight
[taking the flashlight]
Taken.
[taking the flashlight - succeeded]

> [3] take the ball
[taking the ball]
Taken.
[taking the ball - succeeded]

> [4] drop the ball
[dropping the ball]
Dropped.
[dropping the ball - succeeded]
```

I should note that I'm probably not going to show the output like this as much in this guide from this point forward, unless I really want to showcase something specific. After all, with the test statements in place, you're going to hopefully be seeing it for yourself and my repeating what you see on your screen is usually of minimal benefit.

Let's break down what you've just seen. Consider the second action that was processed:

| | |
|---|---|
| **> [2] take the flashlight** | A command is entered by the reader. (In this case, the test statement simulated a command for you.) |
| **[taking the flashlight]** | Since ACTIONS is used, Inform is telling you what action was attempted. The square brackets indicate that this is not text the reader would normally see. |
| **Taken.** | This is the output from Inform of the action being processed. In this case, the default response of Inform for a successful 'taking' action is to report the text "Taken." |
| **[taking the flashlight – succeeded]** | Since ACTIONS is used, Inform now reports on the success or failure of the action. In this case, there was a successful attempt to take the flashlight. |

You can easily create a story where the built-in actions suffice, meaning you don't need to create any new actions at all because you don't anticipate the reader having to use any other commands beyond that which are provided. But, most likely, that's *not* all you want; you will probably want to create actions that are a bit more specific to the circumstances of your particular story.

## Creating a New Action

Let's say that you want to allow the ball to be inflated. The idea of inflating (or deflating) something is not a concept that Inform understands by default. In contrast, as I showed you already, Inform *does* understand the concept of taking and dropping something. But if the reader were to try a command like `INFLATE THE BALL`, Inform would respond with "That's not a verb I recognize."

So what I need to do is create a new action. Remember: an action is something that Inform maps a reader command to. So let's modify our source text like this:

**SOURCE TEXT**

```
Test me with "actions / take the flashlight / take the ball / drop the ball".
Test inflate_action with "actions / inflate the ball".
  ...
A flashlight is a thing in the Learning Lab.

Inflating is an action applying to one thing.
```

Notice that I added a new 'test' statement as well as put in an assertion statement that tells Inform there's an action called 'inflating.' Also, to make going through the next parts of this guide easier, while I kept the 'test' statement at the top of my source text, I put the "inflating is an action..." statement at the bottom of my source text. For the remainder of this guide I'm largely just going to add lines, one after the other, to the bottom of the source text. So unless you see ellipses indicating otherwise, just assume that every new statement should be provided at the bottom of the existing source.

**Try It!**
Start the story and run the command `TEST INFLATE_ACTION`.

The verb, as you'll see, is still not recognized. But that makes sense when you consider that I haven't told Inform about any specific verbs yet. The source text I added above works in the sense of creating an action – but I still haven't told Inform how to map a specific command (verb) to that action. In other words, Inform needs to know what grammar to expect when that action should be attempted. That way Inform will know that if a certain command (with a given grammar) is encountered, it should attempt the new 'inflating' action I just created.

Inform needs to know what verb the reader will use when attempting an action.

So referencing that list of conceptual points I started this section with, I just showed you how to **establish a permissible action**. I now have to show you how to establish action grammar.

Before I do that, let's talk about that test for a second. I know that my end goal is that Inform should respond to an 'inflate' command ('inflating' action). So I've crafted a test that should "pass" – meaning it should give me output that I expect, which, in this simple case so far, is Inform *not* telling me that I'm using a verb it doesn't recognize. So the test "failed" in this case, as the output shows. Inform still does not recognize the verb.

So I'll add the following line to the source text:

**SOURCE TEXT**

```
Understand "inflate [something]" as inflating.
```

I'll run the story again and use `TEST INFLATE_ACTION`. See how nice that 'test' command can be? It's even nicer when you combine running tests with the 'Replay' button.

← This is the "Replay" button.
You can also press the F9 key.

The 'Replay' feature is nice because what it does is take the last 'reader session' that was executed for your story and replays it: re-entering all the commands from the last session for you.

What I did is put an 'Understand' statement in place. I'll come back to what that means in a little more detail but right now it's almost literally what it sounds like: a statement that tells Inform that it has to understand one element – the double-quoted text – as being associated with another element – the action after the double-quoted text. The first element is a verb. The second element is what the verb will be applied to, which would really be the direct object of the command.

The addition of this source text will work in the sense that Inform will no longer complain about the verb not being recognized. Inform now does understand a command like `INFLATE THE BALL` although its understanding is very limited. At this point Inform only knows that this kind of action can be attempted with anything. That's what the "[something]" in the above statement is telling Inform.

What that means is that the reader will be able to type any word – or rather, any word that refers to an object in the current location of the protagonist – with the command INFLATE. More to the point, my 'understand' statement is saying that *any* object (or *any* thing) can be used as part of the 'inflating' action. This might be even more clear if you realize that you can use the following slight modification to the 'understand' statement:

**SOURCE TEXT**

```
Understand "inflate [anything]" as inflating.
```

Any text that appears in double-quoted text and is in square brackets is what Inform refers to as a **text substitution** element. In the case of the above statement, Inform will replace "[something]" (or "[anything]") with whatever direct object was used with the command.

Text that is inside double-quotes is considered to be literal text to be either displayed or processed *as that literal text*. Square brackets within double-quotes are used to describe what to say without giving the literal text.

I'll come back to some details of *what* I just did here but let's not lose the thread of the discussion.

You'll see, via the ACTIONS output, that Inform reports "inflating the ball – succeeded." The only problem is that success at the inflating action doesn't mean much right now. Nothing actually changes in the model world as a result of the reader giving the command ('inflate') or Inform attempting this action ('inflating'). Note, too, that a reader wouldn't even see that much output since ACTIONS is really a testing command. So, in effect, the reader would see no output at all as a result of their command!

**Try It!**
Take the ACTIONS command out of the 'test' statement then rerun the 'test' command. You'll see there's no output as a result of the command being executed as an action.

Think back to how I showed you the output of the 'take' command. At that point Inform already had a built-in notion of how to respond to such an action. Assuming all went well and the object could be taken, Inform simply responded with the report of "Taken." Simple as that message is, at least the reader knew *something* happened as a result of what they did. So what this means is that I need to have Inform report back to the reader when they try an 'inflate' command.

And here's where life gets interesting because I have to start getting into rules. Inform loves rules. Inform is essentially made of rules. So let's start with some heuristics that I found helpful to know to get started.

An action happens due to a command.

From Inform's perspective, any action will either succeed or fail.

Actions are subject to rules.

Those rules can determine how (or even whether) an action is carried out.

Those rules indicate whether the action has succeeded or failed.

So what's a "rule" then? A **rule** is basically a set of circumstances followed by a list of instructions. When those circumstances apply, the instructions are carried out. Circumstances? Instructions? Yeah, it can sound confusing. That definition is easier to explain by example, which I'm going to do shortly. Another thing to understand is that Inform groups rules together into what it calls rulebooks. A **rulebook** is a collection of rules that have a common purpose. Again, it will be easier to show by example than explanation.

So now I've gone through the third of the elements on that conceptual points list I provided at the start of this section: I've shown you how to **establish action grammar**. Before moving on to the fourth item in that list, let's cover a little more about action grammar since it has a few nuances.

### Digression: Qualifying Actions

Before I get too much further though I do want to stop and take a second to look at the two statements I had you add to the source text and consider a few details about them. Here's the two statements:

