COP 3330: Object-Oriented Programming Summer 2011

Classes In Java – Part 2
Inheritance and Polymorphism

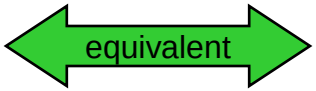
Instructor : Dr. Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 407-823-2790
http://www.cs.ucf.edu/courses/cop3330/sum2011

Department of Electrical Engineering and Computer Science
Computer Science Division
University of Central Florida

The `Object` Class and its Methods

- Every class in Java is descended from the `java.lang.Object` class. If no inheritance is specified when a class is defined, the superclass of the class is `Object` by default.
- For example, the following two class declarations are equivalent:

```
public class Circle {
...
}
```




```
public class Circle extends Object {
...
}
```

- Classes like `String`, `Loan`, `GeometricObject` are implicitly subclasses of `Object` (as are all of the classes we have constructed so far in this course).
- It is important to be familiar with the methods provided by the `Object` class so that you can use them in your classes.

Method Summary

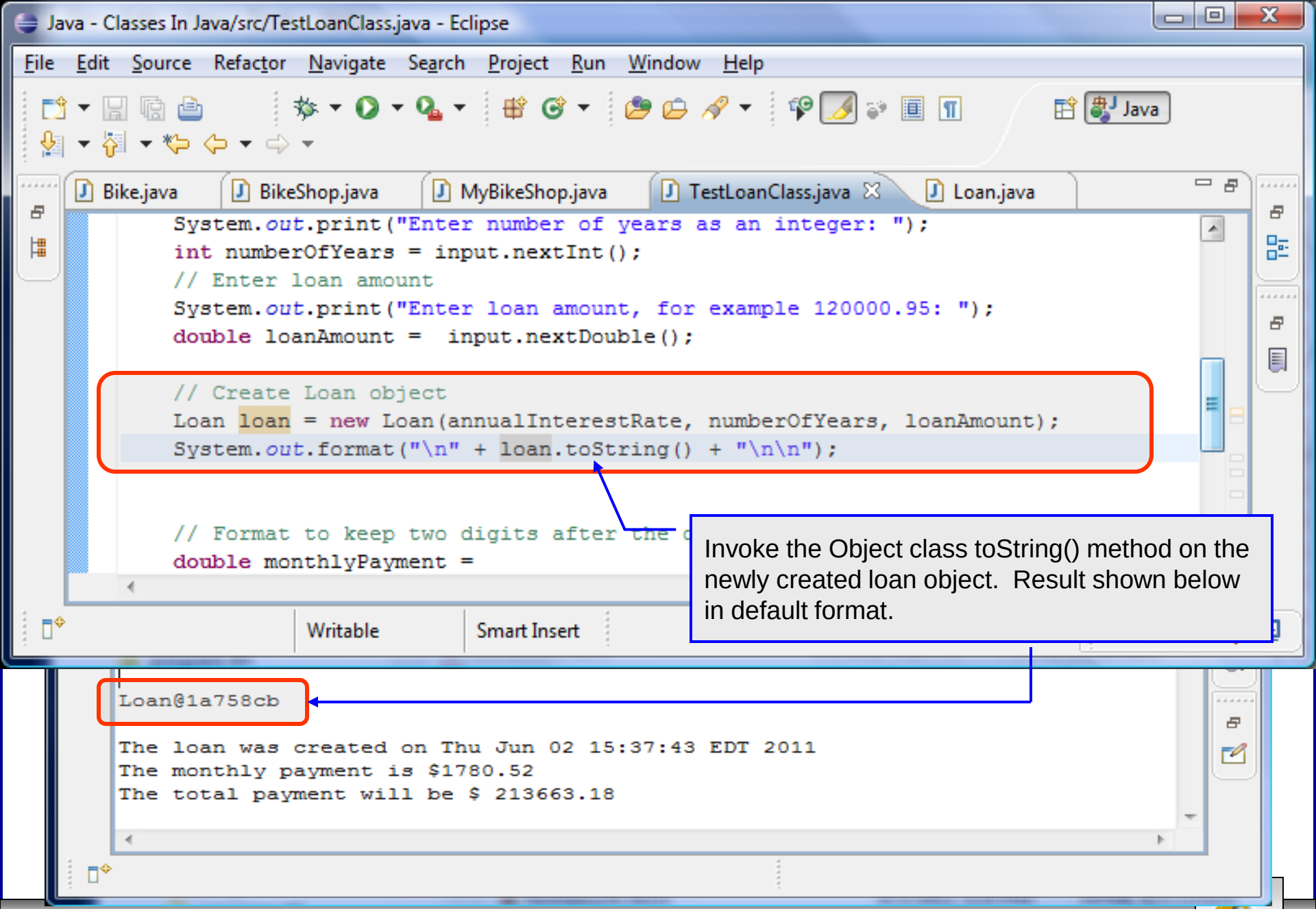
protected Object	clone() Creates and returns a copy of this object.
boolean	equals(Object obj) Indicates whether some other object is "equal to" this one.
protected void	finalize() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?>	getClass() Returns the runtime class of this Object.
int	hashCode() Returns a hash code value for the object.
void	notify() Wakes up a single thread that is waiting on this object's monitor.
void	notifyAll() Wakes up all threads that are waiting on this object's monitor.
String	toString() Returns a string representation of the object.
void	wait() Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
void	wait(long timeout) Causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
void	wait(long timeout, int nanos) Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

The `Object` Class and its Methods

- For right now, we want to focus on two of these methods, the `toString` and `equals`.
- The signature of the `toString()` method is:

```
public String toString()
```

- Invoking `toString()` on an object returns a string that describes the object.
- By default, it returns a string consisting of the class name of which the object is an instance, an at sign (@), and the object's memory address in hexadecimal.
- The following page, shows a slight modification to the `TestLoan` class we wrote in the previous section to illustrate the use of the `toString()` method.


