

---

# CSE 373

## Lecture 00

Review of Object-Oriented Programming and Java

slides created by Marty Stepp  
also based on course materials by Stuart Reges

<http://www.cs.washington.edu/373/>

# Summary

---

- These slides contain material about objects, classes, and object-oriented programming in Java.
- We will lightly cover some of the topics contained in these slides in lecture, but, in general, you are expected to remember from CSE 142 and 143.

---

# **Primitives vs. objects; value and reference semantics**

# A swap method?

---

- Does the following swap method work? Why or why not?

```
public static void main(String[] args) {  
    int a = 7;  
    int b = 35;  
  
    // swap a with b?  
    swap(a, b);  
  
    System.out.println(a + " " + b);  
}
```

```
public static void swap(int a, int b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

# Value semantics

---

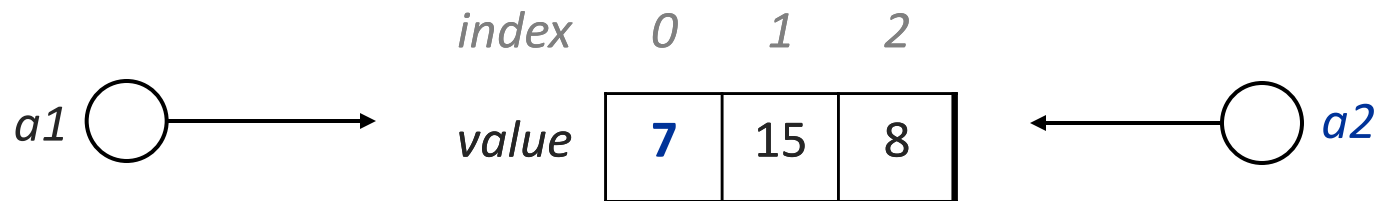
- **value semantics:** Behavior where values are copied when assigned, passed as parameters, or returned.
  - All primitive types in Java use value semantics.
  - When one variable is assigned to another, its value is copied.
  - Modifying the value of one variable does not affect others.

```
int x = 5;  
int y = x;      // x = 5, y = 5  
y = 17;          // x = 5, y = 17  
x = 8;           // x = 8, y = 17
```

# Reference semantics (objects)

- **reference semantics:** Behavior where variables actually store the address of an object in memory.
  - When one variable is assigned to another, the object is *not* copied; both variables refer to the *same object*.
  - Modifying the value of one variable *will* affect others.

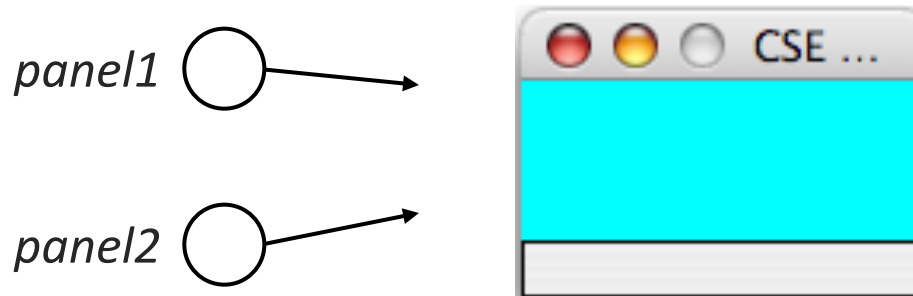
```
int[] a1 = {4, 15, 8};  
int[] a2 = a1;           // refer to same array as a1  
a2[0] = 7;  
System.out.println(Arrays.toString(a1)); // [7, 15, 8]
```



# References and objects

- Arrays and objects use reference semantics. Why?
  - *efficiency*. Copying large objects slows down a program.
  - *sharing*. It's useful to share an object's data among methods.

```
DrawingPanel panel1 = new DrawingPanel(80, 50);  
DrawingPanel panel2 = panel1;    // same window  
panel2.setBackground(Color.CYAN);
```

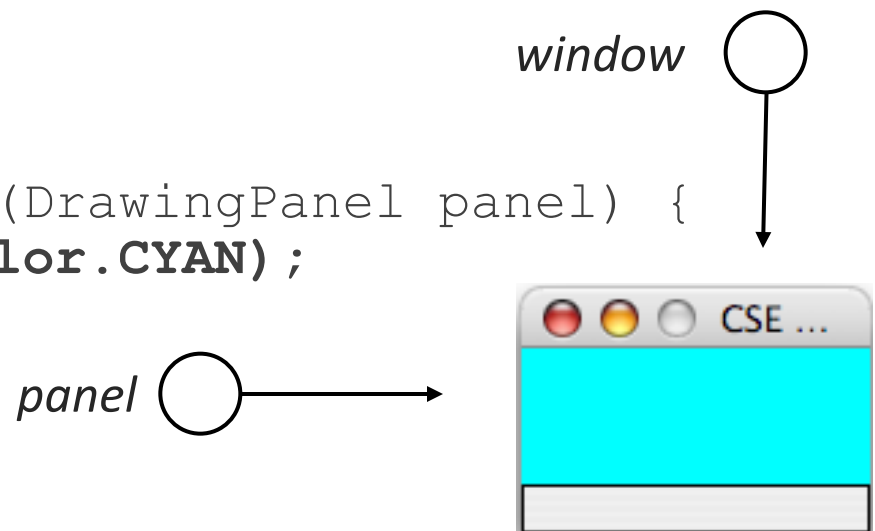


# Objects as parameters

- When an object is passed as a parameter, the object is *not* copied. The parameter refers to the same object.
  - If the parameter is modified, it *will* affect the original object.

```
public static void main(String[] args) {  
    DrawingPanel window = new DrawingPanel(80, 50);  
    window.setBackground(Color.YELLOW);  
    example(window);  
}
```

```
public static void example(DrawingPanel panel) {  
    panel.setBackground(Color.CYAN);  
    ...  
}
```

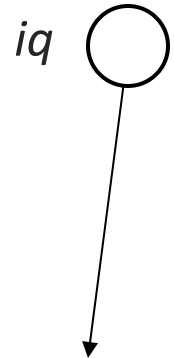




# Arrays pass by reference

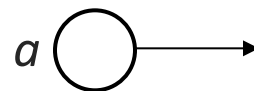
- Arrays are also passed as parameters by reference.
  - Changes made in the method are also seen by the caller.

```
public static void main(String[] args) {  
    int[] iq = {126, 167, 95};  
    increase(iq);  
    System.out.println(Arrays.toString(iq));  
}  
  
public static void increase(int[] a) {  
    for (int i = 0; i < a.length; i++) {  
        a[i] = a[i] * 2;  
    }  
}
```



- Output:

[252, 334, 190]



index	0	1	2
value	252	334	190

---

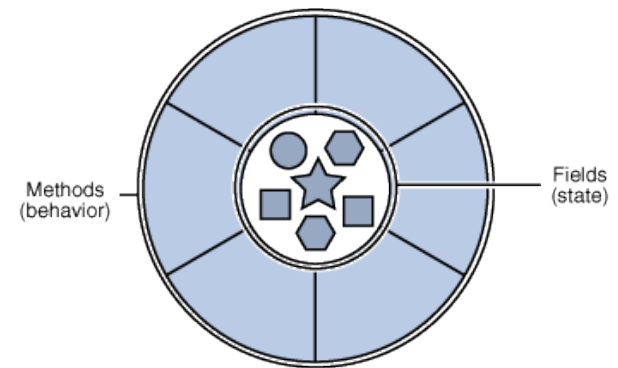
# Classes and Objects

# Objects

- **object:** An entity that encapsulates data and behavior.

- *data:* variables inside the object
- *behavior:* methods inside the object

- You interact with the methods;  
the data is hidden in the object.



- Constructing (creating) an object:

**Type** **objectName** = new **Type** (**parameters**) ;

- Calling an object's method:

**objectName** . **methodName** (**parameters**) ;

# Classes

---

- **class:** A program entity that represents either:
  1. A program / module, or
  2. A template for a new type of objects.
- **object-oriented programming (OOP):** Programs that perform their behavior as interactions between objects.
  - **abstraction:** Separation between concepts and details.  
Objects and classes provide abstraction in programming.

# Blueprint analogy

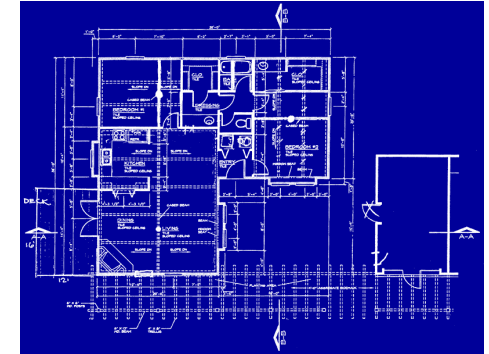
## iPod blueprint

### state:

current song  
volume  
battery life

### behavior:

power on/off  
change station/song  
change volume  
choose random song



*creates*

### iPod #1

#### state:

song = "1,000,000 Miles"  
volume = 17  
battery life = 2.5 hrs

#### behavior:

power on/off  
change station/song  
change volume  
choose random song



### iPod #2

#### state:

song = "Letting You"  
volume = 9  
battery life = 3.41 hrs

#### behavior:

power on/off  
change station/song  
change volume  
choose random song



### iPod #3

#### state:

song = "Discipline"  
volume = 24  
battery life = 1.8 hrs

#### behavior:

power on/off  
change station/song  
change volume  
choose random song



# Point objects

