Inheritance and Polymorphism: Part 2

Stony Brook University

Inheritance: Overriding Methods

- A method is overridden if it is redefined in a derived class (child class) using the same method ***Signature***.

`Person` (parent class):

```
public void reset()
{   age = 0;          }
```

`Student` (child class):

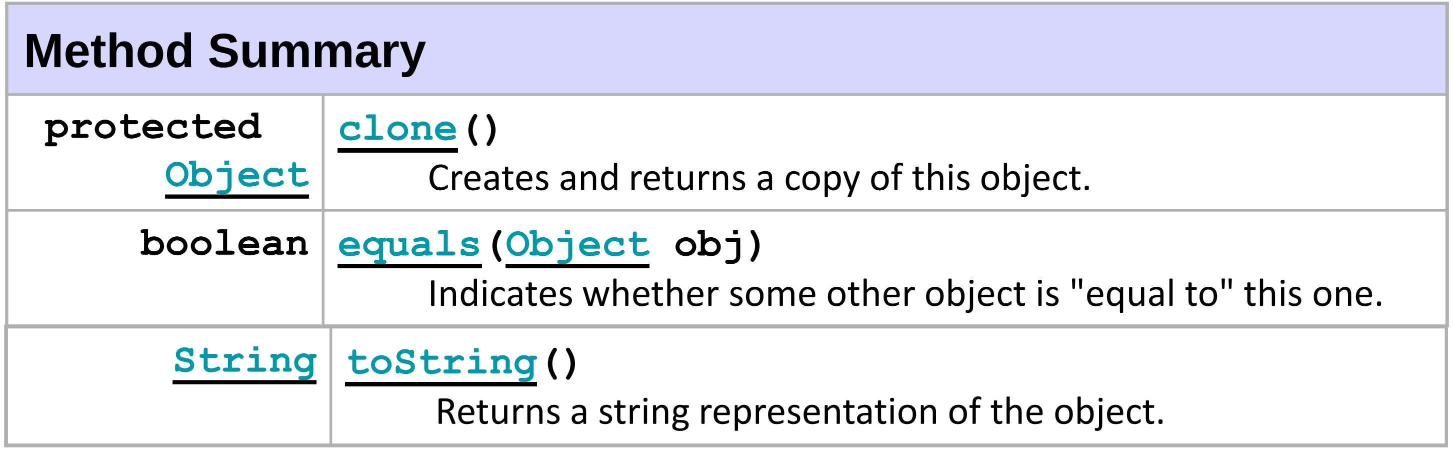
```
public void reset()
{   setAge(0);
    gpa = 0.0;        }
```

OR, Calling an overridden method:

```
public void reset()
{   super.reset();
    gpa = 0.0;        }
```

The Object class

- Every class is a subclass of the `java.lang.Object` class, even if not specified directly.



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.
`String`	`toString()` Returns a string representation of the object.

- The 3 methods above are overridden nearly every time a new class is defined. The Object class also contains 8 other methods we won’t use

Overriding toString() method Example

```
public class Person
{ …
  public String toString()
  {   return name + ", age " + age; }
}

public class Student extends Person
{ …
  public String toString()
  { return super.toString() + ", GPA: "
  + gpa; }
}
```

```
public class Employee
  extends Person
{ …
  public String
  toString()
  {  return
  super.toString() + ",
  Salary: $" + salary; }
}
```

Printing using toString()

```
public class StoogePrinter
{
  public static void main(String[] args)
  {
   Person moe = new Person("Moe Stooge");
   Student larry = new Student("Larry Stooge");
   Employee curly = new Employee("Curly Stooge");

   System.out.println(moe.toString());
   //System.out.println(moe);
   System.out.println(larry.toString());
   System.out.println(curly);
   }
}
```

```
OUTPUT:  Moe Stooge, age 0
         Larry Stooge, age 0, GPA: 0.0
         Curly Stooge, age 0, Salary: $0
```

Exercise to try at home

```java
public class Course
{ public int courseNumber = 114; }

public class Classroom
{ public String building = "Javits";
  public int roomNumber = 100;
  public String toString()
  { return building + roomNumber; }
}

public class LectureHall extends Classroom
{ public int capacity = 650; }

public class ToStringExample
{ public static void main(String[] args)
  {
  System.out.println(new Course());
  System.out.println(new Classroom());
  System.out.println(new LectureHall());
  }
}
```

What output do you get?

