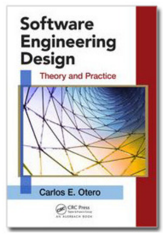
CHAPTER 6: CREATIONAL DESIGN PATTERNS

SESSION II: FACTORY METHOD

Software Engineering Design: Theory and Practice
by Carlos E. Otero



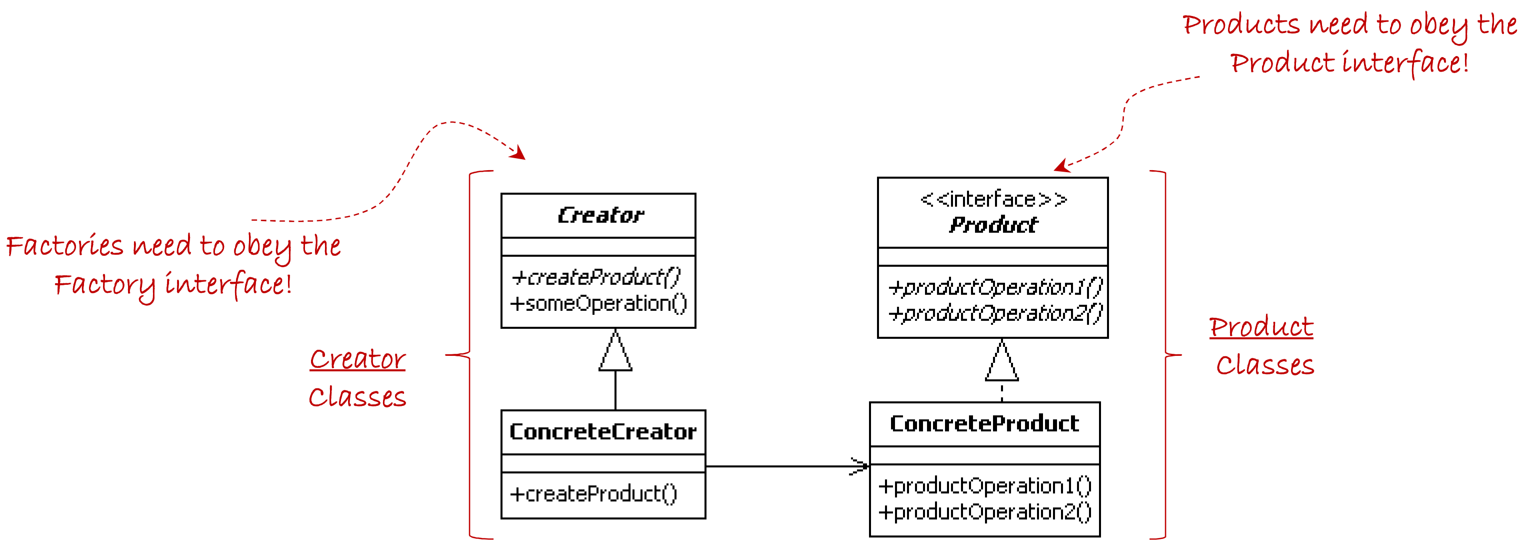
SESSION'S AGENDA

- Creational Patterns in Detailed Design
- Factory Method Design Pattern
  - ✓ Java Example
  - ✓ C++ Example
- Null Object Design Pattern
- Benefits of Factory Method
- What's next…

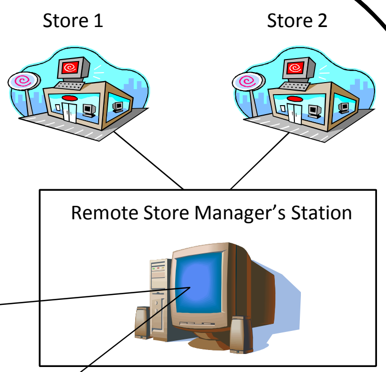
Factory Method Design Pattern

- The Factory Method design pattern is a class creational pattern used to encapsulate and defer object instantiation to derived classes.

- According to the GoF, the intent of the factory method is to [1]
  - ✓ Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses.



Factory Method Example



Consider this code, which handles all computers from stores #1 and #2. What's wrong with it?

Java code

```java
public class ComputerStore {

    public void displayComputer(String type) {

        // The computer object.
        Computer computer = null;

        if( type.equals("DellPC")) {

            // Instantiate a Dell computer.
            computer = new DellPC();

        } else if( type.equals("GatewayPC")) {

            // Instantiate a Gateway computer.
            computer = new GatewayPC();

        } else if(type.equals("Mac")) {

            // Instantiate a MAC computer.
            computer = new Mac();
        }

        // Display the computer's information.
        computer.displayMemoryInfo();
        computer.displayMonitorInfo();
        computer.displayProcessorInfo();
        computer.displayCustomerRatings();
        computer.displayCost();
    }
```

When you see code like this, you know that when it comes time for changes or extension, you'll have to reopen this code and examine what needs to be added or deleted. In addition, more often than not, this kind of code ends up in several parts of the application, making maintenance more difficult and error-prone.

```java
public Computer orderComputer(String type) {
    // The computer object.
    Computer computer = null;

    if( type.equals("DellPC")) {

        // Instantiate a Dell computer.
        computer = new DellPC();

    } else if( type.equals("GatewayPC")) {

        // Instantiate a Gateway computer.
        computer = new GatewayPC();

    } else if(type.equals("Mac")) {

        // Instantiate a MAC computer.
        computer = new Mac();
    }

    // Process the computer sale.
    processSale(computer.getProductId());

    return computer;
}
```

This code would have to support all types of computer in all sites!