```
Inflating is an action applying to one thing.
Understand "inflate [something]" as inflating.
```

Here's a schematic for these two statements:

| **Action Statement** | | |
|---|---|---|
| *{past participle; action}* | *is an action applying to* | *{qualifiers}* |
| Inflating | | one thing |
| **Grammar Statement** | | |
| *Understand* | *"{present tense; action}"* | *as {past participle; action}* |
| | inflate | inflating |

What this is telling Inform is that there is some grammar that can be used in a reader command ('inflate') that will be mapped to a specific action ('inflating'). That action can only apply to *one thing*. What this means is that a command like `INFLATE NORTH` would not work, because the direction of north is not a thing. In a similar way, a command like `INFLATE THE FLASHLIGHT AND THE BALL` would not work. The reason for this is that this command specifies more than one thing as opposed to just one thing.

One thing that I think is helpful to point out is that a grammar statement can be general or very specific. Here's an example:

```
Photographing is an action applying to one visible thing and requiring light.
Understand "photograph [someone]" as photographing.
Understand "photograph [an open door]" as photographing.
```

Here you can see that there are two ways this command can be dealt with: either by photographing any particular person ("[someone]") or photographing a specific type of object ("[an open door]"). Both commands lead to the same action but what they do is constrain how the action can be called in the first place. If the reader were to have the protagonist try to photograph a non-open door or a non-person, the command would not even be recognized.

What I also want to make sure you see is that the action statement can provide different qualifiers. The 'photographing' action, for example, requires light. That means the action would not succeed in a dark room. Also, the addition of *visible* to the qualifier *one thing* means that the object of the action does not need to be capable of being touched by the protagonist; if the object can at least be seen by the protagonist, the action can go forward.

There's lots of detail here and I'm not going to go into every nuance but I do want to make sure these ideas are clear. So I'm going to have you make some temporary changes to your source text just to put a few of these concepts into action.

Add the following lines to the bottom of your source text:

**SOURCE TEXT**

```
The Learning Lab is dark.
The player carries an inner tube.
Test qualifiers with "inflate inner tube".
```

Run the story and type `TEST QUALIFIERS`. You will find that that inflating the inner tube works. You won't get any output but, as I already showed you, that's actually a success in this case since the 'inflate' command was recognized and translated into an 'inflating' action. Even though the room is dark now, this command works because the action statement says "one thing" without any mention of light.

Now I need you to modify your action statement like this:

**SOURCE TEXT**

```
Inflating is an action applying to one thing and requiring light.
```

Run the story and again type `TEST QUALIFIERS`. You'll find that this does not work.

**Test and Replay**
I just want to call out, once again, the handy nature of having a test and then being able to make a change to the source text with an immediate replay option. This allows you to quickly check the results of a localized change before you continue building more source text with that change in place.

The action is stopped in it's tracks because (1) I've asserted the Learning Lab is dark and (2) I've asserted that an 'inflating' action requires light. You should also note that this qualifier holds true even though the protagonist is holding the inner tube and thus presumably would not "really" require light in order to perform this act. Inform isn't looking at that kind of context: it's simply considering exactly what your qualifiers say.

So now let's make another modification to the source to try something new. First, make sure your action statement is back to how it was:

**SOURCE TEXT**

```
Inflating is an action applying to one thing.
```

Then add the following source text (with the understanding that anything with a strikeout in it should be removed):

**SOURCE TEXT**

```
~~The Learning Lab is dark.~~
~~The player carries an inner tube.~~
A container called the glass case is in the Learning Lab.
The glass case is transparent, lockable and locked.
An inner tube is in the glass case.
Test qualifiers with "inflate inner tube".
```

Now run your story and again run `TEST QUALIFIERS`. What you'll find is that the 'inflating' action fails to occur because the inner tube is in a closed, locked container. Thus the inner tube is not currently touchable by the protagonist. So when I specified "one thing" as my qualifier for the action statement, this really means "one *touchable* thing."

Now modify the action statement one more time to look like this:

**SOURCE TEXT**

```
Inflating is an action applying to one visible thing.
```

Run your story one more time, execute `TEST QUALIFIERS` and, lo and behold, the action now works. This is because the extra qualifier of *visible* tells Inform that this is all the "one thing" has to be: visible. It doesn't have to be touchable. Granted, it's a bit odd that the protagonist could inflate something that's inside a closed case but remember that Inform is not focusing on that context. That's parlty why I took you through this: to show you that you have to consider how you use your qualifiers for your actions.

With that being said, I'll stop this digression into extra source text. For now remove all the extra source text I had you add and make sure your action statement is back to its original wording:

**SOURCE TEXT**

```
Inflating is an action applying to one thing.
```

As I close this digression entirely, I'll just note that you can have varying types of structures for action and grammar statements. Here are some examples:

```
Casting xyzzy is an action applying to nothing.
Understand "xyzzy" or "say xyzzy" or "cast xyzzy" as casting xyzzy.
```

```
Adjusting it to is an action applying to one thing and one number.
Understand "adjust [something] to [a number]" as adjusting it to.
```

```
Setting it by time is an action applying to one thing and one time.
Understand "set [clock] to [time]" as setting it by time.
```

There's lots more to learn about this area of Inform. So just understand that I've scratched the surface here; hopefully enough so that you've got some context for the source text you're typing in.

## Reporting an Action

Okay, so where was I? Ah, yes. I needed Inform to report back to the reader when an 'inflate' command was translated to an 'inflating' action. Here's an attempt that I'll add to my source text:

**SOURCE TEXT**

```
A report inflating rule:
    say "You inflate [the noun]."
```

**Indentation Matters**
Inform cares about indentation. Indentation is a structural element within Inform source text. Further, that indentation needs to be a tab character. So when you see source text like the above, as you type it in, please make sure you indent with a tab.

With that new bit of logic in place, run your story and execute a `TEST INFLATE_ACTION` command. Okay, so that's nice. Now at least the reader is told that their command worked.

Except there's a problem and it goes back to the fact that my grammar statement contained the word "[something]", which is a generic qualifier meaning, essentially, *any* object. With that being said, you'll see that you can inflate the flashlight as well as the ball.

Eventually I'm going to want that *not* to happen and, further, I'm going to want to check that it *doesn't* happen by essentially preventing the protagonist from taking that kind of action. I'm also going to want

to make sure that the ball is *always* capable of being inflated. It's situations like this – where you'll want to check conditions and query your model world – that updating your tests comes in really handy.

So first I'm going to update my 'test' statement and I'm going to include a command that should succeed and a command that should fail. Keep in mind why I'm doing this. As I make changes to my source text I can always rerun my test and make sure that what I want still holds true. It's a very powerful way to develop your textual IF, particularly as you start to add more variations on what you do and do not allow in the contet of your model world. So here's my modified test:

**SOURCE TEXT**

```
Test inflate_action with "actions / inflate the ball / inflate the flashlight".
```

I leave it up to you as to whether you want to continue to include the `ACTIONS` command. Some people feel it gets in the way and certainly it can. One technique is to remove the `ACTIONS` command from your 'test' statement and then, if you want action tracing turned on, simply run that command manually before you execute your `TEST` command. See what works for you.

So back to the action. Basically how the source text is set up right now is that anything – or any *something* – that the protagonist attempts to inflate will be inflated. It would be nice to tell Inform that some things simply aren't inflatable. I'll get to that in one second. Before that I want to point out something about "[the noun]" that you see in the report inflating rule. What I'm doing there is applying a rule to an action whose direct object I'm not going to know in advance. Inform recognizes "the noun" as a value that can be used in any rule about actions. Think of "[the noun]" as serving as a placeholder for whatever actual object is used during the processing of the action.

I'm also bringing that huge elephant back into the room: rules. Specifically, report rules. For now, just bear with me a bit and hold off on worrying about details. I'm going to get them but I want to get a few more things in place so that I can discuss rules in a better context.

## Checking an Action

Okay, so now I want to tell Inform that only some things can be inflated. To start this process off, I'll add the following source text:

**SOURCE TEXT**