Multiple Inheritance?

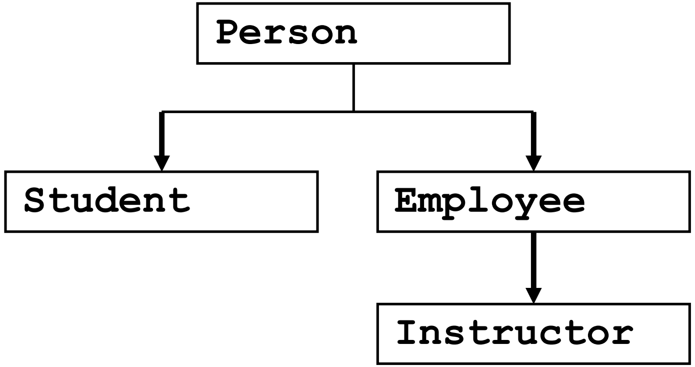
A class may **`extend`** only 1 class, however, when it does so it also inherits all methods and variables that the parent class has already inherited
Example:

```
public class Instructor extends Employee
{  private String dept;

   public Instructor(String initName)
   {     super(initName);      dept = "None Assigned";   }

   public String getDept()                        { return dept;
    }
   public void setDept(String newDept)   { dept = newDept; }

   public String toString()
   {     return super.toString() + ", Dept: " + dept; }

   public static void main(String[] args)
   {     Instructor in = new Instructor("John Smith");
         System.out.println(in);         }
}
```

```
OUTPUT:
   John Smith, age 0, Salary:
  $0, Dept: None Assigned
```

