- Constants cannot be changed after they are initialized and must be initialized when declared

Syntax:

```csharp
private const NAME: type = value;
```
Casting

There is also a more Java-like syntax
(int)(num), but the former way is better for several subtle reasons

Syntax:

```plaintext
var num:Number = 23.5;
var newNum:int = num as int;
```

More: [ragona.com/blog/as3-casting-vs-typevar/](http://ragona.com/blog/as3-casting-vs-typevar/)
Default Parameters

Syntax:

```java
public function name(param1:type = value, ...) {};
```

- If you make default a parameter, all parameters thereafter must also be defaulted.
- The only way to give a value to the nth defaulted parameter is to give a value to all the preceding parameters as well.

More: Default Parameters
The stage is an object that stores all of the items that will be displayed on the screen.

When you run your program, your class is instantiated and added as the first child of the stage object.

The stage object can be referenced by any DisplayObject that is on the display list.

The stage contains some very useful fields that we can alter.

Some of these fields are the same ones we are setting in the SWF command, but there are many others that we can set manually

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stage.stageHeight</td>
<td>Height of the flash movie</td>
</tr>
<tr>
<td>stage.stageWidth</td>
<td>Width of the flash movie</td>
</tr>
<tr>
<td>stage.frameRate</td>
<td>Frame rate of the flash movie</td>
</tr>
<tr>
<td>stage.scaleMode</td>
<td>How the movie reacts when resized</td>
</tr>
<tr>
<td>stage.quality</td>
<td>The quality setting of the movie</td>
</tr>
</tbody>
</table>

DisplayObjectContainer

- An object that can hold things as children for displaying on the screen

- Examples: Stage, Sprite, MovieClip

DisplayObject

- An object that can be displayed on the screen
- Anything inheriting from this can be drawn onto the screen if added to a DisplayObjectContainer

- Examples: Shape, TextField, Sprite

Display List

Stage

Sprite(this)

Children
The display list is simply a list of the things that will be displayed on the screen.

Using the `addChild()` is one way to add something to the display list.

When children are added to the same `DisplayObjectContainer`, they are automatically assigned depths starting at 0.

The order in which things are drawn on screen corresponds to the depths of their parents.

An instance of your class is the first child of stage with a depth of 0.

These functions allow us to manipulate the order in which things are drawn to the screen by manipulating the depth of children on `DisplayObjectContainer`'s.

<table>
<thead>
<tr>
<th>Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cont.addChildAt(obj:DisplayObject, depth:int)</code></td>
<td>Add a child at the given depth between 0 and <code>cont.numChildren</code></td>
</tr>
<tr>
<td><code>cont.setChildIndex(obj:DisplayObject, depth:int)</code></td>
<td>Set the depth of a child anywhere from 0 to <code>cont.numChildren</code> exclusive</td>
</tr>
<tr>
<td><code>cont.getChildIndex(obj:DisplayObject)</code></td>
<td>Returns the depth of the <code>DisplayObject</code></td>
</tr>
<tr>
<td><code>cont.swapChildrenAt(depth1:int, depth2:int)</code></td>
<td>Swaps the children at <code>depth1</code> and <code>depth2</code></td>
</tr>
<tr>
<td><code>cont.removeChild(obj:DisplayObject)</code></td>
<td>Removes <code>obj</code> from <code>cont</code> and shifts deeper children down</td>
</tr>
</tbody>
</table>
Syntax:

```
import flash.geom.Point;
...
var name:Point = new Point(x:Number, y:Number);
```

<table>
<thead>
<tr>
<th>Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point.distance(p1:Point, p2:Point);</td>
<td>Returns the distance between the two points</td>
</tr>
<tr>
<td>Point.polar(len:Number, angle:Number);</td>
<td>Returns a Point with cartesian coordinates, where angle is in radians</td>
</tr>
<tr>
<td>p1.add(p2:Point);</td>
<td>Add the coordinates of p2 to p1</td>
</tr>
<tr>
<td>p1.offset(dx:Number, dy:Number);</td>
<td>Offset p1 by dx and dy</td>
</tr>
</tbody>
</table>