Java - Classes In Java/src/TestLoanClass.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Java
Bike.java
BikeShop.java
MyBikeShop.java
TestLoanClass.java
Loan.java
System.out.print("Enter number of years as an integer: ");
int numberOfYears = input.nextInt();
// Enter loan amount
System.out.print("Enter loan amount, for example 120000.95: ");
double loanAmount = input.nextDouble();
// Create Loan object
Loan loan = new Loan(annualInterestRate, numberOfYears, loanAmount);
System.out.format("\n" + loan.toString() + "\n\n");
// Format to keep two digits after the
double monthlyPayment =
Writable
Smart Insert
Invoke the Object class toString() method on the newly created loan object. Result shown below in default format.
Loan@1a758cb
The loan was created on Thu Jun 02 15:37:43 EDT 2011
The monthly payment is $1780.52
The total payment will be $ 213663.18

The `Object` Class and its Methods

- Obviously, this message is not very helpful or informative.
- Usually, you should override the `toString` method so that it returns a descriptive string representation of the object. This is what we did in the `GeometricObject` class as shown below:

```
/** Return a string representation of this object */
  public String toString() {
    return "created on " + dateCreated + "\ncolor: " + color +
      " and filled: " + filled;
  }
```

NOTE: You can also pass an object to invoke `System.out.println(object)`. This is equivalent to invoking `System.out.println(object.toString())`.

The `Object` Class and its Methods

- The signature of the equals method is:

```
public boolean equals(Object o)
```

- This method test whether two objects are equal.
- The syntax for invoking this method is:

```
object1.equals(object2);
```

- The default implementation of the equals method in the `Object` class is:

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

- This implementation checks whether two reference variables point to the same object using the == operator. You should override this method in your custom class to test whether two distinct objects have the same content.

The `Object` Class and its Methods

- Using the `Circle` class as an example, let's override the `Object` class equals method in the `Circle` class to determine if two circle objects are the same based on their radius.

```
public boolean equals(Object obj) {
   if (obj instanceof Circle) {
      return radius == ((Circle)obj).radius;
   }
   else return false;
}
```

`if (objectA instanceof classB)`
will yield true if objectA can be upcast to objectB.

Inheritance allows one to design objects in terms of other existing objects. This can often lead to objects which inherit from other objects which, in turn, inherit from yet other objects. The instanceof operator will yield true if the right operand appears anywhere up the chain from which the left operand inherits.

The `Object` Class and its Methods

- The == comparison operator is used for comparing two primitive data type values or for determining whether two objects have the same references.
- The `equals` method is intended to test whether two objects have the same contents, provided that the method is modified (overridden) in the defining class of the objects.
- The == operator is stronger than the `equals` method, in that the == operator checks whether the two reference variables refer to the same object.

The `Object` Class and its Methods

* * * Caution * * *

Using the signature `equals(someClassName obj)` (e.g. `equals(Circle c)`) to override the equals method in a subclass is a common mistake.

Why is this a mistake?

It's a mistake because you are not overriding anything! The signature is different from the one in the `Object` class, so you are simply defining an overloaded method.

You should always use the form `equals(Object obj)` in your custom classes.

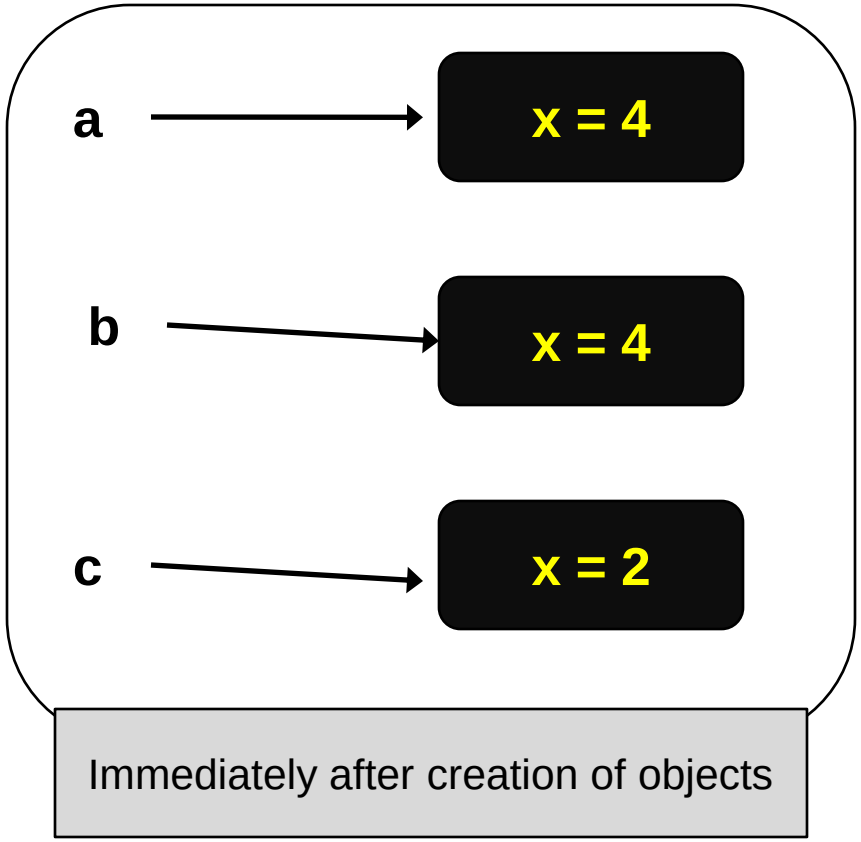
More On Overriding/Overloading And The `Object` Class

