List of Slides

Title

Java Just in Time

John Latham

November 13, 2018

Chapter 11

Object oriented design

Chapter aims

- Take second look at OO technology introduced in previous chapter
  - but see how we approach program **design** with OO from the start.
- First revisit `AgeHistory2`
  - add extra feature of person name as well as birthday
  - have **textual user interface** rather than **command line argument**s.
- Then a model of greedy children eating ice cream
  - e.g. basis of simple computer game
  - has **mutable object**s
  - **accessor method**s and **mutator method**s.

Section 2

Example:
Age history revisited

Aim

AIM: To introduce the principles of **object oriented design**. We also meet `Scanner`, **standard input**, Java's **package** structure and **import statement**, the **null reference**, **final variable**s, multiple **return statement**s, the **line separator system property**, and take a look at making **stub**s of **class**es and using **multi-line comment**s.

Age history revisited

- Previous chapter used `AgeHistory2` to introduce and motivate Java **object oriented programming**
  - would not really write first without OO and then convert!
- Here look at **object oriented design**
  - how approach program development using OO from beginning.

Design: object oriented design

- Developing in **object oriented programming** language requires use of **object oriented design**.
- Start by identifying **class**es
  - examine **requirements statement**
  - problems inherently involve interactions between 'real world' objects
  - modelled in our program – by creating **object**s
    - ∗ **instance**s of the classes we identify.

Design: object oriented design

- An object is entity with some
  - **object state** – maybe changes over time
  - **object behaviour** – probably based on its state.
- Think about state and behaviour of objects in problem.
- Decide how to model
  - behaviour via **instance methods**
  - state via **instance variables**.
- May need **class variables** and **class methods** too.

Age history revisited

Coffee time: Why do you think the pieces of code making up a Java program are called **class**es?

A program is required that will print out, on the **standard output**, the age history of any number of people. Each person has a name and a birth date. The age history of a person consists of a statement of their birth on their birth date, followed by a statement of their age on each of their birthdays which have occurred *before* the present date. Finally it ends with a statement saying what age they will be on their next birthday, including the present date, if their birthday is today. However, if the person has not yet been born, or is born on the present date then their age history consists merely of a statement stating or predicting their birth.

Age history revisited

The program shall be **interactive** with a **textual user interface**. It shall prompt for the present date, to be entered by the user as three **integer**s in the order day, month then year. Then it shall prompt for the number of persons, which is to be entered as an integer. Then, for each person, it shall prompt for his or her name, to be entered as a string, and date of birth, to be entered as three integers in the order day, month then year. Then it shall produce the age history for that person.

The program is allowed to assume that the number of persons and components of dates are entered as strings representing legal integers. If the entered number of persons is **less than** one, the program will quietly do nothing more.

Design: object oriented design: noun identification

- Analyse **requirements statement** – decide what **class**es to have.
- One way: **noun identification**
  - list all nouns and noun phrases
  - objects inherent in solution usually appear as nouns in problem description.
    - Some nouns will be **object**s at **run time**
    - some will be classes.
  - Not all nouns found will be a class or object
    - also sometimes need classes not appearing as nouns in requirements.
  - But generally good starting point.

Identifying the classes

Noun	**Usage in requirements**	**Class, object or how/what?**
age	A number	**`int`**
age history	An effect on the output	A `String` with many lines
birth	An event to be reported	Part of age history
birth date	A date belonging to a person	An object, **instance variable** of a person
birthday	An event to be reported	Part of age history
component of date	Strings entered by the user	Become values of instance variables of `Date`
date of birth	Same as birth date	-

Identifying the classes

Noun	Usage in requirements	Class, object or how/what?
date	Used for present date, birth dates and birthdays	A class
day	Part of a date, a number	Instance variable in date objects
integer	Standard stuff	`int`
month	Part of a date, a number	Instance variable in date objects
name	A string belonging to a person	Instance variable of a person
number	Standard stuff	`int`

Identifying the classes

Noun	Usage in requirements	Class, object or how/what?
person	Many people inherent in problem	A class
present date	A date	An object
program	Standard stuff	A class to contain the main method
standard output	Standard stuff	Via `System.out.println()`
statement	An effect on the output	Via `System.out.println()`
string	Standard stuff	`String`

Identifying the classes

Noun	**Usage in requirements**	**Class, object or how/what?**
textual user interface	User interaction with program	Via **standard input**
today	Same as present date	-
user	The real person using the program	Via standard input and **standard output**
year	Part of a date, a number	Instance variable in date objects

Identifying the classes

- Three classes:

Class list for AgeHistory

Class	**Description**
`AgeHistory`	The main class containing the **main method**. It will interact with the user and make instances of `Date` and `Person`.
`Date`	An instance of this will represent a date.
`Person`	An instance of this will represent a person.

Designing the class interfaces

- The **main method** creates a `Date` for present date
  - where stored?
  - A prevalent **object oriented programming design** principle:
    - **putting the logic where the data is.**
  - So store (reference to) present date in **class variable** in `Date` class.
- Main method then creates `Person` object for each person
  - including a `Date` object for person's birth date.
- For each person obtain age history and print it out
  - have **instance method** in `Person` class
    - **return**s age history as `String`
  - need to access present date from `Date` class.

Designing the class interfaces

Public method interfaces for class `AgeHistory`.

Method	Return	Arguments	Description
`main`		`String[]`	The main method for the program.

Designing the class interfaces

Public method interfaces for class `Date`.

Method	Return	Arguments	Description
`setPresentDate`		`Date`	A **class method**: sets the present date to be the one given. This is ignored if the present date has already been been set.
`getPresentDate`	`Date`		A class method: returns the present date as set by `setPresentDate()`.
Constructor		**`int, int, int`**	Constructs a date representing the given day, month and then year values.

Designing the class interfaces

Public method interfaces for class `Date`.

Method	**Return**	**Arguments**	**Description**
`toString`	`String`		Returns the day/month/year representation of the date.
`equals`	**`boolean`**	`Date`	Returns **`true`** if and only if this object represents the same date as the given other date.
`lessThan`	**`boolean`**	`Date`	Returns **`true`** if and only if this object represents a date earlier than that represented by the given other date.
`addYear`	`Date`		Returns a new date, one year on from this one.

Designing the class interfaces

Public method interfaces for class `Person`.

Method	**Return**	**Arguments**	**Description**
Constructor		`String, Date`	Constructs a person with the given name and birth date.
`ageHistory`	`String`		Returns the age history of this person as a string with **new lines** in it.

- Check design for correct **encapsulation**.

Design: object oriented design: encapsulation

- A principle of **object oriented design** – **encapsulation**
  - in order to use a class, need only know about **public methods** (inc **constructor methods**)
    - what they mean
    - not how they work
    - not what **instance variables** there are.
- Design principle – **putting the logic where the data is**
  - all code about objects behaviour appears in their class
    - not sprinkled around the program.

Design: object oriented design: encapsulation

- Encapsulation is form of **abstraction**
  - abstraction: ignore unnecessary detail.
    - Use a class without knowing how it works
    - design details of one class
      without being concerned with details of other classes.

The `AgeHistory` class

- Can make **new instance**s of our own **class**es.
- Also can make instances of many **API** classes
  - e.g. `Scanner`.

Operating environment: standard input

- As well as **standard output**, programs have **standard input**
  - allows text **data** to be entered into program as it runs.
- In **command line interface** input is typically typed on keyboard.

Standard API: `System`: `in`

- `System` **class** has **class variable** `out`
  - E.g. `System.out.println()`
- Also `in`
  - contains **reference** to an **object** representing **standard input**.
- Java standard input not easy to use
  - typically access via something else
  - e.g. `Scanner`.

