COMP90041
Programming and Software Development

Getting Started

Semester 2, 2015

Focus of the Subject

Object-Oriented (OO) software development:

- the Java programming language
- OO concepts:
  - classes
  - objects
  - encapsulation
  - inheritance
  - polymorphism
- problem solving
- small-scale program design, implementation and testing

Focus of the Subject (2)

- Best practices in software development:
  - Good programming style
  - Good documentation habits
  - Following specifications

> *[A computer is] like an Old Testament god, with a lot of rules and no mercy.*
>
> — *Joseph Campbell*

Subject Structure

- Twelve 2-hour lectures
  - short break in each lecture
- Twelve 1-hour workshops
  - Beginning in week 1
  - Select any one workshop session
- Assessment:
  - 40/100 project work
  - 60/100 Final exam
  - You must pass both components to pass the subject
  - All assessments will be individual
- Textbook:
  Walter Savitch, Absolute Java, 5th Edition, Addison Wesley.
  3rd or 4th Edition also fine.

Project Work

- 5 assessed workshop submissions
  - Due Friday at 5pm in weeks 3, 5, 7, 9, and 11
  - Assessed online; submit as often as you like
  - Only correctness matters
  - Each submission is worth 5 points
  - Drop the lowest mark (20 points in total)
- 1 larger project
  - Due after the semester break (around week 10)
  - Worth 15 points
  - Code quality matters, as well as correctness
- You will be asked to critique your classmates code
  - Worth 5 points

Subject Resources

- All available from the Learning Management System (LMS). `http://www.lms.unimelb.edu.au/`

Student Representatives

- This subject needs 1 or 2 student representatives
- Student reps act as a conduit for anonymous student feedback by email or in time set aside in one lecture
- Reps also report back to department staff on how the class is going (and get a free lunch!)
- Email me if you want to volunteer

Academic Misconduct

- All project work is to be done by you alone
- You can discuss overall approach to solving problems with peers or others
- **Do not** show your code to peers, in person or electronically, or ask peers for code
- When in doubt, ask lecturer or demonstrator
- I will use sophisticated software to identify cheating

QuickPoll

- We will use the QuickPoll system to check your understanding
- Go to `http://bit.ly/schachte` on any internet device
- This is anonymous and not assessed,
- . . . but it's a good way to check your progress,
- . . . and you learn more if you tackle these problems

QuickPoll: Transport

How did you get to uni today?

A. Foot
B. Bicycle
C. Car/Motorcycle
D. Train/Tram/Bus
E. Helicopter

QuickPoll: Background

How much programming experience do you have?

A. I know Java well
B. I know some Java
C. I know C++ or C#, but not Java
D. I know another programming language
E. I dont know any programming language

This Subject

- Has no prerequisites: designed for beginners
- If you know Java well, you'll be bored
- Only covers the Java language, not frameworks, or GUIs, or advanced usage
- People with some programming background will have some advantage, but students with no programming background should be fine
- Don't suffer in silence: if you are having trouble with the subject, speak to your demonstrator or to me

How to succeed in this class

What people think is important:

- Be a “computer person”
- Be good at maths
- Know logic

How to succeed in this class

What people think is important:

- ~~Be a “computer person”~~
- Be good at maths
- Know logic

How to succeed in this class

What people <u>think</u> is important:

- ~~Be a "computer person"~~
- ~~Be good at maths~~
- Know logic

How to succeed in this class

What people think is important:

- ~~Be a "computer person"~~
- ~~Be good at maths~~
- ~~Know logic~~

How to succeed in this class

What really <u>is</u> important:

How to succeed in this class

What really <u>is</u> important:

- It's like learning a new human language

How to succeed in this class

What really <u>is</u> important:

- It's like learning a new human language
- Rewiring your brain

How to succeed in this class

What really is important:

- It's like learning a new human language
- Rewiring your brain
- Be patient with yourself

How to succeed in this class

What really is important:

- It's like learning a new human language
- Rewiring your brain
- Be patient with yourself
- Be persistent

How to succeed in this class

What really <u>is</u> important:

- It's like learning a new human language
- Rewiring your brain
- Be patient with yourself
- Be persistent
- **Practice!**

Things to do in Week 1

- Buy textbook
- Read Chapters 1 and 2
- Attend the first workshop
- Download and install a Java IDE
- Try to compile and run a Java program
- Prepare lab 2 exercises

Operating Systems

- The operating system (OS) controls the computer's hardware devices
  - Such as hard disk and DVD drives, mice, keyboards, display screens, etc.
  - Saves programs from having to directly control each device
- The OS controls which programs run
- Modern OSes allow multiple programs to run at once
- OS shares resources among programs
- Many OSes have a graphical interface and a text-based command line (shell) for control
- Shell is less friendly, but much more powerful

Operating Systems

- Commonly used operating systems:
  - Windows, Mac OS X, iOS, Android, GNU/Linux, Solaris, NetBSD, ...
  - All of these but Windows are based on Unix
- Most desktop and laptop computers run Windows
- Most mobile, server, mainframe, and supercomputers run some Unix variety
- If you plan to pursue a career in computing, you should get comfortable with Unix/Linux

OS for this Subject

- We will be using Windows in the lab
- You can use whatever OS you like to do exercises and the project
- You can bring your laptop to labs and use the OS you prefer
- **You must use the Unix shell** to submit your work
- In this week's workshop, you will learn how to access the Linux servers

Hello, World!

- Our first Java program:

```
// print out a friendly greeting
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

- For now, treat parts you don't understand as boilerplate
- All will be explained. . . .

Java Code

```java
// print out a friendly greeting
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

- Java program is made up of one or more classes

Java Code

```
// print out a friendly greeting
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

- Java program is made up of one or more classes
- Java class is made up of zero or more methods and instance variables

Java Code

```
// print out a friendly greeting
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

- Java program is made up of one or more classes
- Java class is made up of zero or more methods and instance variables
- Java method is made up of zero or more statements

Java Code

```java
// print out a friendly greeting
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

- Java program is made up of one or more classes
- Java class is made up of zero or more methods and instance variables
- Java method is made up of zero or more statements
- There are a few other things, like comments

Developing Code

- Most people use an IDE (Integrated Development Environment) to develop Java code
  - Popular IDEs include Eclipse and Netbeans
  - Both are free to download and use
  - Use whatever development tools you like
- IDEs hide some boring details
  - But you still need to understand the details

Files

- Every line of Java code must be in some text file
- The file name must match the class name, including upper/lower case, and end with "`.java`"

Files

- Every line of Java code must be in some text file
- The file name must match the class name, including upper/lower case, and end with ".java"
- This class should be in the file "Hello.java":

```
public class Hello {
    ⋮
}
```

Files

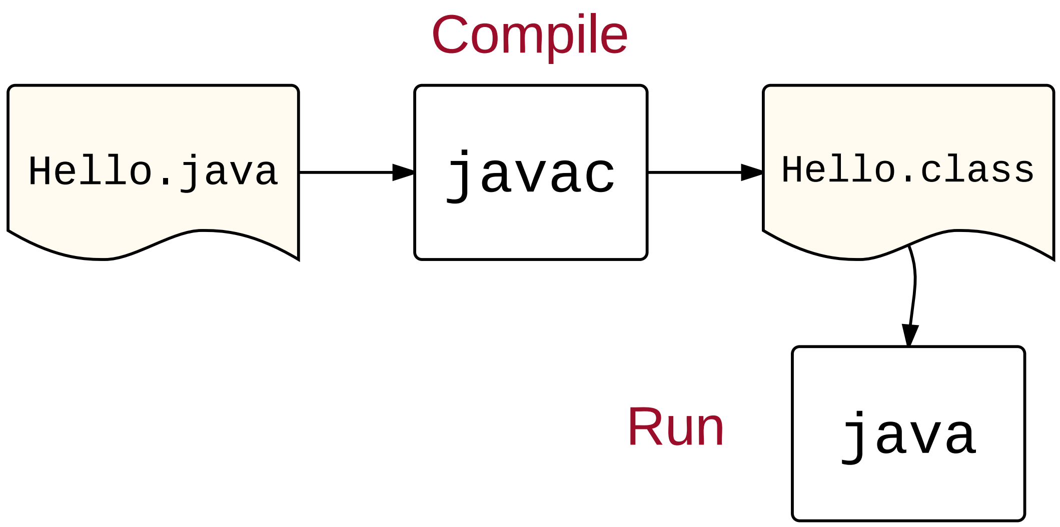
- Every line of Java code must be in some text file
- The file name must match the class name, including upper/lower case, and end with “.java”
- This class should be in the file “Hello.java”:

```
public class Hello {
    ⋮
}
```

- IDEs try to keep these consistent, but be careful

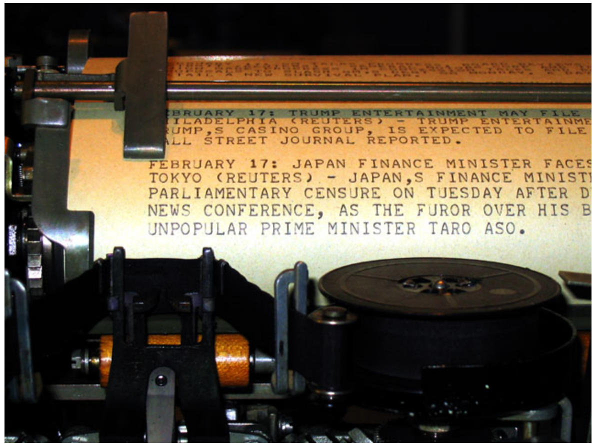
Compilation

- Before a program can be run, it must be compiled with the `javac` command
  - Checks that the program obeys the rules of Java
  - Produces a `.class` file (if OK)
- Once compiled, program is run using the `java` command



Execution

- Java applications run in a console
- Computer consoles originally looked like this



- This subject will focus on console (text) input/output
- IDEs simulate console input/output

Command Line Execution

```
frege%
```

- `frege%` is my OS prompt; yours will be different
- Compile with: `javac` followed by the file name

Command Line Execution

```
frege% javac Hello.java
```

- `frege%` is my OS prompt; yours will be different
- Compile with: `javac` followed by the file name

Command Line Execution

```
frege% javac Hello.java
```

- frege% is my OS prompt; yours will be different
- Compile with: javac followed by the file name
- If no errors detected, just prints next prompt

Command Line Execution

```
frege% javac Hello.java
frege%
```

- frege% is my OS prompt; yours will be different
- Compile with: javac followed by the file name
- If no errors detected, just prints next prompt

Command Line Execution

```
frege% javac Hello.java
frege%
```

- `frege%` is my OS prompt; yours will be different
- Compile with: `javac` followed by the file name
- If no errors detected, just prints next prompt
- Run program with: `java` followed by module (not file) name

Command Line Execution

```
frege% javac Hello.java
frege% java Hello
```

- `frege%` is my OS prompt; yours will be different
- Compile with: `javac` followed by the file name
- If no errors detected, just prints next prompt
- Run program with: `java` followed by module (not file) name

Command Line Execution

```
frege% javac Hello.java
frege% java Hello
```

- `frege%` is my OS prompt; yours will be different
- Compile with: `javac` followed by the file name
- If no errors detected, just prints next prompt
- Run program with: `java` followed by module (not file) name
- Program output is shown, followed by next OS prompt; keyboard input may be needed

Command Line Execution

```
frege% javac Hello.java
frege% java Hello
Hello, World!
frege%
```

- `frege%` is my OS prompt; yours will be different
- Compile with: `javac` followed by the file name
- If no errors detected, just prints next prompt
- Run program with: `java` followed by module (not file) name
- Program output is shown, followed by next OS prompt; keyboard input may be needed

Data

- Two key parts of any program: code and data
- Code is the text of the program, what operations the program performs, the verbs of the program
- Data is what the code operates on, the nouns of the program
- Each datum (singular of data) has a type
- Three kinds of types: primitive, class, and array
  - We will cover class and array types later

Primitive Types

- Building blocks: all data are built from primitives
- Primitives can't be broken into smaller parts

Type	Bytes	Values
`boolean`	1	`false`, `true`
`char`	2	All unicode characters (*e.g.*, `'a'`)
`byte`	1	-2^7 to $2^7 - 1$ (-128 to 127)
`short`	2	-2^{15} to $2^{15} - 1$ (-32768 to 32767)
`int`	4	-2^{31} to $2^{31} - 1$ ($\approx \pm 2 \times 10^9$)
`long`	8	-2^{63} to $2^{63} - 1$ ($\approx \pm 10^{19}$)
`float`	4	$\approx \pm 3 \times 10^{38}$ (limited precision)
`double`	8	$\approx \pm 10^{308}$ (limited precision)

QuickPoll: Primitive Types

Which of the following is <u>not</u> a primitive type??

A. bool
B. byte
C. char
D. int
E. double

QuickPoll: Primitive Types

Which of the following is not a primitive type??

A. bool
B. byte
C. char
D. int
E. double

Variables

- Variables have names and hold data
- Different values at different times
- Variable names begin with a letter, and follow with letters, digits, and underscores (`_`)
- Java convention is:
  - Begin with lower case letter
  - Follow with lower case, except
  - Capitalise first letter of each word in phrase
  - Run words together
- *E.g.*, `height`, `windowHeight`, `tallestWindowHeight`
- Best practice: make them descriptive, but not *toooooo* long (some clear abbrevs OK)

Variable Declaration and Assignment

- Each variable must be declared, specifying its type
  - Specify type first, then variable name, then semicolon
  - *E.g.*, `int count;` or `boolean done;`
- Variable may be assigned a value
  - Specify variable first, then equal sign (`=`), then value and semicolon
  - *E.g.*, `count = 1;` or `done = false;`
- You can combine declaration with initial assignment
  - Specify type, variable, equal sign (`=`), then value and semicolon
  - *E.g.*, `int count = 1;` or `boolean done = false;`

System.out.println

- Prints something out to the console
- The "ln" part means "end the line"
  - Next output will start a new line
- To print something without ending the line, use `System.out.print(`*something*`);`
- Always end each line with `System.out.println`, and be sure to put whitespace in where needed

```
String who = "World";
System.out.print("Hello");
System.out.println(who);
```

Output:

```
HelloWorld
```

- This example needed a space between words

Summary

- Write Java classes in file named *classname*`.java`
- Variables hold values, can be assigned and reassigned
- Variables must be declared, with their types
- Use `System.out.print` or `System.out.println` to print values

COMP90041
Programming and Software Development

Input/Outptut

Semester 2, 2015

Operations for Number Types

- Each type has certain operations that apply to it
- For primitive number types: `+ - * / %`
  - Type of result is same as type of operands
- Use operations to construct expressions, which have values that can be assigned or used as operands
  - *E.g.*, `answer = (2 + 4) * 7; count = count + 1;`
- Comparison operations also work for number types: `< <= > >= == !=`
- NB: `==` is comparison, `=` is assignment!
- Comparisons return `boolean` values
  - *E.g.*, `done = count >= answer;`

Operations for `booleans`

- `&&` (and) is true iff both operands are
  - *E.g.*, `test1 && test2`
- `||` (or) is true iff either operand is
  - *E.g.*, `test1 || test2`
- Both are short circuit operations: they only evaluate the second operand if necessary
  - *E.g.*, `dx != 0 && dy / dx > 0`
  - *E.g.*, `dx == 0 || dy / dx <= 0`
- `&` and `|` are non-short-circuit versions: they always evaluate both operands (rarely needed)
- `!` (not) is true iff its operand is false
  - *E.g.*, `!test1`

Strings

- `String` is a class type, so strings are objects
- Specify a string constant by enclosing in double-quote (`"`) characters
  - *E.g.*, `String s = "Example string"`;
- Include double-quote in string by preceding with backslash (`\`)
- Include backslash in string by preceding with backslash
  - *E.g.*, `"He said \"backslash (\\) is special!\""`
- Certain letters after backslash are treated specially
  - Most important: `\n`=newline (end current line), `\r`=return (go to start of current line), `\b`=backspace, `\t`=tab character
- These work for character constants, like `'\n'`

String Operations

- You can use `+` to append two strings
  - *E.g.*, `System.out.println("Hello " + "World");`
  - Prints "`Hello World`"
- `+` is clever: if either operand is a string, it will turn the other into a string
  - *E.g.*, `System.out.println("a = " + a + ", b = " + b);`
  - If `a` = 1 and `b` = 2, this prints: "`a = 1, b = 2`"
- But beware:

  `System.out.println("1 + 1 = " + 1 + 1);`

  actually prints "`1 + 1 = 11`"

String Operations

- You can use `+` to append two strings
  - *E.g.*, `System.out.println("Hello " + "World");`
  - Prints "`Hello World`"
- `+` is clever: if either operand is a string, it will turn the other into a string
  - *E.g.*, `System.out.println("a = " + a + ", b = " + b);`
  - If `a` = 1 and `b` = 2, this prints: "`a = 1, b = 2`"
- But beware:

  `System.out.println("1 + 1 = " + 1 + 1);`

  actually prints "`1 + 1 = 11`"
- Fix:

  `System.out.println("1 + 1 = " + (1 + 1));`

  prints "`1 + 1 = 2`"

More String Operations

- String class has <u>many</u> more operations, *e.g.*:
- Assume `String s, s2; int i, j;` then:
  - `s.length()` returns the length of the string
  - `s.toUpperCase()` returns ALL UPPER CASE version of string
  - `s.toLowerCase()` RETURNS all lower case VERSION
  - `s.substring(i,j)` returns the substring of `s` from character `i` through `j`-1, counting the first character as 0
  - *E.g.*, `"smiles".substring(1,5)` is `"mile"`
  - `s.equals(s2)` returns `true` iff `s` and `s2` are identical
  - `s.indexOf(s2)` returns the first position of `s2` in `s`
- Don't use `==`, `<`, `>=`, *etc.* to compare strings
- See String class documentation for more
  - Google "java string class"

Special Assignment Operations

- It's common to perform an operation on a variable and store the result back in the same variable
  - *E.g.*, `x = x + 1; x = x * 2; x = x - 2;` ...
- Java has a special form of assignment combined with each of these operations
- Just write the operation symbol followed by `=`
  - *E.g.*, `x += 1; x *= 2; x -= 2; x /= 10; x %= y; done |= x > 10; done &= x == 0;`
- Can also use `+=` to append to a string variable:

```
String msg = "Hello, ";
msg += "World!";
System.out.println(msg);
```

Prints "`Hello, World!`"

Pre/Post Increment/Decrement

- `++x` is a special expression that increments `x` (for any variable `x`) and returns the incremented value
  - *E.g.*, if `x` is 7, `++x` is 8, and after that, `x` is 8
  - Called "pre-increment" because it increments variable before returning it
- `--x` (pre-decrement) is similar: it decrements `x` and returns it
- `x++` (post-increment) returns `x` and then increments it
  - *E.g.*, if `x` is 7, `x++` is 7, and after that, `x` is 8
- `x--` (post-decrement) returns `x` and then decrements it

QuickPoll: Pre/Post Increment/Decrement

What will this code print?

```
int x = 10; int y = 5;
System.out.println(x++ - ++y);
```

A. 3
B. 4
C. 5
D. 6
E. 7

QuickPoll: Pre/Post Increment/Decrement

What will this code print?

