Method overloading

- A class can contain distinct methods with the same name as long as invocations of these methods can be distinguished by the compiler.
- Overloading
- Each overloaded method must differ from each other in the number and/or type of its parameters.
  - public int report (int x)
  - public int report (Object obj)
  - public void report (int x, int y)
- These are completely separate, independent methods.
- The number and types of arguments determine which method is invoked.
  - report(2) invokes the first, report("2") invokes the second, and report(2,3) invokes the third

Method overloading ...

- Note that a class cannot contain two methods with the same name and the same number of parameters, even if the methods have different return types.
- Because Java supports coercion (implicit type conversion) and subtyping, it is not always possible to differentiate methods that differ only by their return type.
- A class cannot contain the following method definitions (even though one is a query method and the other is a command):
  - public void report (Object obj)
  - public Object report (int i)

Method overloading ...

- Overload resolution is static.
  - The method to be executed is determined by the compiler based on the static types of the argument expressions.
  - It does not depend on the actual run-time argument values.
  - Suppose a class Reporter has two methods named report defined as follows:
    - public void report (Object obj) {
        System.out.println("The argument is an object.");
    }
    - public void report (Circle circle) {
        System.out.println("The argument is a circle.");
    }

Method overloading ...

- Assume Circle is a subclass of Figure and consider the following definitions and initializations:
  - Circle circle = new Circle(1);
  - Figure figure = circle;
  - Object object = figure;
- Note that all three have the same value:
- Given the declaration Reporter reporter;
  - consider the following statements:
    - reporter.report(circle);
    - reporter.report(object);
    - reporter.report(figure);

Method overloading ...

- Which method is invoked for each of these calls?
  - reporter.report(circle);
  - reporter.report(object);
  - reporter.report(figure);
We have seen that a subclass inherits all the (public) features of its parent class.

For instance, every Java class comes equipped with an equals() and a toString() method.

The equals() method is effectively defined as:

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

The reason every class has this method is that the equals() method is defined for the class Object, and every class is a subclass of Object.

Method overriding

- We have seen that a subclass inherits all the (public) features of its parent class.
- For instance, every Java class comes equipped with an equals() and a toString() method.
- The equals() method is effectively defined as:
  ```java
  public boolean equals (Object obj) {
      return this == obj;
  }
  ```
- The reason every class has this method is that the equals() method is defined for the class Object, and every class is a subclass of Object.

Method overriding ...

- A class can redefine the implementation of a method that it inherits.
- Such a redefinition is called overriding.
- The redefinition must have the same number and type of parameters as the original (inherited) definition.
- The return type of the redefinition must be the same as the original return type.
  - In Java 1.5, the return type of the redefinition can be a subtype of the original return type.
- Write a toString() method for your TimidPlayer class.

Method overriding ...

- The intent of equals() method is to determine if the content of two objects is the same.
- The behavior of equals() in the Object class is identical to ==
- We normally override equals() to test the content of the objects, that is, to do a field by field comparison.
- Rules for overriding equals():
  - x.equals(x) should return true
  - If x.equals(y) returns true, then y.equals(x) should also return true
  - If x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true
  - x.equals(null) should return false

What makes sense for an equals() method in the Circle class?

```java
public class Circle extends ClosedFigure {
    ...
    public boolean equals (Object c) {
        return c instanceof Circle &&
            this.radius() == ((Circle)c).radius();
    }
    ...
}
```

What makes sense for an equals() method in the Rectangle class?

```java
public class Rectangle extends ClosedFigure {
    ...
    public boolean equals (Object r) {
        return r instanceof Rectangle &&
            this.length() == ((Rectangle)r).length() &&
            this.width() == ((Rectangle)r).width();
    }
    ...
}
```
Method overriding...

ClosedFigure f1;
ClosedFigure f2;
Scanner in = new Scanner(System.in);
int n = in.nextInt();
if (n == 0) {
f1 = new Circle(...);
f2 = new Circle(...);
} else {
f1 = new Rectangle(...);
f2 = new Rectangle(...);
}
boolean b = f1.equals(f2);

Method overriding...

The abstract takeTurn() method is inherited and overridden in each implementing class.
- Each class provides its own implementation of the takeTurn() method, exactly as Circle and Rectangle provide their own implementations of the equals() method.
- When the following statement:
  nextPlayer.takeTurn(pile, MAX_ON_A_TURN);
  is executed, the actual takeTurn() method that is performed depends on the object that nextPlayer references.
- Unlike overload resolution, override resolution is dynamic.

Method overriding...

- Overloading and overriding are easily confused.
- Consider the following attempt to override the equals() method of the Object class:
  public class Circle extends ClosedFigure {
      public boolean equals (Circle c) {
          return this.radius() == c.radius();
      }
  }
- This definition overloads the equals() method inherited from Object, but it does not override it.

Method overriding...

- Overloading and overriding are different aspects of the language.
- Overloading involves several distinct methods with the same name in a single class.
  - Which method is invoked is determined by the compiler, and depends on the (static) type of the argument expressions.
- Overriding involves a single method with different implementations in different classes.
  - Which implementation is performed depends on the object executing the method at run-time.

Method overriding...

- polymorphism:
  - dynamic behavior by which the method performed as the result of a call to a given object is determined at run-time by the class of the object.
- Polymorphism applies to classes in the same way it applies to interfaces.
- Recall that the Player interface was implemented by several classes:
  - TimidPlayer
  - GreedyPlayer
  - CleverPlayer
- The Player interface defines the takeTurn() method:
  public void takeTurn (Pile pile, int maxOnATurn);

Method overriding...

- With the above definition of equals(), the Circle class has two distinct methods named equals():
  public boolean equals (Object obj) {
      return this == obj;
  }
  public boolean equals (Circle c) {
      return this.radius() == c.radius();
  }

Method overriding...

- Inherited from Object
- Explicitly defined here
**Method overriding ...**

- Given:
  
  ```java
  Circle circle1 = new Circle(1);
  Circle circle2 = new Circle(1);
  Object object2 = circle2;
  ```

- The following evaluates to *true*:
  ```java
  circle1.equals(circle2)
  ```

- The following evaluates to *false*:
  ```java
  circle1.equals(object2)
  ```

**Method overriding ...**

- Casting changes the static type of the expression, and hence the method to be invoked.
- The following evaluates to *false*:
  ```java
  circle1.equals((Object)circle2)
  ```

- The following evaluates to *true*:
  ```java
  circle1.equals((Circle)object2)
  ```

- What does `object2.equals(circle1)` evaluate to?

**Feature accessibility**

- To this point, all the features defined for a class:
  - methods
  - instance variables
  - named constants
  have been labeled either *public* or *private*.
- Public features are part of the class specification and are visible to clients of the class.
- Private features are part of the implementation and are not available to clients.

**Feature accessibility ...**

- Java provides two other accessibility categories for class features:
  - *protected*
    - Protected features are intended to be inherited by and accessible in a subclass
  - *package private* (or *restricted*)
    - Package private features are accessible in the class's package.