Let's take a closer look in the next slide…

Factory Method Example

This code is not closed for modification. If a new computer is added to the inventory of store #1, we have to get into this code and modify it for all other stores!

```
public void displayComputer(String type) {

    // The computer object.
    Computer computer = null;

    if( type.equals("DellPC")) {

        // Instantiate a Dell computer.
        computer = new DellPC();

    } else if( type.equals("GatewayPC")) {

        // Instantiate a Gateway computer.
        computer = new GatewayPC();

    } else if(type.equals("Mac")) {

        // Instantiate a MAC computer.
        computer = new Mac();
    }

    // Display the computer's information.
    computer.displayMemoryInfo();
    computer.displayMonitorInfo();
    computer.displayProcessorInfo();
    computer.displayCustomerRatings();
    computer.displayCost();
}
```

Design Principle:
Encapsulate what varies!

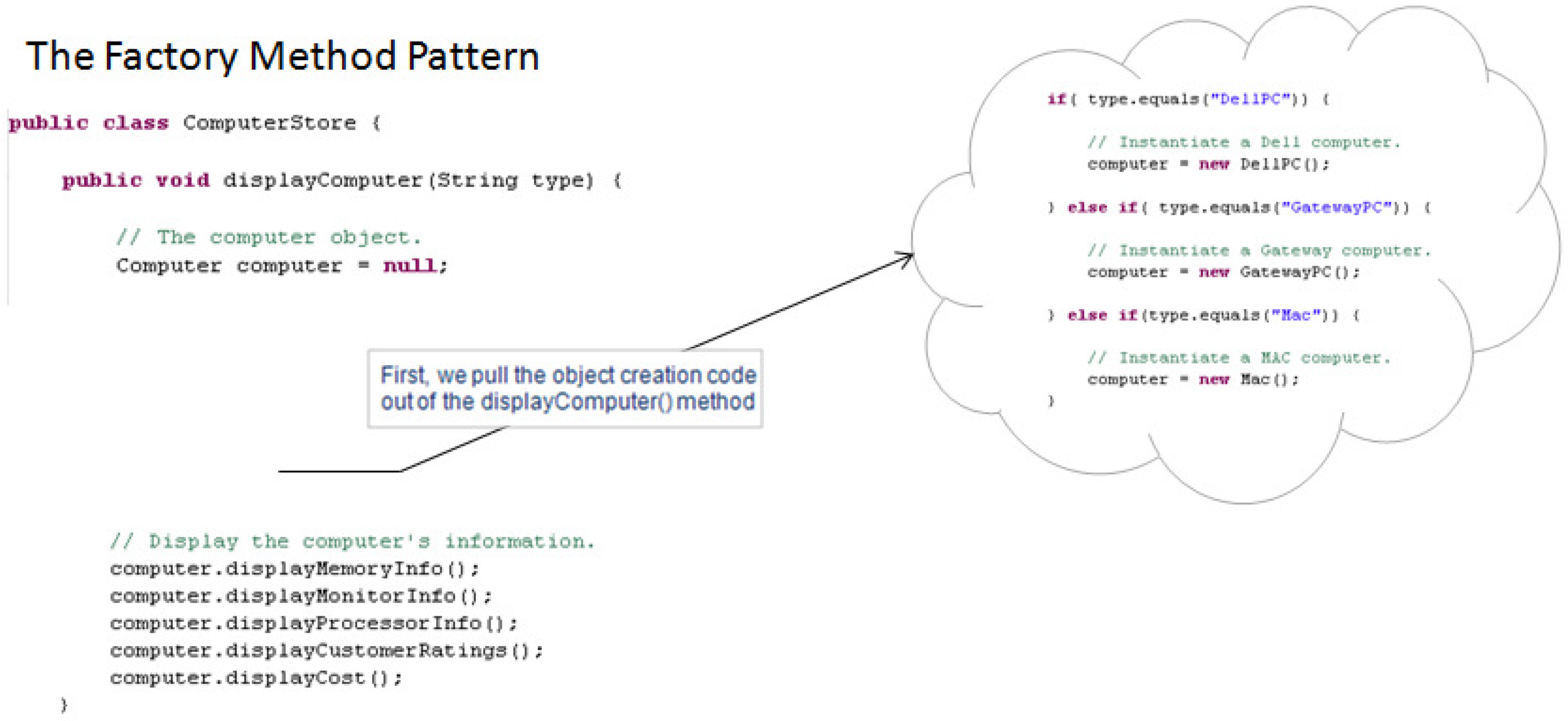
This is what varies. As the computer selection changes over time (for all computer stores), you'll have to modify this code over and over.

This is what we expect to stay the same.

Important:
By coding to an interface, we can insulate ourselves from future changes!

Let's see how the Factory Method can be used to solve this problem...