```
import java.awt.*;  
...  
Point p1 = new Point(5, -2);  
Point p2 = new Point();           // origin (0, 0)
```

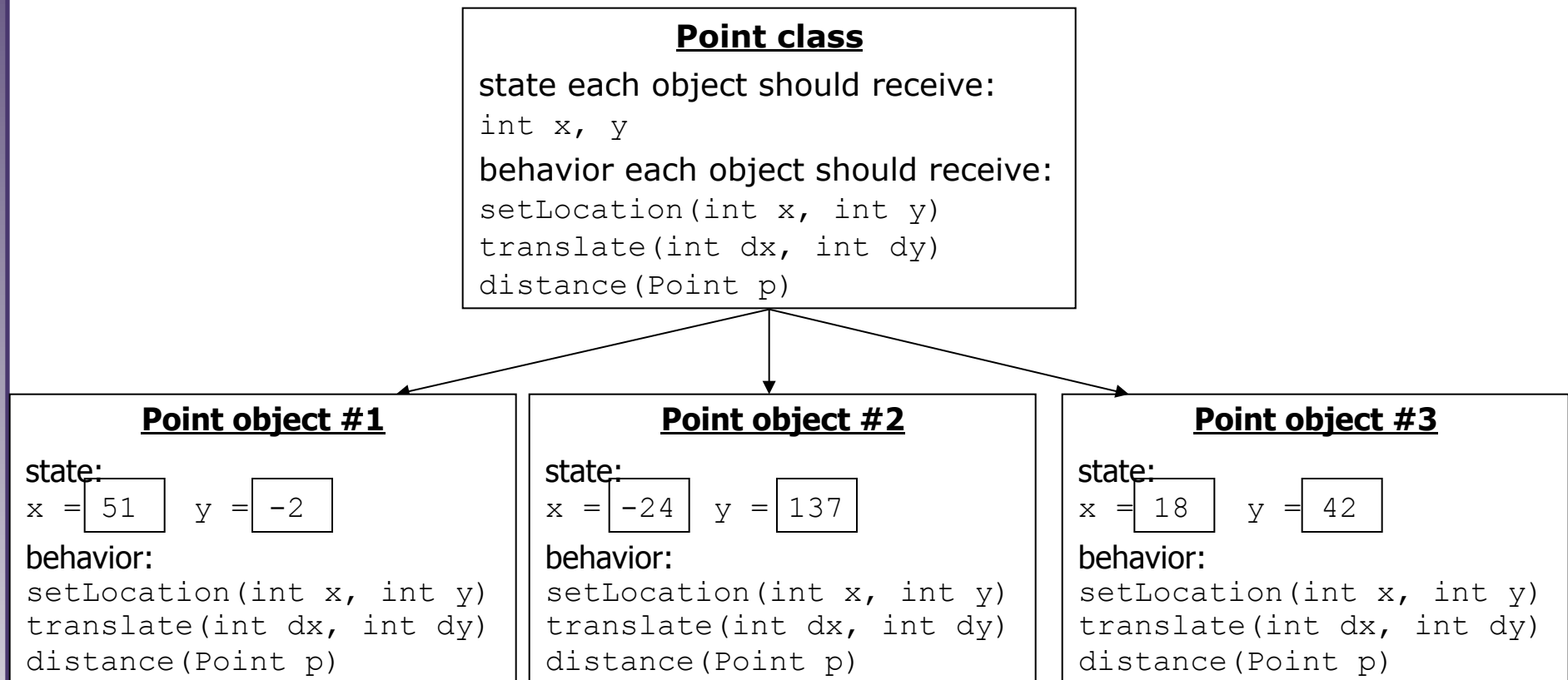
- Data:

Name	Description
x	the point's x-coordinate
y	the point's y-coordinate

- Methods:

Name	Description
setLocation( <b>x</b> , <b>y</b> )	sets the point's x and y to the given values
translate( <b>dx</b> , <b>dy</b> )	adjusts the point's x and y by the given amounts
distance( <b>p</b> )	how far away the point is from point <i>p</i>

# Point class as blueprint



- The class (blueprint) describes how to create objects.
- Each object contains its own data and methods.
  - The methods operate on that object's data.

# Clients of objects

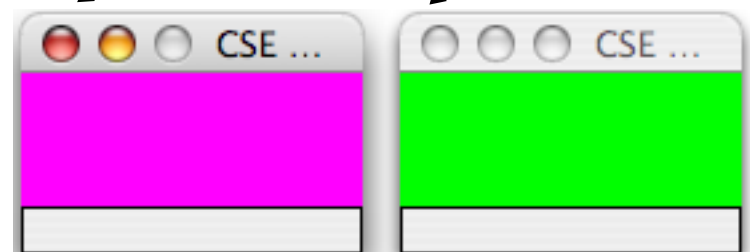
- **client program:** A program that uses objects.
  - Example: Bomb is a client of `DrawingPanel` and `Graphics`.

## Bomb.java (client program)

```
public class Bomb {  
    main(String[] args) {  
        new DrawingPanel(...)  
        new DrawingPanel(...)  
        ...  
    }  
}
```

## DrawingPanel.java (class)

```
public class DrawingPanel {  
    ...  
}
```





# Fields

---

- **field**: A variable inside an object that is part of its state.
  - Each object has *its own copy* of each field.
- Declaration syntax:

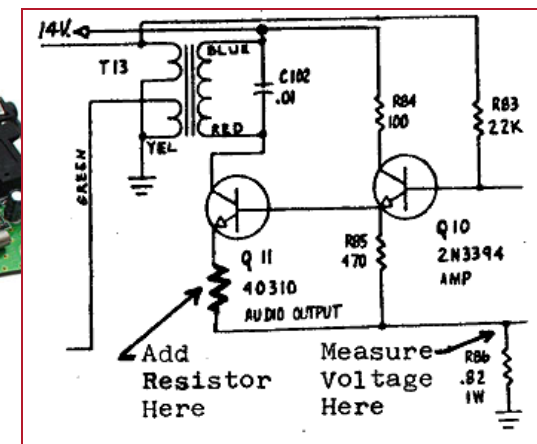
```
private type name;
```

- Example:

```
public class Point {  
    private int x;  
    private int y;  
  
    ...  
}
```

# Encapsulation

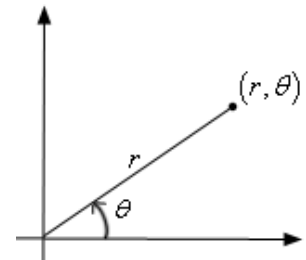
- **encapsulation:** Hiding implementation details from clients.
  - Encapsulation enforces *abstraction*.
    - separates external view (behavior) from internal view (state)
    - protects the integrity of an object's data



# Benefits of encapsulation

---

- Abstraction between object and clients
- Protects object from unwanted access
  - Example: Can't fraudulently increase an `Account`'s balance.
- Can change the class implementation later
  - Example: `Point` could be rewritten in polar coordinates  $(r, \vartheta)$  with the same methods.
- Can constrain objects' state (**invariants**)
  - Example: Only allow `Accounts` with non-negative balance.
  - Example: Only allow `Dates` with a month from 1-12.



# Instance methods

---

- **instance method** (or **object method**): Exists inside each object of a class and gives behavior to each object.

```
public type name (parameters) {  
    statements;  
}
```

- same syntax as static methods, but without `static` keyword

Example:

```
public void tranlate(int dx, int dy) {  
    x += dx;  
    y += dy;  
}
```

# The implicit parameter

---

- **implicit parameter:**

The object on which an instance method is being called.

- If we have a `Point` object `p1` and call `p1.translate(5, 3)`; the object referred to by `p1` is the implicit parameter.
- If we have a `Point` object `p2` and call `p2.translate(4, 1)`; the object referred to by `p2` is the implicit parameter.
- The instance method can refer to that object's fields.
  - We say that it executes in the *context* of a particular object.
  - `translate` can refer to the `x` and `y` of the object it was called on.

# Categories of methods

---

- **accessor:** A method that lets clients examine object state.
  - Examples: `distance`, `distanceFromOrigin`
  - often has a `non-void` return type
- **mutator:** A method that modifies an object's state.
  - Examples: `setLocation`, `translate`
- **helper:** Assists some other method in performing its task.
  - often declared as `private` so outside clients cannot call it

# The toString method

---

*tells Java how to convert an object into a String for printing*

```
public String toString() {  
    code that returns a String representing this object;  
}
```

- Method name, return, and parameters must match *exactly*.
- Example:

```
// Returns a String representing this Point.  
public String toString() {  
    return "(" + x + ", " + y + ")";  
}
```

# Constructors

---

- **constructor**: Initializes the state of new objects.

```
public type (parameters) {  
    statements;  
}
```

- runs when the client uses the `new` keyword
- no return type is specified; implicitly "returns" the new object

```
public class Point {  
    private int x;  
    private int y;  
  
    public Point(int initialX, int initialY) {  
        x = initialX;  
        y = initialY;  
    }  
}
```



# Multiple constructors

---

- A class can have multiple constructors.
  - Each one must accept a unique set of parameters.
- *Example:* A `Point` constructor with no parameters that initializes the point to (0, 0).

*// Constructs a new point at (0, 0).*

```
public Point() {  
    x = 0;  
    y = 0;  
}
```

# The keyword `this`

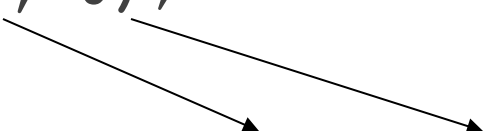
---

- **`this`** : Refers to the implicit parameter inside your class.  
*(a variable that stores the object on which a method is called)*
  - Refer to a field: `this.field`
  - Call a method: `this.method(parameters) ;`
  - One constructor `this(parameters) ;`  
can call another:

# Calling another constructor

---

```
public class Point {  
    private int x;  
    private int y;  
  
    public Point() {  
        this(0, 0);  
    }  
  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    ...  
}
```



- Avoids redundancy between constructors
- Only a constructor (not a method) can call another constructor

---

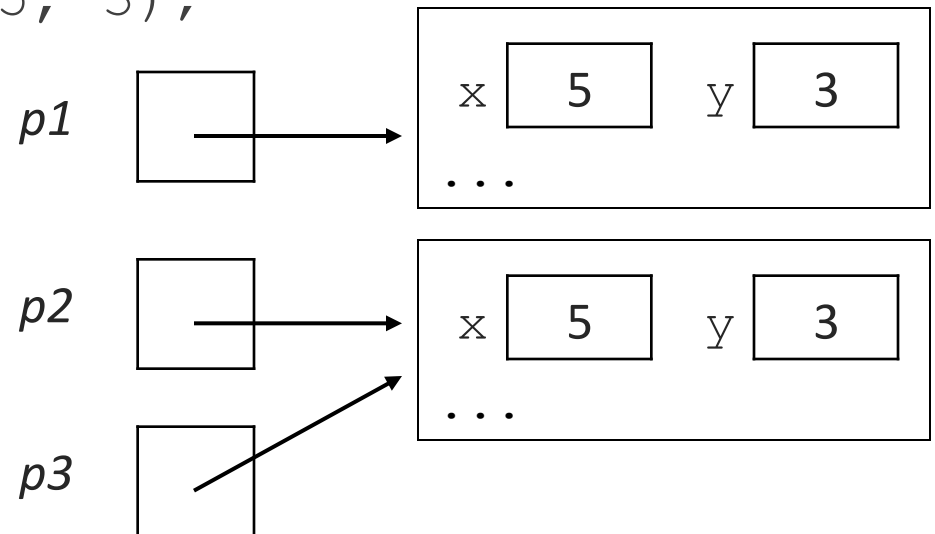
# Comparing objects for equality and ordering

# Comparing objects

- The `==` operator does not work well with objects.  
    `==` compares references to objects, not their state.  
    It only produces `true` when you compare an object to itself.

