# Memory Model and Interfaces

Lecture 2 (2 February 2021)

Radboud University

# OOP vs IP

- The course Imperative Programming (CS, AI) introduced fundamental programming techniques for problem solving (developing algorithms) using variables, loops, methods/functions, arrays…
- The course Object Oriented Programming lays a foundation for object-oriented programming.
  - Classes provide more flexibility and modularity for building reusable software.
  - A problem/software system is modeled as collection of cooperating objects.
  - Each object represents an entity from the problem world.
  - Modeling = abstraction
    - Ignore irrelevant details
    - Abstraction is a mental process.

# OOP vs IP (2)

- E.g. A compiler does not know that `Doats` do not exist.

```java
public class Doat {
   public String bark() {
       return "woof, woof";
   }
   public String meow() {
       return "meow, meow";
   }
}
```

- Only a person can determine whether an abstraction makes sense.

# Abstraction

- question: what are the three most important concepts in programming?
- answer: abstraction, abstraction, abstraction!

# Enumeration types

- Type = Collection, Set
- Enumeration type = (finite) collection of constants.
- Example: `enum Day { Mo, Tu, We, Th, Fr, Sa, Su };`

-
```
Day d = Day.We;
System.out.println( d );
     -----→ We
```

- Comparing
```
Day d1 = Day.We, d2 = Day.Mo;
```

- not:
```
if ( d1 < d2 )
  …
```

- but:
```
if ( d1.compareTo( d2 ) < 0 )
```

# Enum types: example (1)

```
public class Card
{
  public enum Face { Ace, Deuce, Three, Four, Five, Six,
    Seven, Eight, Nine, Ten, Jack, Queen, King  };
  public enum Suit { Clubs, Diamonds, Hearts, Spades };

  private Face cardFace;
  private Suit cardSuit;

  public Card( Face cardFace, Suit cardSuit ) {
     this.cardFace = cardFace; // initialize face of card
     this.cardSuit = cardSuit; // initialize suit of card
  }
   < .... >
}
```

**enum** declared inside Card class

two (private) attributes

constructor

# Enum types: example (2)

```
class Card
{
  < .... >
  @Override
  public String toString() {
    String suits = "cdhs";
    String faces = "a23456789tjqk";
    return suits.charAt( cardSuit.ordinal() ) + "" + faces.charAt( cardFace.ordinal() );
  }
}
```

`ordinal` returns the position in its enum declaration

# Enum types: example (3)

```
public class DeckOfCards
{
  private Card[] deck;

  public static final int DECK_SIZE = 52;

  public DeckOfCards() {
      deck = new Card[ DECK_SIZE ];
      int currentCard = 0;
      for ( Card.Suit suit : Card.Suit.values() ) {
         for ( Card.Face face : Card.Face.values() ) {
            deck[ currentCard ] = new Card( face, suit );
            currentCard++;
         }
      }
  }
 < .... >
}
```

`values`: a static method returning the enum constants as an array.

# More about Enum types (1)

- Enum types are classes:
  - Can have methods and attributes
  - But no other instances (than the constants)

```
enum Direction
{ N, E, S, W;
  public Direction turnLeft() {
    switch ( this ) {
      case N: return W;
      case E: return N;
      case S: return E;
      case W: return S;
    }
  }
}
```

Produces a compiler error:
Missing return statement

# More about Enum types (2)

- Enum types are classes:
  - Can have methods and attributes
  - But no other instances (than the constants)

```
enum Direction
{ N, E, S, W;
  public Direction turnLeft() {
    switch ( this ) {
      case N:  return W;
      case E:  return N;
      case S:  return E;
      default: return S;
    }
  }
}
```

Hint: use default i.s.o. case for the last alternative

# More about Enum types (3)

## Example

```
Direction dir = Direction.N;
System.out.println( dir );
dir = dir.turnLeft();
System.out.println( dir );
```

Output:

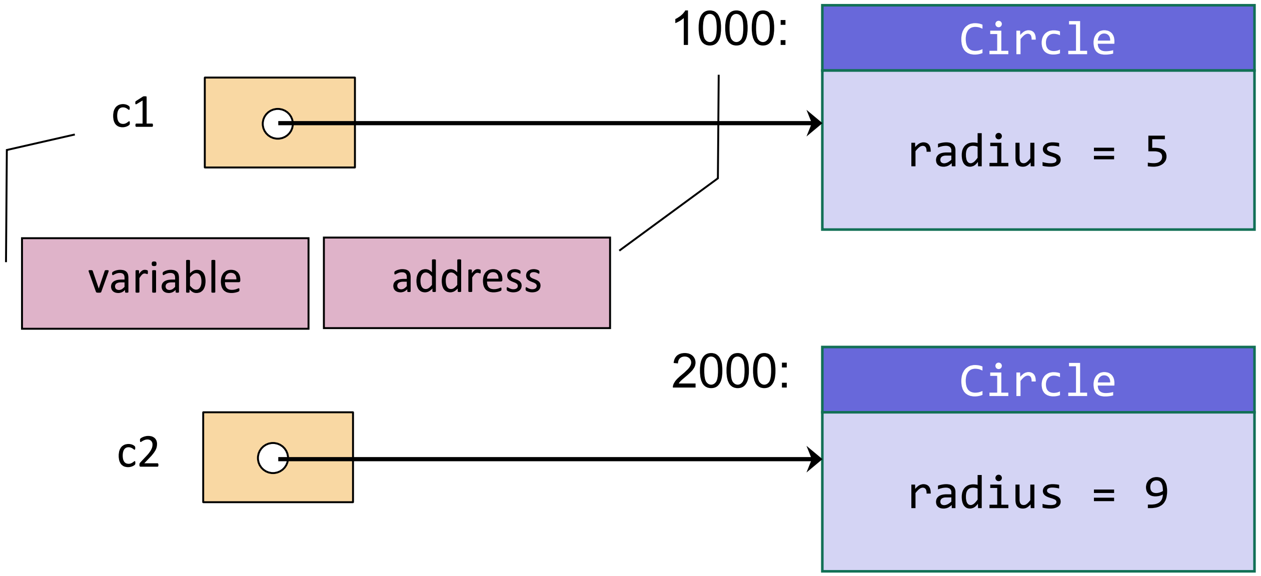
```
N
W
```

# Reference Types

- To access/manipulate an object, assign the (newly created) object to a reference variable.
- Variables (attributes, local variables or parameters) of a class type hold references (pointing to objects)
- The object named by a variable is stored in some location in memory.
- This location is also called the address of the object.
- One can access the data (properties) of an object via its address
- In fact, the value of a variable is an address.
- You will hardly notice this.
    - it is fine to say that a variable `myCircle` is a `Circle` object rather than say that `myCircle` is a variable that contains a reference to a `Circle` object.