```
int x = 10; int y = 5;
System.out.println(x++ - ++y);
```

- A 3
- B 4
- C 5
- D 6
- E 7

Pre/Post Increment/Decrement Use

- Pre/post increment/decrement can be confusing (like the last example!)
- They can also be used as statements rather than expressions
  - *E.g.*, `++x;` or `x++;`
  - Used as statements, these both just increment `x`
- This is the recommended way to use them

Type Conversion

- Primitive operations work on operands of the <u>same</u> type
- But Java can convert types for you automatically
- A <u>widening</u> conversion converts a number to a wider type (so the value can always be converted successfully)
- Automatic conversions in Java:

$$\begin{array}{c} \texttt{byte} \rightarrow \ \texttt{short} \rightarrow \ \texttt{int} \rightarrow \ \texttt{long} \rightarrow \ \texttt{float} \rightarrow \ \texttt{double} \\ \qquad\qquad\qquad\quad \uparrow \\ \qquad\qquad\qquad\quad \texttt{char} \end{array}$$

Casting

- Narrowing conversions are also possible
- But they must be performed explicitly using a cast
- Cast is specified by writing the name of the type to convert to in parentheses before the value to be converted
- Cast can be used to explicitly ask for a widening conversion

```
int sum;
int count;
// compute sum and count...
double average = (double)sum / count;
```

Precedence and Associativity

- Precedence of 2 operators, say $\odot$ and $\oplus$ determines whether $a \odot b \oplus c$ is read as:
  - $(a \odot b) \oplus c$ ($\odot$ has higher precedence), or
  - $a \odot (b \oplus c)$ ($\odot$ has lower precedence)
  - *E.g.*, `2+3*4` (`*` has higher precedence)
- Associativity determines whether $a \odot b \odot c$ is read as:
  - $(a \odot b) \odot c$ (left associativity), or
  - $a \odot (b \odot c)$ (right associativity)
  - *E.g.*, `3-2-1` (`-` associates left)
- Java's rules are mostly as you would expect
- When it doubt, just put in parentheses

Operators, High to Low Precedence

Symbol	Associativity
`.` (method invocation)	
`++ --`	
`-` (unary negation)	
`(`*type*`)` casts	
`* / %`	Left
`+ -`	Left
`< > <= >=`	Left
`= !=`	Left
`&&`	Left
`||`	Left
`= += *=` ...	Right

Formatted Output

- `printf` is like `print`, but it lets you control how data is formatted
- Form:
  `System.out.printf(`*`format-string`*`, `*`args`*`...);`
- `format-string` is an ordinary string, but can contain format specifiers, one for each of the `args`
  - Format specifier begins with `%`,
  - may have a number specifying how to format the next value in the `args...` list
  - ends with a letter specifying the type of the value
- Ordinary text in `format-string` is printed as is

Format Specifiers

- The (optional) number following % is interpreted:
  - The whole number part (before decimal point) specifies the minimum number of characters to be printed
  - The full number will be printed, even if takes more characters
  - If omitted, the value will be printed in its minimum width
  - If the number if negative, the value will be left-justified, otherwise right-justified
  - The part of the number after a decimal point specifies the number of digits of the value to print after the decimal point
  - If no decimal point, Java decides how to format

Format Letters

The final letter in a format specifier can be:

`d`	format an integer (no fractional part)
`s`	format a string (no fractional part)
`c`	format a character (no fractional part)
`f`	format a float or double
`e`	format a float or double in exponential notation
`g`	like either `%f` or `%e`, Java chooses
`%`	output a percent sign (no argument)
`n`	end the line (no argument)

- Good format for money: `$%.2f`

Formatted Output Example

```
public class printfExample {
    public static void main(String[] args) {
        String s = "string"; Double pi = 3.1415926535;
        System.out.printf("\"%s\" has %d characters%n",
                          s, s.length());
        System.out.printf("pi to 4 places: %.4f%n", pi);
        System.out.printf("Right>>%9.4f<<", pi);
        System.out.printf(" Left>>%-9.4f<<%n", pi);
    }
}
```

Generated output

```
"string" has 6 characters
pi to 4 places: 3.1416
Right>>   3.1416<< Left>>3.1416   <<
```

QuickPoll: Formatted Output

How many characters appear before the decimal point in a number `x` printed with `printf("%6.2f", x)`?

- (A) I don't know
- (B) 2
- (C) 3
- (D) 4
- (E) 6

QuickPoll: Formatted Output

How many characters appear before the decimal point in a number `x` printed with `printf("%6.2f", x)`?

- A. I don't know
- B. 2
- C. 3
- D. 4
- E. 6

Handling Command Line Input

- When your program is run, it can be given arguments on the command line
- Allows the user to give information to the program
- For the boilerplate we've been using, the command line arguments can be referred to as:
  - first command line argument: `args[0]`
  - second command line argument: `args[1]`
  - third command line argument: `args[2]`, *etc.*.
- Each of these is a `String`
- This will be explained in detail in a few weeks, but this will be enough for now

Command Line Input Example

```
// print out a friendly greeting
public class Hello2 {
    public static void main(String[] args) {
        System.out.println("Hello, " + args[0] + "!");
    }
}
```

Program Use

```
frege% java Hello2 Peter
Hello, Peter!
frege% java Hello2
Exception in thread "main" java.lang.ArrayIndexOutOfBo
undsException: 0
       at Hello2.main(Hello2.java:4)
```

Reading Console Input

- Interactive programs get input while running
- Java 5 introduces the `Scanner` class for this
- To use `Scanner`:
  - Add this near top of source file:

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

  - Create scanner: add this in `main` before reading input:

```
Scanner keyboard = new Scanner(System.in);
```

  - Use `keyboard` as needed to read input
  - *E.g.*, this reads (rest of) current line as a string:

```
String line = keyboard.nextLine();
```

Reading Console Input

- Other methods to read from a `Scanner` variable `keyboard`:

What	Type	Expression
One word	`String`	`keyboard.next()`
One integer	`int`	`keyboard.nextInt()`
One double	`double`	`keyboard.nextDouble()`

- Similar methods to read other types; see documentation
- These all skip over whitespace and read one "word"
- Whitespace includes spaces, tabs, and newlines
- Error if text is not of expected type

Command Line and Scanner Example

```
import java.util.Scanner;
public class ScannerExample {
    public static void main(String[] args) {
        int num1 = Integer.parseInt(args[0]);
        Scanner kbd = new Scanner(System.in);
        System.out.print("Enter second number: ");
        int num2 = kbd.nextInt();
        System.out.println(num1 + " * " + num2 +
                                  " = " + num1*num2);
    }
}
```

```
frege% java ScannerExample 6
Enter second number: 7
6 * 7 = 42
```

Pitfall: Mixing with nextLine

- nextLine() reads up to and including newline
- Others do not read after the next word
- After next, nextInt, or nextDouble, nextLine just reads rest of current line (maybe nothing!)
- To read a number on one line followed by the next whole line:

```
int num = keyboard.nextInt();
keyboard.nextLine(); // throw away rest of line
String line = keyboard.nextLine();
```

- Ideally, avoid mixing nextLine with the others

QuickPoll: What are s1 and s2 after:

```
Scanner kbd = new Scanner(System.in);
int n       = kbd.nextInt();
String s1   = kbd.nextLine();
String s2   = kbd.nextLine();
```

Console input (on 3 lines):

```
1
+ 2
= 3
```

- A `s1 = "+ 2"`, `s2 = "= 3"`
- B `s1 = ""`, `s2 = "+ 2"`
- C `s1 = "= 3"`, `s2 = ""`

QuickPoll: What are s1 and s2 after:

```
Scanner kbd = new Scanner(System.in);
int n       = kbd.nextInt();
String s1   = kbd.nextLine();
String s2   = kbd.nextLine();
```

Console input (on 3 lines):

```
1
+ 2
= 3
```

- A s1 = "+ 2", s2 = "= 3"
- B s1 = "", s2 = "+ 2"
- C s1 = "= 3", s2 = ""

Pitfall: Multiple Scanners

- A scanner reads a chunk of input at a time
- Holds onto it until it is requested
- If you create multiple scanners, each will hold onto some of the input
- You may get input in the wrong order or even lose some input
- Simple solution: only ever create one scanner in a program

Multiple Scanners Go Wrong

```
Scanner kbd1 = new Scanner(System.in);
Scanner kbd2 = new Scanner(System.in);
System.out.print("Enter two numbers: ");
int num1 = kbd1.nextInt();
int num2 = kbd2.nextInt();
System.out.print("Enter two numbers: ");
int num3 = kbd1.nextInt();
int num4 = kbd2.nextInt();
System.out.println(num1 + " * " + num2 +
                   " = " + num1*num2);
System.out.println(num3 + " * " + num4 +
                   " = " + num3*num4);
```

(Showing only the body of the method.)

Multiple Scanners Go Wrong

Behaviour of previous example

```
frege% java MultiScanner
Enter two numbers: 6 7
3 4
Enter two numbers: 6 * 3 = 18
7 * 4 = 28
```

Summary

- Use `+` to append strings
- `+` converts one operand to string if the other is a string
- `System.out.printf` does formatted output
- `args[`n`]` is the nth command line argument
- Use `java.util.Scanner` to read from the console
- Never create multiple `java.util.Scanner` objects

COMP90041
Programming and Software Development

Control

Semester 2, 2015

Control

- Java's control statements allow you to control execution of code
- Conditional statements determine which statements to execute, possibly bypassing some
- Loop statements repeat some statements some number of times, under programmer control
- Programmer writes the program; user runs it
- It's up to the programmer to control the program based on the situation, including user actions

If

- `if` statement decides whether or not to execute a statement based on a `boolean` expression
- Form:
  `if (`*`expr`*`)` *`Statement`*
- Executes the *`Statement`* if the *`expr`* is `true`, otherwise it does nothing
- The parenthesis are required
- The *`expr`* must be boolean
  - Use `!= 0` to test an int
- *E.g.*, negate `x` if it's negative:

```
if (x < 0) x = -x;
```

Compound Statements

- Most often, you need to execute multiple statements if the condition is true
- A compound statement turns multiple statements into a single statement that can be used in an `if`
- Also used in the other constucts in this lecture
- Form: { $Statement_1$; $\cdots$ $Statement_n$; }
- Don't follow the brace with semicolon
- This is a single statement that executes $Statement_1$; $\cdots$ $Statement_n$; in turn

```
if (x < 0) {
    x = -x;
    System.out.println(x + " is negative!");
}
```

Compound Statements

- Most often, you need to execute multiple statements if the condition is true
- A compound statement turns multiple statements into a single statement that can be used in an `if`
- Also used in the other constucts in this lecture
- Form: { $Statement_1$; $\cdots$ $Statement_n$; }
- Don't follow the brace with semicolon
- This is a single statement that executes $Statement_1$; $\cdots$ $Statement_n$; in turn

```
if (x < 0) {
    x = -x;
    System.out.println(x + " is negative!");Is it?
}
```

Compound Statements

- Most often, you need to execute multiple statements if the condition is true
- A compound statement turns multiple statements into a single statement that can be used in an `if`
- Also used in the other constucts in this lecture
- Form: { $Statement_1$; $\cdots$ $Statement_n$; }
- Don't follow the brace with semicolon
- This is a single statement that executes $Statement_1$; $\cdots$ $Statement_n$; in turn

```
if (x < 0) {
    System.out.println(x + " is negative!");
    x = -x;
}
```

Best Practice

- What's wrong with this?

```
if (x < 0)
    System.out.println(x + " is negative!");
    x = -x;
```

Best Practice

- What's wrong with this?

```
if (x < 0)
    System.out.println(x + " is negative!");
    x = -x;
```

- Best practice: always use braces, even for only one statement

```
if (x < 0) {
    x = -x;
}
```

Best Practice

- What's wrong with this?

```
if (x < 0)
    System.out.println(x + " is negative!");
    x = -x;
```

- Best practice: always use braces, even for only one statement

```
if (x < 0) {
    x = -x;
}
```

- Possible exception: whole `if` statement on one line
  - Unlikely to try to fit another statement on the same line

```
if (x < 0) x = -x;
```

If-Else

- Form:
  `if (`$expr$`)` $Statement_1$ `else` $Statement_2$
- Executes $Statement_1$ if the $expr$ is `true`, else executes `Statement`$_2$
- Always executes exactly one of the statements
- Also best practice to surround $Statement_1$ and $Statement_2$ with braces

Code Layout

- Always use indentation to show code structure
  - More indented code is part of less indented code
  - Indent one level per nesting level of braces
  - Not required by Java, but demanded by human readers
- One common layout:

```
if (x < 0)
{
    System.out.println("negative");
}
else
{
    System.out.println("non-negative");
}
```

Code Layout

- A more compact layout:

```
if (x < 0) {
    System.out.println("negative");
} else {
    System.out.println("non-negative");
}
```

- Amount to indent for each level:
  - 1 is too little; more than 8 too much
  - 4 is popular
- Beware of tabs: they mean different levels of indentation to different programs
  - 8 columns is standard
  - Best to avoid tabs altogether

Else If

- Java has no special form for handling a chain of conditions
  - Just nest one `if-else` in the `else` part of another

```
if (x < 0) {
    System.out.println("negative");
} else if (x == 0) {
    System.out.println("zero");
} else {
    System.out.println("positive");
}
```

- Nest `if` and `if-else` within one another to any depth
  - Braces also makes this easier to read

"Ternary Operator"

- Java also has an if-else expression:

$$expr_1 \ \texttt{?} \ expr_2 \ \texttt{:} \ expr_3$$

  - If $expr_1$ is `true` value is $expr_2$
  - If $expr_1$ is `false` value is $expr_3$

- This:

```
lesser = x < y ? x : y;
```

- does exactly the same as this:

```
if (x < y) {
    lesser = x;
} else {
    lesser = y;
}
```

Switch

- `switch` statement chooses one of several cases based on an `int`, `short`, `byte`, or `char` value
- As of Java 7, it can also be a `String`: more useful
- Form:

```
switch (expr) {
  case value1 :
    statements...
    break;
  ⋮
  case valuen :
    statements...
    break;
}
```

Switch

- Execution begins by evaluating the expression
- It then looks for a `case` with matching *`value`*
- If it finds one, it begins executing with the next statement
- It stops executing when it reaches a `break` or the end of the `switch`
- Cases can be put in any order

Default

- As a special case, can use `default` in place of one `case` *`value`*
- If no `case` *`value`* matches, the code after the `default:` is executed, up to the next `break;`
- If no `case` *`value`* matches and there is no `default:`, `switch` statement finishes without executing any of the statements

Pitfall: Missing `break`

- If there is no `break` before the next `case` label, Java keeps executing until the next break
- **Very** easy to forget a `break`
- Best practice: even put `break` at end of last case
  - You may later add a new case after the last one
- If you leave out a `break` on purpose, put in a comment saying why
  - So whoever reads code (including you, later) knows it was omitted on purpose
- Exception: same code for multiple cases: just put common `case` labels together, followed by code

Example: Spell Out Morse Code

```
switch (ch) {
case '.':
    System.out.print("dot ");
    break;
case '-':
case '_':
    System.out.print("dash ");
    break;
case ' ':
    System.out.println(); // start new line
    break;
default:
    System.out.println("\nbad character '" + ch + "'");
    break;
}
```

While

- Form:
  `while (`*expr*`)` *Statement*
- If *expr* is `true` then:
  - Execute the *Statement*, then
  - Then go back and check *expr* again
  - Keep executing *Statement* as long as *expr* is true
- Stops when *expr* is `false` at top of loop
- Use to execute *Statement* an unlimited number of times, as long as *expr* is true
- Only useful if *Statement* can change value of *expr*
- Best practice again: put *Statement* in braces unless whole `while` fits on one line

`while` Example

```
public class whileExample {
    public static void main(String[] args) {
        int i = 1;
        int limit = 10;
        int sum = 0;
        while (i <= limit) {
            sum += i;
            ++i;
        }
        System.out.println("The sum is " + sum);
    }
}
```

Generated output

```
The sum is 55
```

QuickPoll: What Will This Print?

```
int x=3, y=0;
while (x >= 0) {
    y++;
    x--;
}
System.out.println(y);
```

- A 0
- B 1
- C 2
- D 3
- E 4

QuickPoll: What Will This Print?

```
int x=3, y=0;
while (x >= 0) {
    y++;
    x--;
}
System.out.println(y);
```

- A 0
- B 1
- C 2
- D 3
- E 4

Do While

- Form:
  `do` *Statement* `while (`*expr*`)`
- First execute *Statement*
- Then, if *expr* is `true`, go back and do it again
  - Keep executing *Statement* as long as *expr* is true
- Stops when *expr* is `false` at bottom of loop
- Use when you must execute *Statement* before testing *expr*
- Only useful if *Statement* can change value of *expr*
- Best practice again: put *Statement* in braces unless whole `while` fits on one line

do while Example

```
public class dowhileExample {
    public static void main(String[] args) {
        int i = 1;
        int limit = 10;
        int sum = 0;
        do {
            sum += i;
            ++i;
        } while (i <= limit);
        System.out.println("The sum is " + sum);
    }
}
```

Generated output

```
The sum is 55
```

So What's The Difference?

- `while` executes *Statement* zero or more times
- `do while` executes *Statement* one or more times
- Use `while` if you need to check a condition every time before executing the *Statement*
- Use `do while` if you need to execute the *Statement* before evaluating the *expr* every time
- Changing `limit` to 0 in the `while` example will print a sum of 0
- Changing `limit` to 0 in the `do while` example will print a sum of 1! That's wrong!
- `while` is more commonly used

For

- `for` is like `while` with initialisation and increment
- Form:

  `for (`*init* `;` *test* `;` *update*`)` *Statement*
- *init* is for variable initialisations, *e.g.*, `x = 0`
- *test* is a boolean expression to decide whether to execute `Statement`
- *update* is executed after each iteration
- Useful to execute a specific number of iterations
- Equivalent to:

```
init;
while(test) {
    Statement;
    update;
}
```

for Example

```
public class forExample {
    public static void main(String[] args) {
        int limit = 10;
        int sum = 0;
        for (int i = 0; i <= limit; ++i) {
            sum += i;
        }
        System.out.println("The sum is " + sum);
    }
}
```

Generated output

```
The sum is 55
```

For

- Any of *init*, *test* and *update* parts can be omitted
  - Infinite loop if *test* is omitted, but see below
- Variables declared in *init* part are scoped to the `for`: not available after the loop
- But you can declare variable before loop, and just initialise it in the *init* part
- Can include multiple initialisations and updates by separating them with commas
  - But if you put a declaration in the *init* part, you can only specify one type (not so useful)
- Only one *test* part is allowed, but can use `&&` and `||` to define it

`break` and `continue`

- Inside a `for`, `while` or `do while` loop, a `break` terminates the (innermost) loop immediately
- This is useful inside an `if` inside a loop
- A `continue` statement immediately returns to the top of the innermost loop and continues from there
- Can immediately exit whole program with `System.exit(0);` statement
  - Use 0 to indicate "success" and > 0 to indicate error
  - Will see a better way to handle errors later...

Pitfall: Common Loop Errors

- Infinite loop: loop never terminates
  - Forget to update the counter
  - Use wrong test
- Best practice: use `<` or `<=` (or `>` or `>=`) in loop test, not `=` or `!=`
- Off-by-one (fence post) error: one too many or few iterations
  - Start or end too low or too high
  - Use `<` instead of `<=` or vice-versa
- For `n` iterations, do one of:
  - `for (i=0 ; i<n ; ++i)` or
  - `for (i=1 ; i<=n ; ++i)`

`assert`

- Use `assert(`*`test`*`)` to sanity-check your code
- Often program errors go undetected for a long time
- Very difficult to trace symptom back to cause
- Worst thing a program can do is not crash, but run normally producing wrong results
- `assert` stops the program if something is wrong
- *E.g.*, if at some point `x` must always be positive, add this statement at that point:

```
assert x > 0;
```

- Assertions not normally checked
  - Turn on checking by running program with: `java -enableassertions` *`ProgramName`*

Summary

- Use `if` or `if else` or `switch` to conditionally execute a statement
- Enclose multiple statements in {braces} to treat as a single statement
- Remember: end each `switch` case with a `break`
- Use `while` or `do while` or `for` loops to repeat a statement
- Use `break` to terminate loop immediately
- Use `contine` to restart a loop immediately

COMP90041
Programming and Software Development

Methods

Semester 2, 2015

Methods

- A method is an operation defined by a class
- *I.e.*, it defines how to do something
- The Java library defines many methods
- And you can define your own
- Similar to functions, subroutines, procedures in other languages
- Java supports two kinds of methods:
  - Class or `static` methods, and
  - Instance or non-`static` methods
- Instance methods are more common, but Class methods are simpler, so we start there

Using Class Methods

- Calling a class method runs the code in the body of the method before returning to execute the code following the method call
- Form: *class*.*method*(*expr1*, *expr2*, ...)
- Can omit *class*. if caller defined in same class
- The `expr`s, called arguments, provide data for the method to use
- Arguments are evaluated before executing method
- On completion, the called method can return a value to the caller
- The caller can then use the returned value in further computations

The Math Class

The `Math` class is a library of class methods and constants, including (among many more):

Method	**Type**	**Description**
`abs(int)`	`int`	absolute value
`ceil(double)`	`double`	ceiling
`E`	`double`	$e = 2.71828\ldots$
`max(int, int)`	`int`	larger of two ints
`min(int, int)`	`int`	smaller of two ints
`floor(double)`	`double`	floor
`PI`	`double`	$\pi = 3.14159\ldots$
`pow(double,double)`	`double`	a^b
`sqrt(double)`	`double`	$\sqrt{a}$

Example Method Use

`sqrt` is a class method to compute square root

```
import java.util.Scanner;
public class Hypot {
    public static void main(String[] args) {
        System.out.print("Enter triangle sides: ");
        Scanner kbd = new Scanner(System.in);
        double side1 = kbd.nextDouble();
        double side2 = kbd.nextDouble();
        double hypot = Math.sqrt(side1*side1 +
                                 side2*side2);
        System.out.println("Hypot = " + hypot);
    }
}
```

Example Method Use

Program Use

```
nomad% java Hypot
Enter triangle sides: 3 4
Hypot = 5.0
```

Wrapper Classes

- Many other classes provide class methods
- For each primitive type, there is a wrapper class that defines some useful class methods:

Primitive	**Wrapper**
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`boolean`	`Boolean`
`char`	`Character`

Convert from Primitive to String

Wrapper classes provide methods to convert primitive types to strings

Type	**Convert to** `String`
`byte x`	`Byte.toString(x)`
`short x`	`Short.toString(x)`
`int x`	`Integer.toString(x)`
`long x`	`Long.toString(x)`
`float x`	`Float.toString(x)`
`double x`	`Double.toString(x)`
`boolean x`	`Boolean.toString(x)`
`char x`	`Character.toString(x)`

Parsing Numeric Strings

Command line arguments are always strings.
To convert to number types:

Type	**Convert from** `String s`
`byte`	`Byte.parseByte(s)`
`short`	`Short.parseShort(s)`
`int`	`Integer.parseInt(s)`
`long`	`Long.parseLong(s)`
`float`	`Float.parseFloat(s)`
`double`	`Double.parseDouble(s)`
`boolean`	`Boolean.parseBoolean(s)`
`char`	`s.charAt(0)`

The `main` Method

- `main` is a class method we've been defining
- Java executes the `main` method when running an application
- Begins with:

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

- Ends with:

```
}
```

Defining Class Methods

- General form (for now):

```
public static type name(type1 var1,type2 var2,...) {
    ⋮
}
```

- Each *var* is called a parameter
- Parameter is a variable initialised to value of corresponding expression in method call
- Then body (⋮ part) of method is executed
- Types of corresponding arguments and parameters must match
- *type* is type of result returned

Returning Results

- Return result of method with `return` statement
- Form: `return` *expr*
- Value of expression *expr* is result of method
- Method completes as soon as `return` statement is reached (even if there is code after)
- Type of *expr* must match *type* of method
- Can have multiple `return` statements, but every execution of method <u>must</u> reach a `return` (but see exception below)
- Compiler error if either is violated
- Can use `return` to terminate loop <u>and</u> return from method

Example Method Definition

```
public class Hypot2 {
    public static void main(String[] args) {
        double side1 = Double.parseDouble(args[0]);
        double side2 = Double.parseDouble(args[1]);
        double hypot = hypot(side1, side2);
        System.out.println(hypot);
    }

    public static double hypot(double side1,
                               double side2) {
        return Math.sqrt(side1*side1 + side2*side2);
    }
}
```

Example Method Use

Program Use

```
nomad% java Hypot 3 4.0
5.0
```

- Original Hypot program was interactive: it asked for input it needed and explained its output
- This version takes all input on the command line
- Interactive version is more user-friendly
- This version is more machine-friendly: easier for another program to control
- You will be asked to write many machine-friendly programs for this reason

QuickPoll: What will this return?

