Effective Java

Welcome!

Please tell us your:

- Name
- Responsibility
- Background in Java
- Expectations

Course Sections

1. Creating and destroying objects
2. Common object methods
3. Mastering classes and interfaces
4. Generics
5. Enumerations
6. Mastering methods
7. Programming techniques
8. Working with Exceptions

1: Creating and Destroying Objects

Objectives

- Static factory methods
- Builders
- Singletons
- Avoiding creating unnecessary objects

Boolean's `valueOf` Method

Code: ch01/Test

- To get an instance of a Boolean, say for `True`, you almost always use `Boolean.valueOf(true)` instead of `new Boolean(true)`
  - With former, you may compare two instances like this:

```
Boolean b1 = Boolean.valueOf(true);
Boolean b2 = Boolean.valueOf(true);
// b1==b2 will be true
```

  - With `new`, you can't:

```
Boolean b1 = new Boolean(true);
Boolean b2 = new Boolean(true);
// b1==b2 will  be false
```

Static Factory Method (SFM)

- The above `valueOf` method is a *Static Factory Method*
- A class may provide both regular constructors and SFMs
- SFMs are different from Factory Methods from “Gang of Four” *Design Patterns*
- Advantages of SFMs:
  - Can have meaningful names
  - Don’t have to return a new object each time it’s called
  - Can return an object of any subtype of the return type
  - Reduce the verbosity of creating parameterized type instances

SFMs Can Have Meaningful Names

- An SFM with a well-chosen name is easier to use and the resulting client code easier to read
  - For example, `BigInteger` has an SFM `probablePrime`
- A class can have only a single constructor with a given signature; SFMs don't share the restriction
  - If a class requires multiple constructors with the same signature, replace the constructors with SFMs and carefully chosen names to highlight their differences

SFMs Don't Have to Return a New Object

- Allow classes to use pre-constructed instances, or cache instances as they're constructed, and dispense them repeatedly to avoid creating unnecessary duplicate objects
  - For example, `Boolean`'s `valueOf` method
- Can greatly improve performance if equivalent objects are requested often, especially if they are expensive to create
- Allow classes to maintain strict control over what instances exist at any time
  - Called *instance-controlled*

SFMs Can Return an Object of Subtype

- Can return an object of any subtype of the return type
- Can return objects without making their classes public
  - As long as the classes implement a public interface
  - E.g., `java.util.Collections` provides many convenient SFMs for creating various kinds of collections

```
static <T> Collection<T>
  synchronizedCollection(Collection<T> c)
static <K,V> Map<K,V>
  synchronizedMap(Map<K,V> m)
...
```

  - They all return objects of nonpublic classes

SFMs Reduce Verbosity

- Using a constructor to create an instance of a parameterized class typically requires you to provide the type parameters twice, e.g.:

```
MyCoolMap<String, List<String>> m =
  new MyCoolMap<String, List<String>>();
```

- With SFMs, the compiler can figure out the type parameters for you - known as *type inference*
  - E.g. suppose an SFM `getInstance` is defined for `MyCoolMap`, then you may use:

```
 MyCoolMap<String, List<String>> m =
   MyCoolMap.getInstance();
```


Disadvantages of Providing Only SFMs

- Classes without public or protected constructors cannot be sub-classed
- Not readily distinguishable from other static methods
  - Some common names for static factory methods:
    - valueOf
    - getInstance
    - newInstance
    - get*Type*
    - new*Type*

Constructors with Many Optional Params

- If a class has many optional fields, how do you provide the constructors?
  - Telescoping: provide one that takes all the required parameters, second one with one optional parameter, third with two optional parameters, …
    - Hard to read and write
  - Use `set` methods to set optional fields
    - Object may be in an inconsistent state partway through its construction
    - Impossible to make a class immutable

Builder Pattern to the Rescue

- The client calls a constructor (or static factory) with all of the required parameters and gets a builder object
- Then the client calls setter-like methods on the builder object to set each optional parameter of interest
- Finally, the client calls a parameterless `build` method to generate the object, which is immutable
- The builder is a static member class of the class it builds

A Customer Class Using Builder Pattern

Code: ch01/Customer

```
public class Customer {
    private final String firstName;
    private final String lastName;
    private final String middleName;
    private final String streetAddress;
    private final String city;
    private final String state;
    private final String zipCode;
    private final String homePhone;
    private final String cellPhone;
    private final String email;

    public static class Builder {
            // Required parameters
        private final String firstName;
        private final String lastName;
```

A Customer Class Using Builder Pattern

```
 // Optional parameters
private String middleName = null;
private String streetAddress = null;
private String city = null;
private String state = null;
private String zipCode = null;
private String homePhone = null;
private String cellPhone = null;
private String email = null;

public Builder(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
}
```

A Customer Class Using Builder Pattern

```
public Builder middleName(String s) {
    this.middleName = s;
    return this;
}
public Builder streetAddress(String s) {
    this.streetAddress = s;
    return this;
}
public Builder city(String s) {
    this.city = s;
    return this;
}
public Builder state(String s) {
    this.state = s;
    return this;
}
```

A Customer Class Using Builder Pattern

```
public Builder zipCode(String s) {
    this.zipCode = s;
    return this;
}
public Builder cellPhone(String s) {
    this.cellPhone = s;
    return this;
}
public Builder homePhone(String s) {
    this.homePhone = s;
    return this;
}
public Builder email(String s) {
    this.email = s;
    return this;
}
```

A Customer Class Using Builder Pattern

```
        public Customer build() {
            return new Customer(this);
        }
    }
    private Customer(Builder builder) {
        this.firstName = builder.firstName;
        this.lastName = builder.lastName;
        this.middleName = builder.middleName;
        this.streetAddress = builder.streetAddress;
        this.city = builder.city;
        this.state = builder.state;
        this.zipCode = builder.zipCode;
        this.homePhone = builder.homePhone;
        this.cellPhone = builder.cellPhone;
        this.email = builder.email;
    }
}
```

To Use the Customer Class

```
Customer joe = new Customer.Builder("Joe", "Smith").
      streetAddress("10000 Research BLVD").city("Austin").
      state("TX").zipCode("78759").build();
```


Disadvantages of Builder Pattern

- In order to create an object, you must first create its builder
  - Could be a problem in some performance critical situations
- More verbose than the telescoping constructor pattern
- Should be used only if there are enough parameters

Enforcing Singletons

- A *singleton* is simply a class that is instantiated exactly once
- Before Java 5.0, two common singleton implementations:
  - Singleton with public final field
  - Singleton with an SFM
- Since Java 5.0, one can use a single-element enum type

Singleton with Public Final Field

```
public class Singleton {
   public static final Singleton INSTANCE =
        new Singleton();
   private Singleton() { ... }
   ...
}
```

- Usage:

```
Singleton singleton = Singleton.INSTANCE;
```

- Advantage: declarations make it clear that the class is a singleton

Singleton with an SFM

```
public class Singleton {
    private static final Singleton INSTANCE = new Singleton();
    private Singleton() { ... }
    public static Singleton getInstance() { return INSTANCE; }
    ...
}
```

- Usage:

```
Singleton singleton = Singleton.getInstance();
```

- Advantage: flexibility - may change the class to be a non-singleton without changing the API

Singleton with a Single-Element Enum

```
public enum Singleton {
    INSTANCE;
    ...
}
```

- Usage:

```
Singleton singleton = Singleton.INSTANCE;
```

Serialization of Singletons

- Using first two approaches, implementing a `Serializable` interface is not enough:
  - Each time a serialized instance is de-serialized, a new instance will be created
  - To fix:
    - Declare all instance fields transient, and
    - Provide a `readResolve` method

```
private Object readResolve() {
    return INSTANCE;
}
```

- Using single-element enum, you get serialization for free
  - All enums extend `java.lang.Enum` which implements `Serializable`

Avoid Creating Unnecessary Objects ...