TestPolymorphismCast | *A3.java | A3Prime.java | 6

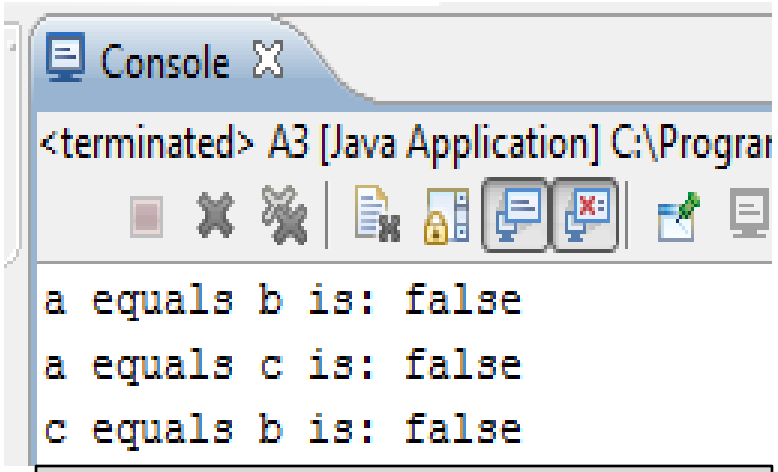
```
//Class to demonstrate use of Object class equals() method
//NOTE:  Object class equals() method is based on memory addresses
public class A3 {
    private int x;
    public A3(int y) { x = y;}

    public static void main(String args[]){
        A3 a = new A3(4);
        A3 b = new A3(4);
        A3 c = new A3(2);
        System.out.println("a equals b is: " + a.equals(b));
        System.out.println("a equals c is: " + a.equals(c));
        System.out.println("c equals b is: " + c.equals(b));
        c = b;  //reassign c to reference b
        System.out.println("c equals b is: " + c.equals(b));
    }
}
```

Since there is no `equals()` method defined in class `A3`, these invocations are all to the `equals()` method in the `Object` class.

More On Overriding/Overloading And The `Object` Class



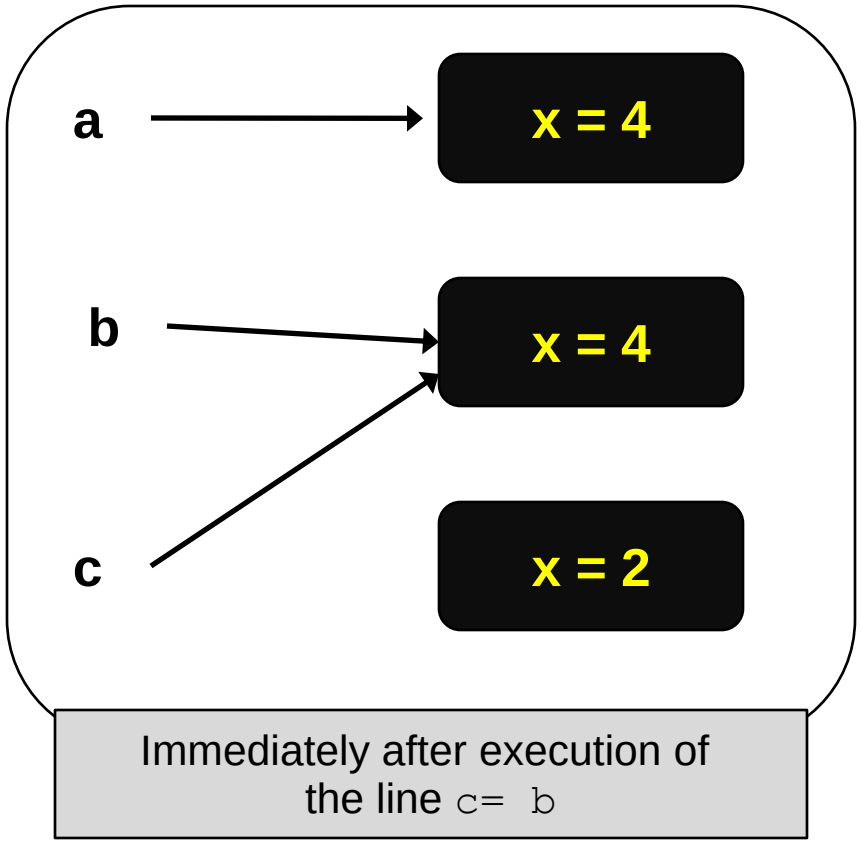
Immediately after creation of objects



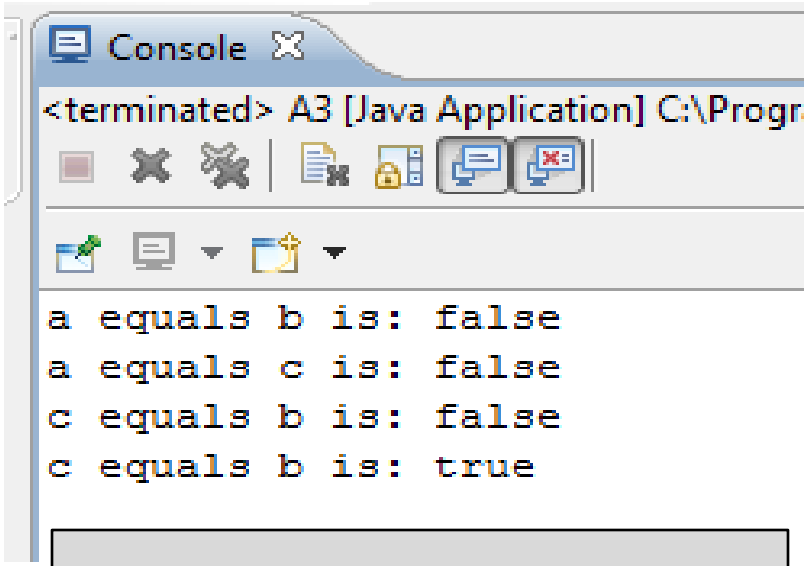
The first three lines of output

The first three lines of output from the calls to the `equals()` method indicate `false`. This is because each of the objects is referring to a different location in memory, i.e., a different instance.

