Outline

- **Information Hiding**
- **Immutability**
  - Immutability: Definition & Advantages
  - Implementing Immutables

Information Hiding

- **Definition**
  - The principle of **information hiding** is the hiding of *design decisions (implementation details)* in a computer program that are most likely to change, thus protecting other parts of the program from change if the design decision is changed.
  - The protection involves providing a stable interface which shields the remainder of the program from the implementation (the details that are most likely to change).

Information Hiding

- **Example**

```
public class RectangleFigure implements Figure {
   public Rectangle bounds;

   public void draw(Graphics g) { … }
   public boolean contains(int x, int y) {
      return bounds.contains(x, y);
   }

   public void move(int dx, int dy) { // controlled state change
      bounds.translate(dx, dy);
      notifyChanges(new FigureChangedEvent(this));
   }
   ...
}
```

Information Hiding

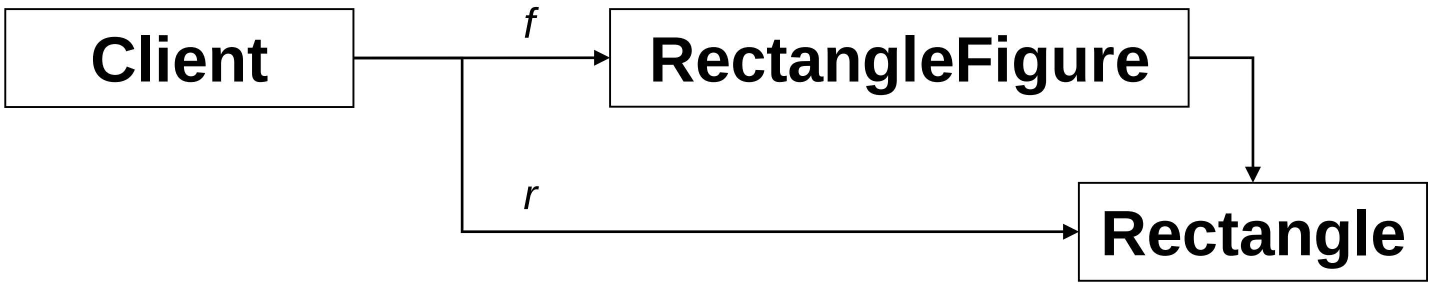
- **Example**

```
public class RectangleFigure implements Figure {
   private Rectangle bounds;
   public Rectangle getBounds() { return bounds; }

   public void draw(Graphics g) { … }
   public boolean contains(int x, int y) {
      return bounds.contains(x, y);
   }

   public void move(int dx, int dy) { // controlled state change
      bounds.translate(dx, dy);
      notifyChanges(new FigureChangedEvent(this));
   }
   ...
}
```

Information Hiding

- **Example**

```
RectangleFigure f = new RectangleFigure();
Rectangle r = f.getBounds();
r.setLocation(100, 100);
r.setSize(50, 30);
```



Mutable Parameters

- **Problem:**
  - Mutable parameters can be modified by the caller
  - Modifications can cause applications to behave incorrectly
  - Modifications to sensitive security state may result in elevated privileges for the attacker
    - E.g. altering the signers of a class can give the class access to unauthorized resources

Example from JDK 1.1

- **getSigners**

```
package java.lang;
public class Class {
   private Object[] signers;
   public Object[] getSigners() { return signers; }
   ...
}
```

  - Actually, getSigners is implemented as native method, but the behavior was equivalent to the above

- **Attacker could change the signers of a class**

```
Object[] signers = obj.getClass().getSigners();
signers[0] = <new signer>;
```

Robust Programming Guideline

- **Make a copy of mutable output parameters**

```
public Object[] getSigners() { return signers.clone(); }
public Rectangle getBounds() { return (Rectangle)bounds.clone(); }
```

- **Remarks**
  - Copies can be created using
    - clone()   creates a new instance with the same dynamic type
    - new Rectangle(r)   copy constructor, creates a new instance of the given type
  - Perform deep cloning on arrays if necessary!