Factory Method Example



The Factory Method Pattern

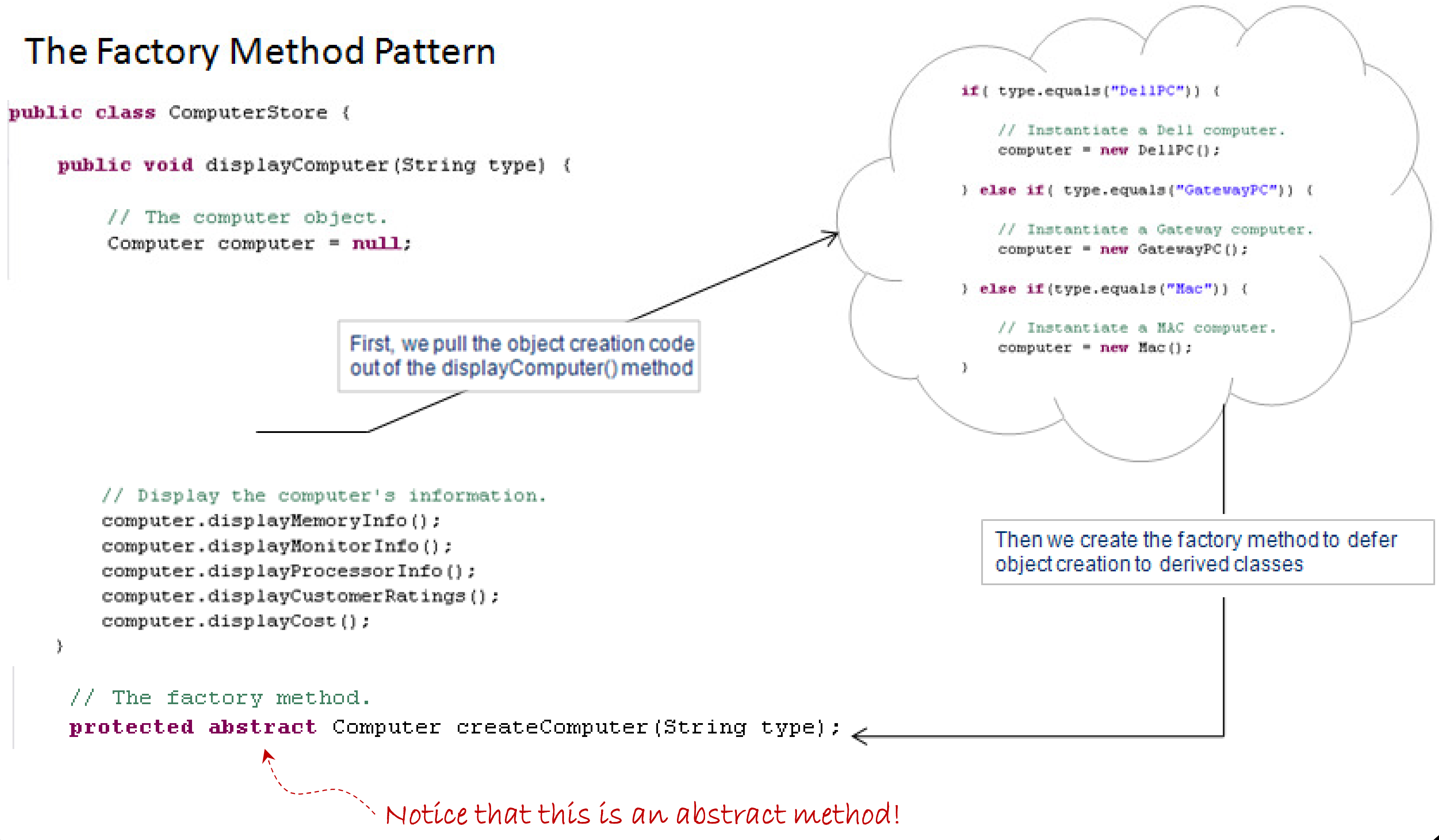
```java
public class ComputerStore {

    public void displayComputer(String type) {

        // The computer object.
        Computer computer = null;
```

```java
if( type.equals("DellPC")) {

    // Instantiate a Dell computer.
    computer = new DellPC();

} else if( type.equals("GatewayPC")) {

    // Instantiate a Gateway computer.
    computer = new GatewayPC();

} else if(type.equals("Mac")) {

    // Instantiate a MAC computer.
    computer = new Mac();
}
```

First, we pull the object creation code out of the displayComputer() method

```java
        // Display the computer's information.
        computer.displayMemoryInfo();
        computer.displayMonitorInfo();
        computer.displayProcessorInfo();
        computer.displayCustomerRatings();
        computer.displayCost();
    }
```

Factory Method Example



The Factory Method Pattern

```java
public class ComputerStore {

    public void displayComputer(String type) {

        // The computer object.
        Computer computer = null;
```

```java
if( type.equals("DellPC")) {

    // Instantiate a Dell computer.
    computer = new DellPC();

} else if( type.equals("GatewayPC")) {

    // Instantiate a Gateway computer.
    computer = new GatewayPC();

} else if(type.equals("Mac")) {

    // Instantiate a MAC computer.
    computer = new Mac();
}
```

First, we pull the object creation code out of the displayComputer() method

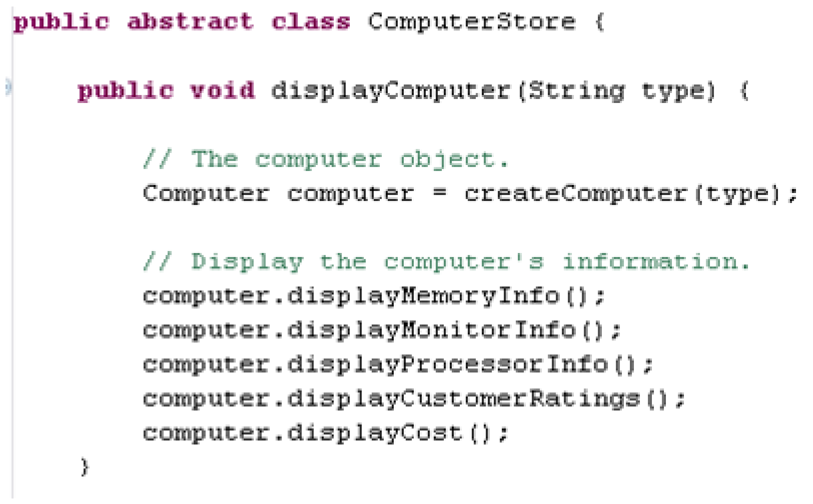
```java
        // Display the computer's information.
        computer.displayMemoryInfo();
        computer.displayMonitorInfo();
        computer.displayProcessorInfo();
        computer.displayCustomerRatings();
        computer.displayCost();
    }

    // The factory method.
    protected abstract Computer createComputer(String type);
```

Then we create the factory method to defer object creation to derived classes

Notice that this is an abstract method!