- Can you spot what's wrong with this program?

```
public static void main(String[] args) {
  Long sum = 0L;
  for (long i = 0; i < Integer.MAX_VALUE; i++) {
      sum += i;
  }
  System.out.println(sum);
}
```

- Because of autoboxing, the program constructs about 2^{31} unnecessary `Long` instances
- Lesson: prefer primitives to boxed primitives, and watch out for unintentional autoboxing

Avoid Creating Unnecessary Objects

- Use SFMs to construct objects *when it's appropriate* to reuse objects
- But don't take it too far - maintaining your own object pools in general is not recommended unless the objects in the pool are extremely heavyweight
- Classic examples that justify for object pools
  - Database connections
  - Threads

Summary

- Use SFMs to create objects when appropriate
- You may use builders to construct objects when the objects have many optional fields
- In addition to the usual ways of creating singletons, you may also use single-element enums
- Avoid creating unnecessary object instances

Exercise

- In the labs, do the exercises:
  - **Create a ConfigDepot Class**
  - **Use Builder Pattern to Create Objects**
  - **Create Singleton ConfigDepot**

2: Common Object Methods

Objectives

- `equals` and `hashCode` methods
- Implementing `toString`
- Cloning objects

`Object`'s `equals` Method

- For any non-null *reference* values `x` and `y`, `Object`'s `equals` method returns `true`, if and only if `x` and `y` refer to the same object
  - `x == y` is `true`
- It is not "logical" equal
- Any class that wants `equals` to mean "logical" equal has to override it

Requirements for `equals`

- *Reflexive*: `x.equals(x)` must return `true`
- *Symmetric*: `x.equals(y)` must return `true` if and only if `y.equals(x)` returns `true`
- *Transitive*: if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` must return `true`
- *Consistent*: multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, if no information used in `equals` comparisons on the objects is modified.
- `x.equals(null)` must return `false`

Pitfalls in Implementing `equals`

Code: ch02/Point2D

- Suppose we have defined an `equals` method for a 2D point class:

```
public class Point2D {
    private double x;
    private double y;
    public Point2D(double x, double y) {
        this.x = x;
        this.y = y;
    }
    @Override public boolean equals(Object o) {
        if (o == this) return true;
        if (!(o instanceof Point2D))
            return false;
        Point2D p = (Point2D)o;
        return p.x == x && p.y == y;
    }
   ...
}
```

Pitfalls in Implementing `equals`

Code: ch02/Point3D

- We now extend the 2D Point to a 3D Point class:

```
public class Point3D extends Point2D {
    private double z;
    public Point3D(double x, double y, double z) {
        super(x, y);
        this.z = z;
    }
     // Broken - violates symmetry!
    @Override public boolean equals(Object o) {
        if (!(o instanceof Point3D))
            return false;
        return super.equals(o) && ((Point3D) o).z == z;
    }
}
```

Pitfalls in Implementing `equals` …

- Let’s test it out:

```
public static void main(String[] args) {
    Point2D p1 = new Point2D(1.0, 2.0);
    Point3D p2 = new Point3D(1.0, 2.0, 3.0);
    System.out.println("p1.equals(p2) is " + (p1.equals(p2)));
    System.out.println("p2.equals(p1) is " + (p2.equals(p1)));
}
```

- Output:

```
p1.equals(p2) is true
p2.equals(p1) is false
```

- It violated the symmetry rule!

Pitfalls in Implementing `equals` …

- If you change the `Point3D` `equals` method to:

```
@Override public boolean equals(Object o) {
    if (!(o instanceof Point2D))
        return false;
        // If o is a 2D Point, do a 2D comparison
    if (!(o instanceof Point3D))
        return o.equals(this);
        // o is a 3D point; do a full comparison
    return super.equals(o) && ((Point3D)o).z == z;
}
```

Pitfalls in Implementing `equals` …

- Let’s test it:

```
public static void main(String[] args) {
    Point3D p1 = new Point3D(1.0, 2.0, 3.0);
    Point2D p2 = new Point2D(1.0, 2.0);
    Point3D p3 = new Point3D(1.0, 2.0, 4.0);
    System.out.println("p1.equals(p2) is " + (p1.equals(p2)));
    System.out.println("p2.equals(p3) is " + (p2.equals(p3)));
    System.out.println("p1.equals(p3) is " + (p1.equals(p3)));
}
```

- Output:

```
p1.equals(p2) is true
p2.equals(p3) is true
p1.equals(p3) is false
```

- It violated the transitivity rule!

Pitfalls in Implementing `equals`

- This is a fundamental problem of equivalence relations in object-oriented languages
- There is no way to extend an instantiable class and add a value component while preserving the *equals* contract, unless you are willing to forgo the benefits of object-oriented abstraction
- In JDK, `java.sql.Timestamp` extends `java.util.Date` and its `equals` method is not symmetric!
  – Its javadoc has a disclaimer on this

Steps for Writing a equals Method

1. Use the == operator to check if the argument is a reference to *this* object
2. Use `instanceof` to check for the right argument type
3. Cast the argument to the correct type
4. For each "significant" field in the class, check if that field of the argument matches the corresponding field of *this* object
   - For float fields, use the `Float.compare` method
   - For double fields, use `Double.compare`
   - For other primitive types, use ==
   - For object reference fields, invoke the `equals` method

The `Object.hashCode` Method

- Returns a hash code value (of `int` type) for the object, used by all hash-based collections, such as `HashMap`, `HashSet`, and `Hashtable`
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same hash code
  – Hence you must override `hashCode` if you override `equals`!

What If You Don't

- All hash-based collections may behave erroneously
- For example:

```
Map<Point2D, String> interestingPoints = new HashMap<Point2D, String>();
interestingPoints.put(new Point2D(0.0, 0.0), "origin");
String p = interestingPoints.get(new Point2D(0.0, 0.0));
```

  – `p` will be `null` instead of "`origin`", even though the two instances of the `Point2D` objects are equal
- The default `Object.hashCode` usually is implemented by converting the internal address of the object into an integer

What Is a Good Hash Function

- Tends to produce unequal hash codes for unequal objects
- Should distribute any reasonable collection of unequal instances uniformly across all possible hash values

A Good (enough) hashCode Recipe …

1. Initialize hash code `result` with some constant nonzero value, say, `17`
2. For each field `f` in your object that plays a role in the `equals` method, do the following:
   a. Compute an int hash code `c` for the field:
      I. If the field is a boolean, compute `(f ? 1 : 0)`
      II. If the field is a `byte`, `char`, `short`, or `int`, compute `(int) f`
      III. If the field is a `long`, compute `(int) (f ^ (f >>> 32))`
      IV. If the field is a `float`, compute `Float.floatToIntBits(f)`

A Good (enough) hashCode Recipe …

V. If the field is a `double`, compute `Double.doubleToLongBits(f)`, and then hash the resulting `long` as in step 2.a.iii

VI. If the field is an object reference and this class's `equals` method compares the field by recursively invoking `equals`, recursively invoke `hashCode` on the field. If the value of the field is `null`, return `0` (or some other constant, but `0` is traditional)

VII. If the field is an array, treat it as if each element were a separate field. That is, compute a hash code for each significant element by applying these rules recursively, and combine these values per step 2.b. If every element in an array field is significant, you can use one of the `Arrays.hashCode` methods added in release 1.5

A Good (enough) hashCode Recipe …

b. Combine the hash code `c` computed in step 2.a into `result` as follows: `result = 31 * result + c`

3. Return `result`.
4. Write unit tests to verify whether equal instances have equal hash codes

Example – hashCode for Point2D

```
@Override public int hashCode() {
    int result = 17;
    long doubleBits = (Double.doubleToLongBits(x));
    result = 31 * result + (int) (doubleBits ^ (doubleBits >>> 32));
    doubleBits = (Double.doubleToLongBits(y));
    result = 31 * result + (int) (doubleBits ^ (doubleBits >>> 32));
    return result;
}
```

`The Object.toString` Method

- The default `Object.toString()` method returns a string that consists of:
  - Fully qualified class name
  - “`@`”
  - Unsigned hexadecimal representation of the `hashCode()`
- Example: `com.scispike.effectivejava.ch02.Point2D@3fd1`
- It is recommended that all subclasses override `toString`

Why Overriding `toString`

- `toString` is automatically invoked when an object is passed to:
  - `print` methods
  - `String` concatenation
  - `assert`
  - print by debugger
- `toString` should return a concise but informative representation that is easy for people to read

