Principles of Software Construction: Objects, Design and Concurrency

Method dispatch, Object contracts, and Exceptions (Oh my!)

Christian Kästner    Charlie Garrod
Adminstrivia

- Section D...
Key concepts from Thursday
Key concepts from Thursday

• More inheritance-related details
  ▪ super, this, instanceof, final
  ▪ type-casting

• Typechecking

• Method dispatch
  ▪ Overloaded method names
  ▪ Overriding inherited methods
Key concepts for today

• Method dispatch practice

• The `java.lang.Object`
  • Behavioral contracts
  • A lesson in equality

• Introduction to Exceptions
Method dispatch, revisited

e.g.: `x.foo(apple, 42)`

- Step 1 (compile time): determine which class to look in
- Step 2 (compile time): determine the method signature to be executed
- Step 3 (run time): Determine the run-time class of the receiver
- Step 4 (run time): Locate the method to invoke
Method dispatch practice

```java
public class GenericAnimal {
    public String getNoise() { return "Noise"; }
}

public class Bird extends GenericAnimal {
    public String getNoise() { return "Chirp"; }
}

public class Cat extends GenericAnimal {
    public String getNoise() { return "Meow"; }
}

public class GenericDog extends GenericAnimal {
    // nothing special to hear here
}

public class Ewokian extends GenericDog {
    public String getNoise() { return "Oonga!"; }
}
```
Method dispatch practice, part A

```java
public class GenericAnimal {
    public String getNoise() { return "Noise"; }
}

public class Cat extends GenericAnimal {
    public String getNoise() { return "Meow"; }
}

public class Bird extends GenericAnimal {
    public String getNoise() { return "Chirp"; }
}

public class GenericDog extends GenericAnimal {
    // nothing special to hear here
}

public class Ewokian extends GenericDog {
    public String getNoise() { return "Oonga!"; }
}

public class GenericAnimal A = new GenericAnimal();
System.out.print(A.getNoise());
```

What is printed by:
public class GenericAnimal {
    public String getNoise() { return "Noise"; }
}

class Bird extends GenericAnimal {
    public String getNoise() { return "Chirp"; }
}

class Cat extends GenericAnimal {
    public String getNoise() { return "Meow"; }
}

class GenericDog extends GenericAnimal {
    // nothing special to hear here
}

class Ewokian extends GenericDog {
    public String getNoise() { return "Oonga!"; }
}

Bird B = new Bird();
System.out.print(B.getNoise());
Method dispatch practice, part B-2

```java
public class GenericAnimal {
    public String getNoise() { return "Noise"; }
}

public class Bird extends GenericAnimal {
    public String getNoise() { return "Chirp"; }
}

public class Cat extends GenericAnimal {
    public String getNoise() { return "Meow"; }
}

public class GenericDog extends GenericAnimal {
    // nothing special to hear here
}

public class Ewokian extends GenericDog {
    public String getNoise() { return "Oonga!"; }
}

public class GenericAnimal B = new Bird();
System.out.println(B.getNoise());
```
Method dispatch practice, part C

```java
public class GenericAnimal {
    public String getNoise() { return "Noise"; }
}

public class Bird extends GenericAnimal {
    public String getNoise() { return "Chirp"; }
}

public class Cat extends GenericAnimal {
    public String getNoise() { return "Meow"; }
}

public class GenericDog extends GenericAnimal {
    // nothing special to hear here
}

public class Ewokian extends GenericDog {
    public String getNoise() { return "Oonga!"; }
}

public class GenericAnimal C = new Cat();
System.out.print(C.getNoise());
```

What is printed by:
public class GenericAnimal {
    public String getNoise() { return "Noise"; }
}

public class Bird extends GenericAnimal {
    public String getNoise() { return "Chirp"; }
}

public class Cat extends GenericAnimal {
    public String getNoise() { return "Meow"; }
}

public class GenericDog extends GenericAnimal {
    // nothing special to hear here
}

public class Ewokian extends GenericDog {
    public String getNoise() { return "Oonga!"; }
}

