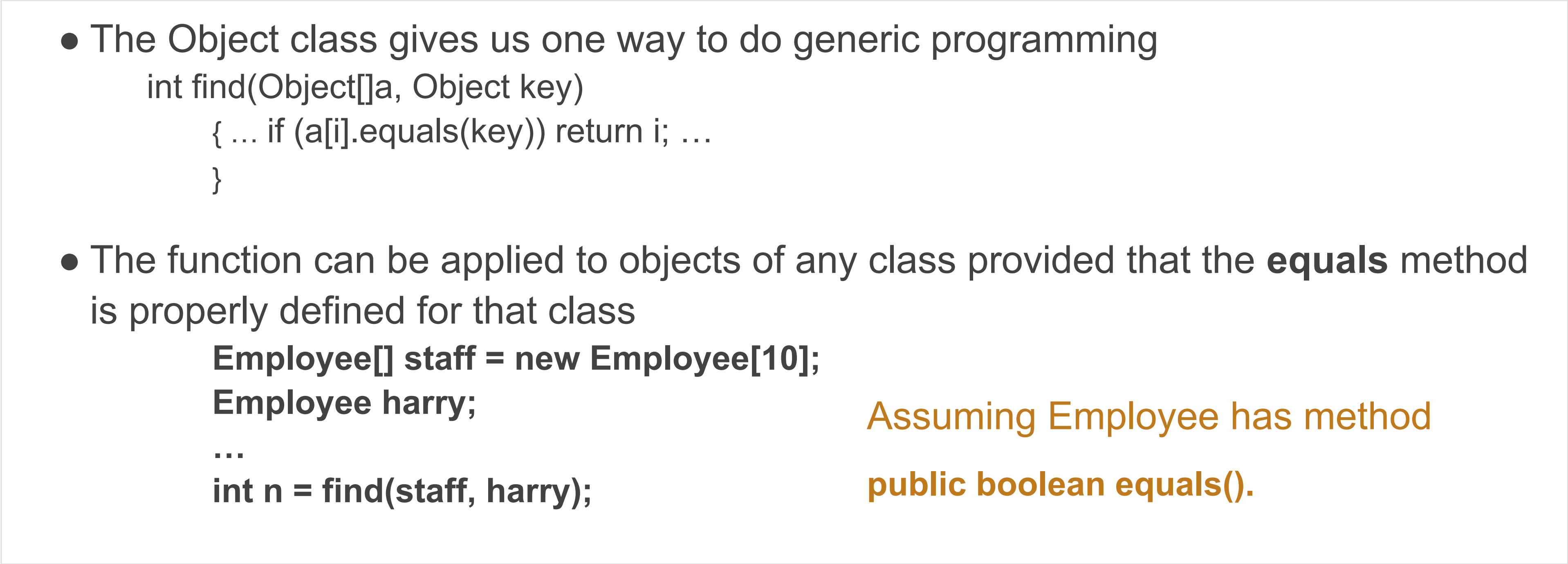
Interfaces

Interfaces/Outline

- Interfaces are a way to describe *what* classes should do without specifying *how* they should do it.

- Outline
  - Interface basics
    - Introduction
    - Defining interfaces
    - Using interfaces
  - Interfaces vs abstract classes
  - Callbacks via interfaces: common use of interfaces
  - The Cloneable Interface: a special interface

Introduction to Interfaces



- The Object class gives us one way to do generic programming

```
int find(Object[]a, Object key)
      { … if (a[i].equals(key)) return i; …
      }
```

- The function can be applied to objects of any class provided that the **equals** method is properly defined for that class

```
Employee[] staff = new Employee[10];
Employee harry;
…
int n = find(staff, harry);
```

Assuming Employee has method

public boolean equals().