```
public static int test() {
    int i = 0, j = 2;
    while (j>0) {
        --j;
        if (i >= 2) return j;
        ++i;
    }
    return i;
}
```

- A 0
- B 1
- C 2
- D 3
- E Compiler error: bad returns

QuickPoll: What will this return?

```
public static int test() {
    int i = 0, j = 2;
    while (j>0) {
        --j;
        if (i >= 2) return j;
        ++i;
    }
    return i;
}
```

- A 0
- B 1
- C 2
- D 3
- E Compiler error: bad returns

When to Define Methods

- In this example, we turned a chunk of code (hypotenuse computation) into a new method
- This is called refactoring
- When to define a new method:
  - When a method gets too big (more than can be easily viewed at once, more than $\approx$ 60 lines)
  - When you repeat similar code multiple times
  - When you can give a good name to a chunk of code (*e.g.*, `hypot`)
- How to break up the work of a program into methods is an important and complex issue
- We will revisit later

Not Returning Results

- Use method call as an expression
- Value of expression is value returned by method
- But can also use method call as statement
- Ignore returned value, just execute for effect (*e.g.*, printing)
- If always want to ignore, use return type `void`
- Means don't return anything
- `main` method has return type `void`
- Then don't need `return` statement
- Can use `return` with no expression to immediately return nothing

Headers and Signatures

- First part of method definition (up to {) is called the method header
- Header defines return type, method name, number and types of parameters, and parameter names
- Method name plus number and types of arguments together are called the method signature
- Signature is used to decide which method to call

Overloading

- `abs`, `min`, and `max` all work on all of `double`, `float`, `int`, and `long` types
- They return the same types
- Overloading: when a method name has multiple definitions, each with different signature
- Java automatically selects the method whose signature matches the call
- You can define your own overloaded methods, too
  - Just define multiple methods with same name but different signatures
- Uses: support multiple types, simulate default arguments

Pitfall: Limitations of Overloading

- You cannot define two methods with the same name and all the same argument types
- You cannot overload based on return type, only parameter types
- You cannot overload operators (*e.g.*, +, *, *etc.*)
- Beware of combining overloading with automatic type conversion!

```
int    bad(int x, double y) {...}
double bad(double x, int y) {...}
```

What if you call `bad(6, 7)`?

`public` and `private`

- The keyword `public` in method header means the method may be used by any method in any class
- The keyword `private` in header means the method may only be used from within that class
- Visibility: from where method can be seen
- `public` implicitly promises to (try hard to) maintain that method without changing its signature
- Best practice: make methods private unless they need to be public
- Best practice: make signature of public methods as simple as possible

Local Variables

- Variables declared inside methods are local to the method
- Variable only exists once declaration is executed
- Value is forgotten once method returns
- Value declared inside a block is forgotten once execution of block is finished
- Local variables can only be referred to inside the method or block in which they are declared
- Parameters are also local variables declared in a method
- Cannot be declared `public` or `private`

Defining Constants

- `Math` class defines constants `PI` and `E`
- You can define your own constants for your code
- Form:

  `public static final` *`type name`* `=` *`value`* `;`
- At top level of class declaration
- Can be `public` or `private`
- Java naming convention: all uppercase, words separated with underscores
- *E.g.*:

```
public static final int DAYS_PER_WEEK = 7;
public static final int CARDS_PER_SUIT = 13;
```

When to Define Constants

- Best practice: don't sprinkle mysterious numbers in your code, define constants instead
- Makes code <u>much</u> easier to understand
- What does this do?

```
x += 168;
```

When to Define Constants

- Best practice: don't sprinkle mysterious numbers in your code, define constants instead
- Makes code much easier to understand
- What does this do?

```
x += 168;
```

Compare that with:

```
x += DAYS_PER_WEEK * HOURS_PER_DAY;
```

When to Define Constants

- Best practice: don't sprinkle mysterious numbers in your code, define constants instead
- Makes code much easier to understand
- What does this do?

```
x += 168;
```

Compare that with:

```
x += DAYS_PER_WEEK * HOURS_PER_DAY;
```

- Also symbolic constants defined in one place are much easier to change if necessary, eg:

```
private static final int CHARS_IN_SUBJECT_CODE = 6;
```

When Not to Define Constants

- Don't define a constant for something you can't meaningfully name
- This is useless:

```
public static final int SEVEN = 7;
```

- Don't (usually) define a name for 0 or 1: `n == 0` is just as good as `n == NONE`

Class Variables

- Class variable is a variable that is local to a class
- Created when program starts, exists until exit
- Value survives through message calls and returns
- Value is unchanged until reassigned
- Only one "copy" of each class variable at a time
- Can be declared either `public` or `private`
- It should almost always be `private`
    - Too difficult to control if every method in every class can modify it
- Use sparingly: better to use local variables if possible

Class Variables

```
public class ClassVar {
    private static String name = "Someone";
    public static void main(String[] args) {
        greet("Hello");
        setName("Kitty");
        greet("Hello");
        greet("Aloha");
    }
    private static void setName(String name) {
        ClassVar.name = name; // 2 vars called name!
    }
    private static void greet(String greeting) {
        System.out.printf("%s, %s!%n", greeting, name);
    }
}
```

Example Method Use

Program Output

```
Hello, Someone!
Hello, Kitty!
Aloha, Kitty!
```

- This program is stateful: behaviour depends on what has come before
- `greet("Hello")` does two different things!
- Makes it harder to predict program behaviour
- Keep use of class variables to a minimum

Summary

- Executing class (static) methods executes their definition
- Method calls can pass arguments
- Methods can can return a value with a `return` statement
- Methods declared with `private` instead of `public` can only be used inside the same class
- Multiple methods can have same name, if number/types of arguments are different

COMP90041
Programming and Software Development

Immutable Objects

Semester 2, 2015

Objects

- Object oriented programming is centered around creating and using objects
- Each object is an instance of some class
- An object holds some data (state) and supports some operations
- *I.e.*, an object combines code and data
- An object represents some thing
  - Either a physical real world thing, like a student or a book or an airplane
  - Or a more abstract real world thing, like a university subject or a library loan or a flight
  - Or a still more abstract thing used for the purposes of a program, like a list or a string

Creating New Objects

- Java has special syntax for instantiating a class (creating a new object)
- Form: `new` *`Class`*`(`*`expr1`*`, `*`expr2`*`, ...)`
- This is an expression returning a fresh object
- Often used as:
  `Class var`` = new `*`Class`*`(`*`expr1`*`, `*`expr2`*`, ...)`
- We saw this for creating a `Scanner` object :
  `Scanner kbd = new Scanner(System.in);`
- The appropriate arguments are determined by the class's constructors (descussed below)
- You don't need to create `"constant strings"` — Java creates them for you

Using Objects

- Two ways to use an object:
  - Send a message to it
  - Pass it as an argument of a message to another object
- Form: `object.msg(expr1, expr2, ...)`
- Note the `expr` arguments can be objects
- We saw both uses of objects in our first program:

```
System.out.println("Hello, World!");
```

  - `System.out` is a `PrintStream` object connected to the console
  - `"Hello, World!"` is a constant `String` object
- The allowed messages (`msg`) are determined by the class of the object (`object`) receiving the message

`null`

- `null` is a special value that means "there is no object here"
- Java lets you use it wherever an object can be used
- But is not an object, and causes an error if you use it as one
  - You cannot send a message to it, or access its instance variables (it has none)
- Be sure a variable is not `null` before sending it a message, *e.g.*:

```
String s = ...
if (s != null) {
    System.out.println("the string is " + s);
}
```

Modelling

- Software modelling: determining an appropriate set of classes and methods to solve a problem
- Best practice: design classes around the "entities" of your program (concrete or abstract)
  - What characteristics are different for different individuals?
  - What can they do or have done to them?
  - Ignore what is irrelevant to your application
- Best practice: Keep It Simple
- These sometimes conflict: use judgement

Modelling Example: Reuniting Loved Ones

- Problem: in a natural disaster, people may be separated from their loved ones
- Software solution, each person registers where they are, and when, and who they're looking for
- Need to model <u>person</u>, <u>location</u>, and <u>date/time</u>
- Person has a name and location
- People may come and go, so need to know when last seen at that location
- Represent name as a single string? Separate given and family name? Include middle name? Title?
- Similarly for location and date
- Go with simplest design that gives the information you need

Modelling Operations

- Consider what the application needs to do with an object
- In reuniting example, person enters their name and location in system, and the name of their missing loved one
- Loved one in other location does the same
- So program needs to compare names of people to match them up
- Need a fuzzy match, eg, "Bill" = "William"
- Need to report person's location and time when a match is found

Classes

- Implement a design by defining classes
- Classes contain:
  - Instance variables, which hold the data of an object
  - (Instance) methods, which define the operations (code) of an object
- A class is a type, which can be used to declare variables that hold instances of that type
- Each object has its own value for each instance variable
- All objects of a class have the same methods
- Each method of a class specifies an operation on that class and how it should behave

Declaring Instance Variables

- Declare instance variable inside class, but outside any method
- Form: `private` *`type`* *`name`*`;`
- Like class variable declaration without `static`
- Declares instance variable *`name`* holding value of type *`type`*
- *E.g.*:

```
public class Person {
    private String familyName;
    private String givenName;
    ⋮
}
```

Make Instance Variables Private

- It's possible to declare instance variables `public` instead of `private`
- But **don't**
- Making an instance variable `public` would allow any method of any class to modify it
- The class would lose control of its own data!
- Instead use methods to access instance variables...

Assigning and Using Instance Variables

- Inside a class's methods, its instance variables can be assigned and used like other variables: by name
- Local variables live in a method; class variables live in a class; instance variables live in an object
  - local variables only live while method is executing
  - instance variables live as long as object is around
  - class variables live as long as the program is running
  - local variables must be initialised every time method executes
  - instance variables must be initialised when declared or when object is created
  - class variables must be initialised where declared, which is executed when program starts
- If local and instance variables with same name, local variable "wins"

Declaring (Instance) Methods

- Declare instance (non-static) methods inside class, but outside any method
- Form:

```
vis type name(type1 name1,...) {
    ⋮
}
```

- Like class method declaration without `static`
- *vis* can be `public` (use from any class) or `private` (use only from same class)
- Declares instance method *name* returning value of *type*, taking arguments of type *type1*, ...

Example Method

```
public class Person {
    private String familyName;
    private String givenName;
    ⋮
    public String getFullName() {
        return familyName + ", " + givenName;
    }
    ⋮
}
```

- Each `Person` object has its own `familyName` and `givenName`
- given a `Person` object `p`, `p.getFullName()` returns that person's full name as a string

QuickPoll: What does this print?

```
public class QP0501 {
    private int x=1, y=2;
    public int m(int y) { return x + y; }
    public static void main(String[] args) {
        QP0501 ob = new QP0501();
        System.out.println(ob.m(3));
    }
}
```

- A 3
- B 4
- C 5
- D Won't compile: `m` cannot access `x`
- E Won't compile: two different `y`s

QuickPoll: What does this print?

```
public class QP0501 {
    private int x=1, y=2;
    public int m(int y) { return x + y; }
    public static void main(String[] args) {
        QP0501 ob = new QP0501();
        System.out.println(ob.m(3));
    }
}
```

- A 3
- B 4
- C 5
- D Won't compile: `m` cannot access `x`
- E Won't compile: two different `y`s

Accessors (Getters)

- If you declare instance variables `private`, how can you access them?
- An accessor (getter) is a very simple method written for this purpose
- Java convention: name accessor after instance variable, with `get` on the front

```
public String getFamilyName() {
    return familyName;
}
```

- Only write accessors for instance variables you really need other classes to be able to access

Constructors

- When creating an object, its instance variables need to be initialised to appropriate values
- Constructors are special methods responsible for this
- Form:

```
public ClassName(type1 var1,...) {
    ⋮
}
```

- Constructors always have same name as class
- And never specify a return type, not even `void`

Example Constructor

- The constructor body can use the parameters to assign the instance variables

```
public Person(String f, String g) {
    familyName = f;
    givenName = g;
}
```

- Why such cryptic parameter names?
- We'll see how to use better parameter names soon. . .

Assigning and Using Instance Variables

- Instance variables can be assigned and used just like local variables
- If an instance variable and a local variable have the same name, the local variable “wins”
- Can use an alternative form: `object.name`
- Refers to the `name` instance variable of `object`
- Works if `object` is an instance of the class this code appears in
- ...or if `name` is a public instance variable (which it shouldn't be)

this

- Inside a method, special keyword `this` always holds the object the current message was sent to
- An implicit parameter to instance methods
- *E.g.*, if message was `foo.bar(baz)`, while executing method `bar`, `this` is `foo`
- Inside constructor, `this` is the object being created

```
public Person(String givenName,
              String familyName) {
    this.givenName = givenName;
    this.familyName = familyName;
}
```

Null Pointer Exception

- If you forget to check that an object is not `null` before sending it a message or accessing its instance variables, you will get a null pointer exception
- This aborts (crashes) your program

```
Exception in thread "main" java.lang.NullPointerExcepti
on
        at Tst.main(Tst.java:4)
```

- But methods don't need to check that `this` is not `null`: if it were, the method would not run
- If you checked a variable, don't need to check again unless it could have been changed

Immutable Objects

- Immutable objects cannot be changed once they are created
- An Immutable class is one all of whose instances are immutable
- To be immutable, a class must not have any public methods that modify the object (just constructor)
- In Java, the `String` class is immutable
- Primitive types are also immutable
- Bad Things can happen when you modify objects (as we will see)
- Best practice: make classes immutable if possible

Finality

- Java can help you make classes immutable
- Add `final` keyword to instance variables before type
- Then Java will only let you set it once

```
public class ImmutablePersion {
    private final String familyName;
    private final String givenName;

    public ImmutablePerson(String familyName,
            String givenName) {
        this.familyName = familyName;
        this.givenName = givenName;
    }
  ⋮
```

Finality Protects You from Yourself

Adding this method:

```
    public void bogus(String s) {
        this.familyName = s;
    }
```

gives this compiler error:

```
ImmutablePerson.java:12: error: cannot assign
 a value to final variable familyName
        this.familyName = s;
            ^

1 error
```

Pitfall: Class *vs.* Instance Messages

- An instance message is sent an individual object
- A class message is sent to the whole class
  - Not to any individual object
- So a class method does not have (cannot use) `this`
- Class method cannot access instance variables!
- Class methods cannot use instance methods (without specifying *`object.`* in front)
- but instance methods <u>can</u> use class methods

The `toString` Method

- *E.g.*, if `p` is an `ImmutablePerson`, what should `"Welcome, " + p` produce?
- What should `System.out.print(p)` print?
- You can determine how Java converts instances of your class to a String
- Define a public method `String toString()`

```
public String toString() {
    return givenName + " " + familyName;
}
```

Summary

- Use `new` *`Class(...)`* to create object
- Use *`object.methodName(...)`* to send message to *`object`*
- Can use `null` in place of an object, but don't send message to it!
- Declare instance variables and methods like class ones, without `static` keyword
- Always make instance variables private
- Write constructor to initialise instance variables
- Make classes immutable when you can

COMP90041
Programming and Software Development

Mutable Objects

Semester 2, 2015

Program Testing

- It is very difficult to write correct software, even for experienced programmers
- Even software that seems to work OK often harbours hidden bugs
- Flawless software often grows bugs as it is modified
- You must test your programs thoroughly
- Plan how to test your program as (or even before) your write it

The Psychology of Testing

- Programmers tend to test their code with the goal of not finding any defects
- Testers test the code with the goal of uncovering defects
- A successful test is one that detects an unknown defect
- Many software houses have separate QA teams for that reason
- You don't have that luxury

How to Test

- Unit testing: testing that individual classes and methods behave correctly
- For each aspect of the specification, write code to establish the conditions of that aspect, and check that the result is as expected
- Regression testing: running a suite of tests whenever you modify your code, looking for new bugs
- Ensure that corner cases (situations where things happen differently than usual) are tested
- Where there are boundary conditions, test just before, at, and just after the boundary
- Write extra tests for the most complicated aspects

Mutable Objects and Classes

- Instance variables not declared `final` can be modified by methods
- Objects and classes that permit modification are mutable
- They are common in object-oriented programming
- Immutable objects can be thought of as values, like primitive values
- Mutable objects must be understood in terms of their behaviour in memory

Memory

- Computer memory is a long sequence of bytes
- Computer can access any byte by its address — its position in the sequence
- A `short` is stored over 2 bytes, an `int` over 4, *etc.*
- Computer reads multiple bytes at once and assembles into a `short`, `int`, `double`, *etc.*

0000	23
0001	41
0002	97
0003	07
0004	41
0005	76
0006	00
0007	71
.	.
.	.
.	.

Objects

- An object is stored in memory by storing its instance variables
- An object at memory address 3270 with 2 `int` variables holding 5 and 10 might appear in memory as at right
- Actual values of a byte are 0–255 or -128–127, not 0–99
- Actual addresses are much larger than this illustration

3270	00
3271	00
3272	00
3273	05
3274	00
3275	00
3276	00
3277	10
.	.
.	.
.	.

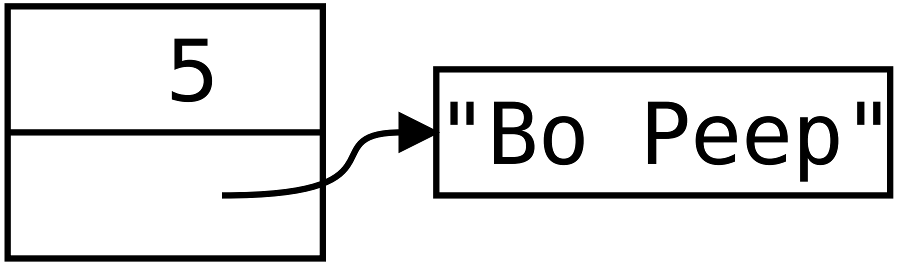
References (Pointers)

- Object types are stored by storing their address
- *E.g.*, to store a `String`, a variable holds its address, not its content
- We show the string stored at 7192, but it could be anywhere in memory
- *E.g.* an object with `int` and `String` variables holding 5 and `"Bo Peep"` would be stored as at right
- The address of an object is called a reference or pointer to it
- Classes are called reference types

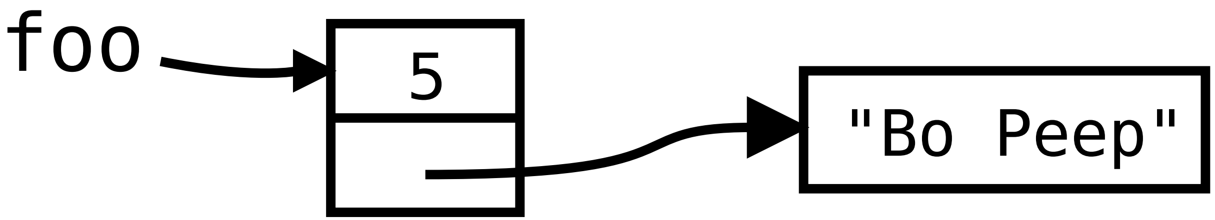
3270	00	
3271	00	
3272	00	
3273	05	
3274	00	
3275	00	
3276	71	
3277	92	
⋮	⋮	
7192	66	B
7193	111	o
7194	32	
⋮	⋮	⋮

Depicting Objects

- Such an object is more abstractly viewed as



- If a variable `foo` holds (a reference to) this object, it could be depicted as



Changing objects

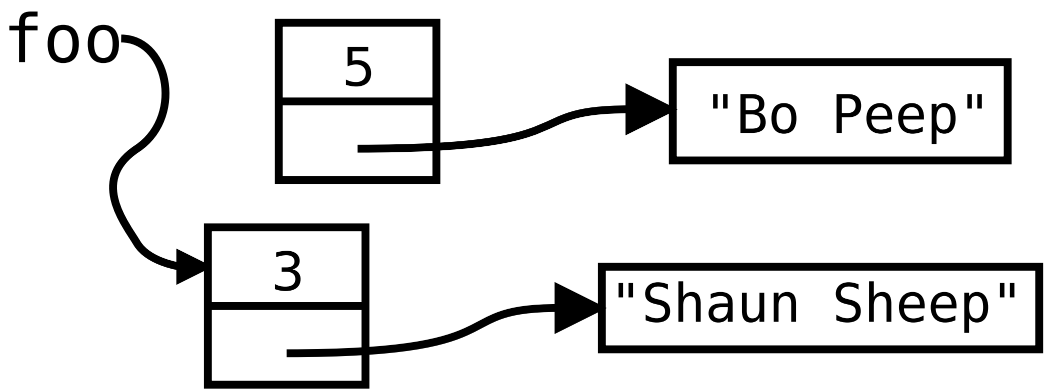
```
public class Person
    private int age;
    private String name;
    public Person(int age, String name) {
        this.age = age; this.name = name;
    }
    public String toString() {
        return name + ", age " + age;
    }
    public static void main(String args[]) {
        Person foo = new Person(5, "Bo Peep");
        ⋮
    }
}
```

Changing objects

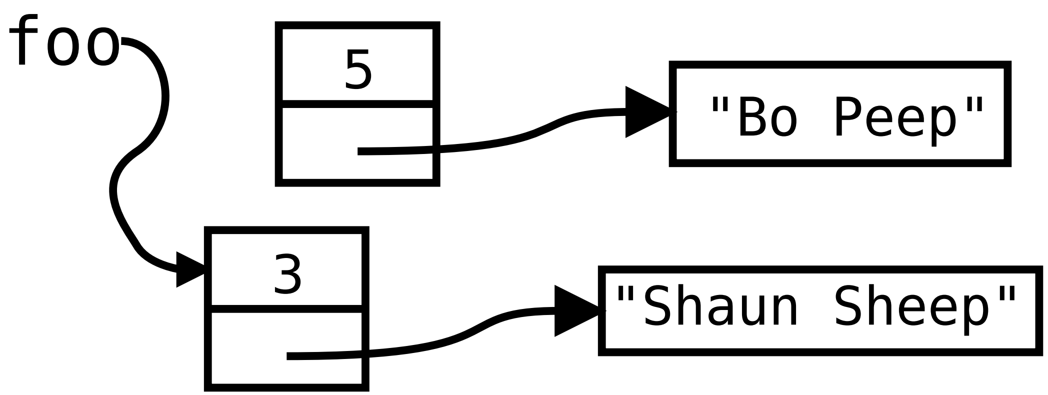
- With immutable `Person` class, only way to change the object `foo` points to is assign a new object