### Syntax:

```javascript
import flash.utils.Timer;
...
var **name:**Timer = new Timer(delay:Number, repeats:int = 0);
```

<table>
<thead>
<tr>
<th><strong>Functions</strong></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>timer.start();</code></td>
<td>Starts the timer</td>
</tr>
<tr>
<td><code>timer.stop();</code></td>
<td>Stops the timer</td>
</tr>
<tr>
<td><code>timer.reset();</code></td>
<td>Resets the timer</td>
</tr>
<tr>
<td><code>timer.addEventListener(type:String, listener:Function);</code></td>
<td>Registers timer for an event</td>
</tr>
</tbody>
</table>

Events are flags that are raised by certain objects for different purposes.

In many cases we want something to happen when this event occurs, some such events may include:
- A button being clicked
- The user pressing a key on the keyboard
- A picture having finished loading
- A timer going off

When we use `addEventListener` we are saying that we wish for the listener function to be executed when the given type of event occurs.
The TimerEvent.TIMER event happens when the timer reaches the time delay it was given.

The TimerEvent.TIMER_COMPLETE event happens when the timer has wrung the number of times specified.

Syntax:

```actionscript
import flash.events.TimerEvent.*;

timer.addEventListener(TimerEvent.TIMER, listener);
timer.addEventListener(TimerEvent.TIMER_COMPLETE, listener2);
```

Listener Functions

- The name of the listener function must exactly match the function name given to `addEventListener()`.
- The listener function must take a single parameter of the same type as that specified when you used `addEventListener()`.
- The parameter contains valuable information about the event that one might want to use.

Syntax:

```
public function listener(ev:Type):void {
    //do stuff
}
```
Events are essential to any interactive flash movies

We can use a Timer to animate things rather easily by listening for the `TimerEvent.TIMER` event

There are also events for mouse clicks and keyboard strokes that can used to create interactive games

Many more events are also at your disposal
Syntax:

import flash.display.Loader;
import flash.display.Bitmap;
...
var loader:Loader = new Loader();
loader.contentLoaderInfo.addEventListener(Event.COMPLETE, loaded);
loader.load("myimage.jpg");
...
public function loaded(e:Event):void {
    var img:Bitmap = new Bitmap(e.target.content.bitmapData);
    addChild(img);
}
Using the `embed` command is a shortcut for loading a resource and responding to its load complete event.

This command must be placed in the fields area of your class.

In practice, it would be better style to initialize `name2` in the constructor.

Syntax:

```
[Embed(source="/filename.ext")]
private var name1:Class;
private var name2:Type = new name1() as Type;
```
Syntax:

import flash.media.Sound;
...
[Embed(source="/filename.mp3")]
private var name1:Class;
private var name2:Sound = new name1() as Sound;

Functions/Fields

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sound.play();</td>
<td>Plays the mp3, returns a SoundChannel object which could be used to tinker with the sound</td>
</tr>
<tr>
<td>sound.id3</td>
<td>Contains metadata about the sound such as artist, duration, title, etc.</td>
</tr>
<tr>
<td>sound.length</td>
<td>Returns the length of the mp3</td>
</tr>
</tbody>
</table>

Syntax:

import flash.display.Bitmap;
...
[Embed(source="/filename.jpg")] //also png or gif
private var name1:Class;
private var name2:Bitmap = new name1() as Bitmap;

<table>
<thead>
<tr>
<th>Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pic.getBounds(obj:DisplayObject)</td>
<td>Returns a Rectangle object that defines the area of pic relative to obj</td>
</tr>
<tr>
<td>pic.hitTestObject(obj:DisplayObject);</td>
<td>Determines if pic is intersecting the given DisplayObject</td>
</tr>
<tr>
<td>pic.hitTestPoint(x: Number, y: Number);</td>
<td>Determines if pic intersects (x, y)</td>
</tr>
</tbody>
</table>

In games it is often important to know when objects on the screen are touching.

Even the simplest games need to respond to objects colliding with either other objects or specific points.

Examples

- Pong – ball bounce
- Worm – eating an apple
- Galaga – bullets hitting enemies
- Halo – pwning newbs