Example `toString` Method

- For `Point2D` class:

```
@Override
public String toString() {
     return String.format("[%.02f, %.02f]", this.x, this.y);
}
```

`Object.clone()` Method

- `Object.clone()` method creates and returns a copy of this object
  - Field-by-field copy is a “shallow” copy, not “deep” copy
- A class must implement the `Cloneable` interface (a tag interface with no method) to indicate to `Object.clone()` that it is legal for it to be called
  - Otherwise, `CloneNotSupportedException` thrown
- By convention, classes that implement `Cloneable` interface should override `Object.clone` (which is protected) with a public method

The (Loose) Contract for `clone()` …

- For any object `x`, the expression `x.clone() != x` will be true
- `x.clone().getClass() == x.getClass()` will be `true` (not an absolute requirement)
- Typically, `x.clone().equals(x)` will be `true` (not an absolute requirement)

The (Loose) Contract for clone() …

- By convention, the returned object should be obtained by calling `super.clone`
  - If a class and all of its super classes (except `Object`) obey this convention, it will be the case that `x.clone().getClass() == x.getClass()`
- By convention, the object returned by this method should be independent of this object (which is being cloned)
  - This requires deep copy

Potential Problems with `clone()`

- Because you need to work off `super.clone()` object, it's not possible to write a properly functioning `clone()` method unless all the class' super classes provide a well-behaved `clone()` method
- `clone()` is incompatible with normal use of final fields referring to mutable objects
- Deep copy can be error-prone

Do You Really Need clone()?

- Some choose to never override the `clone()` method and never invoke it
  - With exception to copy arrays
- A better approach is to provide a copy constructor, one that takes an instance of the class as input, e.g.:

```
public Point2D(Point2D p) {…}
```

- Or a copy SFM

```
public static Point2D copy(Point2D p) {…}
```

Summary

- If you override `equals()`, you must override `hashCode()`
- Always override `toString()`
- Copy constructor or copy SFMs might be better choices than overriding `clone()`

Exercise

- Do the exercise:
  - **Create an Employee Class**

3: Mastering Classes and Interfaces

Objectives

- Accessibility of members and classes
- Accessors vs. fields
- Composition vs. Inheritance
- Interfaces
- Class hierarchies
- Implementing strategies with function objects

Review of Access Levels

- For members (fields, methods, nested classes, and nested interfaces), there are four possible access levels:
  - **private**: Accessible only from the class where it is declared
  - **package-private**: Accessible from any class of the package
    - This is the default access if none is specified
  - **protected**: Accessible from subclasses and any class of the same package
  - **public**: Accessible from anywhere

Minimize the Accessibility …

- To maximize information hiding, make each class or member as inaccessible as possible
- If a top-level class or interface can be made package-private, it should be
  - If you make it public, you are obligated to support it forever to maintain compatibility
- The public class is part of the package's API; the package-private top-level class is part of its implementation

Minimize the Accessibility

- Protected members should be relatively rare
  - They are actually part of the class' exported API and must be supported forever
- Instance fields should never be public
- Classes with public mutable fields are not thread-safe
- Public static final fields (commonly used for defining constants) should only contain either primitive values or references to immutable objects
  - In particular, arrays are always mutable; don't define public static final array fields

Expose Fields or Not

- Public classes should never expose mutable fields
  - Make them private and provide public accessors (getters) and mutators (setters)
- It is less harmful to expose immutable fields
- Sometimes it is desirable for package-private or private nested classes to expose fields, whether mutable or immutable

Immutable Classes

- An immutable class is simply a class whose instances cannot be modified
  - All of the information contained in each instance is provided when it is created and is fixed for the lifetime of the object
  - For example, `String`, `BigInteger`, etc.
- They are easier to design, implement, and use than mutable classes
  - Less error prone and more secure
- Inherently thread-safe

To Make A Class Immutable

1. Don’t provide any mutator methods
2. Ensure that the class can’t be extended
   - Generally accomplished by making the class `final`
3. Make all fields `private final`
4. Ensure exclusive access to any mutable components
   - If the class has any fields that refer to mutable objects, ensure that clients of the class cannot obtain references to these objects
   - Never initialize such a field to a client-provided object or return the object reference from an accessor

An Immutable Bill Class

Code: ch03/ImmutableBill

```
public final class ImmutableBill {
    private final float amount;
    private final Date billingDate;
    public ImmutableBill(float amount, Date billingDate) {
        this.amount = amount;
        this.billingDate = billingDate;
    }
    public float getAmount() {return amount;}
    public Date getBillingDate() {return this.billingDate;}
        // Notice that instead of changing the internal state,
        // we create and return a new instance
    public ImmutableBill addAmount(float amount) {
        return new ImmutableBill(this.amount + amount, this.billingDate);
    }
}
```

Inheritance Can Be Dangerous

- Inheritance is a powerful way to achieve code reuse
- It is safe to use inheritance within a package
  - The subclass and the super class implementations are under the control of the same programmers
- But inheriting from ordinary concrete classes across package boundaries is dangerous

Inheritance Violates Encapsulation

- A subclass depends on the implementation details of its super class for its proper function
- The super-class's implementation may change
  - The subclass may break, even though its code has not been touched
  - A subclass must evolve in tandem with its super-class
- Super-classes should be designed and documented specifically for the purpose of being extended

Example: Extending HashMap

Code: ch03/InstrumentedHashMap

- Suppose we want to extend `HashMap` so that we can keep track of how many `<key, value>` pairs have been put in the map:

```
public class InstrumentedHashMap<K, V> extends HashMap<K, V> {
    private int putCount = 0;
    @Override
    public V put(K key, V value) {
        putCount++;
        return super.put(key,  value);
    }
    @Override
    public void putAll(Map<? extends K,? extends V> map) {
        putCount += map.size();
        super.putAll(map);
    }
```

Example: Extending HashMap

```
    public int getPutCount() {
        return putCount;
    }
    public static void main(String[] args) {
        InstrumentedHashMap<String, String> ihm =
                  new InstrumentedHashMap<String, String>();
        ihm.put("First President", "George Washington");
        HashMap<String, String> rest = new HashMap<String, String>();
        rest.put("Second President", "John Adams");
        rest.put("Third President", "Thomas Jefferson");
        ihm.putAll(rest);
        System.out.printf("%d presidents have been put in the map\n",
                   ihm.getPutCount());
    }
}
Output: 5 presidents have been put in the map
```

Use Composition Instead

- The problem in the above example is `HashMap`'s `putAll` method internally calls `put`
  - Forcing us to know the inner workings of super-class
- Solution: use composition instead of inheritance
- In object-oriented data modeling world:
  - Inheritance represents *is-a* relationship
  - Composition represents a *has-a* relationship
- When inheritance is used for code reuse purpose, prefer composition over inheritance

How to Use Composition and Forwarding

- In the new class, use a private field to reference an instance of the existing class - *composition*
- Each instance method in the new class invokes the corresponding method on the contained instance of the existing class and returns the results
  - This is known as *forwarding*, and the methods in the new class are known as forwarding methods

Example: Use Composition

Code: ch03/ForwardingHashMap

```
public class ForwardingHashMap<K, V> implements Map<K, V> {
    private final Map<K, V> m;
    public ForwardingHashMap(Map<K, V> m) {this.m = m;}
    public int size() {return m.size();}
    public boolean isEmpty() {return m.isEmpty();}
    public boolean containsKey(Object key) {return m.containsKey(key);}
    public boolean containsValue(Object value)
         {return m.containsValue(value);}
    public V get(Object key) {return m.get(key);}
    public V put(K key, V value) {return m.put(key, value);}
    public V remove(Object key) {return m.remove(key);}
    public void putAll(Map<? extends K, ? extends V> m) {this.m.putAll(m);}
    public void clear() {m.clear();}
    public Set<K> keySet() {return m.keySet();}
    public Collection<V> values() {return m.values();}
    public Set<java.util.Map.Entry<K, V>> entrySet() {return m.entrySet();}
}
```

