COP 3330: Object-Oriented Programming
Summer 2011

Introduction to Object-Oriented Programming and the Unified Modeling Language (UML)

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

A Brief Introduction to UML

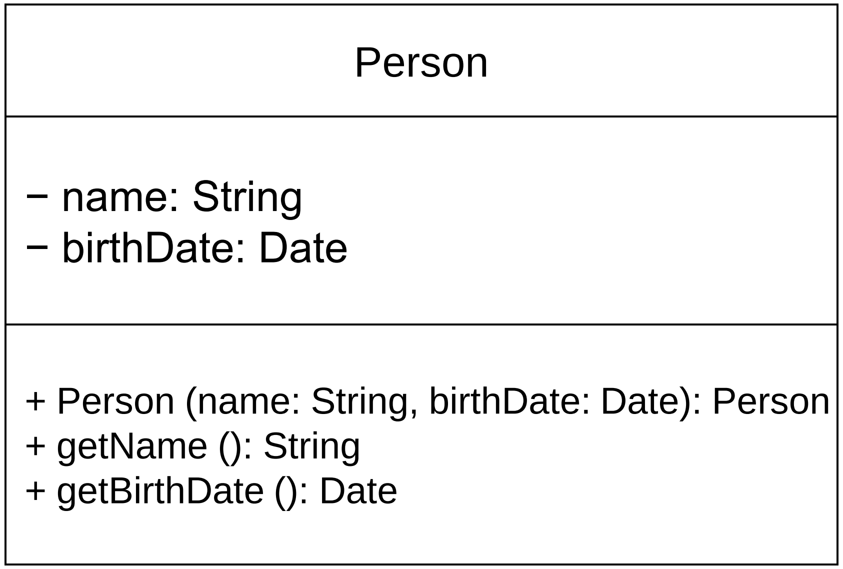
- It is not easy for software designers to keep in mind all of the important properties of classes and the relationships that exist between the classes as the number of classes and relationships grow in a system.
- To aid in visualizing the design, a diagram can be very helpful. The standard notation, or language, that is used for these diagrams is called the Unified Modeling Language, or UML for short.
- There are 13 different types of diagrams included in the UML 2.0 standard (the current standard adopted in 2003). Among the thirteen are class diagrams, state diagrams, and sequence diagrams. These three are the most useful types of diagrams for OO-program developers. For right now, we focus only on class diagrams.
- UML is not a Java-only modeling language, so its notations do not always correspond directly to Java notation or syntax. For example, a method named `practice` that takes an integer `x` as a parameter and returns a `String` is written in Java as: `String practice (int x)`

  but in UML is written as: `practice (x: int): String`

UML Class Diagrams

- A UML class diagram shows classes, interfaces and the relationships between them.
- A class diagram provides a static view of the classes and relationships rather than a dynamic view of the interactions among the objects of those classes.
- A class is represented by a rectangle (box) divided into three sections horizontally.
  - The top section gives the name of the class.
  - The middle section gives the attributes (fields) of the objects of the class. These fields are abstractions of the data or state of an object and as such are usually implemented as instance variables. However, class variables are also represented here.
  - The bottom section gives the operations ("intelligence") of the class, which corresponds to the constructors and methods in Java.
- The example on the next page shows the UML class diagram for the Person class we created in the previous set of notes.

UML Class Diagrams



UML Class Diagram

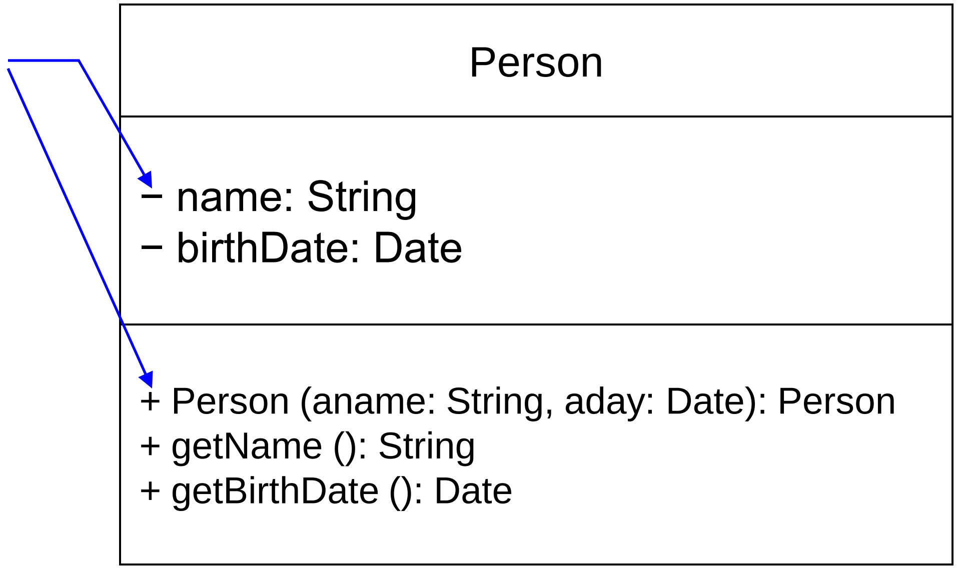
```
public class Person  {
    private String name;
    private Date birthDate;

    public Person (String who, Date bday)   {
         this.name = who;
         this.birthDate = bday;
    }

    public String getName()   {
        return name;
    }

    public Date getBirthDate()  {
        return birthDate;
    }
}
```