More On Overriding/Overloading And The `Object` Class



Immediately after execution of the line `c= b`



The entire output

The last line of output now indicates a value of `true` returned by the call to the `equals()` method since the objects `b` and `c` refer to the same instance now.

More On Overriding/Overloading And The `Object` Class

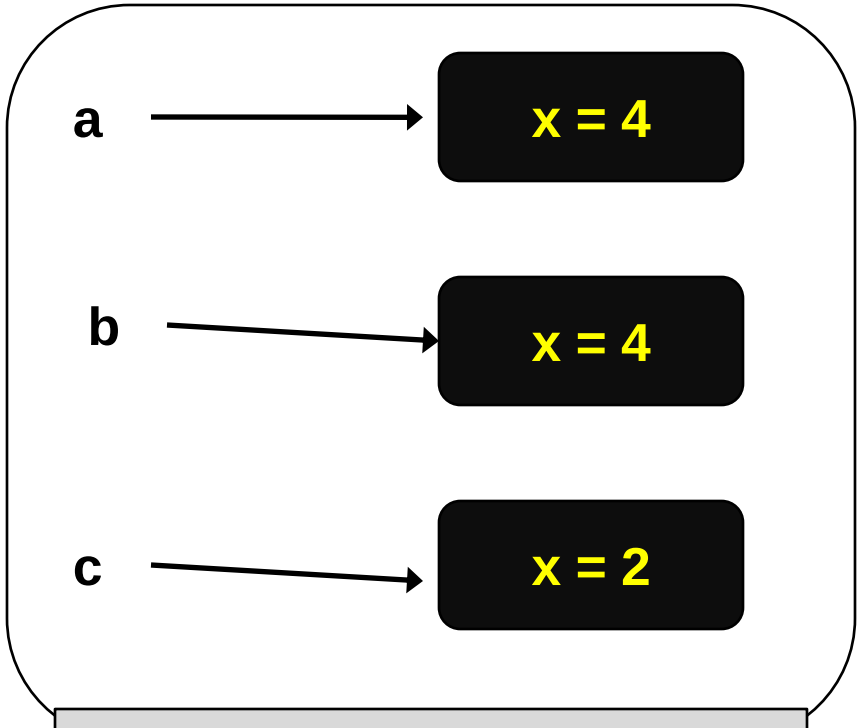
PolymorphismDemo.jav | TestPolymorphismCast | A3.java | *A3Prime.java

```
//Class to demonstrate use of overriding the Object class equals() method
//NOTE:  The overridden method is based on values in the object
public class A3Prime {
    private int x;
    public A3Prime(int y) { x = y;}
    public boolean equals(Object obj){
        if (obj instanceof A3Prime){
            return this.x == ((A3Prime)obj).x;
        }
        else return false;
    }

    public static void main(String args[]){
        A3Prime a = new A3Prime(4);
        A3Prime b = new A3Prime(4);
        A3Prime c = new A3Prime(2);

        System.out.println("a equals b is: " + a.equals(b));
        System.out.println("a equals c is: " + a.equals(c));
        System.out.println("c equals b is: " + c.equals(b));

        c = b;
        System.out.println("c equals b is: " + c.equals(b));
    }
}
```

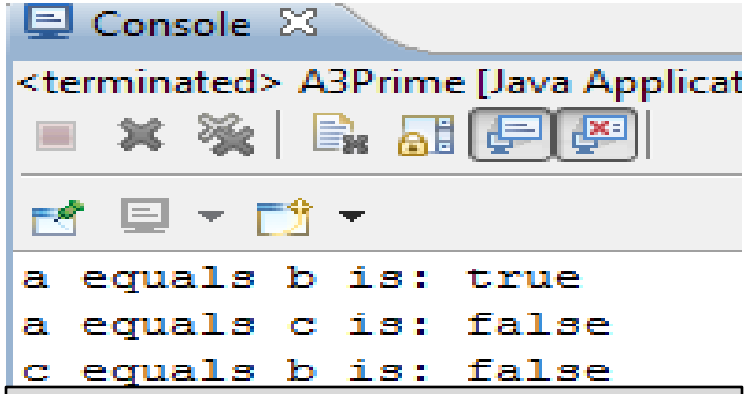
This method overrides the same method in the `Object` class. Note the same signature (see page 3).

These invocations are all to the `equals()` method in the `A3Prime` class which overrides the same method in the `Object` class.