Example: Extending Forwarding Class …

Code: ch03/InstrumentedHashMapWrapper

```
public class InstrumentedHashMapWrapper<K, V>
        extends ForwardingHashMap<K, V> {
    private int putCount = 0;
    public InstrumentedHashMapWrapper(Map<K, V> m) {
        super(m);
    }
    @Override
    public V put(K key, V value) {
        putCount++;
        return super.put(key,  value);
    }
    @Override
    public void putAll(Map<? extends K,? extends V> map) {
        putCount += map.size();
        super.putAll(map);
    }
```

Example: Extending Forwarding Class …

```
    public int getPutCount() {
        return putCount;
    }
    public static void main(String[] args) {
        InstrumentedHashMapWrapper<String, String> ihm =
                new InstrumentedHashMapWrapper<String, String>
                        (new HashMap<String, String>());
        ihm.put("First President", "George Washington");
        HashMap<String, String> rest = new HashMap<String, String>();
        rest.put("Second President", "John Adams");
        rest.put("Third President", "Thomas Jefferson");
        ihm.putAll(rest);
        System.out.printf("%d presidents have been put in the map\n",
                        ihm.getPutCount());
    }
}
Output: 3 presidents have been put in the map
```

Wrapper, Decorator, Delegate

- The `InstrumentedHashMapWrapper` is known as a *wrapper* class
  - Each instance contains ("wraps") another instance of `HashMap`
- This is also known as the *decorator* pattern
  - The `InstrumentedHashMapWrapper` class "decorates" a `HashMap` by adding instrumentation
- Sometimes the combination of composition and forwarding is loosely referred to as *delegation*

Prefer Interfaces over Abstract Classes

- Existing classes can be easily retrofitted to implement a new interface
  - Add the required methods if they don't yet exist
  - Add an implements clause to the class declaration
- Interfaces are ideal for defining *mixins*
  - A type that a class can implement in addition to its "primary type" as some optional behavior
  - E.g. `Comparable`

Prefer Interfaces over Abstract Classes …

- Interfaces allow the construction of nonhierarchical type frameworks
- For example:

```
public interface Singer {
      AudioClip sing(Song s);
}
public interface Songwriter {
      Song compose(boolean hit);
}
public interface SingerSongwriter extends Singer, Songwriter {
      AudioClip strum();
      void actSensitive();
}
```

Prefer Interfaces over Abstract Classes …

- Interfaces enable safe, powerful functionality enhancements via the wrapper class idiom
- You can combine the virtues of interfaces and abstract classes by providing an abstract skeletal implementation class to go with each nontrivial interface that you export

Disadvantages of Interfaces

- It is far easier to evolve an abstract class than an interface
  - Almost impossible to add a method to a public interface without breaking all existing implementing classes
- Public interfaces must be designed carefully
  - Once an interface is released and widely implemented, it is almost impossible to change

Class Hierarchies or Tagged Classes

- When a class whose instances come in two or more flavors, sometimes you see code like this:

```
class Figure {
    enum Shape { RECTANGLE, CIRCLE };
    final Shape shape; // Tag field
    double length; // Used only if shape is RECTANGLE
    double width; // Used only if shape is RECTANGLE
    double radius; // This field is used only if shape is CIRCLE
    ...
    double area() {
        switch(shape) {
        case RECTANGLE:
            return length * width;
        case CIRCLE:
            return Math.PI * (radius * radius);
        ...}}
```

Prefer Class Hierarchies

- It's far better if we create class hierarchies instead:

```
abstract class Figure {
   abstract double area();
}

class Circle extends Figure {
   ...
   double area() { ... }
}

class Rectangle extends Figure {
   ...
   double area() { ... }
}
```

What are Function Objects?

- It is possible to define an object whose methods perform operations on other objects that are passed in
- An instance of a class that exports exactly one such method is effectively a pointer to that method
  - Such instances are known as *function objects*
- For example:

```
class StringLengthComparator {
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
}
```

Function Objects Represent Strategy

- A `StringLengthComparator` instance is a concrete strategy for string comparison
- Usually there is a *strategy interface* that concrete strategies implement, e.g.:

```
public interface Comparator<T> {
    public int compare(T t1, T t2);
}
class StringLengthComparator implements Comparator {
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
}
```

Using Strategy

- The `StringLengthComparator` strategy, for example, can be used in:

```
Arrays.sort(stringArray,
    new StringLengthComparator())
```

Summary

- Try minimize the accessibility of classes and members
- Use accessors instead of public fields
- Immutable objects are your friends
- Inheritance can be dangerous; prefer composition
- Interfaces have many advantages over abstract classes
- Prefer class hierarchies to tagged classes
- Use function objects to represent strategies

Exercise

- Do the exercises:
  - **Create an Immutable Class**
  - **Create an Encryption Strategy**
  - **Create a Decorator**

4: Generics

Objectives

- Avoiding Raw types
- Favoring generic collections
- Creating generic types and methods
- Increasing API flexibility

Review Generics-related Terms

Term	Example
Parameterized type	`List<String>`
Actual type parameter	`String`
Generic type	`List<E>`
Raw type	`List`
Formal type parameter	`E`
Unbounded wildcard type	`List<?>`
Bounded wildcard type	`List<? extends Number>`
Bounded type parameter	`<E extends Number>`
Recursive type bound	`<T extends Comparable<T>>`
Generic method	`public <T> List<T> toList(T[] a)`

Don’t Use Raw Types in New Code

- Generics were added in JDK 1.5
- Any new code targeting JDK 1.5+ should not use raw types
- Generics provide compile time type safety
  - Avoiding raw types’ runtime `ClassCastException`
- With generics, no need to explicitly cast types any more
- Raw types remain in Java for source code backward compatibility

Use Unbounded Wildcard Types

- When using a collection whose element type you don't care, don't use raw types, use unbounded wildcard types:

```
int numCommonElements(Set s1, Set s2) { // Don't do this
    int result = 0;
    for (Object o1 : s1)
        if (s2.contains(o1))
            result++;
    return result;
}
```

  – Instead, do this:

```
int numCommonElements(Set<?> s1, Set<?> s2) { // Do this
    ...// same as above
}
```

Raw Type vs. Unbounded Wildcard Type

- You can put in anything into a raw typed collection and corrupt it at runtime (`ClassCastException`)
- You can't put any element (other than `null`) into a unbounded wildcard type collection
- You must use raw types in:
  - Class literals, i.e. `List.class`, not `List<String>.class`
  - `instanceof` operation

```
if (o instanceof Set) {
  Set<?> s = (Set<?>)o; // Don't cast it to raw type
  ...
}
```

Generic List or Arrays

- Arrays are *covariant*: if `S` is a subtype of `T`, then `S[]` is a subtype of `T[]`
- Generics are *invariant*: for any two distinct types `S` and `T`, `List<S>` is neither a subtype nor a supertype of `List<T>`:

```
List<Object> ol = new ArraysList<String>(); // Incompatible types
```

- Generics and arrays do not mix well
  - You can't have `new ArrayList<String>[]`
  - You can have
    - `ArrayList<String>[] array = new ArrayList[10];`

Creating Your Own Generic Type

Code: ch04/Queue

```
import java.util.LinkedList;
public class Queue<T> {
    private LinkedList<T> items = new LinkedList<T>();
    public void enqueue(T item) {
        items.addLast(item);
    }
    public T dequeue() {
        return items.removeFirst();
    }
    public boolean isEmpty() {
        return (items.size() == 0);
    }
    @Override
    public String toString() {
        return items.toString();
    }
}
```

Creating Your Own Generic Type

Code: ch04/Pair

```
public class Pair<P1,P2> {
    private P1 p1;
    private P2 p2;
    public Pair(P1 p1, P2 p2) {
        this.p1 = p1;
        this.p2 = p2;
    }
    public P1 getFirst() {
        return this.p1;
    }
    public P2 getSecond() {
        return this.p2;
    }
    @Override
    public String toString() {
        return String.format("[%s, %s]", this.p1, this.p2);
    }
}
```

Testing Your Generic Types

Code: ch04/Test

```
public static void main(String[] args) {
    Queue<String> q = new Queue<String>();
    q.enqueue("George Washington");
    q.enqueue("John Adams");
    q.enqueue("Thomas Jefferson");
    System.out.println(q);
    q.dequeue();
    System.out.println(q);
    Queue<Pair<String, String>> q2 = new Queue<Pair<String, String>>();
    q2.enqueue(new Pair<String, String>("George Washington",
                                         "April 30, 1789"));
    q2.enqueue(new Pair<String, String>("John Adams", "March 4, 1797"));
    System.out.println(q2);
}
[George Washington, John Adams, Thomas Jefferson]
[John Adams, Thomas Jefferson]
[[George Washington, April 30, 1789], [John Adams, March 4, 1797]]
```