- Employee inherits the equals method from Object.
    - Logical error if not refined

Introduction to Interfaces

- Use interfaces to ensure that equality test properly implemented

1. Define interface

```
public interface HasMyEquals
{ boolean myEquals(Object key); }
```

2. Change signature of the find method

```
public static Object find( HasMyEquals[] A, Object key)
{ …if ( A[i].myEquals( key ) )…}
```

3. Consequences when writing Employee
    1. Must implement the HasMyEquals interface

       ```
       Class Employee implements HasMyEquals {…}
       ```

    2. Must provide myEquals method

       ```
       boolean myEquals(Object k) {…}
       ```

Interfaces/Outline

- Outline
  - Interface basics
    - Introduction
    - Defining interfaces
    - Using interfaces
  - Interfaces vs abstract classes
  - Callbacks via interfaces: common use of interfaces
  - The Cloneable Interface: a special interface

Defining Interfaces

- General skeleton:

```
public interface NameofInterface [extends AnotherInterface]
{ method1;
method2;
…
constant1;
constant2; …
}
```

  - All methods are abstract by default, no need for modifier **abstract**
  - All fields are constants by default, no need for modifier **static final**
  - All methods and constants have **public** access by default, no need for modifier **public**.

Defining Interfaces

- An example

```
public interface Moveable
{ void move( doube x, double y);
}

public interface Powered extends Moveable
{ String powerSource();
int SPEED_LIMIT = 95;
}
```

Interfaces/Outline

- Outline
  - Interface basics
    - Introduction
    - Defining interfaces
    - <u>Using interfaces</u>
  - Interfaces vs abstract classes
  - Callbacks via interfaces: common use of interfaces
  - The Cloneable Interface: a special interface

Using Interfaces

- Interface to use in in the following: java.lang.Comparable

```
Public interface Comparable
{ int compareTo(Object other);
}
```

- When to use? When defining a class
- How?
  - Declare that your class implements that given interface

    ```
    class Employee implements Comparable {
    ```

  - Provide definition of methods in the interface. The **public** access modifier must be provided here.

    ```
    public int compareTo(Object otherObject) // argument type must
    { Employee other = (Employee)otherObject; // match definition
    if (salary < other.salary) return -1;
    if (salary > other.salary) return 1;
    return 0;
    }
    ```

- **If some methods of the interface are not implemented, the class must be declared as abstract (why?)**

Using Interfaces

- Why do I need to have my class implement an interface?
    - Employee now implements Comparable, so what?
    - Can use the sorting service provided by the Arrays class

        - public static void **sort**(Object[] a)
            - Sorts the specified array of objects into ascending order, according to the *natural ordering* of its elements.
            - **All elements in the array must implement the Comparable interface**. …

Using interfaces

- Although no multiple inheritance, a Java class can implement multiple interfaces

  class Employee implements Comparable, Cloneable

- If a parent class implements an interface, subclass does not need to explicitly use the implement keyword. (Why?)

```
class Employee implements Comparable, Cloneable
{ public Object clone() {…}
}
class manager extends Employee
{ public Object clone() {…}
}
```

Using Interfaces

- Interfaces are not classes. You cannot instantiate interfaces, i.e. cannot use the **new** operator with an interface

  new Comparable(); // illegal

- Can declare interface variables

  Comparable x; // x can refer to an object that has the
  // behavior specified in Comparable
  x = new Employee();
  //ok if Employee implements Comparable

- Can use **instanceOf**

  if (x instanceOf Comparable) …
  // Does x have the behavior specified in Comparable?

Interfaces/Outline

- Outline
  - Interface basics
    - Introduction
    - Defining interfaces
    - Using interfaces
  - Interfaces vs abstract classes
  - Callbacks via interfaces: common use of interfaces
  - The Cloneable Interface: a special interface

Interfaces and Abstract Classes

- Interfaces are more abstract than abstract classes.
- Interfaces cannot have static methods, abstract classes can
- Interfaces cannot contain implementations of methods, abstract classes can
- Interfaces cannot have fields, abstract classes can.

```
abstract class Person
        { public Person(String n)
        { name = n;}
        public abstract String getDescription();
        public String getName()
        { return name;}
        private String name;
        }
```

Interfaces and Abstract Classes

- Are interfaces a necessity (from the point of view of language design) given that we have abstract classes?