Package

- Hundreds of **class**es in Java **API**
  - even more ‘around the world’.
- Grouped into collections of related classes: **package**s.
- Packages also grouped – hierarchy.
- E.g. package groups `java` and `javax`.

Package: java.util

- Package group java has package util (and many others)
  - full name is java.util – dot used as path item separator.
- java.util contains many utility **class**es
  - e.g. Scanner
- Unique **fully qualified name**
  - e.g. java.util.Scanner
    - Scanner in util package in java package group.
- Can refer to a class via fully qualified name
  - e.g.

```
java.util.Scanner inputScanner = new java.util.Scanner(System.in);
```

Class: importing classes

- At start of source **file** can have **import statements**
  - **reserved word `import`**
    followed by **fully qualified name** of class
    then semi-colon(;).
- Imported classes can be referred to just by class name – don't have to keep using fully qualified name.
  - E.g.

```
import java.util.Scanner;
  ...
  Scanner inputScanner = new Scanner(System.in);
```

Class: importing classes

- Can import all classes in a package using *
  - e.g.

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

- Considered lazy – better to import exactly what is needed
  - helps show precisely what is used by importing class.
- Also ambiguity issue:
  - two different packages may have classes with same name. . . .

Class: importing classes

- But, every Java program has automatic import for all classes in **package** `java.lang`
  - e.g. `System` is really `java.lang.System`
  - e.g. `Integer` is really `java.lang.Integer`
  - etc..
- I.e., all classes implicitly have

```
    import java.lang.*;
```

Standard API: Scanner

- Since Java 5.0 – `java.util.Scanner`: simple features to read input **data**.
- Can pass `System.in` to **constructor method**:

```
import java.util.Scanner;
  ...
  Scanner inputScanner = new Scanner(System.in);
  ...
```

- Want line of text, or read an **integer**:

```
  String line = inputScanner.nextLine();
  ...
  int aNumber = inputScanner.nextInt();
  // Skip past anything on the same line following the number.
  inputScanner.nextLine();
  ...
```

Standard API: Scanner

- `System.in` gets **bytes** from **standard input**.
- `Scanner` turns bytes into **characters** (**`char`**)
  - has variety of instance methods to scan characters into lines / tokens
    - separated by **white space**: e.g. space, tab, end of line.

Standard API: Scanner

Public method interfaces for class `Scanner` (some of them).

Method	**Return**	**Arguments**	**Description**
`nextLine`	`String`		Returns all the text from the current point in the character stream up to the next end of line, as a `String`.
`nextInt`	**`int`**		Skips any spaces, tabs and end of lines and then reads characters which represent an integer, and **return**s that value as an **`int`**. It does not skip spaces, tabs or end of lines following those characters. The characters must represent an integer, or a **run time error** will occur.

Standard API: Scanner

Public method interfaces for class Scanner (some of them).

Method	Return	Arguments	Description
`nextBoolean`	**`boolean`**		Similar to `nextInt()` except for a **`boolean`** value.
`nextByte`	**`byte`**		Similar to `nextInt()` except for a **`byte`** value.
`nextDouble`	**`double`**		Similar to `nextInt()` except for a **`double`** value.
`nextFloat`	**`float`**		Similar to `nextInt()` except for a **`float`** value.
`nextLong`	**`long`**		Similar to `nextInt()` except for a **`long`** value.

Standard API: Scanner

Public method interfaces for class `Scanner` (some of them).

Method	Return	Arguments	Description
`nextShort`	**`short`**		Similar to `nextInt()` except for a **`short`** value.

- Can also change what is used to separate tokens.

The AgeHistory class

```java
import java.util.Scanner;
```

- Next comes comment
  - copy and edit some text from **requirements statement**. . .

Code clarity: comments: multi-line comments

- Java permits multi-line **comments**
  - start with /*
  - end with */
  - these symbols and all text between is ignored by **compiler**.
- Can have such a comment on one line, with code either side
  - not often useful, especially with 80 chararacter line limit.

The AgeHistory class

Coffee time: One use of multi-line comments is to `comment out’ a section of code during development, perhaps because it is not completed yet. Do you think we can nest multi-line comments in Java, that is, have such a comment inside another one? Can we have single line comments inside a multi-line comment?

The AgeHistory class

```
/* Program to print out the history of any number of named people’s ages.

   The age history of a person consists of a statement of their birth on their
   birth date, followed by a statement of their age on each of their birthdays
   which have occurred before the present date. Finally it ends with a
   statement saying what age they will be on their next birthday, including
   the present date, if their birthday is today. However, if the person has
   not yet been born, or is born on the present date then their age history
   consists merely of a statement stating or predicting their birth.

```

The AgeHistory class

```
   It first prompts for the present date, to be entered by the user as three
   integers in the order day, month then year. Then it prompts for the number
   of persons, which is to be entered as an integer. Then, for each person, it
   prompts for his or her name, to be entered as a string, and date of birth,
   to be entered as three integers in the order day, month then year. Then it
   produces the age history for that person.
 */
public class AgeHistory
{
  public static void main(String[] args)
  {
    // For interaction with the user.
    Scanner inputScanner = new Scanner(System.in);

```

The AgeHistory class

```java
// The Date class needs to be told the present date.
System.out.print("Enter today’s date as three numbers, dd mm yyyy: ");
int day = inputScanner.nextInt();
int month = inputScanner.nextInt();
int year = inputScanner.nextInt();
Date.setPresentDate(new Date(day, month, year));

// Now find out how many people there are.
System.out.print("Enter the number of people: ");
int noOfPeople = inputScanner.nextInt();
// Skip to the next line of input
// or else first name will be blank!
inputScanner.nextLine();