More On Overriding/Overloading And The `Object` Class



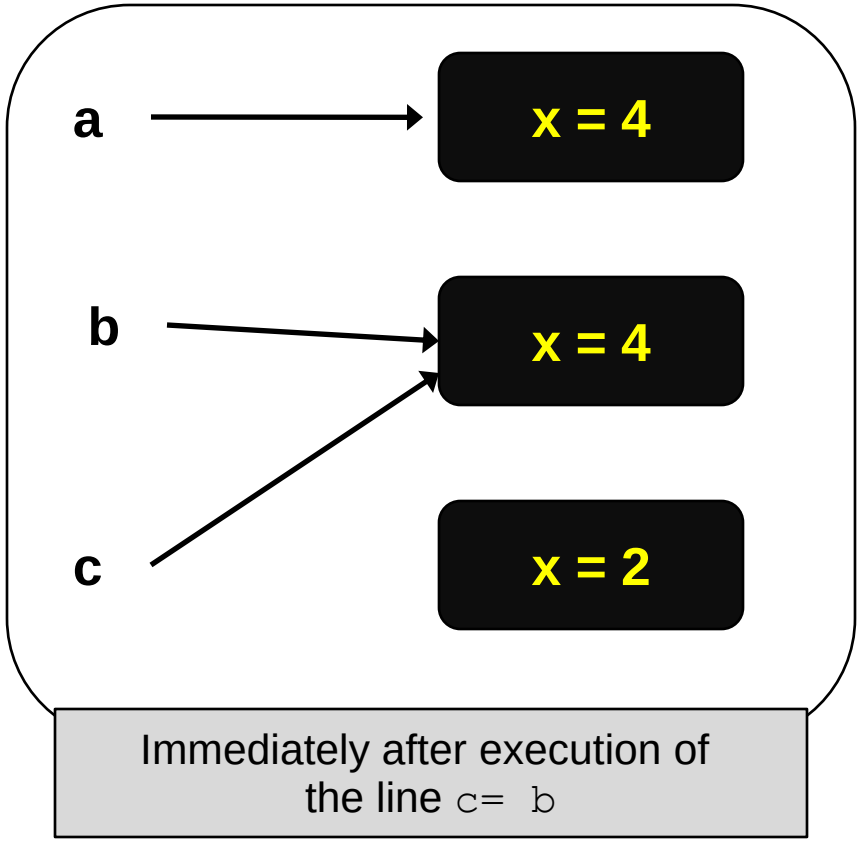
Immediately after creation of objects



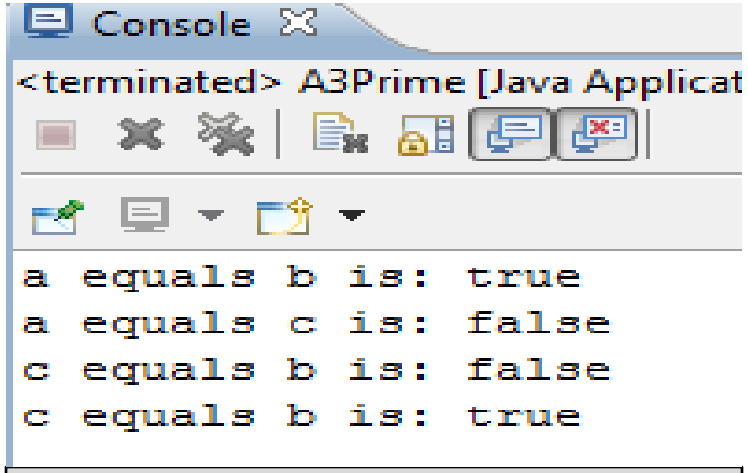
The first three lines of output

The first line of output from the calls to the `equals()` method indicate `true` since the values `a.x` and `b.x` are the same. The second and third lines indicate `false` since neither `a.b == c.x` nor does `c.x == b.x`.

More On Overriding/Overloading And The `Object` Class



Immediately after execution of the line `c= b`



The entire output

The last line of output now indicates a value of `true` returned by the call to the `equals()` method since the objects `b` and `c` refer to the same instance now, so `b.x == c.x`.

Polymorphism and Dynamic Binding

- Three of the most important concepts of object-oriented programming are encapsulation, inheritance, and polymorphism.
- We've already seen the first two concepts in some detail (we'll continue to use and expand on these as we continue through the semester), but now we want to focus for a bit on polymorphism.

Polymorphism and Dynamic Binding

- Before we look in detail at polymorphism, we need to define two terms: subtype and supertype.
- A class defines a type. A type defined by a subclass is called a subtype and a class defined by its superclass is called a supertype.
- All variables must have a declared type. The type of a variable is called its declared type.
- A variable of a reference type can hold a `null` value or a reference to an object (an object is an instance of a class).

Polymorphism and Dynamic Binding

- We've seen that an inheritance relationship enables a subclass to inherit features from its superclass with additional new features peculiar to the subclass.
- A subclass is a specialization of its superclass; every instance of a subclass is an instance of its superclass, but not vice versa.
  - For example, every circle is an object, but not every object is a circle.
- Therefore, you can always pass an instance of a subclass to a parameter of its superclass type.
- Let's look at the following example.

