Principles of Software Construction: Objects, Design, and Concurrency

Inheritance (continued), type-checking, and behavioral contracts

Charlie Garrod    Christian Kästner
Administrivia

- Homework 1 due next Tuesday
Key concepts from Tuesday
Key concepts from Tuesday

- Module systems
- The key encapsulation principle
- Inheritance
  - For code reuse
  - Abstract classes
  - Some design principles
    - Hierarchical modeling
Aside: UML class diagram notation

- **«interface»** brand
- Methods in bottom compartment
- Dashed line, open triangle arrowhead for implements
- Fields in middle compartment
- Optional visibility:
  + for public
  - for private
  # for protected
  ~ for package (not used much)
- Return type comes after method or field
- Name of class or interface in top compartment
- Italic means abstract
- Solid line, open triangle arrowhead for extends

```
«interface» Dog

getName() : String
getBreed() : String
bark() : String
setName(name : String)
toString() : String

AbstractDog

- name : String
- breed : String

+ getName() : String
+ getBreed() : String
+ bark() : String
+ setName(name : String)
# setBreed(breed : String)
+ toString() : String

GermanShephard

bark() : String
play()
```
Today:

• Inheritance and polymorphism (continued)
  - For maximal code re-use
  - Diagrams to show the relationships between classes
  - Inheritance and its alternatives
  - Java details related to inheritance

• Type checking and its limitations
  - The subtype relation
  - Behavioral contracts
    • The java.lang.Object
A better design: An account type hierarchy

CheckingAccount extends Account. All methods from Account are inherited (copied to CheckingAccount)

SavingsAccount is a subtype of Account. Account is a supertype of SavingsAccount.

If we know we have a CheckingAccount, additional methods are available.
A better design: An account type hierarchy

```java
public interface CheckingAccount extends Account {
    ...
}
```

Account methods are inherited (copied to CheckingAccount)

```java
public interface InterestCheckingAccount extends CheckingAccount, SavingsAccount {
    ...
}
```

SavingsAccount is a subtype of Account. Account is a supertype of SavingsAccount.

Multiple interface extension

If we know we have a CheckingAccount, additional methods are available.
The power of object-oriented interfaces

• **Polymorphism**
  - Different kinds of objects can be treated uniformly by client code
    - e.g., a list of all accounts
  - Each object behaves according to its type
    - If you add new kind of account, client code does not change
  - Consider this pseudocode:

    ```
    If today is the last day of the month:
    For each acct in allAccounts:
        acct.monthlyAdjustment();
    ```

• See the DogWalker example
One implementation: Just use interface inheritance
Better: Reuse abstract account code

```java
public abstract class AbstractAccount implements Account {
    protected float balance = 0.0;
    public float getBalance() {
        return balance;
    }
    abstract public void monthlyAdjustment();
    // other methods...
}

public class CheckingAccountImpl extends AbstractAccount implements CheckingAccount {
    public void monthlyAdjustment() {
        balance -= getFee();
    }
    public float getFee() { /* fee calculation */ }
}
```
Better: Reuse abstract account code

```java
public abstract class AbstractAccount implements Account {
    protected float balance = 0.0;
    public float getBalance() {
        return balance;
    }
    abstract public void monthlyAdjustment();
    // other methods...
}

public class CheckingAccountImpl extends AbstractAccount implements CheckingAccount {
    public void monthlyAdjustment() {
        balance -= getFee();
    }
    public float getFee() {
        /* fee calculation */
    }
}
```

- An abstract class is missing the implementation of one or more methods.
- Protected elements are visible in subclasses.
- An abstract method is left to be implemented in a subclass.
- No need to define `getBalance()` – the code is inherited from `AbstractAccount`.

In `AbstractAccount`, `getBalance()` is defined, but `getFee()` is left abstract, allowing subclasses to implement it as needed.
Inheritance and subtyping

• Inheritance is for code reuse
  - Write code once and only once
  - Superclass features implicitly available in subclass

• Subtyping is for polymorphism
  - Accessing objects the same way, but getting different behavior
  - Subtype is substitutable for supertype
Challenge: Is inheritance necessary?

- Can we get the same amount of code reuse without inheritance?
Reuse via *composition* and *delegation*

```
public class CheckingAccountImpl
   implements CheckingAccount {
   BasicAccountImpl basicAcct = new(...);
   public float getBalance() {
      return basicAcct.getBalance();
   }
   // ...
```

*CheckingAccountImpl* is composed of a *BasicAccountImpl*
Java details: extended re-use with super

```java
public abstract class AbstractAccount implements Account {
    protected float balance = 0.0;
    public boolean withdraw(float amount) {
        // withdraws money from account (code not shown)
    }
}

public class ExpensiveCheckingAccountImpl extends AbstractAccount implements CheckingAccount {
    public boolean withdraw(float amount) {
        balance -= HUGE_ATM_FEE;
        boolean success = super.withdraw(amount);
        if (!success) {
            balance += HUGE_ATM_FEE;
        }
        return success;
    }
}
```

Overrides `withdraw` but also uses the superclass `withdraw` method.
public class CheckingAccountImpl
    extends AbstractAccount implements CheckingAccount {

    private float fee;

    public CheckingAccountImpl(float initialBalance, float fee) {
        super(initialBalance);
        this.fee = fee;
    }

    public CheckingAccountImpl(float initialBalance) {
        this(initialBalance, 5.00);
    }

    /* other methods... */

    Invokes another constructor in this same class
    Invokes a constructor of the superclass. Must be the first statement of the constructor.
Java details: final

- A final class: prevents extending the class
  - e.g., public final class CheckingAccountImpl { ...

