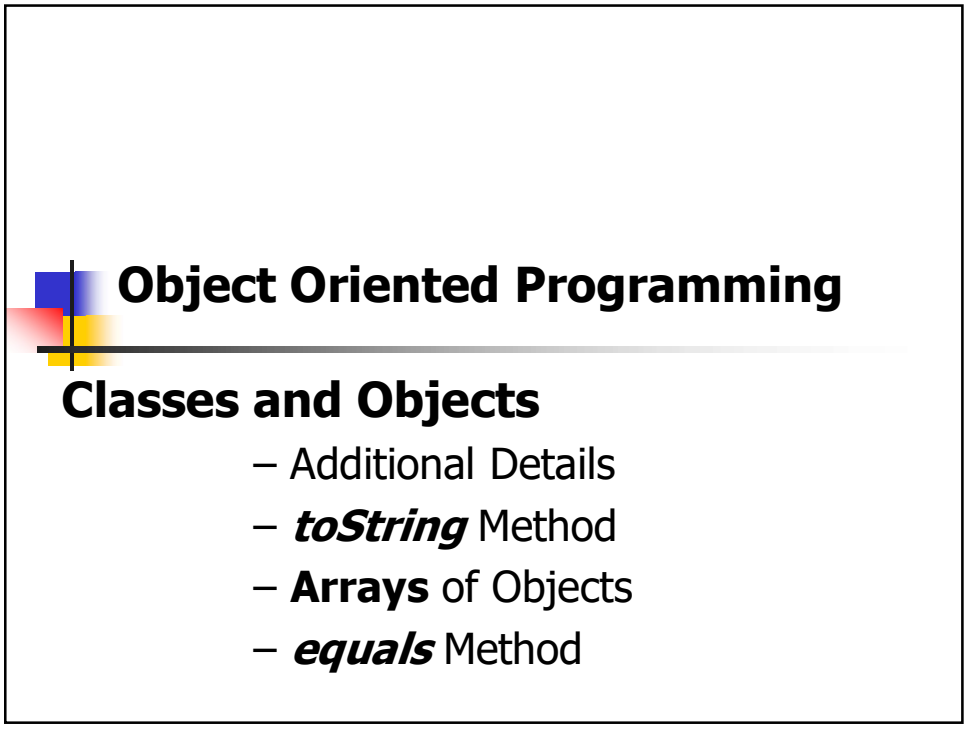
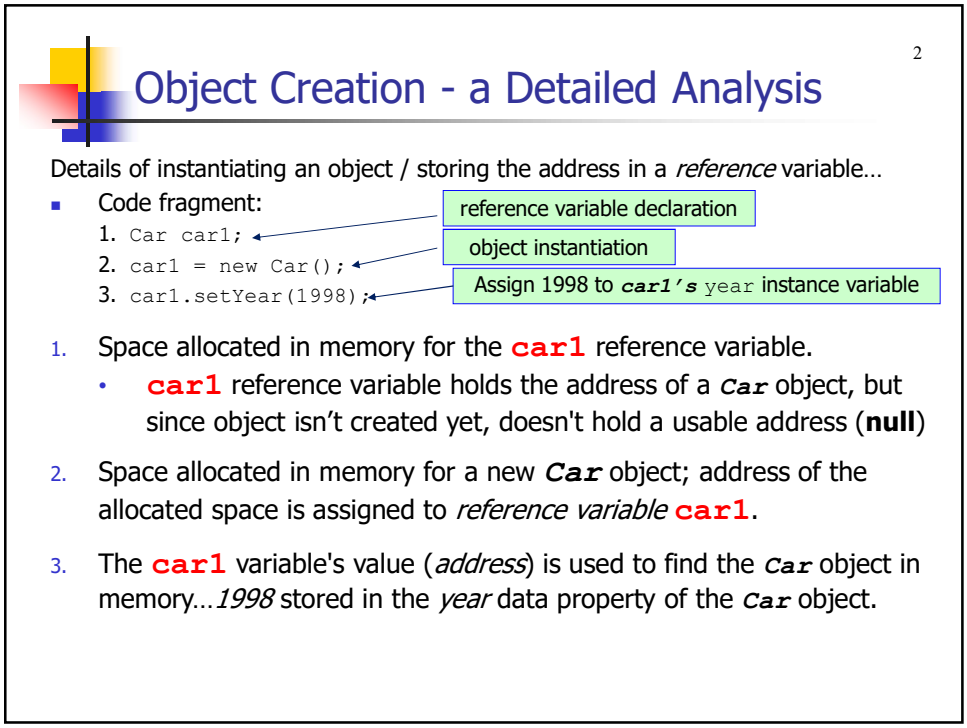