```
A thing can be inflatable.
A thing is usually not inflatable.
```

You'll probably remember from earlier in this guide that I just created a property. (Specifically an behavior or "either/or" property.)

With this text in place, you'll find you can still inflate both objects. (Start your story and run the test to confirm that.) The reason this is happening is because even though I've said things are usually not inflatable, all of that really means nothing to Inform. More specifically, there's nothing that ties the

notion of 'inflatable' (a property) to the action of 'inflating'. What I want to do here is *check* if a certain condition is valid for this particular action. Here's the source text that I'll add:

**SOURCE TEXT**

```
A check inflating rule:
   if the noun is not inflatable:
      say "You can't inflate [a noun].";
      stop the action.
```

With that in place, when you run your test, you're going to find an interesting situation. The problem is that now the protagonist can't inflate the ball either. If you have the `ACTIONS` command in your 'test' statement, you'll see that the action is said to have "failed the A check inflating rule."

**Think About It**
See if you can look at the source text and consider why this is happening. Why can't the reader have the protagonist inflate the ball at this point?

Remember that the ball is a generic thing. And I earlier made the assertion that things are *usually not inflatable*. That would include the ball. So I have to tell Inform what specific objects *are* inflatable. In this case, I'll take the simple route and just add this to the source text:

**SOURCE TEXT**

```
The ball is inflatable.
```

Note that I don't have to tell Inform which objects are *not* inflatable because I've already asserted that objects are *usually* not inflatable. I'm simply asserting here that one such object is inflatable and thus is an exception from the *usual* case.

Start the story, run the test again. Nice. So now my reader can't have the protagonist inflate a flashlight, but they can have the protagonist inflate a ball. All seems to be right with the (model) world.

But ... you might notice that the reader can have the protagonist inflate the ball more than once. That's a bit annoying. What Inform needs is a way to know if an object that is inflatable is already inflated. Further, I just found another test to add:

**SOURCE TEXT**

```
Test inflate_action with "actions / inflate the ball / inflate the ball / inflate the flashlight".
```

That test tries to inflate the ball twice. What I want to see is the first 'inflate the ball' command succeed and the second fail. Now I'm going to add the logic I need to make this happen:

**SOURCE TEXT**

```
A thing can be inflated or deflated.
A thing is usually deflated.
```

This is similar to what I did before, when I introduced the concept of "being inflatable" to Inform with the 'inflatable' property. Now I'm introducing the concept of "is inflated" or "is deflated" (another property). With that concept in place I can now put in a new check rule for the idea of something being in the inflated or deflated state:

**SOURCE TEXT**

```
A check inflating rule:
  if the noun is inflated:
    say "[The noun] is already inflated.";
    stop the action.
```

Hmm. That seems like it should have worked, right? But it didn't. If you run your test, you'll find that the reader can still have the protagonist inflate the ball twice.

**Think About It**
As before, think about what Inform is showing you here. Can you figure out why this might be happening?

The problem is that while I let the action of inflating the ball take place, I didn't do anything to tell Inform that the ball actually *was* inflated.

## Carrying Out an Action

So far I've let the action take place (by telling Inform that the ball is inflatable) and I reported on the results of that action to the reader. But in order for this to be workable I also need to tell Inform the state of the ball or, more specifically, how the state of the ball changes during the course of the action.

Let's add this next bit of source text:

**SOURCE TEXT**

```
A carry out inflating rule:
  now the noun is inflated.
```

With that in place, Inform now has some idea of when the ball is should behave like an inflated ball. What's even nicer is that all my tests now pass. Don't take my word for it. Run your own tests and make sure you're seeing what I'm seeing. Here's my output:

**STORY TEXT**

```
Learning Lab
This is the Inform 7 learning lab.

You can see a ball, an air pump and a flashlight here.

> test inflate_action
(Testing.)

> [1] actions
Actions listing on.

> [2] inflate the ball
[inflating the ball]
You inflate the ball.
[inflating the ball – succeeded]

> [3] inflate the ball
[inflating the ball]
The ball is already inflated.
[inflating the ball - failed the A check inflating rule]

> [4] inflate the flashlight
[inflating the flashlight]
You can't inflate a flashlight.
[inflating the flashlight - failed the A check inflating rule]
```

Another nice thing is that with concepts like I've put in place, is that you can customize the description of the ball for the reader. Here's an example:

**SOURCE TEXT**

```
A ball is a thing in the Learning Lab.
The description of the ball is "The ball is [if inflated]inflated[otherwise]deflated[end if]."
```

To test that this works, I'll modify my 'test' statement accordingly as such:

**SOURCE TEXT**

