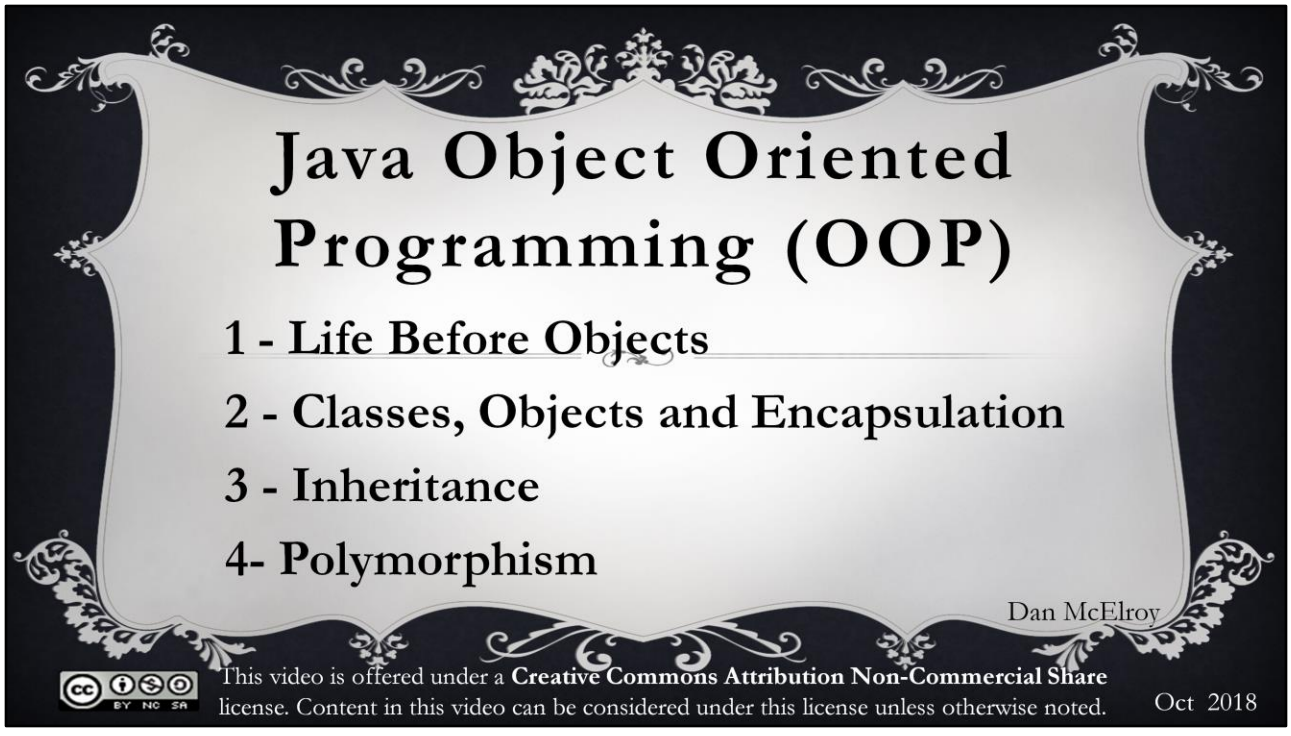


Hello programmers.
This discussion covers classes and objects, encapsulation, inheritance and polymorphism. A lot of information is covered here. You may want to divide watching the presentation in several settings and go back and review this presentation more than one time.

My name is Dan and I am going to make a fortune starting my own online sales company.

I'm going to name my company "dan-azon". I am going to be the biggest retailer on the Internet. I will start by selling books. I just know everybody wants books and lots of them. I was at the library yesterday and saw lots of people using the computers but nobody checking out books. They probably couldn't find the books they wanted because the card catalog was empty. I didn't ask them, but they are probably online buying books. Yeah, I bet that's what they are doing. I am ready to help them out and get rich doing it.

Life Before Objects

Objects and Classes
Encapsulation
Inheritance
Polymorphism

The first part of the discussion is
Life before Objects

Using Parallel Arrays

	String[ ] title	String[ ] author	int[ ] date	double[ ] price	int[ ] inStock
0	ULYSSES	James Joyce	1918	38.95	25
1	THE GREAT GATSBY	F. Scott Fitzgerald	1925	30.95	27
2	BRAVE NEW WORLD	Aldous Huxley	1931	33.95	10
3	TO THE LIGHTHOUSE	Virginia Woolf	1927	34.95	32
4	THE SOUND AND THE FURY	William Faulkner	1929	38.95	43
5	CATCH-22	Joseph Heller	1961	30.95	49
6	THE DEATH OF THE HEART	Elizabeth Bowen	1938	20.95	7
7	DARKNESS AT NOON	Arthur Koestler	1940	39.95	26
8	THE GRAPES OF WRATH	John Steinbeck	1939	17.95	36
9	1984	George Orwell	1949	12.95	27

Back to my company, dan-azon. I need a computerized way of keeping track of inventory.

I could use a collection of parallel arrays to store inventory information about the books. I have an array of strings for the title, another array of strings for the author, an int array for the date published, a double array for the price and another int array for the number of books in stock for each title.

Although I could process my inventory using this method, it would be much better to have all of the information about an individual book grouped together rather than all of the titles grouped together, authors grouped together, and so on.

Having the parallel arrays also makes it messy and things can get unorganized if I end up adding or removing a title in the middle of the multiple arrays.

A Structured Record in C

```c
struct S_BOOK {
    char title[30];
    char author[20];
    int date;
    double price;
    int inStock;
};
typedef struct S_BOOK Book;
```

```c
typedef struct S_BOOK {
    char title[30];
    char author[20];
    int date;
    double price;
    int inStock;
} Book;
```

The discussion of classes and objects it all starts with the C-language and how it has a better way of organizing data in something called **struct**. This is short for structured record. Some languages just call it **Record**. A record is defined with the **struct** statement and can have several **fields** of different data types. The **typedef** statement is used to create a new data type **Book** from the **struct** statement to create the data structure and the new data type at the same time..

Using a Structured Record in C

```
Book myBook = {"Dan the Programmer", "Dan McElroy", 2018, 14.95, 10};
Book secondBook;

strcpy_s(secondBook.title, 30, "Book #2");
// or use  strcpy(secondBook.title, "Book #2");
strcpy_s(secondBook.author, 20, "Dan McElroy");
// or use  strcpy(secondBook.author, "Dan McElroy");
secondBook.date = 2018;
secondBook.price = 20.49;
secondBook.inStock = 25;

printf("My first book is: %s\n", myBook.title);
printf("My next book is:  %s\n", secondBook.title);
```

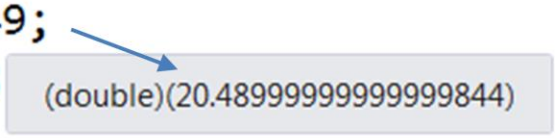
Now that we have a structured record defined in C, here is how we can put data into the record and access the data that is in the record. First, name the record and then use the 'dot-operator' to identify the field. You can either use literals to fill the record when it is being created, or you can fill the record when the program is running.

myBook is defined using literals when the record is created. **secondBook** is only defined at the top and then data is put into the record later. NOTE: some compilers require the use of the secure version of **strcpy_s**, other compilers use the original **strcpy**.

The printf statements show the use of the dot-operator when displaying the title of the two books.

Side note on the double data type

```
secondBook.title = "Book #2";
secondBook.author = "Dan McElroy";
secondBook.date = 2018;
secondBook.price = 20.49;
secondBook.inStock = 25
```



Just as a side note on the double data type. The double data type does a great job when converting from an integer to a double without digits past the decimal place. We need to remember that the digits past the decimal are a very close approximation because numeric values are stored in binary. There is not always an exact conversion from decimal to binary of the digits past the decimal. When I looked at how 20.49 was stored in memory, it came out as 20.48999999999999844 which is as close as it can get to 20.49. When displayed, it will be rounded to 20.49 so everything looks good even though secretly in memory it is not exactly what we specified. WARNING! Don't use the double data type in a for or while loop to count the number of times through the loop. You could be off. Use only the **int** data type to control a loop or index through an array.

OK. Back to the discussion on classes and objects.

Array of Records in C

```
Book BookList[] = {
    {"ULYSSES", "James Joyce    ", 1918, 32.95, 16 },
    {"THE GREAT GATSBY", "F. Scott Fitzgerald   ", 1925, 13.95, 30 },
    {"BRAVE NEW WORLD", "Aldous Huxley", 1931 , 14.95, 28 },
    {"TO THE LIGHTHOUSE", "Virginia Woolf", 1927, 36.95, 19 },
    {"THE SOUND AND THE FURY", "William Faulkner", 1929, 17.95, 11 },
    {"CATCH-22", "Joseph Heller", 1961, 38.95, 25 },
    {"THE DEATH OF THE HEART", "Elizabeth Bowen", 1938, 26.95, 44 },
    {"DARKNESS AT NOON", "Arthur Koestler", 1940, 39.95, 6 },
    {"THE GRAPES OF WRATH", "John Steinbeck", 1939, 12.95, 32 },
    {"1984", "George Orwell", 1949, 24.95, 24 }
};
int bookCount = sizeof(BookList) / sizeof(Book);
```

An array of structured records can also be declared and even initialized at the same time. C does not keep track of the size of an array as part of the language itself. The number of records can be determined by getting the size of the array in bytes and dividing it by the size of each record in bytes.

```
int bookCount = sizeof(BookList) / sizeof(Book);
```

Array of Records in C

```
Book BookList[] = {
    {"ULYSSES", "James Joyce    ", 1918, 32.95, 16 },
    {"THE GREAT GATSBY", "F. Scott Fitzgerald   ", 1925, 13.95, 30 },
    {"BRAVE NEW WORLD", "Aldous Huxley", 1931 , 14.95, 28 },
    {"TO THE LIGHTHOUSE", "Virginia Woolf", 1927, 36.95, 19 },
    {"THE SOUND AND THE FURY", "William Faulkner", 1929, 17.95, 11 },
    {"CATCH-22", "Joseph Heller", 1961, 38.95, 25 },
```

```
// display books from the array that are in stock and their price and title
printf("%3s  %7s  %-30.30s\n", "Num", "Price", "Title"); // top of table
printf("-------------------------------------\n");
for (int i=0; i<bookCount; i++)
    printf("%3d  %7.2f  %-30.30s\n",
        BookList[i].inStock, BookList[i].price, BookList[i].title);
```

Here is a **for** loop that displays the number of items in stock, price and title of each book in the array. Not all of the fields are being displayed by the printf statement, and not even in the same order as when the record was created. **printf**'s format statement shows %3d to print an integer three spaces wide for the **inStock** field. It has %7.2f to print the price using 7 spaces on the display with 2 digits past the decimal, and %-30.30s to print a string that is left justified (see the minus sign) that is 30 spaces wide on the display.

BookList[i].inStock is part of the for loop. BookList[i] identifies the record, followed by the dot-operator and the the field identifier.