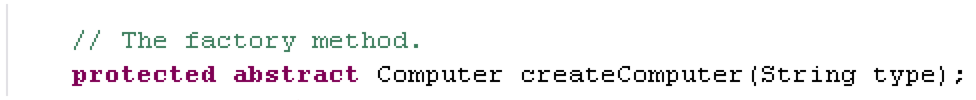
Factory Method Example

The Factory Method Pattern



```java
public abstract class ComputerStore {

    public void displayComputer(String type) {

        // The computer object.
        Computer computer = createComputer(type);

        // Display the computer's information.
        computer.displayMemoryInfo();
        computer.displayMonitorInfo();
        computer.displayProcessorInfo();
        computer.displayCustomerRatings();
        computer.displayCost();
    }
```

Finally, we add the factory method to our code. Keep in mind that by adding the abstract factory method, the ComputerStore now is abstract as well. This enforces object creation in derived classes.

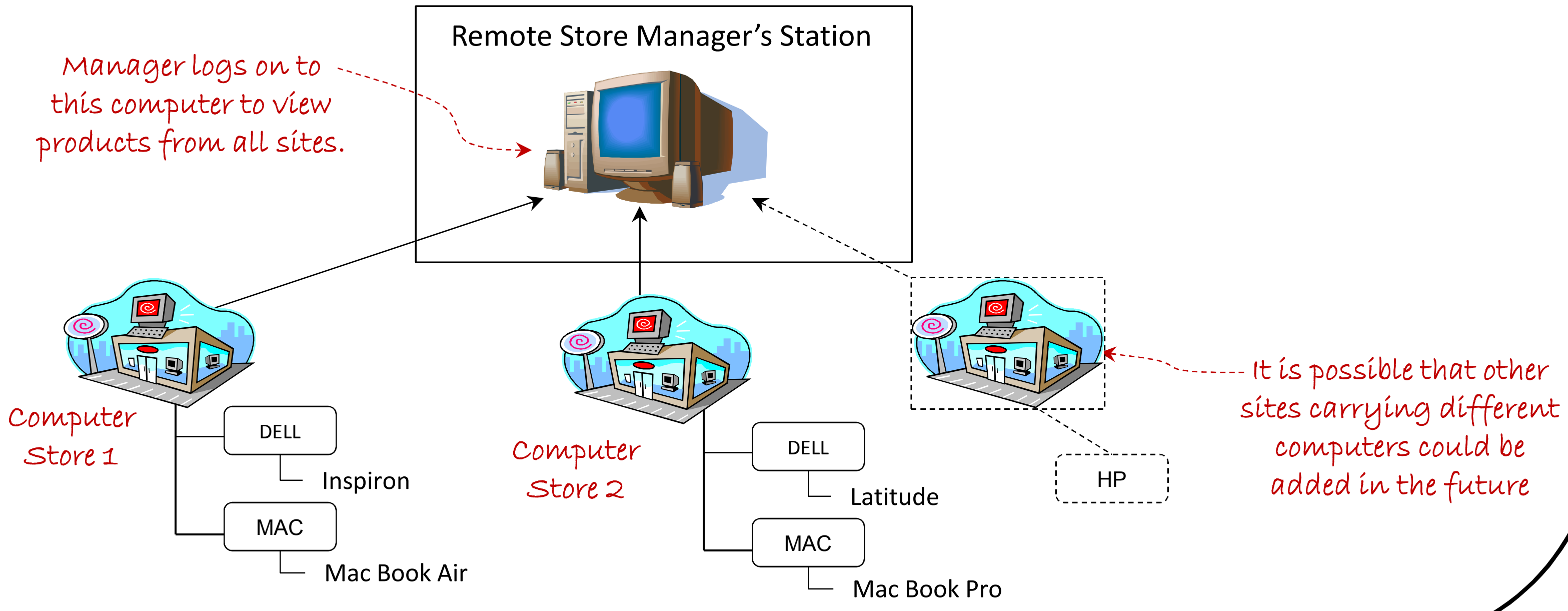


```java
    // The factory method.
    protected abstract Computer createComputer(String type);
```

