

Computer Science 210:  
Data Structures

# Summary

- Today
  - writing a Java program
  - guidelines on clear code
  - object-oriented design
  - inheritance
  - polymorphism
  - this
  - exceptions
  - interfaces
  
- **READING:** GT chapter 2

# Object-Oriented Design

- In an object-oriented language you model/design the world using classes.
- To create the world you instantiate classes thus creating objects. Objects respond to events and this determines how your world behaves.
  - Each class models one part of the world.
  - Usually in a project there is one class that creates the world---it creates the objects and starts the initial events (e.g. timer events); after that the world evolves.
- You model and create your project's world. Your design **goals** are:
  - **Robustness**
    - your world is capable of handling unexpected inputs without crashing
    - your world recovers gracefully from errors
  - **Adaptability**
    - your world can be changed/adapted to new requirements
  - **Reusability**
    - your world is general/simple enough so that it can be re-used
- Note: Code sharing is good.
  - avoids re-inventing the wheel
  - reliable (code is debugged many times)

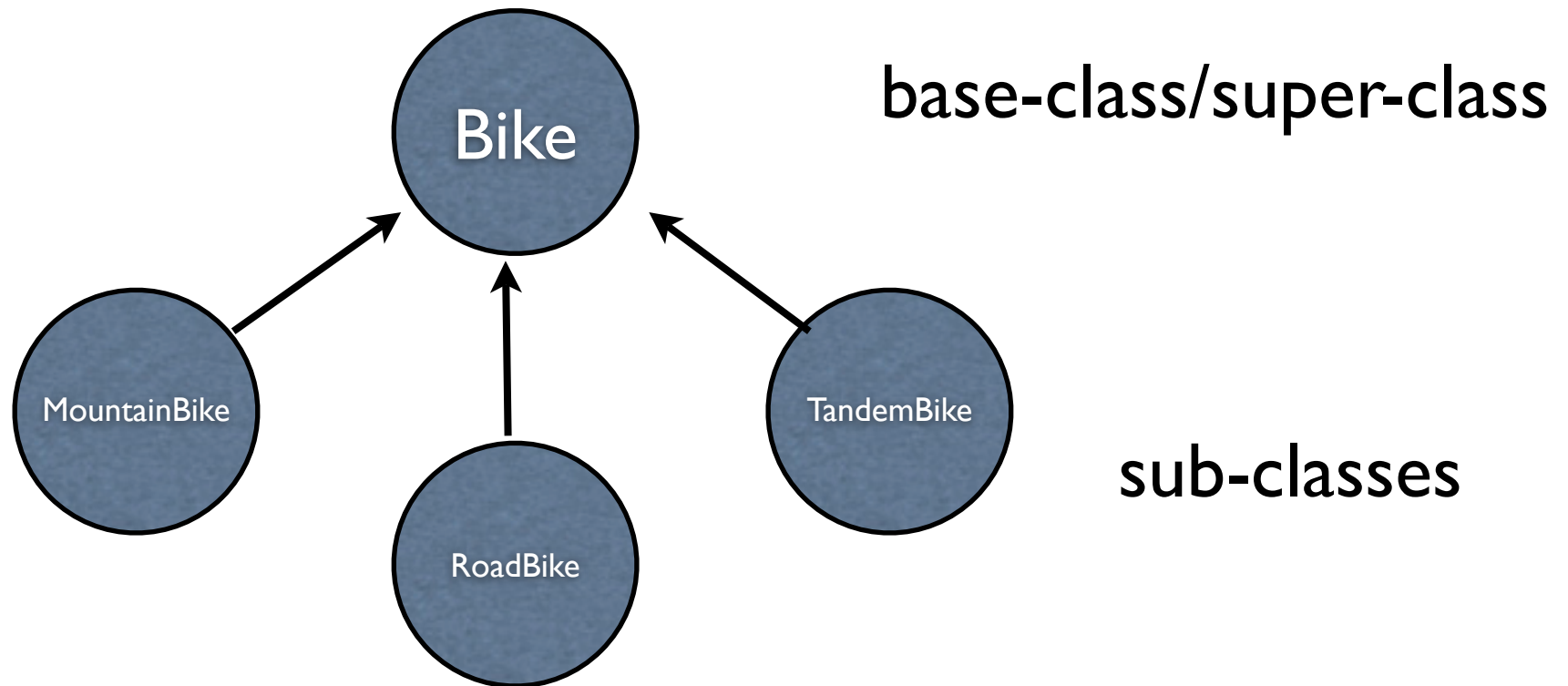
# Design Principles

To achieve the design goals, you follow a couple of **principles**:

- **Abstraction**
  - distill a complicated system down to its most fundamental parts and describe it simply
- **Encapsulation**
  - different components should NOT reveal internal details of their implementation
    - e.g. data of an object is private (not public)
  - one should be able to use a class by reading its interface
    - interface of a class: the set of methods it supports
    - e.g. read Java online docs and use the class; no need to know implementation
- **Modularity**
  - divide the code into separate functional units

# Inheritance

- The capability of a class to use the properties and methods of another class while adding its own functionality.
- A mechanism for sharing/reusing code
  - captures similarities between classes



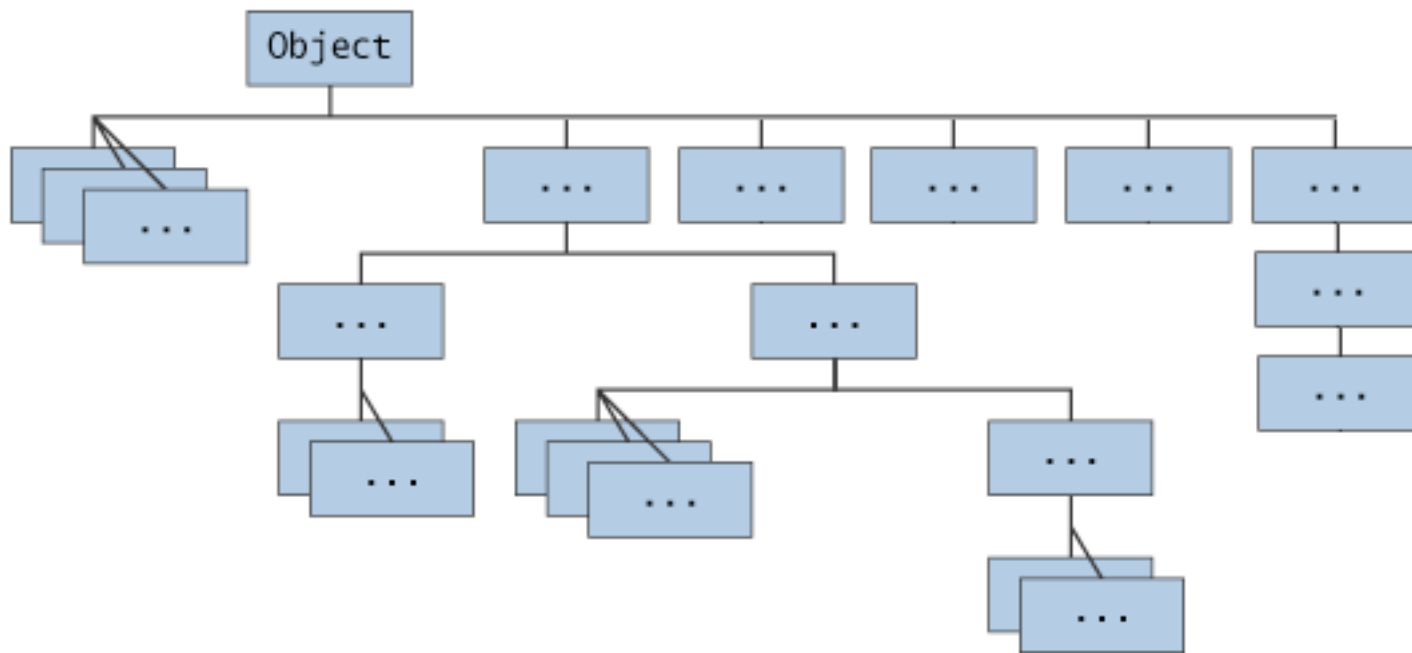
- A sub-class inherits all public and protected members of its parent