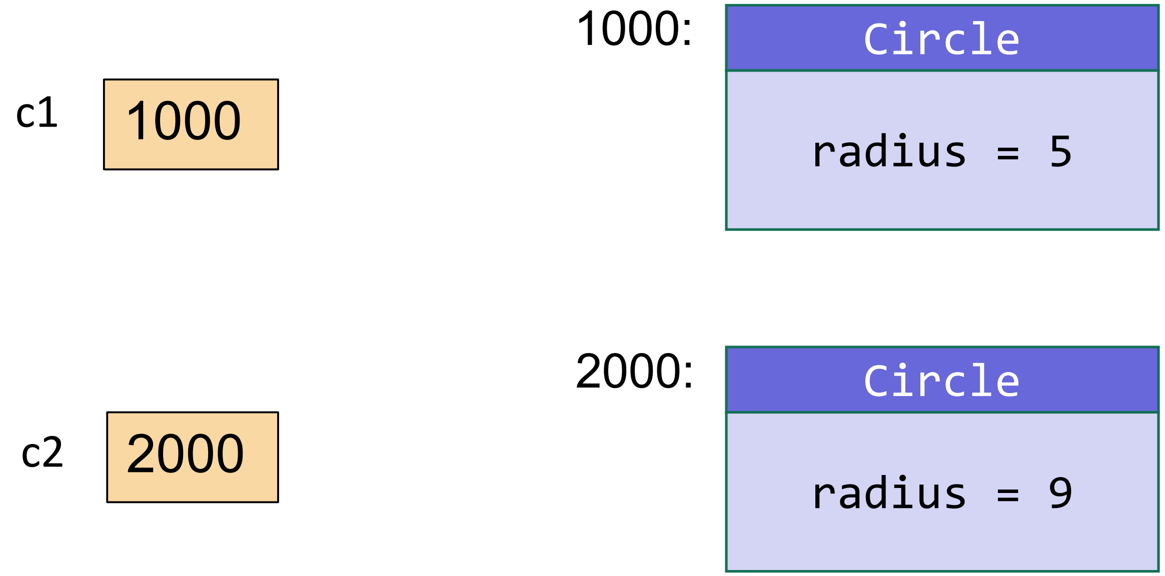
# references and addresses

Variable refers to an Object



# references and addresses

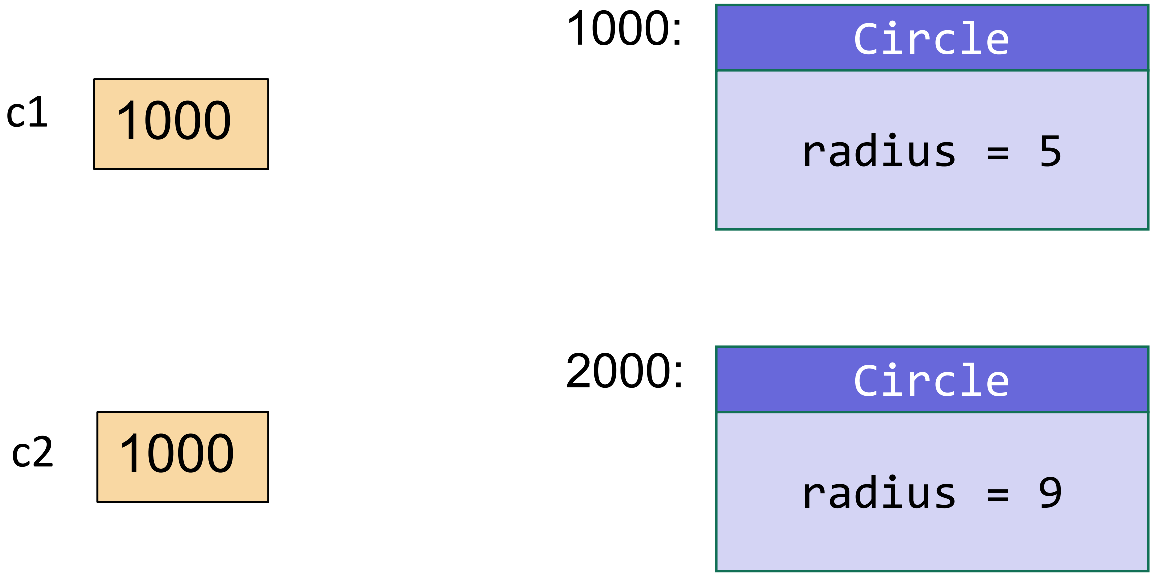
Variable contains an address



# Memory management: Aliasing

Different names for the same object, for example after the assignment

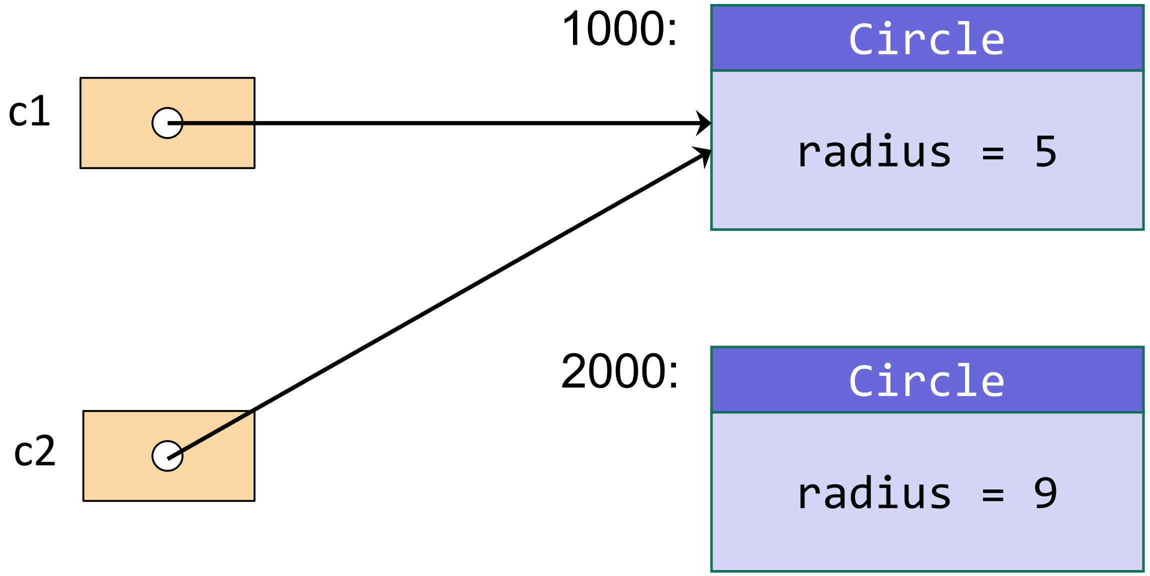
```
c2 = c1
```



# Memory management: Aliasing

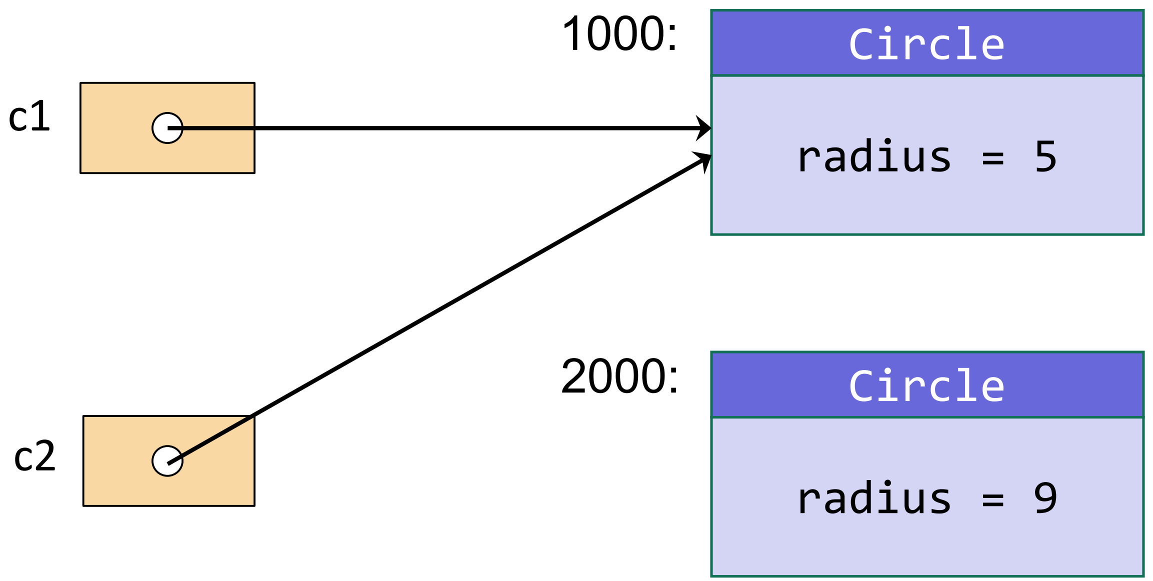
Different names for the same object, for example after the assignment

```
c2 = c1
```



# Memory management: Garbage collection

reclaiming space of unused objects



# Null pointers

- null-pointer refers to no object at all.
- Accessing an object (*dereferencing*) via a null pointer yields a `NullPointerException`

# Example

```
public class Clock
{
   private int min, hou;

   public Clock( int hou, int min ) {
        setTime( hou, min );
   }

   public void setTime( int hou, int min ) {
        this.hou = hou;
        this.min = min;
   }

   public String toString() {
      return String.format( "%2d:%2d", hou, min );
   }
};
```

format specifier: see IJPDS, 168-170

# Primitive vs. Reference Types

```
public static void mainPrim() {
   int i = 10;
   int j = 20;

   System.out.println( i );
   System.out.println( j );

   j = i;
   i = 50;

   System.out.println( i );
   System.out.println( j );

}
```

```
public static void mainRef() {
   Clock c1 = new Clock( 12, 10 );
   Clock c2 = new Clock( 11, 45 );

   System.out.println( c1 );
   System.out.println( c2 );