The equivalent Java code

UML Class Diagrams

Accessibility modifiers:

− indicates private
+ indicates public
# indicates protected
~ indicates package



```
Person
-------------------------------------------
− name: String
− birthDate: Date
-------------------------------------------
+ Person (aname: String, aday: Date): Person
+ getName (): String
+ getBirthDate (): Date
```

UML Class Diagram

Class variables or class methods are indicated by underlining the class variable or class method.

There are other optional parts to UML class diagrams, including a 4th section that would be below the methods in which the responsibilities of the class are outlined. You don’t see this too often, but it is available and is useful when transitioning from CRC cards (class, responsibilities, collaborators) which is a modeling tool used to decide what crc’s are needed (more later).

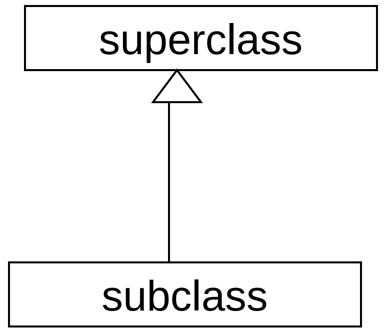
UML Class Diagrams

- Class diagrams are also used to show relationships between classes.
- A class that is a subclass of another class is connected to that class by an arrow with a solid line for its shaft and with a triangular hollow arrowhead. The arrow points from the subclass to the superclass. In UML, such a relationship is called a generalization.
- A similar arrow except using a dashed line for the arrow shaft is used to indicate implementation of an interface. In UML, such a relationship is called a realization.
- An association between two classes means that there is a structural relationship between them. Associations are represented by solid lines. Associations have many optional parts. Both the association and each of its ends can be labeled. Arrows on either or both ends of an association indicate navigability. Also, each end of an association line can have a multiplicity value displayed. An association might also connect a class with itself, using a loop. Such an association indicates that the connection of an object of the class with other objects of the same class.

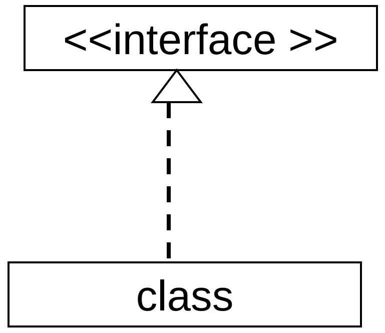
UML Class Diagrams

- An association with an arrow on one end indicates one-way navigability. The arrow means that from one class you can easily access the second associated class to which the association points, but from the second class, you cannot necessarily easily access the first class.
  - Another way to think about this is that the first class is aware of the second class, but the second class is not necessarily directly aware of the first class.
- An association with no arrows usually indicates a two-way association, but it may also means that navigability is not important and was simply left off the diagram.
- The multiplicity of one end of an association means the number of objects of that class associated with the other class. A multiplicity is specified by a nonnegative integer or a range of integers. A multiplicity specified by “0..1” means that there are 0 or 1 objects on that end of the association. Other common multiplicities are “0..*”(0 or more), “1..*” (1 or more), and “*” (shorthand for 0 or more).