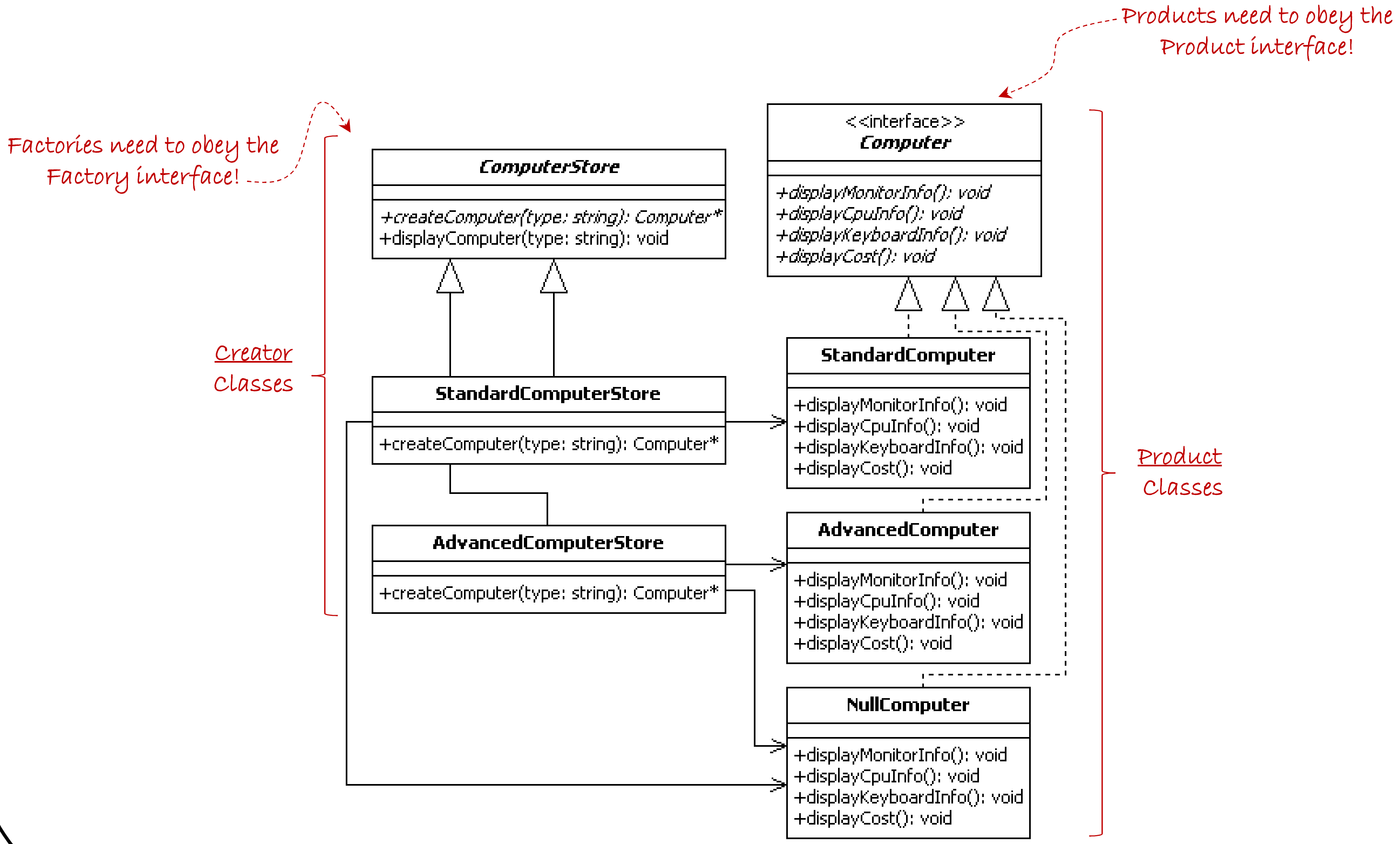
Notice that this is an abstract method!

Factory Method Example

- Consider an information system for a chain of Computer Stores. The chain is currently composed of two sites and the manager wants to have a centralized software that manages all two sites. Mainly, the manager wants this centralized system to keep track of computer sales and display computer information for all sites. Not all sites carry the same computers and new sites could be added in the future, therefore support for computers should be site-specific!


Remote Store Manager's Station
Manager logs on to this computer to view products from all sites.
Computer Store 1
DELL
Inspiron
MAC
Mac Book Air
Computer Store 2
DELL
Latitude
MAC
Mac Book Pro
HP
It is possible that other sites carrying different computers could be added in the future

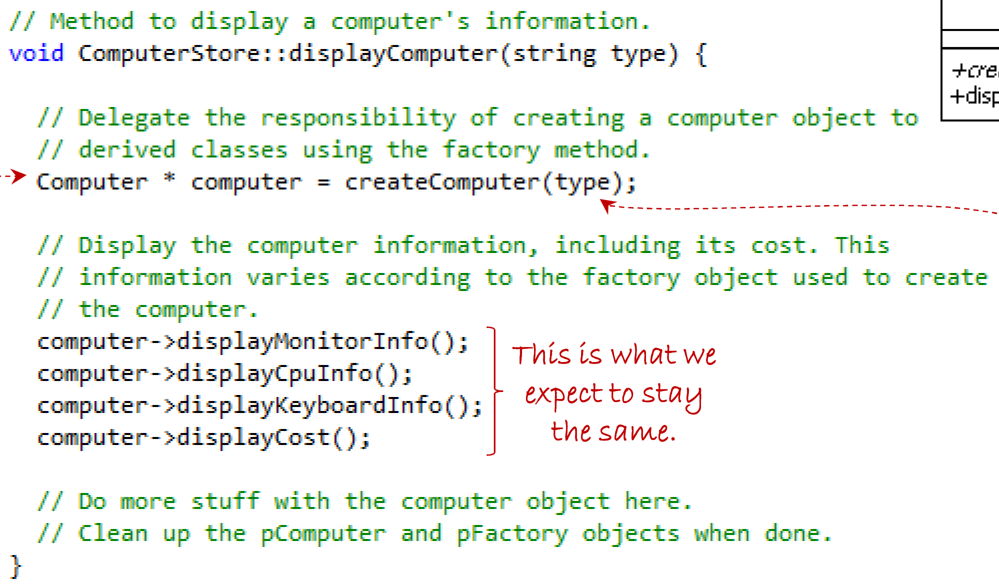
FACTORY METHOD DESIGN PATTERN EXAMPLE



Products need to obey the Product interface!
Factories need to obey the Factory interface!
Creator Classes
Product Classes
ComputerStore
+createComputer(type: string): Computer*
+displayComputer(type: string): void
<<interface>> Computer
+displayMonitorInfo(): void
+displayCpuInfo(): void
+displayKeyboardInfo(): void
+displayCost(): void
StandardComputerStore
+createComputer(type: string): Computer*
AdvancedComputerStore
+createComputer(type: string): Computer*
StandardComputer
+displayMonitorInfo(): void
+displayCpuInfo(): void
+displayKeyboardInfo(): void
+displayCost(): void
AdvancedComputer
+displayMonitorInfo(): void
+displayCpuInfo(): void
+displayKeyboardInfo(): void
+displayCost(): void
NullComputer
+displayMonitorInfo(): void
+displayCpuInfo(): void
+displayKeyboardInfo(): void
+displayCost(): void