```

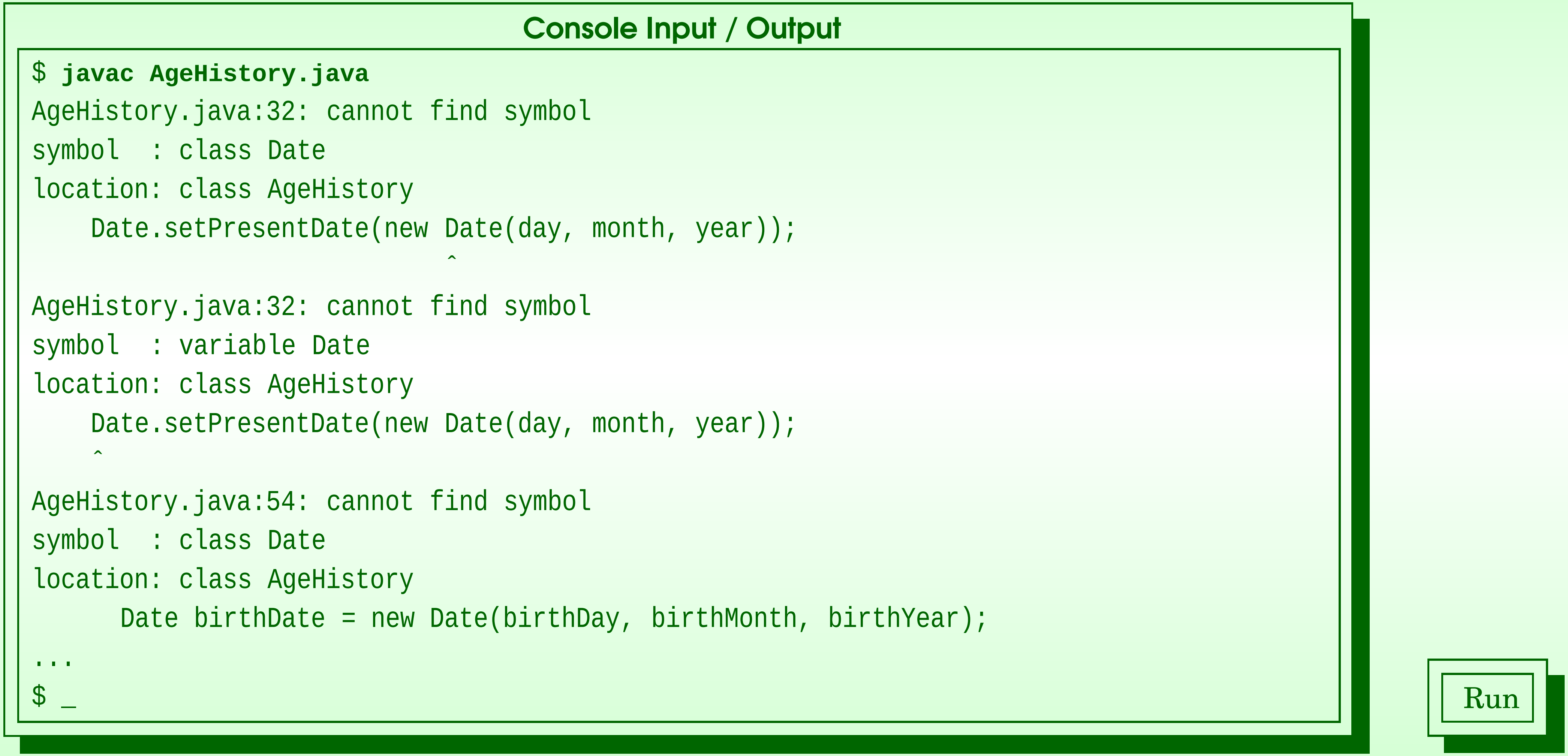
The AgeHistory class

```java
    // For each person...
    for (int personNumber = 1; personNumber <= noOfPeople; personNumber++)
    {
      // Obtain name and birthday.
      System.out.print("Enter the name of person " + personNumber + ": ");
      String personName = inputScanner.nextLine();
      System.out.print("Enter his/her birthday (dd mm yyyy): ");
      int birthDay = inputScanner.nextInt();
      int birthMonth = inputScanner.nextInt();
      int birthYear = inputScanner.nextInt();
      // Skip to next line, or else next name will be blank!
      inputScanner.nextLine();

```

The AgeHistory class

```
      Date birthDate = new Date(birthDay, birthMonth, birthYear);
      Person person = new Person(personName, birthDate);
      System.out.println(person.ageHistory());
    } // for
  } // main

} // class AgeHistory
```

Stubbing Date and Person



```
$ javac AgeHistory.java
AgeHistory.java:32: cannot find symbol
symbol  : class Date
location: class AgeHistory
    Date.setPresentDate(new Date(day, month, year));
                            ^
AgeHistory.java:32: cannot find symbol
symbol  : variable Date
location: class AgeHistory
    Date.setPresentDate(new Date(day, month, year));
    ^
AgeHistory.java:54: cannot find symbol
symbol  : class Date
location: class AgeHistory
      Date birthDate = new Date(birthDay, birthMonth, birthYear);
...
$ _
```

Class: stub

- Often produce **stub**s for classes not yet implemented when developing programs with several **class**es
  - just some/all **public** items
  - empty/almost empty bodies for **method**s.
  - Any **non-void method**s written as single **return statement**
    - yield some temporary value.
  - Bare minimum to allow classes developed so far to **compile**.
- Develop stubs into full class code later.

Person.java

```
public class Person
{
  public Person(String s, Date d) {}
  public String ageHistory() { return "An age history"; }
} // class Person
```

Date.java

```java
public class Date
{
  public Date(int d, int m, int y) {}
  public static void setPresentDate(Date d) {}
} // class Date
```

Stubbing Date and Person

Coffee time: Are you surprised that Java lets us put the body of a **method** on the same line as its heading? That may be fine for stubs which will be thrown away shortly, but do you think it is appropriate for proper code? Ever?

- Can now compile and even **run** the program!
- Can check `AgeHistory` works before developing other classes.

Stubbing Date and Person

Console Input / Output

```
$ javac AgeHistory.java
$ _
$ java AgeHistory
Enter today's date as three numbers, dd mm yyyy: 01 07 2019
Enter the number of people: 0
$ _
$ java AgeHistory
Enter today's date as three numbers, dd mm yyyy: 01 07 2019
Enter the number of people: 2
Enter the name of person 1: John
Enter his/her birthday (dd mm yyyy): 24 4 1959
An age history
Enter the name of person 2: Lizzy
Enter his/her birthday (dd mm yyyy): 1 7 1989
An age history
$ _
```

Run

The `Date` class

- Main difference from previous is **class variable** to store present date
  - plus **class method**s to set and access it.

Variable: of a class type: null reference

- When create **object**
  often store **reference returned** by **constructor method** in a **variable**.

- E.g.

```
Point p1 = new Point(75, 150);
```

- But what if don't want to refer to an object (yet)?
  - Special reference value – the **null reference**
  - is reference, but does not refer to an object.
  - Written using **reserved word `null`**.

- E.g.

```
Point p2 = null;
```

Variable: of a class type: null reference

- We have two `Point` variables – `p1`, `p2`
  - but (at **run time**) only one `Point` object.
- Suppose `Point` has **instance methods** `getX()` and `getY()`.
- Then this is okay:

```
System.out.println(p1.getX());
```

- But next code will cause **run time error** (**exception** called `NullPointerException`):

```
System.out.println(p2.getX());
```

because no object referenced by `p2`, so attempt to follow reference fails.

The Date class

```java
// Representation of a date.
public class Date
{
  // Class variable to hold the present date.
  private static Date presentDate = null;


  // Class method to set the present date.
  // This does nothing if it has already been set.
  public static void setPresentDate(Date requiredPresentDate)
  {
    if (presentDate == null)
      presentDate = requiredPresentDate;
  } // setPresentDate

```

The Date class

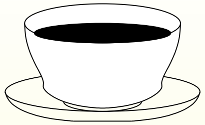
```

  // Class method to obtain the present date.
  public static Date getPresentDate()
  {
    return presentDate;
  } // getPresentDate
```

- As before, we intend `Date` **instance**s to be **immutable object**s.

Class: objects: may be mutable or immutable

- When **design** a **class** decide whether its **instance**s are
  - **immutable object**s
    - once **construct**ed the **object state** cannot be changed
  - or **mutable object**s
    - state can be changed after construction.

The `Date` class



Coffee time: Do you think it was appropriate for us to decide that our `Date` objects should be immutable? For example, suppose you are planning to go on holiday on the 20th July, but the tour operator has to change your departure date to the 21st of July due to a flight cancellation. Has the date known as 20th July itself changed to become the 21st of July? Or are those two dates still distinct, but instead, the details of *your holiday* have changed?

- Simplest way to ensure immutable objects:
  - declare all **instance variable**s as **final variable**s. . .

Variable: final variables

- Can write **reserved word `final`** as **modifier** on a **variable**
  - means value cannot be altered once has been assigned.
- An **instance variable** declared as **final variable**
  - must have value by time **object** has finished being **construct**ed
    * either by assigning value in **variable declaration**
    * or **assignment statement** inside **constructor method**.