   c2 = c1;
   c1.setTime( 21, 30 );

   System.out.println( c1 );
   System.out.println( c2 );
}
```

Output:

```
10
20
50
10
```

Output:

```
12:10
11:45
21:30
21:30
```

# Comparing reference types

```
Clock c1 = new Clock( 10, 12 );
Clock c2 = new Clock( 10, 12 );

System.out.println( c1 == c2 );
```

false

Pointer comparison

# Comparing reference types

```
Clock c1 = new Clock( 10, 12 );
Clock c2 = c1;

System.out.println ( c1 == c2 );
```

true

# Comparing reference types

- Define your own equality method.

```
public class Clock
{
    private int min, hou;
    …
    public boolean equals( Clock c ){
        return min == c.min && hou == c.hou;
    }

};
```

Beware: this is not the recommended  way to define equals; see lecture 4

```
        Clock c1 = new Clock( 10, 12 );
        Clock c2 = new Clock( 10, 12 );


        System.out.println( c1.equals( c2 ) );
```

true

Radboud University

# **this** Object

- 'standard' (non-static) methodes have access to (all) attributes of an object.
- To invoke a method (say *m*) you need an object. This object is passed implicitly to *m*. You can access this hidden parameter via **this**.
- For example, in the class Clock:

```
public void setTime( int hou, int min )
```

  could be simulated by

```
public static void staticSetTime( Clock this, int hou, int min )
```

  then

```
c1.setTime( 21, 30 );
```

  becomes

```
staticSetTime( c1, 21, 30 );
```

- Inside a class, methods can call each other. The calling object is passed implicitly.