Array of

```
Book BookList[] = {
    {"ULYSSES", "James Joyce    ",
    {"THE GREAT GATSBY", "F. Scott
    {"BRAVE NEW WORLD", "Aldous Hu
    {"TO THE LIGHTHOUSE", "Virgini
    {"THE SOUND AND THE FURY", "Wi
// display books from the array tha
printf("%3s  %7s  %-30.30s\n", "Num
printf("--------------------------
for (int i=0; i<bookCount; i++)
    printf("%3d  %7.2f  %-30.30s\n"
        BookList[i].inStock, BookLi
```

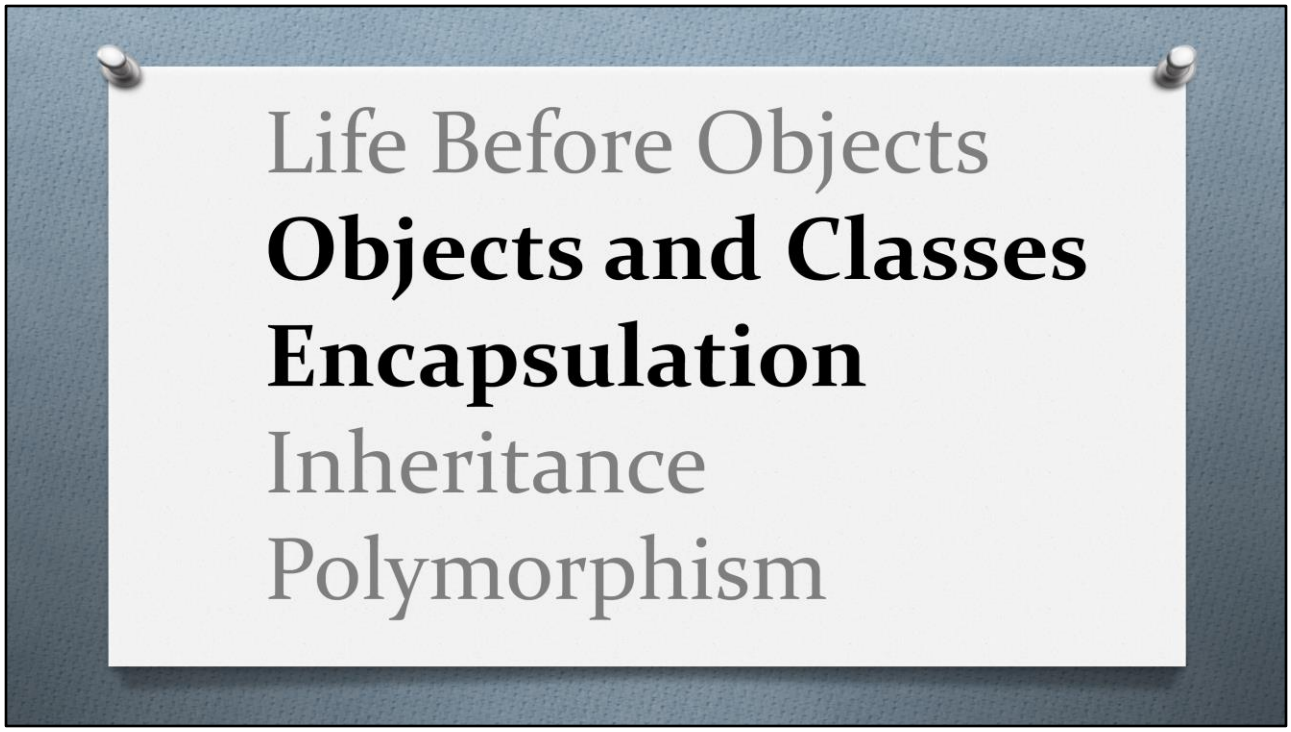
```
My first book is: Dan the Programmer
My next book is:  Book #2

Num  Price  Title
------------------------------------
 16  32.95  ULYSSES
 30  13.95  THE GREAT GATSBY
 28  14.95  BRAVE NEW WORLD
 19  36.95  TO THE LIGHTHOUSE
 11  17.95  THE SOUND AND THE FURY
 25  38.95  CATCH-22
 44  26.95  THE DEATH OF THE HEART
  6  39.95  DARKNESS AT NOON
 32  12.95  THE GRAPES OF WRATH
 24  24.95  1984
```

http://program-info.net/C++/ClassesAndObjects/downloads/C-BookList.c

Here is a sample output from the program.

Code for the C program can be found at:
program-info.net/C++/ClassesAndObjects/downloads/C-BookList.c
Watch the capitaliztion



Part 2
Objects, classes and encapsulation

Definitions of Some Terms

- Class
- Object
- Instance
- Field, attribute or property
- Method (subroutine, function)
- Constructor

I want to start with some definition of terms before discussing objects in any more detail, starting with class and object.
A **class** statement is more like a blueprint of a building. It says how the building is to be constructed. The class statement does not allocate any memory on its own. An **object** is a thing. Comparing blueprints and buildings, an object is like the building. Once a class has been defined, we can have as many objects based on the class as we want.

There are many times in programming that we take English words and give them new meanings. An object is an **instance** of a class. The verb **to instantiate** means making an object from a class definition. **Instantiation** is the process of making objects. You may hear these words tossed around

when people are talking about object oriented programming, so it is a good to have an idea of what they mean.

One of the biggest goals in object oriented programming is to hold all the data and the code that works on it all in one piece. This will be the object. The data that belongs to an object is referred to by several names, all of which mean the same thing: f**ield, attribute, property.** These can be things like integers, doubles, characters, strings or even other objects. The code that belongs to an object is called its method. Methods can either be like **functions** which return a piece of data or a **subroutine/procedure** that does not return anything.

Constructor

A **constructor** is a special type of subroutine called to create an object. It prepares the new object for use, often accepting arguments that the constructor uses to set required member variables.

A constructor resembles an instance method, but it differs from a method in that it has no explicit return type. Constructors have the same name as the declaring class. They have the task of initializing the object. A properly written constructor leaves the resulting object in a *valid* state.

Java allows overloading the constructor in that there can be more than one constructor for a class with differing parameters.

en.wikipedia.org

Here is a definition of a constructor from en.wikipedia.org

A **constructor** is a special type of subroutine called to create an object. It prepares the new object for use, often accepting arguments that the constructor uses to set required member variables.

A constructor resembles an instance method, but it differs from a method in that it has no explicit return type. Constructors have the same name as the declaring class. They have the task of initializing the object. A properly written constructor leaves the resulting object in a *valid* state.

Java allows overloading the constructor in that there can be more than one constructor for a class with differing

parameters.

Constructors in some other languages go by the name **New.**

Constructor

A **constructor** is a special type of subroutine called to create an object. It prepares the new object for use, often accepting arguments that the

```
Automobile ( ) {      // default constructor
 }

 Automobile(string n, int y) {   // constructor with two arguments
    name = n; // copy from the argument list to object's name field
    year = y;   // copy from the argument list to object's year field
  }
```

rom a method
ame name as
ct. A properly
e.

more than one

en.wikipedia.org

There is always a **default constructor** even if you don't declare it yourself. The default constructor has no arguments. For example if the name of the class is Automobile, then the default constructor would be: Automobile ( ) followed by an open and close curly-brace to hold any code.

Since methods can be overloaded with different arguments, additional constructors can be written that accept data that typically is used to initialize the data fields that are part of the object. Example:

```
   Automobile(string n, int y) {
      name = n; // copy from the argument list to object's
name field
      year = y;   // copy from the argument list to object's year
```

field
};
I
If you are creating a constructor that has arguments, you also should create a default constructor, a copy constructor and a destructor method. The copy constructor and destructor are covered much later.

Encapsulation

Encapsulation is one of the fundamentals of OOP (object-oriented programming). It refers to the bundling of data with the methods that operate on that data. Encapsulation is used to hide the values or state of a structured data object inside a class, preventing unauthorized parties direct access to them. Publicly accessible methods are generally provided in the class (so-called getters and setters) to access the values, and other client classes call these methods to retrieve and modify the values within the object.

en.wikipedia.org

Here is a definition of encapsulation from Wikipedia. Encapsulation is one of the fundamentals of OOP (object-oriented programming). It refers to the bundling of data with the methods that operate on that data. Encapsulation is used to hide the values or state of a structured data object inside a class, preventing unauthorized parties direct access to them. Publicly accessible methods are generally provided in the class (so-called getters and setters) to access the values, and other client classes call these methods to retrieve and modify the values within the object.

Location of the Java and C++ Source Files

I recommend that you get a copy of the Java source files used in this discussion. C++ files are provided for your reference and enjoyment.
1) **Book.java** or **Book.h -** the definition for the class
2) **JavaBookList.java** or **CppBookList.cpp** - contains main( ), creates and tests Book

Java Files:
http://program-info.net/Java/downloads/ClassesAndObjects/Version1/

C++ Files:
http://program-info.net/C++/downloads/ClassesAndObjects/Version1/Book.h
http://program-info.net/C++/downloads/ClassesAndObjects/Version1/CppBookList.cpp

From this point on, I will be discussing only how Java works with objects. There is another video almost identical to this one for C++ that is very similar. There are many ways these two languages treat objects the same, but there are several ways that they are very different.. I am providing the equivalent code for both languages in case you want to compare the differences between them. I suggest that you at least get a copy of the Java code and follow along with the discussion.

The Two Files

Book.java

```
// Book.java - class definition for Book

package javabooklist;

public class Book {

    public String title;
    public String author;
    public int date;
    public double price;
    public int inStock;

    // default constructor
    Book ( ) {
        // memory is allocated for the object but
        // nothing else happens here
    };

    // constructor with five arguments
    Book (String t, String a, int d, double p, int i) {
        title = t;
        author = a;
        date = d;
        price = p;
        inStock = i;
    }; // end of five argument constructor

}; // end of class Book
```

The definition for **class Book** is in Book.java

JavaBookList.java

```
public class JavaBookList {

    public static void main(String[] args) {

        Book myBook = new Book(1983, "Dan the Programmer", "Dan McElroy", 2018, 14.95, 10);
        Book secondBook = new Book();

        secondBook.title = "Book #2";
        secondBook.author = "Dan McElroy";
        secondBook.date = 2018;
        secondBook.price = 20.49;
        secondBook.inStock = 25;

        System.out.println ("My first book is: " + myBook.title);
        System.out.println ("My next book is:  " + secondBook.title);
        System.out.println ( ); // blank line
    } // end of main
} // end of class
```

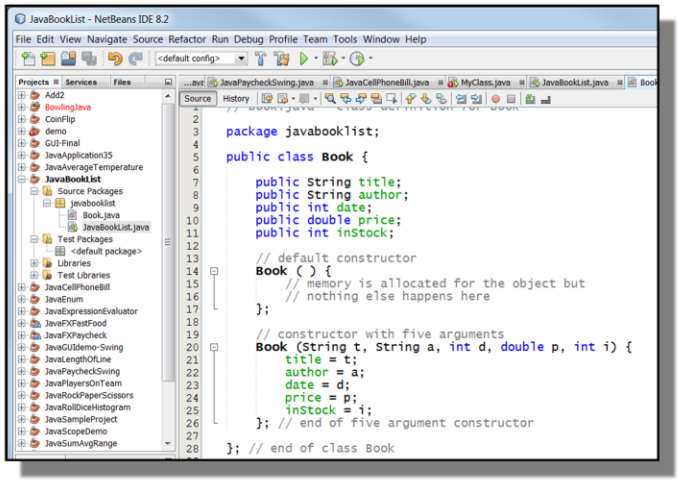
main( ) is located in **JavaBookList.java** and will use **class Book** to build Book objects.

In Java, every class needs to be in its own file. The name of the file must be the same as the class with the **.java** file extension. Even the capitalization of the class name and the file name must match.

The definition for **class Book** is in Book.java

main( ) is part of **class JavaBookList** and is in the file JavaBookList.java.
main( ) uses **class Book** to build Book objects.

Where Does the Class Definition Go?



At this point, our programs will still be fairly small, so put all of the files for the project in the same directory. In NetBeans or Eclipse, click File/New to make a new file as part of the project.

class Book

```
// Book.java - class definition for Book
package javabook;

public class Book {

    public String title;
    public String author;
    public int date;
    public double price;
    public int inStock;

    // default constructor
    Book ( ) {
        // memory is allocated for the object but
        // nothing else happens here
    };

    // constructor with five arguments
    Book (String t, String a, int d, double p, int i) {
        title = t;
        author = a;
        date = d;
        price = p;
        inStock = i;
    }; // end of five argument constructor

}; // end of class Book
```

The access modifier **public** is used for each field definition to give the main program direct access to each of these pieces of data. Later on, the access to the data fields will be made **private** to protect them from being directly accessed or modified. The constructors must remain public.

Two constructors are provided. The default constructor and the constructor that has five arguments.
The five argument constructor receives data as parameters from the part of the program that created the object. This constructor is used to get the data from the main program and store it in the objects own copy of the data. The parameters are named t, a, d p and i. These pieces of data are copied into the object's data fields title, author, date, price and inStock.

Constructors are special. They are only called by the **new** operator. Constructors can't have a return data type or be declared static.

Creating and Accessing Objects

```
1   package javabooklist;
2
3   public class JavaBookList {
4
5       public static void main(String[] args) {
6
7           Book myBook = new Book("Dan the Programmer", "Dan McElroy", 2018, 14.95, 10);
8           Book secondBook = new Book();
9
10          secondBook.title = "Book #2";
11          secondBook.author = "Dan McElroy";
12          secondBook.date = 2018;
13          secondBook.price = 20.49;
14          secondBook.inStock = 25;
15
16          System.out.println ("My first book is: " + myBook.title);
17          System.out.println ("My next book is:  " + secondBook.title);
18
19      } // end of void main( )
20
21  } // end of class
```

Although the listing from the web for JavaBookList.java has an array of objects, let's start slow and just look at how an object is created and its data can be directly accessed.

Line 7 creates a Book object named **myBook** using the constructor that has five arguments. This constructor copies all of the data provided as parameters into the individual data members for the myBook object.

Line 8 creates a Book object named **secondBook** using the default constructor.