```
public static void main(String args[]) {
    Person foo = new Person(5, "Bo Peep");
    foo = new Person(3, "Shaun Sheep");
}
```

After second assignment:



Garbage Collection



- No variable (or object) points to original object
- Can never be used again, as no way to get to it
- Java has a garbage collector (GC) that “reclaims” such objects so the memory can be used again
- No need to worry about it, it all happens automatically

Mutators

- If instance variable is not declared `final`, methods can reassign it
- Simplest such method takes new value for instance variable as argument and assigns it
- This is called a mutator or setter
- Convention: for instance variable *instanceVar*, name mutator set*InstanceVar*

```
public void setAge(int age) {
    this.age = age;
}
```

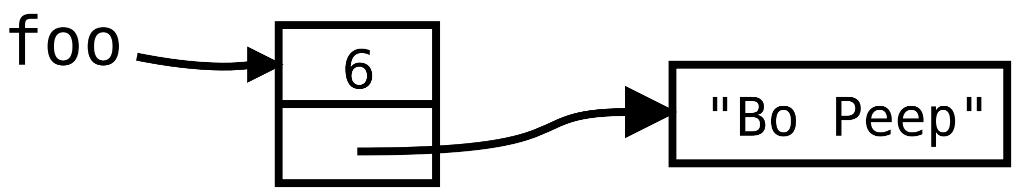
- Using a mutator modifies the existing object, without making a new one

Mutating objects

Mutator gives a second way to change the object referred to by `foo`: mutate it

```
public static void main(String args[]) {
    Person foo = new Person(5, "Bo Peep");
    foo.setAge(6);
}
```

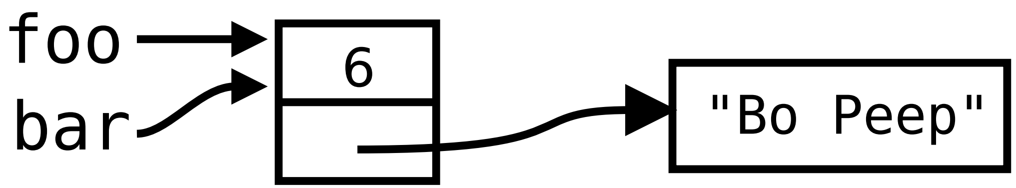
After second statement, only one object has been created, but it has been mutated:



Mutating aliased objects

- If multiple variables refer to same object, they are called aliases
- Changing one alias changes all of them

```
public static void main(String args[]) {
    Person foo = new Person(5, "Bo Peep");
    Person bar = foo;
    foo.setAge(6);
}
```



QuickPoll: What does this print?

If this is in the Person class:

```
public static void main(String args[]) {
    Person foo = new Person(5, "Bo Peep");
    Person bar = foo;
    foo.setAge(6);
    System.out.println(bar.age);
}
```

A. 5
B. 6
C. Won't compile: age is private
D. Won't compile: can't put main method in Person class

QuickPoll: What does this print?

If this is in the Person class:

```
public static void main(String args[]) {
    Person foo = new Person(5, "Bo Peep");
    Person bar = foo;
    foo.setAge(6);
    System.out.println(bar.age);
}
```

- A 5
- B 6
- C Won't compile: age is private
- D Won't compile: can't put main method in Person class

QuickPoll: What does this print?

```
public static void main(String args[]) {
    int foo = 5;
    int bar = foo;
    foo = 6;
    System.out.println(bar);
}
```

- A 5
- B 6

QuickPoll: What does this print?

```
public static void main(String args[]) {
    int foo = 5;
    int bar = foo;
    foo = 6;
    System.out.println(bar);
}
```

A. 5
B. 6

Setting a variable changes only that variable; mutating an object changes that object for every variable or object that references it

Class Invariants

- We carefully declared instance variables `private`, so other classes can't mess with them
- This allows us to ensure all instances have certain properties
- *E.g.*, may want to ensure we have a name and location for a person who is who is seeking a lost loved one
- Condition like this is called a class invariant

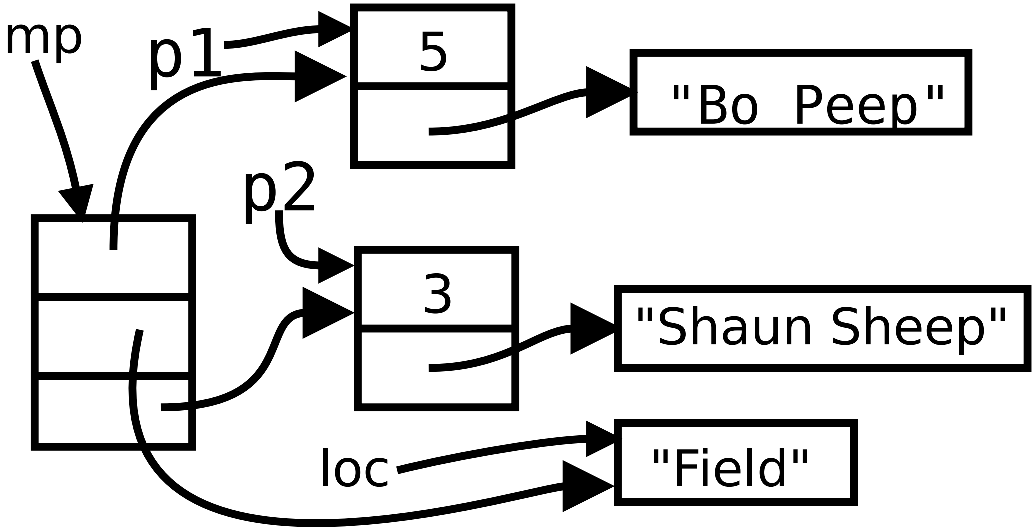
Class Invariants

```
public class MissingPerson {
   private Person seeker;
   private String location;
   private Person sought;
   public MissingPerson(Person seeker, String location,
                        Person sought) {
      if (seeker.getName().isEmpty()
          || location.isEmpty()) {
          System.out.println("Missing name/location");
          System.exit(1);
       }
       this.seeker = seeker;
       this.location = location;
       this.sought = sought;
   }
   ⋮
```

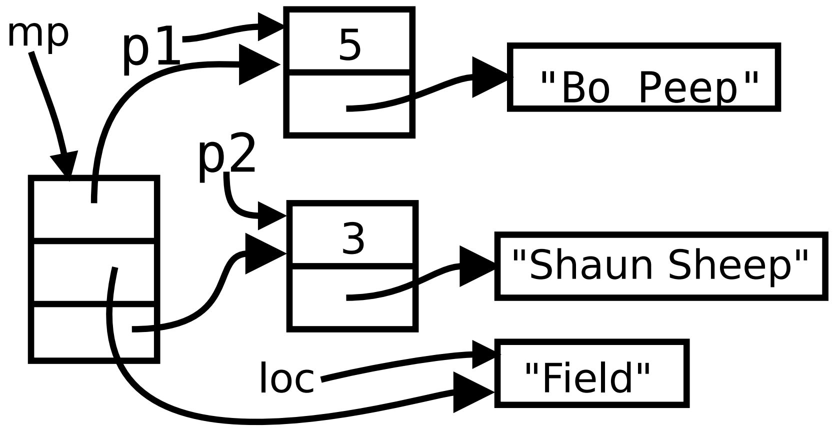
Privacy Leaks

But if an instance variable holds a mutable object and any method of another class can access that object, it can violate conditions of that object we want to ensure

```
Person p1 = new Person(5 "Bo Peep");
Person p2 = new Person(3 "Shaun Sheep");
MissingPerson mp = new MissingPerson(p1, "Field", p2);
```



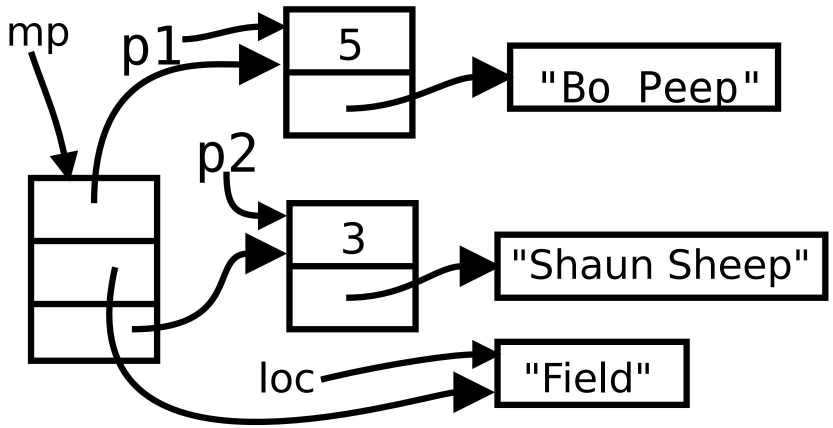
Privacy Leaks



What if that code is followed by this:

```
p1.setName("");
```

Privacy Leaks

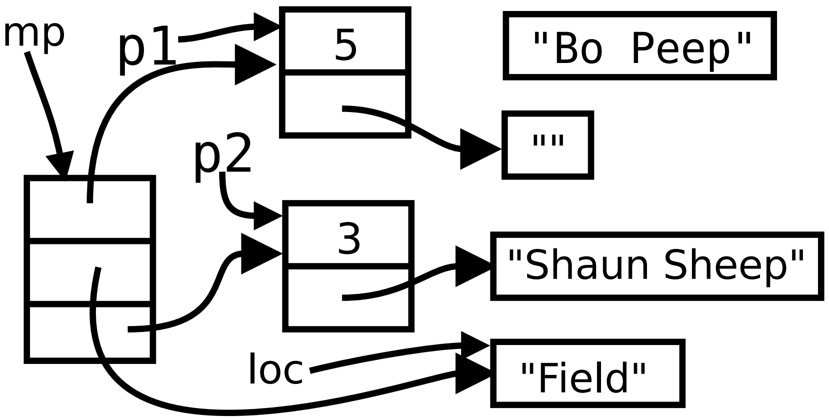


What if that code is followed by this:

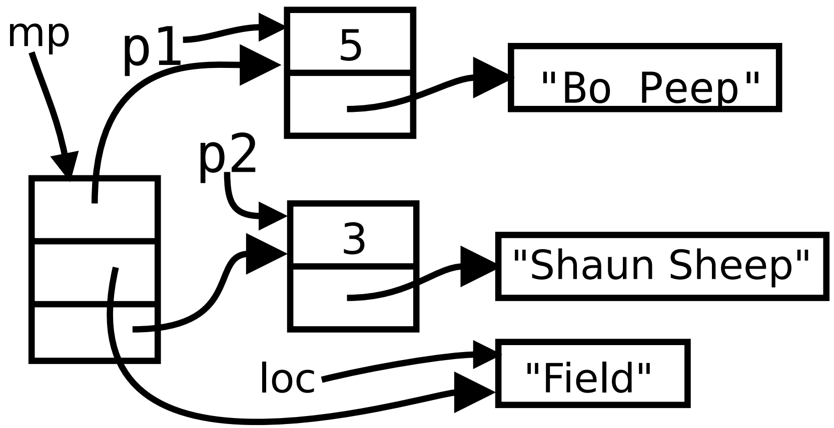
```
p1.setName("");
```

Now `seeker.getName().isEmpty()` is true for `mp`!

That violates our class invariant!



Shared mutable objects



- This problem occurs because `mp` shares its `seeker` with `p1`, and that object is mutable
- If either `p1` or `mp.seeker` is changed, the other changes, too (because they are the same object)
- Note `mp.location` shares with `loc`, but that is not a problem because strings are immutable

Shared mutable object Example

```
public static void main(String[] args) {
        Person p1 = new Person(5, "Bo Peep");
    Person p2 = new Person(3, "Shaun Sheep");
    MissingPerson mp =
            new MissingPerson(p1,"Field",p2);
    System.out.println(mp);
    p1.setName("Bad Wolf");
    p1.setAge(10);
    System.out.println(mp);
}
```

```
Bo Peep, age 5, at Field, seeks Shaun Sheep, age 3
Bad Wolf, age 10, at Field, seeks Shaun Sheep, age 3
```

Defensive Copying

- Solution is to either make `Person` class immutable, or prevent instance variables from sharing with any outside variables or objects
- Do this by copying objects passed in to methods and storing the copy instead of the original
- Same if mutable object stored in instance variable is returned by method: caller can modify it
- Must also copy objects stored in instance variables and return the copy instead of original object
- This is called defensive copying
- Commonly needed in constructors, getters, and setters

Copy Constructor

- For mutable classes, write a copy constructor: a constructor that takes one argument of the same type as the object being constructed
- Just makes the new object an exact copy of the input argument
- *E.g.*:

```
public Person(Person orig) {
    this.age = orig.age;
    this.name = orig.name;
}
```

Deep Copying

- If any instance variables are mutable classes, copy constructor must copy those objects, too
- This is called a deep copy: it does not share any mutable objects with the original object

```
public MissingPerson(MissingPerson orig) {
    this.seeker = new Person(orig.seeker);
    this.location = orig.location; // immutable!
    this.sought = new Person(orig.sought);
}
```

Plugging Privacy Leaks

```
public class MissingPerson {
   private Person seeker;
   private String location;
   private Person sought;
   public MissingPerson(Person seeker, String location,
                        Person sought) {
      if (seeker.getName().isEmpty()
          || location.isEmpty()) {
          System.out.println("Missing name/location");
          System.exit(1);
       }
       this.seeker = new Person(seeker);
       this.location = location;
       this.sought = new Person(sought);
   }
   ⋮
```

Plugging Privacy Leaks (2)

```
public MissingPerson(MissingPerson orig) {
    this.seeker = new Person(orig.seeker);
    this.location = orig.location; // immutable!
    this.sought = new Person(orig.sought);
}
public Person getSeeker() {
    return new Person(seeker);
}
public String getLocation() {
    return location; // immutable
}
public Person getSought() {
    return new Person(sought);
}
⋮
```

Plugging Privacy Leaks (3)

```
    public void setSeeker(Person seeker) {
        this.seeker = new Person(seeker);
    }
    public void setLocation(String location) {
      this.location = location; // immutable
    }
    public void setSought(Person sought) {
        this.sought = new Person(sought);
    }
}
```

With this change, `"Shaun Sheep"` is safe from the `"Bad Wolf"`

Summary

- Instances variables not declared `final` can be modified
- If public methods modify instance variables, the class is mutable
- Mutable classes are common, but dangerous
- When multiple variables/objects refer to an object, mutating the object changes it for <u>all of them</u>
- Privacy leak: when method of another class gets ahold of a mutable object stored in a private instance variable
- Solution: copy mutable object using copy constructor before storing or returning it

COMP90041
Programming and Software Development

Arrays

Semester 2, 2015

Arrays

- Classes allow design of a type holding any combination of values of any types
- But what if you want to store an arbitrary number of values of the same type?
- *E.g.*, the arbitrary number of command line arguments passed to the program
- These are passed as an array
- An array can be used like an arbitrary sequence of separate variables of the same type, but also as a single value

Declaring Arrays

- Declare a single variable to hold a whole array
- Form: `baseType[] varName`
- Where `baseType` is the type of every array element
- *E.g.*: `String[] args;`
- Use the same syntax to declare:
  - local variables
  - instance or class variables
  - method parameters
- Similar syntax to declare method that returns an array:
  `baseType[] methodName(type name,...)`
  (can be `public` or `private`, `static` or not)

Initialising Arrays

- As for class types, declaring a variable does not create an array
- Simplest way to get an array is initialise to a constant
- Form: *baseType*[] *varName* = {*value*,...};
- *E.g.*:

```
public static final int[] DAYS_IN_MONTH
    = {31,28,31,30,31,30,31,31,30,31,30,31};
```

- (Why make this `static final`?)
- But {...} is not an expression; this syntax can only be used in variable initialisations

Creating Arrays

- To create a fresh array, use special `new` syntax
- Form: `new` *`type`*`[`*`size`*`]`
- *E.g.*:

```
double[] data = new double[10];
```

- Note array type never includes size, but `new` expression always does
- Size can be specified as any integer-valued expression, so you can calculate the size
- You cannot specify initial values for array elements in `new` expression
  - Number elements are initialised to 0 or 0.0, `boolean`s to `false`, `char`s to `'\0'`, and objects to `null`

Assign and Use Array Elements

- Access array elements using `array[index]`
- *E.g.*: `args[2]`
- The value inside the brackets is called an array index
- Array indexes range from 0 to array size - 1
- Use the same syntax to assign to an array element
- *E.g.*: `data[2] = data[1] * 10;`

Pitfall: Array Index Out of Bounds

- Size of an array is fixed at the time it is created
- Arrays cannot be expanded (or shrunk)
- Accessing or assigning an array at an index < 0 or $\geq$ the size of the array causes an error:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBo
undsException: 0
        at Tst.main(Tst.java:3)
```

Indexing Arrays

- An array index can be specified as an expression
- Very common to index an array with a variable
- *E.g.*:

```
public static final int[] daysInMonth
        = {31,28,31,30,31,30,31,31,30,31,30,31};
public static void main(String[] args) {
    int month = Integer.parseInt(args[0]);
    int day = Integer.parseInt(args[1]);
    int yearday = 0;
    for (int i = 0; i < month - 1; ++i) {
        yearday += daysInMonth[i];
    }
    System.out.println(yearday + day);
}
```

Array `length`

- The size of an array is *array* `.length`
- *E.g.*:

```
public class ArgCount {
    public static void main(String[] args) {
    System.out.printf("You entered %d arguments%n",
        args.length);
    }
}
```

- Very useful for iterating over a whole array:

```
for (int i = 0 ; i < a.length ; ++i) ... a[i] ...
```

QuickPoll: What does this print?

```
int[] a = {1,1,2}
int sum=0;
for (int i=1; i<=a.length; ++i) sum += a[i];
System.out.println(sum);
```