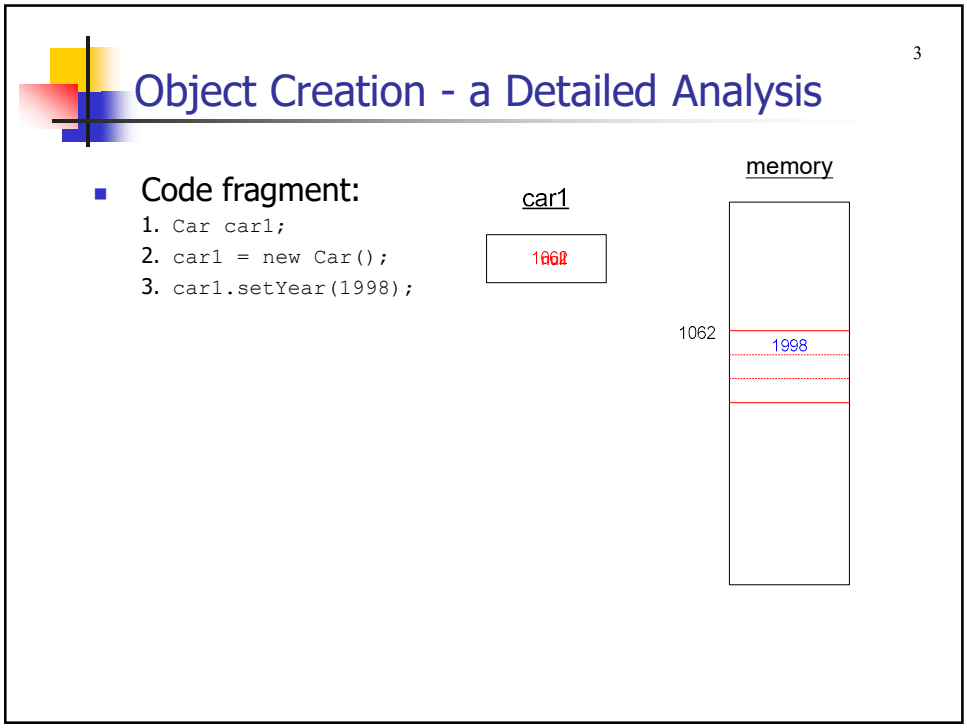