If we look back at the code for the default constructor, it does not do anything. Java only allocates the memory for the object. It is up to the main( ) program to place

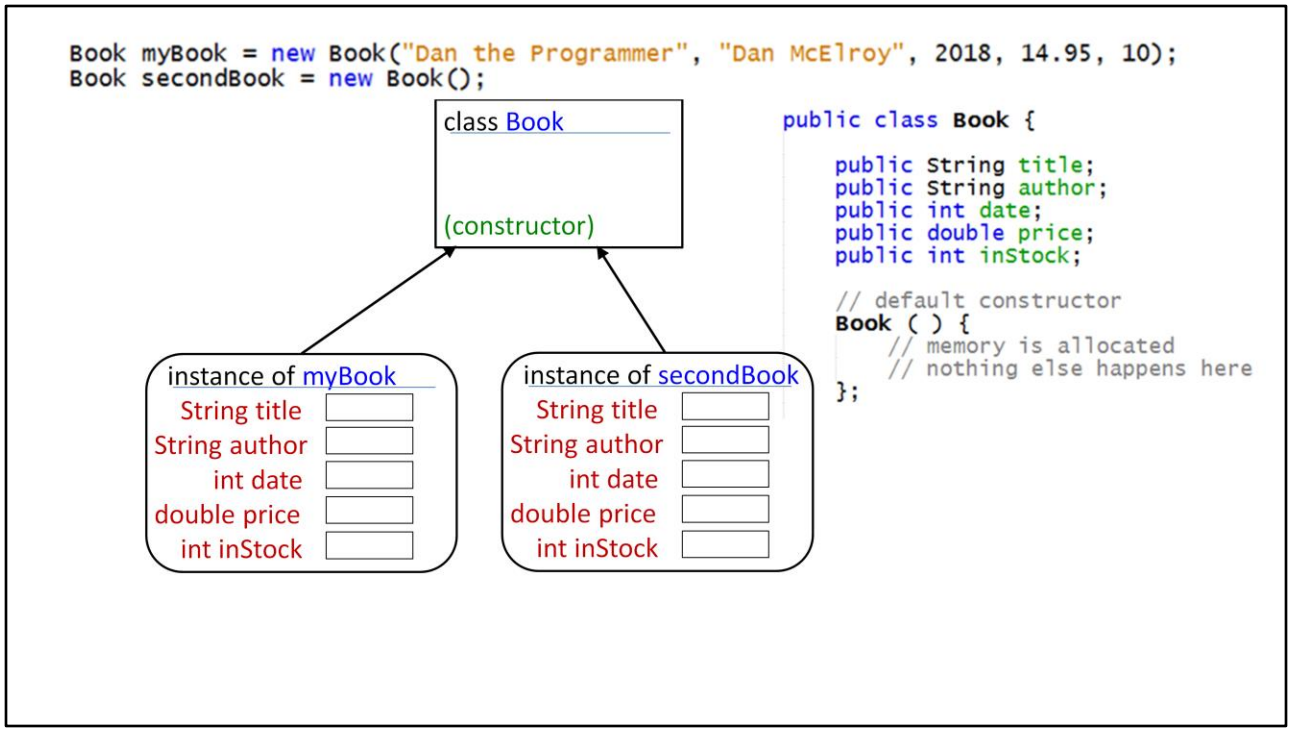
something in the secondBook object's data members. This is done on lines 10 through 14.

Line 10 says **secondBook.title = "Book #2";** The string literal "Book #2" is copied into the **title** data member of SecondBook by the assignment operator. secondBook.title uses the dot-operator to identify both the name of the object and its data member.

main( ) is able to directly access each of the data members of a Book object because they all have their access modifiers set to **public** in the class definition for Book. If the access modifier were set to **private** then main( ) could not directly access these data members.

The println statements on lines 16 and 17 also directly access the **.title** data members of the objects myBook and secondBook.

The other data members such as author, date, etc. had data placed in them but they were not used.



When an object is created, each object gets its own copy of the non-static data members. In this example, two objects are created from the class definition. Each object gets its own copy of title, author, date, price and inStock.

Information about Constructors

A constructor call is more complicated than an ordinary subroutine or function call. It is helpful to understand the exact steps that the computer goes through to execute a constructor call:

1. First, the computer gets a block of unused memory in the heap, large enough to hold an object of the specified type.
2. It initializes the instance variables of the object. If the declaration of an instance variable specifies an initial value, then that value is computed and stored in the instance variable. Otherwise, the default initial value is used.
3. The actual parameters in the constructor, if any, are evaluated, and the values are assigned to the formal parameters of the constructor.
4. The statements in the body of the constructor, if any, are executed.
5. A reference to the object is returned as the value of the constructor call.

From ***Introduction to Programming Using Java***, by David J. Eck Section 5.2.2

Here is some information from ***Introduction to Programming Using Java***, by David J. Eck, Section 5.2.2

A constructor call is more complicated than an ordinary subroutine or function call. It is helpful to understand the exact steps that the computer goes through to execute a constructor call:

1. First, the computer gets a block of unused memory in the heap, large enough to hold an object of the specified type.
2. It initializes the instance variables of the object. If the declaration of an instance variable specifies an initial value, then that value is computed and stored in the instance variable. Otherwise, the default initial value is used.

3. The actual parameters in the constructor, if any, are evaluated, and the values are assigned to the formal parameters of the constructor.
4. The statements in the body of the constructor, if any, are executed.
5. A reference to the object is returned as the value of the constructor call.

When Does the Object Get Memory?

```
1   package javabooklist;
2
3   public class JavaBookList {
4
5       public static void main(String[] args) {
6
7           Book myBook = new Book("Dan the Programmer", "Dan McElroy", 2018, 14.95, 10);
8           Book secondBook = new Book();
9
10  // create reference and object's memory separately
11  Book secondBook; // create the reference
12  secondBook = new Book( ); // create the object
13
14
15  // create the reference and object at the same time
16  Book secondBook = new Book( );
17
18
19
20
21  } // end of class
```

IMPORTANT !!! - Creating an object is really a three-step process

1) Create the reference that will hold the memory address of the object

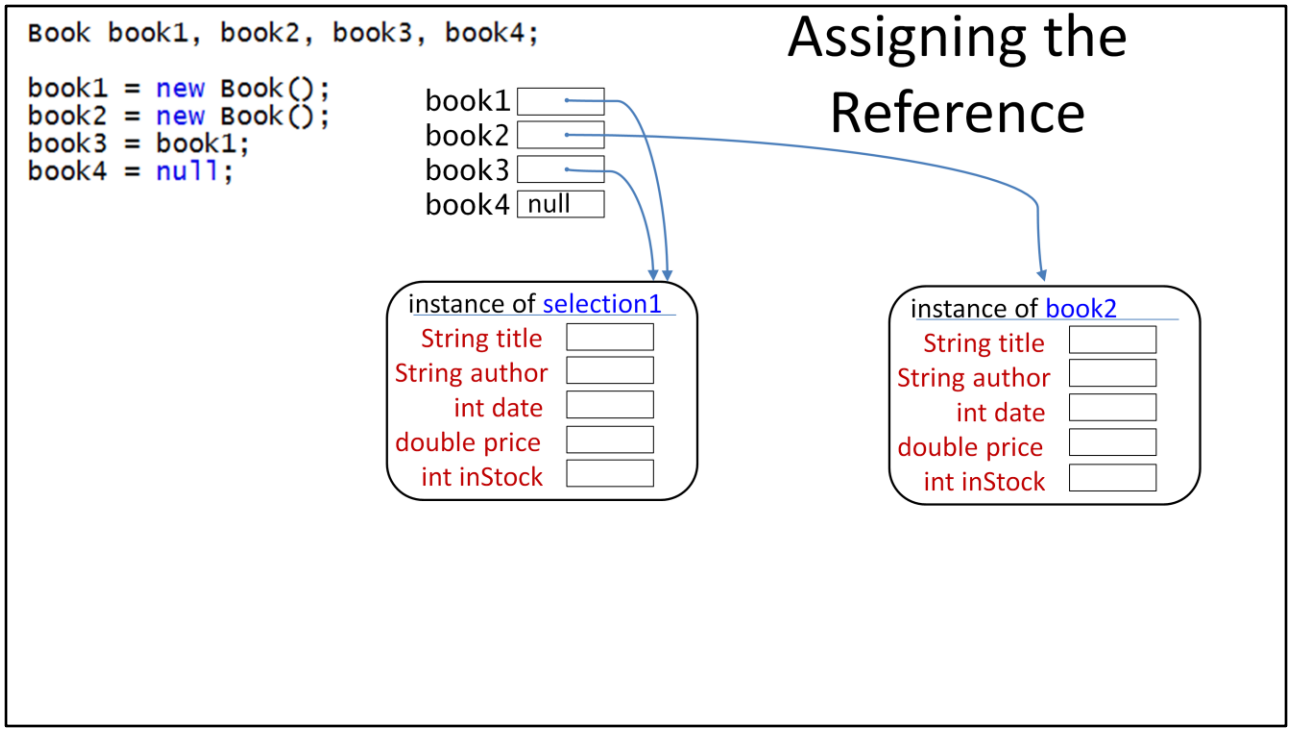
2) Use the **new** operator to obtain memory from the 'heap' that can be used by the object and call the object's constructor. The **new** operator returns the memory address of the object that was created and calls the constructor.

3) Use the assignment operator = to place the memory address of the object into the reference from step 1

Steps 1, 2 and 3 can all be combined on one line
Book secondBook = new Book( );

secondBook is a reference to the object instead of the

object itself but our Java program can treat it as though it were the object. Let's look at a diagram of this process.



Four objects are created from **class Book** in this example. These object variables will hold the address, or reference, to the actual memory location if the **new** operator is called. Book book1, book2, book3, book4;

The **new** operator allocates some memory for book1 and book2 and calls the default constructor. The **new** operator returns the address of memory where the objects are actually located. The memory address for the first call to **new** is stored in book1. The memory address for the second call to **new** is stored in book2.

The line **book3 = book1**; does not copy the contents of book1 into book3 because no object has been created for book3. What actually happens is the memory address that is

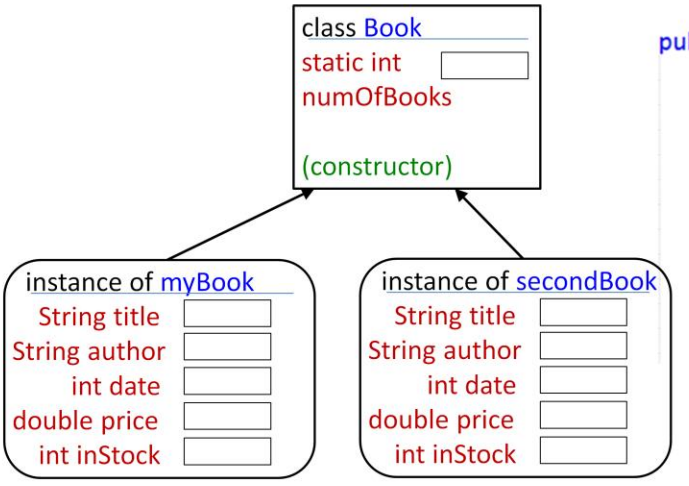
stored in the **book1** variable is copied into the variable for **book3.** Now there are two references for the object book1. We just have two different names for the same object.

If you wanted to see if the contents of book1 and book2 were the same, if(book1 == book2) would not work because you would be comparing the memory addresses of these two different objects, not the contents of the objects. You would need to compare the contents of each member data to see if they were equal.

Strings are objects. Suppose you have created two Strings, string1 and string2, you can't use **if (string1 == string2)** to see if the contents of the strings are the same. However, String has member functions **.equals( )** and **.equalsIgnoreCase( )** that return a boolean true if the contents of the strings are the same. This would work **if (string1.equals(string2))**

If you are clever, you can write a member method called **.equals( )** as part of your own class definition that would compare each data member between two objects and return **true** if everyone of them are the same.

static Data Members




```
public class Book {

    public String title;
    public String author;
    public int date;
    public double price;
    public int inStock;
    static int numberOfBooks;

    // default constructor
    Book ( ) {
        numberOfBooks++;
    };
```

If each constructor increments **numberOfBooks**, then we have a total of all the Book objects.

When an object is created, each object gets its own copy of the non-static data members. A new data member has just been added to **class Book** by the name of **numberOfBooks**. This data member is not part of the individual objects, but is shared by by all of the objects.

static Data Members

```
// Book.java - class definition for Book

package javabooklist;

public class Book {

    public String title;
    public String author;
    public int date;
    public double price;
    public int inStock;
    static int numberOfBooks;

    // default constructor
    Book ( ) {
        numberOfBooks++;
    };

    // constructor with five arguments
    Book (String t, String a, int d, double p, int i) {
        title = t;
        author = a;
        date = d;
        price = p;
        inStock = i;

        numberOfBooks++;
    }; // end of five argument constructor

}; // end of class Book
```

```
package javabooklist;

public class JavaBookList {

    public static void main(String[] args) {

        Book myBook = new Book("Dan the Programmer", "Dan McElroy", 2018, 14.95, 10);
        Book secondBook = new Book();

        System.out.printf ("There are %d books\n", Book.numberOfBooks);

        secondBook.title = "Book #2";
        secondBook.author = "Dan McElroy";
        secondBook.date = 2018;
        secondBook.price = 20.49;
        secondBook.inStock = 25;

        System.out.println ("My first book is: " + myBook.title);
        System.out.println ("My next book is:  " + secondBook.title);
        System.out.println ( ); // blank line

    } // end of main( )

} // end of class
```

Output when the program runs

```
There are 2 books
My first book is: Dan the Programmer
My next book is:  Book #2
```

The main program creates two **Book** objects, **myBook** and **secondBook**.