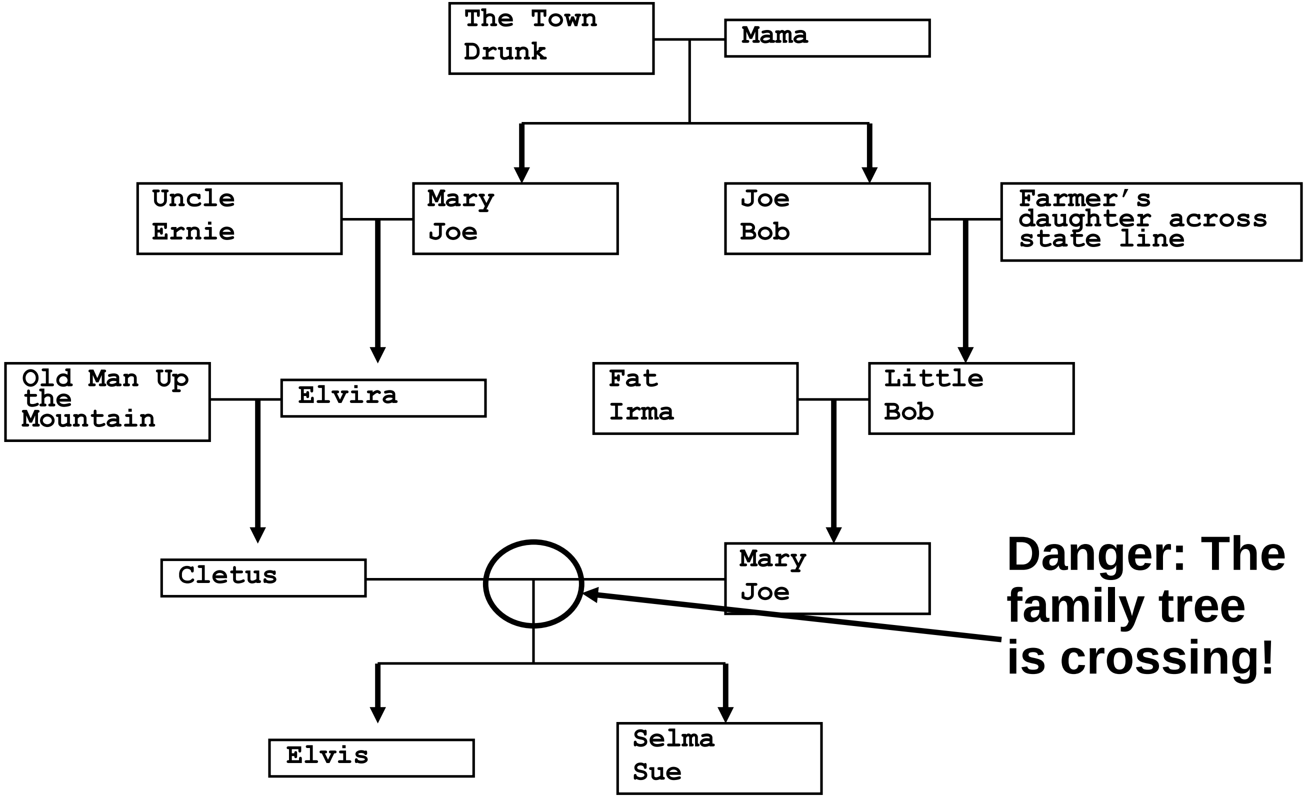
Inheritance Diagram

- Think of an inheritance diagram as a family tree for classes, except for a couple of differences:
  - A class may only have 1 immediate parent
  - No criss-crossing in a class tree
  - Every class has all the properties (state and behavior) of all of its ancestors (so much for Darwinism)



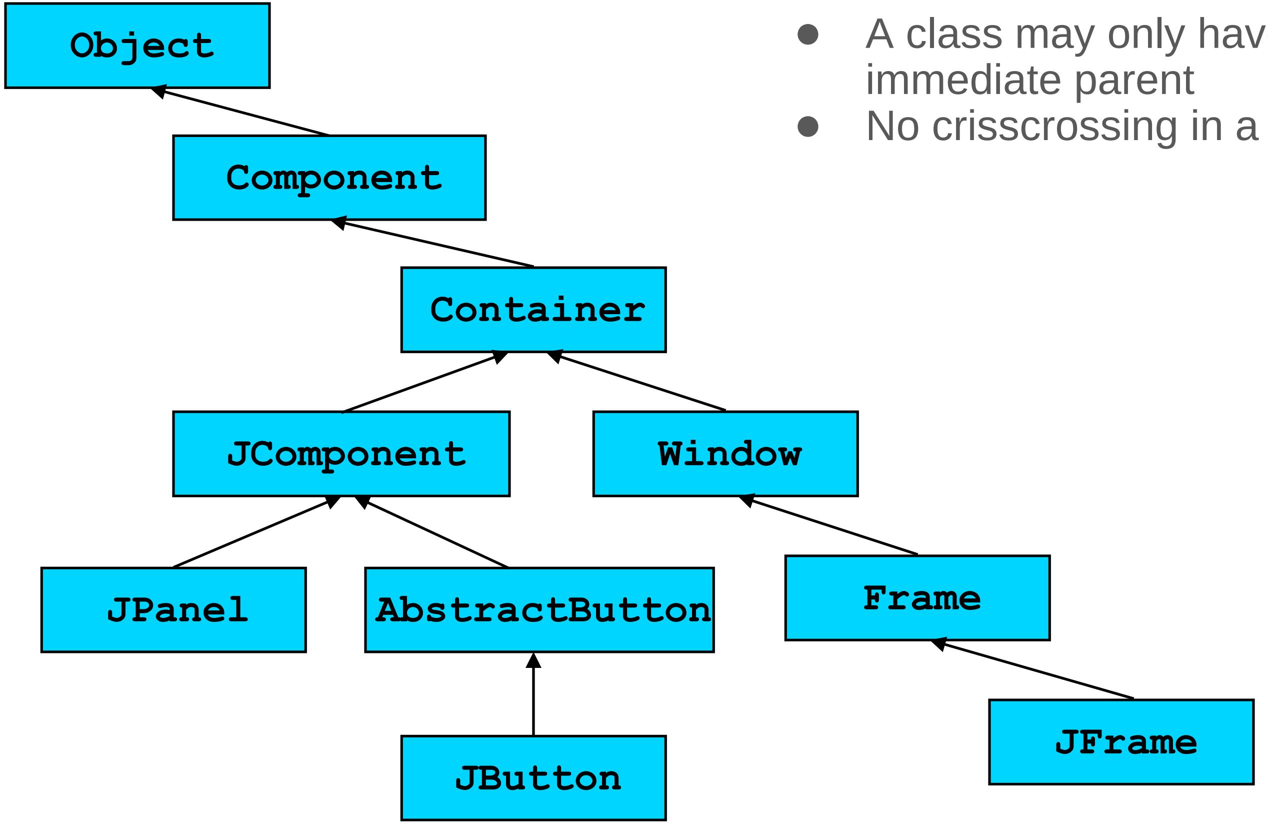
A class may have any number of ancestors, in this case `Instructor` has 2.