```
Point p1 = new Point(5, 3);  
Point p2 = new Point(5, 3);  
Point p3 = p2;
```

```
// p1 == p2 is false;  
// p1 == p3 is false;  
// p2 == p3 is true
```



# The equals method

---

- The equals method compares the state of objects.

```
if (str1.equals(str2)) {  
    System.out.println("the strings are equal");  
}
```

- But if you write a class, its equals method behaves like ==

```
if (p1.equals(p2)) {    // false :- (  
    System.out.println("equal");  
}
```

- This is the default behavior we receive from class Object.
- Java doesn't understand how to compare new classes by default.

# The compareTo method (10.2)

- The standard way for a Java class to define a comparison function for its objects is to define a `compareTo` method.
  - Example: in the `String` class, there is a method:  

```
public int compareTo(String other)
```
- A call of **A.compareTo(B)** will return:
  - a value  $< 0$       if **A** comes "before" **B** in the ordering,
  - a value  $> 0$       if **A** comes "after" **B** in the ordering,
  - or            0      if **A** and **B** are considered "equal" in the ordering.

# Using compareTo

- compareTo can be used as a test in an if statement.

```
String a = "alice";  
String b = "bob";  
if (a.compareTo(b) < 0) { // true  
    ...  
}
```

Primitives	Objects
if (a < b) { ...	if (a.compareTo(b) < 0) { ...
if (a <= b) { ...	if (a.compareTo(b) <= 0) { ...
if (a == b) { ...	if (a.compareTo(b) == 0) { ...
if (a != b) { ...	if (a.compareTo(b) != 0) { ...
if (a >= b) { ...	if (a.compareTo(b) >= 0) { ...
if (a > b) { ...	if (a.compareTo(b) > 0) { ...



# compareTo and collections

- You can use an array or list of strings with Java's included binary search method because it calls `compareTo` internally.

```
String[] a = {"al", "bob", "cari", "dan", "mike"};  
int index = Arrays.binarySearch(a, "dan"); // 3
```

- Java's `TreeSet/Map` use `compareTo` internally for ordering.

```
Set<String> set = new TreeSet<String>();  
for (String s : a) {  
    set.add(s);  
}  
System.out.println(s);  
// [al, bob, cari, dan, mike]
```

# Comparable (10.2)

---

```
public interface Comparable<E> {  
    public int compareTo(E other);  
}
```

- A class can implement the `Comparable` interface to define a natural ordering function for its objects.
- A call to your `compareTo` method should return:
  - a value < 0      if this object comes "before" the other object,
  - a value > 0      if this object comes "after" the other object,
  - or                0      if this object is considered "equal" to the other.

# Comparable template

---

```
public class name implements Comparable<name> {  
  
    ...  
  
    public int compareTo(name other) {  
        ...  
    }  
}
```

# Comparable example

```
public class Point implements Comparable<Point> {
    private int x;
    private int y;
    ...

    // sort by x and break ties by y
    public int compareTo(Point other) {
        if (x < other.x) {
            return -1;
        } else if (x > other.x) {
            return 1;
        } else if (y < other.y) {
            return -1;    // same x, smaller y
        } else if (y > other.y) {
            return 1;    // same x, larger y
        } else {
            return 0;    // same x and same y
        }
    }
}
```

# compareTo tricks

---

- *subtraction trick* - Subtracting related numeric values produces the right result for what you want compareTo to return:

*// sort by x and break ties by y*

```
public int compareTo(Point other) {  
    if (x != other.x) {  
        return x - other.x;    // different x  
    } else {  
        return y - other.y;    // same x; compare y  
    }  
}
```

- The idea:

- if  $x > other.x$ , then  $x - other.x > 0$
- if  $x < other.x$ , then  $x - other.x < 0$
- if  $x == other.x$ , then  $x - other.x == 0$

■ NOTE: This trick doesn't work for doubles (but see `Math.signum`)

# compareTo tricks 2

---

- *delegation trick* - If your object's fields are comparable (such as strings), use their `compareTo` results to help you:

```
// sort by employee name, e.g. "Jim" < "Susan"
public int compareTo(Employee other) {
    return name.compareTo(other.getName());
}
```

- *toString trick* - If your object's `toString` representation is related to the ordering, use that to help you:

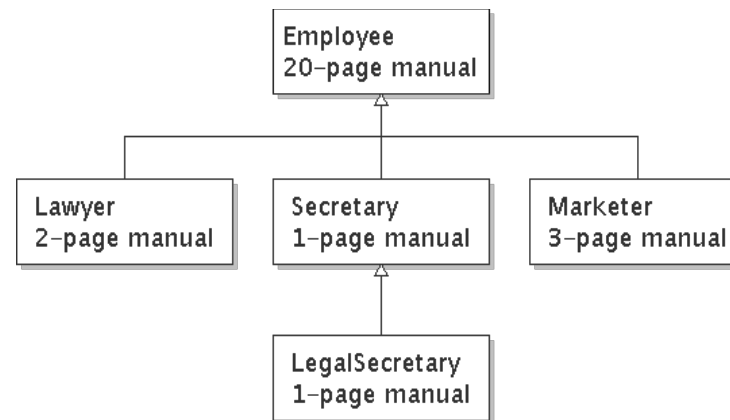
```
// sort by date, e.g. "09/19" > "04/01"
public int compareTo(Date other) {
    return toString().compareTo(other.toString());
}
```

---

# Inheritance

# Inheritance

- **inheritance**: Forming new classes based on existing ones.
  - a way to share/**reuse code** between two or more classes
  - **superclass**: Parent class being extended.
  - **subclass**: Child class that inherits behavior from superclass.
    - gets a copy of every field and method from superclass
  - **is-a relationship**: Each object of the subclass also "is a(n)" object of the superclass and can be treated as one.





# Inheritance syntax

---

```
public class name extends superclass {
```

- Example:

```
public class Lawyer extends Employee {  
    ...  
}
```

- By extending `Employee`, each `Lawyer` object now:
  - receives a copy of each method from `Employee` automatically
  - can be treated as an `Employee` by client code
- `Lawyer` can also replace ("override") behavior from `Employee`.

# Overriding Methods

---

- **override:** To write a new version of a method in a subclass that replaces the superclass's version.
  - No special syntax required to override a superclass method. Just write a new version of it in the subclass.

```
public class Lawyer extends Employee {  
    // overrides getVacationForm in Employee class  
    public String getVacationForm() {  
        return "pink";  
    }  
    ...  
}
```

# The super keyword

---

- A subclass can call its parent's method/constructor:

```
super.method(parameters)           // method
super(parameters);                  // constructor
```

```
public class Lawyer extends Employee {
    public Lawyer(String name) {
        super(name);
    }

    // give Lawyers a $5K raise (better)
    public double getSalary() {
        double baseSalary = super.getSalary();
        return baseSalary + 5000.00;
    }
}
```

# Subclasses and fields

---

```
public class Employee {  
    private double salary;  
    ...  
}  
  
public class Lawyer extends Employee {  
    ...  
    public void giveRaise(double amount) {  
        salary += amount;    // error; salary is private  
    }  
}
```

- Inherited private fields/methods cannot be directly accessed by subclasses. *(The subclass has the field, but it can't touch it.)*
  - How can we allow a subclass to access/modify these fields?

# Protected fields/methods

---

```
protected type name;    // field
```

```
protected type name(type name, ..., type name) {  
    statement(s);        // method  
}
```

- a **protected field** or **method** can be seen/called only by:
  - the class itself, and its subclasses
  - also by other classes in the same "package" (discussed later)
  - useful for allowing selective access to inner class implementation

```
public class Employee {  
    protected double salary;  
    ...  
}
```

# Inheritance and constructors

---

- If we add a constructor to the `Employee` class, our subclasses do not compile. The error:

```
Lawyer.java:2: cannot find symbol
symbol   : constructor Employee()
location: class Employee
public class Lawyer extends Employee {
      ^
```

- The short explanation: Once we write a constructor (that requires parameters) in the superclass, we must now write constructors for our employee subclasses as well.

# Inheritance and constructors

---

- Constructors are not inherited.
  - Subclasses don't inherit the `Employee(int)` constructor.
  - Subclasses receive a default constructor that contains:

```
public Lawyer() {  
    super();           // calls Employee() constructor  
}
```

- But our `Employee(int)` replaces the default `Employee()`.
  - The subclasses' default constructors are now trying to call a non-existent default `Employee` constructor.

# Calling superclass constructor

---

`super (parameters) ;`

- Example:

```
public class Lawyer extends Employee {  
    public Lawyer(int years) {  
        super(years);    // calls Employee c'tor  
    }  
    ...  
}
```

- The `super` call must be the first statement in the constructor.



---

# Polymorphism

# Polymorphism

---

- **polymorphism:** Ability for the same code to be used with different types of objects and behave differently with each.
  - `System.out.println` can print any type of object.
    - Each one displays in its own way on the console.
  - `CritterMain` can interact with any type of critter.
    - Each one moves, fights, etc. in its own way.

# Coding with polymorphism

---

- A variable of type  $T$  can hold an object of any subclass of  $T$ .

```
Employee ed = new Lawyer();
```

- You can call any methods from the `Employee` class on `ed`.
- When a method is called on `ed`, it behaves as a `Lawyer`.

