		Unlocking Script / scriptSig / Witness written by the payee (payment destination) of a previous transaction when it becomes the payer (payment source) of a new transaction in which the UTXO has to be spent	**Locking Script / scriptPubKey / Witness Program** written by the payer (payment source) of a transaction to commit BTCs to the payee (payment destination)'s UTXO
Legacy	**P2PKH**	<Signature> <PubKey>	DUP HASH160 <PubKeyHash> EQUALVERIFY CHECKSIG where: **PubKeyHash = HASH160(PubKey)** ↓ generates **1* address** via Base58Check encoding with 0x00 prefix
	P2SH	*RedeemingData* <RedeemScript> ↓ ↓ Data compliant with BIP16 prescriptions and satisfying RedeemScript (e.g., enough stack-PUSHed signatures in multisignature contract) BIP16's two stages for P2SH: 1) POP the top stack data and verify it against the locking script; 2) if previous check is successful, deserialize the POPped data and use it as new locking script for the remaining part of the unlocking script	HASH160 <RedeemScriptHash> EQUAL where: **RedeemScriptHash = HASH160(RedeemScript)** ↓ generates **3* address** via Base58Check encoding with 0x05 prefix *by checking if address begin with “1” or “3”, the payer can build the right locking script for payee's address*
Segwit Native	**P2WPKH**	*empty **scriptSig** field* *(“spendable by everyone”)* <Signature> <CompressedPubKey> as P2PKH, but everything in “segregated” structure	0 <PubKeyHash> ↓ Segwit version (Q3 2019) ↓ they generate **bc1* address** via Bech32 encoding where: **PubKeyHash = HASH160(CompressedPubKey)** *uncompressed keys are ok in legacy cases, but nonstandard in Segwit*
	P2WSH	*empty **scriptSig** field* *(“spendable by everyone”)* *RedeemingData* <RedeemScript> as P2SH, but everything in “segregated” structure	0 <RedeemScriptHash> ↓ Segwit version (Q3 2019) ↓ they generate **bc1* address** via Bech32 encoding where: **RedeemScriptHash = SHA256(RedeemScript)** *different sizes (for “bc1” addresses too), but same locking scripts form*
Segwit Compatibility	**P2WPKH inside P2SH**	<0 <PubKeyHash>> BIP16 2nd stage is P2WPKH ↕ <Signature> <CompressedPubKey> in “segregated” structure	HASH160 <RedeemScriptHash> EQUAL where: **RedeemScriptHash = HASH160(RedeemScript)** P2SH **RedeemScript = 0 <PubKeyHash>** **PuKeyHash = HASH160(CompressedPubKey)** P2WPKH
	P2WSH inside P2SH	<0 <ActualRedeemScriptHash>> BIP16 2nd stage is P2WSH ↕ *RedeemingData* <ActualRedeemScript> in “segregated” structure	HASH160 <RedeemScriptHash> EQUAL where: **RedeemScriptHash = HASH160(RedeemScript)** P2SH **RedeemScript = 0 <ActualRedeemScriptHash>** **ActualRedeemScriptHash = SHA256(ActualRedeemScript)** P2WSH

Notes	**HASH160(x) = RIPEMD160(SHA256(x))** generates 20 bytes hash **SHA256(x)** generates 32 bytes hash **<x>** transaction script operator that PUSHes in the stack the ***x*** data by means of an opcode declaring ***x***'s size in bytes IMPORTANT: if the hashed or pushed **x** is a script, the actual data being processed is the script serialization
***(not exhaustive)* Credits**	**Andreas M. Antonopoulos**'s *Mastering Bitcoin 2nd Ed.* – O'Reilly (especially chapters 4, 6, 7, appendixes B and D) **Jimmy Song**'s *Understanding Segwit Block Size* on Medium Bitcoin Improvement Proposals : *BIP 16*, *BIP 141*, *BIP 173*, *...* **Greg Walker**'s *P2SH page* on his *learnmeabitcoin.com* website **yaoshiang**'s *bitcoin-script-disassembler*