Object Oriented Programming
Classes and Objects
– Additional Details
– toString Method
– Arrays of Objects
– equals Method


2
Object Creation - a Detailed Analysis
Details of instantiating an object / storing the address in a reference variable…
Code fragment:
1. Car car1;
2. car1 = new Car();
3. car1.setYear(1998);
reference variable declaration
object instantiation
Assign 1998 to car1's year instance variable
1. Space allocated in memory for the car1 reference variable.
• car1 reference variable holds the address of a Car object, but since object isn't created yet, doesn't hold a usable address (null)
2. Space allocated in memory for a new Car object; address of the allocated space is assigned to reference variable car1.
3. The car1 variable's value (address) is used to find the Car object in memory…1998 stored in the year data property of the Car object.

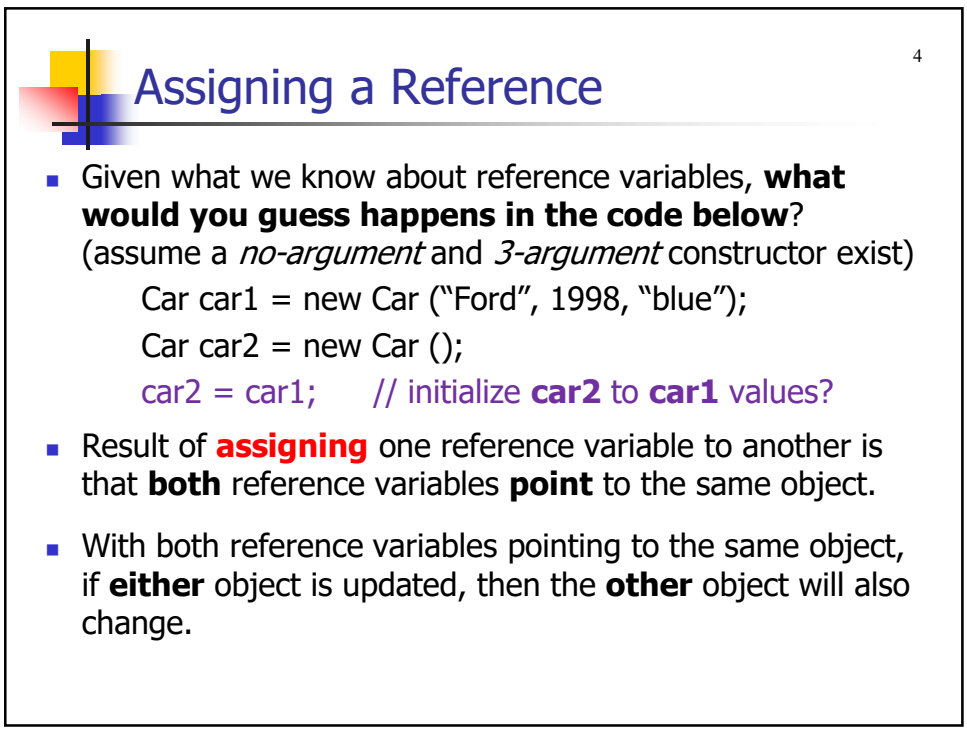


Object Creation - a Detailed Analysis

3

- Code fragment:

```
1. Car car1;
2. car1 = new Car();
3. car1.setYear(1998);
```

Assigning a Reference

4

- Given what we know about reference variables, **what would you guess happens in the code below**? (assume a *no-argument* and *3-argument* constructor exist)

```
Car car1 = new Car ("Ford", 1998, "blue");
Car car2 = new Car ();
car2 = car1;      // initialize car2 to car1 values?
```

- Result of **assigning** one reference variable to another is that **both** reference variables **point** to the same object.
- With both reference variables pointing to the same object, if **either** object is updated, then the **other** object will also change.

Assigning a Reference (Example)

5

- Suppose you want to create two `Car` objects that are the same except for their **color**. Your plan is to create the first car, copy the first car to the second car, and then update the second car's `color` instance variable. *Will this code accomplish that?*

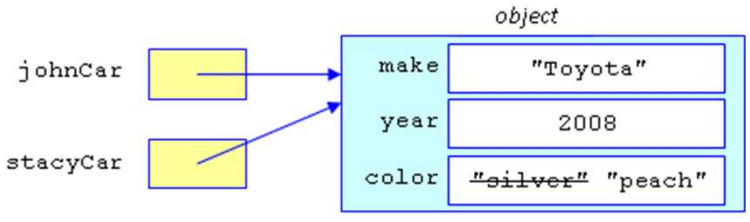
```
Car stacyCar;              // What is the difference in
Car johnCar = new Car(); // these two lines?

johnCar.setMake("Toyota");
johnCar.setYear(2008);
johnCar.setColor("silver");

stacyCar = johnCar;
stacyCar.setColor("peach");
```

Assigning a Reference

6

- `stacyCar = johnCar;` causes the two references to point to the same single `Car` object.
- Thus, `johnCar's` color *unintentionally* becomes "peach"

```
johnCar = new Car();
...
stacyCar = johnCar;
stacyCar.setColor("peach");
```



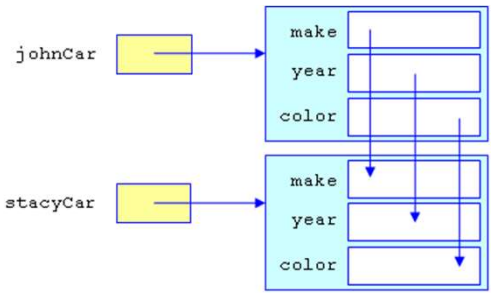
Assigning a Reference

7

- If you want to make a **copy** of a **reference variable**, instantiate a **new** object for the second reference and then assign each *instance variable* one at a time.

```
johnCar = new Car();
stacyCar = new Car();
```

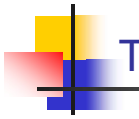
(assign `johnCar`'s instance variables to `stacyCar`'s instance variables)