The Date class

```
// Instance variables: the day, month and year of a date.
private final int day, month, year;
```

- Rest same as previously:

```
// Construct a date -- given the required day, month and year.
public Date(int requiredDay, int requiredMonth, int requiredYear)
{
  day = requiredDay;
  month = requiredMonth;
  year = requiredYear;
} // Date


```

The Date class

```java
// Compare this date with a given other one, for equality.
public boolean equals(Date other)
{
  return day == other.day && month == other.month && year == other.year;
} // equals


// Compare this date with a given other one, for less than.
public boolean lessThan(Date other)
{
  return year < other.year
         || year == other.year
            && (month < other.month
                || month == other.month && day < other.day);
} // lessThan


```

The Date class

```java
  // Return the day/month/year representation of the date.
  public String toString()
  {
    return day + "/" + month + "/" + year;
  } // toString


  // Return a new Date which is one year later than this one.
  public Date addYear()
  {
    return new Date(day, month, year + 1);
  } // addYear

} // class Date
```

The Person class

- Person **instances** are also **immutable objects**.

```
// Representation of a person.
public class Person
{
  // The name and birthday of a person.
  private final String name;
  private final Date birthDate;


  // Construct a person -- given the required name and birthday.
  public Person(String requiredName, Date requiredBirthDate)
  {
    name = requiredName;
    birthDate = requiredBirthDate;
  } // Person
```

The Person class

- The `ageHistory()` instance method **return**s `String` with **new line**s in it.
- For portability use platform dependent **line separator**. . .

Standard API: `System`: `getProperty()`

- The **class method** `System.getProperty()` gives access to various **system property** values
  - e.g. Java version, platform, user home directory, . . .
  - takes name of property as **method parameter**
  - **return**s corresponding `String` value.

Standard API: System: getProperty(): line.separator

- `System.getProperty()` maps `"line.separator"` onto the **line separator** for platform in use.
- E.g.

```
String lineSep = System.getProperty("line.separator");
```

The Person class

- Store (**reference** to) line separator in conveniently named **variable**
  - use that instead of "\n"
  - (also provide facility to change it – resuse this code in a later example which needs "\n" regardless of platform).

```
// The correct line separator for this platform.
private static String NLS = System.getProperty("line.separator");

// Override the default line separator.
public static void setLineSeparator(String requiredLineSeparator)
{
  NLS = requiredLineSeparator;
} // setLineSeparator

```

The Person class

```

  // Return the age history of this person.
  public String ageHistory()
  {
    Date presentDate = Date.getPresentDate();

    // Deal with cases where the person has just been born
    // or is not yet born.
    if (presentDate.equals(birthDate))
      return name + " was, or will be, born today!";
    else if (presentDate.lessThan(birthDate))
      return name + " will be born on " + birthDate;
```

The Person class

```
    else // The person was born before today.
    {
      // Start with the event of birth.
      String result = name + " was born on " + birthDate;

      // Now we will go through the years since birth but before today.
      // We keep track of the birthday we are considering.
      Date someBirthday = birthDate.addYear();
      int ageOnSomeBirthday = 1;
      while (someBirthday.lessThan(presentDate))
      {
        result += NLS + name + " was " + ageOnSomeBirthday
                  + " on " + someBirthday;
        someBirthday = someBirthday.addYear();
        ageOnSomeBirthday++;
      } // while
```

The Person class

```java

      // Now deal with the next birthday.
      if (someBirthday.equals(presentDate))
        result += NLS + name + " is " + ageOnSomeBirthday + " today!";
      else
        result += NLS + name + " will be " + ageOnSomeBirthday
                  + " on " + someBirthday;

      return result;
    } // else
  } // ageHistory

} // class Person
```

The `Person` class

- Generating age history done in `Person`
  - because about persons.
- Printing age history to **standard output** done in `AgeHistory`
  - because is what *this* program needs.
- Another program might want to do something different with age histories
  - can use `Person` class without needing to change it
    - ∗ because achieved good **encapsulation**.
- Guide: **putting the logic where the data is**.

Method: returning a value: multiple returns

- Use **return statement** to say what value **return**ed from **non-void method**
  - causes execution to end,
    control transfers back to code that called **method**.
- Often last **statement** in method
  - but can have one or more anywhere in method.
- Java **compiler** checks:
  - No path through method not ending with return statement.
  - No code in method that can never be reached due to earlier return statement.

The `Person` class

Coffee time: Does the above `ageHistory()` instance method satisfy those rules?

Trying it

- Not a full set of tests.

Console Input / Output

```
$ java AgeHistory
Enter today's date as three numbers, dd mm yyyy: 01 07 2019
Enter the number of people: 1
Enter the name of person 1: Joey
Enter his/her birthday (dd mm yyyy): 01 07 2019
Joey was, or will be, born today!
$ _
```

Run

Console Input / Output

```
$ java AgeHistory
Enter today's date as three numbers, dd mm yyyy: 01 07 2019
Enter the number of people: 1
Enter the name of person 1: Abi
Enter his/her birthday (dd mm yyyy): 2 07 2019
Abi will be born on 2/7/2019
$ _
```

Run

Trying it

Console Input / Output

```
$ java AgeHistory
Enter today’s date as three numbers, dd mm yyyy: 01 07 2019
Enter the number of people: 2
Enter the name of person 1: John
Enter his/her birthday (dd mm yyyy): 24 4 1959
John was born on 24/4/1959
John was 1 on 24/4/1960
John was 2 on 24/4/1961
 (... lines removed to save space.)
John will be 61 on 24/4/2020
Enter the name of person 2: Lizzy
Enter his/her birthday (dd mm yyyy): 1 7 1989
Lizzy was born on 1/7/1989
Lizzy was 1 on 1/7/1990
...
$ _
```

Run

Coursework: `ShapeShift`

(Summary only)

Write a program to create and process two-dimensional shapes.

Section 3

Example: Greedy children

Aim

AIM: To reinforce **object oriented design**, particularly with **mutable objects**. We also meet multiple **constructor methods**, **class constants**, the **return statement** with no value, **accessor methods**, **mutator methods**, the dangers of **method parameters** which are **references**, converting the **null reference** to a string, and `Math.random()`.

Greedy children

- Tongue-in-cheek model of greedy children scoffing ice cream.

  A program is required that will provide a very simple model of the behaviour of greedy children visiting ice cream parlours. Each greedy child has a name and a fixed capacity, which is an amount of ice cream he or she can hold. This capacity can either be specified, or be chosen as a random number up to some maximum. A child also has an amount of ice cream currently in the stomach. This starts off as being zero, but increases through eating, up to his or her capacity. Children can visit ice cream parlours and attempt to eat an amount of ice cream. Being greedy, they may well attempt to eat more than they have room left for, in which case they end up spilling the excess ice cream down their T-shirt! A child keeps track of how much ice cream he or she has spilt, which is initially zero.

Greedy children

Ice cream parlours have a name and an amount of ice cream, initially zero. They can accept deliveries of ice cream, which increases their stock level. They also can serve ice cream to greedy children, which reduces their stock level. Greedy children ask for an amount of ice cream, which they will attempt to eat, unless the parlour's stock level is **less than** that amount, in which case the children are served with as much ice cream as is left.

The program should demonstrate the simple model by creating some children and parlours, and have some deliveries made, and children served, etc.. As this is done, reports should be produced on the **standard output**, enabling the user of the program to follow the events. In this sense then, the main method of the program will tell a little story, and can be made to tell a different story by changing the code.

Identifying the classes

- Analyse requirements:

Class list for GreedyChildren

Class	**Description**
`GreedyChildren`	The main class containing the **main method**. It will make instances of `IceCreamParlour` and `GreedyChild`.
`IceCreamParlour`	An instance of this will represent an ice cream parlour.
`GreedyChild`	An instance of this will represent a greedy child.

Designing the class interfaces

Public method interfaces for class `GreedyChildren`.