```
Test inflate_action with "actions / examine ball / inflate the ball / examine ball / inflate the ball / inflate the flashlight".
```

Here I'm including two `EXAMINE` commands to make sure that the initial state of the ball and the changed state of the ball are reflected.

### Digression: Testing with the Transcript

So I've introduced a lot here regarding rules but I want to digress one more time into another testing aid that's really important when you start getting into rules that affect the results of actions. This involves checking the transcript, which is basically a captured play session with your story. Any inputs and outputs are captured as part of the transcript.

**Transcript**
To see the transcript, you just click the 'Transcript' tab'.

So if you were to run the latest 'test' statement you'd find the following as your transcript:

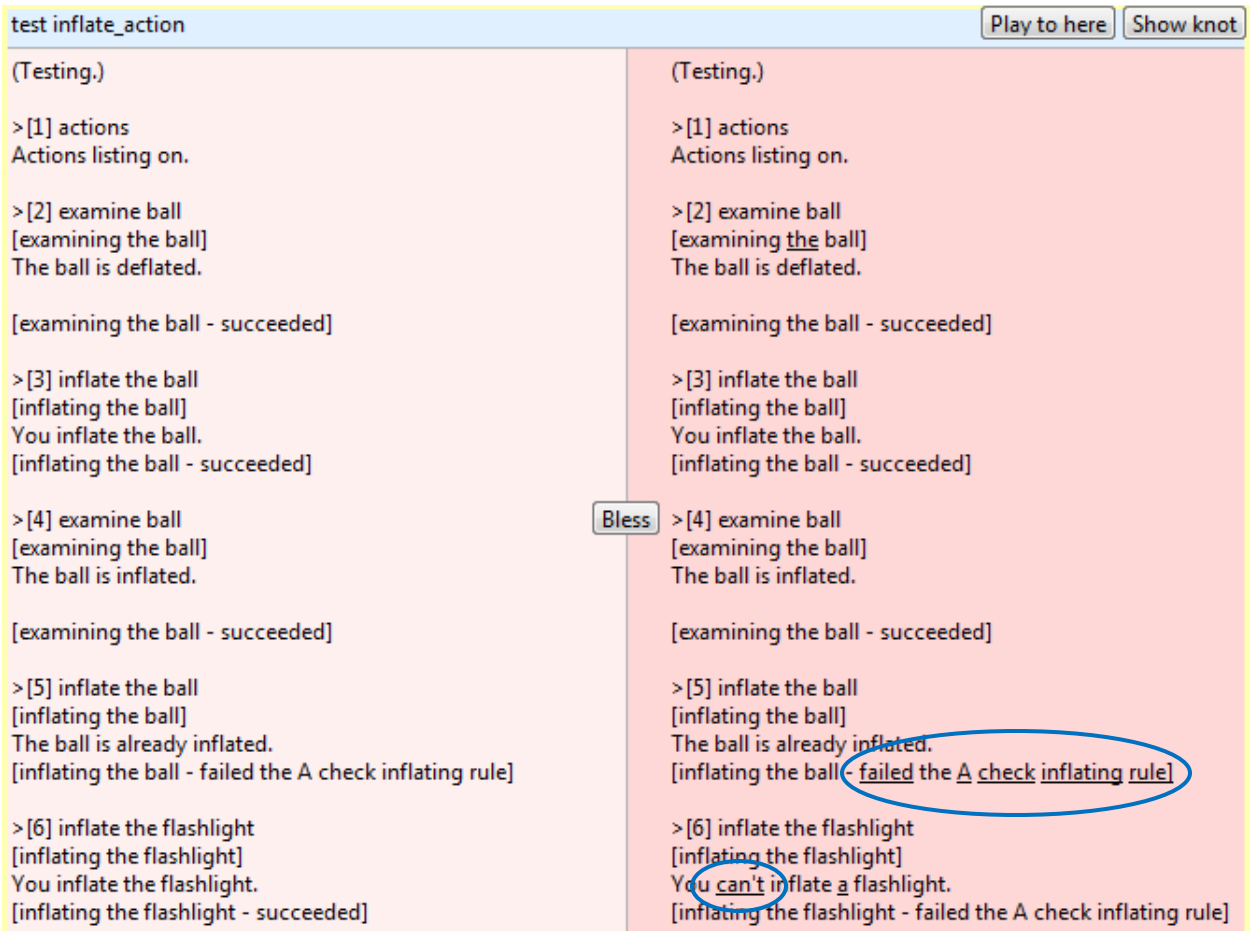

This is showing you everything that the reader entered and everything that the story responded with. The details of this transcript would, of course, change if you changed the reader inputs or the way the story responded to them. Going along with the testing theme, however, sometimes you'll want to 'capture' a transcript as being the 'right' transcript. Put another way, you'll bless that transcript as being how you expect the story to function. To do that, it probably won't come as much of a surprise that you have to click the "bless" button. Here's what your transcript will look like after that:


What I've just done there is say that this path (which is the path taken when `TEST INFLATE_ACTIONS` is used) has been 'blessed' as being what I want to have the output be.

By itself you might think "Well, so what?" What's nice is that you can use this transcript feature to see at a glance if your output has changed from what you expect. To show this I'm going to add some temporary source text that will exist to change the current logic so that the transcript will necessarily differ. The change I need is really simple:

| SOURCE TEXT |
| --- |
| The flashlight is inflatable. |

Now run your test again. Your output will be different of course but when you have a lot of output it might be hard to search through all of it. That's where the transcript can come in handy because it will showcase those differences for you. Here's an example of what I see based on that one additional line of source text:




What that's showing me is where the story output, based on my new source text, has diverged from the previously 'blessed' version. Notice that you have to treat the right-hand side as the blessed version; the left-hand side is the current output. So what I've highlighted with the blue circles is the text that was previously generated and that is no longer being generated.

At this point I could hit "bless" again and that would indicate to Inform that I'm accepting this new output. However, in this case, that's not what I want to do. What Inform has helped me do is isolate that a certain change I've made has led to an effect that I don't want. Granted, in this case it was pretty obvious what the consequences would be when I added my assertion regarding the flashlight. That said, when you have a lot of source text or complicated concepts and model world interaction, it's very easy to make what seems to be a "simple" change that has wider ramifications.

To clean up from this example, simply remove that extra line of source I had you add. And once you do that, rerun your test. You'll find that the transcript is back to its blessed version.

## The Basis for Rules

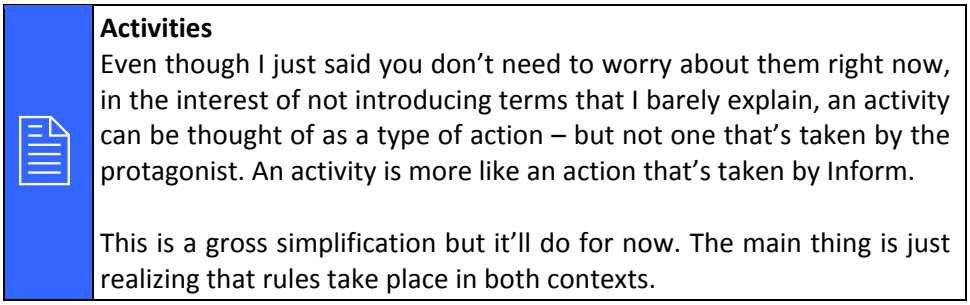
Going back to the conceptual points list at the start of this section, what I've just covered above is speaking to the fourth item on that list: **establish how the world model will respond to actions**. This

was done via those Report, Check, and Carry Out rules you just put in place. Before moving on, let's cover a little about the rules I've been talking about.

In the *Writing with Inform* manual, at least at the time of writing this guide, 18.1 says this:

> "When we open the casing and look inside the machinery of Inform, what we see are rules and rulebooks. We seldom need to know how this machinery works..."

That second sentence is demonstrably false. You pretty much *always* need to know how the "machinery" works because Inform often forces you to use various rules, either in the context of actions or the context of activities, the latter of which is nothing you need to worry about right now.




A **rulebook** is a collection of rules. Those rules are listed in a specific sequence within the rulebook and the sequence does matter. When a rulebook is being "followed" or "processed", its internal sequence of rules are being verified to see if they should execute or not. This happens until one of the rules stops the verification.

There's a *lot* of detail behind what I just said.

For example, how does a rulebook know when to process? What's the meaning of "verification?" What does it mean to stop verification? What happens when a rule executes? Why break rules into rulebooks in the first place? And what exactly is a "rule" anyway?

Well, I'm going to go over the answers to all of those questions. But before I do that, I want to talk about the rules you've already used in context so that you have some basis for the further discussion, which can get a little complicated.

Earlier in this guide, I said two things that I'll now be able to provide some better context for. Those two things were:

- Rules use descriptions to provide the circumstances (or the context) for when a given set of changes should be made to the model world.

- Actions are run through rulebooks.

What I've had you concentrating on in this example is an 'inflating' action. That action is run through a series of rulebooks.

Let's try this – modify your test as such:

**SOURCE TEXT**

```
Test inflate_action with "actions / rules / examine ball / inflate the ball / examine ball / inflate the ball / inflate the flashlight".
```

As with ACTIONS, which you've already been using, RULES is another testing command. This command will show you what rulebooks are being processed as an action is taking place. It may not always be clear what rulebook is necessarily being applied because the output of this comand will focus on each and every rule that is being processed.

Go ahead and run your test and look at the output. From that output, here's what matters for this discussion:

- [Rule "A check inflating rule" applies.]
- [Rule "A carry out inflating rule" applies.]
- [Rule "A report inflating rule" applies.]

This is showing you that when the player types an 'inflate' command, that command is translated into an 'inflating' action, which is run through a series of rules called check, carry out, and report.

*Every* action – whether built in to Inform or that you create – has its own Check, Carry Out and Report rulebooks.

What's important to realize here is that action processing takes place for any action, regardless of the command used to call that action. For example, at this point, now that I know my basic action is working, I can also add more grammar possibilities to map commands to this one action. Add the following to your source text:

**SOURCE TEXT**

