Note: Sidekiq has changed its API since we recorded this video. If you didn't update sidekiq, you should be fine; otherwise take a look at the new doc here for testing:

https://github.com/mperham/sidekiq/wiki/Testing

Note: A bit further explanation on background jobs and Sidekiq:

How come Sidekiq needs Redis? How does Redis fit into the picture?

Sidekiq (and Resque, for that matter) needs to retain a job queue so that the jobs that are queued first will be run first. Redis is used to maintain that queue. You could, of course, use something else to do this, such as a database (that's what delayed_job does https://github.com/collectiveidea/delayed_job), where you essentially add records to a db table to maintain the queue. Redis as a key-value store is much simpler than a full fledged database, and that's why Sidekiq has picked it to store its job queue.

Why it's not a good idea to pass Ruby objects directly to Sidekiq?

The reason you don't want to pass Ruby objects directly into sidekiq jobs is that your job processor could be on a different server than your web app. In fact, your Rails App, Sidekiq workers and the Redis can run on 3 different servers - If you run on Heroku, for examples, your "dynos" (web dyno and worker dynos) may very likely run on different EC2 servers (Heroku uses Amazon EC2 under the cover). And if you use a Redis plugin provider like redis-to-go, that itself runs on a different server. Whenever you are sending data across servers, you want to keep your data in a way that's easily serializable - you can't send Ruby objects directly over the wire, you can only send string representation of objects. The simplest would be integers or strings. You could, of course, host your own server and run all 3 on the same server, but it's good to keep that flexibility in mind.