Method	Return	Arguments	Description
`main`		`String[]`	The main method for the program.

Designing the class interfaces

Public method interfaces for class `IceCreamParlour`.

Method	**Return**	**Arguments**	**Description**
Constructor		`String`	Construct an ice cream parlour with the given `String` name.
`acceptDelivery`		**`double`**	Accept an ice cream delivery of the given amount, which increases the stock level.

Designing the class interfaces

Public method interfaces for class `IceCreamParlour`.

Method	**Return**	**Arguments**	**Description**
`tryToServe`	**`double`**	**`double`**	Attempt to serve the given amount of ice cream, and **return** the amount actually served. This is the amount asked for, or as much as the parlour can provide if the stock is too low. The stock level is reduced by the amount returned.
`toString`	`String`		Returns a representation of the ice cream parlour, showing name and stock level.

Designing the class interfaces

Public method interfaces for class `GreedyChild`.

Method	Return	Arguments	Description
Constructor		`String`, **`double`**	Construct a greedy child with the given `String` name and **`double`** stomach capacity.
Constructor		`String`	Construct a greedy child with the given `String` name and a randomly chosen stomach capacity.
`enterParlour`		`IceCreamParlour`	This child enters the given parlour, implicitly leaving any parlour s/he is already in.

Designing the class interfaces

Public method interfaces for class `GreedyChild`.

Method	Return	Arguments	Description
`leaveParlour`			This child leaves the parlour s/he is currently in, if any, so that s/he is not in any parlour afterwards.
`eat`		**`double`**	If this child is in a parlour, s/he attempts to eat ice cream, served by that parlour. The amount desired is the given **`double`**. The served amount adds to his/her stomach contents, with any excess being spilt once s/he is full. The method has no effect if s/he is not in a parlour.

Designing the class interfaces

Public method interfaces for class **`GreedyChild`**.

Method	Return	Arguments	Description
`toString`	`String`		Returns a representation of the greedy child, showing name, capacity, contents, spillage and which parlour the child is currently in.

- Also fixed maximum value for when stomach capacity chosen randomly.
- Two **constructor methods** for `GreedyChild`?

Method: constructor methods: more than one

- A **class** can have many **constructor methods**
  - as long as **types** of **method parameters** different
  - so **compiler** knows which one to use.

Designing the class interfaces

Coffee time: Look at the interface descriptions above and decide which classes will be used to make **mutable objects**.

The IceCreamParlour class

```
/* Ice cream parlours have a name and an amount of ice cream, initially zero.
   They can accept deliveries of ice cream, which increases their stock level.
   They also can serve ice cream to greedy children, which reduces their stock
   level. Greedy children ask for an amount of ice cream, which they will
   attempt to eat, unless the parlour’s stock level is less than that amount,
   in which case the children are served with as much ice cream as is left.
 */
```

The `IceCreamParlour` class

```java
public class IceCreamParlour
{
  // The name of the parlour.
  private final String name;

  // The amount of ice cream in stock.
  private double iceCreamInStock = 0;
```

Coffee time: What is the significance of us making one of these instance variables be a **final variable**, but not the other? Are **instance**s of `IceCreamParlour` **mutable object**s?

The IceCreamParlour class

```
// Construct an ice cream parlour -- given the required name.
public IceCreamParlour(String requiredName)
{
  name = requiredName;
} // IceCreamParlour
```

- Simplicity: ignore checking negative delivery amounts, etc..

```
// Accept delivery of ice cream.
public void acceptDelivery(double amount)
{
  iceCreamInStock += amount;
} // acceptDelivery
```

The IceCreamParlour class

```java
// Serve ice cream. Attempt to serve the amount desired
// but as much as we can if stock is too low.
// Return the amount served.
public double tryToServe(double desiredAmount)
{
  double amountServed = desiredAmount;
  if (amountServed > iceCreamInStock)
    amountServed = iceCreamInStock;

  iceCreamInStock -= amountServed;
  return amountServed;
} // tryToServe
```

The IceCreamParlour class

```java
  // Return a String giving the name and state.
  public String toString()
  {
    return name + " has " + iceCreamInStock + " in stock";
  } // toString

} // class IceCreamParlour
```

The GreedyChild class

```
/* Each greedy child has a name and a fixed stomach size, which is an amount
   of ice cream he or she can hold. This capacity can either be specified, or
   be chosen as a random number up to some maximum. A child also has a current
   stomach contents which starts off as being zero, but increases, through
   eating, up to his or her stomach size. Children can visit ice cream
   parlours and attempt to eat an amount of ice cream. Being greedy, they may
   well attempt to eat more than they have room left for, in which case they
   end up spilling the excess ice cream down their T-shirt! A child keeps
   track of how much ice cream he or she has spilt, initially zero.
 */
public class GreedyChild
{
```

Variable: final variables: class constant

- A **class variable** declared as **final variable** also known as **class constant**.
- E.g. `PI` in `Math` **class**:

```
public static final double PI = 3.14159265358979323846;
```

- Convention – class constant names use only capital letters
  - words separated by underscores (_).

The GreedyChild class

```
// When a GreedyChild is created with no given capacity
// a random one is chosen up to this maximum.
public static final double MAXIMUM_RANDOM_STOMACH_SIZE = 20.0;
```

The GreedyChild class

```
  // The name of the child.
  private final String name;

  // The amount of ice cream the child can hold before being full.
  private final double stomachSize;

  // The total amount of ice cream that the child has spilt by
  // attempting to eat after being full. Initially zero.
  private double tShirtStainSize = 0;

  // The amount of ice cream currently in the child’s stomach.
  // Initially zero.
  private double stomachContents = 0;

  // The ice cream parlour the child is currently in,
  // or null if s/he is not in one.
  private IceCreamParlour currentParlour = null;
```

The GreedyChild class

```java
// Construct a greedy child  -- given the required name and size.
public GreedyChild(String requiredName, double requiredStomachSize)
{
  name = requiredName;
  stomachSize = requiredStomachSize;
} // GreedyChild
```

Standard API: `Math`: `random()`

- Standard **class** `java.lang.Math` has **class method** `random`
  - no **method arguments**
  - **returns** **`double`** r, such that: $0.0 \leq r < 1.0$
- Pseudo randomly chosen
  - approximately uniform distribution of random numbers.

The GreedyChild class

```java
// Construct a greedy child  -- given the required name
// with a randomly chosen size.
public GreedyChild(String requiredName)
{
  name = requiredName;
  stomachSize = Math.random() * MAXIMUM_RANDOM_STOMACH_SIZE;
} // GreedyChild
```

The GreedyChild class

```java
// Enter an ice cream parlour.
public void enterParlour(IceCreamParlour parlourEntered)
{
  currentParlour = parlourEntered;
} // enterParlour


// Leave an ice cream parlour.
public void leaveParlour()
{
  currentParlour = null;
} // leaveParlour
```

Method: return with no value

- A **void method** may have **return statement**s with no **return** value just **`return`**
  - cause execution of **method** to end
  - control transfer to code that called method.
- Permits **single entry, multiple exit** design
  - method starts at beginning
  - various exits
    - ∗ depending on **condition**s.