```
Understand "blow up [something]" and "blow [something] up" as inflating.
```

Here I've mapped another command (actually, two commands) to the 'inflating' action. You could create a new test to try this out or you could modify your existing one as such:

**SOURCE TEXT**

```
Test inflate_action with "actions / rules / examine ball / blow up the ball / examine ball / blow the ball up / inflate the flashlight".
```

Notice how in that test I cover both variations of the possible grammar for the command. If you now run your test you should see the same output. Although note that if you are comparing with the 'blessed' version, you will see differences in the input. However, the crucial element – the output – has been preserved. (It's up to you which version of the command you want to keep in your test.)

The other thing to notice, however, is that in both cases of the test – whether you used the command `INFLATE THE BALL` or `BLOW UP THE BALL` – the action that was processed was always 'inflating the ball'. And in both cases that action was run through the Check, Carry Out, and Report rulebooks. In fact, notice that you have two Check rules. The important bit of the output is really that first 'inflate' (or 'blow up') command:

**STORY TEXT**

```
> [4] inflate the ball
[inflating the ball]
[Rule "A check inflating rule" applies.]
[Rule "A check inflating rule" applies.]
[Rule "A carry out inflating rule" applies.]
[Rule "A report inflating rule" applies.]
You inflate the ball.
[inflating the ball - succeeded]
```

You'll see that your two check rules are found to apply to the situation. Why? Because they are 'check *inflating*' rules and thus apply when an 'inflating' action is being attempted. The same applies to the carry out and report rules.

I want to keep the ratio of discussion to practice somewhat consistent, so with that bit of basis understood, let's move on to doing some more things with the rules you already have in place.

## Action Processing with Rules

You've already seen that a given action gets processed through rules. Those rules make decisions regarding how the action will process. A perfect example of this are the check rules that you put in place. There are currently two ways that action processing for the 'inflating' action can stop:

- if the noun is not inflatable
- if the noun is inflated

Both of those conditions are handled (checked for) in your two check rules. If either condition is found to be true, the rule uses a 'stop the action' statement to do just that: stop the action in its tracks. In fact, you can see that in the second attempt to inflate the ball:

**STORY TEXT**

```
> [6] inflate the ball
[inflating the ball]
[Rule "A check inflating rule" applies.]
[Rule "A check inflating rule" applies.]
The ball is already inflated.
[inflating the ball - failed the A check inflating rule]
```

Here you should notice that the check rules were run but, unlike the first time the command was execute, the carry out and report commands were not run. In fact, this is a good example of how you can use the RULES command to see what is actually happening during action processing.

The point to get out of this was that check rules managed to stop an action before all of the rules processed. Knowing that, now maybe other things might occur to you to check for in terms of what actions the reader attempts or, in this case, what variations on the 'inflating' action may be attempted. For example, the reader might have the protagonist try to inflate the air pump which (presumably) doesn't make sense. So let's add a test for that.

**SOURCE TEXT**

```
Test inflate_action with "actions / rules / examine ball / inflate the ball / examine ball / inflate the ball / inflate the flashlight / inflate the air pump".
```

Notice how, once again, I wrote the test first, in anticipation of the change I was going to make. In this case, since the air pump is not asserted to be inflatable in the source text, the message the reader gets is technically accurate. But in the case where a reader actually has the protagonist try to inflate *this* particular object, maybe the goal is to provide a different message.

So let's put a check rule in place for that as well:

**SOURCE TEXT**

```
A check inflating rule:
    if the noun is the air pump:
        say "That's clearly not going to work.";
        stop the action.
```

Now, there's something to notice here when I run my test. Even though I have this new check rule in place, I still get the same output. More specifically, I expected to get "That's clearly not going to work" (from my new Check rule) but instead I got "You can't inflate an air pump" (from my previous Check rule). The reason for this is because of something I said earlier about rules and rulebooks: "rules are listed in a specific sequence within the rulebook and the sequence does matter."

In this case, assuming you've been following along with all this and pretty much typing all new source text one after the other, the order of your check rules is like this:

**SOURCE TEXT**

```
...
A check inflating rule:
    if the noun is not inflatable:
        say "You can't inflate [a noun].";
        stop the action.
...
A check inflating rule:
    if the noun is inflated:
        say "[The noun] is already inflated.";
        stop the action.
...
A check inflating rule:
    if the noun is the air pump:
        say "That's clearly not going to work.";
        stop the action.
```

**Think About It**
Consider the output you are getting from the `INFLATE THE AIR PUMP` command. Then consider what output you expected. Now look at the rules above and, with the notion that sequence matters, see if you can figure out why you're not getting the "That's clearly not going to work" message.

The reason for this problem is the position of the checks. I didn't indicate that the air pump is an inflatable object. So when the checks occur the first one that fails for the air pump is the first check ('if the noun is not inflatable'). If I want to capture the specific event of not being able to inflate the air pump, that check has to go first in your source text.

You might think that this is a huge pain since, as your source text gets bigger, it's going to be hard to keep track of everything. I would agree so let's step back here for a bit. Let's look at our source text.

It's a bit messy, right?

There's stuff all over the place and this is one of the things to be watchful for with Inform. Personally I find that the way I was able to write all this shows a very powerful feature of Inform. I say that because, as you've seen, I could just keep taking you through this guide without giving you too many positional cues. But it also means you can write very, very sloppy source text. So "refactoring" is something that should be encouraged. Refactoring is a concept very popular in the software development world but, fear not, you don't have to learn ever nuance about it.

**Refactoring in Inform**
Refactoring is the process of changing your source text in such a way that it does not alter the external behavior of the source (i.e., how the story plays) yet improves its internal structure.

As an example, I'd like to get my check rules a bit closer together so that I can better tell at a glance what I'm checking for. Clearly I could just put each rule next to each other in the source so that I can better determine when I have to change their position. Here's what that would look like:

**SOURCE TEXT**

```
...
A check inflating rule:
    if the noun is not inflatable:
        say "You can't inflate [a noun].";
        stop the action.

A check inflating rule:
    if the noun is inflated:
        say "[The noun] is already inflated.";
        stop the action.

A check inflating rule:
    if the noun is the air pump:
        say "That's clearly not going to work.";
        stop the action.
...
```

The only difference between this and what I previously showed you is that there is no ellipses between the check rules, implying they are all one after the other in the source. However, I'm actually going to go a step beyond that by putting all the checks in one block, or chunk, of source text. Here's what that looks like (and you should do the same thing):

**SOURCE TEXT**

```
A check inflating rule:
    if the noun is not inflatable:
        say "You can't inflate [a noun].";
        stop the action;
    if the noun is inflated:
        say "[The noun] is already inflated.";
        stop the action;
    if the noun is the air pump:
        say "That's clearly not going to work.";
        stop the action.
```

This is a little nicer. You'll notice that I just combined the conditions under one general 'a check inflating rule' statement. What's important to note is that each 'stop the action' statement, except for the last one, must have a semicolon rather than the period they had before.

**Look Closely.**
Look at the two source code sections on this page and make sure you understand how the first became the second.

After every such clean up like this, it certainly doesn't hurt to re-run your tests and make sure you are still getting what you expect. Also, as a testing note, these are times when you might want to consider removing the ACTIONS and RULES from your test statements. Those are handy when you are in debugging mode, checking what's happening in the Inform internals as your story is being played. They tend to provide clutter, however, when you're trying to get a good transcript or trying to see the effects of refactoring like I just did with the check rules.

As long as we're in clean up mode, notice how I have all those 'stop the action' statements? Those are fine and they work. You can also tighten up the code a bit by placing an 'instead' clause at the end of each say phrase. Here's how that looks:

**SOURCE TEXT**