# Static methods (services)

- Don’t have an implicit (**`this`**) argument
- Can’t manipulate ('normal') attributes or call other non-static methods.
- Use the name of the class to call them from another class: `A.sm(...)`
- Most familiar example:

```
public static void main( String [] args ){
    …
}
```

# Static attributes

- Shared by all instances (objects) of that class.
- Can be accessed from static methods.
- Are rarely used.

# Importance of encapsulation

- Encapsulation: bundling data and methods that work on that data within a class.
  - keep the internal representation of an object hidden from the outside.
- Encapsulation aim: minimizing the interdependence between modules.
  - separation of class implementation from the use of a class;
  - decouples the modules of a system, allowing them to be developed, tested, optimized, used, understood, and modified in isolation

```
public class Day
{
    public int year;
    public int month;
    public int day;
}
```

E.g.:
2021, 2, 2 is
2 February 2021

```
public class Day
{
    public int julian;
}
```

Number of days since
Jan. 1, 4713 BCE

E.g:
2.459.248 is
2 February 2021

# Encapsulation

- Suppose we change to the julian format. What to do with:

```
m = d.month;
```

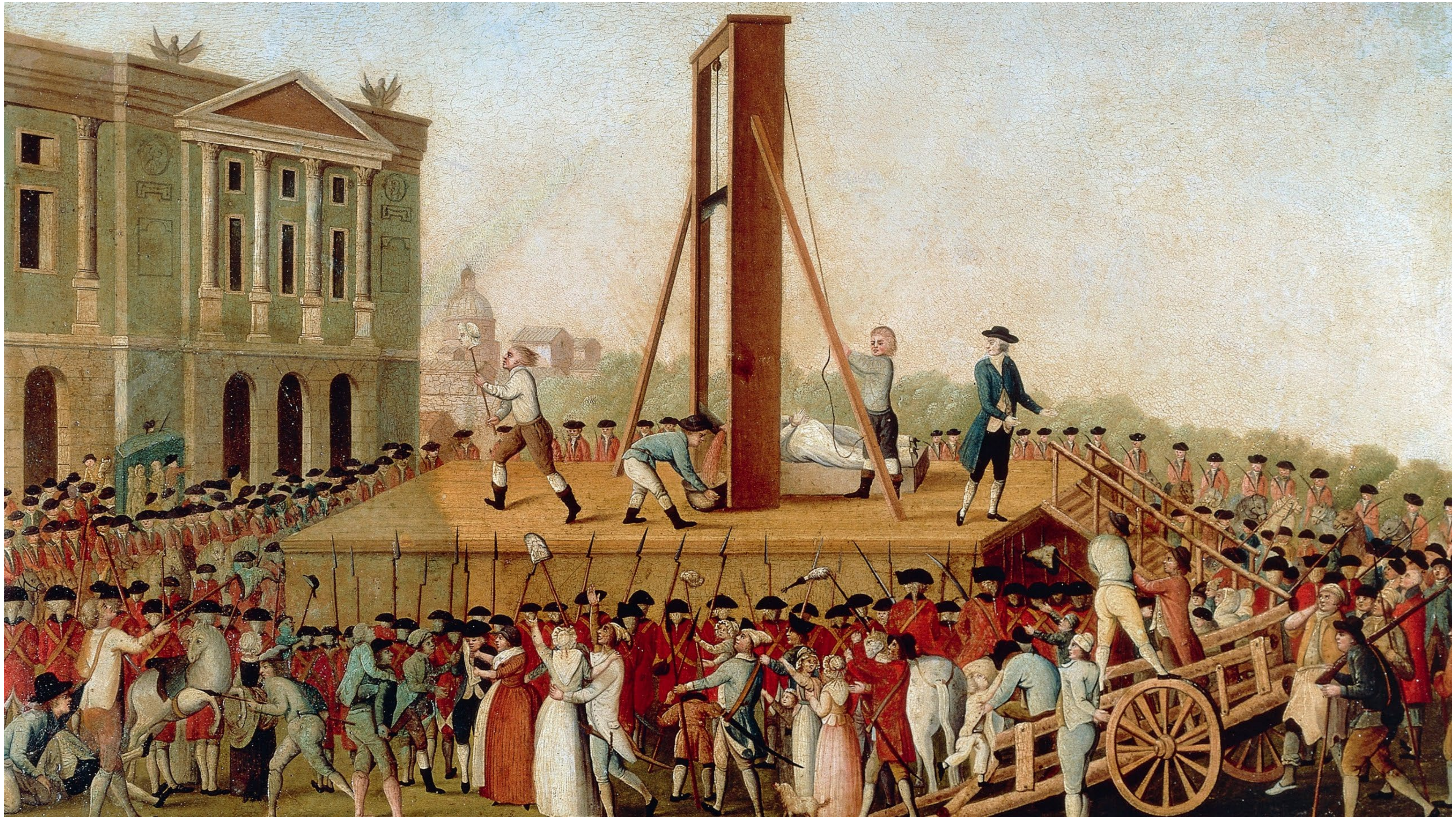
… and with:

```
d.year++;
```

# Encapsulation (2)

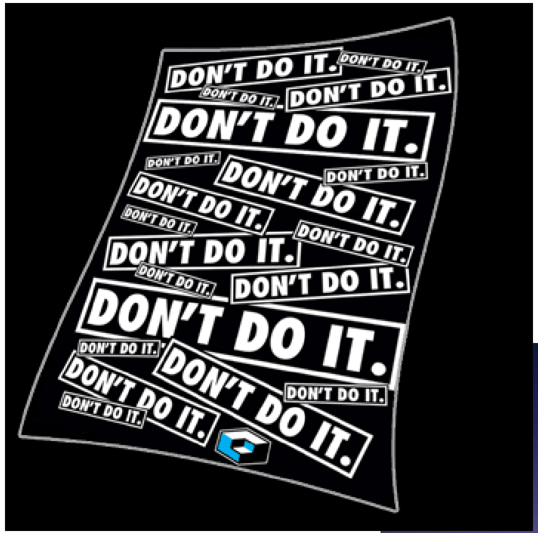
- Don't use public attributes.
- exposes instance’s state to other classes and permits direct modification of state
- completely breaks idea of encapsulation; if you need to provide access to private instance variables, use accessors (and mutators)
- increases your life expectancy …

# if you are caught …



# More bad/dangerous things …

- Attributes used as local variables.
- Attributes instead of parameters or for returning results.
- Asking for trouble…



# Be careful with mutators!

- Mutator: method that changes the object state (= the attribute values). Also called command.
  - Usually, mutators do not return a value (result type = **`void`**)
- Accessor: method that reads object state without changing it. Sometimes called query.
  - Return something (result type ≠ **`void`**)
- Suppose our Day class has a `setMonth(` **`int`** `month )` method

```
Day deadline = new Day( 2021, 10, 31 );
todoList.add (deadline);
     :
     :
deadline.setMonth(11);
```

- What’s the result?

# Making the setter more robust

```
public class Day
{