The first book, **myBook**, initializes each of its object data members through the five-argument constructor. The title, author, date, price and inStock parameters are passed through the five-argument constructor which then loads them into the data members for the object.

secondBook creates the object using the default constructor that does not have an argument list through a call to **new Book( );** The default constructor allocates memory to hold data for **secondBook** but it only increments the **numberOfBooks** variable. The other data members are left uninitilized.

The object data members for **secondBook** are initialized in the main( ) part of the program using the name of the object, **secondBook** followed by the dot operator and then the data member's name - title, author, date, price, inStock. This can be done because each of the data members defined in the class are declared with the public access modifier.

Look at how **static int numberOfBooks** is declared. If a data member is declared static in the class, then there is only **one** variable that is shared among all objects. This variable is incremented inside the constructor each time an object is created. Now the program can keep track of the number of books. The main( ) program has a System.out.printf statement that displays the number of books. Note, instead of selecting an object member by the objectName.fieldName, the static data member is selected using the className.fieldName

```
System.out.printf ("There are %d books\n",
Book.numberOfBooks);
```

When the program runs, the output from the three printf statements display

```
There are 2 books
My first book is: Dan the Programmer
My next book is:  Book #2
```

Setters and Getters

```
String setTitle(String t)      String getTitle()
String setAuthor(String a)     String getAuthor()
int setDate(int d)             int getDate()
double setPrice(double p)      double getPrice( )
int setInStock(int s)          int getInStock( )
```

Giving all of the data members **public** access modifiers provides no protection at all to the data. The main( ) method or any other part of the program has full access to these data members and can change them any way they want. It is much better to make the data members **private** so that they can only be accessed by methods that belong to the class.

For example, if we were creating a class that is used in computing a weekly paycheck for an hourly employee, the absolute maximum number of hours in a week is 168. (7 days * 24 hours). It also would be good to make sure that the hourly rate is not less than the legal minimum rate or that it exceeds some limit imposed by the company.

If the data members are **private** instead of **public** we need member methods to give access to the data. These methods are referred to as getters and setters. Some people prefer the words **accessors** and **mutators** instead of getters and setters.

Going back and using the **Book** class, we can provide the following getters and setters:

String getTitle() String setTitle(String newTitle)
String getAuthor() String setAuthor(String newAuthor)
int getDate() int setDate(int newDate)
double getPrice( ) double setPrice(double newPrice)
int getInStock( ) int setInStock(int newInStock)

Sample Getter and Setter

```
// getter for the price
public double getPrice( ) {
    return price;
} // end of getPrice( )

// setter for the price
public double setPrice (double p) {
    if (p < 0) {
        price = 0.00;
        System.out.println ("Negative price not allowed");
    }
    else if (p > MAX_PRICE) {
        price = MAX_PRICE;
        System.out.println ("Attempted to set pice too high");
    }
    else
        price = p;
    return price;
} // end of setPrice( )
```

Here are sample getters and setters for the price data member.

The **public double getPrice( )** member method just uses the return statement to return the value from the object's price data member back to the main program.

The **public double setPrice (double p)** method is a little more complex. There is only one argument, **double p** which is sent by main( ) when the setter is called. The value in **p** should be placed in the object's **price** member data, but first it is checked to see if it is greater or equal to 0 and less or equal to the **MAX_PRICE** that in our case is defined at the top of the class. We are now providing some protection and an error message. It might even be nicer if the error message also identified the title of the book.

It is best to create the getters and setters when the class is first defined.

Naming Conventions

```
// getter for the price
public double getPrice( ) {
    return price;
} // end of getPrice( )

// setter for the price
public double setPrice (double p) {
    if (p < 0) {
        price = 0.00;
        System.out.println ("Negative price not allowed");
    }
    else if (p > MAX_PRICE) {
        price = MAX_PRICE;
        System.out.println ("Attempted to set pice too high");
    }
    else
        price = p;
    return price;
} // end of setPrice( )
```

Some advanced features of Java require that the getters and setters start with the words **get** and **set** followed by the name of the member data name with the first letter capitalized.

The example shown in this code segment shows how the getter and setter are named for the data field **price**.

getPrice( ) // is the name of the method that returns the contents of the price data member

setPrice( ) // is the name of the method that accepts the price from the main program and stores it into the data field **price**

In the case of boolean data fields, the following getters and setters could be named:

booloean hardCover; // true or false data field

isHardCover( ) // since data field is boolean, use the prefix **is-** instead of get

setHardCover( ) // still use the prefix **set-** for a boolean data field

void return data type

```
private int count; // class data member
public  int  setCount( int c ) {
   count = c;
   return count;
}
```

```
private int count;  // class data member
public void setCount( int c ) {
   count = c;
   return;   // not needed, can't return anything
}
```

Setter methods can either be coded with a return statement that returns data back to the main program, or they can be coded with the **void** return data type. The **void** return data type means that nothing is returned. In this example, there is a return statement but it is not needed. The method does an automatic return when the closing curly-brace is reached. Since the method is declared void, even if a return statement is provided, it can't return any data.

The only advantage to having a method return something is that the setter method can be used as part of an expression in the main program.

Naming Method Argument

OPTION 1

```
private int count; // class data member
public  int  setCount( int c ) {
   count = c;
   return count;
}
```

BAD OPTION

```
private int count; // data member
public  int  setCount( int count ) {
   count = count;  // won't work !!!
   return count;
}
```

Assume that these pieces of code are part of a class definition. There is a field named count that needs to be either initialized by a constructor or a setter method.

The integer parameter that the method setCount is receiving is named **c**, which is then placed into the count data member. It would be nice to name the parameter in the method definition **count** because this would make it much easier to understand, but it would not work to say **count=count;** because they are named the same thing.

this pointer

OPTION 2

```
private int count;  // class data member
public  int  setCount( int count ) {
   this.count = count;
   return  this.count;
}
```

The way to get around this is to let the method argument name be the same as the private data member, but use the **this** pointer to specify the class data member.

this can be used to specifically identify the current object and all of its data and methods. Now we have two things both named **count**. The argument parameter is named count and the private data member is named count. By using **this.count = count;** **this.count** identifies the data member and **count** all by itself identifies the data arriving through the method's parameter list.

The **this** operator can be used other times in code that belongs to the class definition anytime that the the object needs to refer to itself. Some OOP languages use the word

self instead of **this.**

Have Constructors Use the Setters

```
// constructor with five arguments
Book (String t, String a, int d, double p, int i) {
    title = t;
    author = a;
    date = d;
    price = p;
    inStock = i;
}; // end of five argument constructor

// setter for the price
public double setPrice (double p) {
    if (p < 0) {
        price = 0.00;
        System.out.println ("Negative price not allowed");
    }
    else if (p > MAX_PRICE) {
        price = MAX_PRICE;
        System.out.println ("Attempted to set pice too high");
    }
    else
        this.price = p;
    return price;
} // end of setPrice( )

// getter for the price
public double getPrice( ) {
    return price;
} // end of getPrice( )
```

This code from class Book has two methods that set the price. One is the five-argument constructor and the other is the setter **setPrice(double p)**

The constructor is called when an object is first instantiated (created). The **setPrice** can be called within the main( ) program to update the price after the object has been created.

The constructor just sets the price without doing any checking to verify that the value is legal. The **setPrice( )** method has code to check for a legal price. You could copy the code from **setPrice( )** into the constructor, but it would be easier and much better to just have the constructor call the setPrice( ) method. Then the code would be only in one

place if it ever needed to be updated.

Change the constructor from:

```
    price = p;
```

to

```
    setPrice(p);
```

When memory is allocated for an object by the **new** operator, a block of memory is reserved from the heap. The starting address of the memory block is returned by the new operator and can be stored in the reference. For example, **Book selection1 = new Book( );** enough memory is allocated on the heap to store one object. The address of the first byte of memory is returned as a **reference** by the **new** operator and assigned/stored into the variable selection1.

selection1 is a local variable that exists within a set of curly-braces { } and no longer exists when the code within the set of curly braces can be executed. The Java run time executive (JRE) has a garbage collection routine that reclaims memory occupied by objects that are no longer accessible to the

program. The JRE keeps track of the number of references there are to an individual object. When that number reaches zero, the memory allocated to the object is reclaimed.

The memory allocated to selection1 can be reclaimed, assuming that there are no additional references to the object.

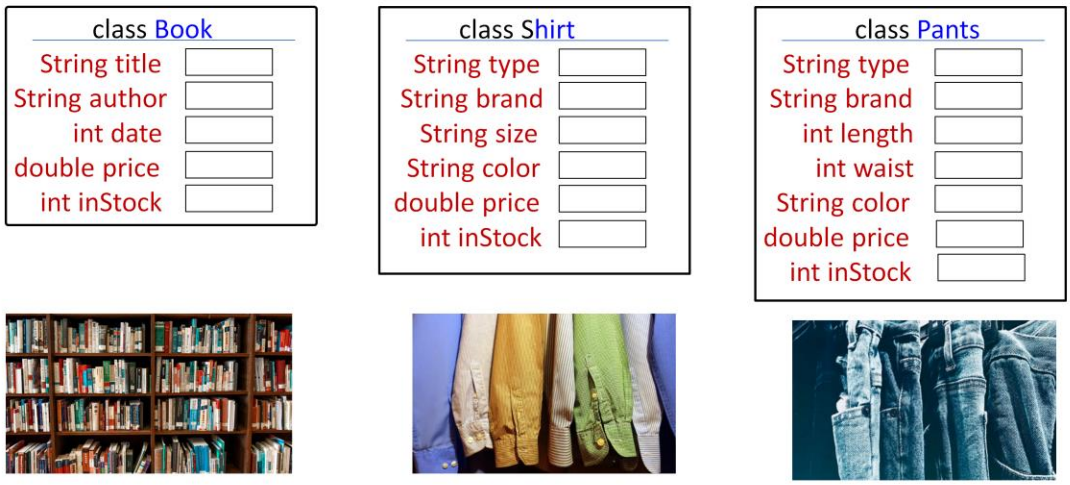
Garbage collection is automatic with Java. The programmer does not need to worry about it. Many other programming languages do not have garbage collection. If the programmer deletes an object even though there are still references to it, this is called a **dangling pointer error**. If a programmer forgets to delete an object when it is no longer used, this causes a **memory leak**. If too many memory leaks occur, the operating system runs out of heap space and the system crashes.

Life Before Objects
Objects and Classes
Encapsulation
Inheritance
Polymorphism

Introducing INHERITANCE

My dan-azon business is growing. I am ready to start selling clothing as well as books. I am going to start with shirts and pants. and I need to upgrade the way to store the inventory data. I already have a class for Books, now I need one for clothes.