A. 1
B. 2
C. 3
D. 4
E. There's a runtime error

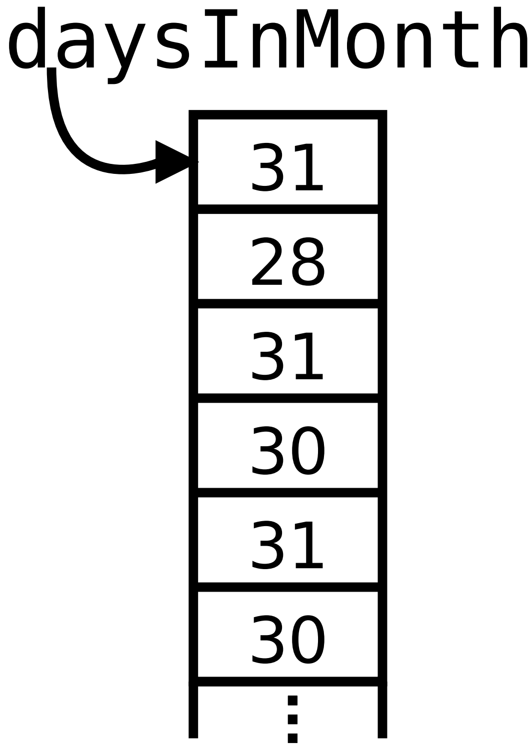
QuickPoll: What does this print?

```
int[] a = {1,1,2}
int sum=0;
for (int i=1; i<=a.length; ++i) sum += a[i];
System.out.println(sum);
```

- A 1
- B 2
- C 3
- D 4
- E There's a runtime error

Arrays are Mutable

- Arrays are reference types
- Similar to objects (but not quite the same)
- Stored in memory as a sequence of elements: efficient to index
- Unlike your own classes, array types cannot be made immutable
- Beware privacy leaks: copy arrays if necessary



Copying Arrays

- You cannot write a copy constructor for arrays
- but you can write a `static` method to copy an array, *e.g.*:

```
public static String[] copyStringArray(String[] orig){
    String[] copy = new String[orig.length];
    for (int i = 0; i<orig.length; ++i) {
        copy[i] = orig[i];
    }
    return copy;
}
```

- But you need to write a separate copy method for each base type of array you need to copy

Arrays of Objects

- Can declare array with a class as base type
- *E.g.*, `String[] args`
- creating an array of objects does not create any objects, just an array full of `null`s
- Be sure to initialise the array elements to new objects if necessary

```
Person[] people = new Person[args.length];
for (int i = 0; i < people.length ; ++i) {
    people[i] = new Person(0, args[i]);
}
```

(creates an array of `Person`s with age 0 and name taken from command line arguments)

Growing Arrays

- You cannot assign to an array's `length` "member" to resize it

```
arr.length = arr.length + 1; // compilation error!
```

- But it's easy to create a new, bigger array and copy
- To grow `arr` and add 42 at the end:

```
int[] tmp = new int[arr.length+1];
for (int i = 0; i < arr.length; ++i) tmp[i] = arr[i];
tmp[arr.length] = 42;
arr = tmp;
```

Partially Filled Arrays

- Copying an array to add one element to the end is expensive and inconvenient
- One common trick is the partially filled array: an array where the first elements are the intended elements, and the rest are meaningless
- A variable holds the number of meaningful elements
- Make effective array bigger by just incrementing the number of meaningful elements
- But when that reaches the size of the actual array, must copy to new, bigger array

Partially Filled Arrays

```
public class ExpandingIntArray {
    public static final int INITIAL_SIZE = 10;
    public static final int EXPANSION = 2;
    private int[] data = new int[INITIAL_SIZE];
    private int length = 0;
    public void append(int val) {
        if (length >= data.length) {
            int[] tmp = new int[EXPANSION*data.length];
            for (int i = 0; i<data.length; ++i) {
                tmp[i] = data[i];
            }
            data = tmp;
        }
        data[length] = val;
        length++;
    }
```

QuickPoll: Avoiding Privacy Leaks

If your class has a `private String[] s` instance variable, how would you write an accessor method to return the strings to the user without causing a privacy leak?

- A. Strings are immutable, so it's OK to just return the array `s`
- B. Make a new array, copy contents of `s` into it, and return the copy
- C. Use the copy constructor for arrays to copy over the array and return the copy
- D. Write the accessor to take an index `i` as input and return only `s[i]`

QuickPoll: Avoiding Privacy Leaks

If your class has a `private String[] s` instance variable, how would you write an accessor method to return the strings to the user without causing a privacy leak?

A. Strings are immutable, so it's OK to just return the array `s`
B. Make a new array, copy contents of `s` into it, and return the copy
C. Use the copy constructor for arrays to copy over the array and return the copy
D. Write the accessor to take an index `i` as input and return only `s[i]`

The Foreach Loop

- As of Java 5, there is a special form of `for` loop: the foreach loop
- Form:
  `for(elementType name : array) body`
- Executes *body* for each element of *array*, with variable *name* bound to the element
- No need to worry about array indices
- *E.g.*:

```
int daysInYear = 0;
for (int monthLength : DAYS_IN_MONTH) {
    daysInYear += monthLength;
}
```

Multidimensional Arrays

- Unlike some languages, Java does not directly support 2 or more dimensional arrays
- However, you can make arrays of arrays, *e.g.*:

```
int[][] tTable = new int[10][];
for (int i = 0; i<10; ++i) tTable[i] = new int[10];
for (int i = 0;  i < tTable.length; ++i) {
    for (int j = 0; j < tTable[i].length; ++j) {
        tTable[i][j] = (i+1)*(j+1);
    }
}
for (int[] row : tTable) {
    for (int n : row) System.out.printf("%4d", n);
    System.out.println();
}
```

Enumerated Types

- An enumerated type is a type whose values are all specific constants
- New feature in Java 5
- (Simplest) form:
- `enum` *typeName* {*value1*, *value2*,...};
- Place this at top level inside a class, or in a file by itself, named *typeName*`.java`
- Convention: values are spelled all uppercase, words separated by underscores
- *E.g.*:

```
enum DayOfWeek {SUNDAY, MONDAY, TUESDAY,
        WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
};
```

Using Enumerated Types

- With `enum` declaration, *`typeName`* is a valid type
- Each *`value`* is a constant referred to as *`typeName.value`*
- Use `==` and `!=` to compare `enum` values for equality
- *E.g.*:

```
public static boolean isWeekend(DayOfWeek day) {
    return day == DayOfWeek.SATURDAY
        || day == DayOfWeek.SUNDAY;
}
```

Enumerated Type Methods

- Java implements `enum`s as classes
- They automatically have useful public methods:
  - `String toString()` returns value as a `String`
  - `static` *`type`* `valueOf(String)` returns enum value of string (Note `static`; error if no exact match)
  - `int ordinal()` returns 0 for the first value, 1 for the second, *etc.*.
  - `static` *`type`*`[] values()` Returns an array of all the enum values for the type, in order
- `values` is very useful, as it allows you to loop over all values of an enum type

`equals` and `compareTo`

- Every class type *t*, including `enum` types, defines:
  - `boolean equals(`*t*`)` returns `true` iff the object is "the same" as the argument
  - `int compareTo(`*t*`)` returns a negative number if the object is "less than" as the argument, zero if the same, and positive if "greater"
- For classes, the programmer must define these
- It's up to you to decide what "the same", "less" and "greater" mean for your class
- For `enum` types, they are defined automatically; "less than" means appearing earlier in the declaration
- Usually must use `a.equals(b)`, not `a==b`, to test objects for equality; enum types are an exception
- Not defined for arrays, only classes and enums

Enumerated Type Example

```
public class DayOfWeekTest {
    enum DayOfWeek {SUNDAY, MONDAY, TUESDAY,
        WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
    };

    public static void main(String[] args) {
        DayOfWeek d = DayOfWeek.valueOf(args[0]);
        System.out.println(d + ": " +
                (d.ordinal()+1) + "th day of week");
        int next = d.ordinal() + 1;
        next %= DayOfWeek.values().length;
        d = DayOfWeek.values()[next];
        System.out.println("Next day is " + d);
    }
}
```

Summary

- An array is a collection of any number of values of the same type (can be primitive or object type)
- `baseType[]` is type of array of *baseType* elements
- `new baseType[size]` returns a fresh array of *size* elements of *baseType*
- Access/set elements using `array[index]` syntax; `array[0]` is first element
- `array.length` is number of elements in array
- `for(type name : array)` iterates over *array*
- `enum type {value1,...};` declares an enumerated type

COMP90041
Programming and Software Development

Inheritance

Semester 2, 2015

Abstract Datatypes

- An abstract data type (ADT) is a type that is defined in terms of its operations and their semantics (meaning)
- The focus of an ADT is on its interface — its publicly visible part
- Clients (users) of a class view it as its interface
- The interface hides the complexity of the implementation from clients
- ADTs are one of the central concepts of object oriented programming

Interface Perspectives

- But an ADT also needs an implementation
- The ADT's implementors view it as its implementation
- The interface hides the complexity of all the ADT's uses from its implementors
- An ADT's interface insulates client from implementor, allowing them to work independently
  - As long as client follows ADT interface, he may use the ADT any way he likes
  - As long as the implementor maintains the ADT interface, she may modify the implementation any way she likes
- A Java class is well-suited to defining an ADT

Design by Contract

- ADT's interface specifies semantics of operations: *"if you supply inputs that meet these conditions, I will supply an output that meets those conditions"*
- Think of an interface as a contract between implementor and client
  - Class clients must supply inputs that meet the contract
  - Class implementor expects (should verify) inputs that meet the contract, and must supply outputs that do
- This leads to design by contract emphasising not just the types of operation inputs and outputs, but their preconditions and postconditions

Inheritance

- The second central concept of object oriented programming is inheritance
- Inheritance allows a derived class to be defined by specifying only how it differs from its base class
- Base class also called superclass or parent class; derived class also called subclass or child class
- Parts of the derived class that are the same as in the base class need not be mentioned again
- Called implementation inheritance because implementation as well as interface is inherited

Declaration

- Put `extends` *`BaseClass`* in declaration of derived class to declare inheritance, *e.g.*:

```
public class LostPerson extends Person {
    private String lastSeenLocation;
    private Time lastSeenTime;
    ⋮
```

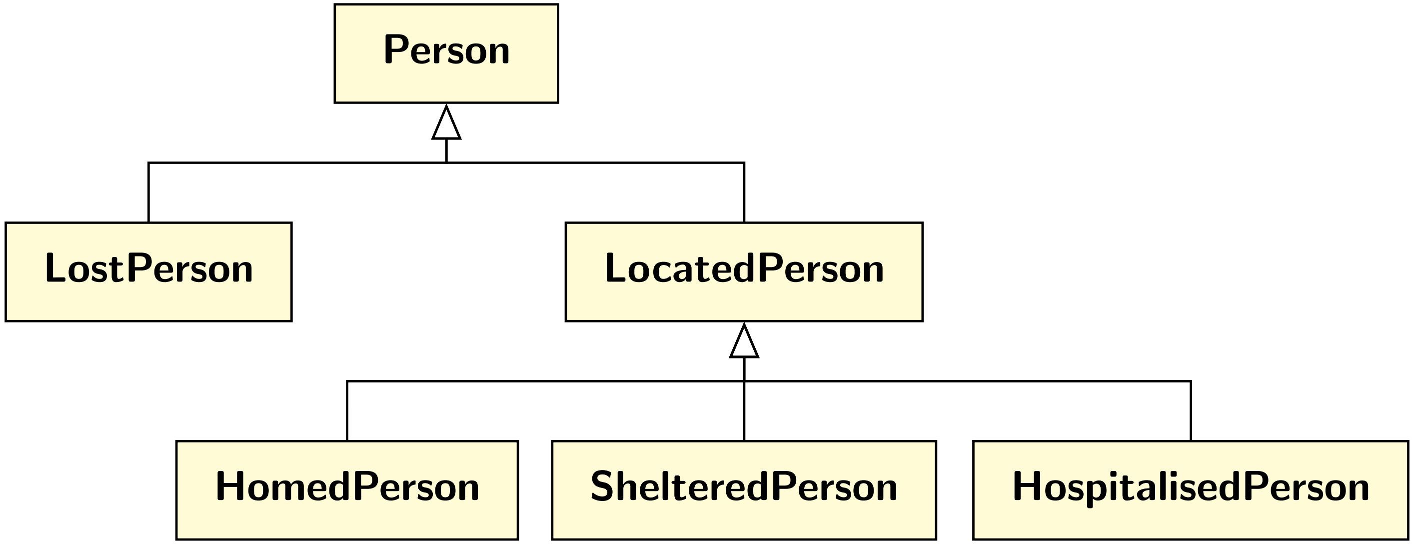
- This `LostPerson` class inherits all the instance variables and methods of the `Person` class
- ... and adds its own
- No need to mention inherited instance variables and methods

Liskov Substitution Principle

- In Java, every instance of the derived class is also an instance of the base class
- *E.g.*, every `LostPerson` is a `Person`
- Things you can do with a `LostPerson` object:
  - Store it in a variable of type `Person`;
  - Pass it as a message argument of type `Person`;
  - Send it any message understood by a `Person`
- Liskov Substitution Principle (LSP) says *it must be possible to substitute an instance of the derived class anywhere the base class could be used*
- Be sure you design and implement derived classes so this is true

Taxonomy

- Each class can be extended by any number of classes
- Java is a single inheritance language: each class can extend only one class
- UML class diagram shows inheritance hierarchy: hollow-headed arrows point to base classes



- Each class inherits from all its ancestors; members are inherited by all descendants

Overriding

- If a class defines a method with the same signature as an ancestor, its definition overrides the ancestor's
- Base or derived class's definition is used depending on class of object
- *E.g.*, `Person` class's `toString` method just shows name and age; `LostPerson`'s `toString`:

```
public String toString() {
    return getName() + ", age " +
           getAge() + ", last seen " +
           lastSeenTime + " at " +
           lastSeenLocation;
}
```

Using Overridden Methods

- Need to use `getName()` and `getAge()` because `name` and `age` instance variables are private
- Would be better to use the overridden `toString()` method: works even if we modify superclass
- We can: inside a method, use `super.`*`methodName`*`(`*`args...`*`)` to invoke the overridden method

```
public String toString() {
    return super.toString() +
            ", last seen " + lastSeenTime +
            " at " + lastSeenLocation;
}
```

Late Binding

- In Java we can always use a `LostPerson` where a `Person` is expected:

```
Person p = new LostPerson(...);
System.out.println(p);
```

- Which `toString` method is used?
  - `LostPerson`'s because that's what `p` actually is?
  - `Person`'s because that's what `p` is declared to be?
- For Java, it's always based on object's actual type
- This is called late binding or dynamic binding, because compiler defers decision to runtime
- No late binding for `static` members (because there is no `this` object on which to base the decision)
- But `static` members are still inherited

Pitfall: Overriding vs. Overloading

- Overloading and overriding are completely different
- If signature of method in derived class is the same as method in base class, it's overriding
- If method name is the same but signature is different, it's overloading
- When sending message to instance of derived class:
  - With overloading, you can access both methods, depending on number and types of arguments
  - With overriding, you can only access the overriding method
- You usually want overriding

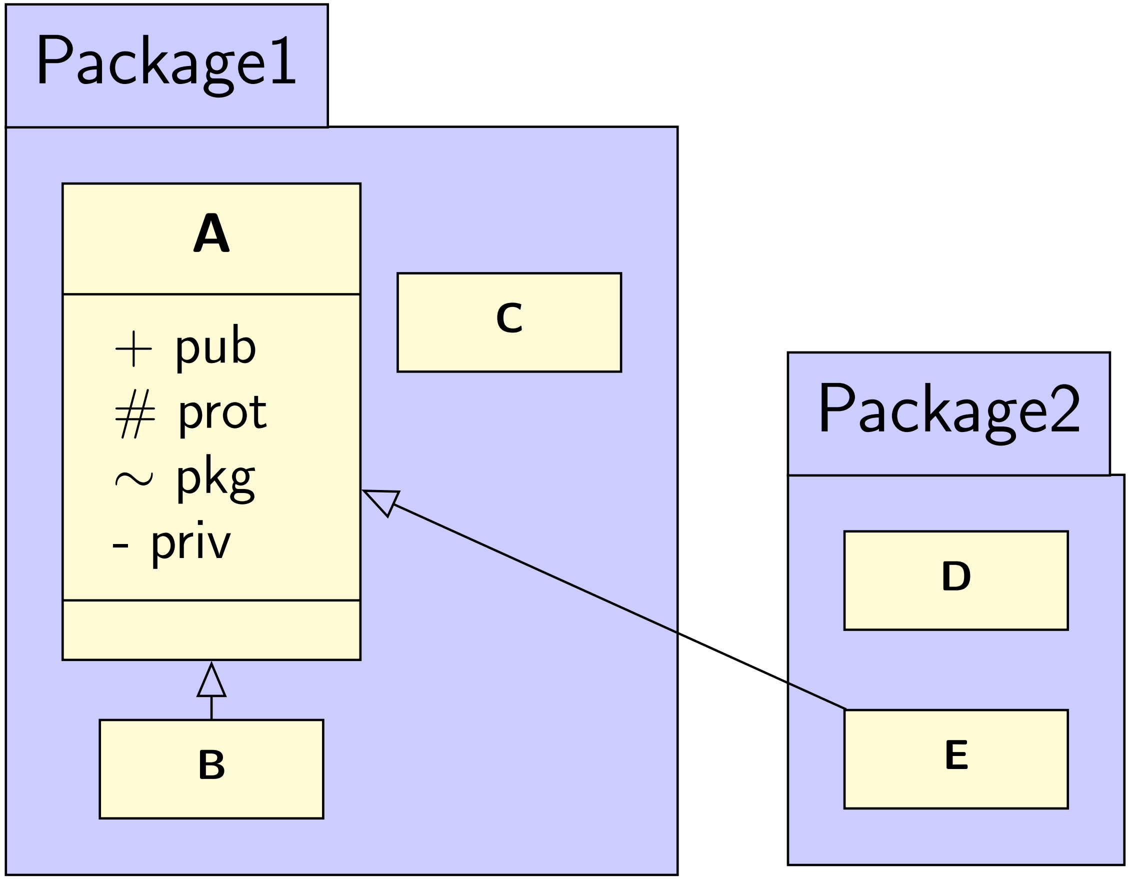
Inheritance and Visibility

- Occasionally it's useful to allow methods in derived classes to access base class instance variables
- Protected visibility allows this
- Form: `protected` *`instanceVar`*`;`
- Alternative to `public` and `private`
- Name is a misnomer: `protected` instance variables are not well very protected
- To access a protected instance variable, you just need to create a subclass
- Can also declare methods `protected`, which may be more useful

Packages and Visibility

- A Java package is a collection of classes
- Class declares package with: `package` *`pkgname`* `;`
- Package classes all in same folder/directory
- Protected members are also visible in all classes in the same package
- Fourth visibility is called default or friendly or package visibility
- This means visible in any class in the same package
- Declare package visibility by not using any visibility keyword (no `public`, `private`, or `protected`)
- In order of least visibility to most:
  `private` < default < `protected` < `public`

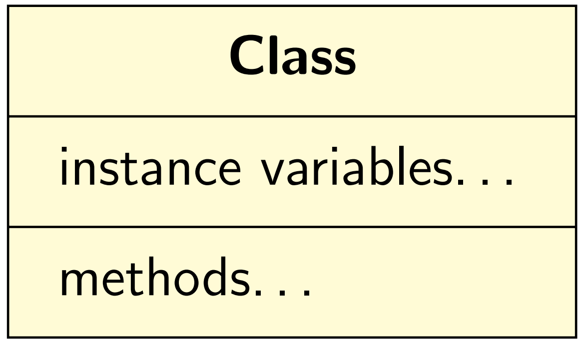
Packages and Visibility



A sees pub, prot, pkg, priv
B sees pub, prot, pkg,
C sees pub, prot, pkg,
D sees pub
E sees pub, prot

UML legend

Class parts:



Visibility:

$+ \rightarrow$ public
$\# \rightarrow$ protected
$\sim \rightarrow$ package
$- \rightarrow$ private

QuickPoll: Which classes can access method `m`?

```
package p1; public class c1 { void m()... }
package p1; public class c2 {...}
package p1; public class c3 extends c1 {...}
package p2; public class c4 {...}
package p2; public class c5 extends c1 {...}
```

- A. c1 only
- B. c1 and c3 only
- C. c1, c2 and c3 only
- D. c1, c2, c3 and c5 only
- E. All of c1, c2, c3, c4 and c5

QuickPoll: Which classes can access method `m`?

```
package p1; public class c1 { void m()... }
package p1; public class c2 {...}
package p1; public class c3 extends c1 {...}
package p2; public class c4 {...}
package p2; public class c5 extends c1 {...}
```

- (A) c1 only
- (B) c1 and c3 only
- (C) c1, c2 and c3 only
- (D) c1, c2, c3 and c5 only
- (E) All of c1, c2, c3, c4 and c5

Visibility and Overriding

- You cannot override a method giving it less visibility
- LSP requires that a visible base class method must be visible for every descendent class
- You can override a method with a <u>more</u> visible method, though

The super Constructor

- Constructors are not inherited, cannot be overriden
- Base class constructor must be run to set up its instance variables (especially if private)
- Constructor chaining: derived class constructor must invoke base class constructor first
- Form: `super(`*`constructor arguments...`*`)`, *e.g.*:

```
public LostPerson(int age, String name,
        String lastLoc, Time lastTime) {
    super(age, name);
    this.lastSeenLocation = lastLoc;
    this.lastSeenTime = lastTime;
}
```

The `this` Constructor

- Sometimes you want to chain to a different overloaded constructor of the same class
- Form: `this(`*constructor arguments*`...)`
- Must be first in constructor, in place of `super`
- The constructor chained to will itself chain to super (or another overloaded constructor that will. . .)

```
public Person(Person orig) {
    this(orig.age, orig.name);
}
```