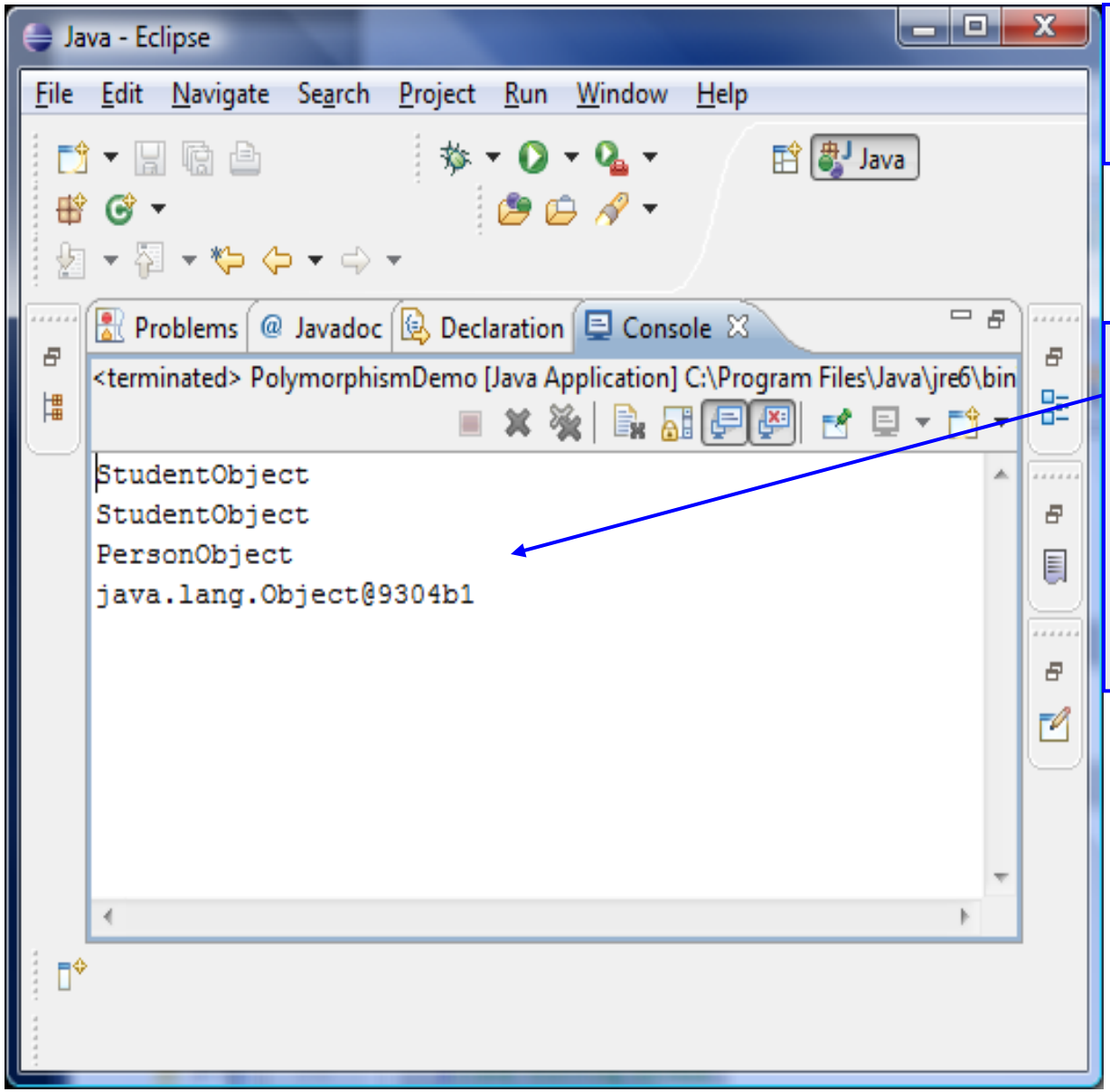
MyBikeShop.java | TestLoanClass.java | PolymorphismDemo.jav

```
public class PolymorphismDemo {
  public static void main(String[] args) {
    m(new GraduateStudentObject());
    m(new StudentObject());
    m(new PersonObject());
    m(new Object());
  }//end main method
  public static void m(Object x) {
    System.out.println(x.toString());
  }//end method m
}
class GraduateStudentObject extends StudentObject {
} //end class GraduateStudentObject

class StudentObject extends PersonObject {
  public String toString() {
    return "StudentObject";
  }//end toString method
}//end class StudentObject

class PersonObject extends Object {
  public String toString() {
    return "PersonObject";
  }//end toString method
}//end class PersonObject
```

PolymorphismDemo Example

What will the output of this program be?

PolymorphismDemo Example



Why does the output from this program look the way it does?

Read on to find out!

Discussion of the Example

- Method `m` takes a parameter of the `Object` type. You can invoke `m` with any object (e.g., `new GraduateStudentObject(), new StudentObject(), new PersonObject`, and `new Object()`, as shown).
- An object of a subclass can be used wherever its superclass object is required. This is commonly known as polymorphism (from a Greek work meaning "many forms").
- In simple terms, polymorphism means that a variable of supertype can refer to a subtype object.

Discussion of the Example

- When the method `m(Object x)` is executed, the argument `x`'s `toString` method is invoked, `x` may be an instance of `GraduateStudentObject`, `StudentObject`, `PersonObject`, or `Object`. Classes `GraduateStudentObject`, `StudentObject`, `PersonObject`, and `Object` have their own implementations of the `toString` method.
- Which implementation is used will be determined dynamically by the Java Virtual Machine (JVM) at runtime. This capability is known as dynamic binding.
- Let's look more closely at dynamic binding for a bit:

Dynamic Binding

- Suppose an object o is an instance of classes C_1, C_2, …, C_{n-1}, and C_n, where C_1 is a subclass of C_2, C_2 is a subclass of C_3, …, C_{n-1} is a subclass of C_n.
- Thus, C_n is the most general class, and C_1 the most specific class.

If o is an instance of C_1, o is an instance of C_2, C_3, . . . C_{n-1}, C_n

Dynamic Binding

- If the object `o` invokes a method `p`, the JVM searches the implementation for the method `p` in C_1, C_2, ..., C_{n-1}, and C_n, in this order until it is found.
- Once an implementation is found, the search stops, and the first-found implementation is invoked.
- Looking back at the example, when `m(new GraduateStudent())` is invoked, the `toString` method defined in the `Student` class is used.

Dynamic Binding

- Matching a method signature and binding a method implementation are two separate issues.
- The declared type of the reference variable decides which method to match at compile time. The compiler finds a matching method according to parameter type, number of parameters, and order of the parameters at compile time.
- A method may be implemented in several subclasses. The JVM dynamically binds the implementation of the method at runtime, decided by the actual class of the object referenced by the variable.

Dynamic Binding

- Dynamic binding enables a new class to be loaded on the fly without recompilation. There is no need for developers to create, and for users to install, major new software versions. New features can be incorporated transparently as needed.
- For example, suppose that we placed the classes `PolymorphismDemo`, `GraduateStudentObject`, `StudentObject`, and `PersonObject` in four separate files. If we modified the `GraduateStudentObject` class as follows:

```
class GraduateStudentObject extends StudentObject {
   public String toString() {
       return "Graduate Student";
   }
}
```


Dynamic Binding

- We now have a new version of `GraduateStudentObject` with a new `toString` method, but we don't have to recompile the classes `PolymorphismDemo`, `StudentObject`, or `PersonObject`.
- When you run `PolymorphismDemo`, the JVM dynamically binds the new `toString` method for the object of `GraduateStudentObject` when executing `m(new GraduateStudentObject())`.
- Try this yourself with these classes.