GenericAnimal D = new GenericDog();
System.out.print(D.getNoise());
Method dispatch practice, part E-1

```java
public class GenericAnimal {
    public String getNoise() { return "Noise"; }
}
```

```java
public class Bird extends GenericAnimal {
    public String getNoise() { return "Chirp"; }
}
```

```java
public class Cat extends GenericAnimal {
    public String getNoise() { return "Meow"; }
}
```

```java
public class GenericDog extends GenericAnimal {
    // nothing special to hear here
}
```

```java
public class Ewokian extends GenericDog {
    public String getNoise() { return "Oonga!"; }
}
```

What is printed by:

```java
GenericAnimal E = new Ewokian();
System.out.print(E.getNoise());
```
public class GenericAnimal {
    public String getNoise() { return "Noise"; }
}

public class Bird extends GenericAnimal {
    public String getNoise() { return "Chirp"; }
}

public class Cat extends GenericAnimal {
    public String getNoise() { return "Meow"; }
}

public class GenericDog extends GenericAnimal {
    // nothing special to hear here
}

public class Ewokian extends GenericDog {
    public String getNoise() { return "Oonga!"; }
}

What is printed by:

Ewokian E = new Ewokian();
GenericAnimal F = E;
System.out.print(F.getNoise());
The `java.lang.Object`

- All Java objects inherit from `java.lang.Object`

- Commonly-used/overridden public methods:
  
  ```java
  String toString()
  boolean equals(Object obj)
  int hashCode()
  Object clone()
  ```
public class Object {
    String toString() { ... }
    boolean equals(Object obj) { ... }
    int hashCode() { ... }
    Object clone() { ... }
}

public class Point {
    final private int x, y;
    public Point(int px, int py) { x = px; y = py; }
    public String toString {
        return "(" + x + ", " + y + ")";
    }
    public boolean equals(Point p) {
        return x == p.x && y == p.y;
    }
    public int hashCode() {
        return toString().hashCode();
    }
    ...
Overriding java.lang.Object’s .equals

• The default .equals:

```java
class Object {
    public boolean equals(Object obj) {
        return this == obj;
    }
}
```

• An aside: Do you like:

```java
class CheckingAccountImpl implements CheckingAccount {
    @Override
    public boolean equals(Object obj) {
        return false;
    }
}
```
The `.equals(Object obj)` contract

- An equivalence relation
  - Reflexive: ∀x x.equals(x)
  - Symmetric: ∀x,y x.equals(y) if and only if y.equals(x)
  - Transitive: ∀x,y,z x.equals(y) and y.equals(z) implies x.equals(z)

- Consistent
  - Invoking x.equals(y) repeatedly returns the same value unless x or y is modified

- x.equals(null) is always false
The `hashCode()` contract

- **Consistent**
  - Invoking `x.hashCode()` repeatedly returns same value unless `x` is modified

- **Equality implies `hashCode()` equality**
  - i.e., `x.equals(y)` implies `x.hashCode() == y.hashCode()`
  - The reverse implication is not necessarily true:
    - `x.hashCode() == y.hashCode()` does not imply `x.equals(y)`

- **Advice:** You should override `.equals()` if and only if you override `.hashCode()`
The `.clone()` contract

- Returns a *deep copy* of an object
- Generally (but not necessarily!):
  - `x.clone() != x`
  - `x.clone().equals(x)`
A lesson in equality

public class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

Recall: The java.lang.Object

- All Java objects inherit from java.lang.Object
- Commonly-used/overridden public methods:
  - String toString()
  - boolean equals(Object obj)
  - int hashCode()
  - Object clone()

Implement the .equals method for the Point class.
A tempting but incorrect solution

```java
public class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

public boolean equals(Point p) {
    return x == p.x && y == p.y;
}
```

Types must match

Recall: The java.lang.Object

- All Java objects inherit from java.lang.Object
- Commonly-used/overridden public methods:
  - String toString()
  - boolean equals(Object obj)
  - int hashCode()
  - Object clone()
A correct solution

public class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public boolean equals(Object obj) {
        if (!(obj instanceof Point))
            return false;
        Point p = (Point) obj;
        return x == p.x && y == p.y;
    }
}
A new challenge