- In order to sort an array of Employees, can we simply do the following?

```
abstract class Comparable
{ public abstract int CompareTo(Object other);}


Void sort(Comparable[] A)

class Employee extends Compareable
{ public int CompareTo(Object other) {…}
…
}
```

Interfaces and Abstract Classes

- Cannot do this if Employee already extends another class

  class Employee extends Person …

  Because we cannot have

  class Employee extends Person, Comparable …

Interfaces/Outline

- Outline
  - Interface basics
    - Introduction
    - Defining interfaces
    - Using interfaces
  - Interfaces vs abstract classes
  - Callbacks via interfaces: common use of interfaces
  - The Cloneable Interface: a special interface

Interfaces and Callbacks

- Interfaces provide a good way to write callbacks
  - The program **TimerTest.java** prints “The time now is …” every second.
  - How does it work?
    - There is a timer (javax.swing.Timer) that keeps track of time.
    - How do we tell the timer what to do when the time interval (1 second) has elapsed?
    - Answer: Callbacks
      - In many languages, we supply the name of a function the timer should call periodically.
      - In Java, we supply the timer with an object of some class.

Interfaces and Callbacks

- Questions
  - What method of the object that the timer should invoke?
  - How do we make sure that the object has the method?

- Solution:
  - The **ActionListener** interface

    ```
    java.awt.event.ActionListener
    public interface ActionListener
    {
    void actionPerformed(ActionEvent event);
    }
    ```
  - **Timer t = new Timer(int Delay, ActionListener obj);**

  - The timer calls the **actionPerformed** method when the time interval has elapsed.

Interfaces and Callbacks

- Make sure listener object has the methods:
  - It must be an object of class that implements that **ActionListener**

```
class TimePrinter implements ActionListener
{
public void actionPerformed(ActionEvent event)
{
Date now = new Date(); //java.util
System.out.println("The time now is " + now);
}
}
```

Interfaces and Callbacks

```
public class TimerTest
{ public static void main(String[] args)
{
ActionListener listener = new TimePrinter();

// construct a timer that calls the listener
// once every 1 second
Timer t = new Timer(1000, listener);
t.start(); // start timer
// continue until told to stop
JOptionPane.showMessageDialog(null,"Quit program?");
System.exit(0);
}
}
```

Interfaces/Outline

- Outline
  - Interface basics
    - Introduction
    - Defining interfaces
    - Using interfaces
  - Interfaces vs abstract classes
  - Callbacks via interfaces: common use of interfaces
  - The Cloneable Interface: a special interface

The Cloneable Interface

A clone of an object is a new object that has the same state as the original but with a different identity. In particular you can modify the clone without affecting the original. (Deep copy)

- In order to clone objects of a class, you must have the class
  - have the class implements the **Cloneable** interface
  - redefine the **clone** method
  - change its access modifier to **public**.

- Example: CloneTest.java
  - The next several slides are based on this example

The Cloneable Interface

- What is the purpose of these rules?
  - Cloning is tricky. This is to reduce programming mistakes

- How the rules are enforced by java?
  - Object has clone method.
  - All other classes are subclasses of Object.
  - So they all inherit the clone method.
  - Why MUST the clone method be redefined, change to public, and the class must implement the Cloneable interface?