# Example

```
public class Bicycle {  
  
    public int gear;  
    public int speed;  
  
    public Bicycle(int startSpeed, int startGear) {...}  
    public void setGear(int newValue) {...}  
    public void applyBrake(int decrement) {...}  
    public void speedUp(int increment) {...}  
}  
  
public class MountainBike extends Bicycle {  
  
    // the MountainBike subclass adds one field  
    public int seatHeight;  
  
    // the MountainBike subclass has one constructor  
    public MountainBike(int startHeight, int startSpeed, int startGear) {  
        super(startSpeed, startGear);  
        seatHeight = startHeight;  
    }  
  
    // the MountainBike subclass adds one method  
    public void setHeight(int newValue) {...}  
  
}
```

# Inheritance in Java

- **Object** is the highest superclass (ie. **root** class) of Java
  - all other classes are subclasses (children or descendants) of Object
- Object class defined defined in the `java.lang` package; includes methods such as:
  - `hashCode()`
  - `toString()`
  - `getClass()`
- when your class does not extend any specific class, it extends Object by default



All Classes in the Java Platform are Descendants of Object

# Inheritance

- Using inheritance
  - When you want to create a new class and there is already a class that includes some of the code that you want, you can derive your new class from the existing class.
  - In doing this, you can reuse the fields and methods of the existing class without having to write (and debug!) them yourself.
- Definitions
  - A class that is derived from another class is called a *subclass* (also a *derived class*, *extended class*, or *child class*).
  - The class from which the subclass is derived is called a *superclass* (also a *base class* or a *parent class*).
  - Excepting `Object`, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of `Object`.
  - Classes can be derived from classes that are derived from classes that are derived from classes, and so on, and ultimately derived from the topmost class, `Object`. Such a class is said to be *descended* from all the classes in the inheritance chain stretching back to `Object`.

## What You Can Do in a Subclass

- The inherited fields and method can be used directly
- You can declare new fields in the subclass that are not in the superclass
- You can declare new methods in the subclass that are not in the superclass
- You can override a method
  - write a new method in the subclass that has the same signature as the one in the superclass
  - you can invoke superclass method using keyword `super`
- You can write a subclass constructor
  - invokes the constructor of the superclass by using `super`

## Calling super in a constructor

```
public MountainBike(int startHeight, int startSpeed, int startGear) {  
  
    //call superclass constructor to create a Bike  
    super(startCadence, startSpeed, startGear);  
  
    seatHeight = startHeight;  
}
```

## Calling super in an overridden method

```
public class Superclass {  
  
    public void printMethod() {  
        System.out.println("Printed in Superclass.");  
    }  
}  
public class Subclass extends Superclass {  
  
    public void printMethod() { //overrides printMethod in Superclass  
        super.printMethod();  
        System.out.println("Printed in Subclass");  
    }  
  
    public static void main(String[] args) {  
        Subclass s = new Subclass();  
        s.printMethod();  
    }  
}
```

# this

- within a method **this** refers to the current object
- Used when a field is shadowed by a method or constructor parameter.

```
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int a, int b) {
        x = a;
        y = b;
    }
}
```