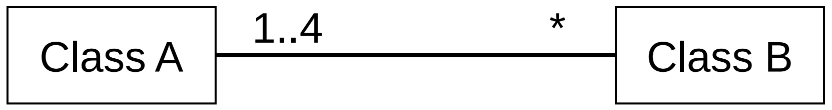
UML Class Diagrams



A generalization



A realization

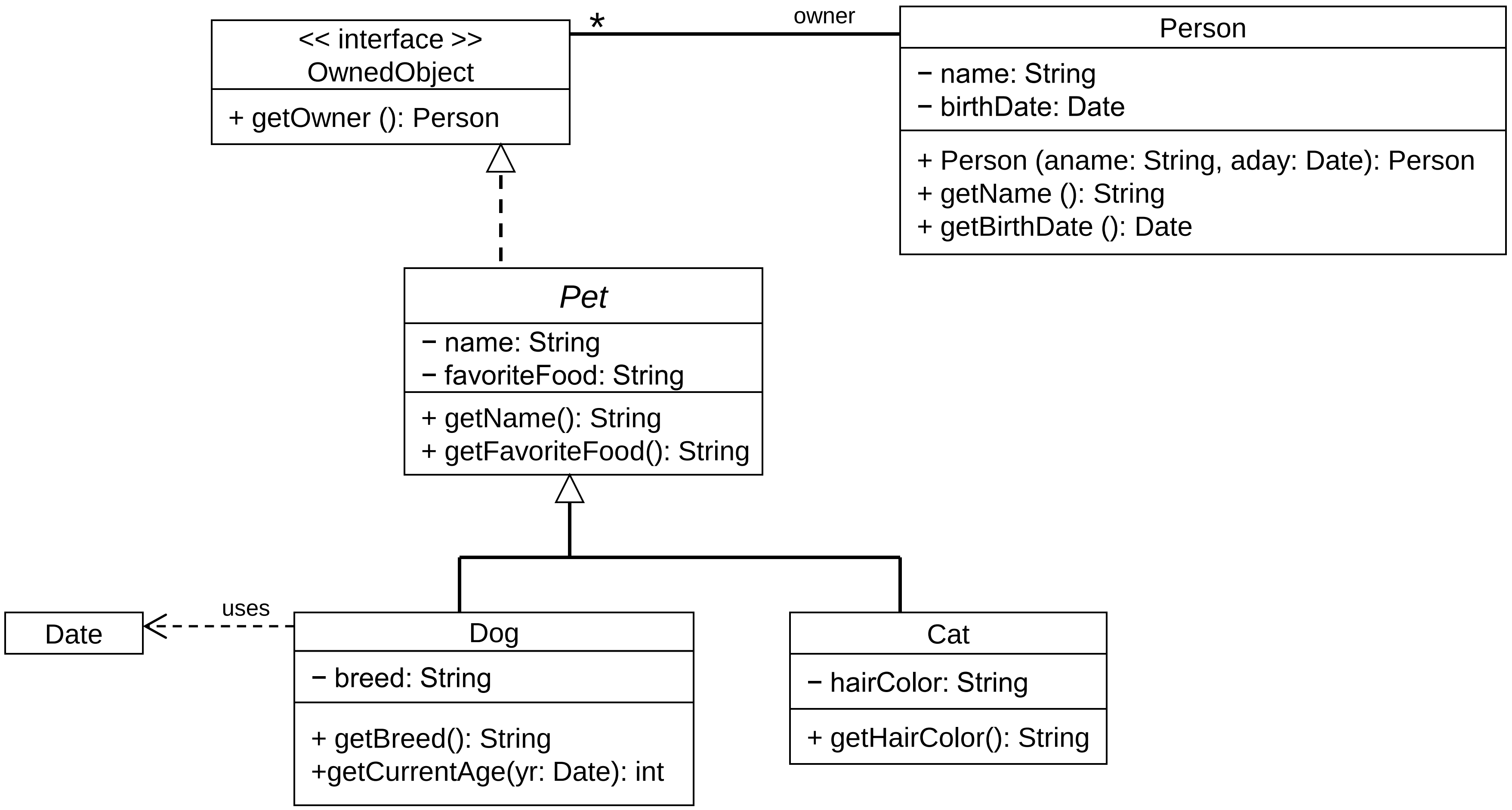


An association with two-way navigability with members of class B related to between 1 and 4 members of class A and a member of class A being related to 0 or more members of class B.

UML Class Diagrams

- Another connection besides an association between classes that can be displayed in a class diagram is the dependency relationship. A dependency is indicated by a dashed line (with optional arrows and optional labels).
- One class depends on another if changes to the second class might require changes to the first class.
  - Note: An association from one class to another automatically indicates a dependency, and so no dashed line is needed between classes if there is already an association between them. However, for a transient relationship, i.e., for a class that does not maintain any long-term connection to another class but does use that class occasionally, you should draw the dependency from the first class to the second class. In the example that follows, the Dog class uses the Date class whenever its getCurrentAge method is invoked, and so the dependency is labeled “uses”.
- Abstract classes or abstract methods are indicated by using italics for the name.
- An interface is indicated by adding the phrase <<interface>> (called a stereotype) above the name.