```
System.out.println(ed.getSalary()); // 50000.0  
System.out.println(ed.getVacationForm()); // pink
```

# Polymorphic parameters

- You can pass any subtype of a parameter's type.

```
public static void main(String[] args) {
    Lawyer lisa = new Lawyer();
    Secretary steve = new Secretary();
    printInfo(lisa);
    printInfo(steve);
}

public static void printInfo(Employee e) {
    System.out.println("pay   : " + e.getSalary());
    System.out.println("vdays: " + e.getVacationDays());
    System.out.println("vform: " + e.getVacationForm());
    System.out.println();
}
```

## OUTPUT:

pay   : 50000.0	pay   : 50000.0
vdays: 15	vdays: 10
vform: pink	vform: yellow

# Polymorphism and arrays

- Arrays of superclass types can store any subtype as elements.

```
public static void main(String[] args) {  
    Employee[] e = {new Lawyer(),    new Secretary(),  
                    new Marketer(), new LegalSecretary()};  
  
    for (int i = 0; i < e.length; i++) {  
        System.out.println("pay   : " + e[i].getSalary());  
        System.out.println("vdays: " + i].getVacationDays());  
        System.out.println();  
    }  
}
```

Output:

pay   : 50000.0	pay   : <b>60000.0</b>
vdays: <b>15</b>	vdays: 10
pay   : 50000.0	pay   : <b>55000.0</b>
vdays: 10	vdays: 10

# Casting references

---

- A variable can only call that type's methods, not a subtype's.

```
Employee ed = new Lawyer();  
int hours = ed.getHours();    // ok; in Employee  
ed.sue();                     // compiler error
```

- The compiler's reasoning is, variable `ed` could store any kind of employee, and not all kinds know how to `sue`.

- To use `Lawyer` methods on `ed`, we can type-cast it.

```
Lawyer theRealEd = (Lawyer) ed;  
theRealEd.sue();    // ok  
  
( (Lawyer) ed ).sue();    // shorter version
```

# More about casting

---

- The code crashes if you cast an object too far down the tree.

```
Employee eric = new Secretary();  
(Secretary) eric).takeDictation("hi");           // ok  
((LegalSecretary) eric).fileLegalBriefs(); // error  
// (Secretary doesn't know how to file briefs)
```

- You can cast only up and down the tree, not sideways.

```
Lawyer linda = new Lawyer();  
((Secretary) linda).takeDictation("hi"); // error
```

- Casting doesn't actually change the object's behavior.  
It just gets the code to compile/run.

```
((Employee) linda).getVacationForm()           // pink
```

---

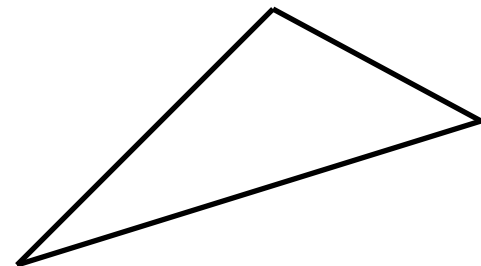
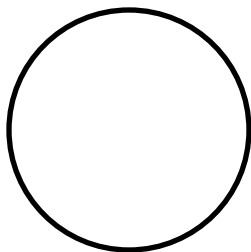
# Interfaces



# Shapes example

---

- Consider the task of writing classes to represent 2D shapes such as `Circle`, `Rectangle`, and `Triangle`.
- Certain operations are common to all shapes:
  - perimeter: distance around the outside of the shape
  - area: amount of 2D space occupied by the shape
  - Every shape has these, but each computes them differently.

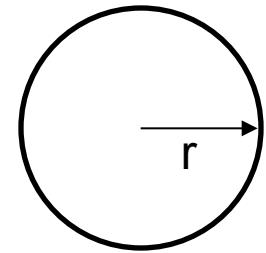


# Shape area and perimeter

- Circle (as defined by radius  $r$ ):

$$\text{area} = \pi r^2$$

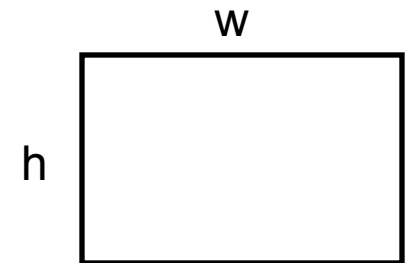
$$\text{perimeter} = 2 \pi r$$



- Rectangle (as defined by width  $w$  and height  $h$ ):

$$\text{area} = w h$$

$$\text{perimeter} = 2w + 2h$$

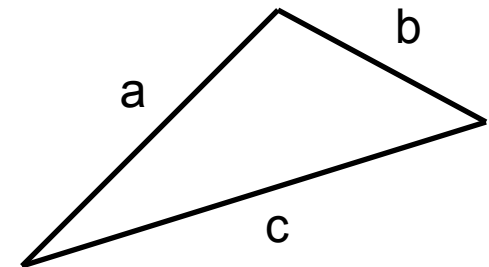


- Triangle (as defined by side lengths  $a$ ,  $b$ , and  $c$ )

$$\text{area} = \sqrt{s(s-a)(s-b)(s-c)}$$

$$\text{where } s = \frac{1}{2}(a+b+c)$$

$$\text{perimeter} = a + b + c$$



# Common behavior

---

- Suppose we have 3 classes `Circle`, `Rectangle`, `Triangle`.
  - Each has the methods `perimeter` and `area`.
- We'd like our client code to be able to treat different kinds of shapes in the same way:
  - Write a method that prints any shape's area and perimeter.
  - Create an array to hold a mixture of the various shape objects.
  - Write a method that could return a rectangle, a circle, a triangle, or any other kind of shape.
  - Make a `DrawingPanel` display many shapes on screen.

# Interfaces

---

- **interface:** A list of methods that a class can promise to implement.
  - Inheritance gives you an is-a relationship *and* code sharing.
    - A `Lawyer` can be treated as an `Employee` and inherits its code.
  - Interfaces give you an is-a relationship *without* code sharing.
    - A `Rectangle` object can be treated as a `Shape` but inherits no code.
  - Analogous to non-programming idea of roles or certifications:
    - "I'm certified as a CPA accountant.  
This assures you I know how to do taxes, audits, and consulting."
    - "I'm 'certified' as a `Shape`, because I implement the `Shape` interface.  
This assures you I know how to compute my area and perimeter."

# Interface syntax

---

```
public interface name {  
    public type name(type name, ..., type name);  
    public type name(type name, ..., type name);  
    ...  
    public type name(type name, ..., type name);  
}
```

Example:

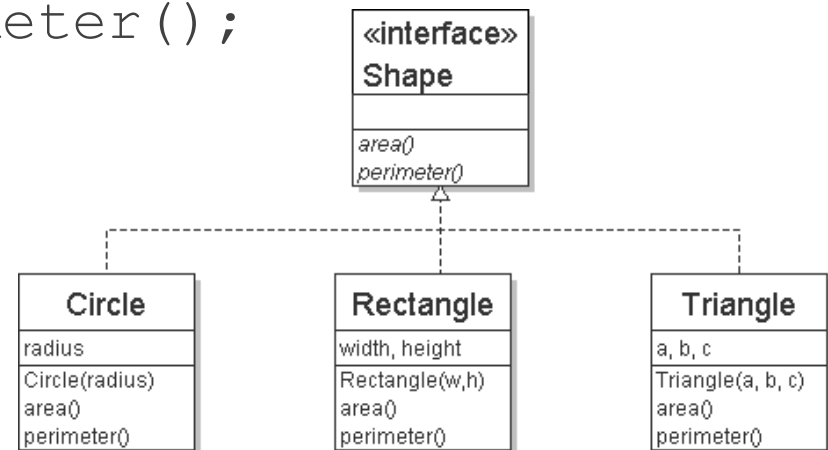
```
public interface Vehicle {  
    public int getSpeed();  
    public void setDirection(int direction);  
}
```

# Shape interface

// Describes features common to all shapes.

```
public interface Shape {  
    public double area();  
    public double perimeter();  
}
```

- Saved as Shape.java



- **abstract method:** A header without an implementation.
  - The actual bodies are not specified, because we want to allow each class to implement the behavior in its own way.

# Implementing an interface

---

```
public class name implements interface {  
    ...  
}
```

- A class can declare that it "implements" an interface.
  - The class promises to contain each method in that interface.  
(Otherwise it will fail to compile.)

- Example:

```
public class Bicycle implements Vehicle {  
    ...  
}
```

# Interface requirements

---

```
public class Banana implements Shape {  
    // haha, no methods! pwned  
}
```

- If we write a class that claims to be a Shape but doesn't implement area and perimeter methods, it will not compile.

```
Banana.java:1: Banana is not abstract and does  
not override abstract method area() in Shape  
public class Banana implements Shape {  
    ^
```



# Interfaces + polymorphism

- Interfaces benefit the *client code* author the most.
  - they allow **polymorphism**  
(the same code can work with different types of objects)

```
public static void printInfo(Shape s) {  
    System.out.println("The shape: " + s);  
    System.out.println("area : " + s.area());  
    System.out.println("perim: " + s.perimeter());  
    System.out.println();  
}  
  
...  
Circle circ = new Circle(12.0);  
Triangle tri = new Triangle(5, 12, 13);  
printInfo(circ);  
printInfo(tri);
```

---

# Abstract Classes

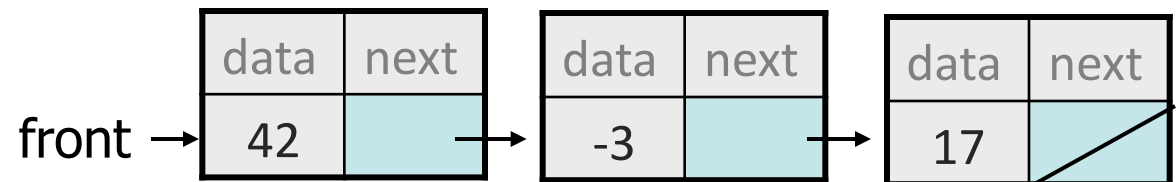
# List classes example

- Suppose we have implemented the following two list classes:

- `ArrayList`

index	0	1	2
value	42	-3	17

- `LinkedList`



- We have a `List` interface to indicate that both implement a List ADT.
- Problem:
  - Some of their methods are implemented the same way (redundancy).

# Common code

---

- Notice that some of the methods are implemented the same way in both the array and linked list classes.
  - `add(value)`
  - `contains`
  - `isEmpty`
- Should we change our interface to a class? Why / why not?
  - How can we capture this common behavior?

# Abstract classes (9.6)

---

- **abstract class:** A hybrid between an interface and a class.
  - defines a superclass type that can contain method declarations (like an interface) and/or method bodies (like a class)
  - like interfaces, abstract classes that cannot be instantiated (cannot use `new` to create any objects of their type)
- What goes in an abstract class?
  - implementation of common state and behavior that will be inherited by subclasses (parent class role)
  - declare generic behaviors that subclasses implement (interface role)

# Abstract class syntax

---

// declaring an abstract class

```
public abstract class name {  
    ...
```

// declaring an abstract method

// (any subclass must implement it)

```
public abstract type name(parameters) ;
```

```
}
```

- A class can be `abstract` even if it has no abstract methods
- You can create variables (but not objects) of the abstract type

# Abstract and interfaces

---

- Normal classes that claim to implement an interface must implement all methods of that interface:

```
public class Empty implements List {} // error
```

- Abstract classes can claim to implement an interface without writing its methods; subclasses must implement the methods.