- but it could have been written like this:

```
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

# this

- Using **this** with a Constructor

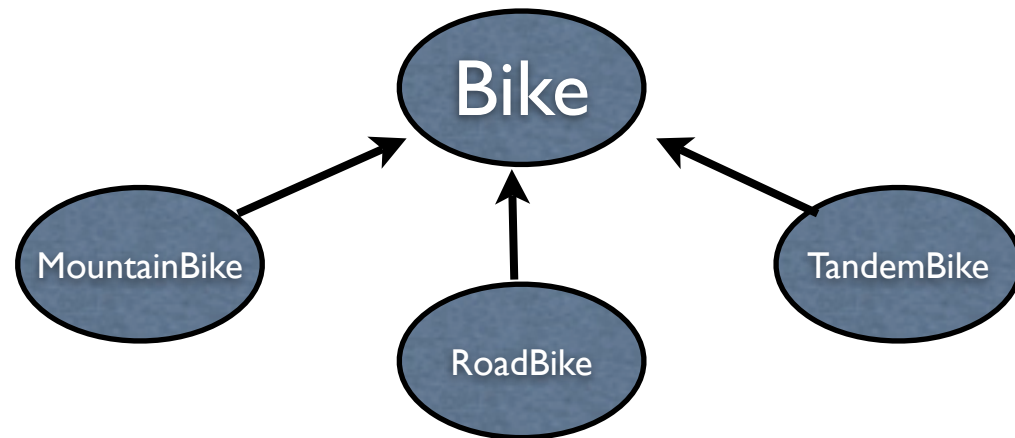
- From within a constructor, you can use `this` keyword to call another constructor in the same class (doing so is called an *explicit constructor invocation*)

```
public class Rectangle {
    private int x, y;
    private int width, height;

    public Rectangle() {
        this(0, 0, 0, 0);
    }
    public Rectangle(int width, int height) {
        this(0, 0, width, height);
    }
    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
    ...
}
```

# Casting objects

- a MountainBike is a Bike
- a MountainBike is also an Object
- a Bike is not (necessarily) a MountainBike



- In Java: A variable of type T can be of type T or of any subclass of T
- Example


```
Object bike;  
//bike is allowed to be any subclass of Object  
bike = new MountainBike();
```
- We'll use this by defining data structures that work generically with Objects; we'll create the data structure on any type of objects.
- this is called **casting**: changing the type of an object

- **Implicit casting in an inheritance hierarchy: a subclass can be used in place of a superclass**

# Casting examples


```
Bike b;  
MountainBike mb;  
mb = new MountainBike(..);  
  
//implicit casting of a MountainBike to a Bike  
b = mb;
```

```
class Person {  
    //any person has a bike  
    Bike b;  
    void Person(Bike b) {  
        this.b = b;  
    }  
}
```



a person that owns a bike

```
...  
MountainBike mb = new MountainBike();  
Person p = new Person(mb);
```



a mountainbike is a bike

# Interfaces

- An interface is a collection of method signatures (with no bodies)
- similar to a class

```
public interface OperateCar {  
  
    // method signatures  
    int turn(Direction direction, double radius,);  
    int changeLanes(Direction direction, double startSpeed, double endSpeed);  
    int signalTurn(Direction direction, boolean signalOn);  
    .....  
}
```

- When a class implements an interface it must implement all methods in that interface

```
public class OperateBMW760i implements OperateCar {  
  
    int signalTurn(Direction direction, boolean signalOn) {  
        //code to turn BMW's LEFT turn indicator lights on  
        //code to turn BMW's LEFT turn indicator lights off  
        //code to turn BMW's RIGHT turn indicator lights on  
        //code to turn BMW's RIGHT turn indicator lights off  
    }  
  
    // other members, as needed  
  
}
```

# Interfaces

- Interfaces are used to describe the functionality of a software in an abstract way (since methods have no bodies)
- Advantage:
  - the implementation can change while interface remains the same
  - multiple implementations
- E.g., a digital image processing library writes its classes to implement an interface, and publishes its interface (API-application programming interface)
  - the implementation of the methods is usually not disclosed
  - moreover, it can change
  - a graphics package may decide to use this library
    - only needs to know the API
- Interfaces in Java
  - a class can inherit from a SINGLE class
  - a class can implement many interfaces