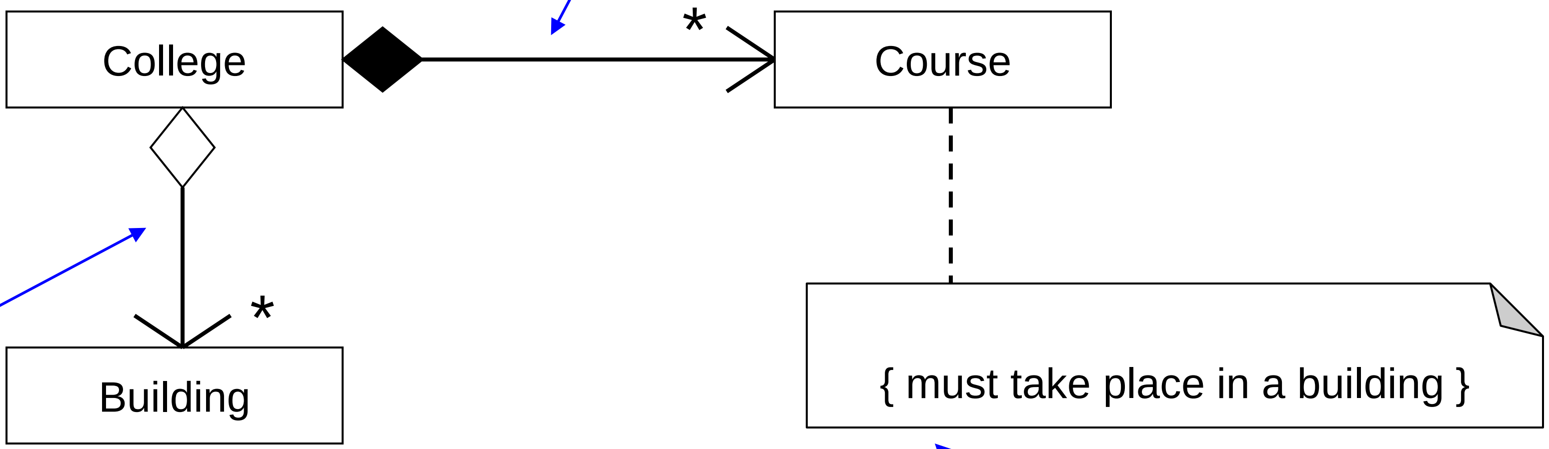
UML Class Diagrams



UML Class Diagrams

- An aggregation is a special kind of association indicated by a hollow diamond on one end of the association link. It indicates a "whole/part" relationship, in that the class to which the arrow points is considered "part" of the class at the diamond end of the association.
- A composition is an aggregation indicating strong ownership of the parts. A composition is indicated by a solid diamond on the "owner" end of the association. In a composition, the parts live and die with the owner because they have no role in the software system independent of the owner.
- Another fairly common element of a class diagram is a note, which is represented by a box with a dog-eared corner that is connected to other elements with a dashed line. It can have arbitrary content (text and graphics) and is similar to a comment in a programming language. It might contain comments about the role of a class or constraints that all objects of the class must satisfy. If the contents are a constraint, the contents are surrounded by braces.

UML Class Diagrams



A composition – a course does not exist outside of the college at which it is offered.

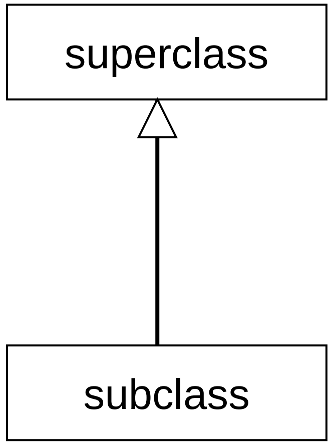
An aggregation – college consists of buildings that make up the campus. If the college were to close the buildings would still exist to be used for some other purpose.

A note that is describing a constraint to be applied to each course object.

Implementation Inheritance

- One of the most significant features of OO programming is implementation inheritance or subclassing.
- Inheritance greatly increases the reusability of classes and also minimizes the duplication of code.
- This is just an introduction to inheritance, we'll examine it in much greater detail later.

Implementation Inheritance



UML notation for subclass and superclass

- A subclass inherits all of the features of its superclass.
- This means all of the variables and methods, but not the constructors.
  - The constructor in the superclass must be invoked to create a superclass object before the constructor for the subclass can specialize the subclass object.
  - If you think about what inheritance means for a minute, this will make sense – the superclass object must exist before it can be turned into a specialized subclass object.