Factory Method Example

➢ Sample code for the displayComputer (…)method:

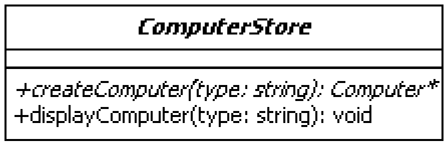


```
// Method to display a computer's information.
void ComputerStore::displayComputer(string type) {

  // Delegate the responsibility of creating a computer object to
  // derived classes using the factory method.
  Computer * computer = createComputer(type);

  // Display the computer information, including its cost. This
  // information varies according to the factory object used to create
  // the computer.
  computer->displayMonitorInfo();
  computer->displayCpuInfo();
  computer->displayKeyboardInfo();
  computer->displayCost();

  // Do more stuff with the computer object here.
  // Clean up the pComputer and pFactory objects when done.
}
```

Factory method!

Optional parameter

This is what we expect to stay the same.



ComputerStore
+createComputer(type: string): Computer*
+displayComputer(type: string): void

Factory Method Example

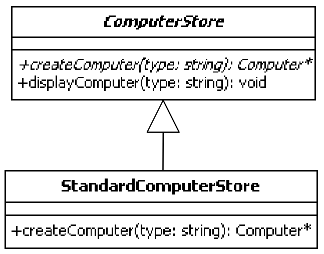
Factory method implementation for standard computer store

Optional parameter

```
// The factory method for creating computer products.
Computer* StandardComputerStore::createComputer(string type)
{
    // Pointer to a computer object.
  Computer* computer = 0;

  // Determine which computer needs to be created.
  if( type.compare("standard") == 0 ) {
    // Create the StandardComputer. Clients are responsible for cleaning
    // up the memory for the computer object. Internally, StandardComputer
    // uses StandardComputerPartsFactory to create a standard computer.
    computer = new StandardComputer;
  }
  else {
    // Computer type not supported at this store. Create a null computer object.
    computer = new NullComputer(type);
  }

  // Return the newly created computer object. Clients are responsible
  // for cleaning up the computer object.
  return computer;
}
```

Rarely a computer store will carry only one computer!

A generic standard computer

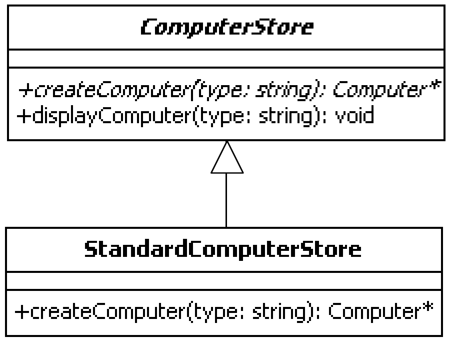


ComputerStore
+createComputer(type: string): Computer*
+displayComputer(type: string): void
StandardComputerStore
+createComputer(type: string): Computer*

We use the optional parameter to support computers that may be carried in the future by the standard computer store

Let's see how to support new standard computers…

Factory Method Example

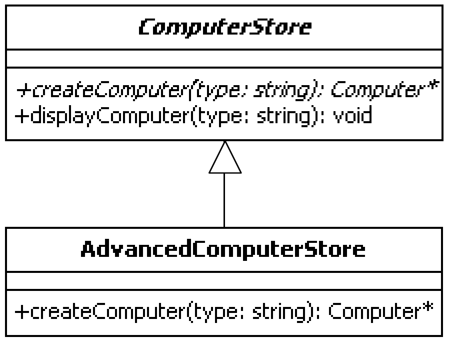

ComputerStore
+createComputer(type: string): Computer*
+displayComputer(type: string): void
StandardComputerStore
+createComputer(type: string): Computer*

The standard computer store now carries Dell Inspiron and Mac Book Air computers

Important:
Notice that these changes are independent from what happens in the AdvancedComputer Store!

```
// The factory method for creating computer products.
Computer* StandardComputerStore::createComputer(string type)
{
    // Pointer to a computer object.
  Computer* computer = 0;

  // Determine which computer needs to be created.
  if( type.compare("standard") == 0 ) {
    // Create the StandardComputer. Clients are responsible for cleaning
    // up the memory for the computer object. Internally, StandardComputer
    // uses StandardComputerPartsFactory to create a standard computer.
    computer = new StandardComputer;
  }
  else if( type.compare("DELL") == 0 ) {
    // Create a standard DELL Computer. Clients are responsible for cleaning
    // up the memory for the computer object. Internally, StandardComputer
    // uses StandardComputerPartsFactory to create a standard computer.
    computer = new DellComputer("DellInspiron");
  }
  else if( type.compare("MAC") == 0 ) {
    // Create a standard MAC Computer. Clients are responsible for cleaning
    // up the memory for the computer object. Internally, StandardComputer
    // uses StandardComputerPartsFactory to create a standard computer.
    computer = new MacComputer("MacBookAir");
  }
  else {
    // Computer type not supported at this store. Create a null computer object.
    computer = new NullComputer(type);
  }

  // Return the newly created computer object. Clients are responsible
  // for cleaning up the computer object.
  return computer;
}
```

Factory Method Example


ComputerStore
+createComputer(type: string): Computer*
+displayComputer(type: string): void
AdvancedComputerStore
+createComputer(type: string): Computer*