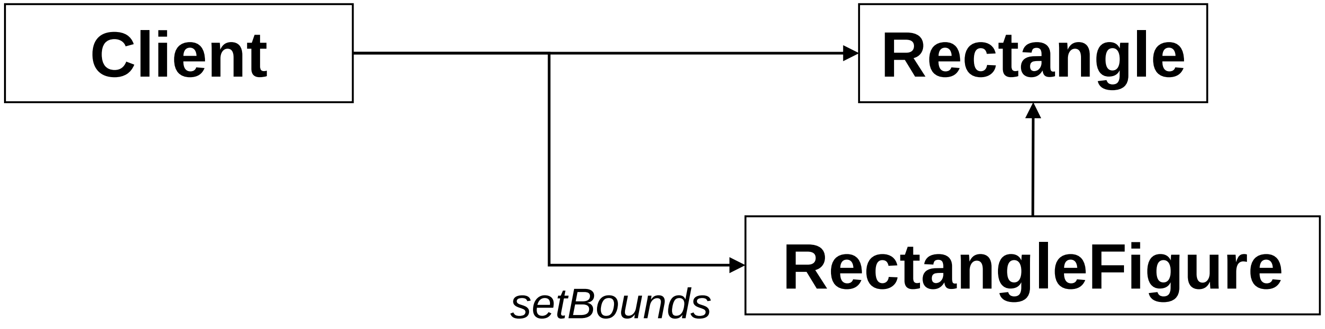
Rectangle Example revisited

```java
public class RectangleFigure implements Figure {
   private Rectangle bounds;
   public Rectangle getBounds() { return (Rectangle)bounds.clone(); }

   public void move(int dx, int dy) { // controlled state change
      bounds.translate(dx, dy);
      notifyChanges(new FigureChangedEvent(this));
   }

   public void setBounds(Rectangle r) { // controlled state change
      this.bounds = r;
      notifyChanges(new FigureChangedEvent(this));
   }
   ...
}
```

Question:

Is an uncontrolled change of the bounds still possible?
(uncontrolled = change without notification)

Robust Programming Guideline



Client

Rectangle

RectangleFigure

setBounds

- **Make a copy of mutable input parameters**

```
    public void setBounds(Rectangle r) {
       this.bounds = (Rectangle)r.clone();
       notifyChanges(new FigureChangedEvent(this));
    }
```

Robust Programming Guideline

- **Input & Output-parameters of which type *T* do not have to be copied?**

```
public class X {
  private T value;
  public T getValue() { return value; }
  public void setValue(T value) {
     this.value = value; notifyChange(…);
  }
}
```

  - Primitive data types (int, long, short, double, float, boolean, char, byte)
  - Immutable objects

Outline

- **Information Hiding**
- **Immutability**
  - Immutability: Definition & Advantages
  - Implementing Immutables

Immutable Objects

- **Definition**
  - Immutable objects are objects whose state cannot be modified after creation (they can only be in one state)
- **Examples**
  - java.lang.String
  - java.awt.Color
  - BigInteger, BigDecimal, MathContext,
  - Boolean, Byte, Character, Double, Float, Integer, Long, Short, Enum
  - URI, URL
  - InetAddress, Inet4Address, Inet6Address
  - LocalTime, LocalDate, LocalDateTime
  - UUID
  - Pattern

red classes are not final

Immutable Objects: Benefits

- **Consistency**
  - Immutables can freely be shared
  - Immutables cannot become inconsistent
- **Thread Safety**
  - Immutables are inherently thread-safe, so you don't have to synchronize access to them across threads

    => simplifies the process of writing thread-safe programs
- **Safe in the presence of ill-behaved code**
  - Methods that take objects as parameters should not change their state, unless documented

    => With mutable object: act of faith

    => With immutable objects: safety

Immutable Objects: Benefits

- **Caching**
  - Immutables can freely be cached, they cannot be changed
  - Fields or method results of immutable types can be cached without worrying about the values becoming stale or inconsistent with the rest of the object's state
  - Example:

```
Date d = new Date(); // current date, not immutable!
scheduler.scheduleTask(task1, d);        // start now
d.setTime(d.getTime() + ONE_DAY);
scheduler.scheduleTask(taks2, d);        // start in one day
```