Specialization

- Let’s consider the following example of a software developer (you!) who has been assigned to create a drawing program in which rectangles can grow, shrink, or move around on a panel under the control of the user.
- In order to deal with the rectangles, it is useful to have a Rectangle class that stores the relevant information about the rectangle such as its size and position.
- Since our developer is smart, they do not immediately code a Rectangle class from scratch, but instead spend a few minutes looking through existing libraries to see if there is already a Rectangle class that can be used.
- Sure enough, there are several Rectangle classes in the Java libraries, including:

  `java.awt.Rectangle`,

  `java.awt.geom.Rectangle2D.Double`,

  and `java.awt.geom.Rectangle2D.Float`.

Specialization

- After studying these classes, you determine that `java.awt.Rectangle` is the closest one to satisfying your needs.
- However, you want a class with a `getCenter()` method and a `setCenter(int x, int y)` method and the Rectangle class does not include such methods.
- *What should you do to get what you want with minimal effort?*

Specialization

- Option #1: If the source code for the existing Rectangle class is available, you could modify the class to suit your needs, including adding the new methods and possibly deleting any methods that you don't need.
- Option #2: You could copy the Rectangle class code and insert it into a new class named `EnhancedRectangle` and then add the new code.

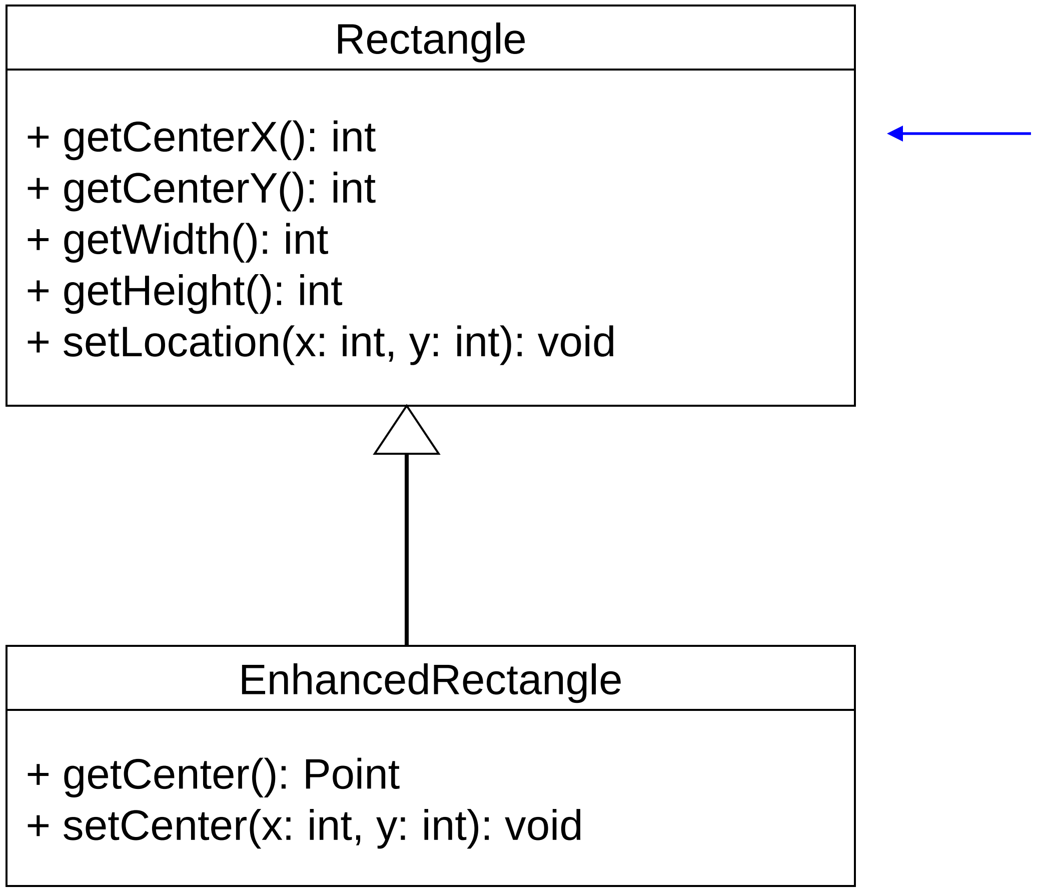
Specialization

- Code reuse is always a very appropriate action to take – however, neither of these techniques are the correct way to reuse code! They both have inelegant aspects (remember we are writing to write only elegant, high-quality code here).
- Problems with option #1: This approach could cause problems with existing code that uses the original Rectangle class – there are now two versions of Rectangle floating around to confuse users and possibly the compiler as well.
- Problems with option #2: This approach is better than the first in that the new class will not affect existing code that uses the original Rectangle class, but there is major code duplication in this case. The code duplication introduces unnecessary complexity (remember that one of the properties of elegance is simplicity). For example, if the original code is found to have bugs, the programmer is going to have to remember to fix the bugs in the copied code.
- Furthermore, neither of these approaches will work if only the compiled code and not the source code for the Rectangle class is available. *So now what?*