Generic Methods

- A generic method is one with type parameters
- Suppose we need to write a method for finding the intersection of two sets
  - If using raw type:

    ```
    public static Set intersection<Set s1, Set s2) {...}
    ```

  - Since we try to avoid using raw type, let's try using generic type. How about:

    ```
    public static Set<T> intersection<Set<T> s1, Set<T> s2) {...}
    ```

  - But compiler will think that `T` is an actual type and will complain that `T` is undefined!
- We need a way to tell the compiler that `T` is a type parameter

Example Generic Method

- The type parameter list that goes between the method's modifiers and its return type, tells the compiler that they are type parameters

```
public static <T> Set<T> intersection(Set<T> s1, Set<T> s2) {
   Set<T> rslt = new TreeSet<T>();
   for (T x : s1)
     if (s2.contains(x))
         rslt.add(x);
   return rslt;
}
```

Bounded Wildcard Increases Flexibility

- Suppose we need a new `addAll` method for `Queue`:

```
public void addAll(Collection<T> collection) {
    for (T item : collection)
        enqueue(item);
}
```

- Suppose we defined a `Queue<Number>` instance:

```
Queue<Number> q = new Queue<Number>();
```

- `q.enqueue(1)` works fine because `1` (boxed to `Integer`) is a subtype of `Number`
- The following is not ok b/c `Collection<Integer>` is not a subtype of `Collection<Number>`:

```
Collection<Integer> integers = new ArrayList<Integer>();
integers.add(1);
q.addAll(integers);
```

Bounded Wildcard Increases Flexibility

- Change the `addAll` method to use bounded wildcard type will make the above code work:

```
public void addAll(Collection<? extends T> collection) {
  for (T item : collection)
     enqueue(item);
}
```

- Now you may pass in any collection of type that is a *subtype* of T
  - Any type is also a subtype of itself

Bounded Wildcard Increases Flexibility

- Suppose we need a new `removeAllTo` method for `Queue`:

```
public void removeAllTo(Collection<? extends T> collection) {
  while (!isEmpty()) {
     T item = dequeue(); // Remove an item from the queue.
     collection.add(item); // Add it to the collection. ILLEGAL!!
  }
}
```

- Using `Collection<T> collection` as input would be too restrictive; but using `Collection<? extends T> collection` will cause error
- Solution:

```
public void removeAllTo(Collection<? super T> collection) {...}
```

The Rule of PECS

- PECS: **P**roducer-**e**xtends, **c**onsumer-**s**uper
- For an *in* argument, its type should be any *subtype* of the target type, i.e. `<? extends T>`
  - Because an input will eventually appear on the right side of an assignment: **`T element = inputElement;`**
- For an *out* argument, its type should be any *super* type of the target type. i.e. `<? super T>`
  - Because an output element will eventually appear on the left side of an assignment: **`outElement = element;`**
- Same reason you can assign a `String` to an `Object`, but not an `Object` to a `String` (without casting)

Misc. Best Practices

- Do not use wildcard types as return types, e.g. `Set<E>`, not `Set<?>`
  - It would force client code to use wildcard types
- Properly used, wildcard types should be nearly invisible to users of a class
  - If the user of a class has to think about wildcard types, there is probably something wrong with the API
- If a type parameter appears only once in a method declaration, replace it with a wildcard

A Generic `max` Function

- It takes a `List` of type `T` as input, where `T` is a subtype of `Comparable` that compares instances of `T`
- Initial definition:

```
public static <T extends Comparable<T>> T max(List<T> list) {...}
```

- Revised to use wildcard types:

```
public static <T extends Comparable<? super T>> T max(List<? extends T>
  list) {...}
```

- Always use `Comparable<? super T>` in preference to `Comparable<T>`
  – A `Comparable` of `T` always *consumes* `T` instances (and produces integers)

Implementing max

```
public static <T extends Comparable<? super T>> T
  max(List<? extends T> list) {
   T result = null;
   for (T e : list) {
     if (result == null || (e != null &&
           e.compareTo(result) > 0)
       result = e;
   }
   return result;
}
```

Summary

- New code should not use raw types any more
- Create your own generic types and generic methods
- Use bounded wildcard types to increase your API's flexibility

Exercise

- Do the exercise:
  - **Create a Generic Stack**

5: Enumerations

Objectives

- Defining enumerations
- Using annotations

What Is `enum` Type

- An enumerated type is a type whose legal values consist of a fixed set of constants, e.g.:

```
public enum Planet {
     MERCURY, VENUS, EARTH, MARS, JUPITER, SATURN, URANUS, NEPTUNE;
}
```

- The `enum` declaration defines a *class* (called an *enum type*) that implicitly extends `java.lang.Enum`
  - `enum` types are full-fledged classes
  - `enum` constants are really just public static final fields
  - `enum` types do not support extension
- You can not create instances of an `enum` type

`enum`'s Builtin Methods …

- The compiler automatically adds some special methods when it compiles an enum type
  - The static `values()` method returns an array containing all of the values of the enum in the order they are declared

```
for (Planet p : Planet.values())
  System.out.printf("%s ", p);
```

Output

```
MERCURY VENUS EARTH MARS JUPITER SATURN URANUS NEPTUNE
```

- The `printf` statement above calls the enum's `toString`
  - By default, `toString` returns the enum's constant as it appeared in the declaration

`enum`'s Builtin Methods

- Enum's base class `java.lang.Enum` implements `Comparable` and `Seriablizable` interfaces
  - Give you the `compareTo` method
- `name()` Returns the name of this enum constant, exactly as declared in its enum declaration
- `int ordinal()` Returns the position in its enum declaration of this enumeration constant, first one is `0`

Associate Data with enum Constants

- You may associate data with enum constants
  1. Define enum constants first
  2. Define any fields or methods
  3. Define package-private or private constructor
- Enums are immutable, hence all fields must be final
- Example, we can associate mass and radius with each planet in our Planet enum and provide a method for computing weight of an object on the planet

Associate Data with Planets

Code: ch05/Planet

```
public enum Planet {
    MERCURY(3.303e+23, 2.4397e6),
    VENUS(4.869e+24, 6.0518e6),
    EARTH(5.976e+24, 6.37814e6),
    MARS(6.421e+23, 3.3972e6),
    JUPITER(1.9e+27, 7.1492e7),
    SATURN(5.688e+26, 6.0268e7),
    URANUS(8.686e+25, 2.5559e7),
    NEPTUNE(1.024e+26, 2.4746e7);

    private final double mass; // in kilograms
    private final double radius; // in meters
    private final double surfaceGravity;
        // universal gravitational constant
    private static final double G = 6.67300E-11;
```

Associate Data with Planets …

```
    private Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
        this.surfaceGravity = G * mass / (radius * radius);
    }
    public double mass() {
        return mass;
    }
    public double radius() {
        return radius;
    }
    public double surfaceGravity() {
        return surfaceGravity;
    }
    public double surfaceWeight(double mass) {
        return mass * surfaceGravity;
    }
}
```

Test enum Planet

Code: ch05/Test

```
public class Test {
    public static void main(String[] args) {
        double myWeightOnEarth = 170.00d;
        double mass = myWeightOnEarth/Planet.EARTH.surfaceGravity();
        for (Planet p : Planet.values())
            System.out.printf("I weigh %.2f on %s%n",
                    p.surfaceWeight(mass), p);
    }
}
I weigh 64.22 on MERCURY
I weigh 153.85 on VENUS
I weigh 170.00 on EARTH
I weigh 64.39 on MARS
I weigh 430.19 on JUPITER
I weigh 181.22 on SATURN
I weigh 153.87 on URANUS
I weigh 193.52 on NEPTUNE
```

Associate with Different Behavior

- If you need to associate different behavior with each enum constant:
  - Define an abstract method in the enum class
  - Each enum constant provides its own implementation of the abstract method
- Example: An `Operation` enum type below shows:
  - How to define different behavior for each enum constant
  - How to override `toString` method