Inheritance Diagram- family tree?



Danger: The family tree is crossing!

Inheritance Diagram- Example



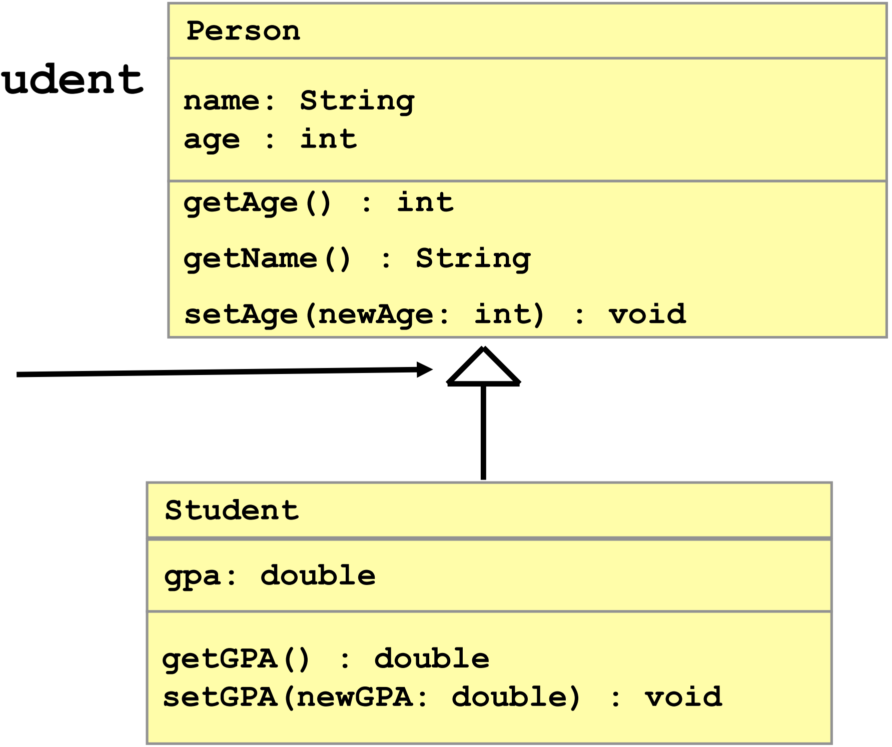
- A class may only have 1 immediate parent
- No crisscrossing in a class tree