Specialization

- One solution would be to be to simply ignore the compiled code and define and implement a new EnhancedRectangle class.
- However, this approach does nothing in terms of code reuse and also results in significant code duplication. While we don't necessarily have exact duplication of the method bodies, we do have duplication of semantics, which can be just as bad.
  - Also, since we would assume that the original Rectangle class was thoroughly tested and we now may face a considerable effort to construct a new class to bring it up to the error-free level of the existing class for the original functionality.
- *So what is the solution to our problem?*
- *Answer:* Use implementation inheritance (available in any OO language), that will allow you to define a new class as a subclass of another class. In this case we want to create a new class named EnhancedRectangle that will be a subclass of the existing class Rectangle (the superclass). A subclass inherits all the features (variables and methods) of its superclass.

Specialization



All of the instance variables and methods of the Rectangle class that are not relevant to this scenario are simply omitted from the UML diagram.

UML diagram for this scenario

Specialization

Subclassing in Java is expressed using the keyword “extends”

```
public class EnhancedRectangle extends Rectangle
{
    //constructor
    public EnhancedRectangle (int x, int y, int w, int h)
    {
        super(x, y, w, h);  //invoke constructor in superclass
    }
    public Point getCenter()
    {
        return new Point((int) getCenterX(), (int) getCenterY());
    }
    public void setCenter(int x, int y)
    {
        setLocation(x-(int) getWidth()/2, y-(int) getHeight()/2;
    }
}
```

Java implementation for this scenario

Specialization

- In the Java code on the previous page, the first line declaration makes the class EnhancedRectangle a subclass of Rectangle and makes Rectangle a superclass of EnhancedRectangle.
- Because it is a subclass, the new EnhancedRectangle class inherits all of the methods and all of the data in the Rectangle class.
- Note that since constructors are not inherited, you must create a constructor for the new subclass.
  - If you do not specify a constructor in a class, Java will automatically create a no argument default constructor that will allow generic objects of the class to be created. In general, you should specify the constructor.
- The call to `super(x,y,w,h)` in the constructor method of the EnhancedRectangle class invokes the superclass constructor to initialize all the Rectangle data. (Remember we cannot specialize a subclass instance unless we have first created an instance of the superclass.)

Specialization

- Notice that the `getCenterX`, `getCenterY`, `setLocation`, `getWidth`, and `getHeight` methods that are used in the Java code to implement the two new methods `getCenter` and `setCenter` methods are all inherited from the Rectangle class.

- Now the clients of the EnhancedRectangle class can use it as follows:

```
EnhancedRectangle rectangle = new EnhancedRectangle(1,2,50,60);
rectangle.setLocation(10,10);  //inherited method
rectangle.setCenter(60,80);  //subclass method
```

- Note that EnhancedRectangle objects behave as if all methods inherited from the Rectangle class have been defined in their class.

Specialization

- In this way, subclassing provides a way to reuse the code and data of an existing class to create a new class that is identical except that it has more features (data and/or behavior). This process of extending an existing class by adding new features is called using inheritance for specialization.