Cloning is tricky

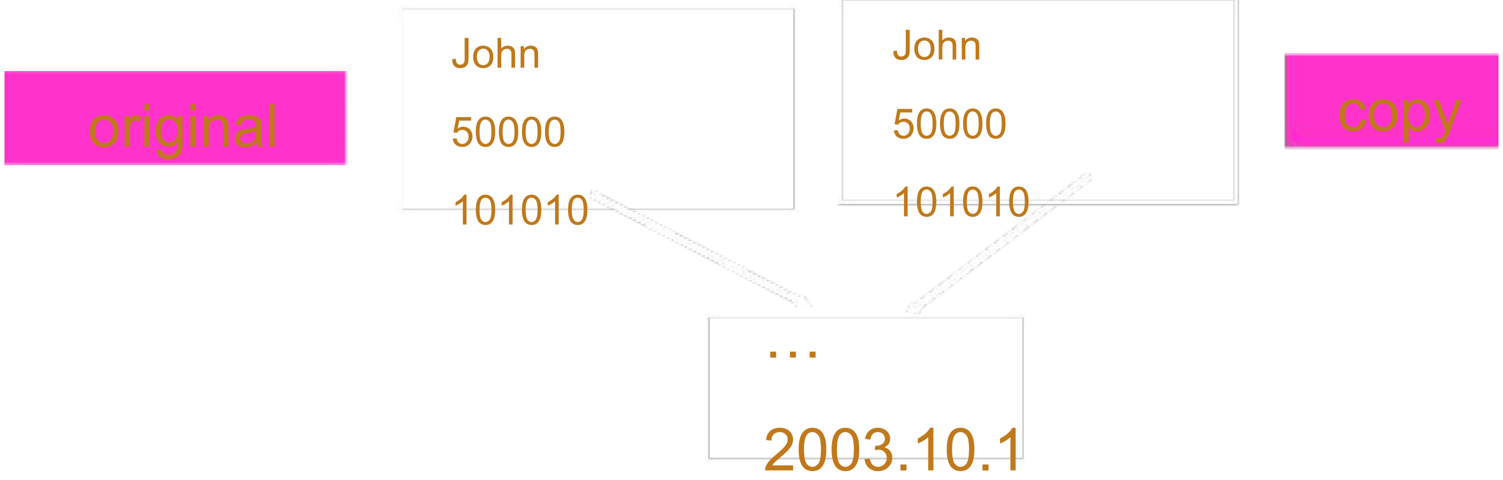
- Default implementation of the **clone()** method of **Object**
  - Copies bit-by-bit.
  - Ok for copying objects whose fields are primitive types
  - Not ok for cloning objects with reference fields.

Cloning is tricky

- clone method of Object does shallow copying.
  - **Employee original = new Employee("John Q. Public", 50000);**
    original.setPayDay(2003, 10, 1);
    Employee copy = original.clone();

  payDay not copied!

Actually compiler error. But let’s assume it is alright for now



What would happen if copy was paid 30 days earlier?

copy.payDay.addPayDay(-30);

But this also affects original.hireDay!

CloneTest1.java

How is the problem solved?

