# Overview:

* The bridge pattern will decouple an abstraction from its implementation so that the two can vary independently:
  + Decouples implementation class and abstract class by providing a bridge structure between them.
* We already understand the benefits of decoupling and abstraction.
  + Decoupling means to have things behave independently from each other.
  + Abstraction is how different things are related to each other conceptually (hiding details).
* Implementations here mean the objects that the abstract class and its derivations use to implement themselves:
  + Not the derivations of the abstract class (concrete classes).
* This pattern helps us to make concrete class functionalities independent from the interface implementer class.
  + Can alter these different kind of classes structurally without affecting each other.

# Examples:

* A standard software company will typically consist of the following two teams:
  + Development team.
  + Technical support team.
* A change in the operational strategy of one team should not have a direct impact on the other team.
* The technical support team plays the role of a bridge between the clients and the development team that implements the product.
* Another example would be in a GUI framework:
  + Separate window abstraction from window implementation.

# When to use the Bridge Pattern:

* When you want to avoid a permanent binding between an abstraction and its implementation.
  + When the implementation must be selected or switched at run-time.
* When both the abstractions and their implementations should be extensible by subclassing
  + Let you combine the different abstractions and implementations and extend them independently.
* When changes in the implementation of an abstraction should have no impact on clients
  + Client’s code should not have to be recompiled.
* When you want to hide the implementation of an abstraction completely from clients.
* When you have a ton of implementation classes
  + A class hierarchy indicates the need for splitting an object into two parts.

# Advantages of the Bridge Pattern:

* Decouples an implementation so that it is not bound permanently to an interface.
* Abstraction and implementation can be extended independently:
  + Allows you to vary the implementation and the abstraction by placing the two in separate class hierarchies.
* Changes to the concrete abstraction classes do not affect the client.
* Adds one more method level redirection to achieve the objective.
* One drawback is that it does slightly increase complexity.

# Compared to the Adapter:

* The Adapter pattern is geared toward making unrelated classes work together:
  + Usually applied to systems after they have been designed.
* In contrast, the Bridge is used up-front in a design:
  + Let abstractions and implementations vary independently.

# Implementation (Overview):

* There are two parts in the Bridge design pattern implementation:
  + Abstraction : an interface or an abstract class
  + Implementation: an interface or abstract class.
* Allows the abstraction and the implementation to be developed independently.
  + The client code can access only the abstraction part.
  + Client not concerned about the implementation part.
* The abstraction contains a reference to the implementer.
* Children of the abstraction are referred to as refined abstractions.
* Children of the implementer are concrete implementers.
* Since we can change the reference to the implementer in the abstraction, we are able to change the abstractions’ implementer at run-time.
* Changes to the implementer do not affect client code.
  + Increases the loose coupling between class abstraction and its implementation.

# Participants in detail:

* **Abstraction:**
  + Core of the bridge design pattern and defines the crux.
    - Defines the abstraction’s interface.
  + Contains a reference to the implementer.
* **RefinedAbstraction:**
  + Extends the abstraction takes the finer detail one level below.
  + Hides the finer elements from implementers.
* **Implementer:**
  + Defines the interface for implementation classes:
    - Does not need to correspond directly to the abstraction interface and can be very different:
      * Implementer provides only primitive operations.
      * Abstractions defines higher-level operations based on these primitives.
  + Provides an implementation in terms of operations provided by Implementer interface.
* **Concrete Implementer:**
  + Impalements the above implementer by providing concrete implementation.

# Composition/Aggregation over inheritance:

* The implementation of bridge design pattern follows the notion to prefer Composition over inheritance.
* When an abstraction can have one of several possible implementations:
  + Usually inheritance will be used to accommodate this.
  + An abstract class defines the interface to the abstraction, and concrete subclasses implement it in different ways.
* This approach is not always flexible enough:
  + Inheritance binds an implementation to the abstraction permanently.
    - Makes it difficult to modify, extend, and reuse abstractions and implementations independently.
* The bridge pattern is an excellent example of following two of the mandates of the design pattern community:
  + “Find what varies and encapsulate it”.
  + “Favor aggregation over class inheritance”.
* The Bridge pattern is one of the toughest patterns to understand in part because it is so powerful and applies to so many situations:
  + Also goes against a common tendency to handle special cases with inheritance.

# High Level Design Example:

* Imagine you’re going to revolutionize “extreme lounging”.
* You are writing the code for a new ergonomic and user-friendly remote control for TVs.
* You already know that you have got to use good OO techniques.
  + While the remote is bases on the same abstraction, there will be lots of implementations:
    - One for each model of TV.

# Problems with current design:

* You know that the remote’s user interface won’t be right the first time.
* You expect that the product will be refined many times as usability data is collected on the remote control.
* Your dilemma is that the remotes are going to change and the TVs are going to change.
* You have already abstracted the user interface:
  + You can vary the implementation over the many TVs your customers will own.
* You are also going to need to vary the abstraction:
  + It is going to change over time as the remote is improved based on the user feedback.
* With the current design we can vary only the TV implementation, not the user interface.

# Bridge Pattern Design:

* Now we have two hierarchies:
  + One for the remotes.
  + One for platform-specific TV implementations.
* The bridge allows us to vary either side of the two hierarchies independently.

# Summary:

* Pattern is extremely helpful when our class and its associated functionalities may change in frequent intervals.
* We remove the concrete binding between an abstraction and the corresponding implementation:
  + Both hierarchies (abstraction and its implementations) can extend through child classes.
  + Both hierarchies can grow independently.
    - If we make any change in abstraction methods, they do not have an impact on the implementer method.
* The abstraction and implementer do not need to be an abstract classes.
  + Can be interfaces.
* The abstraction contains the reference to its implementer.
* You can change the implementers dynamically (at runtime) by changing the reference in the abstraction.