```
A check inflating rule:
    if the noun is not inflatable:
        say "You can't inflate [a noun]." instead;
    if the noun is inflated:
        say "[The noun] is already inflated." instead;
    if the noun is the air pump:
        say "That's clearly not going to work." instead.
```

It's really up to you what format you prefer. What you should choose is something that you believe makes the intention of your source text clear, not just what "tightens up" your text. However, I do want to point out one thing which is that you can see I removed the 'stop the action' statement and put the 'instead' clause on the end of the say statement. It's critically important that the semicolons and periods are put in place correctly. This can be a common source of problems with your source text in Inform.

**Try It!**
You might want to try to remove one of the semicolons or colons just to see what happens when you try to run the story.

Finally, notice how I have been calling this 'A check inflating rule'. I can shorten that a bit:

**SOURCE TEXT**

```
Check inflating:
    if the noun is not inflatable:
    ...
```

Again, what you prefer is really up to you and how you want to structure your source text. That being said, you might be wondering how 'A check inflating rule' can be the same thing as 'Check inflating'. It has to do with the concept of a declaration.

## Rule Declarations

A **declaration** in Inform is anything that is prefaced with some specific declaratory text (that Inform calls a preamble) and a colon. The initial wording of the preamble can determine what type of declaration is being dealt with.

A **rule declaration** is used to create a new rule. With such declarations the preamble can be just a name (which must end with the word 'rule'), just a circumstance, or a circumstance along with a name. Here are some examples of each of these.

- "Just a name" rule declaration:
  - This is the fancy examining rule: ...
  - A carry out looking rule: ...
- "Just a circumstance" rule declaration:
  - Check an actor waving something carryable that is not held by the actor: ...
- "Circumstance and name" rule declaration:
  - Check an actor waving something carryable that is not held by the actor (this is the taking before shaking rule): ...

One thing that's important here is that in these rule declarations, the name of the rule must be in place. In the first case, the name of the rule is "fancy examining." That rule has a very specific name. The other one is a "carry out" rule – which is a general rule – and the specific context for that rule is "looking." In the second case, notice that the rule is a "check" rule, with the rest of that declaration being the circumstance. The third case is similar except that the rule is given a specific name: "taking before shaking." This rule is placed into the "Check" rulebook.

The circumstance is really the context in which the rule would be considered. So far our circumstance in our source text was pretty simple: inflating. When that context is found to be operative – meaning, when an 'inflating' action is being processed by Inform – then any rules in the check rulebook are processed. As you've seen this is the case whether those rules are all separated or brought together. Behind the scenes, to Inform, all of those 'check inflating' rules were put into one Check inflating rulebook.

# Working Example, Continued

The previous section was a really long section! You covered a lot of material there. So let's take a moment to just recap a bit before plowing ahead. What you did was create a new command+action mapping for the reader to use as part of how they communicate with the protagonist. To do this, you needed to tell Inform four things:

1. What inputs – i.e., verb commands – from the reader to understand as referring to that action ('inflate', 'blow up').
2. How to refer to the action within the story logic ('inflating').
3. What to do when the command is received and the action is triggered ('check', 'carry out').
4. What to tell the reader about what happened ('report').

When Inform receives a command and translates it into an action, it first runs that action through a check rulebook that's specific to that action. You'll need to write a check rule for each new action that you create. In any given check rule, you're checking whether the protagonist is trying to do something that makes sense. If it doesn't, you stop the action. If the action does make sense, Inform will go on to a carry out rulebook that's specific to that action. In the carry out rule, Inform actually does the action, which changes one or more objects in the story – in whatever way you define. After the carry out rule, you can add a report rule, which will indicate what happened to the reader.

## Rule Processing in the Model World

The previous section dealt with a lot of talk about actions and eventually segued that into rules. What that probably (and hopefully!) showed you is that the notion of rules and actions are tied very closely together. You've also seen how Inform has a notion of built-in actions ('taking', 'dropping') and a built-in notion of certain types of rule ('check', 'carry out'). Those actions and rules tie in with each other. Inform also has what are sometimes referred to as "basic rules of realism." These provide what you might call the basis for logic in any model world. Some examples of what this means:

- The protagonist can't pick up something that they already have.
- The protagonist can't drop something they don't have.
- An object can't be placed inside of its own self.

What this means is that Inform is providing you with a sort of base level of logic for your model world. You have to specifically add logic for situational elements like these:

- The protgaonist can't pick up something *because it's heavy*.
- The protagonist can't dop something they have *because it's stuck to their hand*.
- The protagonist can't place one object inside another *because the second object is too small to contain the first*.

You can see by that second list that as an author you will be filling out the vanilla model world that Inform provides you with specific circumstances. As you probably remember from the last section, rules are all about circumstances: and those circumstances are brought about via actions.

So let's start with a quick recap that focuses on rules.

Every action is run through rulebooks.

Rulebooks contain rules.

Rules are just statements about specific things that need to be done.

A rulebook contains a series of related rules.

(The "specific things" in the third point above are in reference to how Inform operates.)

With that, let's go back to actions. There are generic ways that particular actions should behave. For example, consider the 'taking' action. When this action happens Inform **checks** certain model world logic (or "realism"), such that the protagonist can see and touch the object in question. Inform then **carries out** the action, which usually just means putting the object in the possession of the protagonist. Inform then **reports** on the results of the action which, in the case of 'taking' usually just means a simple message.

So with that said, every built-in action that Inform provides (or any action that you set up on your own) will automatically establish three rulebooks that you can use for that action: Check, Carry Out, Report. As such, these rulebooks are meant to represent normal behavior. Those are what represent that generic behavior that I mentioned. That's a really important point.

**Normal Behavior**
Check, Carry Out, and Report rules are used to handle the normal behavior that should occur as the result of an action.

So what every action provides are Check rules, to see if the action makes sense – for instance, to see that the reader is not trying to have the protagonist take their own body, or a whole room, or something they already have in their possession. Then Carry Out rules, to actually do what the action is supposed to do. Finally Report rules, which tell the reader what has happened.

So now let's go back to all that stuff I said earlier but use this added context to (hopefully) understand it.

Every action (**taking**) is run through rulebooks (**check taking**, **carry out taking**, **report taking**).

Rulebooks contain rules.
Rules are just statements about specific things that need to be done.

These rules (within the rulebooks) will check for why an action may not happen, how to carry out the action, and what to report about the action.

A rulebook contains a series of related rules. So all 'check taking' rules are in the same rulebook and similarly for the other such rules.

So with the example created so far, there are three objects you can take. Anytime you issue a 'take' command (and thus a 'taking' action), the check taking / carry out taking / report taking rulebooks are processed, with the specific object (ball, air pump or flashlight) being the subject of consideration by those rules.

### Before and After Rule Processing

I mentioned how those check / carry out / report rulebooks handle the usual, generic situations. What about the *unusual* situations? What about *unexpected* events? There are three other rulebooks you can use for actions to handle those kinds of things.

Probably the easiest thing to consider is when you want the rule processing to happen as normal – but you want something else to happen just before all that happens. There is a Before rulebook and you can use that. Let's first a minimal level test:

**SOURCE TEXT**