Implement .equals for the ColorPoint class. You may assume Color correctly implements .equals.
A tempting solution

public class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public boolean equals(Object obj) {
        if (! (obj instanceof Point))
            return false;
        Point p = (Point) obj;
        return x == p.x && y == p.y;
    }
}

public class ColorPoint
extends Point {
    private final Color color;
    public ColorPoint(int x, int y, Color color) {
        super(x, y);
        this.color = color;
    }
    public boolean equals(Object obj) {
        if (! (obj instanceof ColorPoint))
            return false;
        ColorPoint cp = (ColorPoint) obj;
        return super.equals(cp) &&
                color.equals(cp.color);
    }
}
public class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public boolean equals(Object obj) {
        if (!(obj instanceof Point))
            return false;
        Point p = (Point) obj;
        return x == p.x && y == p.y;
    }
}

A problem: \texttt{p.equals(cp)}
but \texttt{!cp.equals(p)}:

Point p = new Point(2, 42);
ColorPoint cp = new ColorPoint(2, 42, Color.BLUE);

public class ColorPoint extends Point {
    private final Color color;

    public ColorPoint(int x, int y, Color color) {
        super(x, y);
        this.color = color;
    }

    public boolean equals(Object obj) {
        if (!(obj instanceof ColorPoint))
            return false;
        ColorPoint cp = (ColorPoint) obj;
        return super.equals(cp) &&
                color.equals(cp.color);
    }
}
public class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public boolean equals(Object obj) {
        if (!((obj instanceof Point))
            return false;
        Point p = (Point) obj;
        return x == p.x && y == p.y;
    }
}

public class ColorPoint extends Point {
    private final Color color;

    public ColorPoint(int x, int y, Color color) {
        super(x, y);
        this.color = color;
    }

    public boolean equals(Object obj) {
        if (!((obj instanceof Point))
            return false;
        if (!((obj instanceof ColorPoint))
            return super.equals(obj);
        ColorPoint cp = (ColorPoint) obj;
        return super.equals(cp) &&
        color.equals(cp.color);
    }
}

Point p = new Point(2, 42);
ColorPoint cp1 = new ColorPoint(2, 42, Color.BLUE);
ColorPoint cp2 = new ColorPoint(2, 42, Color.MAUVE);
public abstract class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public boolean equals(Object obj) {
        if (!(obj instanceof Point))
            return false;
        Point p = (Point) obj;
        return x == p.x && y == p.y;
    }
}

public class ColorPoint extends Point {
    private final Color color;

    public ColorPoint(int x, int y, Color color) {
        super(x, y);
        this.color = color;
    }

    public boolean equals(Object obj) {
        if (!(obj instanceof ColorPoint))
            return false;
        ColorPoint cp = (ColorPoint) obj;
        return super.equals(cp) &&
            color.equals(cp.color);
    }
}

public class PointImpl extends Point {
    public PointImpl(int x, int y) { super(x,y); }
    public boolean equals(Object obj) {
        if (!(obj instanceof PointImpl))
            return false;
        return super.equals(obj);
    }
}
The lesson

- Conforming to behavioral contracts can be difficult
  - Even simple contracts like equality are difficult to implement when inheriting from a non-abstract class
Key concepts for today

• Method dispatch practice

• The java.lang.Object
  - Behavioral contracts
  - A lesson in equality

• Introduction to Exceptions
What does this code do?

```java
FileInputStream fIn = new FileInputStream(filename);
if (fIN == null) {
    switch (errno) {
    case _ENOFILE:
        System.err.println("File not found: " + ...);
        return -1;
    default:
        System.err.println("Something else bad happened: " + ...);
        return -1;
    }
}
DataInput dataInput = new DataInputStream(fIn);
if (dataInput == null) {
    System.err.println("Unknown internal error.");
    return -1; // errno > 0 set by new DataInputStream
}
int i = dataInput.readInt();
if (errno > 0) {
    System.err.println("Error reading binary data from file");
    return -1;
} // I didn’t have enough room to close the file. Oh well.
return i;
```
Exceptions

• Exceptions notify the caller of an exceptional circumstance (usually operation failure)

• Semantics
  ▪ An exception propagates *up the function-call stack* until `main()` is reached or until the exception is caught