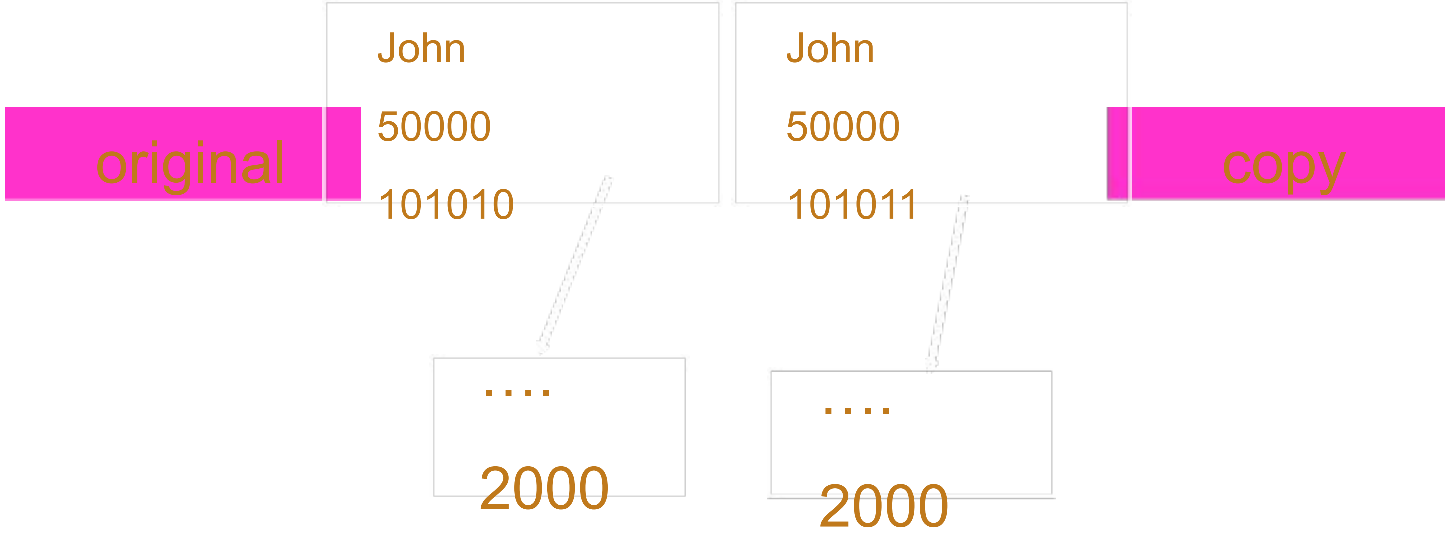
- Suppose we want to clone Employee.

```
public class Employee implements Cloneable
{ …
public Object clone()
{ try
{ // call Object.clone()
Employee copy = (Employee)super.clone();

// clone mutable fields
copy.payDay=(GregorianCalendar)payDay.clone();

return copy;
}
catch (CloneNotSupportedException e)
{return null;}
}
```

CloneTest.java

How is the problem solved?

- **Employee original = new Employee("John Q. Public", 50000);**
  original.setPayDay(2000, 1, 1);
  Employee copy = original.clone();
  payDay copied!



What would happen if copy was paid 14 days earlier?

copy.payDay.addPayDay(-30);

This has NO affects original.payDay!

The Cloneable Interface

- What is the purpose of these rules?
  - Cloning is tricky. This is to reduce programming mistakes

- How the rules are enforced by java?
  - Object has clone method.
  - All other classes are subclasses of Object.
  - So they all inherit the clone method.
  - Why MUST the clone method be redefined, change to public, and the class must implement the Cloneable interface?

How are the rules enforced?

- **Protected access revisited:**
  - **protected** fields/methods can be accessed by classes in the same package.
  - Example:

```
package Greek;
public class Alpha {
protected int iamprotected;
protected void protectedMethod() {…}
}

package Greek;
class Gamma {
void accessMethod() {
Alpha a = new Alpha();
a.iamprotected = 10; // legal
a.protectedMethod(); // legal
}}
```

How are the rules enforced?

- **protected** fields/methods can also be accessed by subclasses

- However: A subclass in a different package can only access protected fields and methods on objects of that subclass (and it’s subclasses)

```
        package Latin;
        import Greek.*;
class Delta extends Alpha {
void accessMethod(Alpha a, Delta d) {
d.iamprotected = 10; // legal
d.protectedMethod(); // legal
a.iamprotected = 10; // illegal
a.protectedMethod(); // illegal
}
}
```

How are the rules enforced?

- How does java disallow calling the clone method of Object to clone an Employee?

- The clone method of Object is protected.
- CloneTest not in the same package as Object.
- Hence cannot call the clone method of Object to clone an Employee.
  - In CloneTest, one can only clone objects of CloneTest

- Must override before use
  - If not, we get compiler error

    “clone() has protected access in java.lang.Object
    Employee copy = (Employee)original.clone();”
    ^

How is the mechanism enforced?

- Can we refine the clone method, but do not implement the Cloneable interface?

```
class Employee //implements Cloneable
{ public Object clone() {…}
}
```

- No. The clone method throws CloneNotSupportedException at runtime if called on objects whose class does not implement the Cloneable interface.