An Operation enum …

Code:
ch04/Operation

```
public enum Operation {
    PLUS("+") {
        double apply(double x, double y) { return x + y; }
    },
    MINUS("-") {
        double apply(double x, double y) { return x - y; }
    },
    TIMES("*") {
        double apply(double x, double y) { return x * y; }
    },
    DIVIDE("/") {
        double apply(double x, double y) { return x / y; }
    };
```

An Operation enum

```
    private final String symbol;
    Operation(String symbol) {
        this.symbol = symbol;
    }
    @Override
    public String toString() {
        return symbol;
    }
    abstract double apply(double x, double y);
}
```

Using the Operation enum

Code:
ch05/TestOperation

```
public static void main(String[] args) {
        double x = 23.45d;
        double y = 10.45;
        for (Operation op : Operation.values())
            System.out.printf("%.2f %s %.2f = %.2f%n",
                                    x, op, y, op.apply(x, y));
}
```

```
23.45 + 10.45 = 33.90
23.45 - 10.45 = 13.00
23.45 * 10.45 = 245.05
23.45 / 10.45 = 2.24
```

Use EnumSet

- java.util.EnumSet takes a set of values from a single enum type as input
- Internally represented as a bit vector
  - If the underlying enum type has sixty-four or fewer elements, the entire EnumSet is represented with a single long

```
public enum Style {
     BOLD, ITALIC, UNDERLINE, STRIKETHROUGH
}
...
applyStyles(EnumSet.of(Style.BOLD, Style.italics))
```

Emulate enum Extensions with Interfaces

- Because every enum type implicitly extends `java.lang.Enum` and Java doesn't allow multiple inheritance, you can't have one enum type extends another
- But an enum type can implement any arbitrary interfaces
- Let both enum types implement a same interface
  - Clients should use the interface type instead of the enum types
  - Wherever one enum type is used, the other can be used too

“Extending” the Operation enum

Code:
ch04/Operation

```
public interface OperationInterface {
    double apply(double x, double y);
}
public enum BasicOperations implements OperationInterface {
    PLUS("+") {
        public double apply(double x, double y) { return x + y; }
    },
    MINUS("-") {
        public double apply(double x, double y) { return x - y; }
    },
    TIMES("*") {
        public double apply(double x, double y) { return x * y; }
    },
    DIVIDE("/") {
        public double apply(double x, double y) { return x / y; }
    };
```


“Extending” the Operation enum …

```
    private final String symbol;
    BasicOperations(String symbol) {
        this.symbol = symbol;
    }
    @Override
    public String toString() {
        return symbol;
    }
}
```

"Extending" the Operation enum

Code: ch05/ExtendedOperation

```
public enum ExtendedOperation implements OperationInterface {
    EXP("^") {
        public double apply(double x, double y) {
            return Math.pow(x, y);
        }},
    REMAINDER("%") {
        public double apply(double x, double y) {
            return x % y;
        }};
    private final String symbol;
    ExtendedOperation(String symbol) {this.symbol = symbol;}
    @Override public String toString() {return symbol;}
}
```

Testing the Operation enum

Code:
ch04/TestOperationInterface

```
public static void main(String[] args) {
    double x = 2.00;
    double y = 10.00;
    test(ExtendedOperation.class, x, y);
}
private static <T extends Enum<T> & OperationInterface> void test
   (Class<T> opSet, double x, double y) {
    for (OperationInterface op : opSet.getEnumConstants())
        System.out.printf("%.2f %s %.2f = %.2f%n",
                          x, op, y, op.apply(x, y));
}
```

```
2.00 ^ 10.00 = 1024.00
2.00 % 10.00 = 2.00
```

Use `@Override` Annotation Consistently

- What does this program print and why?

```
public class NameValuePair {
    private String name;
    private String value;
    public NameValuePair(String name, String value) {
        this.name = name;
        this.value = value;
    }
    public boolean equals(NameValuePair pair) {
        return pair.name.equals(this.name) &&
                  pair.value.equals(this.value);
    }
    public int hashCode() {...}
    public String toString() {...}
```

Use `@Override` Annotation Consistently

```
    public static void main(String[] args) {
        Set<NameValuePair> s = new HashSet<NameValuePair>();
        for (int i = 0; i < 10; i++)
            s.add(new NameValuePair("lastName", "Adams"));
        System.out.println(s.size());
    }
}
```

- It prints out 10
- But we defined a `equals` method and `Set` is supposed to have no duplicates!

Use `@Override` Annotation Consistently

- Reason: we thought we were overriding `Object`'s `equals` method but we were not!
- If we always used the `@Override` annotation to state our intention, the compiler would have caught that we defined the `equals` method wrong
- The right `equals` method:

```
@Override public boolean equals(Object o) {
    if (!(o instanceof NameValuePair))
        return false;
    NameValuePair pair = (NameValuePair)o;
    return pair.name.equals(this.name) && pair.value.equals(this.value);
}
```

- Now it prints out 1

Summary

- Enum types are full-fledged classes
- Enum constants are really just static constants
- You may define fields and methods on Enum types
- Use @Override annotation on every method that you believe to override a super class method

Exercise

- Do the exercise:
  - **Create an enum Class for Eurozone Countries**

6: Mastering Methods

Objectives

- Designing method signatures
- Validate parameters
- Method overloading
- Using variable lists of arguments

Design Method Signatures Carefully …

- Choose method names carefully
  - Should be understandable and consistent with other names in the same package
  - Should be consistent with the broader consensus. Use Java API as guidance
- For parameter types, if there is an appropriate interface to define a parameter, use it in favor of a class that implements the interface
- Prefer two-element enum types to `boolean` parameters
  - Makes your code easier to read and to write
  - Makes it easy to add more options later

Design Method Signatures Carefully …

- Don't go overboard in providing convenience methods
  - Too many methods make a class difficult to learn, use, document, test, and maintain
  - Especially true for interfaces
    - Too many methods complicate life for implementers as well as users
  - Consider providing a "shorthand" only if it will be used often
  - When in doubt, leave it out

Design Method Signatures Carefully …

- Avoid long parameter lists
  - Aim for four parameters or fewer
- Three techniques for shortening parameter lists
  - Break the method up into multiple methods, each of which requires only a subset of the parameters
  - Create helper classes to hold groups of parameters
    - Typically these helper classes are static member classes
  - Adapt the Builder pattern from object construction to method invocation

Check Arguments' Validity

- Most methods and constructors have some restrictions on what values may be passed into their parameters
- You should document such restrictions and enforce them at the beginning of the method
- For public methods, use the Javadoc `@throws` or `@exception` tag to document the exception that will be thrown if a restriction on parameter values is violated
- You don't want to do such checks if they are too expensive or impractical

Check Argument Validity Example

- For example, `Integer`'s `parseInt` method:

```
 ...
  * @throws  NumberFormatException if the <code>String</code>
  *               does not contain a parsable <code>int</code>.
  */
 public static int parseInt(String s, int radix)
                      throws NumberFormatException {
     if (s == null) {
         throw new NumberFormatException("null");
     }
...
```

Use Assertions to Check Arguments

- For nonpublic methods you should generally check the arguments using assertions
  - Assertions throw `AssertionError` if they fail
  - Have no effect unless enabled by passing the `-ea` (or `-enableassertions`) flag to java
  - For example:

```
    // Private helper function for a recursive sort
private static void sort(long a[], int offset, int length) {
    assert a != null;
    assert offset >= 0 && offset <= a.length;
    assert length >= 0 && length <= a.length - offset;
    ...
}
```

Make Your Methods Defensive

- Program defensively, with the assumption that the clients may be ill-behaved - intentionally or not
- If a class has mutable components that it gets from or returns to its clients, the class must defensively copy these components
  - Unless the cost of the copy would be prohibitive and the class trusts its clients not to modify the components

A Simple Timer

Code: ch06/Timer

```
public class Timer {
    private final Date start;
    private final Date end;
    public Timer(Date start, Date end) {
        if (start.compareTo(end) > 0)
            throw new IllegalArgumentException(start + " after " + end);
        this.start = start;
        this.end = end;
    }
    public Date start() {
        return start;
    }
    public Date end() {
        return end;
    }
    ... // Rest omitted
}
```