```
...
Test inflate-rules with "rules / inflate the ball".
...
```

Then add the following rule:

**SOURCE TEXT**

```
Before inflating the ball:
    say "This happens before the normal behavior."
```

Now run `TEST INFLATE-RULES`. What happens here is that before any normal rule books are considered, the Before rules are processed.

Another easy thing to consider is when you want the rule processing to happen as normal – but you want something else to happen *after* all of that happens. There is an After rulebook and you can use that. Add the following source text:

**SOURCE TEXT**

```
After inflating the ball:
    say "This happens after the normal behavior."
```

Run the test to see what you get.

**Read It!**
Make sure to look at the differences in output from the RULES testing command.

| With Before Rule | With After Rule |
|---|---|
| >[2] inflate the ball<br>[Rule "Before inflating the ball" applies.]<br>This happens before the normal behavior.<br><br>[Rule "Check inflating" applies.]<br>[Rule "A carry out inflating rule" applies.]<br>[Rule "A report inflating rule" applies.]<br>You inflate the ball. | >[2] inflate the ball<br>[Rule "Before inflating the ball" applies.]<br>This happens before the normal behavior.<br><br>[Rule "Check inflating" applies.]<br>[Rule "A carry out inflating rule" applies.]<br>[Rule "After inflating the ball" applies.]<br>This happens after the normal behavior. |

The main thing to note here is that with the after rule in place, the default report rule for the 'inflating' action has been overridden.

An After rule takes place after the Carry Out rules have been processed but before the Report rules have been processed.

An after rule automatically ends the action in success. The reason for this is that action has already been carried out (since the Carry Out rules were processed) and so, to Inform, that means the action was successful. Let's say you wanted the Report rules to be processed even with the After rule in place. In that case, you could force the rule to continue. Make a slight modification to your After rule:

**SOURCE TEXT**

```
After inflating the ball:
    say "This happens after the normal behavior.";
    continue the action.
```

Run your test to see the output.

This, however, brings up an important point. Rulebooks will stop or continue based on certain conditions. I'm not going to delve into that too much right now simply because there's a lot of context to this and it depends on what rulebooks you are considering.

At this point you've seen that an After rulebook, by default, stops an action. You've also seen that you can also force the action to continue. What about the Before rules? You saw that those automatically keep going. A Before rule does not automatically stop the action in its tracks. But what if you wanted to stop the action? One way is to modify your Before rule as such:

**SOURCE TEXT**

```
Before inflating the ball:
    say "This happens before the normal behavior.";
    stop the action.
```

Here's another way you could do the same thing:

**SOURCE TEXT**

```
Before inflating the ball:
    say "This happens before the normal behavior." instead.
```

This should be familiar to you from when I had you write the Check rules. That latter format, however, brings up a good segue into yet another rulebook.

### Instead Rule Processing

In the case of the last two versions of the Before rule above, the idea is clearly to subvert the action and not allow it to continue. That's probably a better usage of the Instead rulebook.

An Instead rule takes place after the Check rulebooks have been processed but before the Carry Out rulebooks.

What that means is that an Instead rule can only take effect if the action has already passed the basic reasonability (realism) tests that are provided for in Check rulebooks.

So, for example, if it was *impossible* for the protagonist to inflate the ball – whether they wanted to or not – it wouldn't make sense to say they refuse to inflate it. (Since, whether they refuse or not, it's impossible for them to do so.) A Check procedure would ferret that out first. But if it's not impossible for the action to take place – i.e., it does not violate logic or reality – then the Instead rule can subvert the action.

Add the following to your source:

**SOURCE TEXT**