```
public abstract class Empty implements List {} // ok
```

```
public class Child extends Empty {} // error
```

# An abstract list class

---

*// Superclass with common code for a list of integers.*

```
public abstract class AbstractList implements List {  
    public void add(int value) {  
        add(size(), value);  
    }  
  
    public boolean contains(int value) {  
        return indexOf(value) >= 0;  
    }  
  
    public boolean isEmpty() {  
        return size() == 0;  
    }  
}
```

```
public class ArrayList extends AbstractList { ...
```

```
public class LinkedList extends AbstractList { ...
```



# Abstract class vs. interface

---

- Why do both interfaces and abstract classes exist in Java?
  - An abstract class can do everything an interface can do and more.
  - So why would someone ever use an interface?
- Answer: Java has single inheritance.
  - can extend only one superclass
  - can implement many interfaces
  - Having interfaces allows a class to be part of a hierarchy (polymorphism) without using up its inheritance relationship.

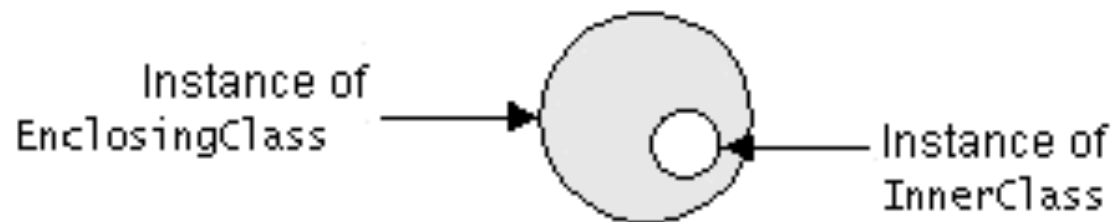
---

# Inner Classes

# Inner classes

---

- **inner class:** A class defined inside of another class.
  - can be created as `static` or non-static
  - we will focus on standard non-static ("nested") inner classes
- **usefulness:**
  - inner classes are hidden from other classes (encapsulated)
  - inner objects can access/modify the fields of the outer object



# Inner class syntax

---

```
// outer (enclosing) class
public class name {
    ...

    // inner (nested) class
    private class name {
        ...
    }
}
```

- Only this file can see the inner class or make objects of it.
- Each inner object is associated with the outer object that created it, so it can access/modify that outer object's methods/fields.
  - If necessary, can refer to outer object as **OuterClassName.this**

# Example: Array list iterator

```
public class ArrayList extends AbstractList {
    ...
    // not perfect; doesn't forbid multiple removes in a row
    private class ArrayIterator implements Iterator<Integer> {
        private int index;    // current position in list

        public ArrayIterator() {
            index = 0;
        }

        public boolean hasNext() {
            return index < size();
        }

        public E next() {
            index++;
            return get(index - 1);
        }

        public void remove() {
            ArrayList.this.remove(index - 1);
            index--;
        }
    }
}
```

---

# Collections

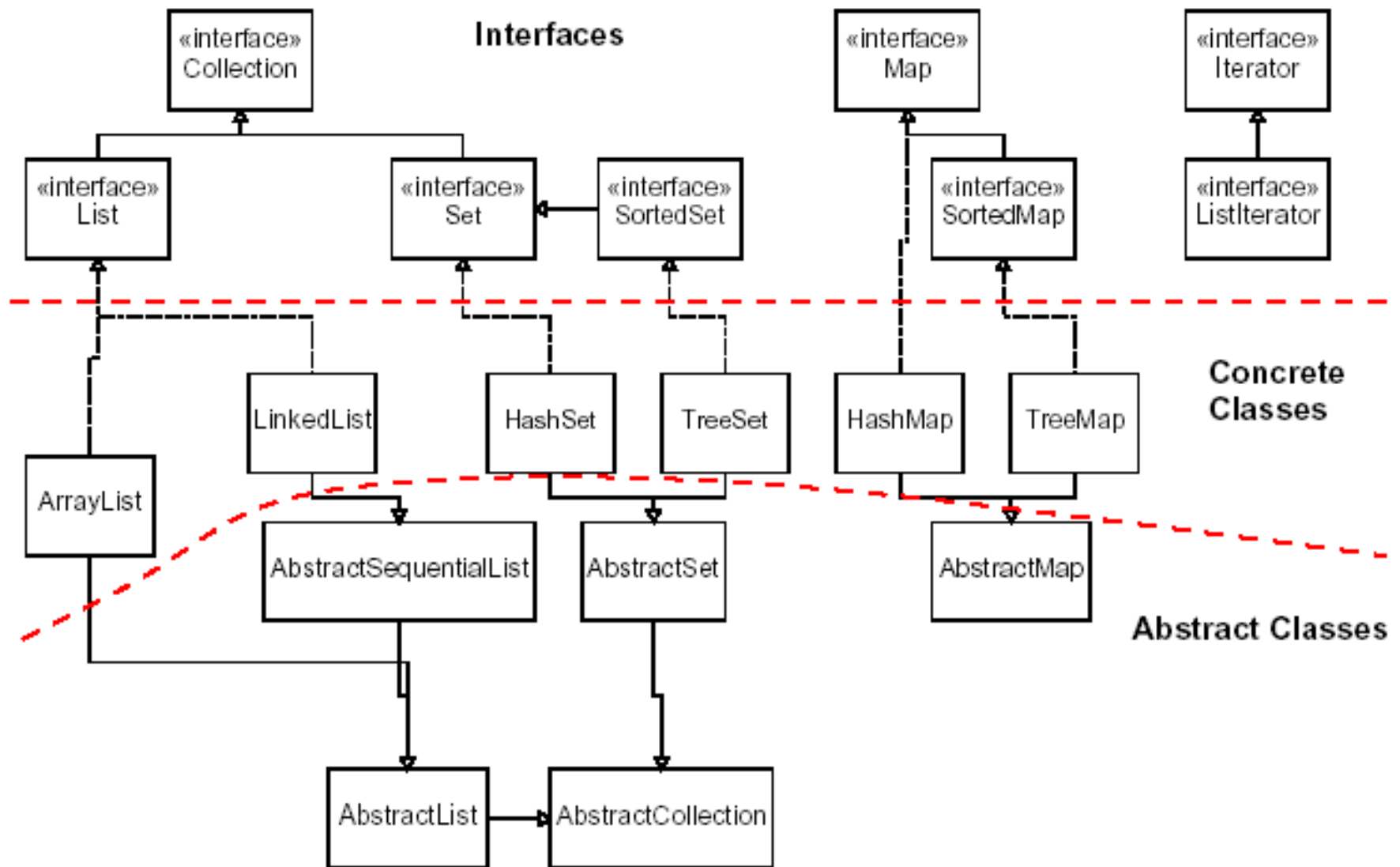
# Collections

---

- **collection**: an object that stores data; a.k.a. "data structure"
  - the objects stored are called **elements**
  - some collections maintain an ordering; some allow duplicates
  - typical operations: *add*, *remove*, *clear*, *contains* (search), *size*
  - examples found in the Java class libraries:
    - `ArrayList`, `LinkedList`, `HashMap`, `TreeSet`, `PriorityQueue`
  - all collections are in the `java.util` package