   private int year, month, day;

   public void setMonth( int month ) {
      if ( month == 2 && this.day > 28 ||
           month == 4 && this.day > 30 ||
          ...) {
         throw wrongDateException;
      } else {
         this.month = month;
      }
   }

}
```

# Better mutators???

- The program was not yet complete

```
Day deadline = new Day( 2021, 10, 31 );
todoList.add ( deadline );
    :
    :
deadline.setMonth( 11 );
deadline.setDay( 30 );
```

- Set the day first, then the month

```
Day deadline = new Day( 2021, 11, 30 );
todoList.add ( deadline );
    :
    :
deadline.setDay( 31 );
deadline.setMonth( 10 );
```

# Disadvantages of mutators

- Set-methods can be dangerous
- Don't supply a mutator for every attribute
- Use a single setter for attributes that are mutually dependent (values of these attributes are related to each other)

# Joke of the week

Three guys stranded on a desert island find a magic lantern containing a genie, who grants them each one wish. The first guy wishes he was off the island and back home. The second guy wishes the same. The third guy says: ‘I’m lonely. I wish my friends were back here.’

Radboud University

# Interfaces

# Introduction to interfaces

- In General, an interface is the place at which independent and often unrelated systems meet and act on or communicate with each other [*Merriam-Webster*]
  - A language is an interface between two people.
  - A remote control is an interface between you and a television.
- In Computing, an interface is a shared boundary across which two or more components of a computer system exchange information.
- In object oriented programming, an interface is a common means for unrelated objects to communicate with each other.

# Introduction to Interfaces

- Goal: minimizing dependencies.
- Interface separates the *concern* of the implementation (server) from the *concern* of the user (client)
- Allows client and server to be developed, implemented, tested, optimized, used, understood, and modified independently.
- Interface specifies a contract:
  - *maximum* functionality a *client* can use
  - *minimum* functionality a *server* has to provide

# Java interfaces

- A Java **`interface`** contains:
  - method specifications (no implementation), called abstract methods
  - constant definitions.
  - In Java 8, also static and default methods.
- A Java interface does not contain:
  - constructors,
  - method bodies,
  - instance variables.