```
Instead of inflating the ball:
    say "This happens instead of the normal behavior."
```

If you run your test, you'll see that the Instead rule stops the action. Note that the above worked because no Check rules intervened before the ball was attempted to be inflated. If a Check rule already stopped the action (perhaps because the ball was already inflated), then the Instead rule would have never gotten a chance to process.

Beyond stopping the rule processing, an Instead rule treats an action as a failure. It's a "failure" because the original action on the reader's part has been subverted and something else happened instead. The

notion of “success” and “failure” is not quite as clear cut as the terms may make it sound but for now just realize that “failure” doesn’t necessarily mean something bad happened.

### Rule Processing Summary

Okay, so let’s do a little recapping here.

An action is ordinarily handled by running it through Inform’s rulebooks of what might be called normal behavior for those circumstances. Those are the Check / Carry Out / Report rules. But it’s possible that different (or even abnormal) behavior should take place in certain circumstances. That’s where the Before / After / Instead rules come into play.

I used the term “circumstances” a couple of times and that’s important because that’s essentially what a rule is: a set of circumstances followed by a list of instructions. When those circumstances apply, the instructions are carried out. Let’s break this down a little:

- Before rules intended so that preliminary activities can happen before the action is tried. Normally action processing will continue.

- Check rules are intended to provide instructions that make sure the action makes sense in terms of basic “laws” of logic and realism.

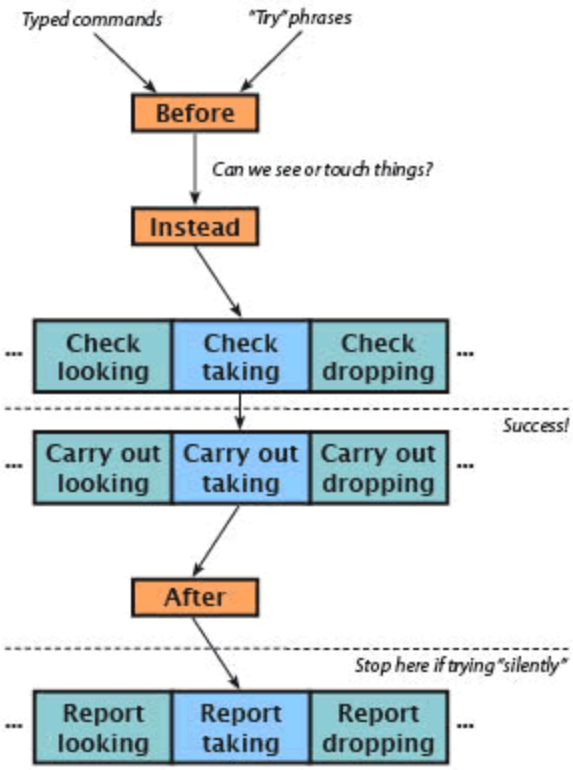
- Instead rules are intended for when you want to block or divert the action, perhaps causing something different to happen. Normally action processing will stop.

- Carry Out rules are intended to provide instructions that tell Inform exactly what effect the action has on the state of the model world.

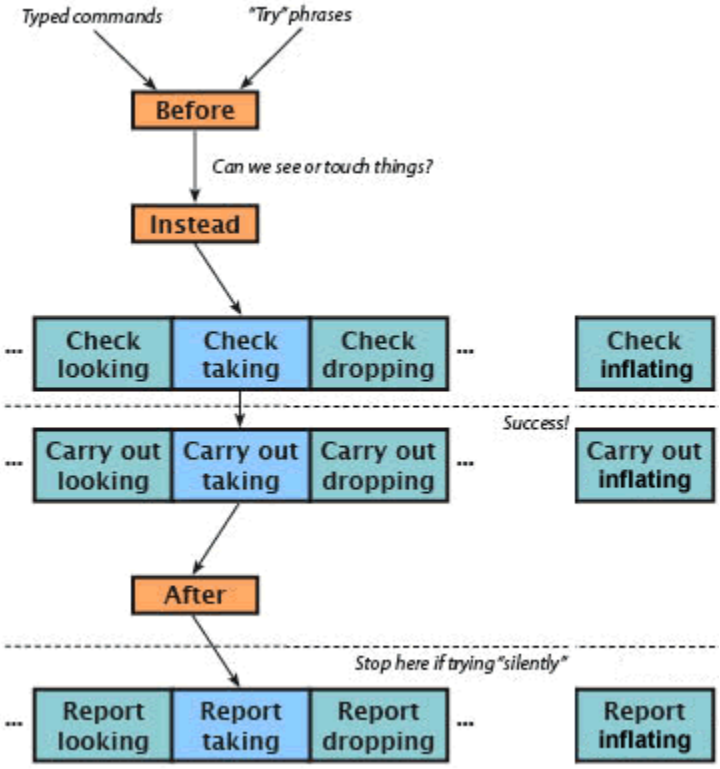
- After rules are intended to allow you to indicate unexpected consequences after the action has taken place. Normally action processing will stop.

- Report rules are intended to provide instructions to tell Inform exactly how it should report the effects of the action to the reader.

So let’s consider this graphically for a bit. In the *Writing with Inform* manual (currently at 12.2), you’ll see the following diagram:




As part of our source text, we created an 'inflating' action. That means you can consider this new figure:



In other words, what you've done by adding in the 'inflating' action is add a new "column" of rulebooks for that action. And, if you followed along with my examples, you've added rules to those rulebooks. Those rules told the story how to act when the 'inflating' action was encountered. Sometimes the story allowed the action, sometimes not.

## Digging Into Rules

I want to quote something I said earlier in this guide. I had said:

> *A **rulebook** is a collection of rules. Those rules are listed in a specific sequence within the rulebook and the sequence does matter. When a rulebook is being "followed" or "processed", its internal sequence of rules are being verified to see if they should execute or not. This happens until one of the rules stops the verification.*

I then indicated that there was a lot of detail behind this statement and it virtually demanded that I answer questions like these:

- How does a rulebook know when to process?
- What's the meaning of "verification?"
- What does it mean to stop verification?
- What happens when a rule executes?
- Why break rules into rulebooks in the first place?
- And what exactly is a "rule" anyway?

So now I'll turn to answering these with all of the previous context in place.

**How does a rulebook know when to process?**

In a very real sense, rulebooks process all the time when Inform is presenting a story to the reader. The rules that are placed into rulebooks are the basis of Inform. You already know that what happens is that Inform reads a command from the reader. That command is parsed into an action. When that happens, two specific rulebooks are consulted:

- The "action processing" rulebook.
- The "turn sequence" rulebook.

There is a lot of detail that happens when these rulebooks are processed. You've already had a good introduction to the "action processing" side of things. That's essentially what you saw happening from the time a command was typed in to the time the story responded back with what happened.

Right now I just want you to understand that these two rulebooks are operating continuously and the context in which they are, which is that of an action occurring. So the action-processing rulebook is always run for an action. That rulebook, in turn, passes control to other rulebooks. Those other rulebooks are more specific in nature. Here the "more specific" is usually in reference to a specific action. So, thinking back, you wrote a series of rules for 'inflating'.

For now, just understand that rulebooks always process because of two things:

1. Inform is event (action) driven.
2. The action-processing rulebook is always processing as part of Inform's normal operation.

**What's the meaning of "verification"?**

To recap a bit, I had said earlier that when a rulebook is being "followed" or "processed", the internal sequence of rules in that rulebook are being verified to see if they should execute or not. That verification is based on how the rulebook is processed and really ties in to what I said about the previous question, regarding more specific processing.

The action processing that takes place goes through two phases: a preliminary (general) phase and a secondary (specific) phase.

The preliminary phase consists of seeing whether the action can even be attempted. For example, perhaps the situation the player is in forbids the action from being tried. In that case, there's no point in trying to process it. One such preliminary is checking whether there are any basic reasons of physical realism why it could not be tried. Another preliminary check is whether the action would require some other character to take some action and whether they agree to. A final preliminary check is whether there is any rule that exists that is set to intervene in the case of this particular action.

Let's say all of these preliminaries are handled and nothing is found to intervene in terms of allowing the action processing to continue. Call what just happened here **General Action Processing**. In the case that General Action Processing encounters no issues, then Inform jumps down to what I'll call **Specific Action Processing**.

The point of all is that while these two phases are going on, Inform is attempting to verify what should happen as a result of the action that it is processing. Inform is using its rulebooks to do just that. This is done by running through all of the rules in the relevant rulebooks and seeing if any stop verification. That leads right to the next question.

**What does it mean to stop verification?**

One thing I didn't mention in the answer to the above question is that verification is really checking what I like call the **operational context**. The operational context is the action and the circumstance in which the action takes place. The operational context is sort of like a pattern and the rulebooks have rules that are designed to look for certain patterns. When that pattern is matched, the verification process stops and the rule itself is executed.

So "stopping verification" just means that some rule in one of the rulebooks has been determined to fit the operational context. It matches a certain pattern. This means the rule is "executed."

One thing I should note here is that inside a given rulebook, the rules are not listed in the order in which they were created. Rules are listed in a specificity order. So a given rule (taking a fish) comes before

another rule (taking something) because it applies in more specific circumstances. In this case, taking a more specific object (a fish) rather than just taking any objec t (something). This is one the main operating concepts behind Inform's rules-based mechanism: a rulebook gathers together rules about making some decision, or taking some action, and sorts them in order to give the more specific rules first choice about whether that rule should be applied to the action. If that rule is not applied to the action, then the execution of the rulebook continues – i.e., verification continues – down the list of rules until a rule is found that should be applied. When that rule should be applied, it is executed, leading to the next question.

**What happens when a rule executes?**

The point of a rule is to make a decision. Whether or not the rule actually gets a chance to do so is part of the verification, i.e., whether or not the rule is deemed to match a pattern. The execution is a separate part. I mentioned earlier that every action that Inform processes will either succeed or fail. I can now amend that to say that every action that Inform processes in the context of a rule will either succeed or fail.

In the most specific terms, a rule is made up of phrases. These phrases are operational elements that tell Inform exactly what to do. For example, some phrases may just instruct Inform to print some text. Other phrases may ask Inform to change the state of the model world (such as unlocking a treasure chest, closing off a previously accessible region, or increasing the score of the player). So when a rule is executed, the phrases of the rule are executed. It's the phrases that actually do the work.

**Why break rules into rulebooks in the first place?**

Rulebooks are composed of sets of rules that are probably more or less related in some sense, meaning their overall affect on the model world (should they be executed) would be similar. In reality, this breakdown is somewhat of a convenience. There's no strict reason why all of the rules in a given rulebook couldn't just be placed into one gigantic rule that covered all of the situations.

However, consider what I said earlier about rules being used to match a pattern. That implies that you might have rules that have vastly different patterns and only slightly different patterns. To some extent, this might suggest a division of rulebooks. But it might also suggest a division within rulebooks and that's one valid reason for the rulebook + rule scheme. Also, while I haven't talked about this yet, Inform does let you intervene in the rule processing, including removing or substituting rules, and in those cases the more rules you have, the more points you have where you can modify how Inform operates.

Now that you know a little bit of the basis, we can consider some very specific rulebooks and how and when they operate. First it might help to understand that Inform likes to make a distinction between the **general implementation of an action** and the **situational implementation of an action**. This distinction breaks down into the six rulebooks you've already seen:

- Situational Implementation: Before, Instead, After
- General Implementation: Check, Carry Out, Report