Special Notes on Inheritance in Java

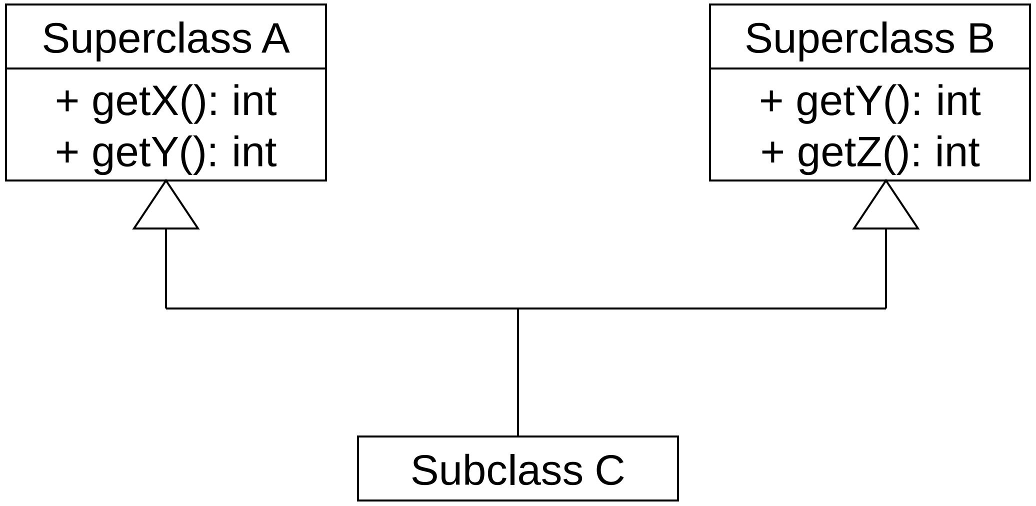
- All Java classes that do not explicitly extend another class implicitly extend the Object class.
- Therefore, all Java classes extend the Object class either directly or indirectly via one or more intermediate classes in an inheritance chain.
- This means that any Java class will automatically inherit the methods in the Object class: `clone`, `equals`, `finalize`, `getClass`, `hashcode`, `notify`, `notifyAll`, `toString`, and three versions of `wait`.

Special Notes on Inheritance in Java

- A Java class can have only one superclass. This is called single inheritance. It means that you can’t inherit some methods from one class and some other methods from a different class using subclassing.

- Single inheritance can interfere with your attempts at code reuse. This is a shortcoming of Java, but it serves the purpose of keeping the implementation of classes and inheritance simple and also simplifies the understanding of such code.

Special Notes on Inheritance in Java



Multiple Inheritance (not allowed in Java)

The developer of class C would like to be able to have a getX, getY, and getZ method available to objects in class C. This is not allowed in Java. Another problem with multiple inheritance is illustrated by the method getY. If method getY is invoked in class C, which version of getY would be used?

Types, Subtypes, and Interface Inheritance

- Another of the most powerful concepts of OO programming is subtype polymorphism. In order to understand this concept, it is important to fully understand what is meant by a "type".
- A type can be thought of as a set of data values and the operations that can be performed on them. For example, the int primitive type in Java can be thought of as the set of all 32-bit integers (values ranging from -2,147,483,648 to +2,147,483,647) together with the set of operations that can be performed on integers, including, for example, addition, subtraction, multiplication, and division.
- For objects, types can be defined similarly, except the focus is more on the operations than on the values. For our purposes, an object type will consist of a set of operations and a set of objects that can perform those operations.
-

Types, Subtypes, and Interface Inheritance

- There are two standard ways in Java to define new types.

1. Any class C implicitly forms a type C. The set of public methods of the class form the set of operations for type C and the objects of class C or its subclasses form the set of objects of that type.

   - For example, the class Person that we built on page 4, defines a type "Person" with operations getName() and getBirthDate(). All objects of class Person or its subclasses can perform these two operations, and these objects form the set of objects of type Person.

Types, Subtypes, and Interface Inheritance

2. The other way to define a type is to use Java interfaces. An interface can be thought of as a named set of operations. All objects whose classes explicitly "implement" the interface form the set of objects of that type.

- For example, the following interface defines a type Runnable.

```
public interface Runnable
{
        public void run();
}
```

Types, Subtypes, and Interface Inheritance

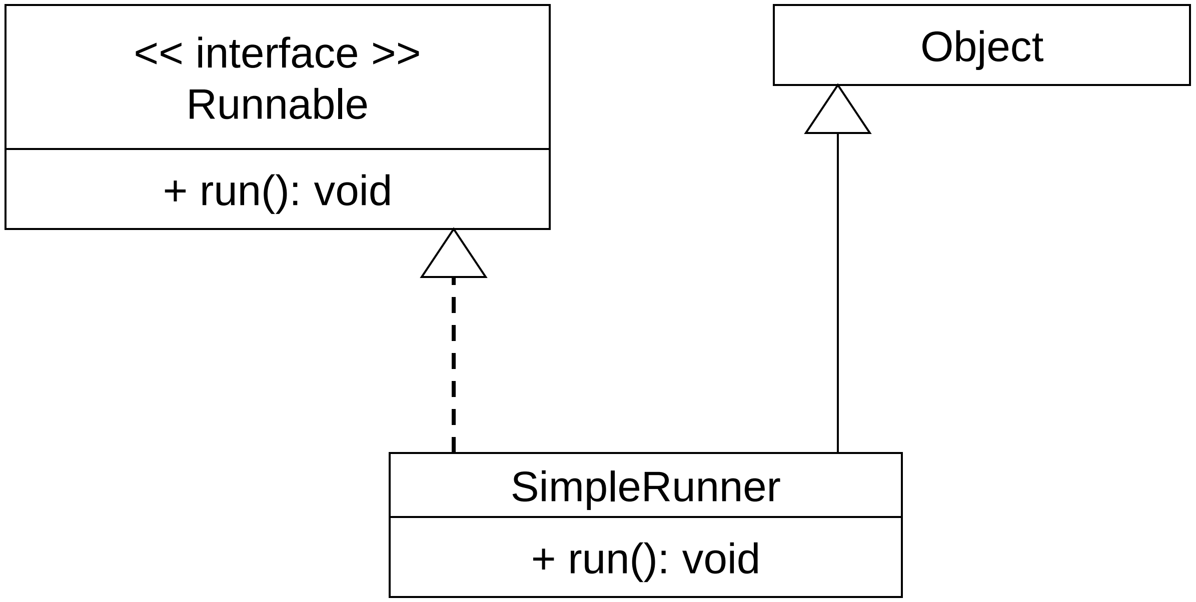
- The operations of type Runnable consist of just the run() method. The set of objects of type Runnable consists of all classes that implement Runnable. For example, consider the following class SimpleRunner.

```
public class SimpleRunner implements Runnable
{
        public void run()
        {
            System.out.println("I'm running.");
        }
}
```