• Sources of exceptions:
  ▪ Programmatically throwing an exception
  ▪ Exceptions thrown by the Java runtime
What does this code do?

```java
FileInputStream fIn = new FileInputStream(filename);
if (fIN == null) {
    switch (errno) {
        case _ENOFILE:
            System.err.println("File not found: " + ...);
            return -1;
        default:
            System.err.println("Something else bad happened: " + ...);
            return -1;
    }
}
DataInput dataInput = new DataInputStream(fIn);
if (dataInput == null) {
    System.err.println("Unknown internal error.");
    return -1;  // errno > 0 set by new DataInputStream
}
int i = dataInput.readInt();
if (errno > 0) {
    System.err.println("Error reading binary data from file");
    return -1;
}  // I didn’t have enough room to close the file. Oh well.
return i;
```
try {
    FileInputStream fileInput = new FileInputStream(filename);
    DataInput dataInput = new DataInputStream(fileInput);
    int i = dataInput.readInt();
    fileInput.close();
    return i;
} catch (FileNotFoundException e) {
    System.out.println("Could not open file " + filename);
    return -1;
} catch (IOException e) {
    System.out.println("Error reading binary data from file " + filename);
    return -1;
}
### Exceptional control-flow

```java
try {
    System.out.println("Top");
    int[] a = new int[10];
    a[42] = 42;
    System.out.println("Bottom");
} catch (IndexOutOfBoundsException e) {
    System.out.println("Caught index out of bounds");
}
```

- **Prints:**
  - Top
  - Caught index out of bounds
Exceptional control-flow, part 2

public static void test() {
    try {
        System.out.println("Top");
        int[] a = new int[10];
        a[42] = 42;
        System.out.println("Bottom");
    } catch (NegativeArraySizeException e) {
        System.out.println("Caught negative array size");
    }
}

public static void main(String[] args) {
    try {
        test();
    } catch (IndexOutOfBoundsException e) {
        System.out.println("Caught index out of bounds");
    }
}

• Prints:
  Top
  Caught index out of bounds
The finally keyword

• The finally block always runs after try/catch:

```java
try {
    System.out.println("Top");
    int[] a = new int[10];
    a[42] = 42;
    System.out.println("Bottom");
} catch (IndexOutOfBoundsException e) {
    System.out.println("Caught index out of bounds");
} finally {
    System.out.println("Finally got here");
}
```

• Prints:
  Top
  Caught index out of bounds
  Finally got here
The `finally` keyword, part 2

- The finally block always runs after try/catch:

```
try {
    System.out.println("Top");
    int[] a = new int[10];
    a[2] = 2;
    System.out.println("Bottom");
} catch (IndexOutOfBoundsException e) {
    System.out.println("Caught index out of bounds");
} finally {
    System.out.println("Finally got here");
}
```

- Prints:
  - Top
  - Bottom
  - Finally got here
Benefits of exceptions

- Provide high-level summary of error and stack trace
  - Compare: core dumped in C

- Can’t forget to handle common failure modes
  - Compare: using a flag or special return value

- Can optionally recover from failure
  - Compare: calling System.exit()

- Improve code structure
  - Separate routine operations from error-handling

- Allow consistent clean-up in both normal and exceptional operation
The exception hierarchy

Object
  ↓
Exception
  ↓
RuntimeException
  ↓
...          IOException
  ↓          ↓
NullPointerException          EOFException
  ↓          ↓
IndexOutOfBoundsException          FileNotFoundException
  ↓          ↓
...          ClassNotFoundException
...          ...
Checked and unchecked exceptions

• Unchecked exception: any subclass of `RuntimeException`
  ▪ Indicates an error which is highly unlikely and/or typically unrecoverable

• Checked exception: any subclass of `Exception` but not `RuntimeException`
  ▪ Indicates an error that every caller should be aware of and explicitly decide to handle or pass on
Creating and throwing your own exceptions

• Methods must declare any checked exceptions they might throw

• If your class extends java.lang.Exception you can throw it:

```java
if (someErrorBlahBlahBlahBlah) {
    throw new MyCustomException("Blah blah blah");
}
```

• See ReadFromFile examples and IllegalBowlingScoreException and ReadBowlingScore example
Guidelines for using exceptions

- Catch and handle all checked exceptions
  - Unless there is no good way to do so...

- Use runtime exceptions for programming errors

- Other good practices
  - Do not catch an exception without (at least somewhat) handling the error
  - When you throw an exception, describe the error
  - If you re-throw an exception, always include the original exception as the cause