- Can also chain to constructor that does part of the work, and then do whatever extra is needed

Default constructor chaining

- If constructor doesn't begin with `super(...)` or `this(...)`, Java automatically inserts `super();`
- If your class doesn't have a no-argument constructor, this will be an error
- If you write a class without writing any constructor, Java will automatically write a no-argument constructor with body `{super();}`
- But if you write any constructor at all, Java does not give you the free no-argument one

The `Object` Class

- A class that does not declare what class it `extends` automatically extends the `Object` class
- So class hierarchy is a tree (but we don't usually show `Object` class in class diagrams)
- The `Object` class defines a `toString()` method
- ...and an `equals(Object other)` method
- These are inherited by all classes, but the definitions are not useful
- They should be overriden if they will be used
- `Object` class has no instance variables
- `Object` has a no-argument constructor that does nothing, so default constructor chaining is fine

QuickPoll: Overriding

If a base class `S` contains only one method, with this header:

```
public void foo(int i)
```

Which of these definitions would not be allowed in derived classes of `S`:

- A `public void foo(int i){...}`
- B `private void foo(int i){...}`
- C `private void foo(char c){...}`
- D All of A, B, and C would be allowed
- E None of A, B, and C would be allowed

QuickPoll: Overriding

If a base class `S` contains only one method, with this header:

```
public void foo(int i)
```

Which of these definitions would <u>not</u> be allowed in derived classes of `S`:

- A `public void foo(int i){...}`
- B `private void foo(int i){...}`
- C `private void foo(char c){...}`
- D All of A, B, and C would be allowed
- E None of A, B, and C would be allowed

The `instanceof` operator

- Usually you use overriding to arrange each method to behave correctly for every descendant of a class
- So most code does not need to worry about which descendant type of a base class an object is
- Occasionally you want a test to see if an object is a descendant of a class
- Form: *`object`* `instanceof` *`ClassName`*
- *E.g.*:

```
if (p instanceof LostPerson) {
    System.out.println(p + " is missing");
}
```

Upcasting and Downcasting

- Derived class may have methods base class does not
- Java will not let you use a method not supported by declared type of an object
- Need to downcast (narrowing cast) to derived class to use derived class-specific methods

```
String lastLoc = "";
if (p instanceof LostPerson) {
    LostPerson lp = (LostPerson) p;
    lastLoc = lp.getLastSeenLocation();
}
```

- Can upcast (widening cast) implicitly or explicitly

```
Person p = (Person) new LostPerson(...);
```

The `getClass` method

- `Object` class also defines a getClass() method
- `ob.getClass()` returns an object that represents the actual class of `ob`
- You can use `==` and `!=` to compare two of these to see if classes are the same
- *E.g.*, `o1.getClass() == o2.getClass()` is true if `o1` and `o2` are actually instances of the same class, not just descendants of some class
- You cannot override `getClass` for your classes, but you don't need to

Defining the `equals` method

- Must override not just overload the `equals` method
- Must have the signature `equals(Object other)` (class of `other` must be `Object`)
- Must check that `other` is not `null`
- Use `getClass` to check objects are the same class
- Must downcast `other` to correct class so you can access members to compare
- For derived class, use `super.equals(other)` to check base class instance variables

Defining the equals method

```
public boolean equals(Object other) {
    if (other == null ||
        this.getClass() != other.getClass() ||
        !super.equals(other)) {
        return false;
    }
    LostPerson lp = (LostPerson) other;
    return (this.lastSeenLocation.equals(
                        lp.lastSeenLocation) &&
            this.lastSeenTime.equals(
                        lp.lastSeenTime));
}
```

Summary

- Methods form a contract between client and implementor
- Use `extends` to define class that inherits members from base class
- Instances of derived class are also instances of the base class
  - Design/implement classes so this makes sense
- Derived class can override base class methods
- Late binding: definition for actual class is used
- Members with default visibility accessible from package; `protected` also accessible from subclasses
- Use `super` constructor to chain to base class constructor; `this` to chain to same class

COMP90041
Programming and Software Development

Code Quality

Semester 2, 2015

Correctness is not enough

- Programs need to be maintained
- Requirements change:
  - They need to work with new systems/formats
  - They need to do new things
  - They need to work faster or handle more data
- Most effort is devoted to a program after it is first completed
- A program is read more than it is written
- Prioritise maintainability and readability
- Following are rules of thumbs; there are exceptions
  - But you should have a good reason for violating them

What does this code do?

```
public void processItem(Item item, int num) {
    int best = -1;
    // handle everything in the item
    for (int i = 0; i<=num; ++i) {
        if (handleOne(item, i)) best = selectO
ne(best, i);
    }
    // finish it off!
    completeProcessing(item, best, itemCost(it
em, best));
}
```

Writing understandable programs

- A well-written program is easy to read
- Structure your program like an essay or a newspaper article:
  - Big picture stuff comes first
  - Then the few most important pieces
  - Details come later
  - . . . as far as the programming language will allow.
- The best way to say what a class, method, or variable is or does is with its name
  - The name appears wherever it is used
  - Take the trouble to choose good names
- Often the name can't say enough; then augment a good name with good documentation

Documentation

- Main program (class with `main` method) should begin with documentation:
  - What program as a whole does
  - Broadly how it works
- Every other file should begin with documentation:
  - Broadly what this file is for
  - Who wrote it (who to blame/praise)
- Every non-trivial method should begin with doc:
  - What the method is for, beyond what the name says
  - Code comments are less important
- In code, only need to explain subtleties
  - Don't document what is obvious from the code

Tidiness

- Make your code look neat
  - Consistent layout
  - Avoid long lines (80 columns is standard)
  - Beware tabs (8 column tab stops is standard; better to avoid tabs altogether)
  - Lay out comments and code neatly
  - Use blank lines to separate parts of code
- Divide long files into sections devoted to different aspects
  - Use a comment to explain each section

Good code

- Avoid copy and paste coding
  - Copied code usually needs to be edited every time it's pasted; easy to miss some
  - If you find an error in pasted code, youll have to fix it everywhere pasted, and youll probably miss some
- Use abstraction
  - Sometimes you can just reorganise code (loops, if-then-elses) to avoid duplication
  - Otherwise define a method rather than duplicating code

How could you rewrite this to simplify?

```
if (a) {
    if (b) {
        X();
    }
    Y();
} else {
    if (b) {
        X();
    }
}
```

How could you rewrite this to simplify?

```
if (a) {
    if (b) {
        X();
    }
    Y();
} else {
    if (b) {
        X();
    }
}
```

Consider cases:

a	**b**	action
true	true	`X(); Y();`
true	false	`Y();`
false	true	`X();`
false	false	none

How could you rewrite this to simplify?

```
if (a) {
    if (b) {
        X();
    }
    Y();
} else {
    if (b) {
        X();
    }
}
```

Consider cases:

a	**b**	action
true	true	`X(); Y();`
true	false	`Y();`
false	true	`X();`
false	false	none

Simpler equivalent code:

```
if (b) X();
if (a) Y();
```

Principles

- KISS (**K**eep **I**t **S**imple and **S**tupid)
  - First try the simplest solution that could possibly work
  - Only make it more complicated if that doesn't work (*e.g.*, too slow)
- DRY (**D**ont **R**epeat **Y**ourself)
  - Define each piece of logic in only one place
  - Define constants rather than repeating magic numbers throughout your code
  - Less code means fewer chances to make mistakes
  - And less to fix when there's a mistake
- Keep definitions short
  - Methods shouldn't be more than a page; shorter is better
  - Split them into multiple methods if necessary
- Follow standards
  - If theres a preferred way to do something, do it that way

How to simplify your code

- Simpler, shorter code is usually better
- Avoid duplication and complication in your code
- Think of working version of code as a first draft — edit as necessary to simplify, clarify, and organise
- This is called refactoring
- An important part of software development
- The following refactoring example uses odd layout to fit on a slide and make changes clear

Whats wrong with this code?

```
public void capture(int rows, int columns) {

    if (Math.abs(rows)==2 && Math.abs(columns)==2) {
        if (king && col+columns<=8 && col+columns>=1
                 && row+rows<=8 && row+rows>=1) {
            row=row+rows;
            col=col+columns;
        } else if (red && rows==2 &&    row<=6
                   && col+columns<=8 && col+columns>=1) {
            row=row+rows;
            col=col+columns;
        } else if (red==false && rows==-2 &&    row>=2
                   && col+columns<=8 && col+columns>=1) {
            row=row+rows;
            col=col+columns;
        }
    }
}
```

Repetition. Factor out common code.

```
public void capture(int rows, int columns) {

    if (Math.abs(rows)==2 && Math.abs(columns)==2) {
        if (king && col+columns<=8 && col+columns>=1
                 && row+rows<=8 && row+rows>=1) {
            row=row+rows;
            col=col+columns;
        } else if (red && rows==2 &&    row<=6
                   && col+columns<=8 && col+columns>=1) {
            row=row+rows;
            col=col+columns;
        } else if (red==false && rows==-2 &&    row>=2
                   && col+columns<=8 && col+columns>=1) {
            row=row+rows;
            col=col+columns;
        }
    }
}
```

Repetition. Factor out common code.

```
public void capture(int rows, int columns) {
    int newRow = row+rows;
    if (Math.abs(rows)==2 && Math.abs(columns)==2) {
        if (king && col+columns<=8 && col+columns>=1
                 && newRow  <=8 && newRow  >=1) {
            row=newRow  ;
            col=col+columns;
        } else if (red && rows==2 &&    row<=6
                   && col+columns<=8 && col+columns>=1) {
            row=newRow  ;
            col=col+columns;
        } else if (red==false && rows==-2 &&    row>=2
                   && col+columns<=8 && col+columns>=1) {
            row=newRow  ;
            col=col+columns;
        }
    }
}
```

Factor out more common code.

```
public void capture(int rows, int columns) {
    int newRow = row+rows;
    if (Math.abs(rows)==2 && Math.abs(columns)==2) {
        if (king && col+columns<=8 && col+columns>=1
                 && newRow  <=8 && newRow  >=1) {
            row=newRow  ;
            col=col+columns;
        } else if (red && rows==2 &&    row<=6
                   && col+columns<=8 && col+columns>=1) {
            row=newRow  ;
            col=col+columns;
        } else if (red==false && rows==-2 &&    row>=2
                   && col+columns<=8 && col+columns>=1) {
            row=newRow  ;
            col=col+columns;
        }
    }
}
```

Factor out more common code.

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2) {
        if (king && newCol     <=8 && newCol     >=1
                 && newRow  <=8 && newRow  >=1) {
            row=newRow  ;
            col=newCol     ;
        } else if (red && rows==2 &&    row<=6
                   && newCol     <=8 && newCol     >=1) {
            row=newRow  ;
            col=newCol     ;
        } else if (red==false && rows==-2 &&    row>=2
                   && newCol     <=8 && newCol     >=1) {
            row=newRow  ;
            col=newCol     ;
        }
    }
}
```

Every condition has same test: factor it

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2) {
        if (king && newCol     <=8 && newCol     >=1
                 && newRow  <=8 && newRow  >=1) {
            row=newRow  ;
            col=newCol    ;
        } else if (red && rows==2 &&    row<=6
                   && newCol     <=8 && newCol     >=1) {
            row=newRow  ;
            col=newCol    ;
        } else if (red==false && rows==-2 &&    row>=2
                   && newCol     <=8 && newCol     >=1) {
            row=newRow  ;
            col=newCol     ;
        }
    }
}
```

Every condition has same test: factor it

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                && newCol     <=8 && newCol     >=1) {
        if (king && newRow  <=8 && newRow  >=1) {
            row=newRow  ;
            col=newCol    ;
        } else if (red && rows==2 &&    row<=6
                                                    ) {
            row=newRow  ;
            col=newCol    ;
        } else if (red==false && rows==-2 &&    row>=2
                                                    ) {
            row=newRow  ;
            col=newCol     ;
        }
    }
}
```

rows==2 && row<=6 → newRow<=8

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                 && newCol     <=8 && newCol     >=1) {
        if (king && newRow  <=8 && newRow  >=1) {
            row=newRow  ;
            col=newCol    ;
        } else if (red && rows==2 &&    row<=6
                                                    ) {
            row=newRow  ;
            col=newCol    ;
        } else if (red==false && rows==-2 &&    row>=2
                                                    ) {
            row=newRow  ;
            col=newCol     ;
        }
    }
}
```

rows==2 && row<=6 → newRow<=8

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                && newCol      <=8 && newCol      >=1) {
        if (king && newRow  <=8 && newRow  >=1) {
            row=newRow  ;
            col=newCol    ;
        } else if (red && rows==2 && newRow<=8
                                                          ) {
            row=newRow  ;
            col=newCol    ;
        } else if (red==false && rows==-2 &&    row>=2
                                                          ) {
            row=newRow  ;
            col=newCol     ;
        }
    }
}
```

row>=2 is a bug; should be row>=3

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                && newCol     <=8 && newCol     >=1) {
        if (king && newRow  <=8 && newRow  >=1) {
            row=newRow  ;
            col=newCol    ;
        } else if (red && rows==2 && newRow<=8
                                                        ) {
            row=newRow  ;
            col=newCol    ;
        } else if (red==false && rows==-2 &&    row>=2
                                                        ) {
            row=newRow  ;
            col=newCol     ;
        }
    }
}
```

row>=2 is a bug; should be row>=3

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                && newCol     <=8 && newCol     >=1) {
        if (king && newRow  <=8 && newRow  >=1) {
            row=newRow  ;
            col=newCol    ;
        } else if (red && rows==2 && newRow<=8
                                                    ) {
            row=newRow  ;
            col=newCol    ;
        } else if (red==false && rows==-2 &&    row>=3
                                                    ) {
            row=newRow  ;
            col=newCol     ;
        }
    }
}
```

rows==-2 && row>=3 → newRow>=1

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                 && newCol     <=8 && newCol     >=1) {
        if (king && newRow  <=8 && newRow  >=1) {
            row=newRow  ;
            col=newCol    ;
        } else if (red && rows==2 && newRow<=8
                                                ) {
            row=newRow  ;
            col=newCol    ;
        } else if (red==false && rows==-2 &&    row>=3
                                                ) {
            row=newRow  ;
            col=newCol     ;
        }
    }
}
```

rows==-2 && row>=3 $\rightarrow$ newRow>=1

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                && newCol     <=8 && newCol     >=1) {
        if (king && newRow  <=8 && newRow  >=1) {
            row=newRow  ;
            col=newCol    ;
        } else if (red && rows==2 && newRow<=8
                                                ) {
            row=newRow  ;
            col=newCol    ;
        } else if (red==false && rows==-2 && newRow>=1
                                                ) {
            row=newRow  ;
            col=newCol     ;
        }
    }
}
```

Combine newRow conditions

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                && newCol     <=8 && newCol     >=1) {
        if (king && newRow  <=8 && newRow  >=1) {
            row=newRow  ;
            col=newCol    ;
        } else if (red && rows==2 && newRow<=8
                                                    ) {
            row=newRow  ;
            col=newCol    ;
        } else if (red==false && rows==-2 && newRow>=1
                                                    ) {
            row=newRow  ;
            col=newCol     ;
        }
    }
}
```

Combine newRow conditions

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                    && newCol     <=8 && newCol     >=1
                    && newRow   <=8 && newRow   >=1) {
        if (king) {
            row=newRow  ;
            col=newCol    ;
        } else if (red && rows==2                        ) {
            row=newRow  ;
            col=newCol    ;
        } else if (red==false && rows==-2
                                                         ) {
            row=newRow  ;
            col=newCol     ;
        }
    }
}
```

Same body for all conditions: combine

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                 && newCol     <=8 && newCol     >=1
                 && newRow   <=8 && newRow   >=1) {
        if (king) {
            row=newRow   ;
            col=newCol     ;
        } else if (red && rows==2                       ) {
            row=newRow   ;
            col=newCol     ;
        } else if (red==false && rows==-2
                                                        ) {
            row=newRow   ;
            col=newCol      ;
        }
    }
}
```

Same body for all conditions: combine

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                 && newCol     <=8 && newCol     >=1
                 && newRow   <=8 && newRow   >=1) {
        if (king



                || red && rows==2



                || red==false && rows==-2
                                                          ) {
            row=newRow   ;
            col=newCol      ;
        }
    }
}
```

if inside if is the same as &&

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                    && newCol     <=8 && newCol     >=1
                    && newRow   <=8 && newRow   >=1) {
        if (king



                   || red && rows==2



                   || red==false && rows==-2
                                                        ) {
            row=newRow   ;
            col=newCol       ;
        }
    }
}
```

if inside if is the same as &&

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                 && newCol     <=8 && newCol     >=1
                 && newRow   <=8 && newRow   >=1
        && (king



                 || red && rows==2



                 || red==false && rows==-2
                                                         )){
            row=newRow   ;
            col=newCol      ;

    }
}
```

==false is the same as negation

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                && newCol     <=8 && newCol     >=1
                && newRow   <=8 && newRow   >=1
        && (king



                || red && rows==2



                || red==false && rows==-2
                                                        )){
            row=newRow   ;
            col=newCol      ;

    }
}
```

==false is the same as negation

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                 && newCol     <=8 && newCol     >=1
                 && newRow   <=8 && newRow   >=1
        && (king



                || red && rows==2



                || !red        && rows==-2
                                                      )){
            row=newRow   ;
            col=newCol      ;

    }
}
```

Value for `rows` depends only on `red`

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                 && newCol     <=8 && newCol     >=1
                 && newRow  <=8 && newRow  >=1
        && (king



                || red && rows==2



                || !red      && rows==-2
                                                        )){
            row=newRow  ;
            col=newCol     ;

    }
}
```

Value for rows depends only on red

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                 && newCol     <=8 && newCol     >=1
                 && newRow   <=8 && newRow   >=1
        && (king


                ||        rows==(red ? 2 : -2)




                                                         )){
            row=newRow   ;
            col=newCol     ;

    }
}
```

Tidy up the definition

```
public void capture(int rows, int columns) {
  int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                 && newCol     <=8 && newCol     >=1
                 && newRow  <=8 && newRow  >=1
        && (king



                 ||        rows==(red ? 2 : -2)





                                                      )){
            row=newRow  ;
            col=newCol     ;


    }
}
```

Tidy up the definition

```
public void capture(int rows, int columns            ) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2    && Math.abs(columns)==2
        && newCol <=8 && newCol >=1 && newRow <=8 && newRow >=1
        && (king || rows==(red ? 2 : -2))){
        row=newRow; col=newCol;
    }
}
```

Concept appearing throughout class

```
public void capture(int rows, int columns           ) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2    && Math.abs(columns)==2
        && newCol <=8 && newCol >=1 && newRow <=8 && newRow >=1
        && (king || rows==(red ? 2 : -2))){
        row=newRow; col=newCol;
    }
}
```

Concept appearing throughout class

```
public void capture(int rows, int columns          ) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2    && Math.abs(columns)==2
        && onBoard(newRow, newCol)
        && (king || rows==(red ? 2 : -2))){
        row=newRow; col=newCol;
    }
}

public boolean onBoard(int row, int column) {
    return row >= 1 && row <= 8 && column >= 1 && column <= 8;
}
```

Checking magnitude of rows twice

```
public void capture(int rows, int columns              ) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2    && Math.abs(columns)==2
        && onBoard(newRow, newCol)
        && (king || rows==(red ? 2 : -2))){
        row=newRow; col=newCol;
    }
}

public boolean onBoard(int row, int column) {
    return row >= 1 && row <= 8 && column >= 1 && column <= 8;
}
```

Checking magnitude of `rows` twice

```
public void capture(int rows, int columns           ) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2    && Math.abs(columns)==2
        && onBoard(newRow, newCol)
        && (king || red==(rows>0)       )){
        row=newRow; col=newCol;
    }
}

public boolean onBoard(int row, int column) {
    return row >= 1 && row <= 8 && column >= 1 && column <= 8;
}
```

Code to move almost same as capture

```
public void capture(int rows, int columns          ) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2    && Math.abs(columns)==2
        && onBoard(newRow, newCol)
        && (king || red==(rows>0)       )){
        row=newRow; col=newCol;
    }
}

public boolean onBoard(int row, int column) {
    return row >= 1 && row <= 8 && column >= 1 && column <= 8;
}
```

Code to move almost same as capture

```
public void capture(int rows, int columns) {
    shift(rows, columns, 2);
}
public void move(int rows, int columns) {
    shift(rows, columns, 1);
}
public void   shift(int rows, int columns, int dist) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==dist && Math.abs(columns)==dist
        && onBoard(newRow, newCol)
        && (king || red==(rows>0)       )){
        row=newRow; col=newCol;
    }
}

public boolean onBoard(int row, int column) {
    return row >= 1 && row <= 8 && column >= 1 && column <= 8;
}
```

Final tidied result