# Example: Java Icon interface

https://docs.oracle.com/javase/8/docs/api/javax/swing/Icon.html

```
public interface Icon {
   int getIconWidth();
   int getIconHeight();
   void paintIcon( Component c, Graphics g, int x, int y );
}
```

Position on screen

# Example: an Icon client

https://docs.oracle.com/javase/8/docs/api/javax/swing/JOptionPane.html

```
public static void showMessageDialog( Component parentComponent,
                          Object message,
                          String title,
                          int messageType,
                          Icon icon )
```

To create a message dialog

uses (is a client of) the Icon interface

- showMessageDialog was programmed independently from any implementation of Icon

# Java interface implementation

- A class implements an interface by
  - naming interface in an implements clause in class heading, and
  - including definitions for **all** methods in interface.

# Example: a predefined Icon implementation

https://docs.oracle.com/javase/8/docs/api/javax/swing/ImageIcon.html

```
public class ImageIcon implements Icon
{
   <...>

   public ImageIcon( String filename )

   <...>
}
```

one of the constructors

Part of the Java API

# Using ImageIcons

```
public class IconTester {

    public static void main( String[] args ) {
       JOptionPane.showMessageDialog(
          null,
          "Hello, World!",
          "Greeting",
          JOptionPane.INFORMATION_MESSAGE,
          new ImageIcon( "images/globe.gif" ) );
    }
```

Creates an ImageIcon from the specified file.

# Example: my own Icon implementation

```
public class MarsIcon implements Icon
{
   private int size;

   /**
      Constructs a Mars icon of a given size.
      @param size the size of the icon
   */
   public MarsIcon( int size ) {
      this.size = size;
   }

   public int getIconWidth() {
      return size;
   }
    <...>
}
```

# Example: my own Icon implementation (2)

```
public class MarsIcon implements Icon
{
   <...>

   public int getIconHeight() {
      return size;
   }

   public void paintIcon( Component c, Graphics g, int x, int y ){
      Graphics2D g2 = (Graphics2D) g;
      g2.setPaint( new GradientPaint ( x, y, Color.YELLOW, x+size, y+size, Color.RED ) );
      Ellipse2D.Double planet = new Ellipse2D.Double( x, y, size-1, size-1);
      g2.fill(planet);
   }

}
```

# Example: Shopping Cart of a Web shop

- Two classes ripe for the picking: cart class, item class

```java
public class Item {
    private final String name;
    private final double price;

    public Item( String name, double price ) {
        this.name = name;
        this.price = price;
    }

    public String getName() { return name; }

    public double getPrice() { return price; }

    @Override
    public String toString() {
        return String.format( "%s %.2f euro", name, price );
    }

}
```

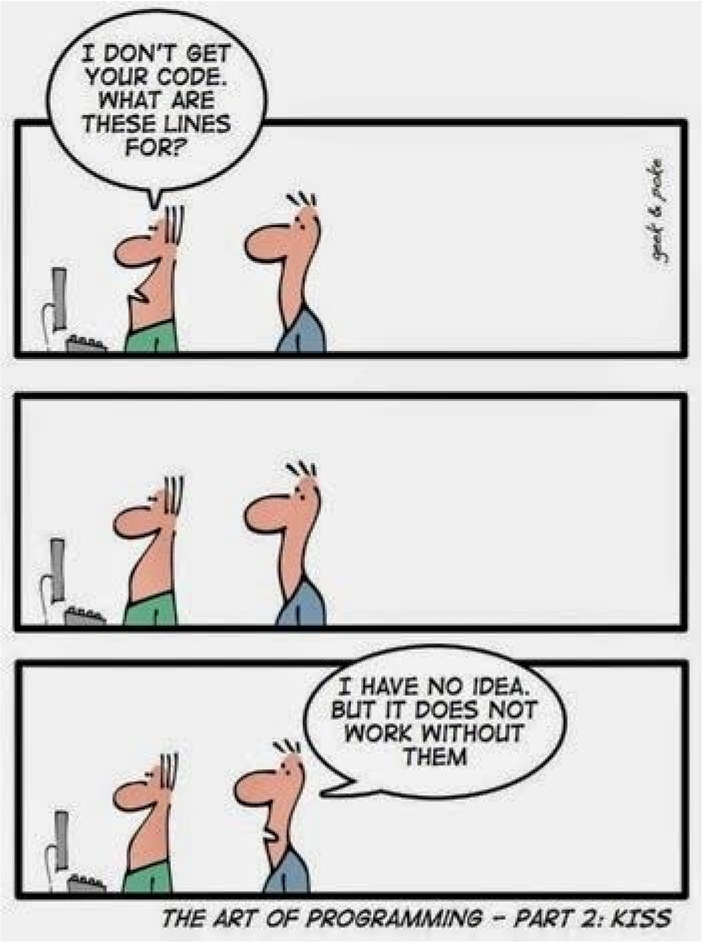
# Example: Shopping Cart of a Web shop

- Testing our implementation

```
public class OrderDemo {

    public static void main( String [] args ) {
        ShoppingCart order = new ShoppingCart();
        order.add( new Item( "Milk", 0.95 ) );
        order.add( new Item( "Shampoo", 0.87 ) );

        System.out.println(order);

        order.pay( new CreditCard( "Sjaak", "12/21", 10203040, 567 ) );
    }
}
```