Creating More Classes

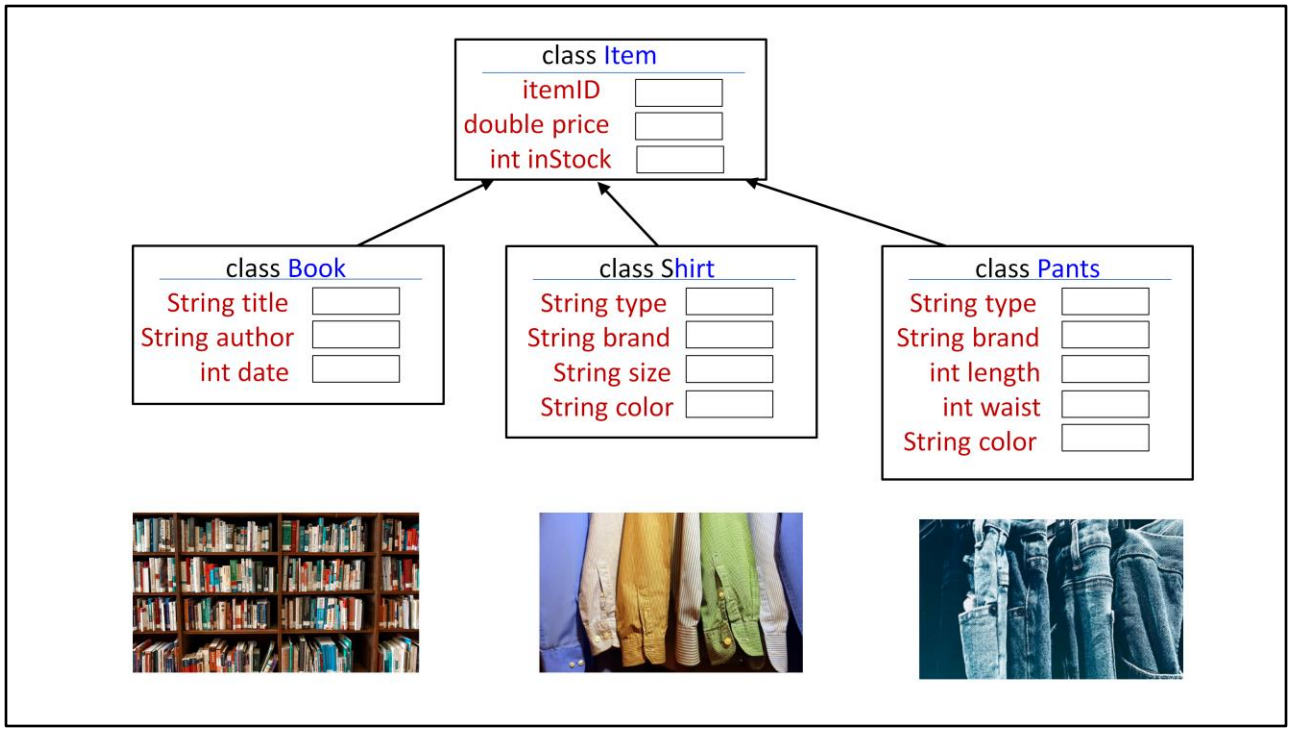


I created additional class definitions: **class Shirt** and **class Pants**. The field **type** in the class Shirt could identify if it is a T-shirt, a long sleeve shirt, a blouse, etc. The field **type** in the class Pants could be used to identify dress pants, jeans, shorts, slacks, etc.

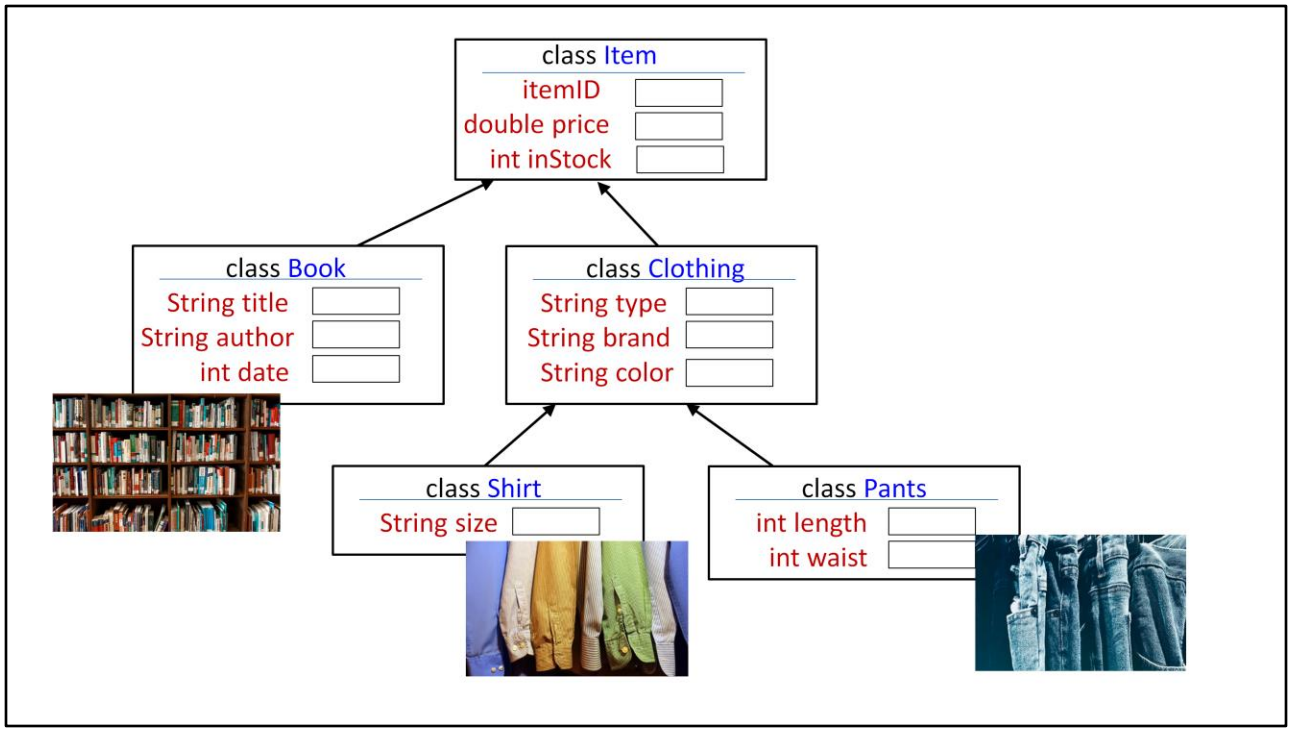
I realize that in the real world more information may need to be added, but for the discussion this is what I am providing. I also need to write the code for the methods to process each field in each of the classes.

One of the thing that I noticed is that there is some duplication of fields. Every one of them has a field for price and inStock. I would also need to duplicate any code in each class that references these fields such as methods for

getPrice, setPrice, getInStock and setInStock.



To get things more organized, I am providing a **super class** called Item that has the fields for price and inStock. Each of the other classes can 'inherit' everything from class Item and then just provide the additional fields that are needed. When I look even closer, I can see that the class definition for the clothing items also share several fields such as the type of clothes, brand and color.



Here is an expanded diagram showing the relationships for class Item, class Book, class Clothing, class Shirt and class Pants. Thinking of this diagram as an upside down tree, **Item** can be considered a root node, **Clothing** is a branch, and **Book, Shirt** and **Pants** are all leaf nodes because they do not have any inherited subclasses derived from them.

See how the items for Shirt and Pants are unique . These two classes can inherit everything from class Clothing and class Item. They just need to include some items of their own that relate to size, and for Pants length and waist. Since each class needs to be in its own file, this project so far would need 5 files plus one more that would have main( ) create a bunch of objects and test them.

One really great thing about having this hierarchy of classes is that if in the future I wanted to add a new field for every class and call it inventoryID, I would only need to add it to **class Item** instead of trying to find all the leaf nodes on a very large system.

Now let's see how this is implemented in Java.

Get Copies of the Java Files

Get copies of the five Java files from
http://program-info.net/Java/downloads/JavaClassesAndObjects/Version2/

The names of the files are:

- Item.java
- Book.java
- Clothing.java
- Shirt.java
- BooksAndClothesInheritance.java

Get copies of the five Java files from
http://program-info.net/Java/downloads/JavaClassesAndObjects/Version2/

The names of the files are:
Item.java
Book.java
Clothing.java
Shirt.java
BooksAndClothesInheritance.java

Item.java

```
public class Item {
    protected double price; // protected variables can be accessed by child class
    protected int inStock;

    // constructors
    Item( ) { }

    // getters
    public double getPrice( )  { return price; }
    public    int getInStock() { return inStock; }

    // setters
    public double setPrice (double price) {
        this.price=price;
        return this.price;
    }
    public int setInStock(int inStock) {
        this.inStock=inStock;
        return this.inStock;
    }

}
```

The Item class does not have many things in it. But it does have two data fields, **price** and **inStock.** Look at the access modifiers. These data fields have the **protected** access modifier instead of **public** or **private**. The **public** access modifier would let any part of the program read or write to these data fields. The **private** access modifier would only let methods in this class read or write the the data fields. The **protected** access modifier lets methods in this class and any subclass that inherits from Item also read or write to the data fields, but no other part of the program can touch them.

The getters and setters are available to be used by any object instantiated from Item, or any of its subclasses.

Book.java (part 1)

```
public class Book extends Item {
    private static final double MAX_BOOK_PRICE = 100.00;
    private String title;
    private String author;
    private int date;
    private static int bookCount = 0;

    // default constructor
    Book ( ) {
        bookCount++;    // keep track of the number of books
    };

    // constructor with five arguments
    Book (String title, String author, int date, double price, int inStock) {
        setTitle(title);
        setAuthor(author);
        setDate(date);
        setPrice(price);        // code in the superclass
        setInStock(inStock);    // code in the superclass

        bookCount++;            // keep track of the number of books
    }; // end of five argument constructor

    ///////// getters and setters //////////
    public String getTitle( )  { return title; }
    public String getAuthor( ) { return author; }
    public    int getDate( )   { return date; }
    public static int getBookCount() { return bookCount; }

    // setters
    public String setTitle (String t)  { title=t; return title; }
    public String setAuthor (String a) { author=a; return author; }
    public    int setDate (int d)      { date=d; return date; }
```

class Book starts out with **extends Item** which makes it an inherited subclass of Item.

The first part of **class Book** has a constant defining the maximum price for a book, the three data fields: title, author and date, and one static field **bookCount**. Since **BookCount** is listed as static, there will only be one copy even if there are many objects instantiated with their own copies of title, author, and date, and one static field, **bookCount.**

Next come the constructors, each of which increment the static variable **bookCount.** The five argument constructor uses the setters to initialize the data fields.

Look at how the **getTitle( )** getter is written:

```
public String getTitle( ) { return title; }
```

Since there is not much code, it is written using only one line. It could also have been separated out and written on three lines. It may be easier to understand, and easier to not make typing errors if it is on three lines, but as long as you understand the syntax of how the parentheses and curly-braces are used, placing everything on one line does exactly the same thing as three lines.

```
public String getTitle( )  {
   return title;
 }
```

Book.java (part 2)

```
    @Override
    public double setPrice(double price) {
        if (price < 0) {
            this.price = 0.00;
            System.out.println ("Negative price not allowed");
        }
        else if (price > MAX_BOOK_PRICE) {
            this.price = MAX_BOOK_PRICE;
            System.out.println ("Attempted to set pice too high");
        }
        else
            this.price = price;
        return this.price;
    } // end of setPrice( )

    @Override
    public String toString() {
        return String.format ("%6.2f %s, by %s", price, title, author);
    }

}; // end of class Book
```

The **Item** already has a method named **setPrice** but the class Book wants to use a different version. The **@Override** operator is used to cause Book's setPrice( ) method to be used instead of the one i Item.

The **Book** class is considered a 'leaf' node because it does not have any subclasses derived from it.

There is a second method named **toString( )** that is also overridden. It will be discussed a little later.

Clothing.java

```
public class Clothing extends Item {
    private String type;
    private String brand;
    private String color;

    ///// getters
    public String getType ( ) { return type; }
    public String getBrand( ) { return brand; }
    public String getColor( ) { return color; }

    //// setters
    public String setType ( String type  ) {
        this.type=type;
        return this.type;
    }
    public String setBrand( String brand  ) {
        this.brand=brand;
        return this.brand;
    }
    public String setColor( String color ) {
        this.color=color;
        return this.color;
    }

    // constructors
    Clothing( ) { } // empty default constructor
    Clothing( String type, String brand, String color) {
        setType(type);
        setBrand(brand);
        setColor(color);
    } // end of three-argument constructor

} // end of class Clothing
```

The **class Clothing** also extends **class Item** and is similar to the first part of **class Book**. However, it does not have any methods that can be overridden. The thing to note here. Since class Clothing will be used as a base class for Shirt and Pants, then the getters and setters here will be the ones that are called as though they were actually a part of Shirt and Pants. What makes it nice here is that all of the code for type, brand and color does not need to be duplicated for Shirt and Pants and any additional subclasses that may be derived later from Clothing.

Shirt.java

```
public class Shirt extends Clothing {

    private String size;
    private static int shirtCount = 0;

   // getters
    public String getSize()  { return size; }

    // setters
    public String setSize( String size ) {
        this.size=size;
        return this.size; }

    ///// constructors
    Shirt() {
        shirtCount++;   // keep track of the number of shirts
    }

    Shirt(String type, String brand, String size, String color, double price, int inStock) {
        setType(type);
        setBrand(brand);
        setSize(size);
        setColor(color);
        setPrice(price);         // located in the superclass
        setInStock(inStock);     // located in the superclass
        shirtCount++;            // keep track of the number of shirts
    } // end of the six argument constructor

    @Override
    public String toString() {
        return String.format ("%6.2f %s, by %s", getPrice(), getType(), getBrand());
    }

} // end of class Shirt
```

class Shirt also starts out **extends Clothing** to identify that it is an inherited subclass of Clothing. It only has getters and setters for the **size** member data field, but the six-argument constructor is calling six setter methods. The other four setter methods are located in the Clothing and Item classes.