```
public void capture(int rows, int columns) {
    shift(rows, columns, 2);
}
public void move(int rows, int columns) {
    shift(rows, columns, 1);
}
public void shift(int rows, int columns, int dist) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==dist && Math.abs(columns)==dist
        && onBoard(newRow, newCol)
        && (king || red==(rows>0))){
        row=newRow; col=newCol;
    }
}

public boolean onBoard(int row, int column) {
    return row >= 1 && row <= 8 && column >= 1 && column <= 8;
}
```

Why this is better than the original code

- Reorganising found and fixed a bug
- This method is shorter and easier to read
- Logic is more self-evident
- Handles twice as much of the problem in about the same number of lines
- Makes clear how `move` and `capture` are related (they're almost the same)
- Range tests on newRow and newCol are replaced by a method call whose name says what it means
- Can use the `onBoard` method throughout the class

Summary

- Document what program does, what each class is for, and what each method does
- Coding is like essay writing: express yourself clearly, briefly, and simply
- Refactor your code as you would edit an essay
- Organise your documentation and code to help reader understand the problem and your solution
- Chose descriptive names; use comments for subtleties
- Verbose, repetitive code obscures your intentions; keep definitions simple, clear, and short

COMP90041
Programming and Software Development

Polymorphism, Abstract Classes, and Interfaces

Semester 2, 2015

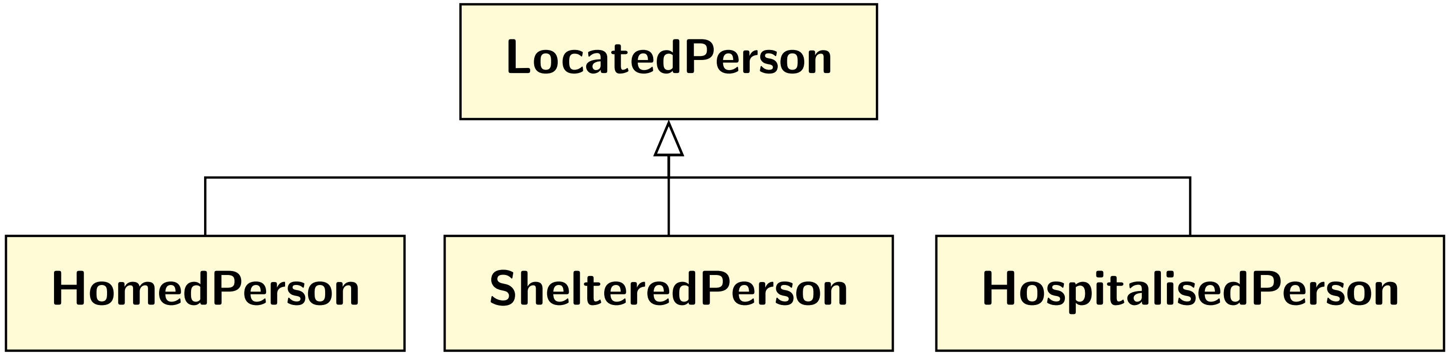
Polymorphism

- Polymorphism refers to the ability to write a single piece of code that handles multiple types of data
- Comes from Greek for many forms or many shapes
- This has two major benefits:
  - It avoids code duplication, making code more succinct
  - It makes code more flexible, allowing code in one part of a program to change without requiring changes elsewhere
- Java supports three kinds of polymorphism:
  - Ad hoc polymorphism — allowing a method to be applied to arguments of different types (overloading)
  - Subtype polymorphism — allowing subtypes to be used in place of supertypes (Liskov substitution; overriding)
  - Parametric polymorphism — allowing types to have parameters (Generics; to be discussed later)

Polymorphism

- `System.out.print` and friends demonstrate both ad hoc and subtype polymorphism:
  - `System.out.print` itself is heavily overloaded to handle all primitive types
  - It handles subclasses of `Object` by calling `toString`
  - `toString` uses overriding to print any object in an appropriate way
- This means code can use `System.out.print` to print anything
- Changing the type of a variable doesn't change how you print it

Abstract Classes



- The way we contact a `LocatedPerson` to report finding a missing person varies
- Each subclass will have its own `contactPerson` method:
  - `HomedPerson` will contact them at home
  - `ShelteredPerson` will contact them through the shelter
  - `HospitalisedPerson` will contact them at the hospital
- `LocatedPerson` is an abstraction of all of these

Abstract Methods

- To know how to contact a `LocatedPerson`, we need to know which kind of `LocatedPerson`
- No general way to define `contactPerson` method for `LocatedPerson` class
- But every descendent class of `LocatedPerson` must define `contactPerson` method
- Solution: `LocatedPerson` should define an abstract `contactPerson` method
- An abstract method is a method with a header but no definition
- It specifies a method that must be defined by subclasses

Abstract Methods

- Form:
  vis `abstract` *type method*`(`*params*`...);`
- Normal method declaration, with the `abstract` keyword, and a semicolon in place of method body
- If abstract method were actually <u>used</u>, there would be no body to execute
- So it must <u>not</u> be possible to make an instance of a class with abstract methods

Abstract Classes

- A class with any abstract methods must be declared to be abstract
- Form: `vis abstract class name { ...}`
- An abstract class is an abstraction of a closely related set of classes
- It serves as a (super-)type for all of these classes, so it is a valid type for variables and parameters
- But you cannot create an instance of an abstract class, only of its concrete subclasses
- Any class not declared `abstract` is a concrete class

Abstract Classes and Inheritance

- Any class that `extends` an abstract class must do one of two things:
  - Implement (override) all its abstract methods; or
  - Be declared `abstract` itself
- An abstract class can implement some but not all of its abstract base class's methods, and may add its own abstract and concrete methods
- An abstract class can extend a concrete class and add some concrete and abstract methods
- A class declared `abstract` is not required to have abstract methods, but it usually does
- A concrete class can extend an abstract base class and add concrete methods, but must override all abstract methods

Abstract LocatedPerson class

```
public abstract class LocatedPerson extends Person {
    public LocatedPerson(int age, String name) {
        super(age, name);
    }
    public abstract String contactInstructions();
}
```

- Now you can't do `new LocatedPerson(...)`
- OK to have variables of type `LocatedPerson`
- Abstract classes can have instance variables and methods
- They can and should have constructors, used by constructor chaining

Concrete HomedPerson class

```
public class HomedPerson extends LocatedPerson {
    private String phoneNumber;
    public HomedPerson(int age, String name,
            String phoneNumber) {
        super(age, name);
        this.phoneNumber = phoneNumber;
    }
    public String contactInstructions() {
        return "Returned home; ring " + phoneNumber;
    }
    ⋮ // getter and setter
}
```

Concrete HospitalisedPerson class

```
public class HospitalisedPerson extends LocatedPerson {
    private Hospital hospital;
    private String roomNumber;
    public HospitalisedPerson(int age, String name,
            Hospital hospital, String roomNumber) {
        super(age, name);
        this.hospital = hospital;
        this.roomNumber = roomNumber;
    }
    public String contactInstructions() {
        return "Contact at " + hospital +
                " room " + roomNumber;
    }
    ⋮ // getters and setters
}
```

Using LocatedPerson

Now each LostPerson can keep track of who to notify when they are found by adding an instance variable:

```
private LocatedPerson[] seekers = new LocatedPerson[0];
```

This can be used to give notification instructions:

```
public void contactSeekers() {
    for (LocatedPerson p : seekers) {
        System.out.println("Contact " + p);
        System.out.println(p.contactInstructions());
    }
}
```

This is safe because LocatedPerson cannot be instantiated

Interfaces

- Abstract classes allow a number of closely related classes to implement common methods
- A Java interface allows unrelated classes to implement common methods
- Defines a capability some classes have
- An interface is even more abstract than an abstract class:
  - Interfaces cannot have instance or class variables
  - Interfaces cannot have non-abstract or static methods[1]
- An interface specifies only abstract methods (and possibly constants)
- Like an abstract class, an interface is a type

[1] these restrictions are relaxed in Java 8

Use Case

- Imagine a dungeon text adventure game
- Some objects, like a `Scroll`, can be inspected more closely to reveal more detail
- Some objects, like a `Sack`, can be opened to reveal contents
- Some objects can be inspected and opened, such as a `CarvedBox`
- If `Inspectable` and `Openable` were abstract classes, which should `CarvedBox` be derived from?
- A Java class cannot `extend` multiple classes
- Not all `Inspectable` things are `Openable`, and vice versa, so neither can be derived from the other
- Interfaces allow multiple interface inheritance

Defining an Interface

- Form: `public interface` *`name`* `{ ...}`
- Like a class, define it in a file by itself
- If you leave off `public`, the interface will have package visibility — `private` and `protected` don't make sense
- The body should contain only abstract method declarations and constant definitions
- All members are implicitly `public`; can omit
- All methods are implicitly `abstract`; can omit
- All variables are implicitly `static final`; can omit
- Method bodies are omitted; just use a semicolon

Using an Interface

- A class can declare that it implements an interface
- Form:
  `public class` *`name`* `implements` *`iface`* `{...}`
- This promises that class *`name`* defines all interface *`iface`*'s abstract methods
- (or the class must be declared abstract)
- Like deriving an abstract base class
- Use interface name as a variable or parameter type

Using an Interface

- A class may implement multiple interfaces by following `implements` with multiple interface names separated by commas
- A class may be derived from a base class and implement interfaces by preceding `implements` with `extends` *`BaseClass`*
- Use *`var`* `instanceof` *`Interface`* in code to check if value of *`var`* implements *`Interface`*
- If so, then cast *`var`* to *`Interface`*, so you can use *`Interface`* methods

Abstract Base Class

- A `DungeonItem` is the root of the hierarchy of things that can appear in the dungeon
- All that unites every class of dungeon item is that it has a description
- It is more flexible to implement this with a method rather than an instance variable
  - Can implement it with an instance variable and a getter
  - Or can piece together description from the object
  - Or even have a method that returns a constant string if every instance of a class always has the same description

```
public abstract class DungeonItem {
    public abstract String getDescription();
}
```

Interfaces

```
public interface Inspectable {
    String detailIntro = "Looking closer, you see ";
    String getDetailedDescription();
}
```

- Note `Inspectable` interface defines a constant
- Classes that implement `Inspectable` can use it

```
public interface Openable {
    boolean isOpen();
    void open();
    void close();
}
```

- Note `public`, `static final`, and `abstract` are omitted

Scroll

```
public class Scroll extends DungeonItem
                    implements Inspectable {
    private final String scrollText;
    public Scroll(String scrollText) {
        this.scrollText = scrollText;
    }
    public String getDescription() {
        return "a faded scroll, covered with barely" +
                " decypherable ornate lettering";
    }
    public String getDetailedDescription() {
        return detailIntro +
               "the following words:\n" + scrollText;
    }
}
```

Sack

```
public class Sack extends DungeonItem
                  implements Openable {
    String detail;
    DungeonItem[] contents = new DungeonItem[0];
    private boolean tied = false;
    public Sack(String detail) {
        this.detail = detail;
    }
    public boolean isOpen() { return tied; }
    public void open() { tied = false; }
    public void close() { tied = true; }
```

Sack (cont'd)

```
public void addItem(DungeonItem item) {
    if (item instanceof Openable) {
        ((Openable)item).close();
    }
    DungeonItem[] newContents =
            new DungeonItem[contents.length+1];
    for (int i = 0; i<contents.length; ++i) {
        newContents[i] = contents[i];
    }
    newContents[contents.length] = item;
    contents = newContents;
}
```

Sack (cont'd)

```
    public String getDescription() {
        String descr = "a " + detail + " sack ";
        if (tied) {
            descr += "tied shut with a bit of twine";
        } else {
            descr += "containing" +
                    (contents.length > 0 ? ":" :
                    " nothing at all");
            for (DungeonItem item : contents) {
                descr += "\n\t"+item.getDescription();
            }
        }
        return descr;
    }
}
```

CarvedBox

```
public class CarvedBox extends DungeonItem
                  implements Openable, Inspectable {
    private final String inscription;
    private DungeonItem contents = null;
    private boolean open = false;
    public CarvedBox(String inscription) {
        this.inscription = inscription;
    }
    public String getDetailedDescription() {
        if (open) return "close the box to inspect it";
        return detailIntro + " the inscription '" +
                inscription + "'";
    }
    public boolean isOpen() { return open; }
    public void open() { open = true; }
    public void close() { open = false; }
```

CarvedBox (cont'd)

```
    public String getDescription() {
        String descr = "an ornately carved wooden box";
        if (open) {
            descr += " standing open to reveal";
            if (contents == null) {
                descr += " that it is empty";
            } else {
                descr += ":\n    " +
                        contents.getDescription();
            }
        } else {
            descr += " with fine runes on the cover";
        }
        return descr;
    }
}
```

DungeonTest

```
public class DungeonTest {
    // main method (next slide)
    private static void showMe(DungeonItem item,
                               String name) {
        System.out.println(name + ": " +
            item.getDescription());
        if (item instanceof Inspectable) {
            Inspectable ins = (Inspectable)item;
            System.out.println("Detail: " +
                ins.getDetailedDescription());
        }
    }
}
```

DungeonTest (main method)

```
public static void main(String[] args) {
    Scroll scroll = new Scroll("Please Turn Over");
    showMe(scroll, "Scroll");
    CarvedBox box = new CarvedBox("Pandora beware");
    showMe(box, "Box");
    box.open();  System.out.println("Opened it");
    showMe(box, "Box");
    box.addItem(scroll); System.out.println("Added scroll
    showMe(box, "Box");
    Sack bag = new Sack("brown felt");
    showMe(bag, "Sack");
    bag.addItem(box); System.out.println("Added box");
    showMe(bag, "Sack");
    bag.close(); System.out.println("Closed it");
    showMe(bag, "Sack");
}
```

Extending Interfaces

- An interface can be declared to `extend` one or more interfaces
- Form:

  `public interface` *`name`*

  `extends` $iface_1$`,`$iface_2$`, ... { ...}`
- The interface *`name`* is the union of all the abstract methods and constants in the specified interfaces
- ...plus the abstract methods and constants in the body

Pitfall: Conflicting Interfaces

- An interface cannot extend two interfaces that specify the same abstract method with different return types
- Or two interfaces with different meanings for the same signature
- Similarly a class can't implement two conflicting interfaces
- Same problem inheriting from a (possibly abstract) class with methods that conflict with a method in an interface the class implements
- Sorry, just can't combine those interfaces/classes

Java Standard Interfaces

- The `Comparable` interface specifies classes that allow two objects to be compared for less or greater
- `Comparable` interface specifies only one method:

```
int compareTo(Object o)
```

- `o1.compareTo(o2)` should return:
  - A negative number if `o1` $<$ `o2`
  - Zero if `o1` $=$ `o2`
  - A positive number if `o1` $>$ `o2`
- If class `C implements Comparable`, and `a` is an array of `C`, then `Arrays.sort(a)` will sort `a`
- You'll need to `import java.util.Arrays;`

Guidance

- Use non-abstract class inheritance if there's a meaningful class with a few variations with different ways to do something
- Use an abstract class when you have a limited variety of types each with its own way of doing something, but no "generic" version
- Use an interface when many unrelated classes should support some operations
- Use interfaces when some classes may support several unrelated operations
- But interfaces can't provide method definitions or instance variables

Summary

- 3 kinds of polymorphism: ad hoc (overloading), subtype (inheritance/overriding), and parametric
- Abstract methods have only method headers without bodies
- Classes with abstract methods must be declared abstract
- Concrete subclasses of abstract classes must specify the definitions
- Interface includes only abstract methods and constants
- Class can `implement` one or more interfaces by giving definitions for abstract methods

COMP90041
Programming and Software Development

Exceptions

Semester 2, 2015

If anything can go wrong. . .

- Error handling is difficult because the code that knows what to do when a problem happens is different from the code that detects the problem
- Two traditional ways to handle errors:
  1. return a result indicating whether operation succeeded
  2. Print an error message and exit
- Approach 1 is terrible;
  - **Very** common for programmers to fail to check return code, and just assume everything is OK
  - A method can only return one result, if it already needs to return one, how to also return the success code?
- Approach 2 is worse:
  - No way to recover from the program exiting!
  - This means every call to a method that could experience a problem must first test if there will be a problem

Exceptions

- Exceptions give the best of both worlds:
- Write code mostly without worrying about what to do if something goes wrong
- In places where you know what to do if something goes wrong, you can still catch it and handle it
- If you don't handle an exception at all, it aborts the program
- Two sides to exception handling:
    - Throwing
    - Catching

Exceptions

- An exception is an object indicating what went wrong
- The class of the exception object indicates broadly what's wrong
- Extra information in the form of a message string gives details
- Some methods of all exception classes:
  - `toString()` returns a string describing the exception
  - `getMessage()` returns a string with detail about the error, or `null`
  - `printStackTrace()` print a backtrace of what was happening when the exception happened (only useful to programmers)

Throwing

- Throwing an exception signals an error has occurred
- Currently executing code is interrupted and abandoned
- If exception is not caught in the current method, it will flow up to the calling method
- If caller doesn't catch it, it percolates up until it is caught
- If it's never caught, the program is aborted, printing a backtrace
- Use exceptions only for <u>exceptional</u> circumstances, not as a quick way to jump around your code

throw

- Form: `throw` *`exceptionObject`*
- Typically create the exception at the same time: `throw new` *`exceptionClass`*`(`*`detailString`*`);`
- Or: `throw new` *`exceptionClass`*`();`
- For example:

```
public Person(int age, String name) {
    if (name == null) {
        throw new NullPointerException(
            "Creating person with null name");
    }
    this.age = age;
    this.name = name;
}
```

Handling Exceptions

- Handling an exception means providing code to execute to recover from code going wrong
- Form:

```
try {
    code that may go wrong...
} catch (ExceptionClass var) {
    code to handle exception...
}
```

- `try` part specifies code that may throw an exception
- `catch` parameter specifies the kind of exception to catch
- Code inside the `catch` clause specifies what to do if that exception occurs

Executing `try...catch`

- Use `try...catch` like an ordinary statement

```
try {
    code that may go wrong...
} catch (ExceptionClass var) {
    code to handle exception...
}
```

- First execute *code that may go wrong...*
- If it completes without exception, ignore (skip over) *code to handle exception*
- If it throws an exception matching *ExceptionClass*, bind *var* to exception object and execute *code to handle exception*
- Either way, then go on to following statements

Catching

- `catch` parameter has two roles:
  - To specify the kind of exception to catch
  - To name a variable to hold the exception object (`e` is often used as variable name)
- `catch` parameter is scoped to (only defined in) that exception handler
- Can have multiple `catch`es in immediate succession
  - Only one handler is executed, others are ignored
  - First one that matches the thrown exception is used
- If no `catch` clause exception class matches thrown exception, it is not caught by that `try...catch`
  - Propagates to caller as if code were not in a `try...catch`

QuickPoll: What will this code print?

```
try {
    int i = 1;
    if (i > 0) throw new Exception();
    System.out.print("A");
} catch (Exception e) {
    System.out.print("B");
}
System.out.println("C");
```

- A: AC
- B: BC
- C: C
- D: ABC
- E: B

QuickPoll: What will this code print?

```
try {
    int i = 1;
    if (i > 0) throw new Exception();
    System.out.print("A");
} catch (Exception e) {
    System.out.print("B");
}
System.out.println("C");
```

A. AC
B. BC
C. C
D. ABC
E. B

Pitfall: Order of catches

- Always put more specific (descendant class) `catch`es before more general (ancestor class) ones
- More general one will always match when the more specific one would, so more specific one will never be used

```
try {
    ...
} catch (Exception e) {
    // This code will be used for any
    // descendent of Exception class
} catch (NumberFormatException e) {
    // This will never be used: NumberFormatException
    // is a descendent of Exception
}
```

Inside a `catch` block

- `catch` block should try to recover from error
- Can use exception object (commonly parameter `e`)
- `e.getMessage()` returns exception message
- Can use known class of exception to decide action
- If `catch` block cannot resolve the problem, it can always throw the same exception (just `throw e;`)
- Or `catch` block can throw a different exception
- Exception thrown inside `catch` block will not be caught by that `try...catch`
  - Only exceptions thrown inside the `try` block are caught by that `try...catch`

The `finally` Block

- A `finally` block allows you to perform clean ups regardless of whether or not an exception is thrown
- `finally` block is attached to a `try...catch` following all the `catch`es
- Form:

```
try {
    ...
} catch (...) {
    ⋮
} finally {
    code to execute regardless
}
```

- Use this with `try` block, with or without `catch`es

When `finally` Block is Executed

- `finally` block is executed almost no matter what
- If `try` block completes without error, `catch`es are skipped and `finally` block is executed
- If inside `try` an exception with matching `catch` is thrown, `catch` is executed, then `finally` block
- If no matching `catch`, `finally` block executes, then exception propagates to caller
- If matching `catch` block throws exception, `finally` block executes, then exception propagates to caller
- Only if `try` or `catch` is infinite loop, or calls `System.exit`, will `finally` block be missed
    - So don't do that!

Kinds of Exceptions

- Java distinguishes a few different kinds of exceptions:
- Everything that can be `thrown` is a descendent of the `Throwable` class
- Subclasses of `Throwable`:
  - `Error`: a serious error that could occur at any time, such as JVM failure
  - `Exception`: exceptional circumstance in executing your code
- Subclasses of `Exception`:
  - `RuntimeException`: an exception that could happen at any time in normal Java code (*e.g.* null pointer)
  - ...and many, many more specific exception classes

Kinds of Exceptions

- You should not try to catch descendents of `Error`; there is not much you can do about such failures
- Descendents of `RuntimeException` indicate an error that your code has failed to prevent
- Rather than catch `RuntimeException`s, you should fix your code to avoid them, *e.g.*:
  - `NullPointerException`: check that variables are not null before sending them a message
  - `ArrayIndexOutOfBoundsException`: ensure that index is within bounds before accessing array element
  - *Etc.*
- (But it's fine to throw `RuntimeException`s)
- Other descendents of `Exception` should be caught and handled where you know how to handle them

QuickPoll: What will this code print?