Types, Subtypes, and Interface Inheritance

- The class SimpleRunner defines and implements a method run() of the form required by the Runnable interface and the class explicitly declares that it “implements Runnable”.
- Therefore, all object of class SimpleRunner can be considered as objects of type Runnable. In fact, since SimpleRunner is also a subclass of the Object class, objects of class SimpleRunner have three types: SimpleRunner, Runnable, and Object.

Types, Subtypes, and Interface Inheritance



UML diagram showing the Runnable interface and the SimpleRunner class

Types, Subtypes, and Interface Inheritance

- Objects of a subclass S of a class T are considered to be both of type S and of type T.
- There is a special relationship between the type of a subclass and the type of a superclass, in that the a subclass of a class defines a subtype of the superclass type.
- In other words, one type S is a subtype of another type T (which, in turn, is called a supertype of S) if the set of objects of type S are a subset of the set of objects of type T and the set of operations of S are a superset of the operations of T.
  – Note that, if type S is a subtype of T, then set of operations of S must include all the operations of T and can possibly include more. For example, the type SimpleRunner is a subtype of Runnable since all objects of type SimpleRunner are also objects of type Runnable since SimpleRunner includes all operations in the Runnable type. Similarly, SimpleRunner is a subtype of Object since it includes (inherits) all operations in Object and its objects are a subset of the set of all Objects.

Types, Subtypes, and Interface Inheritance

- It should also be noted that interfaces can also have subinterfaces that inherit from them similar to the way inheritance works with classes.
- For example, consider the following interface:

```
public interface Movable extends Runnable
   {
          public void walk();
    }
```

- The movable interface defines a new interface with two operations: its walk() operation and the run() operation that it inherits from Runnable.
- As you might expect, a subinterface defines a subtype of the type defined by the superinterface.

Polymorphism

- Object oriented programming languages support these notions of types and subtypes that we’ve just seen by allowing a variable of one type to store and object of a subtype.
- For example, in the Java statement:

```
Runnable r = new SimpleRunner;
```

the variable r of type Runnable refers to an object of the actual class SimpleRunner.

- The fact that an object of a subtype can be legally used wherever an object of a supertype is expected is called subtype polymorphism.

UML – Practice Problem 1

- Extend the UML class diagram on page 10 in the following manner:

1. Include another abstract class named `Toys` that implements the interface `OwnedObject`. The abstract class `Toys` should have two subclasses, one named `Bikes`, the other named `Karts`. Each of these subclasses has a single private attribute named `type` which is a `String`. Each subclass has a single accessor method (a "getter" method) which simply returns the current value of the `type` of each object.

2. Add another private attribute to the `Cat` class which is a `String` type that gives the cat's name. Also provide an accessor method to allow any object to get a cat's name.

UML – Practice Problem 2

- Draw a UML class diagram for a university scenario where students take classes taught by instructors.
  - Students take courses.
  - Instructors teach courses.
  - Students are divided into two subclasses: undergraduate and graduate.
  - Graduate students are divided into two subclasses: master's students and doctoral students.
  - Provide an accessor method for every attribute in each class that you develop. All accessor methods should be public and all attributes should be private.
  - Assume that a student can take 0 or more courses. A course can be taken by 0 or more students.
  - Assume that an instructor can teach between 0 and 4 courses. A course is taught by only 1 instructor.