Each of the constructors increment the static **shirtCount** variable to keep track of the number shirt objects that have been instantiated.

Since Shirt is also a leaf node, the **toString( )** method is also provided to override the default toString( ) method. We can see how it will be used in the main( ) program.

the **toString( )** Method

In the class definition

```
// create a String containing   the price, title and author
 @Override
  public String toString() {
     return String.format ("%6.2f %s, by %s", price, title, author);
  }
```

In main( )

```
Book myBook = new Book("Java Program", "Dan", 2018, 14.95, 10);
System.out.println(myBook);
```

```
14.95 Java Program, by Dan
```

Every class has an automatic **toString()** method, even if you don't specify one for a class that you are building. A **toString()** automatically gets called when you concatenate things using the **print( )** or **println( )** methods. These methods only accept strings as their arguments, so if you use **System.out.println("The answer is " + 42);** the println routine sees the string "The answer is " and then the + string concatenate operator and then the 42. But 42 is an integer and the + string concatenate only works with strings so it calls the **toString( )** method for integers to convert 42 to a string. Then the concatenation of "The answer is " + "42" works fine.

We can use the getters that are part of our class with the print, println or printf methods to display any of the objects

data, and in any order we wish. We can also provide a **toString( )** method for our classes that can be called when ever a string is needed. For example, in **class Book**

```
   @Override
   public String toString() {
      return String.format ("%6.2f %s, by %s", price, title,
author);
   }
```

would display the price, title and author in that order for:

```
   Book myBook = new Book("Java Program", "Dan", 2018,
14.95, 10);
   System.out.println(myBook);
```

The display would be:

```
    14.95   Java Program, by Dan
```

The **@Override** specifier is needed to let java use the method written in our class instead of the default that is part of Java itself

BooksAndClothesInheritance.java

```
public class BooksAndClothesInheritance {

    // define an array of books
    private static final Book[] BOOK_LIST = {
        new Book("ULYSSES", "James Joyce", 1918, 32.95, 16 ),
        new Book("THE GREAT GATSBY", "F. Scott Fitzgerald", 1925, 13.95, 30 ),
        new Book("BRAVE NEW WORLD", "Aldous Huxley", 1931 , 14.95, 28 ),
    };
    private static final int BOOK_COUNT = BOOK_LIST.length; // number of books

    // define an array of shirts
    private static final Shirt[] SHIRT_LIST = {
        new Shirt("T-shirt", "Guess", "M", "Blue", 14.95, 23),
        new Shirt("Dress shirt", "Ralph Lauren", "L", "White", 39.95, 5),
        new Shirt("Blouse", "Versace", "S", "Yellow", 44.95, 6),
    };

    public static void main(String[] args) {
        // create some objects
        Book myBook = new Book("Dan the Programmer", "Dan McElroy", 2018, 14.95, 10);
        Shirt myShirt = new Shirt("T-shirt", "Guess", "M", "Blue", 14.95, 23);

        // display the items that were created in main( )
        System.out.println ("My first book is: " + myBook.getTitle());
        System.out.println ("My shirt selection: "+ myShirt.getType());

        // use a standard for statement to display all the books in BOOK_LIST
        //   using  i  to index through the array
        System.out.println ( ); // blank line
        for (int i=0; i<BOOK_COUNT; i++) {
            System.out.printf("%3d  %7.2f  %-30.30s\n",
              BOOK_LIST[i].getInStock(), BOOK_LIST[i].getPrice(), BOOK_LIST[i].getTitle() );
        }

        // use the enhanced-for statement to display all the books in BOOK_LIST
        //   the variable  b  will refer to the current book as the for loop
        //   steps through the array, one book a time
        System.out.println ( ); // blank line
        for (Book b : BOOK_LIST) {
            System.out.printf("%3d  %7.2f  %-30.30s\n",
              b.getInStock(), b.getPrice(), b.getTitle() );
        } // end of for loop

        // display items in the arrays using the toString method
        System.out.println("\nBooks in the array");
        for (Book b : BOOK_LIST) { System.out.println(b); }

        System.out.println("\nShirts in the array");
        for (Shirt s : SHIRT_LIST) { System.out.println(s); }

    } // end of main()
} // end of class
```

1) Create and initialize an array of Book objects and another array of Shirt objects.

2) Create an individual object for a Book object and another object for Shirt. Use println statements to display the title of the book and the type of shirt.

3) Use a standard **for** loop to display the book list from the array, and then an enhanced for loop to display the same array.

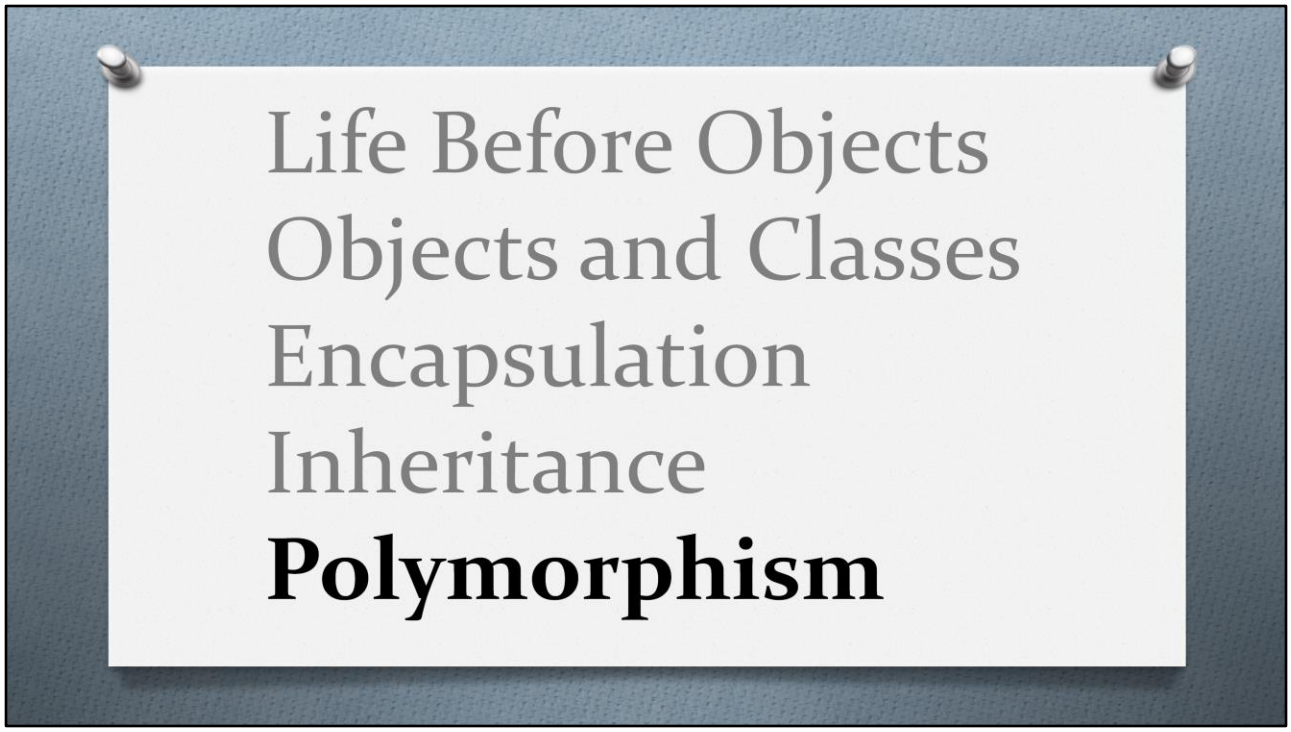
Use enhanced for loops to display the arrays for books and shirts using the overloaded **toString( )** methods.

The class that contains **Main( )** is named **BooksAndClothesInheritance.java**

1) Create and initialize an array of Book objects and another array of Shirt objects.

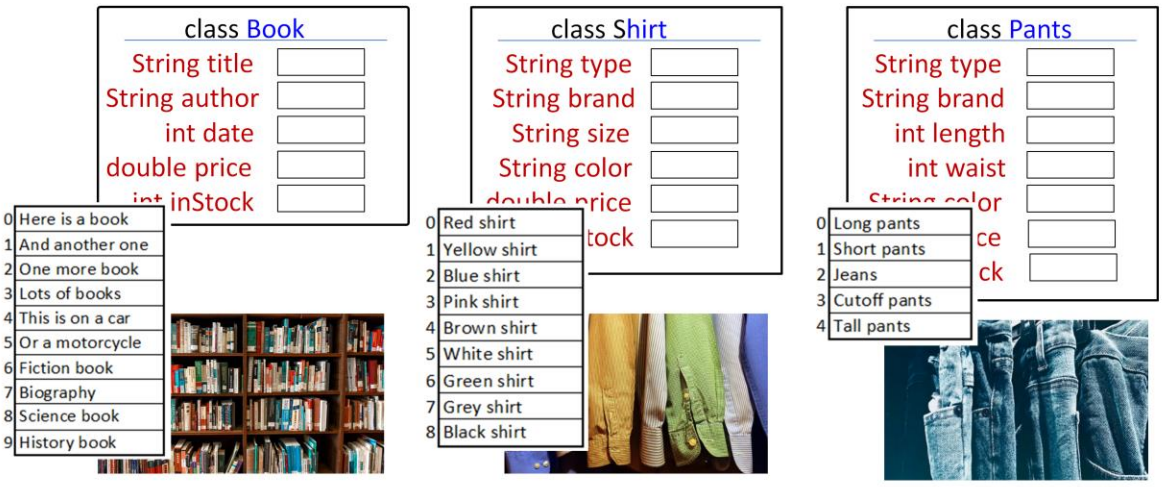
2) Create an individual object for a Book object and another object for Shirt. Use println statements to display the title of the book and the type of shirt.

3) Use a standard **for** loop to display the book list from the array, and then an enhanced for loop to display the same array.
Use enhanced for loops to display the arrays for books and shirts using the overloaded **toString( )** methods.



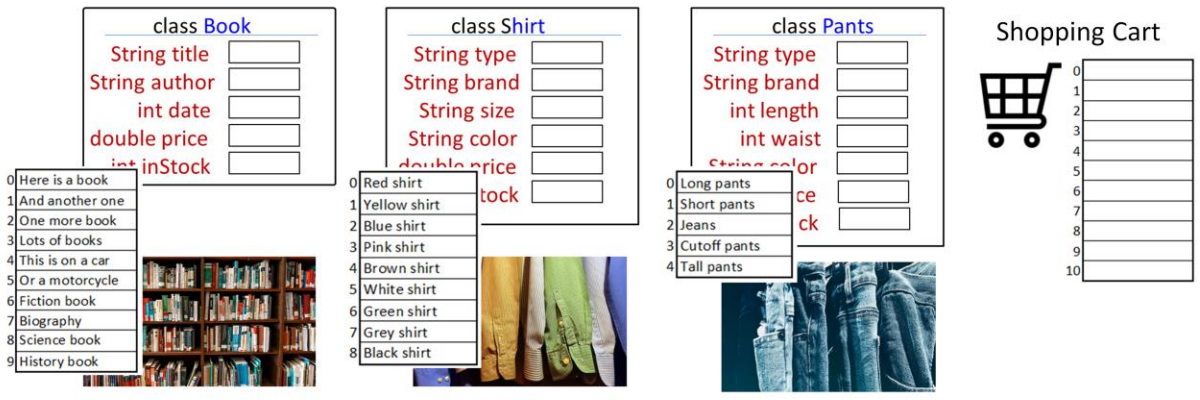
Polymorphism - to be continued

Getting Ready for Polymorphism



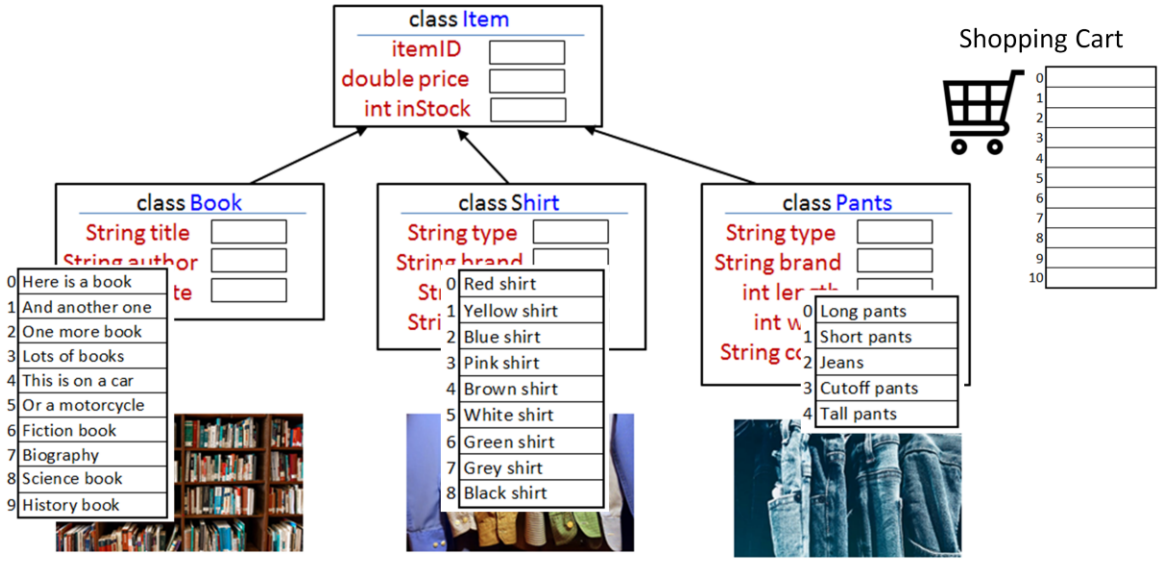
Arrays are nice. I can use them to create a list of each item I want to sell. But every element in an array must be the same data type. I can have an array of integers, an array of doubles, an array of characters, and array of strings, even an array of objects. An object can hold several data types, such as an array of Books, but the array can still only hold books.