```
try {
    String s = null;
    System.out.print(s.toUpperCase());
    System.out.print("A");
} catch (Exception e) { System.out.print("B");
} finally { System.out.print("C");
}
System.out.println("D");
```

- A `ABCD`
- B `AD`
- C `BD`
- D `BCD`
- E a `NullPointerException` error

QuickPoll: What will this code print?

```
try {
    String s = null;
    System.out.print(s.toUpperCase());
    System.out.print("A");
} catch (Exception e) { System.out.print("B");
} finally { System.out.print("C");
}
System.out.println("D");
```

- A. ABCD
- B. AD
- C. BD
- D. BCD
- E. a NullPointerException error

Checked and Unchecked Exceptions

- Descendents of `Error` and `RuntimeException` are called unchecked exceptions
- Other descendents of `Exception` are called checked exceptions
- Each method must declare any checked exceptions that may be propagated out
- "Catch or declare" rule: each method must declare that it may throw:
  - Any checked exceptions that may be thrown by the method, and
  - Any checked exceptions declared to be thrown by methods it calls
  - Except those that are caught and handled by the method body

`throws` clause

- Declare what checked exceptions a method can throw with a `throws` clause
- Form: `throws` *`ExceptionClass`*
- Place this after method signature in concrete or abstract method declaration
- Declare multiple throws by listing all of them separated by commas
- *E.g.*,

```
public LostPerson readLostPersonFile(String fileName)
    throws FileNotFoundException, EOFException { ... }
```

`throws` clause

- A method may not expand overridden method's `throws` clause (but may reduce it)
- Method may throw any descendent of exceptions listed in `throws` clause
- So `throws` clause can generalise multiple exception classes to a common ancestor
- *E.g.*, `IOException` is the base class of both `FileNotFoundException` and `EOFException`, so:

```
public LostPerson readLostPersonFile(String fileName)
    throws IOException { ... }
```

- But this obliges callers to catch or declare all kinds of `IOException`s

Defining Exceptions

- Can define your own exception classes
- Must be descendent of `Exception` class
- Pick the right place in the exception taxonomy
  - *E.g.*, if it's an I/O related error, make it a descendent of `IOException`
- Make sure `getMessage` works right
- Usually you should define constructors with no argument and a single String argument
- Many user exception classes just have constructors:

```
public MyException(String msg) { super(msg); }
public MyException() {
    super("default description string");
}
```

Introduction to Software Engineering

- *Cheops Law: Nothing ever gets built on schedule or within budget.* — Robert Heinlein
- Goes doubly for software projects
- Statistics vary depending on how you define success/failure, but roughly:
  - 1 in 6 projects fail outright
  - 1 in 2 projects complete over budget
  - 1 in 3 projects succeed
- Costs 10s of $Billions every year
- Reasons for failure include:
  - Unrealistic or unarticulated goals
  - Inaccurate estimation of needed resources
  - Badly defined system requirements
  - Poor project management
  - Inability to handle complexity

Victorian Example

- Project started in 2005 to replace the aging LEAP police database Budgeted $50 Million
- In 2009, budget blew out to $130M
- In 2011, new business case saw budget blow out to almost $200M
- Project halted because it didnt handle changing policing requirements
- They're still using LEAP

Victorian Example

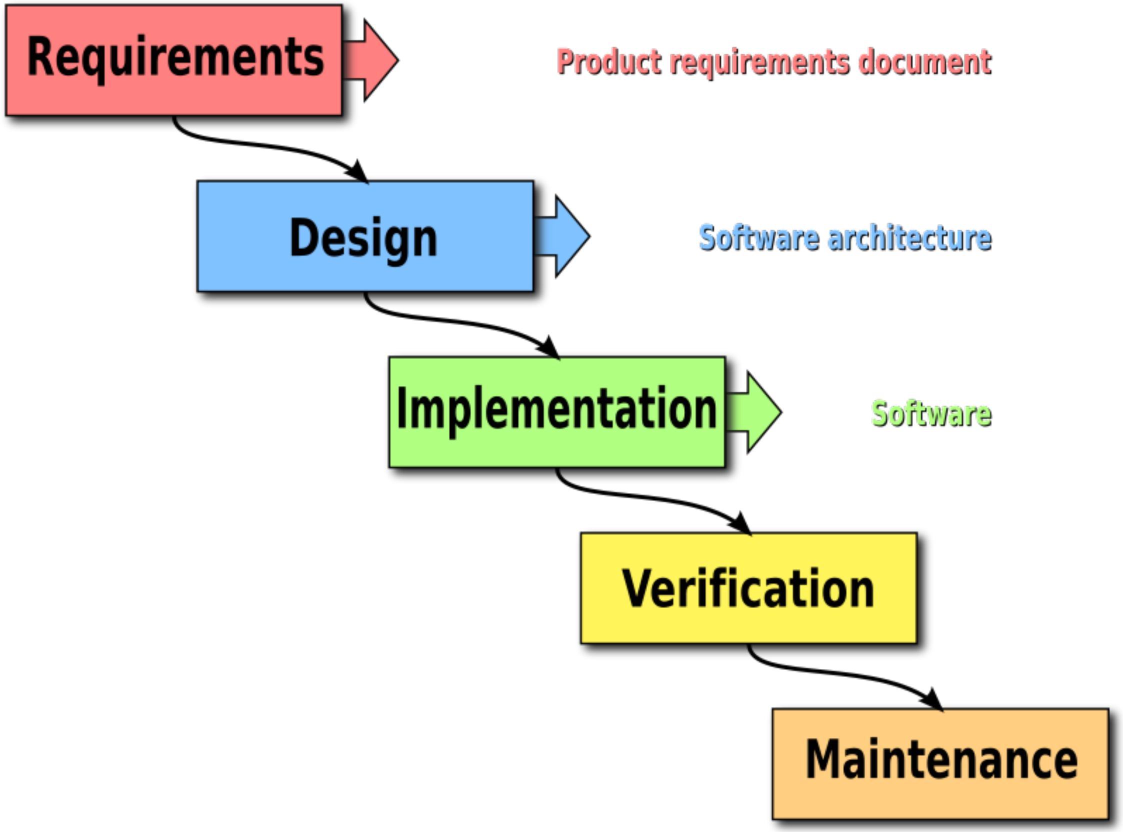
- Project started in 2005 to replace the aging LEAP police database Budgeted $50 Million
- In 2009, budget blew out to $130M
- In 2011, new business case saw budget blow out to almost $200M
- Project halted because it didnt handle changing policing requirements
- They're still using LEAP

Software Engineering

- Software Engineering evolved beginning in the 1960s to solve the "Software Crisis"
- Software development practices and processes are still evolving
- Basic idea: consider phases of software development (software development life cycle):
  - **Requirements specification**: determine what system must do
  - **Design**: determine how to make the system do what it must
  - **Implementation**: code the system
  - **Validation**: make sure the system actually does what it must do
  - **Maintenance**: keep the system doing what it must do as needs change or problems are discovered

Waterfall Model

- Classic life cycle model is the waterfall model
- Phases of development are followed in order



- Phases can backtrack to earlier phase if (when) necessary

Agile Software Development

- The waterfall model often breaks down because:
  - Client rarely know what they need or want
  - Clients and developers rarely communicate visions well
  - Needs change as software is developed
  - Clients get impatient
- Agile development is more popular now
- Agile software development is an incremental software development methodology
- Continuous delivery — delivery every 1–4 weeks
- Focus on communication — daily short meetings
- Focus on client satisfaction and adaptation to changing client needs

Summary

- `throw new ExceptionClass(...)` to signal error
- `try { ...}`
  `catch (ExceptionClass e) { ...}`
  to catch and handle exception
- Use `finally { ...}` to give cleanup code that will almost always be executed
- Must declare uncaught checked exceptions with `throws ExceptionClass` in method declaration
- Unchecked exceptions shouldn't be caught
  - Fix bugs that lead to `RuntimeException`s
- Software engineering focuses on best processes to develop working software on time and within budget

COMP90041
Programming and Software Development

Generic Types

Semester 2, 2015

Generics

- Generic types, or parametric types are types (classes or interfaces) that have parameters
  - Somewhat like methods have parameters
- In Java, the type parameters are always other types
- Arrays are a special parameterised type: you can't declare just an array, only an array of some type
- New in Java 5: you can define classes that take type parameters
- This makes the type system more expressive; allows Java to catch more bugs for you at compile time
- In many cases it allows you to avoid casting

A Non-generic Example

- Java methods can only return one value; what if you want to return two?
- Answer: return an object holding both
- So it would be useful to have a class that can hold two of anything
  - Then we wouldn't need to define a new class every time we want to return a pair of things
- A variable of type `Object` can hold any object
- So just define a type with two `Object` instance variables

A Non-generic Pair Class

```
public class WeakPair {
    private final Object first;
    private final Object second;
    public WeakPair(Object first, Object second) {
        this.first = first;
        this.second = second;
    }
    public Object getFirst() { return first; }
    public Object getSecond() { return second; }
    public String toString() {
        return "(" + first.toString() + "," +
                second.toString() + ")";
    }
}
```

Wrapper Classes

- A primitive value is not an object
- So how can you store a `Pair` of an `int` and a `String`?
- This is what the wrapper classes are for
- Each primitive type has a wrapper class that stores one primitive value
- Each has a one-argument constructor to create the object (boxing)
- Each has a no-argument getter to get back the primitive value (unboxing)
- The wrapper classes are all immutable

Boxing and Unboxing

- Boxing example:

```
integer I = new Integer(42);
```

- Unboxing example:

```
int i = I.intValue();
```

- `Integer` is a descendent of `Object`, so `I` can be stored in a `WeakPair`:

```
WeakPair p1 = new WeakPair(I, "Everything");
```

Autoboxing and Autounboxing

- As of Java 5, Java will automatically box and unbox as necessary
- Autoboxing example:

```
Integer I = 42;
```

- Autounboxing example:

```
int i = I;
```

- Java will even autobox to convert a primitive to the `Object` type:

```
WeakPair p1 = new WeakPair(42, "Everything");
```

What's Wrong with `WeakPair`?

- Weak typing: it's easy to confuse order of arguments, and the compiler can't help

```
WeakPair p1 = new WeakPair(I, "Everything");
WeakPair p2 = new WeakPair("Everything", I);
```

- Java compiler doesn't know correct types of results of `getFirst` and `getSecond`, so you must explicitly down cast

```
int i1 = (int) p.getFirst();
int i2 = (int) p2.getFirst();
```

What's Wrong with `WeakPair`?

- The biggest problem is that mistakes are only discovered at runtime

```
int i1 = (int) p1.getFirst();
int i2 = (int) p2.getFirst();
```

- But `p2.getFirst()` returns a `String`, not an `Integer`
- Can't cast that to an `int`, but compiler doesn't know that
- Only find the problem at runtime

```
Exception in thread "main" java.lang.ClassCastException:
java.lang.String cannot be cast to java.lang.Integer
```

Generic Classes

- Solution: don't just declare object is a pair, but what it is a pair of
  - `p1` is a `Pair` of `Integer` and `String`
  - `p2` is a `Pair` of `String` and `Integer`
- Allow class declaration to specify parameters
- Form: *ClassName*<*Var1*, ...>
- *I.e.*, type variables enclosed in angle brackets are added after class name in class declaration
- Type variables are often single letters (often `T`), or at least very short
- The parameters *Var1*, ... are used as types inside the class definition

A Generic Pair Class

```
public class Pair<T1,T2> {
    private final T1 first;
    private final T2 second;
    public Pair(T1 first, T2 second) {
        this.first = first;
        this.second = second;
    }
    public T1 getFirst() { return first; }
    public T2 getSecond() { return second; }
    public String toString() {
        return "(" + first.toString() + ","
                + second.toString() + ")";
    }
}
```

Using a Generic Class

- For a generic class, class name alone is not a type; must supply type arguments
- Form: *ClassName*<*Type1*, ...>
- To construct a new object of generic type, specify both type arguments and constructor arguments
- Form:
  new *ClassName*<*Type1*, ...>(*expr1*, ...)
- *E.g.*:

```
Pair<Integer,String> p1 =
      new Pair<Integer,String>(I, "Everything");
Pair<String,Integer> p2 =
      new Pair<String,Integer>("Everything", I);
```

Why This is Better

- Java compiler can track types through the code
- Can check that constructor stores right types of arguments in instance variables, because types agree
- Can check that accessors return the right types

```
int i2 = (int) p2.getFirst();
```

- The compiler now knows that `p2.getFirst()` returns a `String` rather than `Object`
- Now compiler points out the error:

```
PairTest.java:18: error: incompatible types: String can
not be converted to int
        int i2 = (int) p2.getFirst();
                                  ^
```

QuickPoll: Which one of these is illegal?

Given these declarations:

```
Integer i;
int j;
String s, t;
Pair<String,Integer> p, q;
```

- A `... = new Pair<Integer,Integer>(i,j);`
- B `String u = q.getFirst();`
- C `... = new Pair<Integer,int>(i,j);`
- D `int k = p.getSecond();`
- E `... = new Pair<Pair<String,Integer>,String>(p,s);`

QuickPoll: Which one of these is illegal?

Given these declarations:

```
Integer i;
int j;
String s, t;
Pair<String,Integer> p, q;
```

- A `... = new Pair<Integer,Integer>(i,j);`
- B `String u = q.getFirst();`
- C `... = new Pair<Integer,int>(i,j);`
- D `int k = p.getSecond();`
- E `... = new Pair<Pair<String,Integer>,String>(p,s);`

Pitfall: Generic Constructor Syntax

- To use a generic type constructor, give type parameters, *e.g.*:

```
Pair<Integer,String> p1 =
      new Pair<Integer,String>(I, "Everything");
```

- But define constructor without type parameter:

```
    public Pair(T1 first, T2 second) {
        this.first = first;
        this.second = second;
    }
```

Pitfall: Can't Instantiate Parameter Type

- Inside generic class, Java knows nothing about type variable
- So you can't use its constructor; this won't work:

```
    public Pair() {
        this.first = new T1();
        this.second = new T2();
    }
```

```
Pair.java:13: error: unexpected type
        this.first = new T1();
                         ^
  required: class
  found:    type parameter T1
  where T1 is a type-variable:
    T1 extends Object declared in class Pair
```

Pitfall: Can't Create Generic Type Array

- You can't make an array with generic element type

```
    public Pair(T1 first) {
        T1[] a = new T1[1];
        a[0] = first;
    }
```

```
Pair.java:13: error: generic array creation
        T1[] a = new T1[1];
                 ^
```

- This is a perfectly reasonable thing to want to do
- Java doesn't allow it because it throws away all information about generics during compilation
- Workaround is beyond the scope of this subject

Comparable Done Right

- The `Comparable` interface as discussed earlier works, but is clumsy
- It does not ensure in `o1.compareTo(o2)` that `o1` and `o2` are of same class
- A class that implements `Comparable` also needs to check that argument is right type and cast object
- As of Java 5, `Comparable` is a generic interface
- `Comparable<T>` means a type that can be compared with type `T`

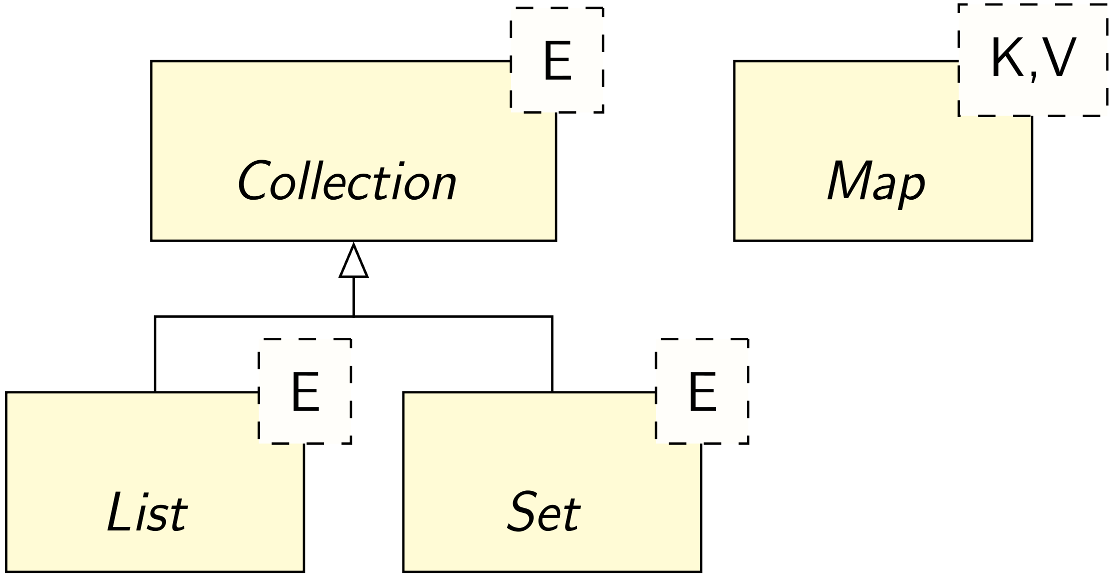
Implementing a Comparable Class

- Make `Person` comparable
  - Order first by name; for same name, order by age

```
public class Person implements Comparable<Person> {
    private int age;
    private String name;
    ⋮
    public int compareTo(Person o) {
        int result = name.compareTo(o.name);
        if (result == 0) {
            result = age - o.age;
        }
        return result;
    }
}
```

Java Collections Framework

- Java library provides several convenient classes and interfaces for working with collections of object
- These are the most important interfaces in the hierarchy:



- All are generic; template parameters are shown in dotted boxes

Collection<E> interface

- All Collection classes have no-argument constructor returning empty collection
- All are expandable to unbounded size
- All support foreach loop syntax
- All support these methods:

Method	**Action / Result**
isEmpty()	Is collection empty?
size()	Number of elements in collection
add(E e)	Add e; true if collection changed
contains(Object o)	Does collection contain o?
remove(Object o)	Remove o if present; true if it was
clear()	Remove all elements
toArray()	Collection content as an array

List<E> interface

- List<E> is a collection that preserves order and duplicates; similar to an array
- In addition to methods of Collection<E>, List<E> supports these:

Method	**Action / Result**
get(int idx)	Return element at index idx
set(int idx, E elt)	Replace object at index idx with elt
indexOf(Object o)	Return first index of o, or -1 if absent
lastIndexOf(Object o)	Return last index of o, or -1 if absent
add(int idx, E elt)	Insert elt at index idx
remove(int idx)	Remove object at index idx

Most important `List<E>` Classes

- `ArrayList<E>` — usually the class you want
  - Fast to add to the end
  - Fast to get element by index
  - Slow to add in the middle
- `LinkedList<E>`
  - Fast to add to either end
  - Fast to insert anywhere
  - Slow to get element by index
- Only matters when you have thousands of elements
- Use `List<E>` interface as variable and parameter type — then code will work for either class of list
- But you `must` use `ArrayList<E>` or `LinkedList<E>` class with `new`

QuickPoll: What Will This Print?

```
ArrayList<String> list = new ArrayList<String>();
list.add("one");
list.add("two");
list.add(1, "three");
list.add(1, "four");
for (String s : list) System.out.print(s + " ");
System.out.println();
```

- A `one two three four`
- B `four three two one`
- C `four three one two`
- D `one four three two`
- E `three four one two`

QuickPoll: What Will This Print?

```
ArrayList<String> list = new ArrayList<String>();
list.add("one");
list.add("two");
list.add(1, "three");
list.add(1, "four");
for (String s : list) System.out.print(s + " ");
System.out.println();
```

- A `one two three four`
- B `four three two one`
- C `four three one two`
- D `one four three two`
- E `three four one two`

`Set<E>` interface

- `Set<E>` does not preserve order or duplicates
- `Set<E>` does not support methods beyond those of `Collection<E>`
- `HashSet<E>` implements `Set<E>` — usually the class you want
  - Fast to add and remove elements
  - Fast to check if object is in set (faster than either kind of `list<E>`)

`Map<K,V>` interface

- A map is sometimes called a dictionary or lookup table or finite function
- It associates a single value of type `V` with each key of type `K`
- *E.g.*, it could let you quickly look up the `Person` object (value) with a given name (key)
- This is very much faster than checking every `Person` object looking for the right name

Map<K,V> interface methods

Method	**Action / Result**
`isEmpty()`	Is the map empty?
`size()`	The number of key-value mappings
`get(Object key)`	Return object mapped to by `key`, or `null`
`put(K key, V value)`	Make `value` the mapping for `key`
`remove(Object key)`	Remove the mapping for `key`, if any
`clear()`	Empty the map

- Most common map class is `HashMap<K,V>` — good performance

Summary

- Generic types have types as parameters written in `<Angle,Brackets>`
- In class declaration, parameters are type variables, which can be used as types in the declaration
- In variable declaration, parameters should be reference types
- All primitive types have wrapper classes; Java will convert
- Use `ArrayList` for lists `HashSet` for sets, `HashMap` for maps