The GreedyChild class

```java
// Attempt to eat a given amount of ice cream from the current parlour.
// No effect if no parlour. Otherwise parlour attempts to serve that amount.
// Excess is spilt once full.
public void tryToEat(double amountDesired)
{
  if (currentParlour == null)
    return;

  double amountServed = currentParlour.tryToServe(amountDesired);
  double roomLeft = stomachSize - stomachContents;
  if (amountServed <= roomLeft)
    stomachContents += amountServed;
  else
  {
    stomachContents = stomachSize;
    tShirtStainSize += amountServed - roomLeft;
  } // if
} // tryToEat
```

The `GreedyChild` class

```java
  // The correct line separator for this platform.
  private static final String NLS = System.getProperty("line.separator");


  // Return a String giving the name and state.
  public String toString()
  {
    return name + " is " + stomachContents + "/" + stomachSize + " full"
           + " and has spilt " + tShirtStainSize + NLS
           + "(currently in " + currentParlour + ")";
  } // toString

} // class GreedyChild
```

Coffee time: In `toString()` above, what do you think will happen when `currentParlour` contains the **null reference**, **`null`**?

Method: accessor methods

- An **accessor method** – **public instance method**
  - reveals some/all of **object state**
  - without changing it.
- E.g. `getXyz()` for **instance variable** `xyz`
  - however perhaps for **class** with good **encapsulation** might not want to reveal instance variables. . .
- More general idea:
  - reveals some feature of the object which might or might not be directly implemented as single instance variable.

Method: mutator methods

- A **mutator method** – **public instance method**
  - alters some/all of **object state**.
- E.g. `setXyz()` for **instance variable** `xyz`.
- More general idea: changes value of some feature which might or might not be directly implemented as single instance variable.
- Obvious(?): only **mutable object**s have mutator methods.

The `GreedyChild` class

Coffee time: Which instance methods in `IceCreamParlour` and `GreedyChild` are **accessor method**s and which are **mutator method**s?

The GreedyChildren class

```java
/* This program demonstrates the simple model of greedy children eating at ice
   cream parlours. It creates some children and parlours, has deliveries made
   to the parlours, and children served at them. As this is done, it reports
   on the standard output, enabling the user of the program to follow the
   events. So the main method tells a story, and can easily be altered to tell
   a different one.
 */
public class GreedyChildren
{
```

The GreedyChildren class

```java
// Private helper method to make a delivery and report it.
private static void deliver(IceCreamParlour parlour, double amount)
{
  System.out.println(parlour);
  System.out.println("accepts delivery of " + amount);
  parlour.acceptDelivery(amount);
  System.out.println("Result: " + parlour);
  System.out.println();
} // deliver
```

- Note: above takes **reference** to an `IceCreamParlour` **object**
  - object gets altered!

Method: changing parameters does not affect arguments: but referenced objects can be changed

- Java uses **call by value**
  - **method parameter**s obtain only value from **method argument**
  - so **method** cannot effect calling environment
    via method parameters of **primitive type**.
- But for method parameters of **reference type**:
  - method can following **reference** and change state of **object**.
  - often what we want, but . . . .

Method: changing parameters does not affect arguments: but referenced objects can be changed

- E.g. assume `changeState()` is **instance method** in `SomeClass`, alters some **instance variable**s:

```
public static void changeSomething(SomeClass object, SomeType value)
{
  object.changeState(value); // This really changes the object referred to.
  object = null;             // This has no effect outside of this method.
  ...
} // changeSomething
  ...
  SomeClass variable = new SomeClass();
  changeSomething(variable, someValueOfSomeType);
```

- First line has had impact outside of method
  - but second line has not.

The GreedyChildren class

```java
// Private helper method to have a child eat at a parlour.
private static void eat(GreedyChild child, double amount,
                        IceCreamParlour parlour)
{
  System.out.println(child);
  System.out.println("is entering " + parlour);
  child.enterParlour(parlour);
  System.out.println(child);
  System.out.println("is eating " + amount);
  child.tryToEat(amount);
  System.out.println("Result: " + child);
  System.out.println();
} // eat
```

The GreedyChildren class

```java
// The main method tells the ‘story’.
public static void main(String[] args)
{
  System.out.println("Greedy children:");
  GreedyChild child1 = new GreedyChild("Bloated Basil", 20);
  System.out.println(child1);
  System.out.println("Making child with random capacity less than "
                     + GreedyChild.MAXIMUM_RANDOM_STOMACH_SIZE);
  GreedyChild child2 = new GreedyChild("Cautious Catherine");
  System.out.println(child2);
  GreedyChild child3 = new GreedyChild("Lanky Larry", 4);
  System.out.println(child3);
  System.out.println();

```

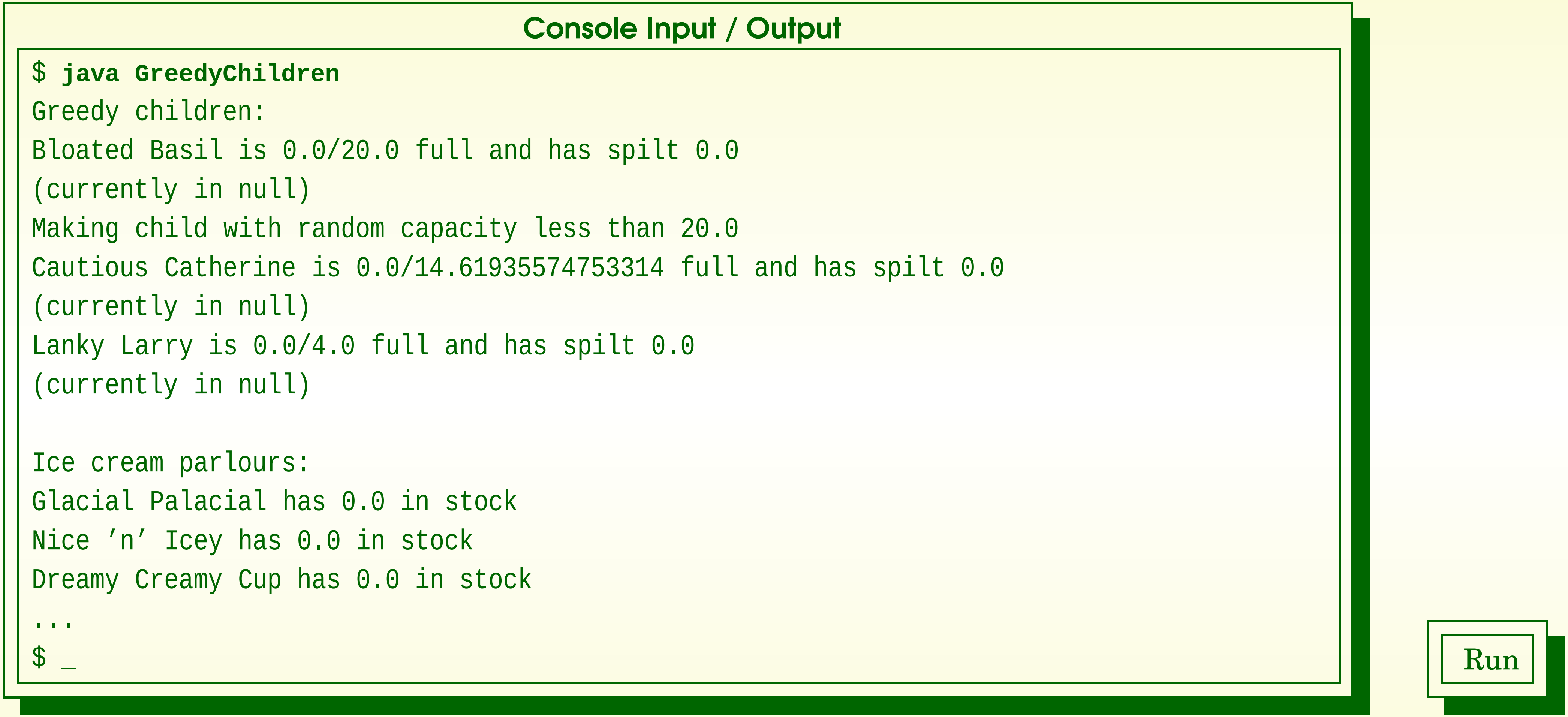
The GreedyChildren class

```
    System.out.println("Ice cream parlours:");
    IceCreamParlour parlour1 = new IceCreamParlour("Glacial Palacial");
    System.out.println(parlour1);
    IceCreamParlour parlour2 = new IceCreamParlour("Nice ’n’ Icey");
    System.out.println(parlour2);
    IceCreamParlour parlour3 = new IceCreamParlour("Dreamy Creamy Cup");
    System.out.println(parlour3);
    System.out.println();

    System.out.println("Deliveries:");
    System.out.println();
    deliver(parlour1, 50);
    deliver(parlour2, 10);
    deliver(parlour3, 30);
```