# Example: Shopping Cart of a Web shop (2)

```
public class ShoppingCart {
    private Item[] items;
    private int nrOfItems;
    private static final int MAX_NR_OF_ITEMS = 10;

    public ShoppingCart( ) {
        items = new Item [MAX_NR_OF_ITEMS];
        nrOfItems = 0;
    }

    public void add( Item item ) {
        items[nrOfItems++] = item;
    }
    private double total() {
        double total = 0;
        for ( int i = 0; i < nrOfItems; i++ ) {
            total = total + items[i].getPrice();
        }
        return total;
    }
    public void pay( CreditCard cc ) {
        cc.pay(  total() );
    }
}
```


I DON'T GET YOUR CODE. WHAT ARE THESE LINES FOR?
geek & poke
I HAVE NO IDEA. BUT IT DOES NOT WORK WITHOUT THEM

THE ART OF PROGRAMMING - PART 2: KISS

# Example: Shopping Cart of a Web shop (3)

```java
public class CreditCard {
    private final String name, date;
    private final int number, cvv;

    public CreditCard( String name, String date, int number, int cvv ) {
        this.name   = name;
        this.date   = date;
        this.number = number;
        this.cvv    = cvv;
    }

    public void pay( double amount ) {
        System.out.format( "Paid %1.2f with card %d of %s\n", amount, number, name );
    }

}
```

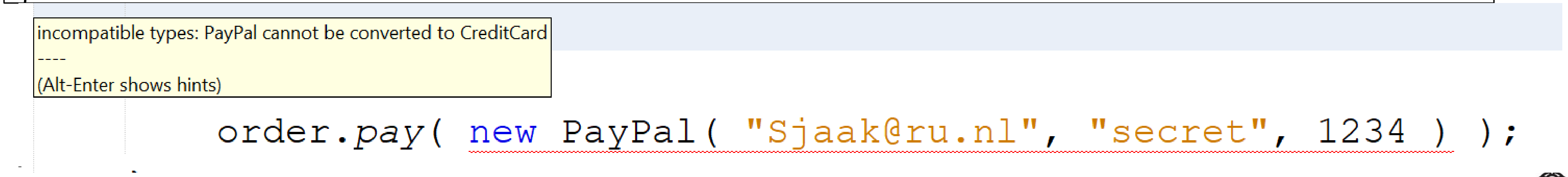
# Shopping Cart: more payment methods

```java
public class PayPal {
    private final String email, password;
    private final int code;

    public PayPal( String email, String password, int code ) {
        this.email = email;
        this.password = password;
        this.code = code;
    }

    public void pay( double amount ) {
        System.out.format( "Pay %1.2f with paypal for %s\n", amount, email );
    }

    @Override
    public String toString() {
        return "PayPalPayment of " + email;
    }
}
```

# Shopping Cart: more payment methods

- Testing our implementation

```
public class OrderDemo {

    public static void main( String [] args ) {
        ShoppingCart order = new ShoppingCart();
        order.add( new Item( "Milk", 0.95 ) );
        order.add( new Item( "Shampoo", 0.87 ) );

        System.out.println(order);

        order.pay( new PayPal( "Sjaak@ru.nl", "•••••", 1234 ) );
    }
}
```



```
incompatible types: PayPal cannot be converted to CreditCard
----
(Alt-Enter shows hints)
```

```
        order.pay( new PayPal( "Sjaak@ru.nl", "secret", 1234 ) );
```

# Shopping Cart: more payment methods (2)

- Solution: adding new payment method to class `ShoppingCart`?
- Bad idea.
  - existing and working implementation has to be adjusted: breaks the idea of independent development.
  - clients of payment methods should not be concerned with actual implementations
- Solution: use an interface!

# Payment interface

```
public interface Payment {
    void pay( double amount );
}
```

- Adjustments

```
public class CreditCard implements Payment {
    // nothing changes here
}

public class PayPal implements Payment {
    // nothing changes here
}

public class ShoppingCart {
    <...>
    public void pay( Payment pm ) { pm.pay(  total() ); }
}
```

# Finally

questions?