Dynamic Binding

- Polymorphism allows methods to be used generically for a wide range of arguments.
- This is known as generic programming.
- If a method's parameter type is a superclass (e.g., `Object`), you may pass to this method an object of any of the parameter's subclasses (e.g., `Student` or `String`).
- When an object (e.g., a `Student` object or a `String` object) is used in the method, the particular implementation of the method of the object invoked (e.g., `toString`) is determined dynamically.

Casting Objects and the `instanceof` Operator

- We've already used casting to convert variables of one primitive type to another primitive type. Casting can also be used to convert an object of one class type to another within an inheritance hierarchy.
- In the polymorphism example we just covered, the statement:

```
m(new StudentObject ());
```

assigns the object `new StudentObject ()` to a parameter of the

`Object` type. This statement is equivalent to:

```
Object o = new StudentObject ();  //implicit casting
m(o);
```


Casting Objects and the `instanceof` Operator

- The statement

```
Object o = new StudentObject ();
```

  is known as implicit casting. It is legal because an instance of `Student` is automatically an instance of `Object`.

- Suppose you want to assign the object reference `o` to a variable of the `Student` type using the following statement:

```
StudentObject b = o;
```

- In this case a compilation error would occur. Why does the statement at the top of this page work while the last statement does not?
  - The reason is that a `StudentObject` object is always an instance of `Object`, but an `Object` is not necessarily an instance of `StudentObject`.

Casting Objects and the `instanceof` Operator

- Even though you can see that `o` is really a `StudentObject` object, the compiler is not clever enough to know it. To tell the compiler that `o` is a `StudentObject` object, use an explicit casting.
- The syntax for an explicit casting of reference types is similar to the one used for casting among primitive data types. Enclose the target object type in parentheses and place it before the object to be cast, as follows:

```
StudentObject b = (StudentObject)o;  //explicit casting
```

- It is always possible to cast an instance of a subclass to a variable of a superclass (this is known as upcasting), because an instance of a subclass is always an instance of its superclass.

Casting Objects and the `instanceof` Operator

- When casting an instance of a superclass to a variable of its subclass (known as downcasting), explicit casting must be used to confirm your intention to the compiler with the `(SubclassName)` cast notation.
- For downcasting to be successful, you must make sure that the object to be cast is an instance of the subclass. If the superclass object is not an instance of the subclass, a runtime `ClassCastException` occurs.
  - For example, if an object is not an instance of `StudentObject`, it cannot be cast into a variable of `StudentObject`.

Casting Objects and the `instanceof` Operator

- It is a good practice, therefore, to ensure that the object is an instance of another object before attempting a casting.
- This can be accomplished by using the `instanceof` operator.

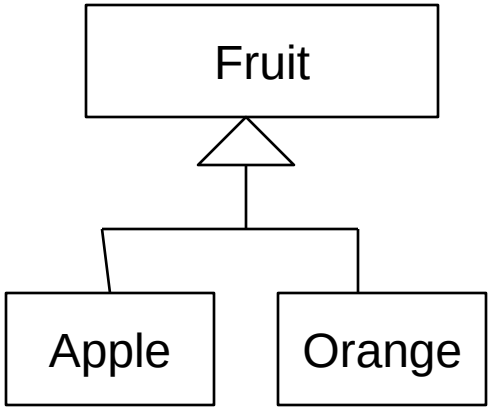
```
Object myObject = new Circle();
. . . //some lines of code
//perform casting if myObject is an instance of Circle
if (myObject instanceof Circle) {
   System.out.println("The circle diameter is " +
      ((Circle)myObject).getDiameter());
. . .
}
```


Casting Objects and the `instanceof` Operator

- Why is casting necessary at all in this example?
- Variable `myObject` is declared `Object`. The declared type decides which method to match at compile time. Using `myObject.getDiameter()` would cause a compilation error, because the `Object` class does not have the `getDiameter` method. The compiler cannot find a match for `myObject.getDiameter()`. So, it is necessary to cast `myObject` into the `Circle` type to tell the compiler that `myObject` is also an instance of `Circle`.
- Why not just declare `myObject` as a `Circle` type in the first place?
  - To enable generic programming, it is a good practice to declare a variable with a supertype, which can accept a value of any subtype.

Casting Objects and the `instanceof` Operator

- To help you understand casting a bit better, you might consider the analogy of fruit, apple, and orange with the `Fruit` class as the superclass for `Apple` and `Orange`. An apple is a fruit, so you can always safely assign an instance of `Apple` to a variable for `Fruit`. However, a fruit is not necessarily an apple, so you have to use explicit casting to assign an instance of `Fruit` to a variable of `Apple`.



