```python
"""
CS241 Checkpoint 7A
Written by Chad Macbeth
"""

"""
File: check07a.py

Starting template for your checkpoint assignment.
"""

# In this checkpoint, we are going to define a Car base class
# with 3 subclasses called Civic, Odyssey, and Ferrari that inherit.
# Not only does inheritance provide for reuse, but it also provides
# an opporutnity to define functions that each subclass must implement.
#
# AN abstract method is a function that must be implemented in the subclasses.
# THis is useful when we want to create a function like attach_doors which takes
# as a paramter a car.  We can call get_door_specs and python will select the
# correct implementation based on the object type.
#
# An Abstract Base Class (ABC) is a class that contains abstract methods.  You
# can not create objects of an ABC.  You can only create objects of the subclasses.
# This is handy when you want your base class to define the types of functions
# that each subclass should create.

from abc import ABC  # YOu need these two imports when working with abstract classes
from abc import abstractmethod

class Car(ABC):  # Inheriting ABC class signals that this is an abstract base class

    def __init__(self):
        self.name = "Unknown model"

    @abstractmethod  # This signals that this function is an abstract method
    def get_door_specs(self):
        return "Unknown doors"  # This will only run if one of the base classes calls
                                # super().get_door_specs().  Normally abstract methods
                                # just say "pass".

class Civic(Car):

    def __init__(self):
        self.name = "Civic"

    def get_door_specs(self):
        return "4 doors"

class Odyssey(Car):

    def __init__(self):
        self.name = "Odyssey"

    def get_door_specs(self):
        return "2 front doors, 2 sliding doors, 1 tail gate"

class Ferrari(Car):

    def __init__(self):
        self.name = "Ferrari"

    def get_door_specs(self):
        return "2 butterfly doors"

def attach_doors(car):
    print("Attaching doors to {} - {}" .format(car.name, car.get_door_specs()))
```

```
    # Python will automatically call the correct get_door_specs based on the original object created
    # If you didn't use abstract methods and if you forgot to provide a get_door_specs function in
    # one of the classes, then you would run into an error here.  This is polymorphisim!


def main():
    car1 = Civic()
    car2 = Odyssey()
    car3 = Ferrari()

    attach_doors(car1)
    attach_doors(car2)
    attach_doors(car3)

if __name__ == "__main__":
    main()

```