```
import java.util.*;
```

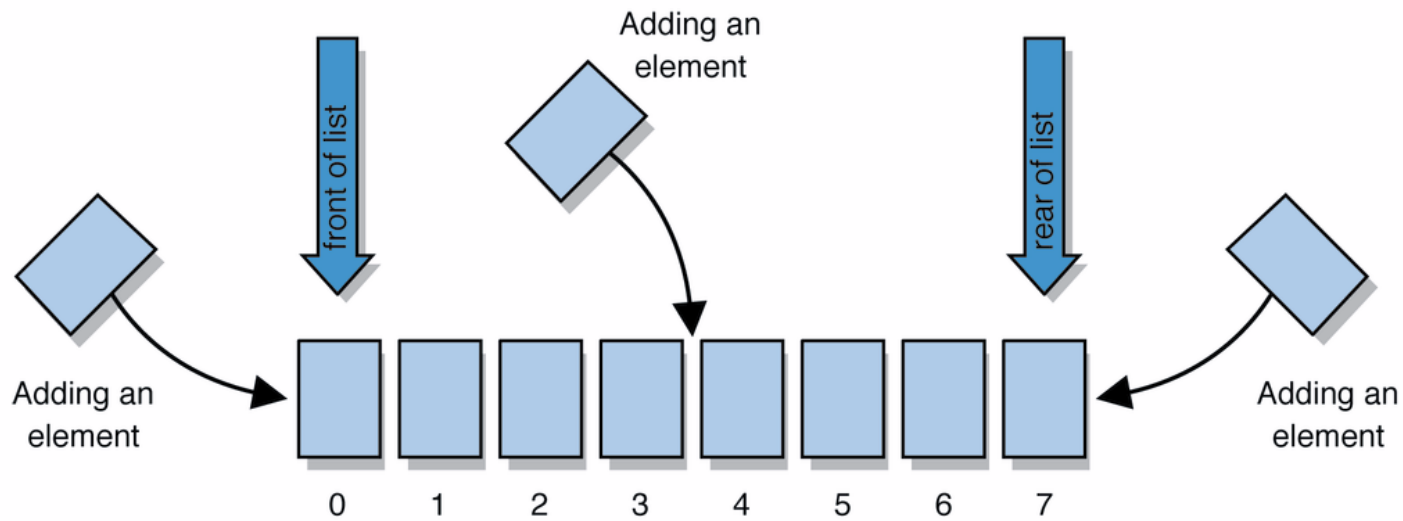
# Java collection framework





# Lists

- **list**: a collection storing an ordered sequence of elements
  - each element is accessible by a 0-based **index**
  - a list has a **size** (number of elements that have been added)
  - elements can be added to the front, back, or elsewhere
  - in Java, a list can be represented as an **ArrayList** object



# Idea of a list

---

- Rather than creating an array of boxes, create an object that represents a "list" of items. (initially an empty list.)

`[]`

- You can add items to the list.
  - The default behavior is to add to the end of the list.

`[hello, ABC, goodbye, okay]`

- The list object keeps track of the element values that have been added to it, their order, indexes, and its total size.
  - Think of an "array list" as an automatically resizing array object.
  - Internally, the list is implemented using an array and a size field.

# ArrayList methods (10.1)

<code>add (value)</code>	appends value at end of list
<code>add (index, value)</code>	inserts given value just before the given index, shifting subsequent values to the right
<code>clear ()</code>	removes all elements of the list
<code>indexOf (value)</code>	returns first index where given value is found in list (-1 if not found)
<code>get (index)</code>	returns the value at given index
<code>remove (index)</code>	removes/returns value at given index, shifting subsequent values to the left
<code>set (index, value)</code>	replaces value at given index with given value
<code>size ()</code>	returns the number of elements in list
<code>toString ()</code>	returns a string representation of the list such as "[3, 42, -7, 15]"

# ArrayList methods 2

addAll ( <b>list</b> )	adds all elements from the given list to this list
addAll ( <b>index</b> , <b>list</b> )	(at the end of the list, or inserts them at the given index)
contains ( <b>value</b> )	returns true if given value is found somewhere in this list
containsAll ( <b>list</b> )	returns true if this list contains every element from given list
equals ( <b>list</b> )	returns true if given other list contains the same elements
iterator() listIterator()	returns an object used to examine the contents of the list
lastIndexOf ( <b>value</b> )	returns last index value is found in list (-1 if not found)
remove ( <b>value</b> )	finds and removes the given value from this list
removeAll ( <b>list</b> )	removes any elements found in the given list from this list
retainAll ( <b>list</b> )	removes any elements <i>not</i> found in given list from this list
subList ( <b>from</b> , <b>to</b> )	returns the sub-portion of the list between indexes <b>from</b> (inclusive) and <b>to</b> (exclusive)
toArray ()	returns the elements in this list as an array

# Type Parameters (Generics)

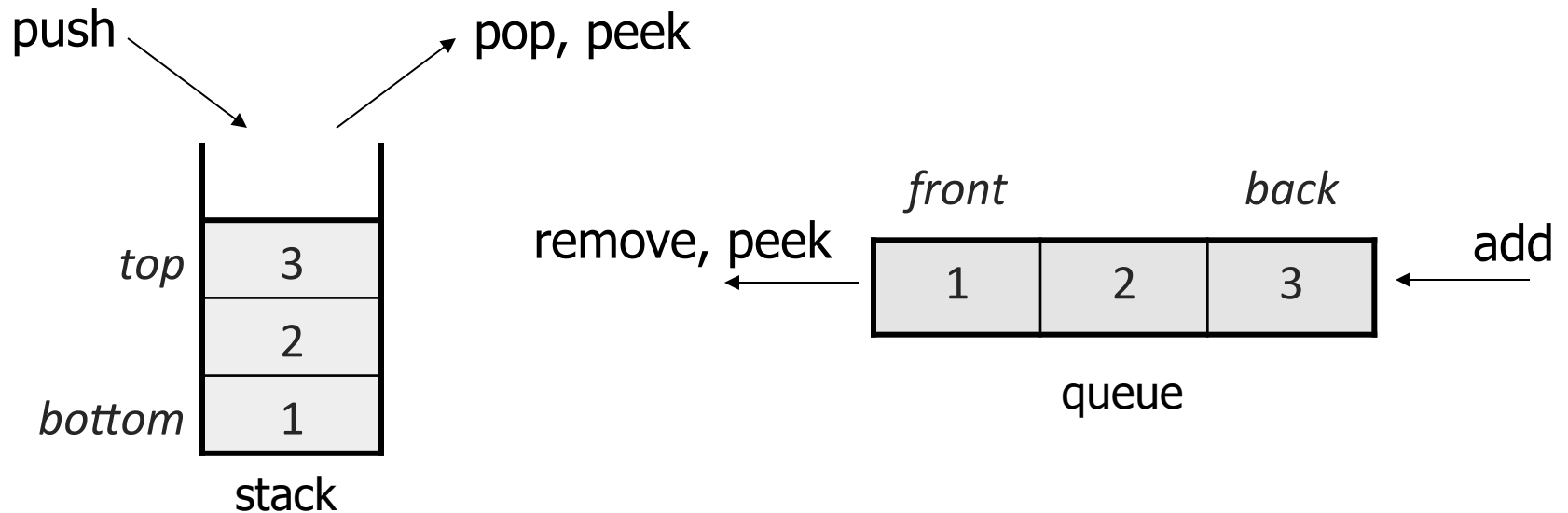
```
List<Type> name = new ArrayList<Type>();
```

- When constructing an `ArrayList`, you must specify the type of elements it will contain between `<` and `>`.
  - This is called a *type parameter* or a *generic* class.
  - Allows the same `ArrayList` class to store lists of different types.

```
List<String> names = new ArrayList<String>();  
names.add("Marty Stepp");  
names.add("Stuart Reges");
```

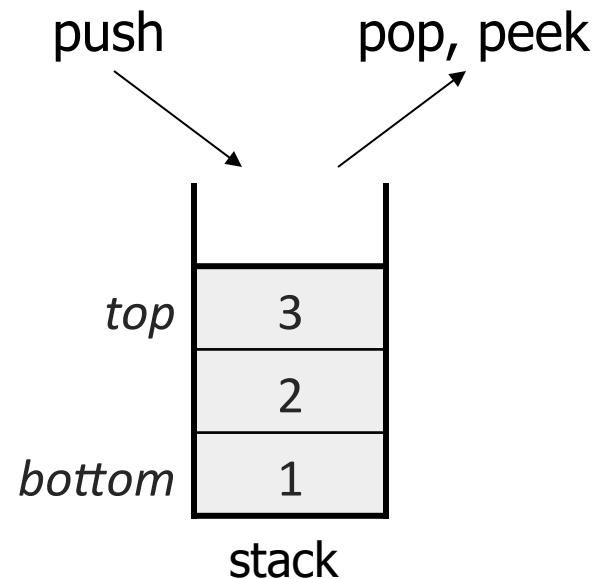
# Stacks and queues

- Sometimes it is good to have a collection that is less powerful, but is optimized to perform certain operations very quickly.
- Two specialty collections:
  - **stack**: Retrieves elements in the reverse of the order they were added.
  - **queue**: Retrieves elements in the same order they were added.



# Stacks

- **stack**: A collection based on the principle of adding elements and retrieving them in the opposite order.
  - Last-In, First-Out ("LIFO")
  - The elements are stored in order of insertion, but we do not think of them as having indexes.
  - The client can only add/remove/examine the last element added (the "top").
- basic stack operations:
  - **push**: Add an element to the top.
  - **pop**: Remove the top element.
  - **peek**: Examine the top element.



# Class Stack

<code>Stack&lt;<b>E</b>&gt;()</code>	constructs a new stack with elements of type <b>E</b>
<code>push(<b>value</b>)</code>	places given value on top of stack
<code>pop()</code>	removes top value from stack and returns it; throws <code>EmptyStackException</code> if stack is empty
<code>peek()</code>	returns top value from stack without removing it; throws <code>EmptyStackException</code> if stack is empty
<code>size()</code>	returns number of elements in stack
<code>isEmpty()</code>	returns <code>true</code> if stack has no elements

```
Stack<Integer> s = new Stack<Integer>();  
s.push(42);  
s.push(-3);  
s.push(17); // bottom [42, -3, 17] top  
System.out.println(s.pop()); // 17
```

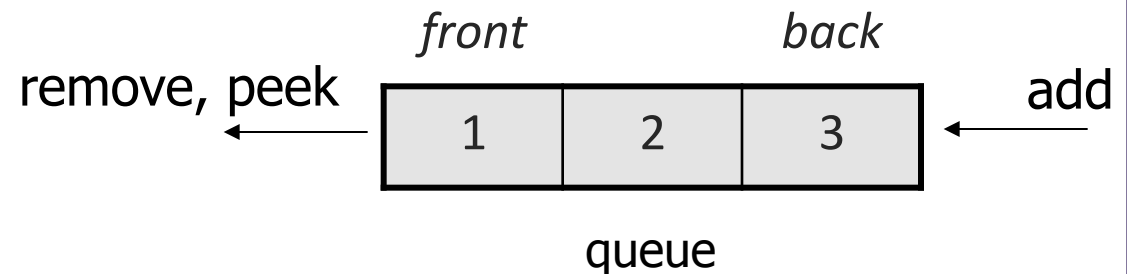
- Stack has other methods, but you should not use them.



# Queues

- **queue**: Retrieves elements in the order they were added.

- First-In, First-Out ("FIFO")
- Elements are stored in order of insertion but don't have indexes.
- Client can only add to the end of the queue, and can only examine/remove the front of the queue.



- basic queue operations:

- **add** (enqueue): Add an element to the back.
- **remove** (dequeue): Remove the front element.
- **peek**: Examine the front element.

# Programming with Queues

<code>add (value)</code>	places given value at back of queue
<code>remove ()</code>	removes value from front of queue and returns it; throws a <code>NoSuchElementException</code> if queue is empty
<code>peek ()</code>	returns front value from queue without removing it; returns <code>null</code> if queue is empty
<code>size ()</code>	returns number of elements in queue
<code>isEmpty ()</code>	returns <code>true</code> if queue has no elements

```
Queue<Integer> q = new LinkedList<Integer>();  
q.add(42);  
q.add(-3);  
q.add(17);           // front [42, -3, 17] back  
System.out.println(q.remove()); // 42
```

- **IMPORTANT:** When constructing a queue you must use a new `LinkedList` object instead of a new `Queue` object.

# Queue idioms

---

- As with stacks, must pull contents out of queue to view them.

