Statement of agreement

This page stands in for whatever it is that needed signing. Everything below the rule is about the signature rather than the agreement, and the signature itself is in the box at the foot of this page - drawn by this demo, because a signature appearance is decoration the caller supplies and not something the library invents.

A PDF signature is a chicken-and-egg problem the format solves by cheating. The signature covers the file and the signature is in the file, so the bytes to be hashed cannot be known until the signature is written, and the signature cannot be computed until they are.

The way out is to write the file with a hole of a known size where the signature will go, and a /ByteRange saying "everything except that hole" - then compute the signature over what was written and patch it into the hole without moving a single byte. Every field involved is written at a fixed width, and that is what makes the second pass a patch rather than a re-layout.

What is signed, and what is not

The two spans either side of the hole, which between them are the whole file. A signature is entitled to cover only part of one, and a signature covering only part of a file is precisely how a document gets altered without the alteration showing - so "does it verify" and "does it cover everything" are two questions and both have to be answered yes.

The revision is appended, never rewritten

Not an optimisation. A signature covers a byte range of the file, so rewriting the file invalidates it - and rewriting is exactly what Save does. Signing therefore goes through SaveIncremental, and a document already signed keeps every earlier signature intact. This is also why this demo overrides how it is written out: the base class saves the document it was handed, and doing that here would throw the signature away.

The signature appearance is drawn into this box

Signed by the sample app

Self-signed. Trusted by nothing.

The tick a reader shows is about the certificate,
not about the drawing in this box.

What the signature says

Read back with PdfSignatures.InDocument, which walks the interactive form's field tree and reports what it finds. Nothing here has been checked - the name, the reason and the time are text the producer wrote, and are evidence of nothing on their own.

Field name	`Signature1`
/SubFilter	`/ETSI.CAdES.detached`
Signer name	`PdfPinata sample app`
Reason	`To demonstrate what a signed PDF contains`
Location	`The sample app's output directory`
Claimed time	`2026-09-22 12:20:58Z`
Certification level	`0 – an approval signature, not a certifying one`
/ByteRange	`0 1603 34373 672`
The hole	`1,603 to 34,373 – 32,770 bytes reserved`
Signature written	`16,384 bytes, padded with zeros`
File signed	`35,045 bytes`

PAdES or PKCS#7

The /SubFilter is what says how to read /Contents, and it belongs to the signer rather than to the writer. PAdES - /ETSI.CAdES.detached - hashes the signing certificate into the signed attributes, which closes a real hole: without it the certificate is merely carried alongside the signature, and an attacker who can find a second certificate whose key verifies the same signature can swap it in and change who the document appears to have been signed by.

The time is not evidence

It is the producer's own clock, and a reader will say so. Making the time of signing provable needs a timestamp token from a time-stamping authority - PAdES B-T - and that is not implemented. PAdES also asks that the CMS signing-time attribute be left out when /M carries the claim, because two claimed times that can disagree help nobody, so Pkcs7Signer leaves it out for PAdES and puts it in for PKCS#7.

Certifying, rather than approving

PdfSignatureOptions.Certification turns the signature into a /DocMDP one, which declares what a later revision is still allowed to do - nothing, form filling, or form filling and annotation. An ordinary signature says "I signed this"; a certifying one says "and here is what may still happen to it".

What verifying it proves

PdfSignatureVerifier answers two questions and refuses to conflate them. IsIntact says the signature verifies over the bytes it covers. CoversWholeDocument says those bytes are the whole file but for the signature itself. A signature over the first page of a five page document is perfectly intact, and reporting only that would report the document as sound when a reader would not.

IsIntact	`true`
CoversWholeDocument	`true`
IsValid	`true - both of the above`
Problem	`(none)`
Certificate subject	`CN=PdfPinata Sample App, O=Not a real signer`
Certificate expires	`2027-09-22`
Digest	`SHA-256; SHA-1 and MD5 are refused by the signer`

Integrity checking, not validation

No certificate chain is built, no trust store is consulted, no revocation is checked and no timestamp is evaluated - all of which a reader showing a green tick has done. A signature reported valid here may have been made with a certificate nobody should trust, and the one on this file was: it is self-signed by a key this demo generated a moment ago and threw away.

It is still the check that catches things

A document edited after signing, a signature covering only part of the file, a producer that got its byte range wrong - those are what actually goes wrong, and all three are questions about bytes rather than about who is believed.

Where the split is

The core package holds all the PDF machinery - the placeholder, the byte range, the patching - and no cryptography at all, behind the IPdfSigner seam. PdfPinata.Signing is the package that carries a dependency the core refuses, and it is the one shipped package that does not target netstandard2.1. A signer of your own - a smart card, an HSM, a remote signing service - implements IPdfSigner and needs neither.