Getting Ready for Polymorphism



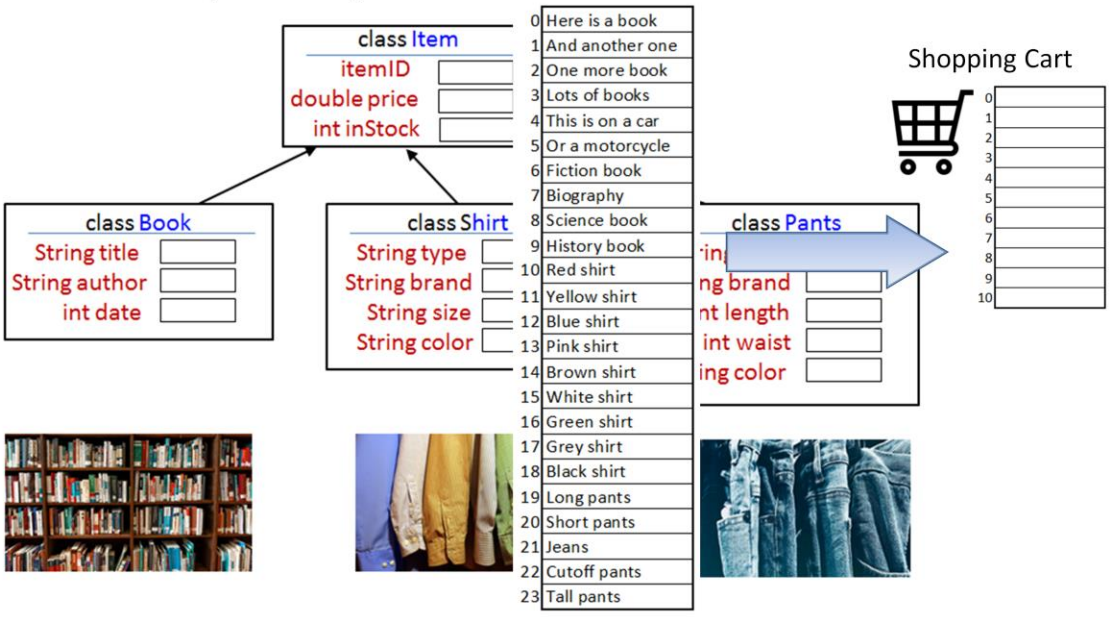
I want to use an array to hold all the items in the customer's shopping cart. I want a shopping cart that can hold any type of item. I don't want a separate shopping cart for books, a different cart for shirts or another one for pants. I only want one shopping cart.

Polymorphism to the Rescue



Polymorphism to the rescue! Since books, shirts and pants are derived from class Item, I can create an array of type Item. I will create a shopping cart of type Item. Then it can hold books, shirts and pants.

Polymorphism to the Rescue



While I am at it, I may as well put everything I want to sell in one array. As they say in "The Lord of the Rings" - one array to hold them all.

An Array of *Items*

```
import java.util.Scanner;
import java.util.InputMismatchException;

public class BooksAndClothesPolymorphism {
    private static Item[] shoppingCart = new Item[10];   // contains items in user's shopping cart
    private static int shoppingCartCount = 0;  // number of total items in the cart

    // define an array of books and shirts
    private static final Item[] ITEM_LIST = {
        new Book(1176,"ULYSSES", "James Joyce", 1918, 32.95, 16 ),
        new Book(1252,"THE GREAT GATSBY", "F. Scott Fitzgerald", 1925, 13.95, 30 ),
        new Book(1376,"BRAVE NEW WORLD", "Aldous Huxley", 1931 , 14.95, 28 ),
        new Book(1463,"TO THE LIGHTHOUSE", "Virginia Woolf", 1927, 36.95, 19 ),
        new Book(1545,"THE SOUND AND THE FURY", "William Faulkner", 1929, 17.95, 11 ),
        new Book(1605,"CATCH-22", "Joseph Heller", 1961, 38.95, 25 ),
        new Book(1824,"THE DEATH OF THE HEART", "Elizabeth Bowen", 1938, 26.95, 44 ),
        new Book(1873,"DARKNESS AT NOON", "Arthur Koestler", 1940, 39.95, 6 ),
        new Book(1909,"THE GRAPES OF WRATH", "John Steinbeck", 1939, 12.95, 32 ),
        new Book(1945,"1984", "George Orwell", 1949, 24.95, 24 ),
        new Shirt(2443,"T-shirt", "Guess", "M", "Blue", 14.95, 23),
        new Shirt(2867,"Dress shirt", "Ralph Lauren", "M", "White", 39.95, 5),
        new Shirt(2868,"Dress shirt", "Ralph Lauren", "L", "White", 39.95, 4),
        new Shirt(2869,"Dress shirt", "Ralph Lauren", "XL", "White", 39.95, 1),
        new Shirt(2905,"Blouse", "Versace", "S", "Yellow", 44.95, 6),
        new Shirt(2923,"Tank top", "Zoned Out", "XLT", "White", 15.45, 2),
    };
```

The program starts out with the import statements.
private static Item[] shoppingCart = new Item[10];
creates the shopping cart which will be an array of type Item and it can hold 10 things in the cart.

shoppingCart.length could be used to identify the size of the cart, but not the number of items in the cart. Another variable is needed for this.

private static int shoppingCartCount = 0;
shoppingCartCount is used to keep track of the number of items actually in the cart. It needs incremented each time an item is placed in the cart.

Another array of Items is used to hold all the things for sale.

private static final Item[] ITEM_LIST

Anything that is derived from Items can be placed in the **ITEM_LIST** array. The data type is Item. Since it is listed as static final, the name of the array is capitalized, ***ITEM_LIST.*** This array holds books and shirts. Each object in the array now starts with an item number such as 1176, 1252, etc. This number will be used when searching the list when the customer wants to add things to the shopping cart.

We need to remember that in Java, creating an object is a two step process. If I were to only create a single object, I could do it two ways.

```
Book myBook;  // creates a reference to identify an object when created
myBook = new Book(1252,"JAVA FOR ME", "Dan", 2018, 13.95, 5);
// instantiates the object and places its pointer in myBook
```

private static final Item[] ITEM_LIST
only creates an array of references. In order to be used, each reference needs to hold a pointer to the memory location where the object is instantiated by using the **new( )** constructor.

main( ) - Start by Displaying the List

```
public static void main(String[] args) {
    // create the stdin object (to use the keyboard)
    Scanner stdin = new Scanner(System.in);
    int itemSelected = 0;    // Item ID selected by user, 0 for not available
    int itemIndex = 0;       // selected index into array of items

    // display items in the arrays using the toString method
    System.out.printf ("%-4.4s %6.6s %-11.11s\n", "Item", "Price", "Description");
    for (Item b : ITEM_LIST) { System.out.println(b); }
```

main( ) starts out declaring two integers, **int itemSelected = 0;** and **int itemIndex = 0;**
These will be used searching the list when the customer selects an item by its number.

Main then displays a list of all the items in the ***ITEM_LIST*** array starting with a title at the top of the displayed list. The System.out.printf method is used to make it easier to line up the columns for the title with the columns for the items being displayed. printf's format string shows three fields, each identified with a percent sign
"%-4.4s %6.6s %-11.11s\n"
The first field is a string, 4 characters wide and left justified. the next string is 6 characters wide and the third string is 11 characters wide, left justified. The strings that are displayed

are "Item", "Price", and "Description"

The enhanced for loop is used.
Item b : ITEM_LIST says that **b** will be set to the next **Item** from **ITEM_LIST** each pass through the for loop.

System.out.println(b); calls the toString method for whatever item is currently selected in the loop. For example, if a book is the current item, then **Book.toString( )** method is called for that item. Or, if the item is a shirt, then **Shirt.toString( )** is called for that item.

main( ) - Sell an Item to the Customer

```
System.out.println ("\nSelect an item by its item number. Enter 0 to quit");
do {
    try {
        System.out.print ("item: ");
        itemSelected = stdin.nextInt( );        // read line from keyboard
        if (itemSelected == 0)
            continue;    // EXIT: jump to the end of the loop

        // Search ITEM_LIST looking for the user's requested itemID
        for (itemIndex=0; itemIndex<ITEM_LIST.length; itemIndex++)
            if (itemSelected == ITEM_LIST[itemIndex].getItemID())
                break;  // it was found, itemIndex = position in the LIST

        if (itemIndex == ITEM_LIST.length)  // reached the end and not found
            System.out.println("Item is not available");

        else {  // The item was found
                shoppingCart[shoppingCartCount] = ITEM_LIST[itemIndex];
                shoppingCartCount++;  // keep track of items in the cart
        }
    }
    catch (InputMismatchException |  StringIndexOutOfBoundsException e) {
        System.out.println ("Illegal selection. Try again");
    }
} while (itemSelected != 0);    // loop until a '0' is entered
```

The prompt asks the customer to select an item by its item number, and enter a zero to quit.

A do-while loop is used to gather the customer's selections and place them in the shopping cart. System.out.print("Item: "); prompts the customer to enter an item number. Note that print is used rather than println. This keeps the cursor on the same line as the prompt. The stdin.nextInt( ); is used to input an integer from the keyboard. The do-while loop already has a try-catch built in to process any illegal entries the customer may enter for the item selection.

If the customer enters a zero, the **continue** statement jumps to the end of the loop, and lands on the **while**

(itemSelected != 0); Since the itemSelected is 0 in this case, the loop ends and the program can display the shopping cart.

A for loop is used to search ITEM_LIST. If the requested item is found,
if (itemSelected == ITEM_LIST[itemIndex].getItemID())
then the **break;** statement exits the loop with itemIndex still set to the value where the item was found. If the end of the loop is reached and not exited with the break; statement, then itemIndex will be set to ITEM_LIST.length. Testing for this value being true indicates that the item was not found.

If the itemID was found in ITEM_LIST, then place it int the shopping cart and keep track of how many items are in the cart.

Look back at **class Item**. It has a private variable named **inStock** that gets initialized when an object is instantiated. For example, the last parameter in
new Book(1176,"ULYSSES", "James Joyce", 1918, 32.95, 16)
indicates that there are 16 of this book in stock.

**** YOU HAVE TWO THINGS TO

UPLDATE FOR THIS PROJECT ****

UPDATE #1

The array that is used for the shopping cart only has room for 10 items. The **shoppingCartCount** is incremented each time an item is placed in the cart. It is your task to test to see if the cart if full before trying to place add any more items. If the cart is full, the display a message, "Your shopping cart is full". The cart is full if shoppngCartCount is equal to the length of the array for the shopping cart.

UPDATE #2

class item also has a getter and setter for **inStock** that can be used to read and update the inStock variable. It is your task to check to see if any items are available before placing them in the shopping cart and then subtract 1 from the inStock variable. If no items are available, then display a message, "Out of stock. Please try again later"

You will need to use the **getInStock( )** and **setInStock(*newValue)*** with the selected item.