```
// process (and destroy) an entire queue  
while (!q.isEmpty()) {  
    do something with q.remove();  
}
```

- another idiom: Examining each element exactly once.

```
int size = q.size();  
for (int i = 0; i < size; i++) {  
    do something with q.remove();  
    (including possibly re-adding it to the queue)  
}
```

# Abstract data types (ADTs)

---

- **abstract data type (ADT):** A specification of a collection of data and the operations that can be performed on it.
  - Describes *what* a collection does, not *how* it does it
- We don't know exactly how a stack or queue is implemented, and we don't need to.
  - We just need to understand the idea of the collection and what operations it can perform.

(Stacks are usually implemented with arrays; queues are often implemented using another structure called a linked list.)

# ADTs as interfaces (11.1)

---

- Java's collection framework uses interfaces to describe ADTs:
  - `Collection`, `Deque`, `List`, `Map`, `Queue`, `Set`
- An ADT can be implemented in multiple ways by classes:
  - `ArrayList` and `LinkedList` implement `List`
  - `HashSet` and `TreeSet` implement `Set`
  - `LinkedList`, `ArrayDeque`, etc. implement `Queue`
    - They messed up on `Stack`; there's no `Stack` interface, just a class.

# Using ADT interfaces

---

When using Java's built-in collection classes:

- It is considered good practice to always declare collection variables using the corresponding ADT interface type:

```
List<String> list = new ArrayList<String>();
```

- Methods that accept a collection as a parameter should also declare the parameter using the ADT interface type:

```
public void stutter(List<String> list) {  
    ...  
}
```

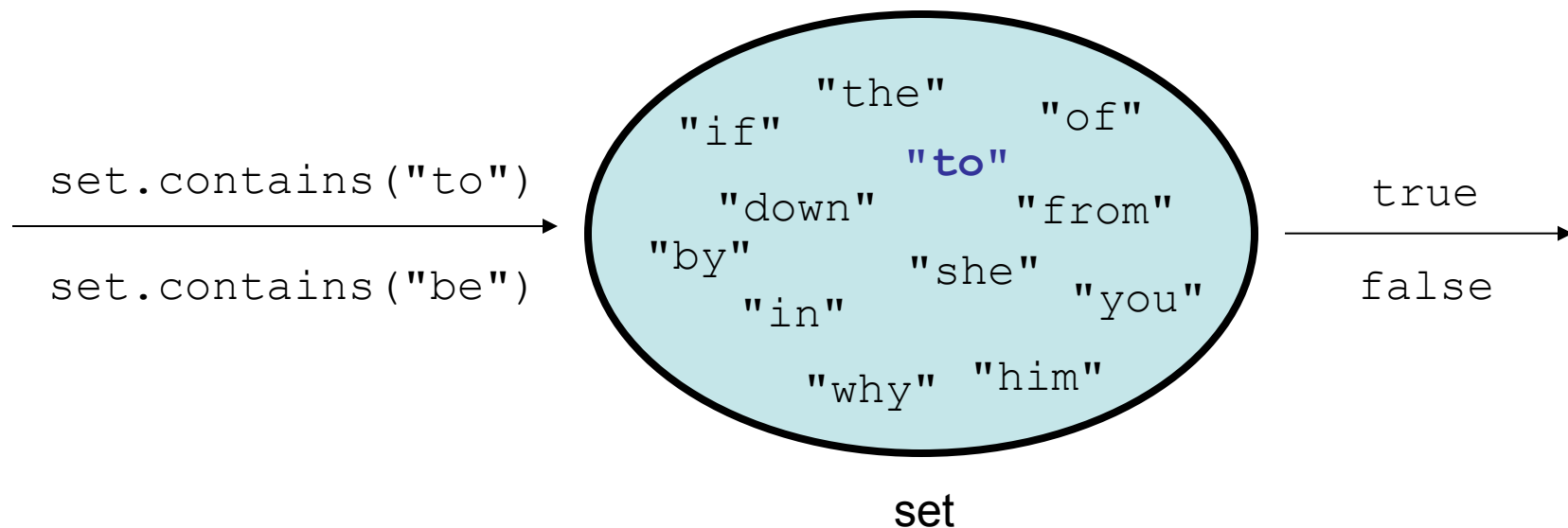
# Why use ADTs?

---

- Why would we want more than one kind of list, queue, etc.?
- Answer: Each implementation is more efficient at certain tasks.
  - `ArrayList` is faster for adding/removing at the end;  
`LinkedList` is faster for adding/removing at the front/middle.  
Etc.
  - You choose the optimal implementation for your task, and if the rest of your code is written to use the ADT interfaces, it will work.

# Sets

- **set**: A collection of unique values (no duplicates allowed) that can perform the following operations efficiently:
  - add, remove, search (contains)
  - We don't think of a set as having indexes; we just add things to the set in general and don't worry about order





# Set implementation

---

- in Java, sets are represented by `Set` interface in `java.util`
- `Set` is implemented by `HashSet` and `TreeSet` classes
  - `HashSet`: implemented using a "hash table" array;  
very fast:  **$O(1)$**  for all operations  
elements are stored in unpredictable order
  - `TreeSet`: implemented using a "binary search tree";  
pretty fast:  **$O(\log N)$**  for all operations  
elements are stored in sorted order
  - `LinkedHashSet`:  **$O(1)$**  but stores in order of insertion

# Set methods

```
List<String> list = new ArrayList<String>();
```

```
...
```

```
Set<Integer> set = new TreeSet<Integer>();           // empty
```

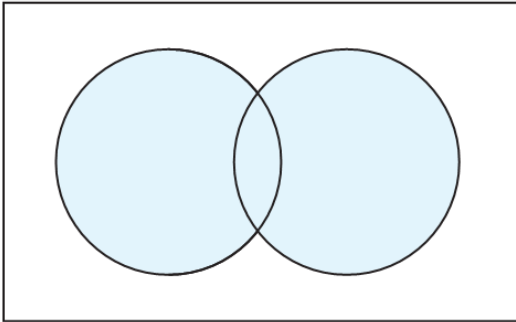
```
Set<String> set2 = new HashSet<String>(list);
```

- can construct an empty set, or one based on a given collection

<code>add (value)</code>	adds the given value to the set
<code>contains (value)</code>	returns <code>true</code> if the given value is found in this set
<code>remove (value)</code>	removes the given value from the set
<code>clear ()</code>	removes all elements of the set
<code>size ()</code>	returns the number of elements in list
<code>isEmpty ()</code>	returns <code>true</code> if the set's size is 0
<code>toString ()</code>	returns a string such as "[3, 42, -7, 15]"

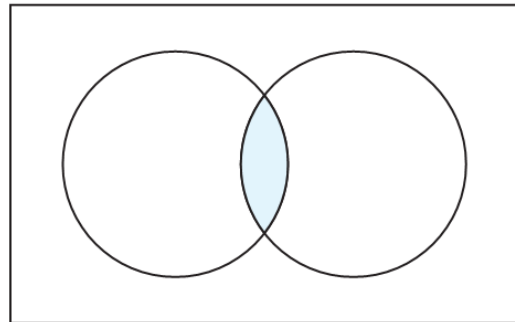
# Set operations

$A \cup B$  Union



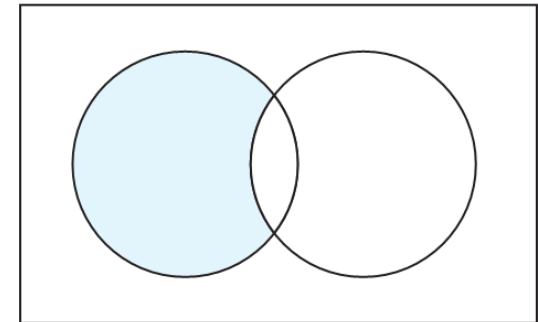
`addAll`

$A \cap B$  Intersection



`retainAll`

$A - B$  Difference



`removeAll`

<code>addAll (<b>collection</b>)</code>	adds all elements from the given collection to this set
<code>containsAll (<b>coll</b>)</code>	returns <code>true</code> if this set contains every element from given set
<code>equals (<b>set</b>)</code>	returns <code>true</code> if given other set contains the same elements
<code>iterator()</code>	returns an object used to examine set's contents ( <i>seen later</i> )
<code>removeAll (<b>coll</b>)</code>	removes all elements in the given collection from this set
<code>retainAll (<b>coll</b>)</code>	removes elements <i>not</i> found in given collection from this set
<code>toArray()</code>	returns an array of the elements in this set

# Sets and ordering

---

- HashSet : elements are stored in an unpredictable order

```
Set<String> names = new HashSet<String>();  
names.add("Jake");  
names.add("Robert");  
names.add("Marisa");  
names.add("Kasey");  
System.out.println(names);  
// [Kasey, Robert, Jake, Marisa]
```

- TreeSet : elements are stored in their "natural" sorted order

```
Set<String> names = new TreeSet<String>();  
...  
// [Jake, Kasey, Marisa, Robert]
```

- LinkedHashSet : elements stored in order of insertion

```
Set<String> names = new LinkedHashSet<String>();  
...  
// [Jake, Robert, Marisa, Kasey]
```

# The "for each" loop (7.1)

---