Optional parameter

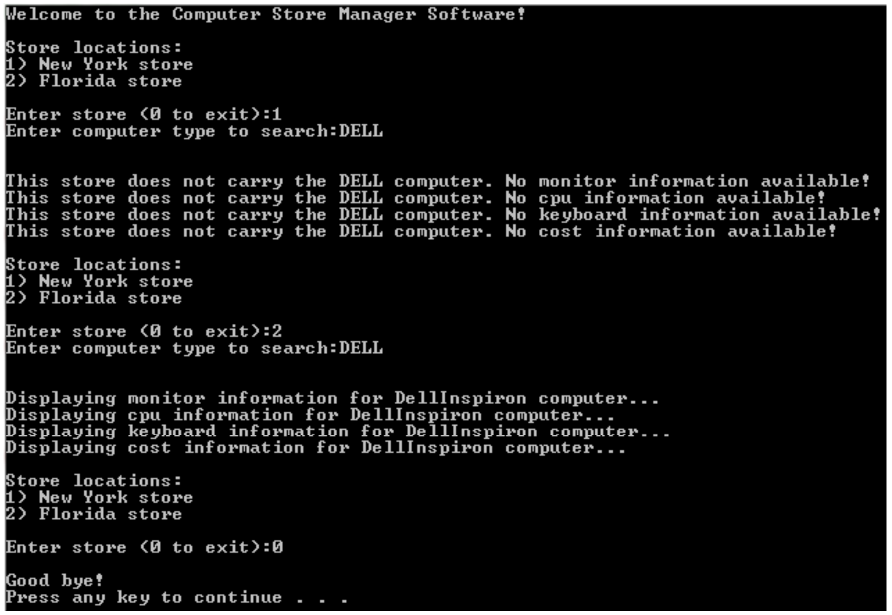
```
// The factory method for creating computer products.
Computer* AdvancedComputerStore::createComputer(string type)
{
    // Pointer to a computer object.
  Computer* computer = 0;

  if( type.compare("MAC") == 0 ) {
    // Create the advanced MAC computer. Clients are responsible for cleaning
    // up the memory for the computer object. Internally, AdvancedComputer
    // uses AdvancedComputerPartsFactory to create the advanced computer.
    computer = new MacComputer("MacBookPro");
  }
  else {
    // Computer type not supported at this store. Create a null computer object.
    computer = new NullComputer(type);
  }

  // Return the newly created computer object. Clients are responsible
  // for cleaning up the computer object.
  return computer;
}
```

Currently, the advanced computer store only carries Mac Book Pro computers

Hopefully, by now, you are able to see how the pattern works to localize changes and support the addition of future computers with minimal impact!

Factory Method Example

```
int main(int argc, char* argv[])
{
    // User input.
    int option = 0;

    // A computer store.
    ComputerStore* pStore = 0;

    // Type of computer.
    string type;

    // Display welcome message.
    cout<<"Welcome to the Computer Store Manager Software!\n\n";

    while(true)
    {
        cout<<"Store locations:\n"
            <<"1) New York store\n"
            <<"2) Florida store\n\n"
            <<"Enter store (0 to exit):";

        cin>>option;

        if( option == 0 || option < 1 || option > 2 )
        {
            cout<<"\nGood bye!\n";
            break;
        }
        else if( option == 1 )
            pStore = new AdvancedComputerStore;
        else if( option == 2 )
            pStore = new StandardComputerStore;

        cout<<"Enter computer type to search:";
        cin>>type;

        pStore->displayComputer(type);
        delete pStore;
    }

    return 0;
}
```

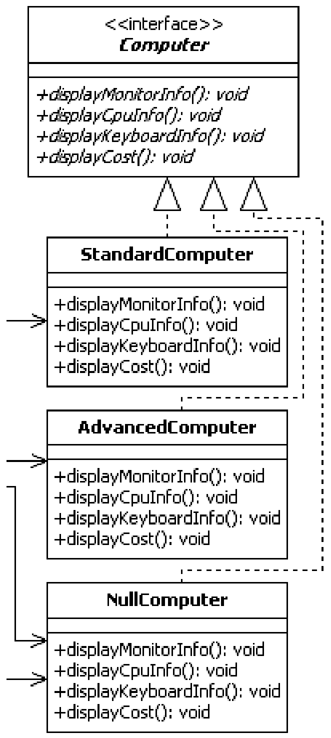


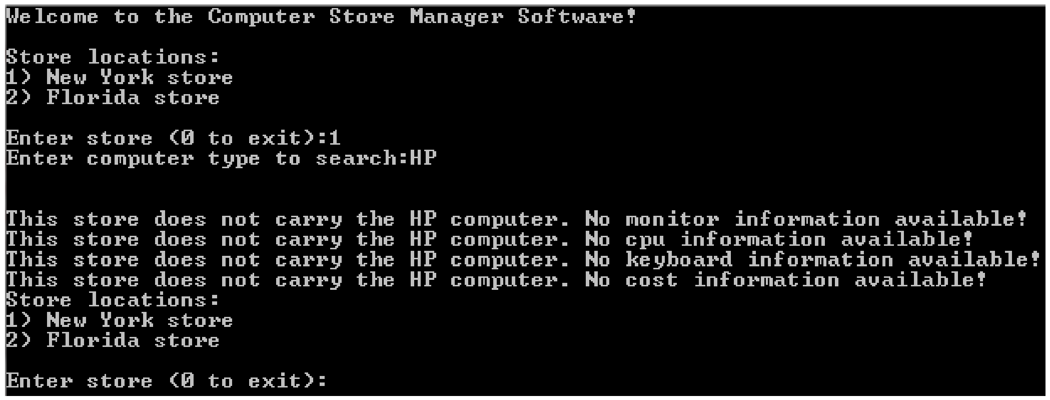
```
Welcome to the Computer Store Manager Software!

Store locations:
1) New York store
2) Florida store

Enter store (0 to exit):1
Enter computer type to search:DELL


This store does not carry the DELL computer. No monitor information available!
This store does not carry the DELL computer. No cpu information available!
This store does not carry the DELL computer. No keyboard information available!
This store does not carry the DELL computer. No cost information available!

Store locations:
1) New York store
2) Florida store

Enter store (0 to exit):2
Enter computer type to search:DELL


Displaying monitor information for DellInspiron computer...
Displaying cpu information for DellInspiron computer...
Displaying keyboard information for DellInspiron computer...
Displaying cost information for DellInspiron computer...

Store locations:
1) New York store
2) Florida store

Enter store (0 to exit):0

Good bye!
Press any key to continue . . .
```