UML class Diagram and Inheritance

```
public class Student
extends Person
```

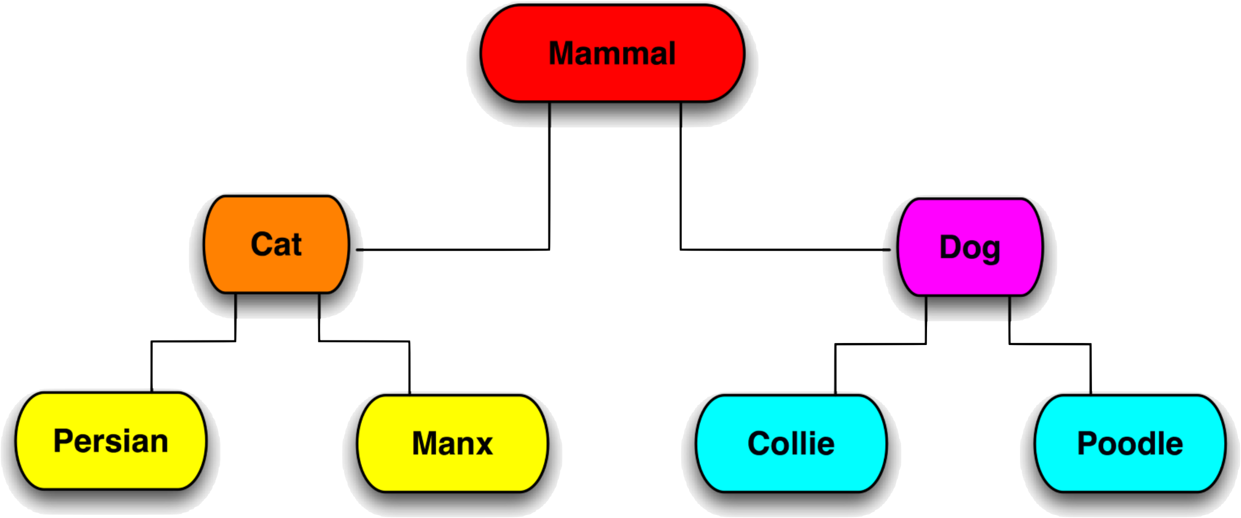
Triangle denotes inheritance

`Student` IS-A `Person`



```
Person
name: String
age : int
getAge() : int
getName() : String
setAge(newAge: int) : void

Student
gpa: double
getGPA() : double
setGPA(newGPA: double) : void
```

Cats and Dogs (yes, again)



Cats and Dogs Revisited

```
public class Cat extends Mammal
{       ...
        public void makeSound()
  {
                System.out.println(“Meow!”);
        }
}

Public class Dog extends Mammal
{       ...
        public void makeSound()
  {
                System.out.println(“Woof!”);
        }
}
```

Cats and Dogs Revisited (II)

```
Cat garfield = new Cat();
Dog odie = new Dog();

garfield.makeSound();
odie.makeSound();

Mammal m = new Cat();
m.makeSound();
```

What does each call to `makeSound()` produce?

Polymorphism

- Polymorphism is the ability of a language to have **duplicate method names** in an inheritance hierarchy and to decide which method is appropriate to call depending on the **class of the object** to which the method is applied.

Assignment

- We can always assign a derived class to any parent class variable
  - the derived class IS-A parent object

- Mammal m = new Cat();
  - Cat meets all of Mammal’s criteria, plus Cat-specific features

- **We CANNOT go the other way, though!**

- **Cat c = new Mammal(): // Illegal**

Assignment (II)

- c expects to point to an object with all of Cat’s features

- There is no guarantee that Mammal has all of those features
  - In fact, it probably doesn’t

Method invocation

- Suppose that Mammal and Cat both define a makeSound() method
    - Cat’s version overrides Mammal’s

- **Which version of makeSound() will be called?**

- **Mammal m = new Mammal();**

- **Mammal x = new Cat();**

Basic Rules

- Every variable has ***two types***:
    - ***reference type (declared type)***
    - ***actual type***

- The ***reference type determines what methods can be called.***

- The ***actual type determines which versions of those methods are called.***

Examples

- Mammal m = new Mammal();
  - Reference and actual types are both Mammal

- Mammal x = new Cat();
  - ***Reference type: Mammal***
  - ***Actual type: Cat***

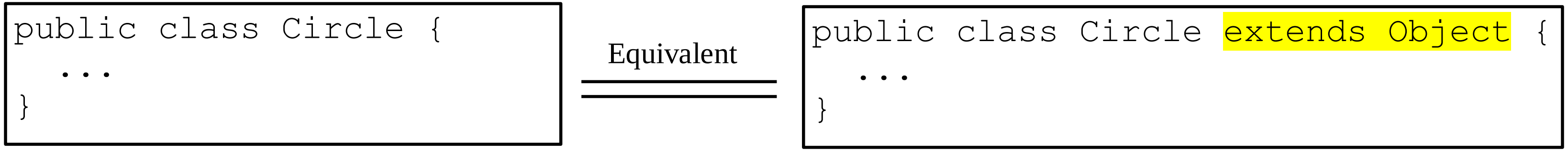
Polymorphism

- Polymorphism means that ***you can send the same message to different objects***
  - Inheritance means that objects share the same code

- ***Results*** may ***vary depending upon the specific object involved***
  - Cat responds differently from Dog