```
for (type name : collection) {  
    statements;  
}
```

- Provides a clean syntax for looping over the elements of a `Set`, `List`, `array`, or other collection

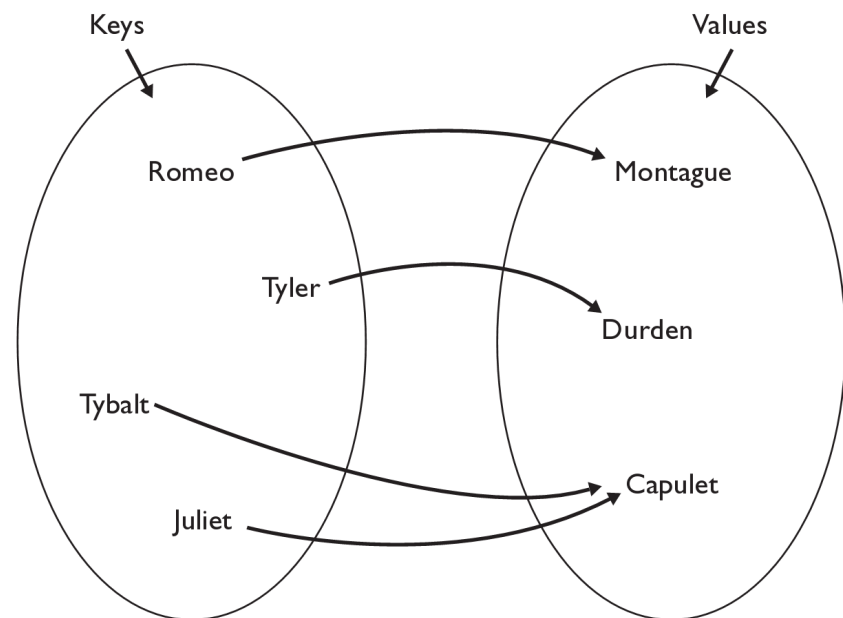
```
Set<Double> grades = new HashSet<Double>();  
...
```

```
for (double grade : grades) {  
    System.out.println("Student's grade: " + grade);  
}
```

- needed because sets have no indexes; can't get element `i`

# The Map ADT

- **map**: Holds a set of unique *keys* and a collection of *values*, where each key is associated with one value.
  - a.k.a. "dictionary", "associative array", "hash"
- basic map operations:
  - **put**(*key*, *value*): Adds a mapping from a key to a value.
  - **get**(*key*): Retrieves the value mapped to the key.
  - **remove**(*key*): Removes the given key and its mapped value.



`myMap.get("Juliet")` returns "Capulet"

# Map concepts

- a map can be thought of as generalization of a tallying array
  - the "index" (key) doesn't have to be an `int`

- recall previous tallying examples from CSE 142

▪ count digits: 22092310907

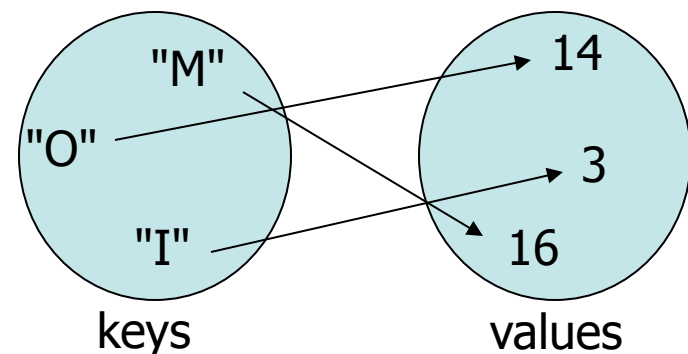
→

index	0	1	2	3	4	5	6	7	8	9
value	3	1	3	0	0	0	0	1	0	2

// (M)cCain, (O)bama, (I)ndependent

- count votes: "MOOOOOOMMMMMOOOOOOOMOMMIMOMMMIMOMMMIO"

key	"M"	"O"	"I"
value	16	14	3



# Map implementation

---

- in Java, maps are represented by `Map` interface in `java.util`
- `Map` is implemented by the `HashMap` and `TreeMap` classes
  - `HashMap`: implemented using an array called a "hash table"; extremely fast:  **$O(1)$**  ; keys are stored in unpredictable order
  - `TreeMap`: implemented as a linked "binary tree" structure; very fast:  **$O(\log N)$**  ; keys are stored in sorted order
  - A map requires 2 type parameters: one for keys, one for values.

// maps from String keys to Integer values

```
Map<String, Integer> votes = new HashMap<String, Integer>();
```



# Map methods

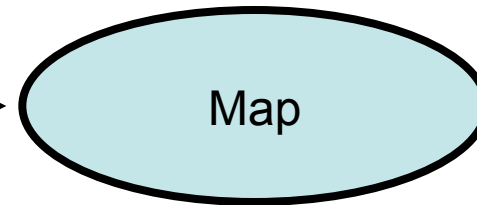
<code>put (key, value)</code>	adds a mapping from the given key to the given value; if the key already exists, replaces its value with the given one
<code>get (key)</code>	returns the value mapped to the given key ( <code>null</code> if not found)
<code>containsKey (key)</code>	returns <code>true</code> if the map contains a mapping for the given key
<code>remove (key)</code>	removes any existing mapping for the given key
<code>clear ()</code>	removes all key/value pairs from the map
<code>size ()</code>	returns the number of key/value pairs in the map
<code>isEmpty ()</code>	returns <code>true</code> if the map's size is 0
<code>toString ()</code>	returns a string such as " <code>{a=90, d=60, c=70}</code> "
<code>keySet ()</code>	returns a set of all keys in the map
<code>values ()</code>	returns a collection of all values in the map
<code>putAll (map)</code>	adds all key/value pairs from the given map to this map
<code>equals (map)</code>	returns <code>true</code> if given map has the same mappings as this one

# Using maps

- A map allows you to get from one half of a pair to the other.
  - Remembers one piece of information about every index (key).

//    **key**            **value**

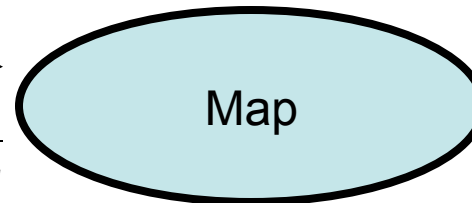
`put("Joe", "206-685-2181")`



- Later, we can supply only the key and get back the related value:  
Allows us to ask: *What is Joe's phone number?*

`get("Joe")`

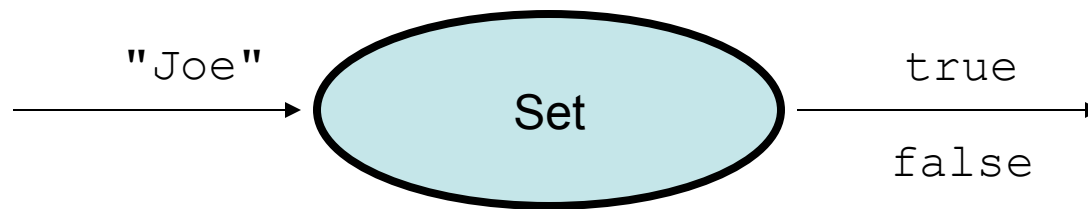
`"206-685-2181"`



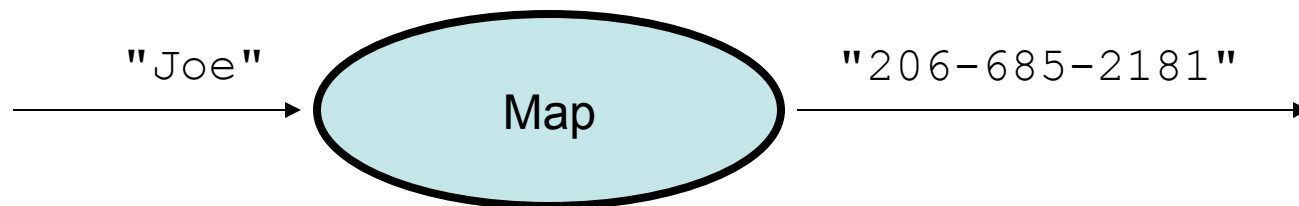
# Maps vs. sets

---

- A set is like a map from elements to `boolean` values.
  - *Set: Is Joe found in the set? (true/false)*



- *Map: What is Joe's phone number?*



# keySet and values

---

- `keySet` method returns a `Set` of all keys in the map
  - can loop over the keys in a `foreach` loop
  - can get each key's associated value by calling `get` on the map

```
Map<String, Integer> ages = new TreeMap<String, Integer>();
ages.put("Joe", 19);
ages.put("Geneva", 2); // ages.keySet() returns Set<String>
ages.put("Vicki", 57);
for (String name : ages.keySet()) { // Geneva -> 2
    int age = ages.get(name); // Joe -> 19
    System.out.println(name + " -> " + age); // Vicki -> 57
}
```

- `values` method returns a collection of all values in the map
  - can loop over the values in a `foreach` loop
  - no easy way to get from a value to its associated key(s)

# Priority queue ADT

---

- **priority queue**: a collection of ordered elements that provides fast access to the minimum (or maximum) element
  - usually implemented using a tree structure called a *heap*
- **priority queue operations**:

▪ add	adds in order;	$O(\log N)$ worst
▪ peek	returns <b>minimum</b> value;	$O(1)$ always
▪ remove	removes/returns <b>minimum</b> value;	$O(\log N)$ worst
▪ isEmpty, clear, size, iterator		$O(1)$ always

# Java's PriorityQueue class

```
public class PriorityQueue<E> implements Queue<E>
```

Method/Constructor	Description	Runtime
<code>PriorityQueue&lt;E&gt;()</code>	constructs new empty queue	$O(1)$
<code>add(E value)</code>	adds value in sorted order	$O(\log N)$
<code>clear()</code>	removes all elements	$O(1)$
<code>iterator()</code>	returns iterator over elements	$O(1)$
<code>peek()</code>	returns minimum element	$O(1)$
<code>remove()</code>	removes/returns min element	$O(\log N)$

```
Queue<String> pq = new PriorityQueue<String>();  
pq.add("Stuart");  
pq.add("Marty");  
...
```

# Priority queue ordering

---

- For a priority queue to work, elements must have an ordering
  - in Java, this means implementing the `Comparable` interface

- Reminder:

```
public class Foo implements Comparable<Foo> {  
    ...  
    public int compareTo(Foo other) {  
        // Return positive, zero, or negative number  
    }  
}
```