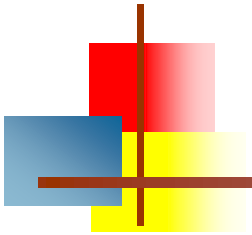
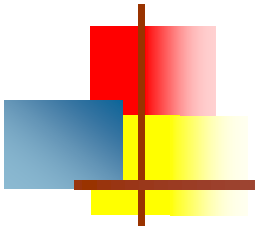


Abstract Classes and Interfaces

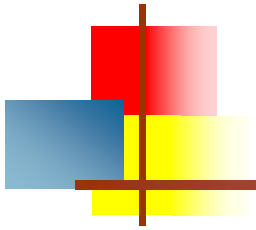


Sample Problem

- Say you need to design a system for a shop that sells different types of bicycles. You will most likely set up a Bicycle class that defines what a bike can and cannot do in terms of "bicycleness" (I.e. number of wheels, etc.). You will also most likely create some subclasses of bicycle like (RacingBike, MountainBike, HybridBike) to hold specialized characteristics.

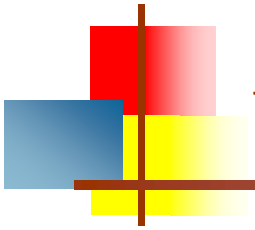
- But bicycles interact with the world on other terms. For example, say you want to include your Bicycle class in an inventory system. For this system you will want to include inventory characteristics with the bicycle (and its subclasses), such as price, tracking number and quantity. In OOA you would want to create a Product class and have each type of bicycle be a subclass. So, for example, you would want to make MountainBike a subclass of Product.

- But wait, this is a problem…why?



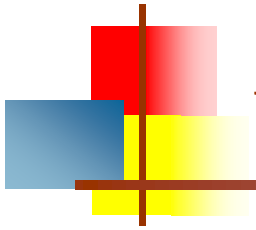
Sample Problem

- You've already extended MountainBike from Bicycle. How can you make this class extend from Product as well?
- Problem: Java does not allow multiple inheritance like C++ does.
- The Java designers did this on purpose. Multiple inheritance can cause a lot of problems and make things inefficient.
- The solution they came up with was Interfaces which give most of the benefits of multiple inheritance without the complexities and inefficiencies.



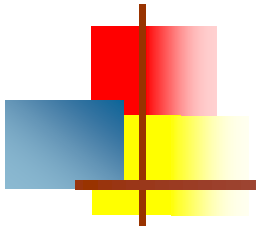
What are Interfaces?

- Interfaces:
  - are Java's way of overcoming the problem of multiple inheritance.
  - describe what a class should do without telling it how to do it.
  - are devices or systems that unrelated entities use to interact
    - for example, a remote control is an interface between you and your tv
  - are a set of requirements and are analogous to a protocol (an agreed upon behavior)
    - some other languages actually call their interfaces protocols
- To sum up the concept of interfaces:
  - Knowing how to use something means knowing how to interface with it
  - The methods an instance can respond to can be defined as the interface to the instance
  - Interfaces are used to assure that an instance has a defined set of methods:



What are Interfaces?

- So, let’s go back to our Bicycle example:
  - An inventory program doesn't care what class of items it manages as long as each item provides certain information, such as price and tracking number.
  - Instead of forcing class relationships on otherwise unrelated items, the inventory program sets up a protocol of communication. This protocol comes in the form of a set of constant and method definitions contained within an interface.
  - The inventory interface would define, but not implement, methods that set and get the retail price, assign a tracking number, and so on.
  - To work in the inventory program, the bicycle class must agree to this protocol by implementing the interface.
  - When a class implements an interface, the class agrees to implement all the methods defined in the interface. Thus, the bicycle class would provide the implementations for the methods that set and get retail price, assign a tracking number, and so on.

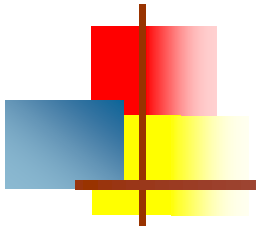


Interfaces

- An interface declares (describes) methods but does not supply bodies for them

```
interface KeyListener {
    public void keyPressed(KeyEvent e);
    public void keyReleased(KeyEvent e);
    public void keyTyped(KeyEvent e);
}
```

- All the methods are implicitly public and abstract
  - You can add these qualifiers if you like, but why bother?
- You cannot instantiate an interface
  - An interface is like a *very* abstract class—*none* of its methods are defined
- An interface may also contain constants (final variables)

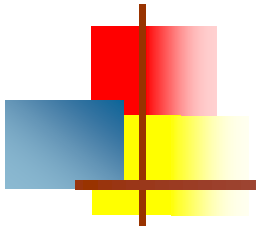


Designing interfaces

- Most of the time, you will use Sun-supplied Java interfaces
- Sometimes you will want to design your own
- You would write an interface if you want classes of various types to all have a certain set of capabilities
- For example, if you want to be able to create animated displays of objects in a class, you might define an interface as:

```
public interface Animatable {
    install(Panel p);
    display();
}
```

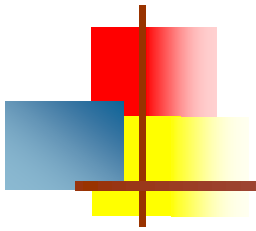
- Now you can write code that will display *any* Animatable class in a Panel of your choice, simply by calling these methods