  - Because `Date` is mutable, the `scheduleTask` method must be careful to defensively copy the `date` parameter, otherwise both task1 & task2 execute tomorrow
  - Date objects in the new data API are immutable

Immutable Objects: Benefits

- **Good Keys**
  - Immutables generally make the best map keys, as some mutables define their hashCode value depending on their state
  - Example

```
public static void main(String[] args) {
   HashSet<Date> set = new HashSet<>();
   Date key = new Date();
   set.add(key);
   key.setTime(key.getTime() + 24 * 60 * 60 * 1000);
   for(Date d : set) System.out.println(d);
   System.out.println(set.contains(key));
}
```

  - Output

```
Fri Apr 28 09:44:12 CEST 2017
false
```

Immutable Objects: Example Fraction

- **Immutables may protect yourself**

```
public class Fraction { // class Fraction is mutable
   private int n, d;

   // Constructors
   public Fraction(int numer, int denom) {
      if(denom == 0) throw new IllegalArgumentException();
      int g = gcd(numer, denom);
      this.n = numer / g; this.d = denom / g;
      if(this.d < 0) { this.n = -this.n; this.d = -this.d; }
   }
   public Fraction(int numer) { this(numer, 1); }
   public Fraction(Fraction f) { this(f.n, f.d); }

   // Accessors
   public double getNumerator() { return this.n; }
   public double getDenominator() { return this.d; }
   public String toString() { return n+ " / " + d; }
```

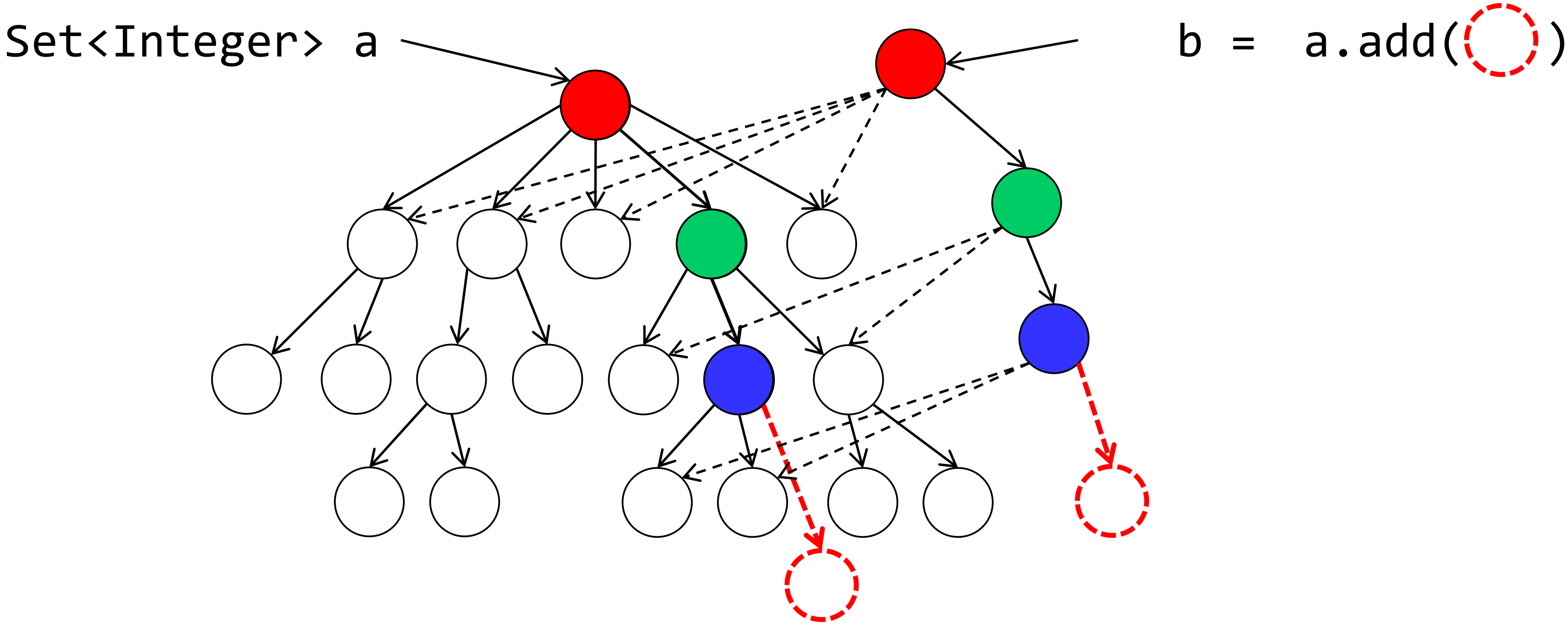
Immutable Objects: Example Fraction

```
public void divide(Fraction y) {      // this = this / y
   this.n *= y.d;                     // i.e. Fraction is mutable
   this.d *= y.n;
   int g = gcd(this.n, this.d);
   this.n /= g;
   this.d /= g;
   if(this.d < 0) { this.n = -this.n; this.d = -this.d; }
}
```