Use **getInStock( )** to read the current value and **setInStock(*newValue)*** to update the inStock count of the selected item.

main( ) - Display the Shopping Cart

```
        // display the shopping cart
        System.out.println("\n\nThank you for shopping at dan-azon");
        double total=0;
        for (int i=0; i<shoppingCartCount; i++) {
            System.out.println(shoppingCart[i]);
            total += shoppingCart[i].getPrice();
        }
        System.out.println(shoppingCartCount + " items in your cart");
        System.out.printf("Your total is $%.2f\n\n", total);

    } // end of main()

} // end of class
```

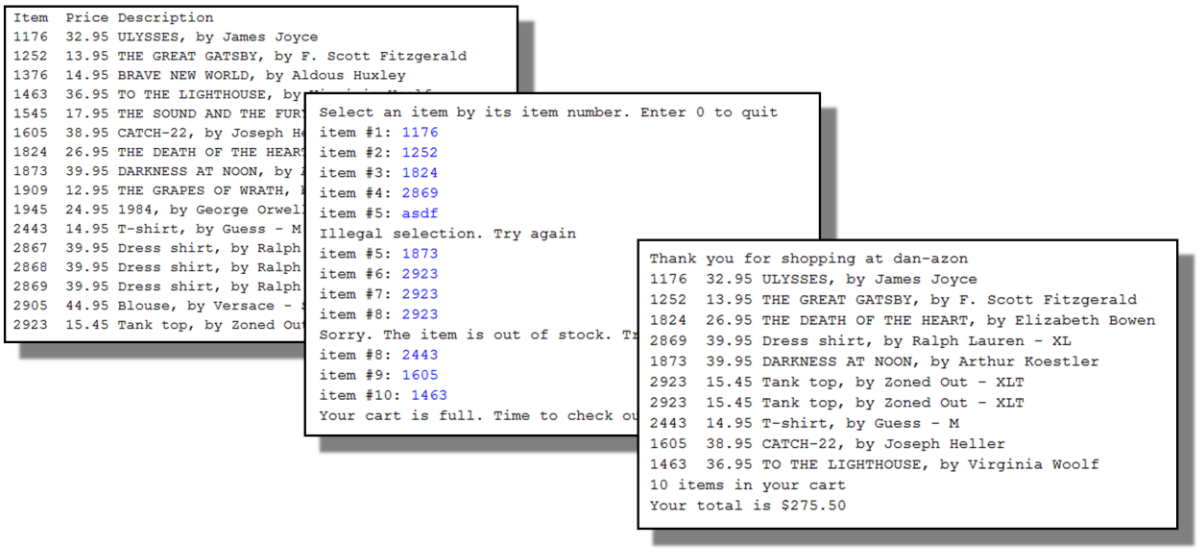
The last thing the program does is display the contents of the shopping cart, the number of items in it and the total bill for all the items.

The for loop can't use the enhanced version because it is not necessarily going to display the entire array that holds the shopping cart, because the cart might not be completely full. The standard for loop is used to only display the number of items placed in the array, identified by the variable **shopping CartCount.**

One more thing in the for loop, since it is already going through each item in the cart for the display, it can also add up the total bill at the same time.

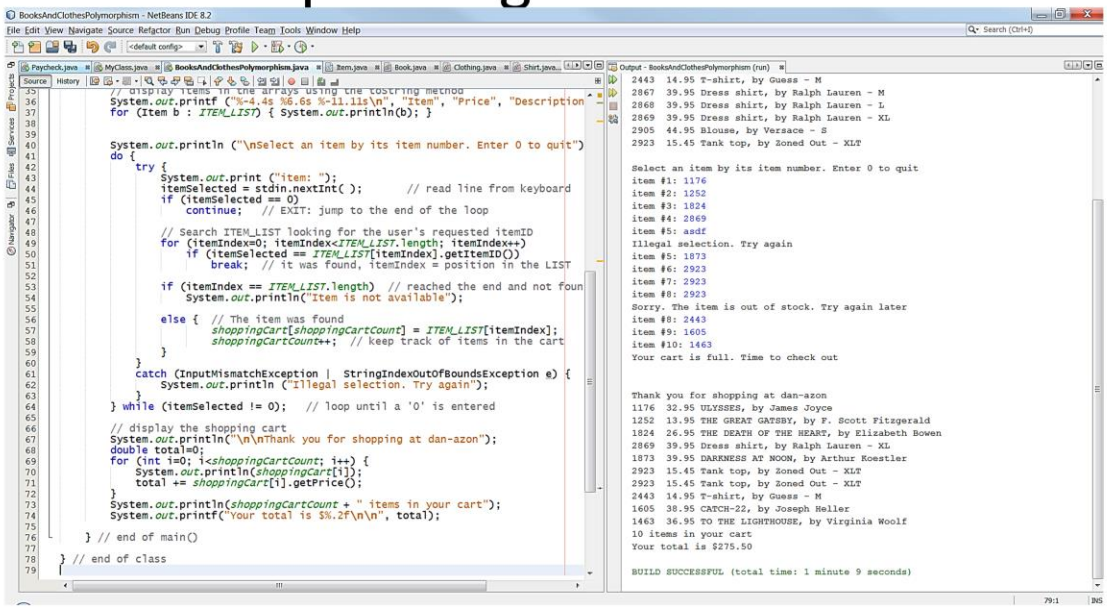
The program could use either a printf or a println statement to display the number of items in the cart. However, in order to make sure that two digits are displayed past the decimal for the total bill, the printf statement is needed.

Sample Program Execution



The sample program execution first shows the list of items in the inventory and then lets the customer start choosing items from the list. The loop that lets customers choose items ends either when the shopping cart is full with 10 items, or the customer types a 0 (zero) to indicate that shopping is finished.

Sample Program Execution

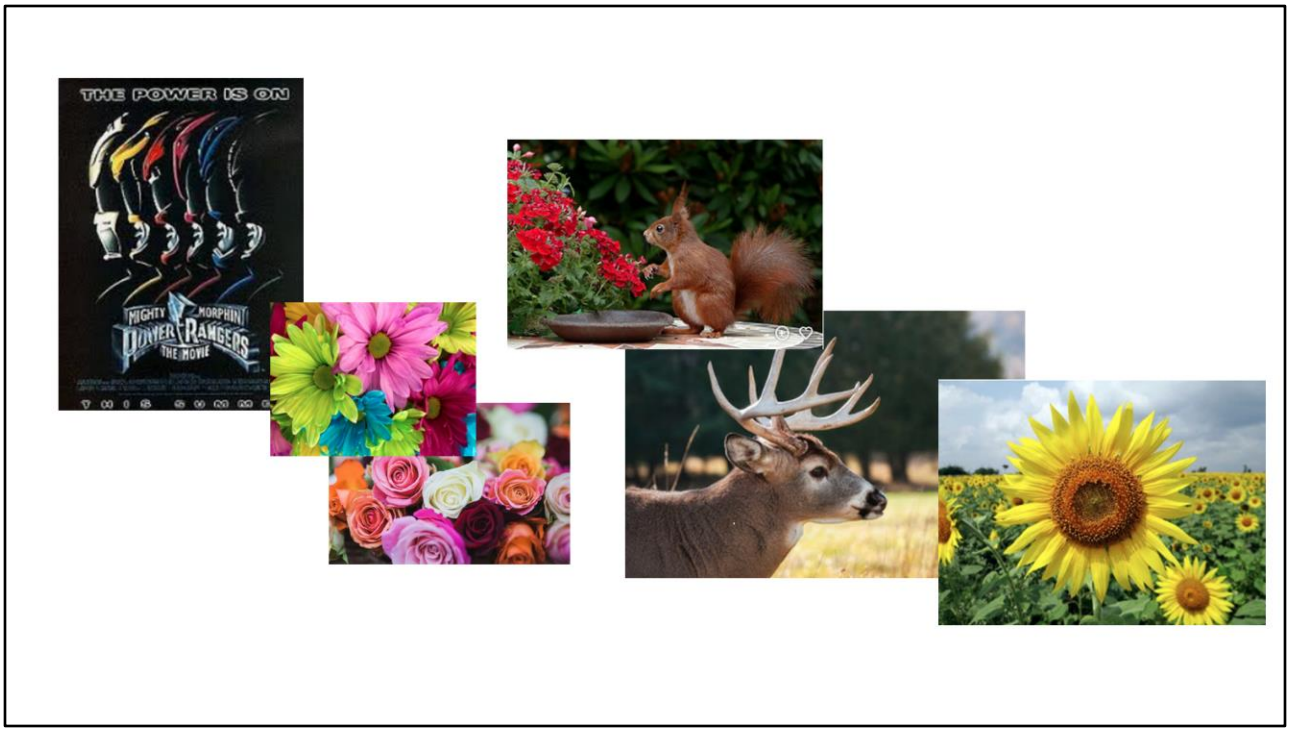


And lastly, here is a screenshot showing the program using Netbeans. The output window was moved and docked on the right side of Netbeans.

CREDITS

- Bookshelf - pexels.com, Element5 Digital
- Garbage Collection - pexels.com, Steve Johnson
- Shirts and Pants - pexels.com
- Shopping cart - freeiconshop.com

I would like to give credit to:
Bookshelf - pexels.com, Element5 Digital
Garbage Collection - pexels.com, Steve Johnson
Shirts and Pants - pexels.com
Shopping cart - freeiconshop.com

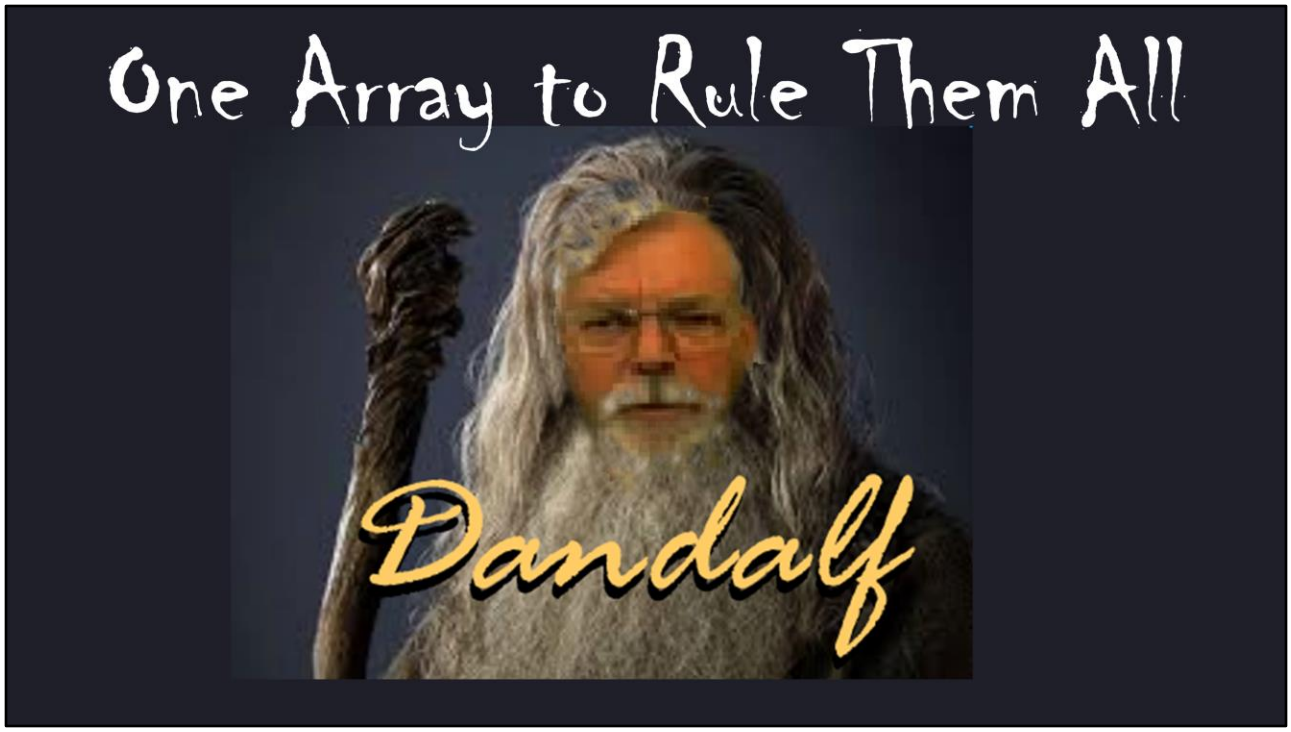


As long as we are talking about polymorphism, I have a copy of the original - "The Mighty Morphin Power Rangers - Only they can save our planet from evil!"

"Power up with six incredible teens who out-maneuver and defeat evil everywhere as the Mighty Morphin Power Rangers! But this time, the Power Rangers may have met their match when they face off with the most sinister monster the galaxy has ever seen - Ivan Ooze" "YOU OOZE, YOU LOSE!"

And now the sequel. Only available at dan-azon. The Mighty Polymorphin Flower Rangers. "Flower, flower, I want to be a flower ranger" The buck stops here dude. Da-da-da-dum. See how polymorphism and virtual functions can conquer

classes, implement run-time dynamic linking and save the world from abstract methods.



And remember that with Polymorphism, we can have ONE ARRAY TO RULE THEM ALL