- All Arrays implement the Cloneable iterface

Consequences

- Redefinition of **clone** method is necessary even when the default is good enough.
    - Compiler does not any idea whether the default is good enough.

```
class Person implements Cloneable { …
public Object clone()
{ try
{ return super.clone();
} catch (CloneNotSupportedException e)
{
return null; }
//This won't happen, since we are Cloneable□
}
}
```

Tagging Interface

- What are the methods in **Cloneable**?

- The **clone** method is inherited from class **Object**. It is not a method of the **Cloneable** interface.

- The **Cloneable** interface has no methods and hence called a tagging interface.
- Technically, it prevents the clone method to throw CloneNotSupportedException at runtime.

- For programmers, it indicates that the class designer understand the clone process and can decide correctly whether to refine the **clone** method.

- The **Serializable** interface is another tagging interface.

Inner Classes/Outline

- Introduction
  - Inner classes through an example
- Local inner classes
- Anonymous Inner classes
- Static inner classes

Introduction to Inner Classes

- An inner class is a class defined inside another class
- Similar to nested classes in C++, but more flexible & more powerful.

- Useful because:
  - Object of inner class can access private fields and methods of outer class.
  - Can be hidden from other classes in the same package. Good for, e.g., nodes in linked lists or trees.
  - Anonymous inner classes are handy when defining callbacks on the fly
  - Convenient when writing event-driven programs.

Inner Classes Through An Example

- Task: Write a program that adds interest to a bank account periodically. Following the TimerTest example, we write

```
public class AddInterest
{ public static void main(String[] args)
{ // construct a bank account with initial balance of $10,000
BankAccount account = new BankAccount(10000);
// construct listerner object to accumulate interest at 10%
ActionListener adder = new InterestAdder(10,
account);
// construct timer that call listener every second
Timer t = new Timer(1000, adder);
t.start();
… // termination facility
}
} // AddInterest.java
```

```
class InterestAdder implements ActionListener
    ….//print out current balance
    {
    public InterestAdder(double rate,
    BankAccount account)
    { this.rate = rate; this.account = account;
    }
    public void actionPerformed(ActionEvent event)
    { // compute interest & update account balance
    double interest = account.getBalance() * rate/ 100;
    account.setBalance( account.getBalance()+interest);
    }
    private BankAccount account; private double rate;
    }
```

- Note that InterestAdder requires BankAccount class to provide public accessor getBalance and mutator setBalance.

Inner Classes Through An Example

- The BankAccount class

```
class BankAccount
   { public BankAccount(double initialBalance)
   { balance = initialBalance;}
   public void setBalance( double balance)
   { this.balance = balance;}

   public double getBalance()
   { return balance;}

   private double balance;
   }
```

Inner Classes Through An Example

- The program AddInterest.java works

- BUT, not satisfactory:
  - **BankAccount** has **public** accessor **getBalance** and mutator **setBalance**.
  - Any other class can read and change the balance of an account!

- Inner classes provide a better solution
  - Make **InterestAdder** an inner **private** class of **BankAccount**
  - Since inner classes can access fields and methods of outer classes, **BankAccount** no longer needs to provide **public** accessor and mutator.
  - The inner class **InterestAdder** can only be used inside **BankAcccount**.

Inner Classes Through An Example

```
class BankAccount
{ …
private double balance;
private class InterestAdder implements ActionListener
{ public InterestAdder(double rate)
{ this.rate = rate; }

public void actionPerformed(ActionEvent event)
{ // update interest
double interest = balance * rate / 100;
balance += interest;
…// print out current balance
}
private double rate;
}
}
```

Access field of outer class directly

Access field of outer class directly

Inner Classes Through An Example

- Only inner classes can be **private**.
- Regular classes always have either **package** or **public** visibility.

Inner Classes Through An Example

- InterestAdder can only be used inside BankAccount, so we need to place timer inside BankAccount also:

```
class BankAccount
{
public BankAccount(double initialBalance)
{ balance = initialBalance;}

public void start(double rate)
{ ActionListener adder = new InterestAdder(rate);
Timer t = new Timer(1000, adder);
t.start();
}
…
}
```

Inner Classes Through An Example

- The driver class:

```
public class InnerClassTest
{
public static void main(String[] args)
{
// construct a bank account with initial balance of $10,000
BankAccount account = new BankAccount(10000);

// start accumulating interest at 10%
account.start(10);

JOptionPane.showMessageDialog(null,"Quit program?");
System.exit(0);
}
} //InnerClassTest.java
```

How to prevent the start method of BankAccount being called more than once?