- Result?

```
Fraction f = new Fraction(1, 2);         // f = 1/2
f.divide(f);
System.out.println(f);
```

Persistent / Immutable Data Structures



```
Set<Integer> a
```

```
b =  a.add( )
```

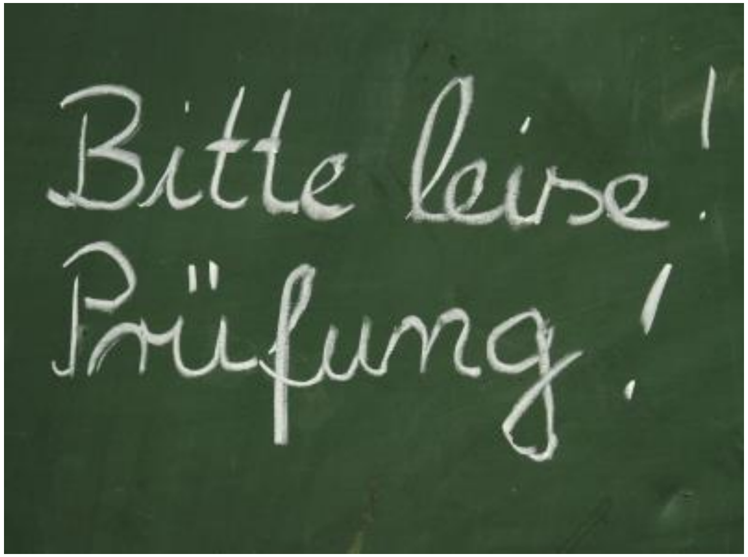
- **Operations do not modify a structure in-place but yield a new updated structure**
- **Structural sharing (path copying) makes "copies" cheap**
- **Safe to share among threads and iteration-safe**

Summary

- **Immutable Objects**
  - are much easier to work with than mutable objects
  - can only be in one state and so are always consistent,
    they are inherently thread-safe, and they can be shared freely
  - No cloning problems (cycles, alias references)

Pro Memoria

- **Mid-Term Exam**
  - Date: Tuesday, 13.11.2018
  - Time: 09:45 - 11:15
    - Part 1: 30Min closed book
    - Part 2: 60Min summary 2 pages A4
  - Location: 3.-111 (Aula)
  - Topics:
    - OOP / Collections
    - Observer
    - State / Strategy
    - Composite
    - Prototype



Pro Memoria

- **Mid-Term Exam**
  - Date: Tuesday, 13.11.2018
  - Time: 11:30 - 13:00
    - Part 1: 30Min closed book
    - Part 2: 60Min summary 2 pages A4
  - Location: 3.-111 (Aula)

  - Topics:
    - OOP / Collections
    - Observer
    - State / Strategy
    - Composite
    - Prototype