The GreedyChildren class

```
    System.out.println("Eating:");
    System.out.println();
    eat(child1, 15, parlour1);
    eat(child2, 1, parlour1);
    eat(child3, 2, parlour1);
    eat(child1, 8, parlour2);
    eat(child2, 1, parlour2);
    eat(child3, 2, parlour2);
    eat(child1, 10, parlour3);
    eat(child2, 1, parlour3);
    eat(child3, 2, parlour3);
  } // main

} // class GreedyChildren
```

Trying it



```
$ java GreedyChildren
Greedy children:
Bloated Basil is 0.0/20.0 full and has spilt 0.0
(currently in null)
Making child with random capacity less than 20.0
Cautious Catherine is 0.0/14.61935574753314 full and has spilt 0.0
(currently in null)
Lanky Larry is 0.0/4.0 full and has spilt 0.0
(currently in null)

Ice cream parlours:
Glacial Palacial has 0.0 in stock
Nice 'n' Icey has 0.0 in stock
Dreamy Creamy Cup has 0.0 in stock
...
$ _
```

- Note **null reference** printed as `null`.

Type: `String`: conversion: from object: null reference

- An **operand** of **concatenation** which is **object reference** has `toString()` **instance method** invoked
  - but what if is **null reference**?
    - Uses string `"null"` instead.

- Assume `someString` is `String`, `myVar` is **reference type**, then:

```
someString + myVar
```

treated as:

```
someString + (myVar == null
                 ? "null"
                 : (myVar.toString() == null ? "null" : myVar.toString()))
```

Type: `String`: conversion: from object: null reference

- Most Java programmers prefer

  ```
  "" + myVar
  ```

  to

  ```
  myVar.toString()
  ```

  - avoids possibility of **exception** if `myVar` is `null`.

Coursework: `StudentsCalling`

(Summary only)

Write a program that simulates the behaviour of students using their mobile phones.

Section 4

Example: Greedy children gone wrong

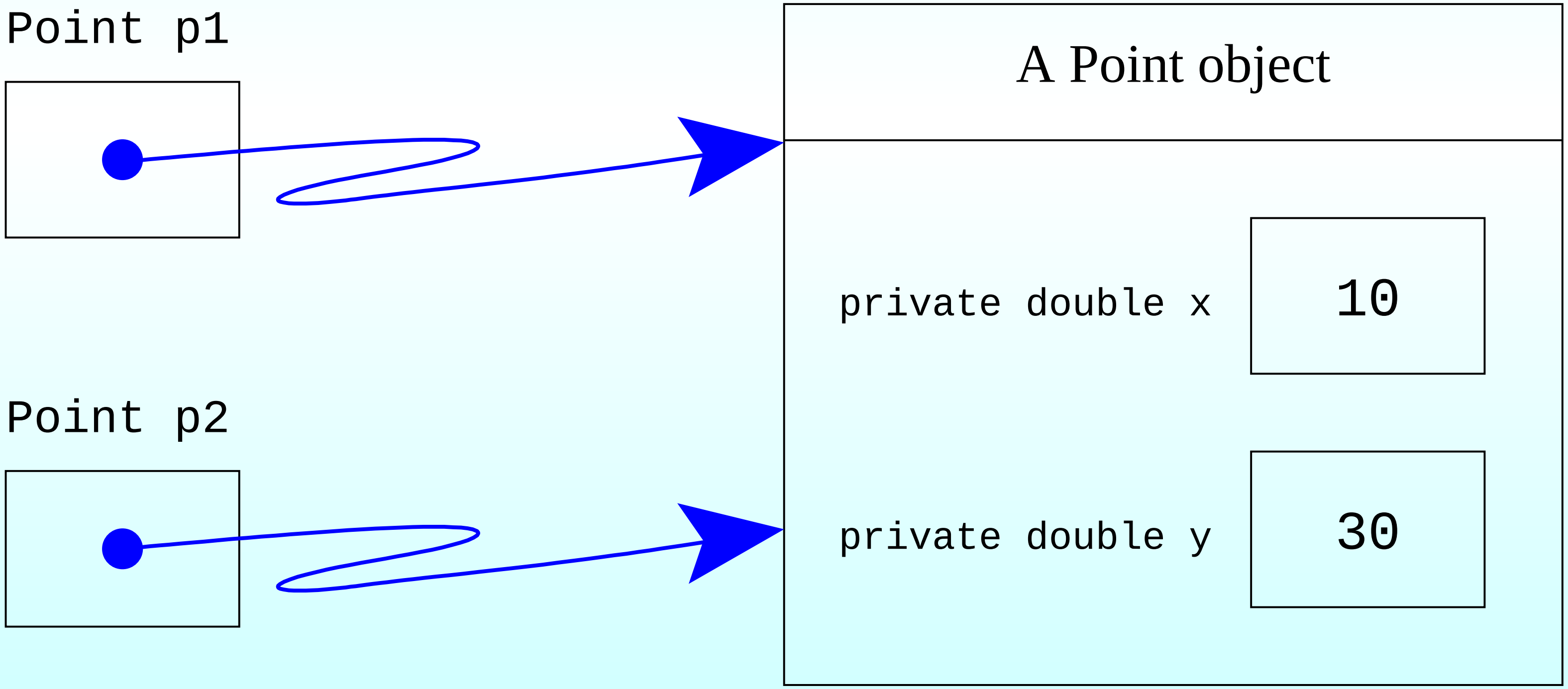
Aim

AIM: To look at the idea of an **object referenced by more than one variable** and the danger this presents when it is a **mutable object**.

Variable: of a class type: holding the same reference as some other variable

- Two or more **variable**s can hold **reference** to same **instance** of a class. E.g:

```
Point p1 = new Point(10, 30);


Point p2 = p1;
```



Variable: of a class type: holding the same reference as some other variable

- Causes no problems if is **immutable object**
  - cannot change object's state
    no matter which variable used to access it
  - in effect object referred to behaves the same as if two different objects.
- Following has almost same *effect*:

```
Point p1 = new Point(10, 30);


Point p2 = new Point(10, 30);
```

- Only difference is `p1 == p2` and `p1 != p2`
  - **`true`** and **`false`** versus **`false`** and **`true`**.

Variable: of a class type: holding the same reference as some other variable

- If **object referenced by more than one variable** is **mutable object** we must be careful
  - any change made via any one variable has effect on (same) object referred to by other variables.
    - May be what we want
    - may be a problem if poor **design** or mistake in code.
- E.g. . . .