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 **`java.lang.Object`**



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

Equivalent

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

Polymorphism, Dynamic Binding and Generic Programming

```
public class PolymorphismDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }
  public static void m(Object x) {
    System.out.println(x.toString());
  }
}
class GraduateStudent
         extends Student {
}
class Student extends Person {
  public String toString() {
    return "Student";
  }
}

class Person extends Object {
  public String toString() {
    return "Person";
  }
}
```

Method m takes a parameter of the Object type – can be invoked with any object

Polymorphism: an object of a subtype can be used wherever its supertype value is required

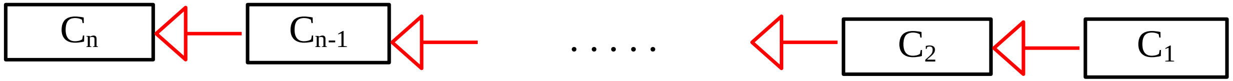
Dynamic binding: the Java Virtual Machine determines dynamically at runtime which implementation is used by the method When the method m(Object x) is executed, the argument x's toString method is invoked.

Output:

```
Student
Student
Person
java.lang.Object@12345678
```

Dynamic Binding

- Suppose an object **o** is an instance of classes C_1, C_2, ..., C_{n-1}, and C_n
  - ○ C_1 is a subclass of C_2, C_2 is a subclass of C_3, ..., and C_{n-1} is a subclass of C_n
  - ○ C_n is the most general class, and C_1 is the most specific class
  - ○ **If 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, the search stops and the first-found implementation is invoked**



Since o is an instance of C_1, o is also an instance of $C_{2,}$ $C_{3,}$ **. . . ,** C_{n-1}**,** and C_n

Dynamic Binding: Example

```
public class PolymorphismDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }
  public static void m(Object x) {
    System.out.println(x.toString());
  }
}
class GraduateStudent extends Student {
}
class Student extends Person {
  public String toString() {
    return "Student";
  }
}
class Person extends Object {
  public String toString() {
    return "Person";
  }
}
```

Output:

```
Student
Student
Person
java.lang.Object@12345678
```

Method Matching Vs. Binding

- The compiler **finds a matching method** according to parameter type, number of parameters, and order of the parameters **at compilation time**

- The Java Virtual Machine **dynamically binds the implementation of the method at runtime**

Generic Programming

```
public class PolymorphismDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }
  public static void m(Object x) {
    System.out.println(x.toString());
  }
}
class GraduateStudent extends Student {
}
class Student extends Person {
  public String toString() {
    return "Student";
  }
}
class Person extends Object {
  public String toString() {
    return "Person";
  }
}
```

Generic programming: polymorphism allows methods to be used generically for a wide range of object arguments: if a method’s parameter type is a superclass (e.g.,Object), **you may pass an object to this method of any of the parameter’s subclasses** (e.g., Student or String) and the particular implementation of the method of the object that is invoked is determined dynamically

Casting Objects

- Casting can be used to convert an object of one class type to another within an inheritance hierarchy

```
    m(new Student());
```

is equivalent to:

```
  Object o = new Student();     // Implicit casting
  m(o);
```

Legal because an instance of Student is automatically an instance of Object

Why Casting is Necessary

```
    Student b = o;
```

- A **compilation error** would occur because an Object o is not necessarily an instance of Student

- We use **explicit casting** to tell the compiler that o is a Student object - syntax is similar to the one used for casting among primitive data types

```
    Student b = (Student)o;
```

- This type of casting may not always succeed (check this with `instanceof` operator)

The instanceof Operator

- Use the **`instanceof`** operator to test whether an object is an instance of a class:

```
Object myObject = new Circle();
...
if (myObject instanceof Circle) {
 System.out.println("The circle diameter is "
            + ((Circle)myObject).getDiameter());
  ...
}
```

```
public class CastingDemo{
        public static void main(String[] args){
                Object object1 = new Circle(1);
                Object object2 = new Rectangle(1, 1);
                displayObject(object1);
                displayObject(object2);
        }
        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());
                }else if (object instanceof Rectangle) {
                        System.out.println("The rectangle area is " +
                                ((Rectangle)object).getArea());
                }
        }
}
```

The `equals` Method

- The **`equals()`** method compares the **contents** of two objects - the default implementation of the **`equals`** method in the Object class is as follows:

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

- **Override the `equals()` method** in other classes:

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

Stony Brook
University