Summary on **Passing References as Arguments**

8

- At this point, you should be fairly comfortable with passing **primitive** types (int, double, char) to a method (**pass-by-value**)
  Ex: **multiply (a, b, c);** // where int a, b, c
- When you pass the value of a ***reference* variable** to a method (Arrays, Objects), remember it holds the **address** of an object (or Array), **not** the object itself
  Ex: **update (car1);** // where car1 is a Car Object
- A **copy** of the object's **address** (not a copy of the object) is passed to the method and stored in the method's parameter variable.
  Ex: **public void update (Car tempCar)**
- Since the parameter variable (**tempCar**) is passed the same address value contained by the reference variable (**car1**), BOTH point to (reference) the same object.
- If one of the object's instance variables is updated, you are also updating the original object's instance variable: **tempCar.setColor("Peach");**
  - This is how you **pass-by-reference** in Java



The **toString**() Method

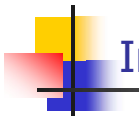
9

- All defined classes should contain a special method called **toString** ()
- **toString** () methods are called and used to display **some** or **all** of an object's data properties
- To work properly, **all** *toString* methods must have the following method signature:

public **String** toString ()

- The **method call** can then be placed inside a **print** statement to display all of the desired data properties of an object

DEMO: In the **Car.java** class, change the **display** method to a **toString** method



Invoking the **toString**() Method

10

- Remember that the **toString** method **returns a string**
- It is **called** (invoked) in either two ways:

1) System.out.println("\nNathan's Car:" + nathanCar.**toString**());
2) System.out.println("\nNick's Car:" + nickCar);

- As long as method signature **public String toString()** is used, **explicit** toString () method call (in #1) is *optional* (call to toString is **implicit**)

DEMO: Change **CarApp.java** class to invoke the **toString** method using both ways shown above

11

Summary on the **toString()** Method

Reminder:

1) Method must **return a String** and have **nothing** passed to it
2) **No *print* statements** are placed within a ***toString*** method

- Since ***you*** are the developer of the class (and ***toString*** method) **YOU** decide **what** is displayed (and **how**) within your **toString** method

Arrays of **primitive** data types

- So far, we have seen Arrays hold multiple **primitive** data type values...

Examples:

```
char[] grades = new char[27];
double[] averages = new double[27];
int [] testScores = new int[3];
```

- Remember that Array variables are **reference** variables
- Since Arrays are numerically indexed, **for** loops are *usually* used to process them

```
for (int i=0; i < testScores.length; i++)
  sum += testScores[i];
```

12

Arrays of **OBJECTS** in Java

- Arrays can also be used to hold **Objects**

For Example…

```
Car [ ] carArray = new Car [10];      // holds 10 Car objects
Fraction [ ] fractionArray = new Fraction [50]; // holds 50 Fractions
Point [ ] point = new Point [100];    // holds 100 (x,y) Point objects
```

DEMO: Modify **CarApp** class to hold **10 Car Objects**

- Instantiate only the first two objects and use **toString** to display only those initialized objects

13

14

Testing Objects for **Equality**

- Comparing two String **objects** with '=='

```
String s1 = "Cis 200";
String s2 = "Cis 200";

if (s1 == s2)
{
  System.out.println("the same");
}
else
{
  System.out.println("different");
}
```

This expression returns **false**. Why?

15

Testing Objects for **Equality**

- Using the '**==**' operator with Objects:

```
Car car1 = new Car();
car1.setColor("red");
Car car2 = new Car();
car2.setColor("red");
if (car1 == car2)
{
  System.out.println("the same");
}
else
{
  System.out.println("different");
}
```

The expression **car1 == car2** returns **false**. Why?

16

Testing Objects for Equality

- When comparing objects, the '==' operator returns ***true*** **only** if the two reference variables **point** to the same object (i.e., contain the same address)
- When testing for the equality of two objects, you (usually) want to compare the **contents** of two objects
- Done by defining an ***equals*** method in the object's class definition that compares the contents of the two objects.
- *Where have we seen the equals method before?*

  Given **String s1, s2***:* **if** (**s1.equals(s2)**)…

17

Summary on the **equals**() method

- Like the ***toString***() method, all defined classes should include an `equals` method to test the **equality** of two objects
- When testing for the equality of two objects, you usually compare the **contents** (i.e. *data properties*) of two objects
- Like toString(), since **you** are the developer of the class (and the *equals* method) **YOU** decide when two objects are equal (same *make/year*? Same *VIN#*?)
- All ***equals*** methods have the ***same method signature***:

  public **boolean** **equals** (<ObjectType> parameterName)