Sample output

Just enough code to demonstrate
the Factory Method in action!

Factory Method Example





```
Welcome to the Computer Store Manager Software!

Store locations:
1) New York store
2) Florida store

Enter store (0 to exit):1
Enter computer type to search:HP


This store does not carry the HP computer. No monitor information available!
This store does not carry the HP computer. No cpu information available!
This store does not carry the HP computer. No keyboard information available!
This store does not carry the HP computer. No cost information available!
Store locations:
1) New York store
2) Florida store

Enter store (0 to exit):
```

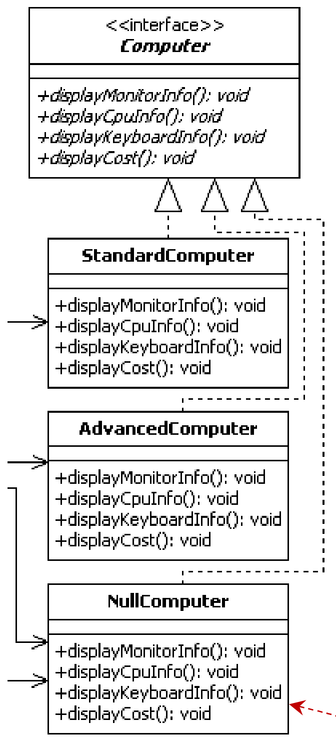
```
// The factory method for creating computer products.
Computer* AdvancedComputerStore::createComputer(string type)
{
   // Pointer to a computer object.
  Computer* computer = 0;

  if( type.compare("MAC") == 0 ) {
    // Create the advanced MAC computer. Clients are responsible for cleaning
    // up the memory for the computer object. Internally, AdvancedComputer
    // uses AdvancedComputerPartsFactory to create the advanced computer.
    computer = new MacComputer("MacBookPro");
  }
  else {
    // Computer type not supported at this store. Create a null computer object.
    computer = new NullComputer(type);
  }
```

The Null Object Pattern!

```
void NullComputer::displayCpuInfo()
{
    cout<<"This store does not carry the " + _type + " computer. No cpu information available!\n";
}

void NullComputer::displayKeyboardInfo()
{
    cout<<"This store does not carry the " + _type + " computer. No keyboard information available!\n";
}
```

Did you notice this NullComputer class?

The Null Object Pattern Example



```
// The factory method for creating computer products.
Computer* StandardComputerStore::createComputer(string type)
{
    // Pointer to a computer object.
  Computer* computer = 0;

  // Determine which computer needs to be created.
  if( type.compare("standard") == 0 ) {
    // Create the StandardComputer. Clients are responsible for cleaning
    // up the memory for the computer object. Internally, StandardComputer
    // uses StandardComputerPartsFactory to create a standard computer.
    computer = new StandardComputer;
  }
  else if( type.compare("DELL") == 0 ) {
    // Create a standard DELL Computer. Clients are responsible for cleaning
    // up the memory for the computer object. Internally, StandardComputer
    // uses StandardComputerPartsFactory to create a standard computer.
    computer = new DellComputer("DellInspiron");
  }
  else if( type.compare("MAC") == 0 ) {
    // Create a standard MAC Computer. Clients are responsible for cleaning
    // up the memory for the computer object. Internally, StandardComputer
    // uses StandardComputerPartsFactory to create a standard computer.
    computer = new MacComputer("MacBookAir");
  }
  else {
    // Computer type not supported at this store. Create a null computer object.
    computer = new NullComputer(type);
  }

  // Return the newly created computer object. Clients are responsible
  // for cleaning up the computer object.
  return computer;
}
```

The Null Object Pattern!

With this pattern, we treat invalid inputs as objects, therefore allowing us to treat them as other valid expected objects!

FACTORY METHOD DESIGN PATTERN

➢ Steps for designing with the Factory Method:

1. Identify and design the product interface (e.g., Computer)
2. Identify and design the concrete products that realize the interface from step 1 (e.g., StandardComputer, AdvancedComputer, DellComputer, etc.)
3. Design the factory interface (e.g., ComputerStore), which contains one abstract factory interface method for delegating product creation to derived classes.
4. Design one or more concrete factories for each product identified in step 2.
5. Associate each factory from step 4 with its respective product from step 2.

➢ Benefits of the Factory Method pattern

- ✓ Separates code from product-specific classes; therefore, the same code can work with various existing or newly created product classes.
- ✓ By separating the code, development becomes efficient, since different developers can work on the different parts of the project at the same time.
- ✓ By separating the code, it becomes easier to reuse and maintain specific parts of the code.

WHAT'S NEXT…

- In this session, we continued the discussion on creational design patterns, including:
  - ✓ Factory Method

- In the next sessions, we will finalize the presentation on creational design patterns. Specifically, we will cover:
  - ✓ Builder
  - ✓ Prototype
  - ✓ Singleton

References

- [1] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley, 1995.