Implementing an interface I

- You extend a class, but you implement an interface
- A class can only extend (subclass) one other class, but it can implement as many interfaces as you like
- Example:

```
class MyListener
        implements KeyListener, ActionListener { … }
```

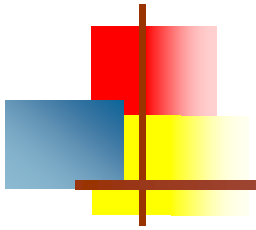


Implementing an interface II

- When you say a class implements an interface, you are promising to *define* all the methods that were *declared* in the interface
- Example:

```
class MyKeyListener implements KeyListener {
      public void keyPressed(KeyEvent e) {...};
      public void keyReleased(KeyEvent e) {...};
      public void keyTyped(KeyEvent e) {...};
}
```

  - The "..." indicates actual code that you must supply
- Now you can create a new MyKeyListener

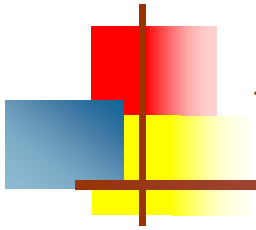


Partially implementing an Interface

- It is possible to define some but not all of the methods defined in an interface:

```
abstract class MyKeyListener implements KeyListener {
    public void keyTyped(KeyEvent e) {...};
}
```

- Since this class does not supply all the methods it has promised, it is an abstract class
- You must label it as such with the keyword abstract
- You can even *extend* an interface (to add methods):
  - interface FunkyKeyListener extends KeyListener { ... }

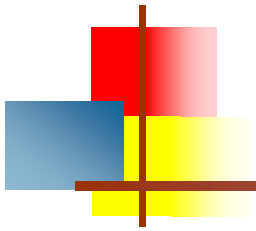


What are interfaces for?

- Reason 1: A class can only extend one other class, but it can implement multiple interfaces
  - This lets the class fill multiple "roles"
  - In writing Applets, it is common to have one class implement several different listeners
  - Example:

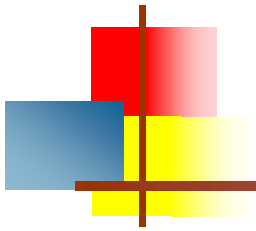
```
class MyApplet extends Applet
        implements ActionListener, KeyListener {
...
}
```

- Reason 2: You can write methods that work for more than one kind of class



How to use interfaces

- You can write methods that work with more than one class
- interface RuleSet { boolean isLegal(Move m, Board b);
  void makeMove(Move m); }
  - Every class that implements RuleSet must have these methods
- class *CheckersRules* implements RuleSet { // one implementation
  public boolean isLegal(Move m, Board b) { ... }
  public void makeMove(Move m) { ... }
  }
- class *ChessRules* implements RuleSet { ... } // another implementation
- class *LinesOfActionRules* implements RuleSet { ... } // and another
- RuleSet rulesOfThisGame = new ChessRules();
  - This assignment is legal because a rulesOfThisGame object *is* a RuleSet object
- if (rulesOfThisGame.isLegal(m, b)) { makeMove(m); }
  - This statement is legal because, *whatever* kind of RuleSet object rulesOfThisGame is, it *must* have isLegal and makeMove methods

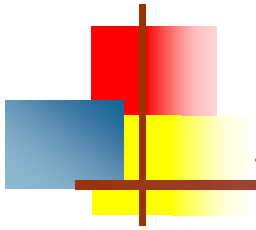


Interfaces, again

- When you implement an interface, you promise to define *all* the functions it declares
- There can be a *lot* of methods

```
interface KeyListener {
    public void keyPressed(KeyEvent e);
    public void keyReleased(KeyEvent e);
    public void keyTyped(KeyEvent e);
}
```

- What if you only care about a couple of these methods?

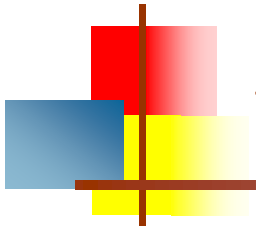


Adapter classes

- Solution: use an adapter class
- An adapter class implements an interface and provides empty method bodies

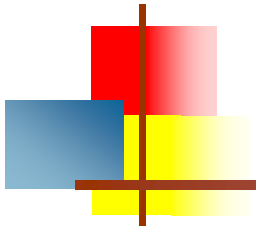
```
class  KeyAdapter implements KeyListener {
    public void keyPressed(KeyEvent e) { };
    public void keyReleased(KeyEvent e) { };
    public void keyTyped(KeyEvent e) { };
}
```

- You can override only the methods you care about
- This isn't elegant, but it does work
- Java provides a number of adapter classes



Vocabulary

- abstract method—a method which is declared but not defined (it has no method body)
- abstract class—a class which either (1) contains abstract methods, or (2) has been declared abstract
- instantiate—to create an instance (object) of a class
- interface—similar to a class, but contains only abstract methods (and possibly constants)
- adapter class—a class that implements an interface but has only empty method bodies



The End

Complexity has nothing to do with intelligence, simplicity does.

— Larry Bossidy

Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.

— Antoine de Saint Exupery