What's the Problem …

- The class should maintain a invariant of start <= end
- But a client can attack like the following:

```
Date start = new Date();
Date end = new Date();
Timer t = new Timer(start, end);
end.setYear(80); // Modified the internal object!
```

- Or this:

```
t.getEndTime().setYear(80);
```

Solution: Make a Defensive Copy

```java
public class Timer {
    private final Date start;
    private final Date end;
    public Timer(Date start, Date end) {
        if (start.compareTo(end) > 0)
            throw new IllegalArgumentException(start + " after " + end);
        this.start = new Date(start.getTime());
        this.end = new Date(end.getTime());
    }
    public Date getStartTime() {
        return new Date(start.getTime());
    }
    public Date getEndTime() {
        return new Date(end.getTime());
    }
    ... // Rest omitted
}
```

Defensive Copying Can Be Costly

- Defensive copying can have a performance penalty and isn’t always justified
- If the class and its client are both part of the same package, then it may be appropriate to do without
- Alternatively, where possible, use immutable objects as components of your objects

Be Careful with Overloading

Code: ch04/OverloadingCollection

- What does this program print out?

```
public class OverloadingCollection {
    public static String classify(Set<?> s) {
        return "Set";
    }
    public static String classify(Collection<?> c) {
        return "Unknown Collection";
    }
    public static void main(String[] args) {
        Collection<?> c = new HashSet<String>();
        System.out.println(classify(c));
    }
}
```

Overloading Is Different from Overridden

- It prints out “Unknown Collection”
- The choice of which *overloaded* method to invoke is made at compile time
- But choice among *overridden* methods is at run time
- To achieve what the program meant to achieve, define a single method that uses `instanceof` to differentiate

```
public static String classify(Collection<?> c) {
    return c instanceof Set ? "Set" : "Unknown Collection";
}
```


Rules of Thumb Regarding Overloading

- A safe, conservative policy is never to export two overloadings with the same number of parameters
- If a method uses varargs, a conservative policy is not to overload it
- For constructors, you can't use different names but you can expose static factory methods instead of constructors

Watch Out for Generics and Autoboxing

- When designing overloaded methods, remember that two types are radically different if it is clearly impossible to cast an instance of either type to the other
- With generics and autoboxing, this can be tricky

Overloading, Generics, and Autoboxing

Code: ch06/Test

```
public static void main(String[] args) {
    Set<Integer> set = new TreeSet<Integer>();
    List<Integer> list = new ArrayList<Integer>();
    for (int i = -3; i < 3; i++) {
        set.add(i);
        list.add(i);
    }
    System.out.println("Original: " + list);
    for (int i = 0; i < 3; i++) {
        set.remove(i);
        list.remove(i);
    }
    System.out.printf("Remaining: Set = %s, List = %s", set, list);
}
```

```
Original: [-3, -2, -1, 0, 1, 2]
Remaining: Set = [-3, -2, -1], List = [-2, 0, 2]
```

What Happened

- The `Set<E>` `remove(Object o)` is used with autoboxing
- But the `List<E>` interface has two overloaded methods:
  - `remove(Object o)` removes the first element `o` from the list
  - `remove(int i)` removes the element at position `i`
- Before generics and autoboxing, `Object` and `int` are radically different, not any more

Use Varargs Judiciously

Code: ch06/VarargTest

- Varargs accept zero or more arguments of a type, e.g.

```
static int sum(int... args) {
    int sum = 0;
    for (int arg : args)
        sum += arg;
    return sum;
}
```

- To accept one or more arguments, you would define one normal parameter, and then one varargs parameter

```
static int min(int firstArg, int... remainingArgs) {
    int min = firstArg;
    for (int arg : remainingArgs)
        if (arg < min)
            min = arg;
    return min;
}
```

Varargs Has Performance Implication

- Varargs works by:
  - First creating an array of size the number of arguments
  - Putting the argument values into the array
  - Finally passing the array to the method
- If you want varargs' flexibility but concerned with the performance, and say the majority of the calls of a method takes 3 parameters or less, you may do this:

```
public void foo() { }
public void foo(int a1) { }
public void foo(int a1, int a2) { }
public void foo(int a1, int a2, int a3) { }
public void foo(int a1, int a2, int a3, int... rest) { }
```

Summary

- Carefully designing the methods will make your API easier to learn and use and less prone to errors
- You should document and explicitly check the restrictions on the parameters
- Sometimes it's necessary to make defensive copies of objects that come from or return to clients
- The choice among overloaded methods is at compile-time, not at runtime
- Varargs gives you flexibility but has performance implication

Exercise

- Do the exercise:
  - **Create an Order System**

7: Programming Techniques

Objectives

- For-each loops
- Minimizing scope of local variables
- Working with exact numbers
- Strings vs. more meaningful classes
- Interfaces vs. Reflection

Prefer `for-each` to `for` Loops

- The for-each loops (since JDK 1.5) are much cleaner, less error-prone than traditional for loops

```
enum Suit { CLUB, DIAMOND, HEART, SPADE }
enum Rank { ACE, DEUCE, THREE, FOUR, FIVE, SIX, SEVEN,
            EIGHT, NINE, TEN, JACK, QUEEN, KING }
Collection<Suit> suits = Arrays.asList(Suit.values());
Collection<Rank> ranks = Arrays.asList(Rank.values());
...
for (Iterator<Suit> i = suits.iterator(); i.hasNext(); )
    for (Iterator<Rank> j = ranks.iterator(); j.hasNext(); ) {
        ...
Vs.

for (Suit suit : suits) {
  for (Rank rank : ranks) {
     ...
```

For-each Loops Work on any Iterables

- For-each loops let you iterate over collections, arrays, or any type that implements `Iterable` interface

```
public interface Iterable<T> {
  // Returns an iterator over the elements in this iterable
  Iterator<T> iterator();
}
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

- If writing a class representing a group of objects, implement `Iterable` even if not implementing `Collection`
  – Allow users to iterate over the type with for-each loop

When Can't You Use for-each Loop

- Filtering — If you need to traverse a collection and remove selected elements, then you need to use an explicit iterator so that you can call its remove method
- Transforming — If you need to traverse a list or array and replace some or all of the values of its elements, then you need the list iterator or array index in order to set the value of an element
- Parallel iteration — If you need to traverse multiple collections in parallel, then you need explicit control over the iterator or index variable, so that all iterators or index variables can be advanced in lockstep

Minimize the Scope of Local Variables …

- Java lets you declare variables anywhere a statement is legal
- To minimizing the scope of a local variable, declare it where it is first used
- Declaring a local variable prematurely can cause its scope not only to extend too early, but also to end too late
- Nearly every local variable declaration should contain an initializer
  - If you don’t yet have enough information to initialize a variable sensibly, you should postpone the declaration until you do

Minimize the Scope of Local Variables

- Loops present a special opportunity to minimize the scope of variables
- The `for` loop, in both its traditional and for-each forms, allows you to declare *loop variables*
  - Their scope consists of the body of the loop as well as the initialization, test, and update preceding the body

```
for (int i = 0, n = expensiveComputation(); i < n; i++) {
  doSomething(i);
}
```

- Keep methods small and focused
  - If you combine two activities in the same method, local variables relevant to one activity may be in the scope of the code performing the other activity

Floats and Doubles are not Exact

- The float and double types are designed primarily for scientific and engineering calculations
  - Designed to furnish accurate approximations quickly
- They do not provide exact results and should not be used where exact results are required
- Particularly ill-suited for monetary calculations because it is impossible to represent 0.1 (or any other negative power of ten) as a float or double exactly

Problem Using Double for Currency

Code: ch07/TestFloat

- Suppose you have a dollar and there are pencils priced at 10¢, 20¢, 30¢, and so forth (at 10¢ increase). You buy one of each, starting with the one that costs 10¢, until you can't afford to buy the next one:

```
public static void main(String[] args) {
      double funds = 1.00;
      int itemsBought = 0;
      for (double price = .10; funds >= price; price += .10) {
              funds -= price;
              itemsBought++;
      }
      System.out.println(itemsBought + " items bought.");
      System.out.println("Change: $" + funds);
}
3 items bought.
Change: $0.3999999999999999
```

BigDecimal for Currency Calculations

Code: ch07/TestBigDecimal

- But we should have been able to buy 4 items (0.1 + 0.2 + 0.3 + 0.4 = 1.00)
- Use `BigDecimal`, `int`, or `long` for currency calculation

```
public static void main(String[] args) {
    final BigDecimal TEN_CENTS = new BigDecimal(".10");
    int itemsBought = 0;
    BigDecimal funds = new BigDecimal("1.00");
    for (BigDecimal price = TEN_CENTS; funds.compareTo(price) >= 0;
              price = price.add(TEN_CENTS)) {
        itemsBought++;
        funds = funds.subtract(price);
    }
    System.out.println(itemsBought + " items bought.");
    System.out.println("Money left over: $" + funds);
}
4 items bought.
Money left over: $0.00
```

When Should You Avoid Using Strings

- Strings are poor substitutes for other value types
  - Data originally comes into a program from a file, network, or keyboard, and it is often in string form
  - It should be translated into the appropriate type
- Strings are poor substitutes for enum types
- Strings are poor substitutes for aggregate types
  - If an entity has multiple components, it is usually a bad idea to represent it as a single string, e.g.:

```
String compoundKey = className + "#" + i.next();
```

    - What if "#" occurs in one of the fields?
    - To access individual fields, you have to parse the string

String Concatenation Is Slow

- Strings are immutable; when two strings are concatenated, the contents of both are copied
  - Concatenating `n` strings requires time quadratic in `n`
- For string concatenation, use `StringBuilder`'s `append` method instead

Reflection Is Powerful As Well As Costly

- `java.lang.reflect` offers programmatic access to information about loaded classes
  - Given a Class object, you can obtain `Constructor`, `Method`, and `Field` instances
  - With them you can programmatically construct instances, invoke methods, and access fields, etc.
- This power comes at a price:
  - Lose all the benefits of compile-time type checking
  - The code required to perform reflective access is clumsy and verbose
  - Performance suffers. Reflective method invocation is much slower than normal method invocation

Use Reflection Only to Instantiate

- If at all possible, use reflection only to instantiate objects
- Then access the objects using interface or super-class
  - Their type is known at compile time, hence type-safety

```
Class<?> cl = Class.forName(className);
Set<String> s = (Set<String>) cl.newInstance();
// Exercise the set
s.addAll(...);
```

Summary

- When possible, use for-each loops instead of the for-loops
- One should minimize the scope of local variables
- `float` and `double` are not precise types, use `BigDecimal`, `long`, etc. for monetary calculation
- Avoid using strings to represent other data types
- String concatenations are expensive
- When you have to use reflection, try only use it to instantiate an instance and then use interface or super-class to interact with the objects

Exercise

- Do the exercise:
  - **Create a Stock Portfolio**

8: Working with Exceptions

Objectives

- Handling exceptional conditions
- Using standard exceptions
- Unnecessary checked exceptions
- Exceptions and abstractions
- Capture failure information

Exceptions Are for Exceptional Conditions

- Exceptions should not be used for *ordinary* control flow
- A well-designed API must not force its clients to use exceptions for ordinary control flow
  - A class with a "state-dependent" method should generally have a separate "state-testing" method indicating whether it is appropriate to invoke the state-dependent method
  - For example, the `Iterator` interface has the state-dependent method `next` and the corresponding state-testing method `hasNext`

Three Kinds of Throwables

- *Checked exceptions* for conditions from which the caller can reasonably be expected to recover
- *Runtime exceptions* to indicate programming errors
  - Most indicate precondition violations, for example, `ArrayIndexOutOfBoundsException`
- *Errors* are reserved for use by the JVM
  - It's best not to implement any new Error subclasses
- All of the unchecked throwables should subclass `RuntimeException`, directly or indirectly

Avoid Unnecessary Checked Exceptions

- Checked exceptions force the programmer to deal with exceptional conditions
- But overuse of checked exceptions can place a nontrivial burden on the programmer
- The burden is justified if:
  - The exceptional condition cannot be prevented by proper use of the API
  - The programmer using the API can take some useful action once confronted with the exception

Technique to Reduce Checked Exception

- Sometimes it is possible to break the method that throws the checked exception into two methods:
  - The first returns a `boolean` that indicates whether the exception would be thrown
  - The second is the actual method

```
// Invocation with state-testing method and unchecked exception
if (obj.actionPermitted(args)) {
    obj.action(args);
} else {
   // Handle exceptional condition
  ...
}
```

Favor the Use of Standard Exceptions

- Reusing preexisting exceptions has several benefits
  - Makes your API easier to learn and use
  - Programs using your API are easier to read because they aren't cluttered with unfamiliar exceptions
  - Fewer exception classes mean a smaller memory footprint and less time spent loading classes

Commonly Used Standard Exceptions

Exception	Occasion for Use
`IllegalArgumentException`	Non-null parameter value is inappropriate
`IllegalStateException`	Object state is inappropriate for method invocation
`NullPointerException`	Parameter value is null where prohibited
`IndexOutOfBoundsException`	Index parameter value is out of range
`ConcurrentModificationException`	Concurrent modification of an object has been detected where it is prohibited
`UnsupportedOperationException`	Object does not support method

Exception Translation

- When a method simply propagates an exception thrown by a lower-level abstraction, it may:
  - Expose exceptions that have no apparent connection to the higher layer
  - Pollute the API of the higher layer with implementation details
  - Break existing client programs if lower level implementation changes
- Higher layers should catch lower-level exceptions and throw exceptions fit for the higher-level abstraction
  - This idiom is known as *exception translation*

Exception Translation Example

- The following code segment is from `List` implementation

```
/**
 * Returns the element at the specified position in this list.
 * @throws IndexOutOfBoundsException if the index is out of range
 * ({@code index < 0 || index >= size()}).
 */
public E get(int index) {
    ListIterator<E> i = listIterator(index);
    try {
        return i.next();
    } catch(NoSuchElementException e) {
        return throw new IndexOutOfBoundsException("Index: " + index);
    }
}
```

Exception Chaining

- Lower level exceptions are helpful in debugging the root cause of higher level problems
- The lower level exception (cause) is passed to the higher-level exception
- This idiom is called *exception chaining*, a special form of exception translation

```
try {
    ... // calling lower-level code
} catch (LowerLevelException cause) {
    throw new HigherLevelException(cause);
}
```

- Most standard exceptions have chaining-aware constructors

Don’t Over Do It

- Where possible, the best way to deal with exceptions from lower layers is to avoid them
  - By ensuring that lower-level methods succeed
    - E.g. Checking the validity of the higher-level method’s parameters before passing them on to lower layers
  - Have the higher layer silently work around these exceptions
    - E.g. The higher-level method simply catch and log the exception

Capture Failure Contributing Factors

- To capture the failure, the detail message of an exception should contain the values of all parameters and fields that contributed to the exception
  - For example, ideally the detail message of an `IndexOutOfBoundsException` should contain
    - The lower bound
    - The upper bound
    - The index value that failed to lie between the bounds
- One way to help enforce this is to require the information in their constructors

A Better `IndexOutOfBoundsException`

```java
/**
* Construct an IndexOutOfBoundsException.
*
* @param lowerBound the lowest legal index value.
* @param upperBound the highest legal index value plus one.
* @param index the actual index value.
*/
public IndexOutOfBoundsException(int lowerBound, int upperBound,
                                  int index) {
   // Generate a detail message that captures the failure
   super("Lower bound: " + lowerBound + ", Upper bound: " + upperBound +
         ", Index: " + index);
   // Save failure information for programmatic access
   this.lowerBound = lowerBound;
   this.upperBound = upperBound;
   this.index = index;
}
```

Summary

- Exceptions are for exceptional conditions, not for ordinary control flow
- Checked exceptions are for recoverable conditions
- Runtime exceptions are for program errors
- Reuse standard exceptions as much as possible
- Exception translation and chaining help shield higher-level code from lower-level details and provide more meaningful exceptions for the layer
- Exceptions should include all contributing factors in the detail message

Exercise

- Do the exercise:
  - **Exception Chaining**