- A final method: prevents overriding the method

- A final field: prevents assignment to the field
  - (except to initialize it)

- Why might you want to use final in each of the above cases?
Recall: type-casting in Java

• Sometimes you want a different type than you have
  - e.g.,
  ```java
  float pi = 3.14;
  int indianaPi = (int) pi;
  ```

• Useful if you know you have a more specific subtype:
  - e.g.,
  ```java
  Account acct = ...;
  CheckingAccount checkingAcct = (CheckingAccount) acct;
  float fee = checkingAcct.getFee();
  ```
  - Will get a ClassCastException if types are incompatible

• Advice: avoid downcasting types
Recall: `instanceof`

- Operator that tests whether an object is of a given class

  ```java
  public void doSomething(Account acct) {
    float adj = 0.0;
    if (acct instanceof CheckingAccount) {
      checkingAcct = (CheckingAccount) acct;
      adj = checkingAcct.getFee();
    } else if (acct instanceof SavingsAccount) {
      savingsAcct = (SavingsAccount) acct;
      adj = savingsAcct.getInterest();
    }
    ...
  }
  ```

- Advice: avoid `instanceof` if possible
Avoiding `instanceof` with the Template Method pattern

```java
public interface Account {
    ...
    public float getMonthlyAdjustment();
}

public class CheckingAccount implements Account {
    ...
    public float getMonthlyAdjustment() {
        return getFee();
    }
}

public class SavingsAccount implements Account {
    ...
    public float getMonthlyAdjustment() {
        return getInterest();
    }
}
```
Avoiding `instanceof` with the Template Method pattern

```java
float adj = 0.0;
if (acct instanceof CheckingAccount) {
    checkingAcct = (CheckingAccount) acct;
    adj = checkingAcct.getFee();
} else if (acct instanceof SavingsAccount) {
    savingsAcct = (SavingsAccount) acct;
    adj = savingsAcct.getInterest();
}
```

Instead:

```java
float adj = acct.getMonthlyAdjustment();
```
Today:

• Inheritance and polymorphism (continued)
  ▪ For maximal code re-use
  ▪ Diagrams to show the relationships between classes
  ▪ Inheritance and its alternatives
  ▪ Java details related to inheritance

• Type checking and its limitations
  ▪ The subtype relation
  ▪ Behavioral contracts
    • The java.lang.Object
Typechecking

• The key idea: Analyze a program to determine whether each operation is applicable to the types it is invoked on

• Benefits:
  ▪ Finds errors early
    • e.g., int h = “hi” / 2;
  ▪ Helps document program code
    • e.g., baz(frob) { /* what am I supposed to do with a frob? */ } void baz(Car frob) { /* oh, look, I can drive it! */ }
Value flow and subtyping

- **Value flow**: assignments, passing parameters
  - e.g., `Foo f = expression;`
  - Determine the type $T_{\text{source}}$ of the source expression
  - Determine the type $T_{\text{dest}}$ of the destination variable $f$
  - Check that $T_{\text{source}}$ is a subtype of $T_{\text{dest}}$

- **Aside**: The subtype relation $A <: B$
  - Base cases:
    - $A <: B$ if $A$ extends $B$ or $A$ implements $B$
    - $A <: A$ (reflexivity)
  - Inductive case:
    - If $A <: B$ and $B <: C$ then $A <: C$ (transitivity)
Typechecking expressions in Java

- **Base cases:**
  - variables and fields
    - the type is explicitly declared
  - Expressions using `new ... ()`
    - the type is the class being created
  - Type-casting
    - the type is the type forced by the cast

- **For method calls, e.g., e1.m(e2)**
  1. Determine the type $T1$ of the receiver expression $e1$
  2. Determine the type $T2$ of the argument expression $e2$
  3. Find the method declaration $m$ in type $T1$ (or supertypes), using dispatch rules
  4. The type is the return type of the method declaration identified in step 3
Subtyping rules

• If a concrete class B extends type A
  ▪ B inherits all concrete methods declared in A
  ▪ B can override non-final inherited methods
  ▪ B must override abstract or undefined interface methods

• If B overrides a method declared in type A
  ▪ The argument types must be the same as in A
  ▪ The result type must be subtype of result type from A

• Behavioral subtyping
  ▪ If B overrides a method declared in A, it should conform to the specification from A
  ▪ If Cowboy.draw() overrides Circle.draw() somebody gets hurt!
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()`
Overriding `java.lang.Object`'s `.equals`

- **The default `.equals`:**

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

- **An aside: Do you like:**

  ```java
  public class CheckingAccountImpl {
      @Override
      public boolean equals(Object obj) {
          return false;
      }
  }
  ```
Recall 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

• .equals() always terminates and is side-effect free
The `.hashCode()` contract

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

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

- Returns a *deep copy* of an object
- Generally (but not required!):
  - `x.clone() != x`
  - `x.clone().equals(x)`
Conforming to behavioral contracts

- Complete to support object equality checks:

```java
public class Person {
    private String firstName;
    private String lastName;
    public Person(String name) {
        this.firstName = name.split(" ")[0];
        this.lastName = name.split(" ")[1];
    }
}
```
Next week

- Exceptional control flow
- Type polymorphism (and Java Generics)
- Introduction to specification and testing