Inner Classes/Outline

- Introduction
    - Inner classes through an example
- Local inner classes
- Anonymous Inner classes
- Static inner classes

Local Inner Classes

- Note that in BankAccount class:
    - The class InterestAdder is used only once in method start.

- In this case, Java lets you define class InterestAdder locally inside method **start**.

Local Inner Classes

```
public void start(double rate)
{ class InterestAdder implements ActionListener
    { public InterestAdder(double rate)
    { this.rate = rate; }

    public void actionPerformed(ActionEvent event)
    { double interest = balance * rate / 100;
    balance += interest;
    …// print out current balance
    }
    private double rate;
    }
ActionListener adder = new InterestAdder(rate);
    Timer t = new Timer(1000, adder);
    t.start();
```

Local Inner Classes

- A local class is never declared with an access modifier. Its scope restricted to the scope of the method within which it is defined.

- Local class can be accessed only by the method within which it defined.

- It can access local variables if there are **final**. Example on the next slide.

Local Inner Classes

```
public void start( final double rate)
{ class InterestAdder implements ActionListener
    { // no constructor now needed in this case

    public void actionPerformed(ActionEvent event)
    { double interest = balance * rate / 100;
    balance += interest;
    …// print out current balance
    }
    // the rate field is gone.
    }
ActionListener adder = new InterestAdder();
    Timer t = new Timer(1000, adder);
    t.start();
}
```

Inner Classes/Outline

- Introduction
  - Inner classes through an example
- Local inner classes
- Anonymous Inner classes
- Static inner classes

Anonymous Inner Classes

- Anonymous inner classes take this one step further.
  - Kind of replacing the usage of **InterestAdder** with its definition.

```
public void start( final double rate)
{
ActionListener adder = new ActionListener()
    { public void actionPerformed(ActionEvent event)
    {
    double interest = balance * rate / 100;
    balance += interest;
    … // print out current balance
    }
    }
Timer t = new Timer(1000, adder);
    t.start();
} // AnonymousInnerClassTest.java
```

Analogy

X=A+B+10;

Y=2+X;

Substitution:

Y=2+A+B+10;

Anonymous Inner Classes

- General syntax:
  - **new someInterface()** {…}

    creates an object of an anonymous inner class that implements **someInterface**
  - **new someClass(constructionParameters){…}**

    creates an object of an anonymous inner class that extends **someClass**.
- Note: An anonymous inner class cannot have constructors
  - Reason: constructors must have the same name as class and as anonymous class has no name.
  - Implication: Construction parameters passed to super class constructor.
  - Implication: An anonymous class that implements an interface cannot have construction parameters.

Question

- Question: Why is this?
  - new someInterface(); // illegal
  - new someInterface(){..}; //ok

Inner Classes/Outline

- Introduction
  - Inner classes through an example
- Local inner classes
- Anonymous Inner classes
- Static inner classes

Static Inner Classes

- Static inner classes are inner classes that do not have reference to outer class object.

```
class ArrayAlg
{ public static class Pair
{ public Pair(double f, double s) {…}
public double getFirst(){…}
public double getSecond(){…}
private double first;
private double second;
}
public static Pair minmax(double[] d)
{ // finds minimum and maximum elements in array
return new Pair(min, max);
}
}
```

Same as nested classes in C++

Static Inner Classes

- Static inner classes can be used to avoid name clashes.
- For example, the Pair class in our example is known to the outside as ArrayAlg.Pair.
- Avoid clashing with a Pair class that is defined elsewhere and has different contents, e.g. a pair of strings.

StaticInnerClassTest.java