Variable: of a class type: holding the same reference as some other variable

```
public class Employee
{
  private final String name;
  private int salary;

  public Employee(String requiredName, int initialSalary)
  {
    name = requiredName;
    salary = initialSalary;
  } // Employee

  public String getName()
  {
    return name;
  } // getName
```

Variable: of a class type: holding the same reference as some other variable

```
  public void setSalary(int newSalary)
  {
    salary = newSalary;
  } // setSalary

  public int getSalary()
  {
    return salary;
  } // getSalary

} // class Employee
```

Variable: of a class type: holding the same reference as some other variable

```
...

Employee debora = new Employee("Debs", 50000);
Employee sharmane = new Employee("Shaz", 40000);

...

Employee worstEmployee = debora;
Employee bestEmployee = sharmane;

...
```

- Accidental code:

```
worstEmployee = bestEmployee;
```

Variable: of a class type: holding the same reference as some other variable

- Continued intentional code:

```
...

bestEmployee.setSalary(55000);
worstEmployee.setSalary(0);

System.out.println("Our best employee, " + bestEmployee.getName()
                   + ", is paid " + bestEmployee.getSalary());
System.out.println("Our worst employee, " + worstEmployee.getName()
                   + ", is paid " + worstEmployee.getSalary());
```

- Result: Debora keeps her 50,000; Sharmane increased to 55,000 but then cut to 0. Output:

```
Our best employee, Shaz, is paid 0
Our worst employee, Shaz, is paid 0
```

Greedy children gone wrong

```java
public class GreedyChildren
{
  // Private helper method to make a delivery and report it.
  private static void deliver(IceCreamParlour parlour, double amount)
  {
    System.out.println(parlour);
    System.out.println("accepts delivery of " + amount);
    parlour.acceptDelivery(amount);
    System.out.println("Result: " + parlour);
    System.out.println();
  } // deliver
```

Greedy children gone wrong

- Simplified – make only **instance**s of `IceCreamParlour`.

```
public static void main(String[] args)
{
  IceCreamParlour parlour1 = new IceCreamParlour("Glacial Palacial");
  System.out.println(parlour1);
  IceCreamParlour parlour2 = new IceCreamParlour("Nice ’n’ Icey");
  System.out.println(parlour2);
  IceCreamParlour parlour3 = new IceCreamParlour("Dreamy Creamy Cup");
  System.out.println(parlour3);
  System.out.println();
```

- ‘Accidental’ piece of code

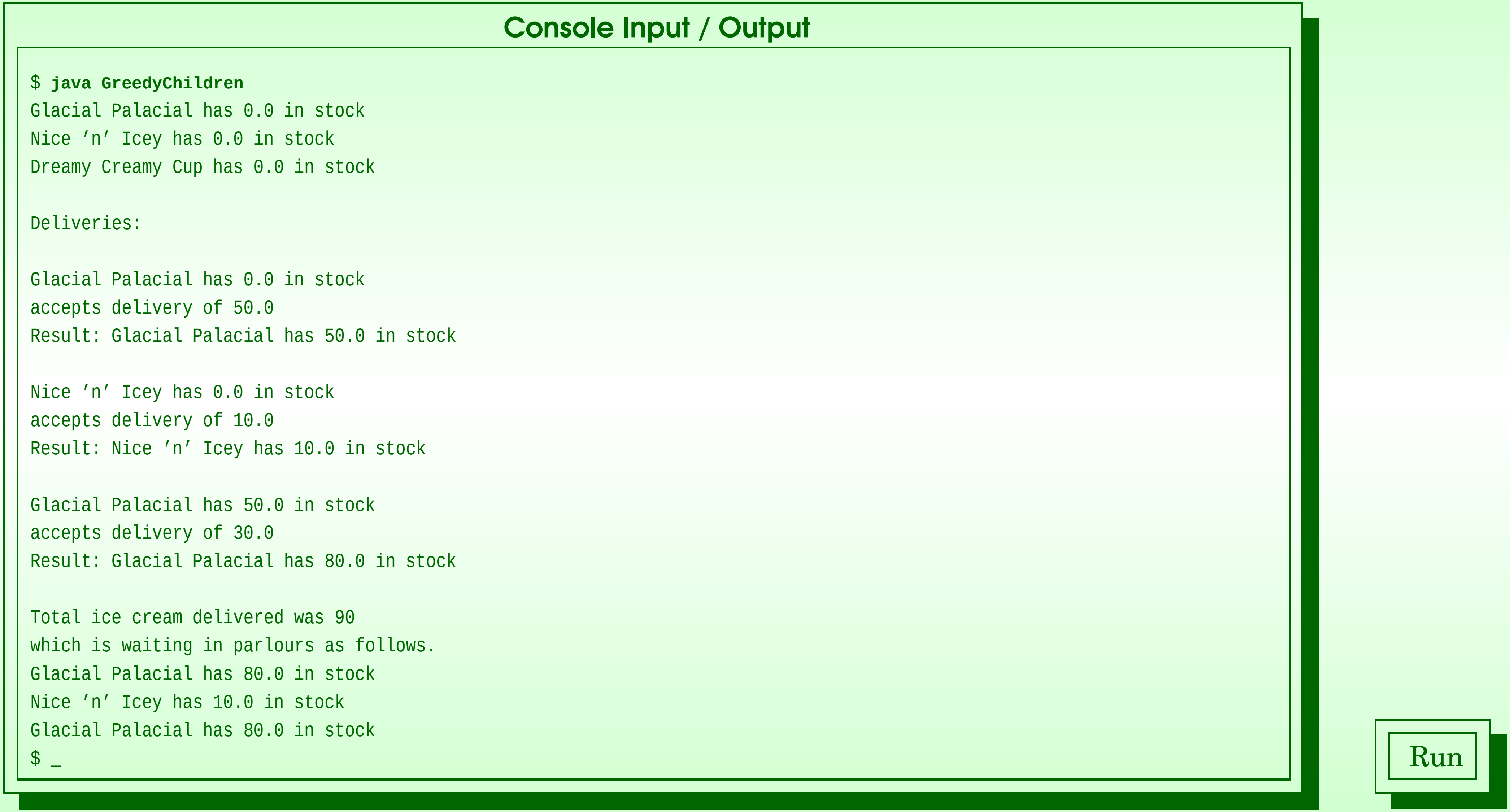
```
  parlour3 = parlour1;
```

Greedy children gone wrong

```
    System.out.println("Deliveries:");
    System.out.println();
    deliver(parlour1, 50);
    deliver(parlour2, 10);
    deliver(parlour3, 30);

    System.out.println("Total ice cream delivered was " + (50 + 10 + 30));
    System.out.println("which is waiting in parlours as follows.");
    System.out.println(parlour1);
    System.out.println(parlour2);
    System.out.println(parlour3);
  } // main

} // class GreedyChildren
```

Coffee time: Before reading on, predict what the output of the program will be.

Trying it



Console Input / Output

```
$ java GreedyChildren
Glacial Palacial has 0.0 in stock
Nice ’n’ Icey has 0.0 in stock
Dreamy Creamy Cup has 0.0 in stock

Deliveries:

Glacial Palacial has 0.0 in stock
accepts delivery of 50.0
Result: Glacial Palacial has 50.0 in stock

Nice ’n’ Icey has 0.0 in stock
accepts delivery of 10.0
Result: Nice ’n’ Icey has 10.0 in stock

Glacial Palacial has 50.0 in stock
accepts delivery of 30.0
Result: Glacial Palacial has 80.0 in stock

Total ice cream delivered was 90
which is waiting in parlours as follows.
Glacial Palacial has 80.0 in stock
Nice ’n’ Icey has 10.0 in stock
Glacial Palacial has 80.0 in stock
$ _
```

Concepts covered in this chapter

- Each book chapter ends with a list of concepts covered in it.
- Each concept has with it
  - a self-test question,
  - and a page reference to where it was covered.
- Please use these to check your understanding before we start the next chapter.