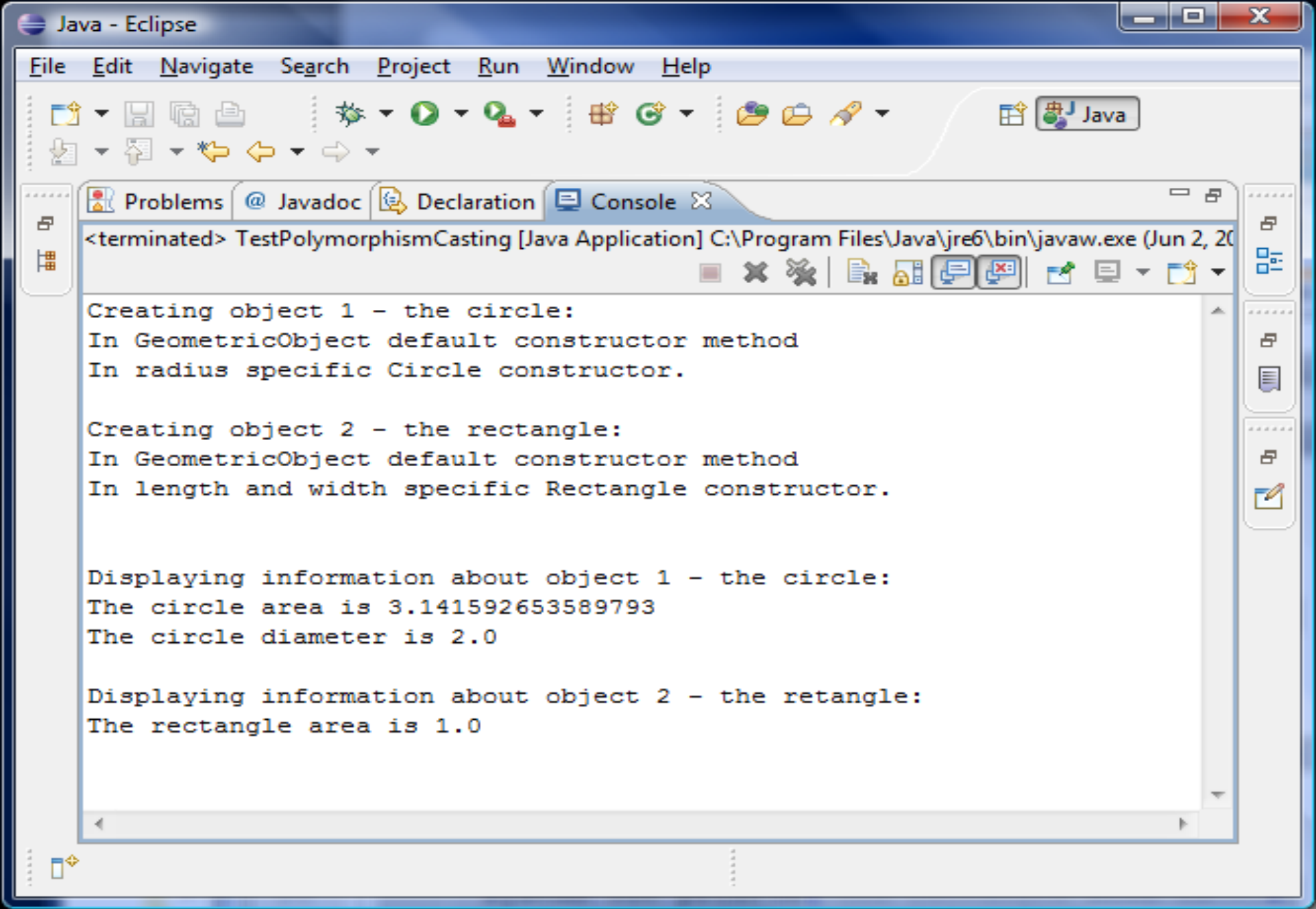
Casting Objects and the `instanceof` Operator

- A final example may also help to clarify casting objects.
- The program on the next page creates two objects, a circle and a rectangle, and invokes the `displayObject` method to display them.
- The `displayObject` method displays the area and diameter if the object is a circle and the area if the object is a rectangle.
- Be sure that you really understand this example.

MyBikeShop.java | TestLoanClass.java | PolymorphismDemo.jav | TestPolymorphismCast

```
public class TestPolymorphismCasting {

  public static void main(String[] args) {
    // Declare and initialize two objects
    System.out.println("Creating object 1 - the circle:");
    Object object1 = new Circle(1);
    System.out.println("\nCreating object 2 - the rectangle:");
    Object object2 = new Rectangle(1, 1);
    // Display circle and rectangle
    System.out.println("\n\nDisplaying information about object 1 - the circle:");
    displayObject(object1);
    System.out.println("\nDisplaying information about object 2 - the retangle:");
    displayObject(object2);
  }//end main method

  /** A method for displaying an object */
  public static void displayObject(Object object) {
    if (object instanceof Circle) {
      System.out.println("The circle area is " +
        ((Circle)object).getArea());
      System.out.println("The circle diameter is " +
        ((Circle)object).getDiameter());
    }//end if
    else if (object instanceof Rectangle) {
      System.out.println("The rectangle area is " +
        ((Rectangle)object).getArea());
    }//end if
  }//end method displayObject
}//end class TestPolymorphismCasting
```

This method is an example of generic programming. It can be invoked by passing any instance of Object.


Java - Eclipse
File Edit Navigate Search Project Run Window Help
Java
Problems Javadoc Declaration Console
<terminated> TestPolymorphismCasting [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jun 2, 20
Creating object 1 - the circle:
In GeometricObject default constructor method
In radius specific Circle constructor.
Creating object 2 - the rectangle:
In GeometricObject default constructor method
In length and width specific Rectangle constructor.
Displaying information about object 1 - the circle:
The circle area is 3.141592653589793
The circle diameter is 2.0
Displaying information about object 2 - the retangle:
The rectangle area is 1.0

Casting Objects and the `instanceof` Operator

- The program uses implicit casting to assign a `Circle` object to `object1` and a `Rectangle` object to `object2`.
- In the `displayObject` method, explicit casting is used to cast the object to `Circle` if the object is an instance of `Circle`, and the methods `getArea` and `getDiameter` are used to display the area and diameter of the circle.
- Since casting can only be done when the source object is an instance of the target class, the code uses the `instanceof` operator to ensure that the source object is an instance of the target class before performing the casting.
- Explicit casting to `Circle` and `Rectangle` is necessary because the `getArea` and `getDiameter` methods are not available in the `Object` class.

Casting Objects and the `instanceof` Operator

- A word of caution regarding the casting operation…

  the precedence of the object member access operator (the period) is higher than the casting operator. You must use parentheses to ensure that the casting is done before the . operation, as in:

```
((Circle)object).getArea());
```

and **not**: `(Circle)object.getArea();`